Assembly

Programlama Dili

Temmuz 2003

Hazırlayan : Fehmi Noyan İSİ
fni18444@gantep.edu.tr – fnoyanisi@yahoo.com
http://www2.gantep.edu.tr/~fni18444

Bu dokümanda Intel firmasının 80x86 serisi olarak nitelendirilen 8086, 80186, 80286, 80386, 80486 ve Pentium işlemcileri için 16-bit Assembly programlama dili genel yapısı ile anlatılacaktır . Bu sebepten, dokümanda sadece 8086, 80186 ve 80286 (Intel firmasının 16-bit işlemcileri) işlemciler için assembly dili anlatılacaktır. Örnekler MS-DOS işletim sistemi için yazılmıştır. Windows işletim sistemi için assembly programlama 32-bit olup bu dokümanda ele alınmayacaktır. Dokümanda bulunan "Linux Altında Assembly Dili" bölümünde, Linux işletim sistemi altında assembly dilinin kullanımı anlatılmaktadır.

Assembly programlama dili ile ilgili elinize geçecek birçok kaynakta bazı temel terim ve ifadeler sürekli orijinal İngilizce halleriyle kullanıldıkları için ben de birçok terimin ve ifadenin Türkçe karşılığını kullanmadım. Bu, bazı durumlarda anlatımı biraz vasatlaştırmışsa da kavramların çok bilinen adları ile öğrenmenin daha faydalı olacağını düşünüyorum.

Fehmi Noyan İSİ

İçindekiler

Giriş

Assembly programlama dili düşük seviyeli bir dil olup C, C++, Pascal, Basic gibi yüksek seviyeli programlama dillerine göre anlaşılması biraz daha zordur. Assembly dili ile program yazarken kullanılan bilgisayarın donanım özellikleri programcı için önemlidir. Yazılan kodlar çoğunlukla donanıma bağlı yazılır ki bu da programın taşınabilirliğini azaltan bir faktördür.

Assembly dili ile program yazarken programcı doğrudan bilgisayarın işlemcisi ve hafızası ile uğraşır. Yani hafizadaki (RAM'deki) ve işlemci gözlerindeki değerleri doğrudan değiştirme olanağı vardır.

Yüksek seviyeli dillerdeki derleyicilerden farklı olarak, assembly kaynak dosyalarını çalışabilir dosya haline getirebilmek için "assembler" ve "linker" adı verilen programlar kullanılır. Aslında derleyiciler de bir tür assembler programıdır denebilir. Fakat derleyiciler, ekstra bir parametre kullanılmadığı taktirde, kaynak dosyasını önce gerekli *Object* dosyasına çeviriler daha sonra, bir hata ile karşılaşılmaz ise, elde edilen object dosyası linker yardımı ile çalışabilir dosya haline getirilir.

Bilgisayarımızda çalıştırılan tüm programlar önce bilgisayarımızın RAM'ine yüklenir. Daha sonra RAM üzerinde çalıştırma işlemi gerçekleştirilir. RAM'e yüklenen bilgi programımızın makine dili karşılığından başka bir şey değildir. Makine dilinin kullanıcı tarafından anlaşılabilir şekline ise assembly dili demek pek yanlış olmaz.

Aslında assembly programlarının en önemli özellikleri boyutlarının yüksek seviyeli bir dil ile yazılan programlara nazaran çok küçük olması ve buna bağlı olarak çok daha hızlı çalışmalarıdır.

Programların hızlı çalışmaların kodlarının sadeliğinden kaynaklanmaktadır. Fakat günümüzde kullanılan yüksek hızlı işlemciler ve büyük kapasitelere sahip sabit diskler assembly programlarının bu özelliklerini önemsiz kılmaktadır. Aşağıdaki örnekte ekrana "A" harfini basan bir program önce assembly dili ile daha sonra C ve Pascal dilleri ile yazılmıştır. Programların yaptıkları işlerin aynı olmasına karşın boyutları arasındaki büyük farka dikkat edin.

Assembly Programı	**C Programı**	**Pascal Programı**

MOV AH,02
MOV DL,41
INT 21
INT 20
	#include <stdio.h>
	
	main()
	{
	printf("A");
	}
		begin
		write('A')
		end.

Assembler Bilgileri MS-DOS DEBUG	**Derleyici Bilgileri** MS-DOS için Turbo C 2.01	**Derleyici Bilgileri** MS-DOS için FreePascal 0.9
Boyut : 8 bayt	**Boyut : 8330 bayt**	**Boyut : 95644 bayt**

Gördüğünüz gibi C ile yazılan programın boyu assembly ile yazılanınkinin boyunun 1000 katından daha büyük! Pascal ile yazılan programın boyu ile assembly ile yazılanınkinin boyunu karşılaştırmaya bile gerek yok sanırım. Bu fark eski bir bilgisayar için önemli olabilir fakat günümüz standartlarındaki bir bilgisayar için pek önemli değildir. Bu sebepten assembly programlama dili günümüzde daha çok sistem programcıları tarafından ve inline olarak diğer

programlama dilleri içerisinde kullanılmaktadır. “inline assembly” ile kastedilmek istenen, assembly kodlarının olduğu gibi yüksek seviyeli bir dil içerisinde kullanılmasıdır. Bu, bize sabit diskin herhangi bir bölümüne (mesela MBR), BIOS gibi sistem kaynaklarına veya belirli bellek bölgelerine kolayca erişme olanağı sağlar.

Bölüm 1 : Temeller

1.1 Sayı Sistemleri

Günlük hesaplamalarımızda kullandığımız sistem onluk sayı sistemidir ve bu sistem 0,1,2,3,4,5,6,7,8 ve 9 rakamlarından oluşur. Diğer sayılar ise bu rakamlar kullanılarak elde edilir. Kullandığımız bilgisayar için (aslında tüm elektronik cihazlar için dersek daha iyi olur) durum böyle değildir. Bilgisayar binary sayı sistemi dediğimiz ikilik sayı sistemini kullanır ki bu sistemde sadece 0 ve 1 vardır. Bilgisayar için 0’ın anlamı “yanlış” (FALSE) ve 1’in anlamı
(TRUE) “doğru”dur. Buna karşın assembly programları yazılırken kullanılan sayı tabanı hexadecimal olarak bilinen on altılık sayı tabanıdır. Bu sistemde kullanılan ilk on rakam onluk sistemdeki ile aynı olup 0,1,...,9 rakamlarından oluşur. 10, 11, 12, 13, 14 ve 15 için sırasıyla A, B, C, D, E ve F harfleri kullanılır. On altılık sayılar gösterilirken sonlarına “h” veya “H”harfi konur. Assembly dili ile onaltılık sayı sisteminin kullanılmasının sebebi, bellek adresi gibi uzun rakamların ikilik sistem ile gösterilmesinin zorluğudur. Sayı tabanı büyüdükçe herhangi bir sayıyı göstermek için gereken basamağın sayısının azalacağı açıktır. Mesela “1BA5:010F” gibi bir bellek bölgesinin adresini ikilik sistem ile göstermek isteseydik “0001101110100101:0000000100001111” şeklinde olacaktı ki bu hem akılda tutması hem de yazması zor bir sayı.

1.2 Veri Tipleri

1.2.1 Bit

Bilgisayarın ikilik sayı sistemini kullandığından bahsettik. Bu sistemdeki her bir basamağa *Binary Digit* anlamına gelen “bit” denir. Yani bir bit içerisinde 0 veya 1 olmak üzere iki bilgiden biri bulunabilir. Bilgisayar için en küçük bilgi birimi bit’tir.

1.2.2 Bayt

Sekiz adet bit’in oluşturduğu topluluğa “bayt” denir. Bir bayt içerisinde 0-255 arasında olmak üzere 256 değişik değer tutulabilir.

(ikilik) 00000000 = 0 (onluk)
(ikilik) 11111111 = 255 (onluk)

Görüldüğü gibi bir bayt’ın alabileceği maksimum değer 255 ve minimum değer 0’dır.

1.2.3 Word

İki bayt’lık (16-bit’lik) bilgiye “Word” denir. Bir word’un alabileceği maksimum değer 65535 ve minimum değer 0’dır. Bu da bir word içerisinde 65536 farklı değer saklanabileceği anlamına gelir.

1.2.4 Double Word (Long)

İki ayrı word’un birleştirilmesi ile bir “Double Word” elde edilir. Bir double word 32-bit uzunluğundadır.

8086, 80186 ve 80286 işlemcilerde aynı anda işlenebilecek bilgi sayısının 16 bit uzunluğunda olmasından dolayı bu işlemcilere 16-bit işlemci adı verilir. Intel firması 80386 ve sonrası işlemcilerinde 32 bitlik bilgi işleme sistemi kullanmıştır ve bu işlemcilere de 32-bit işlemci adı verilir.

1.3 Bitler Üzerinde Mantıksal İşlemler

İşlemciler birçok işlemi mantıksal karşılaştırmalar yardımı ile yaparlar. Assembly programlama dili içerisinde AND, OR, XOR ve NOT olmak üzere dört adet mantıksal komut kullanılır. Aşağıda bu komutların doğruluk tabloları verilmiştir.

AND	1	0
1	1	0
0	0	0

OR	1	0
1	1	1
0	1	0

XOR	1	0
1	0	1
0	1	0

NOT	1	0
	0	1

Bilgisayarımızda kullanılan baytlar da 1 ve 0'lardan oluştuğu için CPU her mantıksal işlemde bitler üzerinde ayrı ayrı işlem yapar. Aşağıda birkaç örnek verdim.

```
    1001 0110 1001 1111
    1011 1101 0001 1110
AND _____________________
    1001 0100 0001 1110
```

```
    1011 0010 1110 1010
    1011 0000 0000 0000
OR _____________________
    1011 0010 1110 1010
```

```
    1010 1010 1000 1111
    1110 1110 1100 0001
XOR_____________________
    0100 0100 0100 1110
```

```
    0010 0000 0000 1111
NOT_____________________
    1101 1111 1111 0000
```

1.4 İşretli ve İşaretsiz Sayılar

Daha önce veri tipleri anlatılırken bayt için değer aralığı 0-255 word için 0-65535 olarak tanımlandı. Peki işlemlerimizde negatif sayıları kullanmak istersek ne yapmalıyız? Bu durumda bayt, word veya long için ayrılan bölgenin yarısı negatif diğer yarısı da pozitif sayılar için tahsis edilir. Yani bir baytın alabileceği değerler –128....-1 ve 0....127 arasında olur. Aynı şekilde word için bu değer –32.768.....+32.767 arasında olur. Bir bayt, word yada long için en soldaki bite işaret biti denir. İşaretli sayımızın işaret bitinin değeri 0 ise sayı pozitif, 1 ise negatif olarak değerlendirilir. Yani

0110 0101 1110 1100 → pozitif bir sayı (word)
1000 1100 1010 0000 → negatif sayı (bayt)

Herhangi bir sayının negatifini bulmak için ikiye tümleyeni bulunur. Bu iş için

1) Sayı NOT işleminden geçirilir
2) Elde edilen sonuca 1 eklenir

Şimdi, –13'ü bulmaya çalışalım.

+13 = 0000 1101 (bayt)

1) Sayı NOT işleminden geçirilir

0000 1101
NOT_____
1111 0010

2) Elde edilen sonuca 1 eklenir

1111 0010
1
+----------
1111 0011 → -13

Şimdi elde ettiğimiz sonucu bir test edelim.

13+(-13) = 0000 1101 + 1111 0011

Yukarıdaki işlemin sonucu sıfır olamlı.

0000 1101
1111 0011
+--------------
1 0000 0000

Eldeki 1 göz ardı edilirse sonuç sıfır bulunur ki bu bizim işlemimizin doğru olduğunu gösteriyor.

1.5 Shift (Kaydırma) ve Rotate (Döndürme) İşlemleri

Bit dizgileri üzerinde yapılan bir diğer mantıksal işlemler de kaydırma (shift) ve döndürme (rotate) işlemleridir. Bu iki işlem kendi içlerinde sağa kaydırma (right shift), sola kaydırma (left shift) ve sağa döndürme (right rotate), sola döndürme (left rotate) olarak alt kategorilere ayrılabilir.

1	0	1	1	1	0	0	1

Yukarıdaki gibi bir bayta sağa kaydırma (right shift) işlemi uygulanırsa 7. bit 6.nın yerine, 6. bit 5.nin yerine, 5. bit 4.nün yerine .... geçer. Boş kalan 7. bit pozisyonuna 0 yazılır ve 0. bit içersindeki 1 değeri bit dışarısına atılır.

Sola kaydırma (left shift) işlemi de aynı şekilde gerçekleştirilmektedir. Bu sefer boş kalan 0. bit pozisyonuna 0 yazılır ve 7. bit işlem dışı kalır.

0 1 0 1 1 1 0 0 → Sağa kaydırma sonrası

0 1 1 1 0 0 1 0 → Sola kaydırma sonrası

Döndürme (rotate) işleminde de yine kaydırma işleminde olduğu gibi bitler bir sağa veya sola kaydırılır fakat burada boş kalan 7. veya 0. bit yerine sıfır değil de 7. bit için 0. bitin ve 0. bit için de 7. bitin değeri yerlerine yazılır. Yani yukarıdaki baytımıza sırasıyla sağa ve sola döndürme işlemleri uygulanırsa aşağıdaki gibi sonuçlar elde edilir.

1 1 0 1 1 1 0 0 → Sağa döndürme sonrası

0 1 1 1 0 0 1 1 → Sola döndürme sonrası

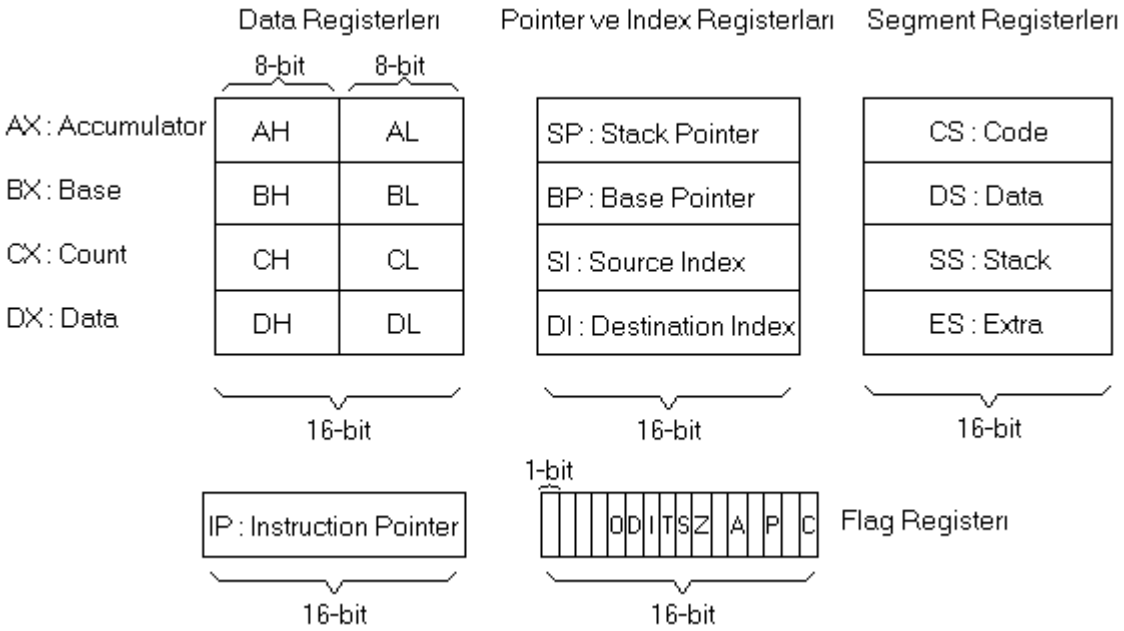
Bölüm 2 : Mikro İşlemci (CPU) ve Bellek

2.1 Mikro İşlemci ve Yapısı

Bugün kullandığımız bilgisayarlarda bilgileri yorumlayan ve işleyen kısım bilgisayarın merkezi işlem ünitesidir. Merkezi işlem ünitesi (Central Processing Unit – CPU) bellek ve çeşitli giriş/çıkış üniteleri ile bus adı verilen veri yollarını kullanarak haberleşmektedir.

İşlemciler, dahili ve dışsal hatlarının bilgi transferi ve işleyebilecekleri maksimum bilgi kapasitesine göre 8 bitlik, 16 bitlik ve 32 bitlik işlemciler olarak adlandırılır. Bu dokümanda anlatılan assembly dili 16 bitlik bir işlemci olan Intel 8086 (dışsal ve dahili veri transferi word'ler halinde yapılmaktadır) serisi içindir. Dilin 8086 serisi için olması, yazdığınız programların 8088, 80186, 80286, 80386, 80486, Pentium ve Celeron serileri ve AMD işlemciler üzerinde çalışmayacağı anlamına gelmemektedir.

İşlemci, *register* adı verilen bölmelerden oluşur. Bu register'ları Data register'ları, Pointer ve Index register'ları, Segment register'ları, Instruction Pointer (komut göstergeci) ve Program Status Word (Flag register'ı) olarak gruplandırabiliriz. Aşağıda 8086 işlemcinin register yapısı basitçe gösterilmiştir.



2.1.1 Data Register'ları

Şekilde de görüldüğü gibi 8086 işlemci 14 adet 16-bit kapasiteli register'a sahiptir. Bu register'lardan Data register'ları (AX, BX, CX ve DX) kendi içlerinde 8-bit kapasiteli iki register'a bölünmektedir. Bu register'lar AX için AH ve AL, BX için BH ve BL, CX için CH ve CL ve DX için DH ve DL olarak adlandırılır. AH, 16-bitlik AX register'ının 8-bitlik yüksek(High) seviyeli bölümü ve AL ise 16-bitlik AX register'ının 8-bitlik alçak (Low) seviyeli bölümü olarak adlandırılır. Programlarımızı yazarken 8-bitlik verilerin işlenmesinde bu 8-bitlik register'lardan faydalanırız. Daha sonraki bölümlerde de göreceğimiz gibi data register'ları bir çok işlemde (döngülerde, interrupt kullanılmasında, matematiksel işlemlerde...) CPU tarafından rezerv edilmişlerdir. Bu durumlarda bizim register içerisindeki değeri değiştirme şansımız yoktur.

Programlarımızda BH veya BL register'ının değerinin değişmesi doğrudan BX register'ının değerini değiştirecektir. Aynı durum diğer data register'ları için de geçerlidir. Bir örnek vermek gerekirse: BX register'ımızın değeri A43F olsun. Bu durumda BH=A4 ve BL=3F olur. Simdi BL'nin değerini 05 olarak değiştirelim. Son durumda BX=A405, BH=A4 ve BL=05 olacaktır. (Yukarıdaki şekil incelenirse verilen örnek daha kolay anlaşılacaktır)

2.1.2 Pointer ve Index Register'ları

Pointer ve index register'ları bellek içerisindeki herhangi bir noktaya erişmek için kullanılır. Bu iş için erişilmek istenen noktanın offset ve segment adresleri gerekli register'lara atanarak işlem gerçekleştirilir. (Bölüm 2.2, Bellek ve Yapısı kısmında bu konu daha ayrıntılı ele alınacaktır.)

2.1.3 Segment Register'ları

Segment register'ları (CS, DS, SS ve ES) , programımız bilgisayarın belleğine yüklendiği zaman bellek içerisinde oluşturulan bölümlerin (Segment) başlangıç adreslerini tutarlar. Yani bu register'lara bir çeşit yer göstergeci denebilir. CS, programımızın çalıştırılabilir kodlarını barındıran (Code Segment) bellek bölgesinin başlangıç adresini tutar. Yani CS ile gösterilen yerde makine dili kodlarımız vardır. DS, programımız içerisindeki değişkenlerin saklandığı bölümdür. SS, bellekte programımız için ayrılan stack bölümünün başlangıç adresini tutar. Stack, ileride göreceğimiz PUSH ve POP komutları ile belleğe değer atılması ve alınması için, programların ve fonksiyonların sonlandırıldıktan sonra nereye gideceklerini belirten adres bilgisini tutmak için ve program ve fonksiyonlara gönderilen parametreleri saklamak için kullanılır. UNIX tabanlı sistemlerde kesme (interrupt) kullanımı için gerekli parametrelerin belirtilmesi için de stack kullanılır. ES (Extra segment) daha çok dizgi işlemleri için kullanılır.

2.1.4 Instruction Pointer

Instruction Pointer register'ı işlemci tarafından işlenecek bir sonraki komutun bellekteki adresini saklar. Bu register üzerinde programcı tarafından herhangi bir işlem yapılamaz. Her komut işletildikten sonra CPU otomatik kullanılan komuta göre gerekli değeri bu register'a atar.

2.1.5 Flag Register

Flag Register diğer register'lardan daha farklı bir durumdadır. Diğer resigterların 16-bit halinde bir bütün olarak ele alınmalarından farklı olarak flag register'ın her biti CPU için ayrı bir değere sahiptir (8086 işlemci tarafından sadece 9 tanesi kullanılmaktadır). Bu bitlerin özel olarak isimlendirilmeleri aşağıda verilmiştir.

O : Overflow flag
D : Direction flag
I : Interrupt flag
T : Trace flag
S : Sign flag
Z : Zero flag
A : Auxilary Carry flag
P : Parity flag
C : Carry flag

Bilgisayar ikilik sayı sistemini kullandığına göre bu register'lar içerisindeki değer herhangi bir anda ya 1 yada 0'dır. Bir bitin 1 olma durumuna "Set", 0 olma durumuna ise "Reset" denir. İşlemci birçok komutu icra ederken bu bitlerin durumlarından faydalanır.

2.1.5.1 Carry Biti

İşlemci tarafından yapılan herhangi bir işlem sırasında alıcı alana yerleştirilen sayının alıcı alana sığmamasından doğan olaya "carry"(taşma) denir. CPU bir işlem sonucunda taşma ile karşılaşırsa carry flagın değeri 1 yapılar.

Aşağıda 16-bitlik iki sayı toplanmıştır ve sonuçta bir taşma olmuştur. (Taşan bit kırmızı ile gösterilmiştir)

```
  1111 1001 0110 1010      → F96A
  0010 1010 1111 1001      → 2AF9
 +---------------------------
 1 0010 0100 0110 0011     → 2463        CF=1
```

F96A ile 2AF9'un toplanması sonucu 2463 sayısı elde edilmiştir ki bu iki sayıdan da küçüktür. Elimizdeki sayılar pozitif olduğuna göre toplama işlemi sonucunda elimizdeki sayılardan daha küçük bir sayı elde etmemiz imkansızdır. CPU bu durumda carry flagın değerini 1 yaparak 17-bitlik bir sayı elde eder.

```
C:\WINDOWS\Desktop>debug
-a100
1E40:0100 MOV AX,F96A
1E40:0103 MOV BX,2AF9
1E40:0106 ADD AX,BX
1E40:0108
-t

AX=F96A  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000 DI=0000
DS=1E40  ES=1E40  SS=1E40  CS=1E40  IP=0103   NV UP EI PL NZ NA PO NC
1E40:0103 BBF92A        MOV     BX,2AF9
-t

AX=F96A  BX=2AF9  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000 DI=0000
DS=1E40  ES=1E40  SS=1E40  CS=1E40  IP=0106   NV UP EI PL NZ NA PO NC
1E40:0106 01D8          ADD     AX,BX
-t

AX=2463  BX=2AF9  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000 DI=0000
DS=1E40  ES=1E40  SS=1E40  CS=1E40  IP=0108   NV UP EI PL NZ AC PE CY
1E40:0108 A394D3        MOV     [D394],AX              DS:D394=A2A2
```

Yukarıdaki ekran görüntüsü verdiğim örneğin debug programı altında denenmesi sonucu elde edilmiştir.

İlk önce "MOV AX,F96A" ile AX register'ının değeri F96A yapılmıştır. Bunu ilk t komutundan sonra AX=F96A ile görebiliriz (1. t komutundan sonraki bölümde AX'in değerini inceleyin). Daha sonra "MOV BX,2AF9" ile BX'in değeri 2AF9 yapılmıştır (2. t komutundan sonraki bölümde BX'in değerini inceleyin). En son "ADD AX,BX" ile bu iki değer toplanıp AX'e atanmıştır. Son "t" komutundan sonra AX=2463 olmuştur ve en sondaki "NC" ifadesi "CY"ye dönüşmüştür ki bu da işlemin sonucunda bir taşma olduğunu gösterir.

2.1.5.2 Parity Biti

Bir işlem sonucunda word'un düşük seviyeli baytı iki ile tam bölünüyorsa bu bite 1 aksi taktirde 0 atanır. İşlemin yüksek seviyeli baytındaki sayının iki ile bölünüp bölünmemesi bu flag için önemli değildir.

2.1.5.3 Auxilary Carry Biti

CPU tarafından gerçekleştirilen işlem sonucunda alıcı alanın ilk dört biti üzerinde bir taşma gerçekleşiyorsa bu flagın değeri 1 yapılır.
Yukarıda carry flag için verilen örnekte ilk sayının ilk dört biti olan 1010 (hex A) ve ikinci sayının ilk dört biti olan 1001 (hex 9) toplamı sonucu 0011 sayısı elde edilmiş ve bir sonraki bite bir elde sayı aktarılmıştır. ADD işlemi öncesinde NA (0) olan bitin değeri, işlem sonrasında AC (1) olmuştur.

2.1.5.4 Zero Biti

Yapılan herhangi bir işlem sonucu sıfır ise bu flag set edilir.

```
-a100
1E40:0100 mov ax,12
1E40:0103 xor ax,ax
1E40:0105
-t

AX=0012  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000 DI=0000
DS=1E40  ES=1E40  SS=1E40  CS=1E40  IP=0103   NV UP EI PL NZ NA PO NC
1E40:0103 31C0          XOR     AX,AX
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000 DI=0000
DS=1E40  ES=1E40  SS=1E40  CS=1E40  IP=0105   NV UP EI PL ZR NA PE NC
1E40:0105 F0            LOCK
1E40:0106 01D8          ADD     AX,BX
```

Yukarıda önce AX register'ının değeri 12 yapılıyor. Daha sonra bu register kendisi ile XOR işlemine tabi tutulmuştur. Bir sayının kendisi ile XOR işlemine tabi tutulması sonucu 0 elde edilir. Bu durum zero flagın değerini set etmiştir (NZ → ZR).

2.1.5.5 Sign Biti

İşaretli sayılarda bayt (8 bit) için sayının 7. ve word (16 bit) için sayının 15. biti işaret biti olarak adlandırılır. Yapılan bir işlem sonucunda alıcı alan içersindeki işaret biti sign flag içerisine kopyalanır. Yani sign bitinin değeri 0 ise elde edilen sonuç pozitif, 1 ise elde edilen sonuç negatif kabul edilir.

2.1.5.6 Trace Biti

CPU'nun sadece bir komut çalıştırıp beklemesi için kullanılır. DEBUG'ın kullandığı "Trace" işlemi bu flagın set edilmesi ile gerçekleştirilir.

2.1.5.7 Interrupt Biti

CPU'nun çeşitli aygıtlardan gelen kesme isteklerini dikkate alıp almayacağını bildirir. 0 olması durumunda istekler dikkate alınmayacaktır.

2.1.5.8 Direction Biti

Bu flagın değeri genellikle dizgi işlemleri üzerindeki işlemin yönünü belirtmek için kullanılır.

2.1.5.9 Overflow Biti

İşaretli sayılar üzerindeki taşmayı kontrol etmek için kullanılır. Hatırlarsanız işaretsiz sayılar için carry flagı kullanılmıştır. Fakat durum işaretli sayılar için biraz daha farklıdır. İşaretli sayılarda meydana gelecek bir taşma bayt veya wordun işaret bitini etkileyeceği için pozitif (işaret biti 0) olan bir işaretli sayının taşma sonucu negatif (işaret biti 1) gibi algılanması mümkündür. Bu durumda overflow flagı set edilir ve işlem sonucunun yanlış algılanması engellenir.

2.2 Bellek ve Yapısı

Bilgisayarımızda bir program çalıştırdığımız zaman program önce RAM'e yüklenir ve daha sonra çalıştırılır. Yükleme işlemi programın kod, data gibi bölümlerinin çeşitli belek alanlarına ayrı ayrı yüklenmesi ile yapılır. Bu alanlara segment adı verilir. 8086 işlemcide bu segmentlerden her birinin boyu 64 Kb (65535 bayt) boyundadır. Bu değer 32-bit işlemcilerde 4 Gb kadardır. Biz 8086 işlemci ve 16-bit uzunluğundaki bellek bölgeleri ile çalıştığımıza göre adresleyebileceğimiz maksimum alan 64 Kb boyundadır. 8086 işlemci programcıya üzerinde çalışabileceği dört adet segment sağlar. Bunlar : CODE, DATA, EXTRA ve STACK segmentleridir. Bu segmentlerin başlangıç adresleri CODE SEGMENT için CS'de, DATA SEGMENT için DS'de, EXTRA SEGMENT için ES'de ve STACK SEGMENT için SS'de saklanır.

Yukarıda anlattığımız 64 Kb'lık segmentler de kendi içlerinde 1 baytlık bölümlere ayrılmışlardır. Segmentlerin içerisindeki bu bir baytlık bölümlere OFFSET adı verilir. Yani bir segment içerisinde 65535 tane offset vardır. Programcı tarafından girilen her komut tuttuğu bayt sayısı kadar offset adresini ileri alır.

Bellek içerisindeki herhangi bir noktaya erişmek için SEGMET:OFFSET ikilisi kullanılır. Code segment içerisindeki bir noktaya erişmek için segmet adresi olarak CS'deki değer ve offset adresi olarak IP içerisindeki değer kullanılır. Aynı şekilde stack segment içerisindeki bir değere ulaşmak için segment adresi olarak SS içerisindeki değer ve offset adresi olarak SP veya BP'den biri kullanılır.

Bellek ve verilerin nasıl depolandığı konusunda aşağıdaki örnek size yardımcı olacaktır.

```
-A100
1E40:0100 MOV AH,01
1E40:0102 INT 21
1E40:0104 CMP AL,45
1E40:0106 JNE 0100
1E40:0108 INT 20
1E40:010A
-G
GGHJRE
Program normal olarak sonlandırıldı
-U 100
1E40:0100 B401          MOV     AH,01
1E40:0102 CD21          INT     21
1E40:0104 3C45          CMP     AL,45
1E40:0106 75F8          JNZ     0100
1E40:0108 CD20          INT     20
1E40:010A D3E8          SHR     AX,CL
1E40:010C 94            XCHG    SP,AX
1E40:010D 00726D        ADD     [BP+SI+6D],DH
```

```
1E40:0110 E87000        CALL    0183
1E40:0113 A396D3        MOV     [D396],AX
1E40:0116 8A04          MOV     AL,[SI]
1E40:0118 3C20          CMP     AL,20
1E40:011A 740C          JZ      0128
1E40:011C 3400          XOR     AL,00
1E40:011E 2F            DAS
1E40:011F 1E            PUSH    DS
-
```

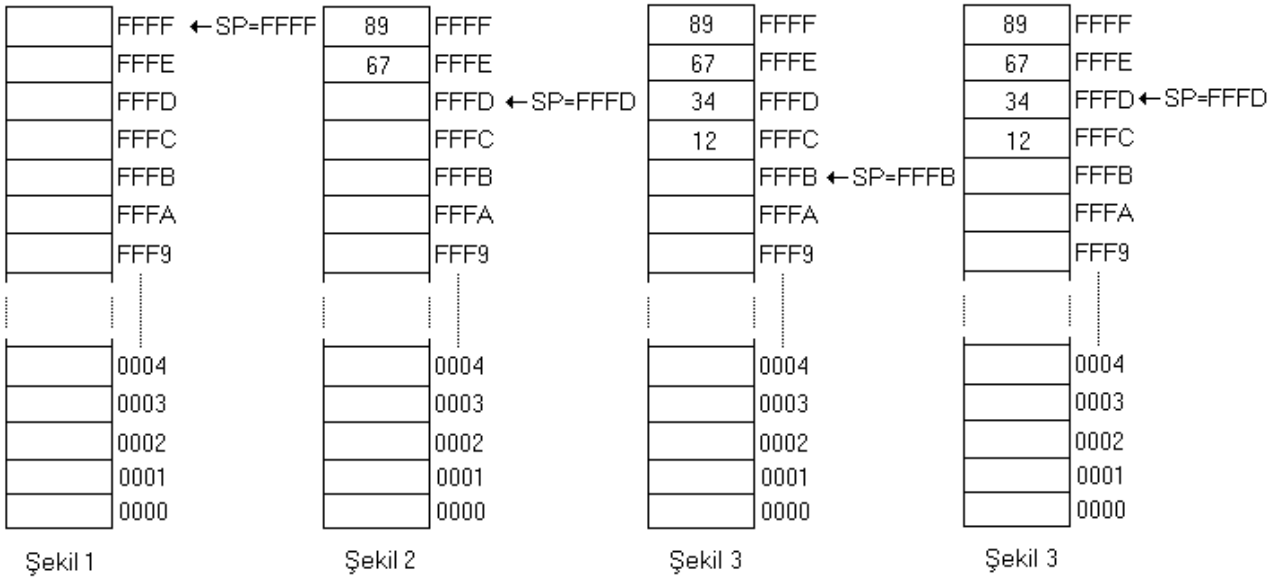
Yukarıdaki kısa program klavyeden girilen her tuşu okur ve girilen 'E' oluncaya kadar okuma işlemine devam eder. (Bu arada programın boyunun sadece 10 bayt olduğuna dikkat edin. C veya Pascal ile aynı program kaç KB tutardı acaba!! ☺). Önce programı yazdık ve daha sonra "G" ile çalıştırdık. En son olarak da "U 100" ile 100. offset adresinden itibaren dağıtma (unassemble) işlemi yaptık. Dikkat ederseniz her satırın başında "1E40:0104" gibi, iki nokta ile ayrılmış iki sayı var. Bu sayılar sırası ile komutun segment ve offset adresleridir. Yani bilgisayarımızın 1E40. segmenti o anki geçerli kod segmentimiz ve bu segment içerisindeki 0104. offset de (aslında 0104. ve 0105. offsetler) içerisinde "CMP AL,45" komutunu barındırmaktadır. Bir sonraki satırda bu sayılar "1E40:0106" halini alıyor ki buda bize offset adresimizin 2 arttığını gösteriyor. Yani "CMP AL,45" komutu bilgisayarımızın belleğinde 2 baytlık yer kaplıyor. Dağıtma işleminde segment ve offset adreslerinin sağındaki sayılar ise sizin girdiğiniz komutların bellekte bulundukları hallerinin on altılık tabandaki karşılıklarıdır. İşte bilgisayar asıl olarak bilgiyi belleğinde sayılar halinde saklar ve assembly dili bu sayıların insan için daha okunur hale gelmiş halidir. Yani CPU "3C45" (0011 1100 0100 0101 = 16-bit) ile karşılaştığı zaman bunun bir karşılaştırma komut olduğunu anlar ve ona göre işlem yapar. Diğer komutlarda aynı şekilde işlem görmektedir. Eldeki bilginin komut mu yoksa bir değişken mi olduğu, bilginin bulunduğu segment aracılığı ile anlaşılır.

2.3 Stack

Yüksek seviyeli programlama dillerinde değerler değişken adı verilen alanlarda saklanır. Assembly dilinde ise programcı değerleri saklamak için CPU register'larını ve stack'ı kullanılır. Stack, bilgilerin geçici olarak depolandığı bir bölümdür. 8086 işlemci programcıya 64 Kb boyundu bir stack segment sağlar. Bu segmentin başlangıç adresi SS içerisinde ve o anki geçerli offset adresi ise SP (Stack Pointer) içerisinde bulunur.

Stack içerisinde, program içerisindeki fonksiyonların geri dönüş değerleri, fonksiyonlara aktarılan argümanlar, komut satırı parametreleri gibi değerler saklanır. Ayrıca programcı bazı değerleri geçici olarak bu alana depolayıp daha sonra kullanabilir.

Stack içersinde yerleştirme işlemi kelimeler (word) halinde olur. Yani, stack üzerinde bir işlemde her seferinde iki offset geriye veya ileriye gidilir. Stack programcıya sıralı erişim sağlar. Stack segmentimizin 0000-FFFF arasında değişen offset adresleri ve SP'nin o anki değerinin FFFF olduğunu varsayarsak, stack üzerine konulacak ilk değer FFFF ve FFFE adreslerine yazılır (İki bayt veri yazılıp okunabilir). Daha sonra SP'nin değeri yazılacak bir sonraki değerin adresini
belirtmek üzere FFFD olarak değiştirilir.



Yukarıda stack pointer'ın ilk değeri FFFF'dir (Şekil1). Stack üzerine ilk bilgi aktarımından sonra bir sonraki bilginin yazılacağı adresi belirtmek için SP=FFFD oluyor (Şekil 2). FFFF ve FFFE adreslerine sırasıyla 89 ve 67 sayıları yazılmış. Bu bize stack üzerine yazılan bilginin 6789h olduğunu gösteriyor. Aynı şekilde, daha sonra 1234h değeri stack üzerine aktarılıyor (Şekil 3) ve SP'nin değeri FFFB olarak değişiyor. Şekil 4'te stack üzerinden bilgi alınıyor. İlk iki işlemin tersine bu sefer SP'nin değeri 2 artıyor ve FFFB+2=FFFD oluyor. Yani o andan sonra yazılacak ilk değer FFFD ve FFFC adreslerine yazılacaktır. Kısaca, stack üzerine bilgi yazıldığı zaman SP'nin değeri azalır ve bilgi okunduğu zaman artar.

Stack LIFO (Last-In First-Out) prensibi ile çalışır. Yani stack üzerine yazılan en son bilgi alınabilecek ilk bilgi ve yazılan ilk bilgi de alınabilecek son bilgidir. Yukarıdaki örnekte ilk önce 6789h sayısı daha sonra 1234h sayısı stack üzerine yazıldı. İlk yazılan değer olan 6789h'a ulaşmak için önce 1234h okunmalı.

Stack üzerindeki işlemler PUSH ve POP komutları ile gerçekleştirilmektedir.

MOV komutu: Bir sonraki bölüm olan "80x86 İşlemcilerde Bellek Adresleme" kısmında bellek ve CPU register'ları arasındaki bilgi transferi MOV komutu kullanılarak anlatılacaktır.

MOV hedef,kaynak

MOV komutunun kullanım şekli yukarıdaki gibidir. Bu komut "kaynak" içerisindeki bilgiyi "hedef"e aktarır. Yani

```
MOV AX, 1234h
MOV BX, AX
```

Yukarıda ilk önce AX register'ına 1234h değeri yazılmıştır. Daha sonra bu değer AX register'ından BX register'ına aktarılmıştır. Yani Program sonunda AX=1234h ve BX=1234h olur.

Komutun kullanımında dikkat edilmesi gereken en önemli nokta "hedef" ve "kaynak" alanlarının eşit boyutlarda olmasıdır (Örneğimizde 16-bit.).

2.4 80x86 İşlemcilerde Bellek Adresleme

8086 işlemci programcıya belleğe ulaşım için 17 çeşit adresleme metodu sağlar. Assembly programlama dilinde bellek adresleme ve belleğe erişim en önemli ve iyi bilinmesi gereken konulardandır.

- Displacement-only
- Base
- Displacement + Base
- Base + Indexed
- Displacement + Base + Indexed

Kullanılan 17 çeşit adresleme metodu yukarıdaki beş ana kategoriden türetilmektedir.

2.4.1 Doğrudan Adresleme (Direct Addressing)

Bu adresleme metodu anlaşılması en kolay olandır. Bu yolla adresi verilen herhangi bir bellek gözüne okuma veya yazma yapılır. Kullandığımız bellek 16-bit'lik adreslerden oluştuğuna göre adres sabitlerimiz de dört hanelik on altılık sayılardan oluşmalıdır.

```
MOV AX, [4126]
MOV CX, CS:[4A4C]
MOV DS:[41C8], BX
```

Yukarıda ilk satırda "MOV AX, [4126]" ile bellekteki 4126h ve 4127h adreslerindeki değerler AX register'ına atanmıştır. 4126h adresi 1 bayt ve AX register'ı 2 bayt olduğu için hem 4126h hem de 4127h adresindeki değerler okunmuştur.

İkinci örnekte Code Segment içerisindeki 4A4Ch ve 4A4Dh adresindeki değerler CX register'ına atanmıştır. Gördüğünüz gibi "CS:[4A4C]" gibi bir ifade kullanarak istediğiniz herhangi bir segment içerisindeki bilgiye erişebiliyorsunuz.

Son olarak da "MOV DS:[41C8], BX" komutu ile BX içerisindeki değer Data Segment içerisindeki 41C8h ve 41C9h adreslerine yazılıyor.

2.4.2 Dolaylı Adresleme (Indirect Addressing)

Bu metot ile herhangi bir register içerisinde barındırılan değerin gösterdiği bellek bölgesindeki alan ile işlem yapılır. Bu metot için kullanılabilecek dört register BX (Base), BP (Base Pointer), SI (Source Index) ve DI (Destionation Index)'dır. Bu yol ile adreslemeye aşağıda birkaç örnek verdim.

Based

```
MOV BX, 1234
MOV AX, [BX]
```

Önce BX'in değeri 1234h olarak değiştiriliyor. Daha sonra bellekte 1234 ile işaret edilen yerdeki değer AX içerisine atanıyor. Yani yandaki komut kümesi

```
MOV AX, [1234]
```

ile aynı işi yapıyor.

Based

```
MOV BP, 45C8
MOV AX, [BP]
```

BP = 45C8h

```
MOV AX, [45C8]
```

Indexed

MOV DI, 76A7	DI = 76A7h
MOV AX, [DI]	MOV AX, [76A7]

Indexed

MOV SI, D78C	SI = D78Ch
MOV AX, [SI]	MOV AX, [D78C]

İlk iki adresleme yönteminde Base register'lar olan BX ve BP kullanıldığı için bu yöntem Based olarak adlandırılır. Aynı şekilde DI ve SI ile yapılan adreslemelerde de Index register'lar kullanıldığı için Indexed olarak adlandırılır.

Bir önceki bölümde olduğu gibi bu yöntemde de ulaşılmak istenen offset için ayrı bir segment belirtilebilir. Yukarıda BX, DI ve SI ön tanımlı olarak DS (Data Segment)'i kullanır. BP için ön tanımlı segment SS (Stack Segmenttir).

2.4.3 Indexed Adresleme

Bu metot ile istenilen yere ulaşmak için hem bir sabit hem de yukarıdaki gibi bir register kullanılır. İstenilen offset adresi kullanılan sabit ile register içerisindeki değerin toplamıdır. Yine yukarıda olduğu gibi BX, DI ve SI ön tanımlı olarak DS'i ve BP ön tanımlı olarak BP'yi segment olarak kullanır. Tabi ki istenildiği taktirde farklı segmentlerdeki verilere ulaşmak mümkündür.

MOV BX, 7A82	BX = 7A82
MOV AL, 78[BX]	7A82 + 78 = 7AFA
	MOV AL, [7AFA]

Gördüğünüz gibi yukarıda "MOV AL, 78[BX]" komutu ile BX içerisinde bulunan 7A82h değeri ile sabitimiz olan 78h değerini toplamı olan 7AFAh offseti içerisindeki değer AL'ye atanıyor. Tabi, BX kullanıldığı için kaynak segment olarak Data Segment kullanılmaktadır.

2.4.4 Based Indexed Adresleme

Bu yöntem dolaylı adresleme yöntemi ile benzerlik göstermektedir. Tek fark burada bir değil ,biri Base diğeri Index register olmak üzere iki farklı register kullanılmasıdır. Hedef bölgeye atanacak değer bu iki register içerisindeki değerin toplamının gösterdiği offset içerisindeki değerdir.

MOV BX, D854	BX = 7A82
MOV SI, 278C	SI = 278C
MOV AL, [BX][SI]	D854 + 278C = FFE0
	MOV AL, [FFE0]

```
MOV BP, 54AC          BP = 54AC
MOV DI, 4444          DI = 4444
MOV AL, [BP][DI]      54AC + 4444 = 98F0
                      MOV AL, [98F0]
```

Yukarıdaki ilk örnekte ön tanımlı segment Data Segment ve ikincide ise Stack Segmenttir.

2.4.5 Based Indexed + Sabit(Disp) Adresleme

Bu yöntem yukarıdaki bütün yöntemleri kapsar. Genel formu aşağıdaki gibidir ve register'lar içerisindeki değer ile sabitin toplamının verdiği offset adresindeki değer ile işlem yapılır.

```
MOV AL, DISP[BX][SI]
MOV BL, DISP [BX+DI]
MOV BH, [BP+SI+DISP]
MOV AH, [BP][DI][DISP]
```

Bölüm 3 : 80x86 Komut Kümesi

Bu bölümde Intel firmasının 80x86 serisi işlemcilerini programlamak için kullanılan assembly komutlarından bir kısmını inceleyeceğiz.

- Transfer komutları
  - MOV
  - XCHG
  - LEA
  - PUSH
  - PUSHF
  - POP
  - POPF
  - LAHF
  - SAHF

- Giriş/Çıkış komutları
  - IN
  - OUT

- Aritmetiksel Komutlar
  - ADD
  - ADC
  - SUB
  - SBB
  - MUL
  - IMUL
  - DIV

IDIV
INC
DEC
CMP

- Mantıksal Komutlar
  - AND
  - OR
  - XOR
  - NOT
  - TEST

- Kaydırma ve Döndürme Komutları
  - SAL/SHL
  - SHR
  - SAR
  - RCL
  - ROL
  - RCR
  - ROR

- Dallanma Komutları
  - JMP
  - JZ/JE
  - JNZ/JNE
  - JB/JC/JNAE
  - JBE/JNA
  - JNB/JNC/JAE
  - JG/JNLE
  - JA/JNBE
  - JL/JNGE
  - JLE/JNG
  - JS ve JNS
  - JO ve JNO
  - JCXZ

- Döngü Komutları
  - LOOP
  - LOOPZ/LOOPE
  - LOOPNZ/LOOPNE

3.1 Transfer Komutları

Bu grup içerisindeki komutlar herhangi bir bilgiyi register-register, bellek-register ve register-bellek bölgeleri arasında transfer etmek için kullanılır.

3.1.1 MOV Komutu

MOV komutuna daha önce bellek adresleme bölümünde kısaca değinilmişti. Bu kısımda komutun kullanımı hakkında daha ayrıntılı bilgi verilecektir. Mov komutunun çeşitli kullanım biçimleri aşağıdaki gibidir.

Genel Form : mov hedef, kaynak

```
mov register, register
mov bellek, register
mov register, bellek
mov bellek, sabit değer
mov register, sabit değer
```

Yukarıda mov komutunun çeşitli kullanım şekilleri gösterilmiştir. Dikkat ederseniz kullanım şekilleri arasında bellek-bellek arasında transfer yok. Yukarıdaki "register" ifadeleri hem genel amaçlı register'ları hem de segment register'larını temsil etmektedir. Yalnız, kesinlikle bir sabit değer doğrudan bir segment register'ına atılamaz. Böyle bir işlem aşağıdaki gibi iki aşamada gerçekleştirilir.

```
mov ax, 1234
mov cs, ax
```

Yukarıdaki örnekte CS içerisine 1234h değeri atanmıştır.

Dikkat edilmesi gereken bir başka önemli nokta da transferi yapılacak değerlerin boylarının aynı olmasıdır. Yani AH içerisine 16-bit'lik bir değer atamsı yapılamaz. Aynı şekilde 8-bit'lik bir register'dan 16-bitlik bir resgister'a da transfer yapılamaz.

Herhangi bir bellek bölgesi dolaylı adresleme ile adreslenip içerisine sabit bir değer atanmak istendiği zaman atanacak değerin uzunluğu belirtilmelidir.

```
mov [bx], 12          → Hata !

mov byte ptr [bx], 12
 yada
mov word ptr [bx], 1234
```

"byte ptr" ve "word ptr" önekleri ile transfer edilecek bilginin boyu hakkında işlemciye bilgi verilmektedir.

NOT: 32-bit işlemcilerde (80386 ve sonrasında) "dword ptr" öneki ile 32-bit bilgi transferi yapılabilmektedir.

MOV komutu flag register'lar üzerinde herhangi bir değişiklik yapmamaktadır.

3.1.2 XCHG Komutu

XCHG (exchange) komutu iki değeri karşılıklı olarak değiştirmek için kullanılır. Tabi yine değişiklik yapılacak değerlerin aynı boyda olması gerekmektedir. Genel formu aşağıdaki gibidir.

```
xchg register, register
xchg bellek, register
```

xchg komutunun kullanımında operandların yerleri önemli değildir. Yani "xchg bellek, register" ile "xchg register, bellek" aynı işi yapmaktadır.

```
mov ax,1234        → AX=1234h
mov bx, 5678       → BX=5678h
xchg ax,bx         → AX=5678h  BX=1234h
```

XCHG komutu flag register'lar üzerinde herhangi bir değişiklik yapmamaktadır.

3.1.3 LEA Komutu

LEA komutunun kullanımı MOV komutu ile benzerlik göstermesine karşın bazı durumlarda programcıya iki yada üç komut ile yapılacak bir işlemi tek komut ile yapma olanağı sağlamaktadır. Genel formu aşağıdaki gibidir.

```
lea register, bellek
```

Komutun kullanımına birkaç örnek verince programcıya sağladığı kolaylıklar daha basit anlaşılacaktır.

```
lea ax, [bx]      → mov ax, [bx]
lea bx, 3[bx]     → BX = BX+3
```

Yukarıdaki kullanımlar "lea" komutu dışında tek bir komut ile icra edilebilecek durumlardır. (BX = BX+3 ifadesinin assembly dilindeki karşılığı daha anlatılmadığı için normal bir gösterim kullanılmıştır) Aşağıdaki örnekler incelenirse komutun sağladığı kolaylık açıkça fark edilecektir.

```
lea ax, 3[bx]          → BX = BX+3 → mov ax, bx
lea ax, 8[bx+di]
```

LEA komutunun genel kullanım amacı herhangi bir register'a bir bellek adresi saklamaktır. Komut kullanımı sırasında flag register'lar üzerinde herhangi bir etki yapmamaktadır.

3.1.4 PUSH Komutu

PUSH komutu herhangi bir bilgiyi bilgisayarın stack adı verilen bölümüne kaydetmek için kullanılır. PUSH komutu ile stack üzerine atılacak bilgi 16-bit uzunluğunda olmalıdır. Komutun genel formu aşağıdaki gibidir.

PUSH değer

Yukarıda "değer" ile gösterilen kısım daha öncede belirtildiği gibi 16-bit uzunluğunda olmalıdır. Bunun yanı sıra "değer" ile gösterilen kısım sabit bir değer alamaz. Yani PUSH ile stack üzerine yazılacak değer ya bir register içerisindeki değer yada bir bellek bölgesindeki değer olmalıdır.

PUSH komutu ile stack üzerine bilgi yazıldığı için SP'nin değeri değişmektedir. İcra edilen her PUSH komutu SP'nin değerini 2 azaltacaktır (PUSH ile 2 bayt bilgi aktarıldığı için). Aşağıda PUSH komutu ile stack üzerine bilgi aktarılmasına bir örnek verilmiştir.

```
mov ax,1234    → AX = 1234h
push ax        → AX > STACK , SP = SP-2
```

Stack üzerine bilgi yazma işlemi gerekli değerlerin geçici bir süre saklanması için ve bazı UNIX sistemlerde kesme kullanımında gerekli parametrelerin aktarılması için işaretçiler ile birlikte kullanılır.

3.1.5 PUSHF Komutu

PUSHF komutu ile PUSH komutuna benzer olarak stack üzerine bilgi aktarılır. Yalnız burada tek fark, PUSHF komutu ile aktarılacak bilginin herhangi bir register yada bellek bölgesinden değil de flag register'dan alınmasıdır. Komutun kullanımı aşağıdaki gibidir.

PUSHF

Görüldüğü gibi komutun kullanımı sırasında hiçbir register yada bellek adresi kullanılmamıştır. Onun yerine stack üzerine atılacak bilgi doğrudan 16-bit uzunluğundaki flag resigter içerisindeki değerdir.

Komut herhangi bir işlem sırasında flag register'ın mevcut değerini korumak için kullanılır. Yine PUSH komutunda olduğu gibi PUSHF komutu da SP'nin değerini 2 azaltacaktır.

3.1.6 POP Komutu

POP komutu ile stack üzerinden bilgi okuması yapılır. Yani PUSH komutu ile stack üzerine yazılan bilgi POP komutu ile geri okunur. Okunan bilgi 16-bit uzunluğunda olmalıdır.

POP komutu ile alınacak bilgi stack üzerine yazılan son bilgidir. PUSH ve POP komutları ile bilgi transferi yapılırken yazılan ve okunan bilgilerin sıralaması önemlidir. Programlar yazılırken stack üzerindeki işlemlerde hesaplama hatası yapılması sık karşılaşılan durumlardandır. Komutun genel kullanımı aşağıdaki gibidir.

POP alıcı_alan

Yukarıda alıcı alan bir bellek bölgesi veya register olabilir.

```
mov ax,1234    → AX = 1234h
push ax        → AX > STACK , SP = SP-2
mov ah,01      → AH = 01 ,AX'in değeri değişti!
pop ax         → AX = 1234h
```

Örneğimizde önce AX'e bir değer atanıyor ve daha sonra AH'ın değeri değiştirilmek sureti ile dolaylı olarak AX'inde değeri değiştiriliyor. Son işlemde de POP komutu ile AX'in önceden stack üzerine PUSH ile atılan eski değeri geri alınıyor.

3.1.7 POPF Komutu

POP komutu ile PUSH eşleştirilir ise PUSHF komutu ile de POPF komutunu eşleştirmek yanlış olmaz. POPF komutu ile stack üzerinden 16-bit'lik bilgi flag register'a yazılır. Alınan 16 bitin hepsi işlemci tarafından dikkate alınmaz. Bitlerin word içerisindeki sıralarına göre işlemci için ifade ettikleri değerler aşağıda verilmiştir.

0. bit → Carry biti
2. bit → Parity biti
4. bit → Auxilary biti
6. bit → Zero biti
7. bit → Sign biti
8. bit → Trap biti
9. bit → Interrupt biti
10. bit → Direction biti
11. bit → Overflow biti

POPF komutu anlaşılacağı üzere flag register'ın değerini tamamen değiştirmektedir. Komut tıpkı PUSHF komutunda olduğu gibi tek başına kullanılır.

PUSH, PUSHF, POP ve POPF komutlarının stack üzerinde işleyişlerini tam olarak kavrayabilmek için yazının daha önceki bölümlerinde yer alan "Stack" kısmını tekrar okumanızı tavsiye ederim.

3.1.8 LAHF Komutu

LAHF (Load AH from Flags) komutu AH register'ına flag register'ın düşük seviyeli baytını kopyalar. Kopyalanan bitler sign, zero, auxilary, parity ve carry bitleridir. Bunların dışında kalan overflow, direction, interrupt ve trace bitlerinin komut ile bir ilgisi yoktur. Komutun kullanımı aşağıdaki gibidir.

```
lahf
```

Görüldüğü gibi komut tek başına kullanılmaktadır. Komutun icrası sırasında flag register'da herhangi bir değişiklik olmaz.

3.1.9 SAHF Komutu

SAHF (Store AH into Flags) komutu da LAHF komutu gibi flag register üzerinde işlem yapar. AH içerisindeki değer flag register'ın düşük seviyeli baytına kopyalanır. Yine işlemden

etkilenen bitler sign, zero, auxilary, parity ve carry bitleridir. Komutun kullanımı LAHF komutunda olduğu gibi tek başınadır.

3.2 Giriş/Çıkış Komutları

80x86 serisi işlemcilerde giriş ve çıkış birimlerine ulaşmak için "in" ve "out" komutları kullanılır. Aslında bu komutlar bir bakıma MOV komutu ile benzer şekilde iş yapmaktadır. Tek fark bu komutların özel olarak bilgisayarın G/Ç birimi için ayrılmış olan bellek bölgesi ile çalışmalarıdır.

3.2.1 IN Komutu

```
in ax/al, port
in ax/al, dx
```

IN komutunun genel kullanımı yukarıda gösterilmiştir. IN komutu ile "port" veya dx ile belirtilen port adresinden okunan bilgi boyutuna göre AX yada AL içerisine kopyalanır. Erişilmek istene port 0-255 arasında ise port numarası kullanılır. Aksi taktirde, yani erişilmek istenen port 256 ve 80x86 işlemcinin maksimum desteklediği port numarası olan 65535 arasında ise istenen port numarası DX içerisine atılır ve daha sonra IN komutu ile erişim sağlanır. Komutun icrasında flag register herhangi bir değişikliğe uğramaz.

3.2.2 OUT Komutu

```
out port, ax/al
out dx, ax/al
```

OUT komutunun kullanımı IN komutu ile benzerlik göstermektedir. Tek fark IN komutunda port içerisindeki bilgi AX/AL'ye atanırken OUT komutunda AX/AL içerisindeki bilgi porta gönderilir. Yine IN komutunda olduğu gibi 0-255 arası adreslerde doğrudan port numarası girilirken 256-65535 arası adreslerde DX ile adresleme yapılır. Komutun icrasında flag register herhangi bir değişikliğe uğramaz.

```
mov al, 2e    → AL = 2Eh
out 70, al    → 70h. Porta 2E değeri gönderiliyor
in al,71      → Gönderilen isteğe karşılık 71h.
                porttan geliyor. Gelen değer AL
                içerisine alınıyor.
```

Yukarıda OUT komutu ile 70h CMOS portuna bilgi yollanıyor. CMOS gelen sinyale 71h portundan cevap veriyor ve biz gelen cevabı IN komutu ile AL içerisine kaydediyoruz.

3.3 Aritmetiksel Komutlar

80x86 programcıya toplama, çıkarma, çarpma, bölme gibi temel aritmetiksel işlemlerin yanı sıra elde edilen sonuçları değişik biçimlerde saklama olanağı sağlar. Aritmetiksel komutların icrası sırasında flag register değişikliğe uğramaktadır.

3.3.1 ADD Komutu

ADD komutu toplama işlemini gerçekleştirmek için kullanılır. Genel formu aşağıdaki gibidir.

add hedef, kaynak

ADD komutu ile "kaynak" içerisindeki değer "hedef" ile toplanıp "hedef" içerisine kaydedilir. "hedef" ve "kaynak alanları register-register, bellek-register, register-bellek çiftlerinden birisi olabilir.

```
mov ax, 1234                → AX = 1234h
mov word ptr [4444], 1000   → [4444] = 1000h
add word ptr ax, [4444]     → AX = AX + [4444]
                                 = 1234h + 1000h
                                 = 2234h
```

Yukarıdaki örnekte ilk önce AX'e 1234h değeri atanmıştır. Daha sonra 4444h adresli bellek gözüne word uzunluklu 1000h değeri yazılmıştır (Bu işlem için iki bellek gözü kullanılmıştır). En son olarak da AX içerisindeki değeri 4444h bellek gözü ile işaret edilen word ile toplanıp sonuç yine AX içerisine atılmıştır.

3.3.2 ADC Komutu

ADC komutu da tıpkı ADD komutu gibi toplama işlemi için kullanılır. Tek fark ADC komutunda toplama bir de carry flag'ın değerinin eklenmesidir. Genel formu aşağıdaki gibidir.

adc hedef, kaynak

Yapılan işlemi aritmetiksel olarak göstermek gerekirse aşağıdaki gösterim yanlış olmayacaktır.

hedef = hedef + kaynak + carry flag'ın değeri

ADC komutu ile peş peşe yapılan toplama işlemlerinde eldelik sayının göz ardı edilmemesi sağlanmaktadır.

3.3.3 SUB Komutu

SUB komutu çıkarma işlemi için kullanılır. Kullanımı ADD komutunda olduğu gibidir.

sub hedef, kaynak

"kaynak" içerisindeki değer "hedef" içerisinden çıkartılıp sonuç "hedef" içerisinde saklanır. İşlemin aritmetiksel gösterimi

hedef = hedef – kaynak

şeklindedir. Aslında CPU, SUB komutu ile "hedef" ile "-kaynak" değerlerini toplamaktadır. Gerçekte yapılan işlem yine bir toplama işlemidir.

```
mov ax, 8da7      → AX = 8DA7h
mov bx, 4a43      → BX = 4A43h
sub ax, bx        → AX = AX – BX
                       = 8DA7h – 4A43h
                       = 4364h
```

3.3.4 SBB Komutu

sbb hedef, kaynak

SBB komutu ile SUB arasındaki ilişki, ADD komutu ile ADC arasındaki ilişki ile aynıdır. SUB komutu ile aynı işe yapılır yalnız burada "hedef" alana atılan değerden carry flag'ın değeri de çıkartılır. İşlemin aritmetiksel gösterimi

hedef = hedef – kaynak – carry flag'ın değeri

3.3.5 MUL Komutu

MUL komutu çarpma işlemini gerçekleştirmek için kullanılan komuttur. Aritmetiksel olarak çarpma işlemi iki değer ile gerçekleştirilmesine karşın MUL komutu sadece bir değer alır. MUL komutu ile kullanılan değer gizli olarak ax/al içerisindeki değer ile çarpma işlemine tabi tutulur.

```
mov ax,0045     → AX = 0045h
mov bx,11ac     → BX = 11ACh
mul bx          → AX = AX * BX
                     = 0045h * 11Ach
                     = C35Ch
```

Yukarıda çarpma işleminin bir elamanı olan 0045h sayısı AX içerisine atılmıştır. Bir sonraki adımda işleme sokulmak istenen diğer sayı olan 11ACh sayısı BX içerisine atıldıktan sonra "mul bx" komutu ile BX içerisindeki sayı doğrudan AL (AH=00) ile işleme sokuluyor ve elde edilen çarpım AX içerisinde saklanıyor.

```
mov ax, 0005                  → AX = 0005h
mov word ptr [4000], 1212     → [4000] = 12 , [4001] = 12
mul word ptr [4000]           → AX = AX*[4001][4000]
```

Yukarıda çarpma işlemi için kullanılacak ikinci sayımız bir bellek bölgesinden okunmaktadır. Dikkat ederseniz işlemlerimde "word ptr" ile atama yaptığım değerin uzunluğu hakkında işlemciye bilgi veriyorum. MUL komutunu kullanırken işlem yapacağınız sayı bir bellek bölgesinde ise "word ptr" ve "byte ptr" gibi yardımcı bilgilerle işlemciye üzerinde işlem yapılacak bilginin uzunluğu hakkında bilgi vermeniz gerekmektedir

Diyelim ki çarpma işlemi sonunda bulduğunuz sonuç 16-bit'lik bir alana sığmıyor. Böyle bir durumda bulunan sonuç DX:AX ikilisi içerisinde saklanır.

```
mov ax, 4321        → AX = 4321h
mov cx, 4586        → CX = 4586h
mul cx              →  AX * CX = 4321h * 4586h
                       = 123B0846h (32-bit)
                       = DX = 123Bh , AX =  0846h
```

3.3.6 IMUL Komutu

IMUL komutu da MUL komutu gibi çarpma işlemi için kullanılır tek fark IMUL komutunun işaretli sayılar üzerindeki işlemler için kullanılan bir komut olmasıdır. IMUL komutu ile gerçekleştirilen bir işlem sonucunda AH (sonuç AX içerisine sığıyor ise) veya DX (sonuç AX içerisine sığmıyor ise) sıfırdan farklı ise çarpma işleminin sonucu negatiftir. Eğer işlem sonucunda bir taşma olmuş ve AH veya DX içerisinde sonucun işaretine değil de kendisine ait bir değer varsa carry ve overflow flag'ları set edilmiş olacaktır.

3.3.7 DIV Komutu

Bölme işlemi için kullanılan bir komuttur. DIV komut da MUL komutundan olduğu gibi sadece bir değer ile işleme girer ve gizli olarak AX register'ını kullanır. Genel formu aşağıdaki gibidir.

```
div bölen_değer
```

Bölme işleminde "bölen_değeri"in uzunluğu, işlem sırasında kullanılacak bölünen değerin uzunluğunu da belirler. Sözgelimi, "bölen_değeri"in 8-bit'lik bir değer olması halinde bölünen olarak 16-bit'lik AX register'ı işleme alınacaktır. İşlem sonunda bölüm değeri AL, kalan değeri de AH içerisine kopyalanır. Aynı şekilde "bölen_değer"i 16-bit'lik bir değer ise bölünün değeri olarak DX:AX çifti işleme alınır. Yine işlem sonundaki bölüm değeri AX ve kalan değeri de DX içerisine atılır.

8-bit'lik bir sayıyı yine 8-bit'lik bir sayıya ve 16-bit'lik bir sayıyı yine 16-bit'lik bir sayıya bölmek için sırasıyla AH ve DX register'larına 0 değeri atanır.

"bölen_değer" olarak bir bellek bölgesi kullanılması halinde işlemciye kullanılan değerin uzunluğu hakkında bilgi verilmelidir.

```
mov ax, 4a2c
div  byte ptr [2155]
```

DIV komutunu kullanırken dikkat edilmesi gereken bir husus da sıfır ile bölme durumu ile karşılaşmamak ve bölüm kısmındaki değerin alıcı alana sığıp sığmadığıdır. Mesela aşağıdaki gibi bir işlem sonunda hata ile karşılaşılacaktır.

```
mov ax, aaaa
div 4
```

Yukarıdaki işlemin sonucu (AAAAh / 4H = 2AAAH → 16-bit) olan 2AAAh sayısı 8-bit bir register olan AL içerisine sığmayacağı için hatalı olacaktır.

3.3.8 IDIV Komutu

IDIV komutu DIV komutu gibi bölme işlemi için kullanılır. DIV komutundan farkı, IDIV komutunun işaretli sayılar üzerinde işlem yapmak için kullanılmasıdır. DIV komutu için yukarıda anlatılanların dışında IDIV komutunun kullanımında dikkat edilmesi gereken bir nokta AH veya DX değerleri sıfırlanırken sayının işaret bitinin korunmasıdır. Mesela işleme konulacak bölen değer 8-bit bir negatif sayı ve bölünen de 8-bit bir sayı ise AH'ın bütün bitlerine AL içerisindeki 8-bit'lik negatif sayının işaret biti olan 1 değeri atanmalıdır.

3.3.9 INC Komutu

INC komutu kendisine verilen register yada bellek bölgesi içerisindeki değeri bir arttırır. C dilindeki "++" komutu ile aynı işi yapmaktadır. Aşağıda, komutun kullanımını C dilindeki gibi göstermeye çalıştım.

```
mov ax, 000f        → degisken = 15 ;
inc ax              → degisken++ ;
```

Flag register üzerinde, carry flag dışında, ADD komutu ile aynı etkiyi yapar.

```
add ax,1
inc ax
```

Yukarıdaki iki komut da aynı işi yapmaktadır. Fakat INC komutu ile gerçekleştirilen işlem, işlemci tarafından daha hızlı bir şekilde gerçekleştirilir ve bellekte kapladığı alan ilkinin üçte biri kadardır. Bu sebeplerden "add ax,1" gibi kullanımlar yerine "inc ax" komutu kullanılmaktadır.

```
inc byte ptr [125a]
inc word ptr [7ad8]
inc byte ptr [bx+di]
```

Bir bellek bölgesindeki değer INC komutu ile işleme alınacaksa, işlemciye üzerinde işlem yapılacak bilginin uzunluğu belirtilmelidir.

3.3.10 DEC Komutu

DEC komutu kendisine verilen register yada bellek bölgesi içerisindeki değeri bir azaltır. Yine INC komutundan olduğu gibi C dilinden bir örnek vermek istiyorum.

```
mov ax, 000f        → degisken = 15 ;
dec ax              → degisken-- ;
```

Yukarıda da görüldüğü gibi DEC komutu C dilindeki "--" komutuna karşılık gelmektedir. DEC komutu ile yapılan bir azaltma işlemi "sub ax,1" komutu ile de gerçekleştirilebilirdi fakat DEC komutu SUB komutuna göre bellekte daha az yer kaplar ve daha hızlı çalışır.

```
dec word ptr a8[bx+di]
```

DEC komutu ile azaltılacak değer bir bellek bölgesinde ise değerin uzunluğu "word ptr" yada "byte ptr" ile işlemciye bildirilmelidir.

3.3.11 CMP Komutu

CMP komutu SUB komutu gibi çalışır.

```
cmp deger1, deger2
```

Yukarıda "deger1" ve "deger2" yerine sırasıyla register-register, register-bellek bölgesi, bellek bölgesi-register ve register-sabit değer değerleri kullanılabilir.

CMP komutu kısaca "deger1"den "deger2"yi çıkarmaktadır. Fakat SUB komutundan farklı olarak sonuç herhangi bir yere kaydedilmez. Onun yerine, işlem sonucunda değişen flag register bitlerinin değerine göre programda dallanma yapılır. CMP komutundan sonra sonuca göre genellikle koşullu dallanma komutları ile programın akışı değiştirilir.

```
cmp ax, bx
```

Komut flag register üzerinde kısaca şu gibi etkilerde bulunur :

- AX, BX'e eşit ise işlemin sonucu sıfır olur ve zero flag 1 değerini alır.
- AX'in değeri BX'inkinden küçük ise çıkarma işlemi sonunda carry flag 1 değerini alır.
- Sonuç negatif ise sign biti 1, pozitif ise 0 değerini alır.

3.4 Mantıksal Komutlar

Bu komutlar herhangi bir işlem sırasında mantıksal karşılaştırmalar yapmak için kullanılır. Bölüm içerisindeki komutlar incelenmeden önce dokümanın "*Bitler Üzerinde Mantıksal İşlemler*" kısmının tekrar gözden geçirilmesinde fayda var.

Bu komutlar icra görürken kullanılan register yada bellek bölgesinin içerisindeki değerler ayrı ayrı bitler halinde işleme sokulur.

3.4.1 AND Komutu

```
and hedef, kaynak
```

Komutun kullanımında "hedef" ve "kaynak" alanlarına sırasıyla register-register, bellek bölgesi-register, register-bellek bölgesi, register-sabit değer ve sabit değer-register çiftlerinden biri kullanılabilir.

İşlem sırasında "hedef" ve "kaynak" bölgesindeki değerler mantıksak VE işlemine sokulur ve işlemin sonuncu "hedef" alana kaydedilir.

```
mov ah, a5   → AH = A5h = 1010 0101 (binary)
mov al, c1   → AL = C1h =  1100 0001 (binary)
and ah, al   →          AND_____________________
                        1000 0001 (binary) = 81h
```

Yukarıda AH'a A5h ve AL'ye C1h değerleri atanmış ve daha sonra "and ah, al" komutu ile iki değer "mantıksal ve" işlemine sokulmuştur. İşlem sonucunda 81h değeri elde edilmiştir. Elde edilen değer AH registeri içerine atılmıştır.

3.4.2 TEST Komutu

TEST komutu tamamen AND komutu gibi çalışır. Tek fark elde edilen sonucun hedef alana aktarılmaması onun yerine değişen flag bitlerine göre programın akışının kontrol edilmesidir.

3.4.3 OR Komutu

```
or hedef, kaynak
```

OR komutu "mantıksal veya" işlemini gerçekleştirmek için kullanılır. "hedef" ve "kaynak" alanları yerine kullanılabilecek değerler AND komutu ile aynıdır. İşlem gerçekleştikten sonra elde edilen sonuç hedef alan içerisine kaydedilir.

3.4.4 XOR Komutu

```
xor hedef, kaynak
```

"hedef" ve "kaynak" alanları için kullanılabilecek değerler AND komutu ile aynıdır. İşlem sonucunda elde edilen değer hedef alan içerisine kaydedilir.

XOR komutu herhangi bir register'ın değerini sıfır yapmak için sıkça kullanılır. "xor ax, ax" komutu "mov ax, 0" komutundan daha hızlı çalışır ve bellekte daha az yer kaplar.

3.4.5 NOT Komutu

not hedef

NOT komutu diğer mantıksal komutlardan ayrı olarak flag register üzerine etki etmeyen tek komuttur. Kullanım şekli ile de diğer komutlardan farklılık gösterir. "hedef" alan içerisindeki değerin bitleri ters çevrilip yine aynı alana yazılır.

3.5 Kaydırma ve Döndürme Komutları

Bu bölümde daha önce anlatılan "*Shift (Kaydırma) ve Rotate (Döndürme) İşlemleri*" başlıklı kısımdaki işlemlerin assembly komutları ile gerçekleştirilmesi anlatılacaktır.

3.5.1 SHL/SAL Komutları

SHL (Shift Left) ve SAL (Shift Arithmetic Left) komutları eştir.

shl hedef, sayaç

Her iki komut da "hedef" alan içerisindeki bit pozisyonunu "sayaç" defa sola kaydırır. Kaydırma işlemi sırasında 0. bit pozisyonuna sıfır yazılır ve 7. veya 15. bit pozisyonundaki değer de carry flag içerisine yazılır.

3.5.2 SHR Komutu

SHL (Shift Right) komutunun kullanımı aşağıdaki gibidir.

shr hedef, sayaç

SHR komutu ile "hedef" bölgesindeki bayt dizilimi "sayaç" defa sağa kaydırılır. Sağa kaydırma işleminde boşta kalan 7. yada 15. bit pozisyonuna 0 değeri atanır ve 0. Bit pozisyonundaki değer de carry flag'a yazılır. Sign biti sonucun her zaman sıfır olan yüksek seviyeli bit değerini alır.

Aritmetiksel olarak herhangi işaretsiz bir sayıyı "SHR sayı, a" işlemine sokmak, sayıyı 2^a ile bölmek ile aynı şeydir.

3.5.3 SAR Komutu

sar hedef, sayaç

SAR (Shift Arithmetic Right) komutu SHR komutunda olduğu gibi "hedef" alan içerisindeki bit dizilimini "sayaç" kere sağa kaydırır. SHR komutundan farkı, sondaki 7. veya 15. bit pozisyonuna sıfır değil de yine 7. veya 15. bit pozisyonundaki değerin yazılmasıdır. Flag'lar üzerine etkisi, SHR'den farklı olarak, sign flag içerisine 7. veya 15. bit pozisyonundaki değerin yazılmasıdır.

3.5.4 RCL Komutu

```
rcl hedef, sayaç
```

RCL (Rotate through Carry Left) komutu hedef alandaki bit dizilişini sola döndürme hareketine sokar. İşlem sonrasında carry flag içerisindeki değer 0. bit pozisyonuna yazılır ve arta kalan bit de carry flag içerisine aktarılır.

3.5.5 ROL Komutu

```
rol hedef, sayaç
```

ROL (Rotate Left) komutu da RCL komutu gibi sola döndürme işlemi için kullanılır. Tek farkı en sonda boş kalan 0. bit pozisyonuna carry flag'ın değerinin değil de en yüksek seviyeli bitin değerinin yazılmasıdır. Bu işlemde de yine yüksek seviyeli bit değeri carry flag içerisine kopyalanır.

3.5.6 RCR Komutu

```
rcr hedef, kaynak
```

RCR (Rotate through Carry Right) komutu da aynen RCL komutu gibi çalışmaktadır. Adından da anlaşılacağı gibi tek farkı sadece döndürme işleminin sola değil sağa doğru yapılmasıdır. Yine düşük seviyeli bit pozisyonundaki değer carry flag içerisine ve carry flag içerisindeki değer de yüksek seviyeli bit pozisyonuna kopyalanır.

3.5.7 ROR Komutu

```
ror hedef, kaynak
```

ROR (Rotate Right) komutunu RCR komutu ile ROL komutunu RCL komutu ile ilişkilendirdiğimiz şekilde ilişkilendirebiliriz. Yani ROR komutu da RCR komutu gibi çalışmaktadır. Tek fark ROR komutundan yüksek seviyeli bit pozisyonuna atanan değer carry flag içerisindeki değil alçak seviyeli bit pozisyonu içindeki değerdir.

NOT: Yukarıda "shift" ve "rotate" işlemlerini gerçekleştirilen komutlarda kullanılan "sayaç" ifadesi normalde 1 olarak kullanılır. 80286 ve sonrası işlemcilerde herhangi bir sabit değer kullanımı getirilmiştir. Daha önceki işlemcilerde 1 dışında bir değer kullanmak için aşağıdaki yol izlenmelidir.

```
mov cl, sayaç
shr ax, sayaç
```

Yukarıdaki gibi CL içerisine istenilen değer atanır ve daha sonra hedef alan ile işleme sokulur.

3.6 Dallanma Komutları

Dallanma komutları, programın normalde yukarıdan aşağı doğru giden akışını herhangi bir koşula bağlı olarak yada koşulsuz olarak başka bir yere yönlendirmek amacı ile kullanılır. Kullanılan bu dallanma komutları yüksek seviyeli dillerdeki "if" veya "goto" komutları gibi düşünülebilir.

Assembly dilinde kullanılan dallanma komutları atlama yaptıkları yere göre FAR veya NEAR özelliği taşır. NEAR özelliği taşıyan komutlar 2-3 bayt yer tutarken FAR özellikli olanlar 5 bayt yer tutmaktadır. Komutlar arasındaki fark, FAR özelliği taşıyanların farklı segment içerisindeki noktalara dallanma için kullanılmasıdır. Gerekli olmamakla beraber dallanmanın türü "FAR PTR" yada "NEAR PTR" ile belirtilebilir.

3.6.1 JMP (Koşulsuz Dallanma) Komutu

```
jmp hedef
```

JMP komutu belirtilen herhangi bir noktaya koşulsuz olarak dallanma yamak için kullanılır. DEBUG kullanılarak yazılan programlarda gidilecek noktanın offset adresi "hedef" kısmına yazılır. Eğer farklı bit segment içerisindeki bir adrese dallanma yapılıyorsa "hedef" kısmında SEGMENT:OFFSET şeklinde bir adresleme kullanılır.

Eğer aynı segment içerisinde –128 veya +127 bayt'dan daha uzak noktalara dallanma yapılıyorsa bu dallanma da FAR özelliği taşır. Bellekte kapladığı alan 3 bayt olacaktır.

Aslında JMP komutu ile yapılan sadece IP değerine "gidilecek_adres - bulunulan_adres" değerini eklemektir. JMP komutunun icrası sırasında IP'ye eklenecek değer JMP komutundan bir sonraki komutun adresinden itibaren hesaplanır.

Dallanama yapılırken doğrudan gidilecek adres yazılmayıp, bellek adresleme yolu ile de işlem gerçekleştirilebilir.

```
mov bx, 0401
mov si, 0002
mov word ptr [bx+si+04], 1a8c
jmp far ptr [bx+si+04]
```

Yukarıdaki örnek JMP komutu ile gidilecek nokta BX+SI+04 = 0407h değeri değil aynı segment içerisinde 0407h numaralı offset içerisindeki değerin gösterdiği yerdir. Örneğimizde bu değer 1A8Ch değeridir.

Benzer şekilde herhangi bir genel amaçlı16-bit register kullanılarak da dallanma işlemi gerçekleştirilebilir.

```
mov ax, 1111
jmp ax
```

Yukarıda "JMP AX" komutundan sonra programın akışı 1111h numaralı offset adresinden devam edecektir.

Başka bir segment içerisine dallanma yapmak için SEGMENT:OFFSET kalıbı kullanılır. Yani 4C70:0100 adresine dallanma yapmak için "JMP 4C70:0100" komutunu kullanmak gerekir. Komut bellekte 5 bayt yer kaplayacaktır.

Yukarıda ele alınan JMP komutu dışında assembly dilinde kullanılan koşullu dallanma komutları da vardır. Bu komutlar herhangi bir işlem sonucunda flag register'ların değerine göre programın akışını etkilerler. Bu komutlar C'deki "if" ve Pascal'daki "if...then" kalıbı ile eşleştirilebilir.

Karşılaştırma işlemi herhangi bir aritmetiksel işlem olabileceği gibi bir karşılaştırma (CMP) yada döndürme olabilir. Fakat koşullu dallanma komutları çoğunlukla CMP komutundan sonra karşılaştırma amaçlı kullanılır.

3.6.2 JZ/JE Komutları

JZ/JE (Jump if Zero/Jump if Equal) komutları herhangi bir işlem sonrasında zero flag'ın değerine göre programın akışını düzenler. Komutun icrası sırasında zero flag içerisindeki bit değeri 1 ise program JZ komutu ile gösterilen yere atlar aksi taktirde işlemci JZ komut yokmuş gibi programın akışına devam eder.

```
dec cx
cmp cx,0
jz 010a
jmp 0110
```

Yukarıdaki örnekte CX içerisindeki değer bir azaltılıyor. Daha sonra elde edilen değer sıfır ile karşılaştırılıyor ve eğer sonuç sıfır ise 010Ah adresine atlanıyor. Sonuç sıfırdan farklı ise program normal akışına devam ediyor ve bir sonraki komut olan "JMP 0110" çalıştırılıyor. Bu sefer koşulsuz olarak 0110h adresine bir dallanma yapılıyor.

Örneğimizdeki yazım şekli DEBUG içerisinde kullanılan yazım şeklidir. Assembly programlarını bir assembler programı aracılığı ile birleştiren programcılar için yazım şekli biraz daha farklıdır. Assembler kullanan programcılar offset adresleri yerine programın çeşitli yerlerine koydukları etiketleri kullanırlar.

```
.model small
.code
org 0100h
basla:
        mov dl,41h
        mov ah,02h
        int 21h
dongu:  inc dl
        int 21h
        cmp dl,5Ah
        jnz dongu

        mov ah,4Ch
```

```
                int 21h
end basla
```

Yukarıda MASM ile yazılmış bir assembly programı görülmektedir. Bu aşamada programı anlamaya çalışmayın yalnızca program içerisinde kullanılan etikete dikkat edin. “INC DL” komutunun olduğu satır “dongu” etiketi ile işaretlenmiştir. “JNZ dongu” komutu ile bir offset adresi belirtilmeyip bir etiket yardımı ile istenilen noktaya atlama yapılmıştır. Yukarıda DL içerisindeki değer 41h’tan başlayıp 5Ah olana kadar arttırılmaktadır.

Pratikte, DEBUG program yazmak için uygun bir ortam değildir. Yazacağınız assembly programını herhangi bir assembler yazılımı ile birleştirip kullanırsınız.

Yukarıdaki program COM dosyası olarak derlenirse 19 bayt alan kaplar ve ekrana alfabedeki büyük harfleri yazar.

3.6.3 JNZ/JNE Komutları

JNZ/JNE (Jump if Not Zero/Jump if Not Equal) komutları JZ ve JE komutlarının zıttı olarak kullanılır. Herhangi bir işlem sonrasında zero flag içerisindeki değer sıfır değil ise program komutun gösterdiği yere dallanarak akışına devam eder.

Örnek olarak JZ için verilen MASM programı incelenebilir. “jnz dongu” komutu ile zero flag içerisindeki değer kontrol ediliyor. Değerin sıfırdan farklı olması halinde “dongu” ile işaretli yere gidiliyor ve döngü tekrar işletiliyor. En son olarak DL içerisindeki değer 5Ah’a ulaşınca programın akışı bir sonraki komut olan “mov ah, 4ch” satırına geçiyor.

Yukarıdaki program ile aynı işi yapan C programını aşağıda verdim. Assembly ile yazıla COM dosyasının boyu 19 bayt iken C ile yazılanınki 8378 bayt!

```
#include <stdio.h>

int main()
{
  int ch=65;   /* hex 41 */

dongu:
            if (ch!=90) /* hex 5A */
              {
                printf("%c",ch);
                ch++;
                goto dongu;
              }

    return 0;
}
```

C örneğinde, ilk programdaki “cmp” ve “jnz” ile gerçekleştirilen işlemler “if” ve “goto” deyimleri ile gerçekleştirilmiştir.

3.6.4 JB/JC/JNAE Komutları

JB/JC/JNAE (Jump if Below/Jump if Carry/Jump if Not Above or Equal) komutları carry flag’ın değerine göre dallanma gerçekleştirirler. Herhangi bir işlem sonrasında carry flag içerisinde 1 değeri varsa programın akışı JB komutu ile gösterilen yere gider. Aksi taktirde JB komutu dikkate alınmadan programın normal akışına devam edilir.

3.6.5 JBE/JNA Komutları

JBE/JNA (Jump if Below or Equal/Jump if Not Above) komutlarının icrası sırasında carry ve zero flag’ları kontrol edilir. Bir işlem sonrasında iki flag’dan birinin 1 olması durumunda JBE komutu ile gösterilen noktaya dallanma yapılır.

3.6.6 JNB/JNC/JAE Komutları

JNB/JNC/JAE (Jump if Not Below/Jump if Not Carry/Jump if Above or Equal) komutlarından birisi ile karşılaşıldığı zaman CPU carry flag’ın değerini kontrol eder. Carry flag içerisindeki değerin sıfır olması halinde JNB komutu ile gösterilen noktaya dallanma gerçekleştirilir.

3.6.7 JG/JNLE Komutları

JG/JNLE (Jump if Greater than/Jump if Not Less than or Equal) komutlarının gerçekleşmesi için gerekli koşul sign ve overflow flag’larının değerlerinin eşit olması veya zero flag içerisinde sıfır değerinin bulunmasıdır. Gerekli koşulların sağlanması halinde JG komutu ile gösterilen noktaya dallanma yapılacak ve programın akışı o noktadan devam edecektir.

3.6.8 JA/JNBE Komutları

JA/JNBE (Jump if Above/Jump if Not Below or Equal) komutlarının icrası sırasında dikkate alınan flag’lar carry ve zero flag’larıdır. Eğer herhangi bir işlem sonrasında bu iki flag bitinin de değeri sıfır ise programın akışı JA komutu ile gösterilen yerden devam eder. İki flag bitinden birisinin sıfır olmaması halinde program normal akışına devam edecektir.

3.6.9 JL/JNGE Komutları

JL/JNGE (Jump if Less than/Jump if Not Greater or Equal) komutlarının gerçekleşmesi için gerekli koşul carry ve overflow flag bitlerinin birbirinden farklı değerler taşımasıdır. İki bitin farklı değerler içermesi durumunda programın akışı JL komutu ile gösterilen noktaya yönlendirilir.

3.6.10 JLE/JNG Komutları

JLE/JNG (Jump if Less than or Equal/Jump if Greater) komutlarının icrası sırasında zero, sign ve overflow flag’larının değerleri dikkate alınır. Komut ile gösterilen yere dallanmak için zero flag içerisinde 1 değerinin olması yada sign ve overflow flag’larının değerlerinin farklı olması gerekmektedir. Gerekli koşul sağlanırsa programın akışı JLE komutu ile gösterilen noktaya kaydırılacaktır.

3.6.11 JS ve JNS Komutları

JS (Jump if Sign) ve JNS (Jump if No Sign) komutlarının icrası sırasında sign flag kontrol edilir. Sign flag içerisindeki değerin 1 olması JS için, sıfır olması JNS için gerekli koşuldur.

Gerekli koşulun sağlanması durumunda program JS veya JNS komutu ile gösterilen noktadan akışına devam edecektir.

3.6.12 JO ve JNO Komutları

JO (Jump if Overflow) ve JNO (Jump if No Overflow) komutlarının icrası sırasında overflow flag içerisindeki değere bakılır. Overflow flag içerisinde 1 JO komutu için ve sıfır olması JNO komutu için gerekli koşuldur.

3.6.13 JCXZ Komutu

JCXZ (Jump if CX is Zero) komutu CX register'ının değerini kontrol eder. Eğer CX içerisindeki değer sıfır ise program JCXZ komutu ile gösterilen yerden akışına devam eder.

Yukarıda verilen koşullu dallanma komutları ile sadece +128/-128 bayt'lık bir atlama gerçekleştirilebilir. Eğer program içerisinde daha uzak bir mesafeye dallanma yapmak gerekiyorsa aşağıdaki gibi bir yol izlenmelidir.

- JMP komutu ile istenilen noktaya dallanma yapılır
- JMP komutundan sonraki noktaya bir etiket verilir
- Kullanılacak komutun tersi JMP komutunun önüne yazılarak JMP komutundan sonraki etikete dallanma yapılır

Örneğin DEBUG içerisinde 0100 numaralı offsetde iken "JE 184" komutunu veremezsiniz. Bu komutu çalıştırmak için yapmanız gerekenler sırasıyla

```
???? : 0100 jne 105
???? : 0102 jmp 184
???? : 0105
```

komutlarını vermektir. Dikkatli bir şekilde incelenirse yukarıdaki komut kümesinin "JE 184" komutu ile aynı şeyi yaptığı görülecektir.

3.7 Döngü Komutları

Bu kategorideki komutlar herhangi bir rutini belirli kereler tekrarlamak için kullanılır. Assembly dilinde kullanılan döngü komutları bir bakıma C dilindeki "for" ve "while" döngüleri ile eşleştirilebilir.

3.7.1 LOOP Komutu

Bu komut herhangi bir durumu belirli kereler tekrarlamak için kullanılır. Sayaç olarak CX register'ı içerisindeki değer alınır. Her döngüde CX içerisindeki değer bir azaltılır ve CX'in değeri sıfır oluncaya kadar işlem devam eder. Komutun genel kullanım kalıbı aşağıdaki gibidir.

```
loop hedef
```

"hedef" ile belirtilen değer DEBUG için bir offset adresi, herhangi bir assembler kullanılarak birleştirilecek bir program için bir etiket olmalıdır. LOOP komutu ile yapılabilecek maksimum atlama +128/-128 bayt'tır. Komutun kullanımı gereği atlanılacak nokta LOOP komutundan önceki adreslerden birisi olduğu için maksimum gidilebileceğimiz adres –128 bayt mesafededir.

```
1E27:0100 XOR  AX, AX
1E27:0102 MOV  CX,0005
1E27:0105 INC  AX
1E27:0106 LOOP  0105
```

Yukarıdaki programın sonunda AX'in değeri 5 olacaktır. Programın başında CX içerisine atılan 5 değeri LOOP ile gerçekleştirilecek döngü sayısını belirlemektedir.

```
1E27:0100 XOR  AX, AX
1E27:0102 MOV  CX,0005
1E27:0105 INC  AX
1E27:0106 DEC  CX
1E27:0107 JNZ  0105
```

Yukarıdaki program bir önceki örnekteki program ile aynı işlemi gerçekleştirmektedir. Tek fark "LOOP 0105" komutu yerine "INC CX" ve "JNZ 0105" komutlarının kullanılmasıdır. Gerçekte de LOOP ile yapılan bu iki komutun yaptığı işlemden başka bir şey değildir.

3.7.2 LOOPZ/LOOPE Komutları

LOOPZ/LOOPE komutları da herhangi bir durumu belirli kereler tekrarlamak için kullanılır. Komutun kullanımı LOOP komutu ile aynıdır. Tek fark LOOPZ komutu icrası sırasında CX register'ının değerinin yanı sıra zero flag'ı da kontrol eder. CX'in değeri sıfırdan farklı olduğu sürece ve zero flag içerisindeki değer 1 olduğu sürece komut ile gösterilen yere dallanma yapılır.
Her iki durumdan birisinin gerçekleşmemesi halinde döngüden çıkılır.

3.7.3 LOOPNZ/LOOPNE Komutları

LOOPNZ/LOOPNE komutları da LOOP komutu gibi CX register'ını sayaç olarak kabul ederek bir koşulu belirli kereler tekrar etmek için kullanılır. Döngü işlemi CX'in değeri sıfırdan farklı iken ve zero flag içerisindeki değer 1 değil iken devam eder.

Bölüm 4 : Kesme (Interrupt) Kullanımı

Normal olarak işlemci, kod segment içerisindeki komutları sırası ile işletir. Bazı durumlarda bilgisayara ait çevre aygıtlar ile işlem yapmak üzere, RAM veya BIOS üzerindeki belirli rutinleri kullanmak üzere yada bazı geçersiz sonuçların oluşması durumunda işlemci tarafından kesme (interrupt) adı verilen rutinler çalıştırılır.

Benim burada kısaca değineceğim RAM yada BIOS üzerinde tanımlı alt rutinlerin programlar tarafından çağırılması ile oluşan kesmelerdir. Bu alt rutinler bir çok işi programcının yerine yapmaktadır. Bu alt rutinleri yüksek seviyeli dillerdeki hazır kütüphane fonksiyonları ile eşleştirmek yanlış olmaz.

Kullanılabilecek her kesmenin önceden tanımlı bir numarası vardır. Kesmenin kullanımı bu numaranın "INT" komutuna parametre olarak aktarılması ve gerekli register'lara değerlerin atanması ile gerçekleştirilir. Örnek olarak DOS altında çok sık kullanılan INT 21 verilebilir. Bir DOS kesmesi olan INT 21'in AH = 01 numaralı fonksiyonunu kullanalım.

İşlem için AH register'ının değeri 01h olmalı ve ardından "INT 21" komutu ile kesme çağırılmalıdır. INT 21 komutu ile kesme çağırıldığı zaman AH içerisindeki değere bakılır. AH içerisinde bulunan değere göre çalıştırılacak alt rutin belleğe yüklenir.

```
C:\WINDOWS\Desktop>debug
-a 100
1E27:0100 mov ah,01
1E27:0102 int 21
1E27:0104 cmp al,41
1E27:0106 jnz 100
1E27:0108 mov ah,4c
1E27:010A int 21
1E27:010C
-g
ssdfaA
C:\WINDOWS\Desktop>
```

Yukarıdaki program AH içerisine 01h değerini atadıktan sonra INT 21 komutu ile gerekli komutları belleğe yüklüyor. Program klavyeden girilen tuşları okur, okunan her tuş değerinin ASCII kodu AL içerisine yazılır (INT 21 AH=01h gereği). CMP komutu ile okunan tuşun değeri 41h (A) olana kadar programın çalıştırmasını sağlıyoruz. Daha sonra yeni bir INT 21 alt rutini ile karşılaşıyoruz. AH = 4Ch alt rutini çalıştırılmakta olan programdan çıkıp kontrolü tekrar DOS işletim sistemine bırakmaktadır.

Aynı şekilde, UNIX ve benzeri işletim sistemlerinde de sistem çağrıları (system calls) denen alt rutinler INT 80 ile belleğe yüklenip icraları gerçekleştirilmektedir.

Bu dokümanda sekmelerin fonksiyonları tek tek ele alınmayacaktır. Konu hakkında edinilebilecek en ayrıntılı kaynak "Ralf Brown's Interrupt List" adlı dokümandır. Dokümanların yanında "RBILViewer" adlı yardımcı program da size uzun kesme listelerini düzenli bir şekilde görüntüleme olanağı sağlayacaktır.

Bölüm 5 : DEBUG Programı

DEBUG, MS-DOS altında kullanılan ve kullanıcıya çalıştırılabilir dosyaları bellek üzerinde değiştirip kaydedebilme olanağı sağlayana bir yazılımdır. Bugün piyasada aynı işi yapan çok daha fonksiyonel ve gelişmiş programlar mevcuttur fakat DEBUG'ın MS-DOS ile birlikte gelmesi programın elde edilmesi konusundaki zorluğu ortadan kaldırmaktadır.

Bu bölümde DEBUG programı içerisinde kullanılan bazı komutları örnekleri ile ele aldım.

DEBUG programını başlatmak için MS-DOS kipinde "debug" komutun vermeniz yeterlidir. Komutu verdikten sonra aşağıdaki gibi bir ekran görüntüsü ile karşılaşırsınız.

```
Microsoft(R) Windows 98
   (C)Telif Hakkı Microsoft Corp 1981-1999.

C:\WINDOWS\Desktop>debug
-
```

Yukarıdaki ekran debug programını ilk çalıştırdığınız zaman karşınıza gelecek ekrandır. Program çalıştığı zaman sadece ekranın sol kenarında bir çizgi belirir. Bu aşamada debug sizden bir sonraki işlem için komut beklemektedir. Aşağıdaki tabloda en çok kullanılan komutları açıklamaları ile birlikte verdim.

A – Assemble	Belirli bir adresten itibaren program yazımı
G – Go	Bellekteki programın çalıştırılması
L – Load	Belleğe yükleme yapılması
N – Name	Yüklenecek veya yazılacak programın ismi
Q – Quit	DEBUG'tan çıkmak
R – Register	Register değeri atama
T – Trace	Adım adım işletim
U – Unassemble	Yüklü programın dağıtılması
W - Write	Bellekten diske yazma işlemi
? - Yardım	Kullanılabilecek komutların listesi

Yukarıda sadece kullanacağımız komutları açıklamaları ile birlikte verdim. Şimdi isterseniz MS-DOS kipinde debug'ı çalıştırıp bir program yazalım ve programı diske kaydedelim.

```
C:\WINDOWS\Desktop>debug
-A 100
1E27:0100 mov ah,02
1E27:0102 mov dh,07
1E27:0104 mov dl,0c
1E27:0106 int 10
1E27:0108 mov ah,4c
1E27:010A int 21
1E27:010C
-N set_curs.com
-R CX
CX 0000
:000C
-W
0000C bayt yazılıyor
-
```

Yukarıdaki program ile imleç DH ve DL ile belirtilen noktaya taşınıyor. Öncelikle programın debug içerisinde yazılması için "A" komutu kullanıldı. "A" komutu ile kullanılan 100 parametresi programın yazılmaya başlanacağı offset adresini gösteriyor. Yazacağımız program bir COM dosyası olduğu için başlangıç offseti 0100 olarak belirledik. Bir sonraki satırda assembly programımızı yazmaya başladık. Programın yazım kısmı bittikten sonra "N" komutu ile bellekte bulunan bilgiyi diske yazacağımız zaman kaydedilecek dosya ismini belirledik. "R" komutu ile CX register'ı içerisindeki değere programımızın bayt olarak uzunluğunu atıyoruz (Örneğimizde 010C – 0100 = 000C). En son olarak da "W" komutu ile bellekteki komutlar "set_curs.com" dosya adı ile diske kaydediliyor.

Şimdi de yazdığımız programı tekrar DEBUG yardımı ile dağıtalım.

```
C:\WINDOWS\Desktop>debug
-N set_curs.com
-L
-U
1E4A:0100 B402          MOV     AH,02
1E4A:0102 B607          MOV     DH,07
1E4A:0104 B20C          MOV     DL,0C
1E4A:0106 CD10          INT     10
1E4A:0108 B44C          MOV     AH,4C
1E4A:010A CD21          INT     21
1E4A:010C D6            DB      D6
1E4A:010D C70619DFAD82  MOV     WORD PTR [DF19],82AD
1E4A:0113 E8C901        CALL    02DF
1E4A:0116 0BC0          OR      AX,AX
1E4A:0118 7478          JZ      0192
1E4A:011A 8BE8          MOV     BP,AX
1E4A:011C BF02D2        MOV     DI,D202
1E4A:011F 8B36C6DB      MOV     SI,[DBC6]
-Q

C:\WINDOWS\Desktop>
```

Yukarıda yine komut satırından “debug” komutu verilerek DEBUG programı çalıştırılıyor. Bu sefer “N” komutu diske yazılacak dosyanın ismini değil, diskten belleğe aktarılacak dosyanın ismini vermek için kullanılıyor. Daha sonra “L” komutu ile ismi verilen dosya belleğe yükleniyor. Belleğe yüklenen program “U” komutu ile dağıtılıyor. Kodları dikkatli incelerseniz programın ilk altı satırının bizim bir önceki örnekte yazdığımız assembly kodlarından oluştuğunu göreceksiniz. Kodların solundaki sayılar assembly komutlarının bellekte bulundukları şeklidir (makine dili). 010C ve sonrasındaki satırlardaki assembly komutları o anda bellekte rasgele bulunan kodlardır. Bizim programımız ile bir ilgileri yoktur. En son olarak da “Q“ komutu ile DEBUG’tan çıkılıyor.

“T” komutu ile bellekteki herhangi bir program adım adım işletilebilir. Komutun kullanımına basit bir örnek aşağıda verilmiştir.

```
Microsoft(R) Windows 98
   (C)Telif Hakkı Microsoft Corp 1981-1999.

C:\WINDOWS\Desktop>debug
-A 100
1E2C:0100 mov ax,1234
1E2C:0103 mov bx,4567
1E2C:0106 mov ax,bx
1E2C:0108
-T = 100

AX=1234  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1E2C  ES=1E2C  SS=1E2C  CS=1E2C  IP=0103   NV UP EI PL NZ NA PO NC
1E2C:0103 BB6745        MOV     BX,4567
-T

AX=1234  BX=4567  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1E2C  ES=1E2C  SS=1E2C  CS=1E2C  IP=0106   NV UP EI PL NZ NA PO NC
1E2C:0106 89D8          MOV     AX,BX
```

```
-T

AX=4567  BX=4567  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1E2C  ES=1E2C  SS=1E2C  CS=1E2C  IP=0108   NV UP EI PL NZ NA PO NC
1E2C:0108 107420        ADC     [SI+20],DH                      DS:0020=FF
-
```

Gördüğünüz gibi DEBUG ile yazılan program adım adım işletilmektedir. “T” komutunu yazdığınız kısa kodların hata kontrolü için kullanabilirsiniz.

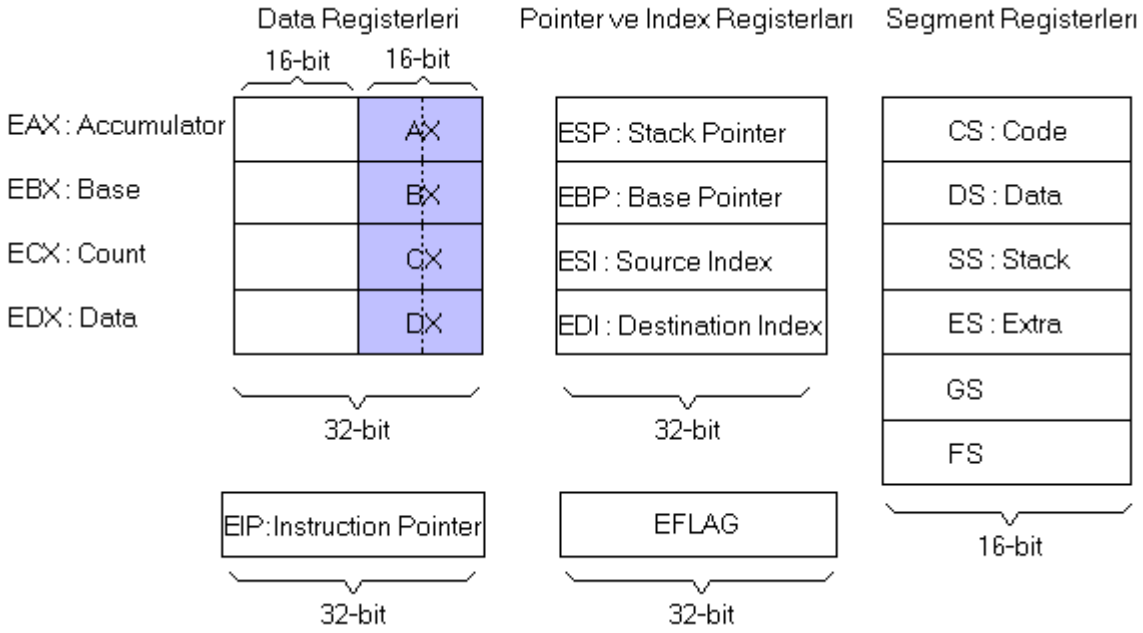
Bölüm 6 : Linux İşletim Sistemi Altında Assembly Kullanımı

Bu dokümanda linux’un kullanımı ve sistemin işleyişi hakkında bilgi verilmeyecektir. Bu bölümde kullanıcının daha önce Linux yada benzeri bir UNIX türevi sistemi kullanmış olduğu varsayılmaktadır.

Linux işletim sisteminin çalışabileceği en eski CPU modeli 80386 mimarisidir. Yani linux 8086, 80186 yada 80286 kullanan bir bilgisayar üzerine kurulamaz. Intel firmasının 80386 ile getirdiği en büyük yenilik, eski modellerde (8086, 80186 yada 80286) kullanılan 16-bit register’lar yerine 32-bit register’ların kullanımına olanak sağlamasıdır. Bu da programcıya daha çok adreslenebilir bellek alanı sağlamaktadır. 80386’dan önceki modellerde bir segment 64KB boyutunda olurken bu rakam 80386 ve sonrasında 4GB’dan daha fazladır. (Bu dokümanı yazdığım sıralarda Intel firması yeni 64-bit kapasiteli işlemciler üzerinde çalışıyordu!)

80386’da, daha önce kullanılan register’lara ek yapılmış ve bazı yeni register’lar eklenmiştir. Her biri 16-bit olan AX, BX, CX, DX, SI, DI, BP ve SP register’ları yerine her biri 32-bit olan, sırasıyla EAX, EBX, ECX, EDX, ESI, EDI, EBP ve ESP register’ları kullanılmaktadır. Daha önceki işlemcilerde kullanılan 16-bit segment register’ları olan CS, DS, ES ve SS’e ek olarak iki 16-bit’lik segment register’ı olan FS ve GS eklenmiştir. 80386’da önceki modeller gibi 16-bit’lik segment register’ları kullanmaktadır. Tüm bunlara ek olarak flag register da 32-bit kapasiteye sahip olmuştur. Genişletilmiş yeni flag register’ın 16. biti “debug resume”(RF) ve 17. biti “virtual mode” (VM) bitleri olarak adlandırılırlar. İşlemci sanal mod (virtual mode) altında çalışırken 8086 mimarisi ile işlem yapar. Bunalara ek olarak, 80486’da “alignment check flag” biti 18. bit pozisyonuna eklenmiştir.

32-bit programlar yazılırken 16-bit’lik ve 8-bit’lik register’lar kullanılabilmektedir. 80386’nın kullandığı register şeması aşağıdaki gibidir.



Linux altında yazacağımız programların hepsi 80386'nin sağladığı 32-bit kapasiteli register'ları kullanabilmektedir. Linux altında assembly dili kullanımı programcıya DOS altındaki kadar büyük avantajlar sağlamamaktadır. Çalıştırılan programlar korumalı mod (protected mode) altında çalıştığı için programcının BIOS tarafından sağlanan kaynaklara erişimi kısıtlanmıştır. (Aslında bu korumalı modda çalışan tüm sistemler için geçerlidir)

Linux altında kullanılacak assembly rutinleri çoğunlukla C gibi diller içerisinde satır içi (inline) olarak kullanılmaktadır. Yazılan programların DOS altında yüksek seviyeli bir dil ile ve assmbly ile yazılmış iki program arasındaki kadar büyük bir boyut farkı göstermemesi ve korumalı moddan dolayı kullanımdaki kısıtlamalar dilin Linux (yada herhangi bir UNIX türevi sistem) altındaki popülaritesini azaltmıştır. Fakat yine de bazı noktalarda başka alternatif bulunmamakta ve assembly dilinin kullanımı zorunlu olmaktadır. (Linux Kernel Ver1.0 kaynağındaki boot dosyaları, http://www.kernel.org)

Linux altında kaydedilen assembly dosyaları .s yada .S uzantısını alır. Program kullanılan assembler programına göre AT&T yada Intel sözdizimi kullanılarak yazılır. Eğer assembler olarak linux ve DOS altında kullanılan NASM (Netwide Assembler) kullanılacaksa yazılan program Intel sözdizimi kullanılarak yazılır. Dokümanda buraya kadar anlatılan assembly dili kuralları Intel sözdizimi içindir. Buna karşın linux altında sıkça kullanılan debugger programlarından birisi olan GDB ile yazılacak kodlarda, C ile kullanılacak satır içi kodlarda ve linux çekirdeğindeki kodlarda hep AT&T sözdizimi kullanılmaktadır. Aslında GCC derleme sırasında "as" adında bir assembler programı kullanır. "as", AT&T sözdizimini kullandığı için GCC ile derlenen C dosyalarındaki kodlar da mecburen AT&T sözdizimi ile yazılmalıdır.

Bu dokümanda temel olarak Intel ve AT&T sözdizimleri arasıdaki farklar üzerinde durulacaktır ve GNU Assembler kullanımına bazı örnekler verilecektir.

6.1 Intel ve AT&T Sözdizimleri

Yukarıda da belirttiğim gibi linux altında assembly programları yazılırken kullanılan sözdizimi AT&T standartlarındadır. DOS altında kullanılan assembler programları Intel sözdizimi ile yazılmış kodları birleştirirken, linux (ve UNIX türevi sistemlerde) kullanılanlar, bir iki istisna dışında AT&T sözdizimini şart koşmaktadır.

Intel	AT&T
mov eax, 01h mov bx,1234h	movl $0x01, %eax movw $0x1234, %bx

Yukarıda da görüldüğü gibi AT&T sözdizimi Intel'inkine göre biraz daha karmaşıktır. Şimdi isterseniz sıra ile farkları inceleyelim.

6.1.1 Kaynak-Hedef Yönü

Intel sözdiziminde genel form "komut hedef, kaynak" olmasına karşın AT&T'de bu dizilim "komut kaynak, hedef" şeklindedir. Aslında, göze normal gelen de değerin soldan sağa taşınmasıdır.

Intel	AT&T
komut hedef, kaynak mov ebp ,esp	komut kaynak, hedef movl %esp, %ebp

6.1.2 Önekler

AT&T sözdiziminde register'lar "%" ve sabit değerler de "$" işareti ile öneklerini alır. Sabit değer olarak kullanılan sayı onaltılık sistemde yazılıyor ise "0xsayı" gibi bir ifade kullanılır.

Intel	AT&T
mov ah, 12h	movb $0x12, %ah

Yukarıdaki örnekte AH içerisine 12h (onluk 18) değeri atanmıştır. AT&T sözdiziminde sabit değerin başında "$" işareti olduğuna ve 12h değerini temsil etmek için 0x12 yazım biçimini kullandığımıza dikkat edin. "movb $0x12, %ah" ile "movb $12, %ah" arasında çok fark vardır. İkinci komutun GDB içerisindeki karşılığı "movb $0xC, %ah" şeklinde olacaktır.

6.1.3 Sonekler

Sözdizimleri arasındaki bir diğer fark da AT&T sözdiziminde komutların sonuna konulan "l", "w" ve "b" sonekleridir. Bu sonekler üzerinde işlem yapılan bilginin boyu hakkında bilgi vermek için kullanılır. Intel sözdiziminde kullanılan "dword ptr", "word ptr" ve "byte ptr" eklerine karşılık gelmektedirler.

32-bit veri işlemleri için "l" (long), 16-bit veri işlemleri için "w" (word) ve 8-bit veri işlemeleri için "b" (byte) sonekleri kullanılır.

Intel	AT&T
mov byte ptr bh, 45h mov eax, dword ptr [ebx]	movb $0x45, %bh movl (%ebx), %eax

6.1.4 Bellek İşlemleri

80386'nın yeni register'ların yanı sıra programcılara sağladığı en büyük yenilik bellek adresleme metotlarındadır. Bu bölümü okumadan önce 8086 serisinde bellek adresleme konusunda bilgi sahibi olmanız gerekmektedir.

8086 serisinde dolaylı adresleme yapılırken sadece BX register'ı kullanılırken, 80386 ve sonrası işlemcilerde herhangi bir genel amaçlı (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) register kullanılabilmektedir. Yani aşağıdaki yazım 80386 öncesi bir işlemcide çalışmazken 80386 ve sonrasında sorunsuz çalışmaktadır.

```
mov al, [edx]
```

Dikkat edilmesi gereken bir nokta da 80386 işlemcide programınızı 16-bit gerçek modda (real mode) çalıştırıyorsanız yine adresleyebileceğiniz belek bölgesi 64KB boyundadır ve erişim sırasında 32-bit register'ları kullanmanız gerekmektedir.

80386 herhangi bir register'ı base yada index adresleme için kullanmanıza olanak sağlar. Tek kısıtlama ESP'yi bir index değer olarak kullanamayacağınızdır.

Bunların yanı sıra 80386'da "index" değeri 1, 2, 4 veya 8 gibi sabitlerden birisi ile çarpılıp daha etkin bir bellek erişimi sağlanabilmektedir. Genel Intel ve AT&T sözdizimi şekilleri aşağıda verilmiştir.

INTEL → komut hedef, segment_register:[base + index * n + yer_değiştirme]
AT&T → komut %segment_register:yer_değiştirme(base, index, n), hedef
(n = 1, 2, 4 veya 8)

Aşağıdaki örnekler, yukarıdaki formalrın kafanızda oluşturabileceği soru işaretlerini silecektir sanırım.

Intel	AT&T
mov eax, [ecx*8h] mov edx, [ebx+ecx*08h-05h]	movl 0x08(%ecx), %eax movl –0x05(%ebx,%ecx,0x08)

Bu noktada AT&T sözdizimi Intel'inki ile kıyaslandığında, gerçekten hiç de programcı dostu değil!

80386 ile yazılan kodlarda bellek adresleme yapılırken kullanılan register'lardan ilkinin "base" ve ikincisinin "index" olduğu varsayılır. Programcı bu noktada dikkatli olmalıdır. Çünkü işlemci bilgiyi alacağı segmenti kullanılan "base" register'a göre belirler. Sadece base olarak ESP yada EBP kullanıldığı zaman geçerli segment Stack Segment'tir. Diğer durumlarda varsayılan olarak Data Segment kullanılır.

6.1.5 INT 0x80 ve Linux Sistem Çağrıları (Syscalls)

Linux altında assembly programları yazılırken çok kullanılan rutinlerden bazıları da sistem çağrılarıdır. Herhangi bir sitem çağrısının kullanılabilmesi için, sistem çağrısına ait numara EAX içerisine atılır ve eğer sistem çağrısı en fazla beş agrüman alıyorsa gerekli argümanlar sırası ile EBX, ECX, EDX, ESI ve EDI register'larına atılır. Altı ve daha fazla argüman alan

sistem çağrılarında ise yine sistem çağrı numarası EAX içerisine atılır. Daha sonra gerekli agrümanlar sondan başa doğru stack içerisine atılır ve ESP EBX içerisine kopyalanır. Sistem çağrıları /usr/include/asm/unistd.h içerisinde listelenmiştir. Ayrıca sistem çağrıları hakkında bilgiyi de "man 2 sistem_çağrısının_adı" komutu ile alabilirsiniz.

Örnekler :

ilk.s

```
fnoyan@tux:~$ cd asm
fnoyan@tux:~/asm$ vi ilk.s
```

"ilk.s" dosyası içerisine aşağıdaki kodları yazın.

```
.text
.globl _start
_start:
                movl $12,%eax
                movl $0x12, %eax

                xorl %eax, %eax
                inc %eax
                int $0x80
```

```
fnoyan@tux:~/asm$ as ilk.s -o ilk.obj
fnoyan@tux:~/asm$ ld ilk.obj -o ilk
fnoyan@tux:~/asm$ ./ilk
fnoyan@tux:~/asm$
fnoyan@tux:~/asm$ gdb -q
(gdb) file ilk
Reading symbols from ilk...(no debugging symbols found)...done.
(gdb) disas _start
Dump of assembler code for function _start:
0x8048074 <_start>:     mov    $0xc,%eax
0x8048079 <_start+5>:   mov    $0x12,%eax
0x804807e <_start+10>:  xor    %eax,%eax
0x8048080 <_start+12>:  inc    %eax
0x8048081 <_start+13>:  int    $0x80
End of assembler dump.
(gdb)
```

Yukarıda ilk.s dosyasının adım adım birleştirilmesi ve GDB yardımı ile içerisindeki assembly kodlarını incelenmesi gösterilmiştir. Örneğimizde, GDB ile kullanılan "file" komutu debug

içerisindeki “n” komutu ile ve “disas” komutu da debug içerisindeki “u” komutu ile aynı işi yapmaktadır.

Bu örnekte yukarıda verdiğim “movl $12,%eax” ve “movl $0x12, %eax” komutları arasındaki farkı rahat bir şekilde görebilirsiniz.

“disas _start” komutu ile GDB programımız içerisindeki “_start” noktasından itibaren dağıtma işlemi yapar.

iki.s

Bu örneğimizde bir sistem çağrısı olan “chmod”un kullanımı örneklendirilmiştir. Sistem çağrı numarasını (chmod için 15) /usr/include/asm/unistd.h dosyasından öğrendikten sonra gerekli agrümanları linux yardım dosyalarını okuyarak öğrenebilirsiniz.

```
fnoyan@tux:~$ man 2 chmod
```

Gerekli argümanlar öğrenildikten sonra aşağıdaki kodu iki.s adı kaydedin.

```
.data
   path:   .ascii   "./dosya"
.text
.globl _start
_start:
                    movl $15, %eax
                    movl $path, %ebx
                    xorl %ecx, %ecx
                    int $0x80

                    xorl %eax, %eax
                    incl %eax
                    int $0x80
```

Yukarıdaki program, bulunduğu dizindeki “dosya” isimli dosyanın erişim haklarını sıfırlamaktadır. Yani programımız “chmod 000 ./dosya” komutu ile aynı işi yapmaktadır.

```
fnoyan@tux:~/asm$ touch dosya; chmod 777 dosya
fnoyan@tux:~/asm$ ls -l dosya
fnoyan@tux:~/asm$ as iki.s -o iki.obj
fnoyan@tux:~/asm$ ld iki.obj -o iki
fnoyan@tux:~/asm$ ./iki
fnoyan@tux:~/asm$ ls -l dosya
....
....
fnoyan@tux:~/asm$ gdb -q
(gdb) file iki
Reading symbols from iki...(no debugging symbols found)...done.
```

```
(gdb) disas _start
Dump of assembler code for function _start:
0x8048074 <_start>:     mov    $0xf,%eax
0x8048079 <_start+5>:   mov    $0x8049088,%ebx
0x804807e <_start+10>:  xor    %ecx,%ecx
0x8048080 <_start+12>:  int    $0x80
0x8048082 <_start+14>:  xor    %eax,%eax
0x8048084 <_start+16>:  inc    %eax
0x8048085 <_start+17>:  int    $0x80
End of assembler dump.
(gdb)
```

"dosya" isimli bir dosya oluşturduk ve erişim haklarını –rwxrwxrwx (777) olarak düzenledik. Daha sonra yazdığımız program ile "dosya" isimli dosyanın erişim haklarını ----------(000) olarak değiştirdik.

"man 2 chmod"a göre fonksiyon iki argüman alıyor. Birincisi dosyaya giden yol (path) ve ikincisi yeni erişim yetkileri (bizim örneğimizde sıfır). İsterseniz adım adım programı inceleyelim.

```
.data
   path:   .ascii   "./dosya"
```

Yukarıdaki parçada program içerisinde ascii türünde sabit tanımlaması yapılıyor. ".data" Data Segment içerisinde olduğumuzu belirtir.

```
.text
.globl _start
_start:
```

Code Segment'e geçiliyor.

```
movl $15, %eax
movl $path, %ebx
xorl %ecx, %ecx
int $0x80
```

İşte burada chmod sistem çağrısı çağırılıyor. EAX içersine sistem çağrı numarası atıldıktan sonra gerekli argümanlar register'lara yükleniyor. Burada "movl $path, %ebx" komutu ile yapılan sabitimizin adresini EBX içerisine yüklemektir (programın GDB çıktısında bu görülmektedir.). Daha sonra ECX içerisine sıfır atanıyor ve en son olarak "int $0x80" çalıştırılıyor.

```
xorl %eax, %eax
incl %eax
int $0x80
```

Programımızın sonunda da, programı bitirmek için _exit() sistem çağrısı (sistem çağrı numarası 1) kullanılıyor. Bu DOS altındaki AH=4Ch INT 21h çağrısı ile aynı işi yapmaktadır.

uc.s

Son örneğimizde de yukarıda oluşturduğumuz "dosya" isimli dosyanın adını "yeni_dosya" olarak değiştiriyoruz. Bu işe için yine bir sistem çağrısını kullandık: rename (sistem çağrı numarası 38).

Kodumuz aşağıdaki gibi.

```
.data
   old_path:     .ascii   "./dosya\0"
   new_path:     .ascii   "./dosya_yeni\0"
.text
.globl _start
_start:
                 movl $38, %eax
                 movl $old_path, %ebx
                 xorl $new_path, %ecx
                 int $0x80

                 xorl %eax, %eax
                 incl %eax
                 int $0x80
```

Yukarıda da yine bir önceki örnekte olduğu gibi sistem çağrısı kullanıldı. Bu sefer de sistem çağrı numarasına ek olarak iki argüman gerekti ve ilki EBX ikincisi de ECX içerisine yazıldı. "movl $old_path, %ebx" komutuyla belirtilen "old_path" sabiti derlenme sırasında gerekli offset adresi ile değiştirilmektedir. Aşağıdaki GDB çıktısı bunu daha kolay anlamanızı sağlayacaktır.

```
fnoyan@tux:~/asm$ gdb -q
(gdb) file uc
Reading symbols from uc...(no debugging symbols found)...done.
(gdb) disas _start
Dump of assembler code for function _start:
0x8048074 <_start>:     mov    $0x26,%eax
0x8048079 <_start+5>:   mov    $0x804908c,%ebx
0x804807e <_start+10>:  xor    $0x8049094,%ecx
0x8048084 <_start+16>:  int    $0x80
0x8048086 <_start+18>:  xor    %eax,%eax
```

```
0x8048088 <_start+20>:  inc    %eax
0x8048089 <_start+21>:  int    $0x80
End of assembler dump.
```

GDB ile görüntülenen kodda komutların soneklerinin olmadığına dikkat ediniz.

İnternet Adresleri :

Dokümanın bu son kısmında assembly ile daha geniş çapta uğraşmak isteyenler için bazı internet sitelerinin adresleri bulunmaktadır.

http://linuxassembly.org	Linux ve diğer UNIX tabanlı sistemler altında assembly dili ve kullanımı hakkında kaynak bulabileceğiniz bir site.
http://www.janw.easynet.be/eng.html	Linux altında assembly dilinin kullanımını anlatan bir başka yazı.
http://www.geocities.com/SiliconValley/Ridge/2544/	Linux altında GCC ile satır içi assembly kullanımı hakkında bilgi bulabileceğiniz bir adres.
ftp://www6.software.ibm.com/software/developer/library/l-ia.pdf	GCC ile satır içi assembly hakkında başka bir doküman.
http://members.lycos.co.uk/kasiasyg/	Adresleri yukarıda verilen bazı dokümanların Türkçe çevirilerinin bulunduğu bir site.
http://www.programmersheaven.com	Sadece assembly değil daha birçok programlama dili hakkında örnek kod ve kaynak bulabileceğiniz geniş içerikli bir site.
http://www.web-sites.co.uk/nasm/	Linux ve MS-Windows işletim sistemleri altında kullanılan bir assembler olan NASM'ın sitesi.
http://win32asm.cjb.net	Windows işletim sistemi altında 32-bit assembly programlamayı öğrenebileceğiniz en iyi sitelerden biri.

Yukarıdakilerin dışında adreslerini bilmediğim ama sizin bir arama motorunda gerekli bilgileri yazarak kolayca erişebileceğiniz bazı faydalı kaynaklar.

The Art Of Assembly Language	
PC Assembly Language	Yazar : Paul.A Carter
Ralf Brown's Interrupt List	Yazar : Ralf Brown