Prepared By IcEN

www.frtrk.org >> IcEN

Bu makalemizde 80x86 gerçek mod komutlarını inceleyeceğiz.

80x86 KOMUT SETİ (Bölüm 1)

X86 tabanlı mikroişlemcilerin icra ettiği makine kodları sabit olmasına rağmen, programlama dillerinin komut ve ifadeleri farklı olabilir. Assembly programlama dilininde diğer programlama dillerinde olduğu gibi bir dizi komutu vardır. Bu komutlar genelde mnemonik'ler (nivmonik diye okunur) şeklindedir. Örneğin LEA mnemoniği Load Effective Adres kelimelerinin kısaltılmış şeklidir. Bu makalemizde x86 Assembly programlama dilinin komutlarını anlatmaya çalışacağım ve bu makalede açıklanan komutları öğrendiğinizde kendi başınıza program yazabilir hale geleceksiniz. Aslında 80386 ve sonrası mikroişlemciler için daha birçok komut mevcuttur ve bu komutlar assembly dilinde program yazma işini kolaylaştırır. Bu komutları ilerleyen makalelerimizde açıklamaya çalışacağım.

80x86 komutları genelde 8 grup altında incelenir.

1) Veri taşıma komutları
   mov, lea, les , push, pop, pushf, popf
2) Dönüştürme komutları
   cbw, cwd, xlat
3) Aritmetic komutlar
   add, inc sub, dec, cmp, neg, mul, imul, div, idiv
4) Mantıksal, kaydırma, çevirme ve bitsel işlemler için komutlar
   and, or, xor, not, shl, shr, rcl, rcr
5) I/O (Giriş/Çıkış) komutları
   in, out
6) Karakter dizi (String) komutları
   movs, stos, lods
7) Program akış kontrol komutları
   jmp, call, ret, Jxx (şartlı dallanma komutları)
8) Diğer komutlar
   clc, stc, cmc

Veri Taşıma Komutları

Veri taşıma komutları bir değeri bir yerden başka bir yere taşımaya yarar. mov, xchg, lds, lea, les, lfs, lgs, lss, push, pusha, pushad, pushf, pushfd, pop, popa, popad, popf, popfd, lahf, ve sahf komutları veri taşıma komutlarıdır.

MOV komutu

Bu komutun kullanım şekilleri aşağıdaki gibidir.

```
mov     reg, reg
mov     mem, reg
mov     reg, mem
mov     mem, immediate data
mov     reg, immediate data
mov     ax/al, mem
mov     mem, ax/al
mov     segreg, mem16
mov     segreg, reg16
mov     mem16, segreg
mov     reg16, segreg
```

MOV komutu assembly dilinde çok kullanılan bir komuttur. Yukarıdaki kullanım şekilleri için İngilizce ifadeler kullanılmıştır, çünkü internetten erişebileceğiniz komut seti referanslarında hep bunlar karşınıza çıkacak. Tüm komutlar için geçerli olan bu İngilizce ifadelerin Türkçe karşılıkları aşağıdaki gibidir.

reg : register à kaydedici,

mem : memory à hafıza (RAM-ROM veya Giriş/Çıkış portları olabilir)

immediate data à acil adresleme ile kullanılan direkt veri

segreg : segment register à segment kaydedicisi

mem16 : memory 16 bit à 16 bitlik hafıza alanı (dw direktifi ile tanımlanan veriler)

reg16 : register 16 bit à 16 bitlik kaydedici (AX, BX .. gibi)

MOV komutunu kullanırken yapamayacağınız iki şey vardır, bunlardan birincisi “mem, mem” tipinde bir kullanımdır. Yani hafızanın bir konumunda diğer bir konumuna doğrudan taşıma yapamazsınız. Bu işlemi yapmak için taşınacak veri önce mikroişlemci kaydedicilerinden birine getirilmelidir.

```
MOV  sayi1, sayi2        ;yanlış kullanım
```

Yukarıdaki gibi bir komut satırı yazarsanız, derleyiciniz hata mesajı verir. Böyle bir işlemi yapmak için genel amaçlı bir kaydediciyi kullanmanız gerekir.

```
MOV      AX, sayi1
MOV      sayi2, AX      ;sayi1 ve sayi2 değişkenlerinin word türünden olduğunu varsayıyoruz.
```

MOV komutu ile yapamayacağınız ikinci şey ise segment kaydedicilerine doğrudan bir veri taşımaktır. Yani acil adresleme modunu segment kaydedicilerine uygulayamazsınız.

```
MOV      DS, 1525h ; bu kullanım hatalıdır.
```

Segment kaydedicilerine bir değer yükleyebilmek için genellikle genel amaçlı kaydedicileri kullanılır. Ayrıca segment kaydedicilerine ancak 16 bitlik boyutunda değerler yüklenebileceğinden genel amaçlı kaydedicilerin 8 bitlik kısımlar değil 16 bitlik kısımları kullanılabilir.

```
MOV      AX, 1525h
MOV      DS, AX
```

Bunların dışında operandların boyutları eşit olmak zorundadır.

```
MOV      AX, toplam  ; burada toplam değişkeninin boyutu kesinlikle word tipinde yani iki byte uzunluğunda olmalıdır.
```

Şayet acil adresleme kullanarak bir veri taşıyorsanız işlemci operandın boyutunu kaydediciye uyarlar.

```
MOV      AX, 15h  ; Bu komut işlenince AX’in içinde 0015h değerini görürsünüz.
```

Dikkat edilemesi gereken diğer bir husus ise hafıza operandlarıdır. Örneğin MOV [BX], 5 gibi bir komut ile hafızaya neyin yükleneceği belli değildir (Burada BX kaydedicisine değil hafızaya taşıma yapıldığına dikkat edin) MOV [BX], 5 gibi bir komutla acaba hafızaya byte boyutunda bir 5 değerimi (05) yoksa word boyutunda bir 5 değerimi (0005) yüklenecek? Bunu kodlarınızda belirtmeniz gerekir. Derleyici bu komut satırına hata verir. Doğru kullanım aşağıdaki gibi olmalıdır.

```
mov     byte ptr [bx], 5
mov     word ptr [bx], 5
mov     dword ptr [bx], 5      (*)
```

(*) 80386 ve sonrası işlemcilerde kullanılabilir

mov byte ptr [bx], 5 satırını açıklayalım. Burada BX=0000 olduğunu varsayalım, böyle bir durumda ds:0000 adresine bir taşıma işlemi gerçekleşeceltir. Ama bu adresten itibaren 05'mi yoksa 0005'mi yoksa 00000005'mi taşınacaktır? İşte bunu ptr operatörü belirler. mov byte ptr [bx], 5 komut satırı için ds:0000 adresine 1 byte'lık bir veri yani 05 taşınır. Şayet operatör byte ptr değilde word ptr olsaydı o zaman ds:0000 ve ds:0001 adreslerine dırasıyla 05 ve 00 değerleri taşınacaktı.

XCHG komutu

xchg (exchange) komutu operandlarındaki değerleri yer değiştirir.

80x86 ailesi için dört değişik kullanım şekli vardır;

```
xchg    reg, mem
xchg    reg, reg
xchg    ax, reg16
xchg    eax, reg32          (*)
```

(*) 80386 ve sonrası işlemcilerde kullanılabilir

LDS, LES, LFS, LGS, ve LSS komutları

Bu komutlar 32 bitlik bir hafıza bölgesindeki değeri bir segment kaydedicisine ve bir genel amaçlı kaydediciye bir defada yükler. Kullanım formatı aşağıdaki gibidir;

```
LxS     hedef, kaynak
```

Bu komutları aşağıdaki gibi kullanabilirsiniz;

```
lds     reg16, mem32
les     reg16, mem32
lfs     reg16, mem32    (*)
lgs     reg16, mem32    (*)
lss     reg16, mem32    (*)
```

(*) 80386 ve sonrası işlemcilerde kullanılabilir

Reg16 genel amaçlı herhangi bir kaydedici olabilir mem32 ise double word boyutunda bir veri olmalıdır, bunu "dd" direktifi ile oluşturabilirsiniz. Daha önce bu komutlardan biri olan LES komutu için "X86 Assembly Dilinde Değişken Bildirimi -1" adlı makalede çok güzel bir örnek vermiştim. Erişmek istediğimiz adresin segment ve ofset bölümlerini birleştirerek bir değişken oluşturuyor sonrada bunu program çalışırken istediğimiz gibi kullanıyorduk.

LEA Komutu

LEA (Load Effective Address – Etkin Adresi Yükle) sadece offset adreslerini hedef operandına yükleyen bir pointer gibi düşünebilirsiniz. Genel kullanım formatı

```
lea    dest, source
```

şeklindedir.

```
lea    reg16, mem
lea    reg32, mem     (*)
```

(*) 80386 ve sonrası işlemcilerde kullanılabilir.

MOV komutunu da LEA komutu yerine kullanabilirsiniz, fakat hangi komutu kullanacağınıza adresleme moduna göre seçmelisiniz. Bazen MOV komutu LEA dan daha hızlı çalışabilir, tüm bu bilgilere herhangi bir intel komut setinden faydalanarak bakabilirsiniz.

Daha önceki makalelerimizde ekrana bir karakter dizisini yazdırmıştık, bunun için kaynak kodumuzda ekrana yazdılıralacak olan veriyi aşağıdaki gibi tanımlamıştık;

```
Dizi    DB "Merhaba Assembly",0Ah,0Dh,24h
```

Daha sonra bu dizinin adresini DX kaydedicisine yüklememiz gerektiğinde aşağıdaki komutu kullanmıştık;

```
MOV     DX,OFFSET Dizi
```

Bu komutun yaptığı işi LEA kullanarakta yapabiliriz;

```
LEA     DX, Dizi
```

PUSH ve POP komutları

80x86 push ve pop komutları Stack Memory (Yığın hafıza bölgesi) ile ilgili işlemlerde kullanılır. Yğın hafıza bölgesini sizler .exe tipi program hazırlarken .Stack direktifi ile belirliyorsunuz. İşte bu bölge genellikle programdaki dallanma veya altrutinlerin çalışması sırasında, dönüş adreslerinin ve bayrak kaydedicisinin durumlarını saklamak için kullanılır. Push komutu bu yığın olarak adlandırılan hafıza bölgesine verileri iterken, pop komutuda bu bölgeden veri almada kullanılır.

```
push  reg16
pop   reg16
push  reg32            (**)
pop   reg32            (**)
push  segreg
pop   segreg           (CS hariç)
push  memory
pop   memory
push  immediate_data   (*)
pusha                  (*)
popa                   (*)
pushad                 (**)
popad                  (**)
pushf
popf
pushfd                 (**)
popfd                  (**)
enter imm, imm         (*)
leave                  (*)
```

(*) 80286 ve sonrası işlemcilerde kullanılabilir
(**) 80386 ve sonrası işlemcilerde kullanılabilir

Push ve pop komutları kullanıldığında yığın hafıza bölgesinin işaretçisi olan SP kaydedicisi değişir. Tabiî ki bu yığına itilen veya yığından çekilen değerin boyutuna bağlıdır. Bu komutlar 2 veya 4 byte'lık değerler ile işlem yaptığından yığına 2 bytelık bir değer itildiğinde (mesela bu AX kaydedicisinin içeriği olabilir) SP'nin değeri 2 byte azalır. Şayet yığına 4 byte'lık değer itilirse SP'nin değeri 4 azalır. Unutulmaması gereken önemli bir hususta yığın hafıza bölgesine itilen en son değerin çekilecek olan ilk değer olmasıdır. Tabiî ki yığına birden fazla word ya da doubleword itildiyse aralardaki değerler ile işlem yapmak adresleme modlarıyla mümkündür fakat bu SP'de herhangibir değişiklik yapmaz. Yığın hafıza bölgesi ile ilgili unutulmaması gereken üç önemli kural vardır. Segment kaydedicilerinden olan SS yığın hafıza bölgesinin segment adresini gösterir. Yığına bir şeyler itildikçe SP azalır eökildikçe artar. SS:SP her zaman yığının tepesi olarak tabir edilen noktayı gösterir.

LAHF ve SAHF Komutları

Bu komutlar bayrakları AH kaydedicisine yükler veya AH'a yüklenen bayrak kaydedicilerinin durumlarını kayar nokta kaydedicisine (floating point register) yükler. Bu komutlar 8086 zamanından kalma ve günümüzdeki modern assembly programlarında pek kullanılmayan komutlardır.

Genişletme İşlemleri

Bazen byte boyutundaki bir değeri word boyutuna veya word boyutundaki bir değeri doubleword boyutuna genişletmek gerekebilir. Bu gibi durunlarda aşağıdaki komutlar kullanılır.

```
movzx   hedef, kaynak     ;Hedef kaynağın iki katı büyüklüğünde olmalıdır.
movsx   hedef, kaynak     ;Hedef kaynağın iki katı büyüklüğünde olmalıdır.
cbw
cwd
cwde
cdq
bswap   reg32
xlat
```

MOVZX, MOVSX, CBW, CWD, CWDE, ve CDQ Komutları

cbw (convert byte to word) AL kaydedicisinin 1 byte'lık içeriğini AX'e genişletir. Şayet AL'deki değer pozitifse AH'ın tüm bitleri '0' değerini alır. AL'deki değer negatifse AH'ın tüm bitleri '1' olur.

```
cbw
```

cwd (convert word to double word) komutu AX'in değerini DX:AX'e genişletir. CBW komutundaki kurallar bu komut içinde geçerlidir.

```
cwd
```

Bu komut 80386 ve sonrası işlemcilere özeldir. CWD komutunda olduğu gibi word boyutundaki bir değeri double word boyutuna genişletmede kullanılır. CWD AX'i DX:AX'e genişletirken bu komut AX'i EAX'e genişletir.

```
cwde
```

cdq komutu EAX kaydedicisindeki 32 bit'lik değeri EDX:EAX 'e genişletir. Bu komutda 80386 ve sonrası işlemlerde kullanılır.

```
cdq
```

Örnekler:

; AL' deki 8 bitlik değeri 32 bitlik dx:ax'e genişletmek için

```
cbw
cwd
```

; AL' deki 8 bitlik değeri 32 bitlik eax'e genişletmek için

```
cbw
cwde
```

; AL' deki 8 bitlik değeri 64 bitlik edx:eax'e genişletmek için

```
cbw
cwde
cdq
```

movsx komutuda yukarıdaki komutlara benzer iş yapar, kullanım formatları aşağıdaki gibidir.

```
movsx   reg16, mem8
movsx   reg16, reg8
movsx   reg32, mem8
movsx   reg32, reg8
movsx   reg32, mem16
movsx   reg32, reg16
```

Örnekler:

```
movsx   ax, al        ;CBW komutunun yaptığı işi yapar.
movsx   eax, ax        ;CWDE komutunun yaptığı işi yapar.
movsx   eax, al       ;CBW ve CWDE komutlarının birlikte yaptığı işi yapar.
```

movzx komutu movsx komutu gibi kullanılır fakat negatif değerleri genişletmek için kullanılmaz, çünkü movzx komutu genişletme işleminde sadece bitleri '0' yapabilir. Bu komutun sonundaki zx harfleri İngilizcede zero extend yani sıfır ile genişlet gibi bir anlam taşır.

Tüm bu genişletme komutları genellikle aritmetik işlemlerde ve özellikle bölme komutlarında kullanılır.

BSWAP Komutu

Bildiğiniz gibi x86 hafızası little endian yapıya sahiptir, bununla beraber big endian hafızaya sahip bilgisayarlarda çok sayıda mevcuttur. Örneğin Apple'ın Machintosh bilgisayarları big andian hafıza yapısına sahiptir. BSWAP komutu bu farklı hafıza sistemlerine sahip olan bilgisayarlar arasında veri haberleşmesi yapılması için kullanılır. BSWAP operandında belirtilen 32 bitlik kaydedicinin içindeki değeri byte-byte ters çevirir. Düşük değerlikli sekiz biri en yüksek değerlikli bölgeye, 8-15 arasındak bitleri 16-23 arasına, 16-23 arasındaki bitleri 8-15 arasına ve son olarak 24-31 arasındaki bitleride 0-7 arasına yerleştirir.

BU komut sadece 80486 ve sonrası işlemcilerde kullanılabilir. Kullanım formatı aşağıdaki gibidir.

```
bswap   reg32
```

XLAT Komutu

Genellikle tablo olarak tasarlanan dizilere erişmek için kullanılır. AL kaydedicisine tablonun elemanlarından birini yükler. Bu komutu aşağıdaki örneğe bakarak daha iyi anlayabilirsiniz.

```
Tablo DB   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
```

Bu tablonun 11. elemanını AL'ye yüklemek istersek;

```
MOV      AL, 0Ah              ; İndeks değeri (0 dan 0Ah'a kadar 11 tane değer var)
LEA      BX, Tablo            ; BX'e (taban kaydedicisi) Tablonun ofset adresi yükleniyor
XLAT                          ; Tablonun 11. elemanına erişilip ASCII karakterin hex
karşılığı AL'ye yükleniyor (AL=41h)
```

Şimdi yukarıda açıkladığımız komutlardan birkaçını kullanarak bir program yazalım ve daha sonra da inceleyelim. Öncelikle Push ve Pop komutlarını DS kaydedicisinin değerini belirlemek için kullanabilirim. .exe tipi programlarda DS kaydedicisinin değerini belirlemek için bu güne kadar hep aşağıdaki iki satırı kullandık,

```
MOV      AX, @data
MOV      DS, AX
```

Yığın komutlarınıda veri taşımak için kullanabileceğimizden,

```
MOV      AX, @DATA
PUSH     AX
POP      DS
```

Yukarıdaki üç satırda DATA segmentin adresini önce AX kaydedicisine yükledik, sonra bu kaydedicideki değeri yığına ittik son olarakta yığından bu değeri çekip DS kaydedicisine yükledik.

Kullandığım sistemin saat bilgisini ekrana yazdırmak istiyorum. Tabiî ki bu gibi işlemler için hali hazırda DOS'un kesmeleri mevcuttur. Fonksiyon 2Ch sistem zamanı ile ilgili bilgileri işlemcinin kaydedicilerine getirir.

```
MOV      AH,2Ch
INT      21h
```

Bu komutlar işlenince CX ve DX kaydedicileri saat bilgisi ile yüklenirler,

```
CH = saat CL = dakika DH = saniye DL = salise
```

Kaydedicilerin içindeki değerlerin binary olduğunu ve debugger programlarında bu değerlerin hexadecimal formatta göründüğünü unutmayalım. Biz bu programda kullanıcı ekranına desimal formatta değerler yazdıracağımızdan ya çevirme işlemi yapacağız yada tablo oluşturup hex bilgilerin karşılığına tablodan bakacağız.

Çevirme işlemleri için çarpma, bölme gibi komutları bilmeniz gerekir ama henüz o komutlarla ilgili örnekler çözmedik. Bunun yerine biz hex-decimal tablolar oluşturup bu tablolardan desimal değerleri bulalım ve ekrana yazdıralım. Fakat tabloların sınırlarını belirlemek için elimizdeki bilgiyi iyi tanımamız gerekir.

En büyük saat bilgisi 23:59 yani 0 ile 59 elemanlarını kapsayan bir tablo oluşturmam yeterli.

```
          SaatTablo              DB
"00$","01$","02$","03$","04$","05$","06$","07$","08$","09$"
                                 DB
"10$","11$","12$","13$","14$","15$","16$","17$","18$","19$"
                                 DB
```

```
"20$","21$","22$","23$","24$","25$","26$","27$","28$","29$"
                                DB
"30$","31$","32$","33$","34$","35$","36$","37$","38$","39$"
                                DB
"40$","41$","42$","43$","44$","45$","46$","47$","48$","49$"
                                DB
"50$","51$","52$","53$","54$","55$","56$","57$","58$","59$"
```

Yukarıdaki tablo bize 60 elamanlı gibi görünebilir fakat hafızada 60*3=180 byte yer kaplar, çünkü x86 hafızası byte adreslenebilir ve her rakam veya karakter hafızada 1 byte'lık yer kaplar.
Diyelim ki CH'taki saat bilgisi 0Ah, bu saat 10 demek. Tabloda "10$" olan kısım acaba tablonun hangi adresidir? Sayacak olursanız 30. elemanın 1 31. elemanın 0 yani 30 elemandan itibaren 10 değerinin mevcut olduğunu görürsünüz. Tabloda her saat değeri için 3 byte'lık değer ayrılmıştır ve bu programda CH veya CL deki değerler yardımıyla indeks adresi hesaplanırken 3 ile çarpmak gerekir.

CH'ta saat 10'u temsilen 0Ah değeri bulunuyorsa 0Ah*3 = 1Eh = 30 hesabı yapılmalıdır. Tüm bu işlemleri yapıp ekrana yazdırılıcak değeride DX kaydedicisine yüklemeliyiz. Çünkü ekranda bir karakter dizisini yazdırmak için INT 21h'ın 9. fonksiyonu karakter dizisinin başladığı adresi DX kaydedicisinde bulunmasını ister. Tüm bu işlemleri yapan kod satırları aşağıdaki gibidir.

```
        MOV     AL, CH              ;Şimdi saat bilgisi,
        MUL     Uc                  ;3 ile çarpılarak tablodaki desimal karşılığı bulundu.
        MOV     DI, AX              ;Bu adres indeks olarak düşünüldüğünden DI ya
yüklendi.
        LEA     DX, SaatTablo[DI]   ;Ve BX'e saat bilgisinin desimal karşılığı olan tablo
konumu yüklendi.
```

MUL komutunu önümüzdeki makalelerde inceleyeceğiz, çok fazla kullanım formatı mevcuttur. Bizim için burada, sadece CH veya CL deki değerleri 3 ile çarpsın yeter. Fakat MUL komutu sadece AL deki değeri bir değişken yada AL kaydedicisi ile çarpabildiğinden önce CH yada CL deki bilgileri AL'ye taşımamız gerekiyor. Daha sonra 3 değeri ile çarpıp tablodan adresi buluyoruz. Tabiî ki Uc bir değişken ve data segmentinde tanımlı olmalıdır.

```
        Uc      DB      3
```

Al deki değer 3 ile çarpıldığında sonuç 16 bitlik AX kaydedicisinde saklanır. Bu noktadan sonra bu adresi DX kaydedisine yükleyip ekranda string yazdırma kesmesini çağırırsak işlem tamam olacaktı ama, SaatTablo taban adresine göre hesapladığımız indeks adresi halen AX kaydedicisinin içinde ve LEA DX, SaatTablo[AX] gibi bir komut satırı kullanamam çünkü böyle bir adresleme modu mevcut değil. Indeks olarak kullanılacak adresler 8086 assembly dilinde SI, DI veya BX kaydedicilerinin birinde olmalıdır, bu yüzden bizde hesapladığımız indeks değerini DI ya yükledik. Son olarak ekranda bir string yazdıracağımızdan DX kaydedicisinin içine yazdıracağımız stringin başlangıç adresini atarak ve daha önce de kullandığımız INT 21h fonksiyon 9h ı kullanabiliriz.
Bununla beraber ":" karakterinide saat bilgisini yazdırırken saat ile dakika arasına yerleştirelim, tabiî ki bu bilgide data segmentinde bir adreste saklı kalsın.

```
        Ayirac      DB      ':'
```

Birde saat kelimesini data segmentte tanımlayalım,

```
        Saat        DB      'Saat $'
```

Şimdi kabataslak ne yapacağım belli oldu, programımın algoritmasını da yazdıktan sonra kodlarımı yazmaya başlayabilirim.

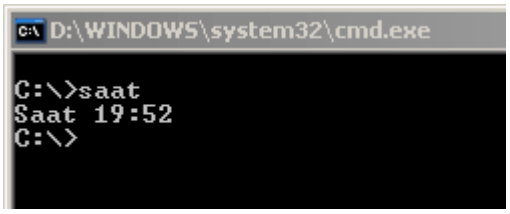
1- Ekrana "Saat" yazdır
2- Sistem saatini ilgili kaydedicilere yükle
3- Tablodan saat bilgisinin desimal karşılığını bul
4- Ekrana yazdır
5- : karakterini ekrana yazdır
6- Sistem dakikasının karşılığını bul

7- Ekrana yazdır

8- Dur

```
        .MODEL SMALL
        .STACK 32
        .DATA

        SaatTablo       DB      "00$","01$","02$","03$","04$","05$","06$","07$","08$","09$"
                        DB      "10$","11$","12$","13$","14$","15$","16$","17$","18$","19$"
                        DB      "20$","21$","22$","23$","24$","25$","26$","27$","28$","29$"
                        DB      "30$","31$","32$","33$","34$","35$","36$","37$","38$","39$"
                        DB      "40$","41$","42$","43$","44$","45$","46$","47$","48$","49$"
                        DB      "50$","51$","52$","53$","54$","55$","56$","57$","58$","59$"

        Ayirac          DB      ":$"
        Saat            DB      "Saat $"
        Uc              DB      3

        .CODE

ANA PROC

        MOV     AX,@DATA        ;Data segment
        PUSH    AX
        POP     DS              ;ayarlanıyor.

        MOV     AH,09h          ;Ekrana,
        LEA     DX,Saat         ;Saat kelimesi,
        INT     21h             ;yazdırılıyor.

        MOV     AH,2Ch          ;CX'e saat bilgisi,
        INT     21h             ;getiriliyor.

        MOV     AL,CH           ;Şimdi saat bilgisi,
        MUL     Uc              ;3 ile çarpılarak tablodaki desimal karşılığı bulundu.
        MOV     DI,AX           ;Bu adres indeks olarak düşünüldüğünden DI ya yüklendi.
        LEA     DX,SaatTablo[DI];Ve BX'e saat bilgisinin desimal karşılığı olan tablo konumu yüklendi.

        MOV     AH,09h          ;Saat bilgisi ekrana,
        INT     21h             ;yazdırılıyor.

        LEA     DX,Ayirac       ; : ayıracı,
        MOV     AH,09h          ;ekrana,
        INT     21h             ;yazdırılıyor.

        MOV     AL,CL           ;Şimdi dakika bilgisi,
        MUL     Uc              ;3 ile çarpılarak tablodaki desimal karşılığı bulundu.
        MOV     DI,AX           ;Bu adres indeks olarak düşünüldüğünden DI ya yüklendi.
        LEA     DX,SaatTablo[DI];Ve BX'e dakika bilgisinin desimal karşılığı olan tablo konumu yüklendi.

        MOV     AH,09h          ;dakika bilgisi ekrana,
        INT     21h             ;yazdırılıyor.

        MOV     AH,4CH          ;DOS'a
        INT     21h             ;dönüş

ANA ENDP
        END ANA
```

Şekil 1 - Saat bilgisini ekranda gösteren program.



Şekil 2 - Programın ekran çıktısı.

Elbette bu program daha kısa veya pratik bir şekilde yapılabilirdi. Bunun için komut bilgimizi genişletmeliyiz. Ayrıca değişken bildirimlerini de şu ana kadar tam olarak anlatmadım. Bu konularla ilgili makalelerimiz yolda fakat şu anda sizlerin mevcut bilgilerinizle bu programa birde salise kısmını ekleyebilmeniz gerekir.

Bu programa ait assembly kaynak kodlarını buradan, kodların derlenmiş halini buradan download edebilirsiniz. Bir sonraki makalede görüşmek üzere.

Bu makalemizde 80x86 komut kümesinin bir kısmını daha inceleyeceğiz.

80x86 KOMUT SETİ (Bölüm 2)

Aritmetik ve mantık (lojik) işlemler mikroişlemcinin ALU (Arithmetic Lojic Unit) denen kısmında yapılır. ALU bir dizi elektronik toplama, çıkarma ve mantık devrelerinden oluşmuştur. Bu devrelerin çalışma mantıkları ise sayma temelinden geçer. Bizlerde ilkokul sıralarında temel işlemleri parmaklarımızla sayarak yapardık. 3 ile 5'i toplarken 3'ün üzerine 5 tane parmak sayardık. Mikroişlemcide her saykılda (saat darbesinde) ALU'da bir sayma işlemi yapar. Bu saat darbesi ne kadar hızlı olursa işlemler o kadar hızlı gerçekleşir. Örneğin 1 GHz. hızında bir işlemci saniyede 1 milyar elektronik darbe üretebilir ve bu saniyede milyonlarca işlem yapabileceği anlamına gelir.

Aritmetik komutların genel kullanım formatları aşağıdaki gibidir. Bu komutları kullanırken de adresleme modlarına dikkat etmemiz gerektiğini unutmayalım.

ADD ve ADC komutları:

Toplama ve elde ile toplama komutlarıdır. ADD komutu işlemci durum kaydedicisinin C bitini hesaba katmazken ADC toplama işlemini C bitinide dahil ederek yapar.

```
MOV AX, 5
ADD AX, 6
```

Bu işlemden sonra AX kaydedicisinde 11'in karşılığı olan 000Bh değeri görülür.

```
MOV AX, 5
ADC AX, 6
```

Bu işlemden sonra şayet C=0 ise sonuç 000Bh C=1 ise sonuç 000Ch olacaktır.

x := y + z + t işlemini;

```
MOV AX, Y
ADD AX, Z
ADD AX, T
MOV X, AX
```

şeklinde yapabilirsiniz. Tabiki bu x,y,z,t'ler birer hafıza konumu veya kaydedici olabilir.

x := x + z işlemini düşünelim. x ve z hafızadaki birer değer olsun yani değişkenlerimiz. Bunu en hızlı şekilde işlemciye nasıl hesaplatabiliriz?

1.yol

```
MOV AX, X
MOV BX, Z
ADD AX, BX
MOV X, AX
```

Yukarıdaki şekilde bu işlemi yapabilrsiniz ama bu çokta iyi bir yol değildir.

2.yol

```
MOV AX, X
ADD AX, Z
MOV X, AX
```

Bu yol daha iyi gibi görünsede bundan daha iyi çalışacak kodlar aşağıdaki gibidir.

3. yol

```
MOV AX, Z
ADD X, AX
```

Adresleme modlarını akıllı bir şekilde kullanabilirseniz çok hızlı çalışan programlar hazırlayabilirsiniz. Yukarıdaki üç program parçası aynı işi yapmasına rağmen en hızlı çalışanı 3. südür. Günümüzde kullanıcıya daha yakın ve program yazması daha kolay olan üst seviye programlama dillerine göre assembly dilinin en büyük avantajı budur.

ADD ve ADC komutları işlemcinin bayrak kaydedicindeki bitlere şöyle etki ederler.

1. İşaretli sayılarla işlem yaparken işaret taşmalarını göstermesi amacıyla V (overflow) bitine. Çünkü bazen negatif bir sayı ile negatif başka bir sayı toplanır ve pozitif bir sonuç elde edilebilir.

2. İşaretsiz sayılarda işlem yaparken boyut taşmalarını göstermesi amacıyla C bitini. Örneğin 2 byte'lık bir değişken olan FFFFh (65535) ile 1 i toplarsınız sonuçta 0 elde edersiniz. Sonucun alana sığmadığı bu gibi durumlarda programcıyı haberdar etmesi amacıyla C=1 olur.

3. Şayet sonuç negatif bir değerse S=1 değilse S=0 olur.

4. 0 sonucu bir çok yerde önemli olabilir. Bu yüzden toplama komutları ile işlem yapıldığında sonuç 0 olursa Z=1 aksi halde Z=0 olur.

5. Binary değerler ile desimal sayıları kodlamak gerekebilir (BCD). Bu durumda düşük değerlikli 4 bitte taşma olursa bayrak kaydedicisinin A biti 1 olur. Örneğin 0000 1111b değerine 1 eklendiğinde sonuç 0001 0000b olacaktır ve düşük değerlikli 4 bitteki en büyük sayı olan 1111 birden 0 a düşecektir. Bu yüzden A=1 olur ve işlemci programcıyı bu durumdan haberdar eder.

6. Elde edilen sonuçtaki binary 1 ler çift sayıda olursa, örneğin 0000 0101 değerinde 2 tane 1 vardır bu durumda P=1 olur.

INC Komutu:

ADD X, 1 gibi çalışır. X kaydedici veya hafıza alanı olabilir. Kısaca hedefi 1 arttırır. Döngülerde

çok kullanılan bir komuttur. Bu yüzden çok önemlidir. INC komutunun 1,2 veya 4 bytelık operandı olabilir. Yani bu komutu aşağıdaki formatlarda kullanabilirsiniz.

```
INC AL              ; 1 bytelık kaydedici
INC AX              ; 2 bytelık kaydedici
INC EAX             ; 4 bytelık kaydedici
INC HAFIZAADRESI    ; Byte word veya doubleword boyutundaki değişkenler olabilir.
```

INC komutu genelde ADD mem,1 veya ADD reg,1 formatına tercih edilir çünkü daha hızlıdır, buna rağmen peşpeşe 1 den fazla inc komutu kullanmak gerekirse komut setinin incelenmesinde fayda vardır. Çünkü bu işi ADD reg,2 veya ADD mem,2 şeklindede yapabilirsiniz ve bu durumda sadece 1 adet komut satırı yazarsınız. Ayrıca INC bayrak kaydedicisinin C bitine etki etmez. Bu yüzden dizi işlemleri için çok uygun bir komuttur. Oysa ADD ve ADC C bitine etki ederler ve büyük dizilerde bu komutlar kullanılırsa dizinin içindeki elemanları işaret etme işleminde bazen yanlış sonuç gösterebilirler.

XADD Komutu:

80486 ve sonrası işlemciler için geçerli bir toplama komutudur. Bu komutu aşağıdaki örnek ile daha iyi anlayacağınız kanaatindeyim.

```
; AX = 07h ve BX = 03h olsun

XADD AX, BX         ; Bu komuttan sonra

; AX = 0Ah yeni toplama işleminin sonucunu gösterir,
; ve BX = 07h olur, yani AX'teki kaybolan operand buraya taşınır.
```

Çıkartma i**ş**lemini yapan komutlar:

SUB ve SBB komutlar**ı**:

SUB (Subtract) yani çıkartma SBB ise borç ile çıkart (SuBtract with Borrow) anlamına gelir. Her iki çıkartma işlemi bir çıkartma sonucu üretmenin yanında bayrak kaydedicisinin C bitinide etkilerler. Bu komutların genel kullanım formatları aşağıdaki gibidir;

```
sub reg, reg
sub reg, mem
sub mem, reg
sub reg, immediate data
sub mem, immediate data
sub eax/ax/al, immediate data
```

Bu komutların ne yaptığını örnekler ile daha iyi anlayabiliriz;

```
MOV AX, 0Ah
MOV BX, 04h
SUB AX, BX
```

Yukarıdaki komutlar ile işlemci 0Ah-04h işlemini yapar ve 6 sonucunu AX kaydedicisinin yani hedef kaydedicide saklar. Bu işlemde büyük değerden küçük değer çıkartıldığından C bitinin durumunda bir değişiklik olmaz.

```
MOV AX, 04h
MOV BX, 0Ah
SUB AX, BX
```

Yukarıdaki komutlar işlenince 04h-0Ah işlemi yapılır ve 2 byte'lık AX kaydedicisinin içinde FFFAh sonucu görülür. Bu işlemde ayrıca C biti set edilir ve C=1 olur. Programcı C'nin bu durumunu göz önünde bulundurmalıdır. Çünkü sonucu işaretsiz bir tamsayı gibi değerlendirirse yanılır. Böyle bir sonuç elde edildiğinde sonucun tümleyeni alını ve 1 eklenir.

1111 1111 1111 1010 ; FFFAh'ın binary karşılığı
0000 0000 0000 0101 ; tümleyeni
0000 0000 0000 0110 ; 1 fazlası yani 6

Ayrıca bu sonuç incelenirken, kaydedicideki FFFAh değerinin 15. biti 1 olduğundan sonuç negatif olarak değerlendirilmeli ve yukarıdaki işlem yapılarak sonucun gerçek değeri hesaplanınca;

-6 değerine ulaşabilirsiniz.

Aslında FFFAh sonucunun sağlamasını yaparsanız, yani çıkana bu sonucu eklerseniz;

FFFAh
000Ah

0004h

yukarıdaki 4h sonucuna erişirsiniz.

SUB komutunun kullanımını SUB hedef, kaynak şeklinde genellersek;

hedef = hedef - kaynak;

SBB komutu ise;

hedef = hedef - kaynak - C işlemlerini yapar.

Çıkartma komutları toplama komutlarında da olduğu gibi bayrak kaydedicisinin, Z, S, V, A, P ve C bitini etkilerler. Tabiki bu etkilenen bayraklar yapılan işleme göre programcı tarafından değerlendirilmelidir.

Çıkartma Aslında Toplamadır!

3 - 4 aslında 3 + (-4) değil midir? Bu tür basit bilgileri unutmamak bazen sizin işinizi kolaylaştırabilir. Aşağıdaki örneği inceleyelim.

x = x - y -z işlemini yapmak için;

```
MOV AX, X
SUB AX, Y ; x-y işlemi yapılıyor, sonucu AX'e yükleniyor.
SUB AX, Z ; x - y - z işlmi yapılmış oluyor
MOV X, AX ; sonuç x'e yüklenerek x = x - y - z işlemi yapılmış oluyor.
```

Fakat bu işlem aslında x = x - (y + z) değilmidir?

```
MOV AX, Y
ADD AX, Z ; y - z işlemi yapılıyor
SUB X, AX ; x - y - z işlemi yapılıp sonuç x'e yükleniyor.
```

DEC komutu:

Decrement yani azalt anlamına gelir. hedef operandını 1 eksiltir, başka bir deyişle -1 ekler. Kullanım formatları aşağıdaki gibidir.

```
DEC reg
DEC mem
DEC reg16
```

C biti hariç çıkartma komutların etkilediği bayrakları etkileyen bir komuttur. INC komutu gibi genelde döngülerde her iterasyondan sonra sayacı azaltmak için kullanılır.

CMP komutu:

SUB komutu ile aynı işi yapar fakat çıkarma işleminin sonucunu herhangi bir kaydediciye yüklemez. Bu komut genelde şartlı dallanma komutlarından önce bayrakları etkilemek için kullanılır. CMP'nin anlamı "compare" yani karşılaştır demektir. Bakın neleri karşılaştırabiliyoruz;

genel kullanım formatları,

```
cmp    reg, reg
cmp    reg, mem
cmp    mem, reg
cmp    reg, immediate data
cmp    mem, immediate data
cmp    eax/ax/al, immediate data
```

Bu komut A, C, O, P, S ve Z bayraklarını etkiler. Programcı etkilenen bu bayrakları göreceli olarak yorumlayabilir, Şöyle ki;

A ara elde biti yani işlem yapılırken 3. bite gelindiğinde eldenin olup olmadığı hakkında bilgi verir ve P işlem sonucundaki değeri binary olarak düşündüğümüzde 1'ler tekmi yoksa çift mi durumunu gösterir. A ve P bayraklarından ziyade programcılar Z, C, O ve S bitlerinin durumları ile ilgilenirler. Bu bayrakları değerlendirirken de işlemlerin işaretli yada işaretsiz sayılar ile yapıldığının bilinmesi büyük önem taşır.

1- Z bayrağı sayılar ister işaretli ister işaretsiz olsun eşitlik yada eşit olmama durumunu gösterir.

```
mov ax,5
mov bx,5
CMP ax,bx ; Z=1 yani operandlar eşit.
```

2- C bayrağı işaretsiz sayılarda;

C=1 ise 2. operand 1.operand dan büyük demektir. C=0 ise 1. operand büyüktür.

C bayrağının işaretli sayılarda bize verdiği sonuçların bir anlamı yoktur.

3- S ve O bayrakları işaretsiz sayılarda anlamsız olurken işaretli sayılarda 2 değişik durumu gösterirler. Bunlar;

a- S=0, O=1 veya S=1, O=0 ise 2. operand 1. operand'tan büyüktür.

b- S=0, O=0 veya S=1, O=1 ise 1. operand 2. operand'tan büyüktür.

Şartlı Dallanma Komutları:

İngilizce karşılığı Conditional Jump Instructions'dır. Bu tür komutlar işlendikten sonra program ya normal akışına yani komutları satır-satır işlemeye devam eder ya da normal akışından sapıp başka bir adresteki komutu işler. Karar alma mekanizmaları bu komutlar ile yapıldığından çok önemli komutlar olduğunu sanırım tahmin edebilirsiniz.

Şartlı dallanma komutlarının ilk harfi J ile başlar ve takip eden 1,2 yada 3 harf şartı gösterir. Bu tür komutları bundan sonra JXXX komutları olarak kullanacağım.

JXXX komutlarının da CMP komutları gibi işaretli ve işaretsiz değerler için farklı anlamları vardır. Tüm bu anlamlar ve komutları aşağıdaki 3 tabloda özetleyebiliriz.

Bayrakların Durumunu Test Etmek İçin Jxxx Komutları

Komut	Açıklama	Şart	Eş Komut	Karşıt Komut
JC	Jump if carry *(carry (taşıma) bayrağı 1 ise)*	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry *(carry (taşıma) bayrağı 0 ise)*	Carry = 0	JNB, JAE	JC
JZ	Jump if zero *(zero (sıfır) bayrağı 1 ise)*	Zero = 1	JE	JNZ
JNZ	Jump if not zero *(zero (sıfır) bayrağı 0 ise)*	Zero = 0	JNE	JZ
JS	Jump if sign *(sign (işaret) bayrağı 1 ise)*	Sign = 1	-	JNS
JNS	Jump if no sign *(sign (işaret) bayrağı 1 ise)*	Sign = 0	-	JS
JO	Jump if overflow *(overflow (taşma) bayrağı 1 ise)*	Ovrflw=1	-	JNO
JNO	Jump if no Overflow *(overflow (taşma) bayrağı 0 ise)*	Ovrflw=0	-	JO
JP	Jump if parity *(parity (eşlik) bayrağı 1 ise)*	Parity = 1	JPE	JNP
JPE	Jump if parity even *(sonuçtaki 1'ler çift ise)*	Parity = 1	JP	JPO
JNP	Jump if no parity *(parity (eşlik) bayrağı 0 ise)*	Parity = 0	JPO	JP
JPO	Jump if parity odd *(sonuçtaki 1'ler tek ise)*	Parity = 0	JNP	JPE

İşaretsiz Sayılarda Jxxx Komutları

Komut	Açıklama	Şart	Eş Komut	Karşıt Komut
JA	Jump if above (>) *(Yukarısındaysa)*	Carry=0, Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not <=) *(Aşağısında değil yada eşit değilse)*	Carry=0, Zero=0	JA	JBE
JAE	Jump if above or equal (>=) *(Yukarısında veya eşitse)*	Carry = 0	JNC, JNB	JNAE
JNB	Jump if not below (not <) *(Aşağısında değilse)*	Carry = 0	JNC, JAE	JB
JB	Jump if below (<) *(Aşağısındaysa)*	Carry = 1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not >=) *(Yukarısında değil veya eşit değilse)*	Carry = 1	JC, JB	JAE
JBE	Jump if below or equal (<=) *(Aşağısında veya eşitse)*	Carry = 1 or Zero = 1	JNA	JNBE
JNA	Jump if not above (not >) *(Yukarısında değilse)*	Carry = 1 or Zero = 1	JBE	JA
JE	Jump if equal (=) *(Eşitse)*	Zero = 1	JZ	JNE
JNE	Jump if not equal (!=) *(Eşit Değilse)*	Zero = 0	JNZ	JE

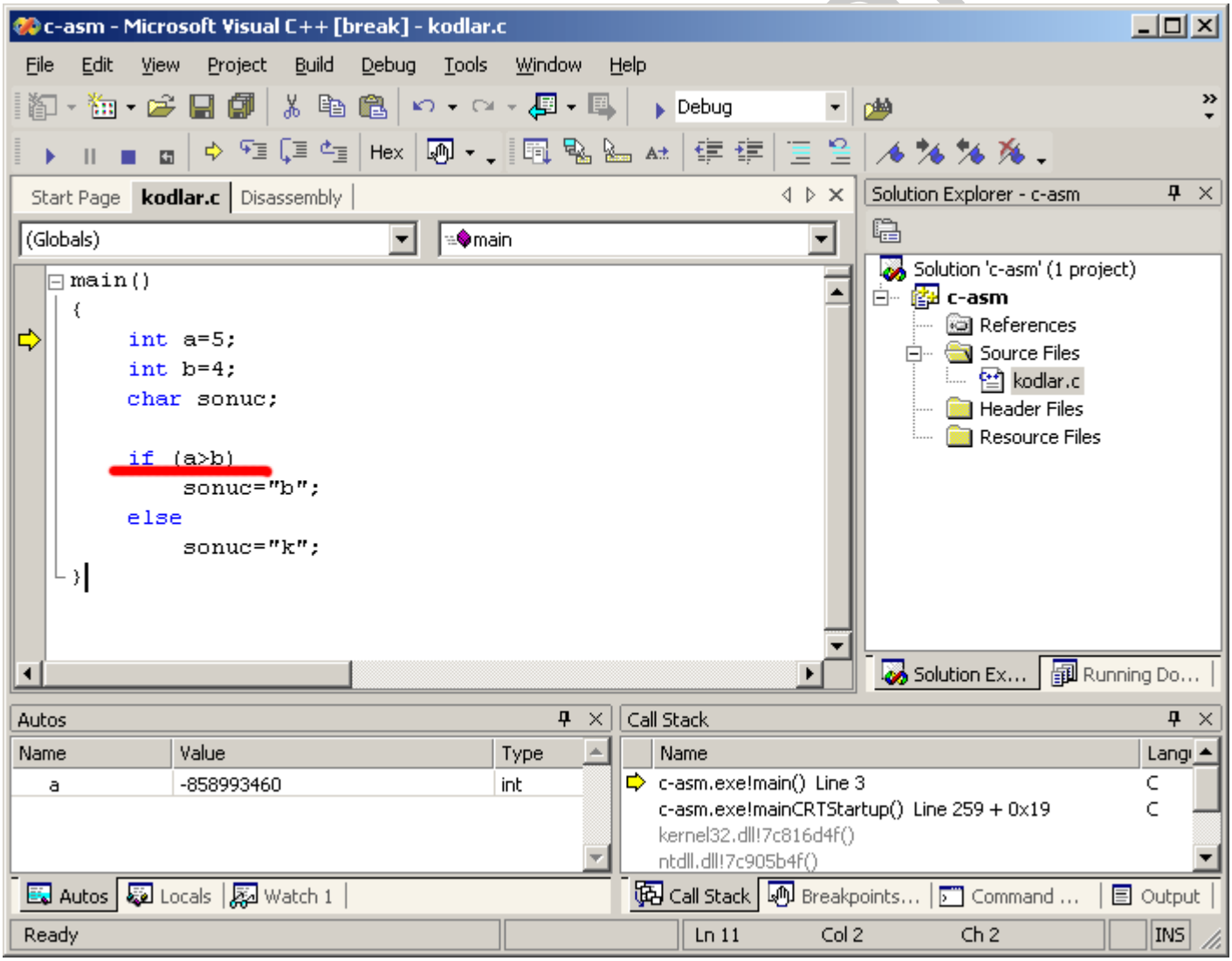
İşaretli Sayılarda Jxxx Komutları

Komut	Açıklama	Şart	Eş Komut	Karşıt Komut
JG	Jump if greater (>) *(Büyükse)*	Sign = Ovrflw or Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=) *(Düşük değilse yada eşit değilse)*	Sign = Ovrflw or Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=) *(Büyükse veya eşitse)*	Sign = Ovrflw	JNL	JGE
JNL	Jump if not less than (not <) *(Düşük değilse)*	Sign = Ovrflw	JGE	JL
JL	Jump if less than (<) *(Düşükse)*	Sign Ovrflw	JNGE	JNL
JNGE	Jump if not greater or equal (not >=)	Sign Ovrflw	JL	JGE

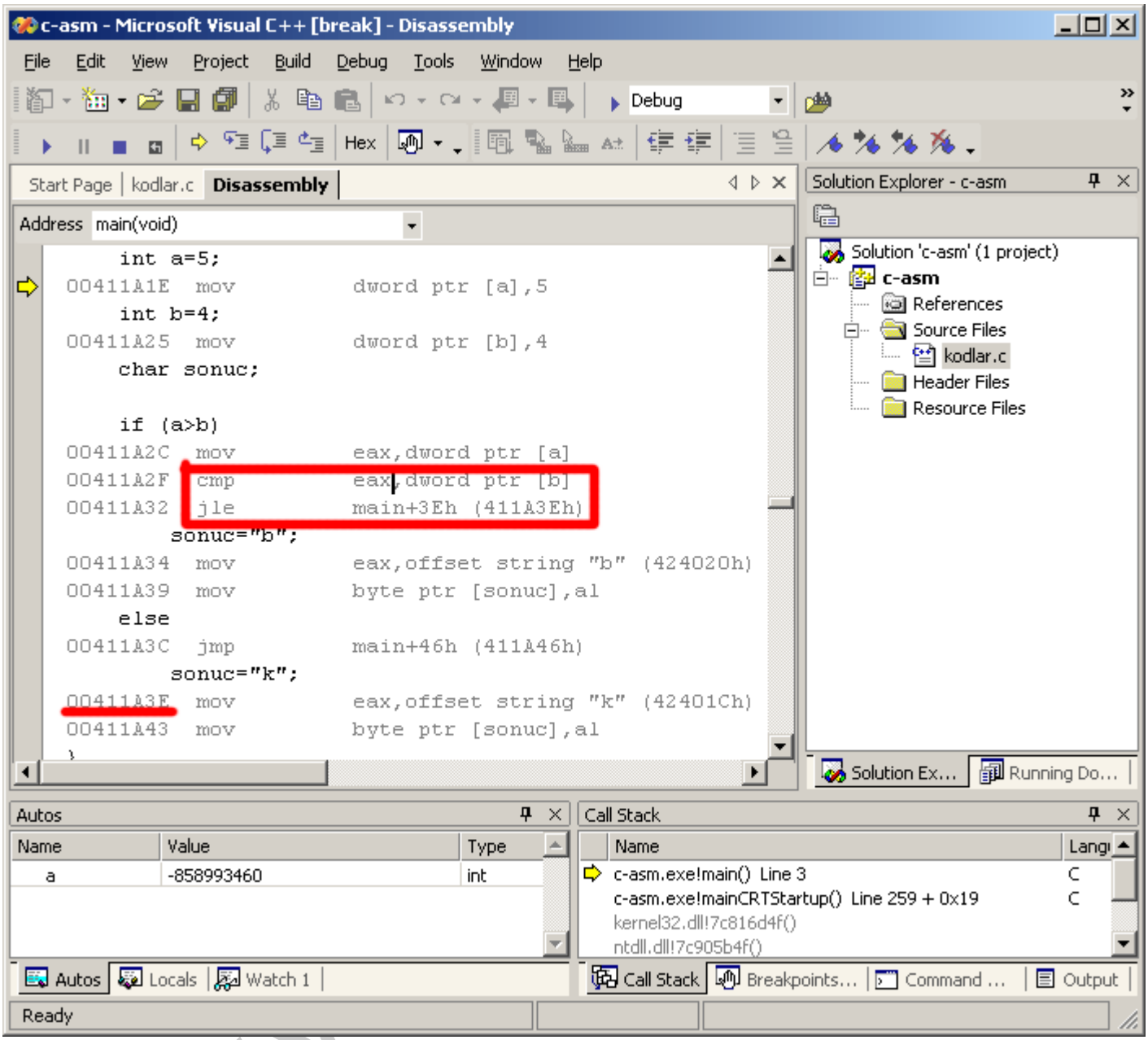
	(Büyük değilse veya eşit değilse)			
JLE	Jump if less than or equal (<=) *(Düşükse veya eşitse)*	Sign Ovrflw or Zero = 1	JNG	JNLE
JNG	Jump if not greater than (not >) *(Büyük değilse)*	Sign Ovrflw or Zero = 1	JLE	JG
JE	Jump if equal (=) *(Eşitse)*	Zero = 1	JZ	JNE
JNE	Jump if not equal (!=) *(Eşit değilse)*	Zero = 0	JNZ	JE

Tabiki yukarıda kullandığım büyük, küçük, yukarısında veya aşağısında kelimeleri sayıların değerleri açısındandır. Tablolarda aslında aynı işi yapan hatta makine düzeyinde aynı kodu üreten komutlar vardır. Bunlara örnek olarak JA ve JNBE'yi örnek verebiliriz. Assembler her iki komut içinde aynı makine kodunu üretir, fakat programcıyı assembly kodlarını yazarken birazcık olsun rahatlatması için Intel bu tür yazımı uygun görmüştür.

Tüm programlama dillerinde kullanılan if, for, while vb. deyimler aslında şartlı dallanma komutları ile düzenlenmiş bir kaç assembly komutuna benzetilebilir. Hatta işi biraz daha netleştirelim C dilinde kullandığımız "if" deyimi aslında bir adet cmp ve bir adet Jxxx komutundan oluşur, xxx kısmı ise şarta göre değişir. Aşağıdaki örnekle bu olayı daha iyi anlayabilirsiniz.



Şekil 1 - C Dili ile yazılmış bir program.



Şekil 2 - Programın assembly kodlarına baktığımızda if deyimine karşılık CMP ve JLE komutlarını görüyoruz.

1. şekilde C dilinde yazılmış basit bir program görüyorsunuz. Bu programda tamsayı türünden 2 adet değişken (a ve b) ve karakter türünden 1 adet değişken mevcut (sonuc). IF deyimi ile şayet a, b'den büyükse (ki zaten öyle) sonuç değişkenine büyüğü temsilen b karakteri aksi durumda küçüğü temsilen k karakteri yüklenecek. Şimdi burada bizi ilgilendiren C dilinde IF acaba assembly'de neye karşılık geliyor?

Şekil 2 ye baktığınızda bu sorunun cevabını hemen görüyorsunuz. IF aslında CMP ve JLE komutlarının bir kombinasyonuymuş. JLE yerine başka bir şartlı dallanma komutu gelebilirdi, bu tamamen IF ile kullandığınız şarta bağlıdır, bu programda IF (a>b) kullanılmıştı. Assembly karşılığında a'nın değeri olan 5 önce eax kaydedicisine yükleniyor ve sonra hafızadaki b değişkeni değeri ile yani 4 ile karşılaştırılıyor.

JLE satırına dikkat edin, JLE'nin anlamı less or equal yani düşük yada eşitse. Yani a'nın değeri olan 5 b'nin değeri olan 4'ten düşük yada eşitse, program 00411A3E adresinden devam edip, önce eax'e k karakterini yükleyip (daha doğrusu k'nın ASCII kod karşılığını yükleyecek), bunu hafızadaki "sonuc" adlı yere (sonuc da aslında bir adres :)) kaydedecektir. Tabi 5 4'ten büyüktür. Hal böyle olunca bu programda JLE komutu işlendikten sonra bir dallanma söz

konusu değildir. Yani program normal akışını devam ettirecek ve eax'e "b" karakterini ardından da sonuc adlı adrese kopyalayacaktır.

Bu programın assembly kodlarını daha önce görmediğimiz bir biçimde birazcık farklı görmeniz doğaldır. Bunun nedeni Visual Studio.NET'in C derleyicisinin 32 bitlik olduğundandır. Yani bu güne kadar biz 16 bitlik TASM veya MASM'ı kullandık. Bu yüzden eax kaydedicisini göremedik veya adresleri buradaki gibi 32 bitlik değil de hep 16 bitlik offsetler halinde gördük. Bu yüzden assembly kodları biraz farklı. 32 bitlik kodlama bizim için henüz çok erken bir kavram, bu yüzden bundan sonraki makalelerimizde 16 bitlik kodlamaya devam edeceğiz.

Şimdi bu makaleyi okuyan sizlerin akıllarına başka sorularda takılabilir. Şayet böyle bir durum varsa hemen aşağıdaki "yorum yazmak istiyorum" linkine tıklayıp sorularınızı sorabilirsiniz. Tüm sorularınızı memnuniyetle cevaplayacağım. Özellikle şekil 2'ye ait sorularınızı bekliyorum.

Bir sonraki makalemizde 80x86 komutlarını öğrenmeye devam edeceğiz. O zamana kadar her şey gönlünüzce olsun...

Bu makalemde 80x86 komutlarını değişik örneklerle anlatmaya devam edeceğim. Bu makalede toplama ve çıkartma işlemlerine bir nokta koyup çarpma ve bölme işlemlerine bir giriş yapmak istiyorum.

80x86 KOMUT SETİ (Bölüm 3)

NEG komutu

NEG negatif kelimesinin kısaltmasıdır. Tek operandı vardır. Kullanım formatı aşağıdaki gibidir.

```
neg    reg
neg    mem
```

yani operandı herhangi bir kaydedici veya hafıza adresi olabilir. Yaptığı iş operandın değerinin negatifini almaktır. Daha doğru bir deyişle operandını 0'dan çıkartır. Binary düzende düşünecek olursanız 1'lein yerine sıfır 0'ların yerine 1 getirir ve bu sonuca 1 ekler.

```
mov    al,0fh
neg    ax
```

yukarıdaki işlemden sonra AX'in içeriği F1h olacaktır.

```
0Fh    = 0000 1111
tersi  = 1111  0000
1111 0000 +1 = 1111 0001 = F1h
```

Bu komutun ne amaçla kullanılabileceğini makalenin sonlarına doğru anlayacaksınız.

MUL ve IMUL Komutları

MUL çarpma IMUL ise işareti dikkate alarak çarpma işlemlerini yapar. Kullanım formatı aşağıdaki gibidir.

İşaretsiz çarpma:

```
mul    reg
mul    mem
```

İşaretli çarpma:

```
imul    reg
imul    mem
```

```
imul    reg, reg, imm   (*)
imul    reg, mem, imm (*)
imul    reg, imm        (*)
imul    reg, reg        (**)
imul    reg, mem        (**)
```

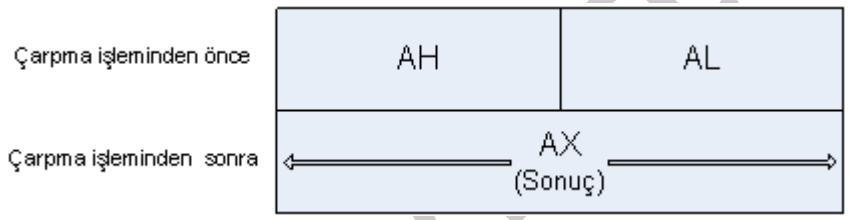
*- Sadece 286 ve sonrası işlemcilerde.
**- Sadece 386 ve sonrası işlemcilerde.

IMUL komutunun 286 ve 386 dan sonraki kullanım formatlarına bakacak olursanız, bu güne kadar gördüğümüz formatlardan biraz farklı olduğunu görürsünüz. Bu kullanım şekilleri programcıların kafasını biraz karıştırmakla beraber tek bir komut satırı ile çabucak çarpma işlemini yapmasını sağlar. Nede olsa Intel karmaşık komut setini benimsemiştir.

Çarpma komutları bayt-bayt, word-word veya Doubleword-Doubleword çarpma yapabilir. Tabi ki Doubleword çarpım için 386 ve sonrası işlemci kullanmanız gerekiyor, çünkü 32 bitlik kaydediciler 386 ile birlikte gelmiştir.

Ayrıca çarpma işlemi toplamadan daha büyük sonuçlar çıkarabilir. Yani 2 basamaklı bir değeri başka bir 2 basamaklı değer ile toplarsanız sonuç en fazla 3 basamaklı çıkarken çarpmada bu 4 basamağa kadar çıkabilir. Daha fazla basamaklı sayıların çarpımında sonuç çarpılan veya çarpandan çok daha fazla basamaklı çıkabilir. Bu gerçeği göz önüne alarak işlemci tasarımcıları sonucu her zaman çarpan ve çarpılanın boyutundan daha büyük bir kaydedicide saklama yoluna gitmişlerdir. Bunları aşağıdaki şekil ve açıklamalarla daha iyi anlayacaksınız.

Byte Çarpma



27h ile 17h'ı çarpmak için;

```
mov    al, 27h
mov    dl, 17h
mul    dl
```

komutlarını kullanabilirsiniz. Burada "mul dl" komutu ile DL*AL işlemi yani bu kaydedicilerdeki değerler olan 17h ve 27h sayıları çarpılır. Peki sonuç nerede? Yukarıdaki şekle baktığınızda sonucun AX içinde olacağını görebilirsiniz. Bu çarpma işleminden sonra AX'te 0381h değeri görülür.

Pratik olarak 8 bitlik bir değerin karesini almak için;

```
mov    al,sayi
mul    al
```

komutlarını kullanabilirsiniz.

Word Çarpma

	DX	AX
Çarpma işleminden önce	İçinde ne olduğu önemli değil	Çarpılan
Çarpma işleminden sonra	Sonuç (Yüksek değerlikli)	Sonuç (Düşük değerlikli)

Bu tür bir çarpma işleminde operand AX ile çarpılır ve sonuç DX-AX kaydedicilerinden okunur. DX'te daha önce ne olduğu önemli değildir çünkü çarpmadan sonra buraya sonucun yüksek değerlikli byte'ı yerleşir. Sonucun düşük değerlikli byte'ı ise AX kaydedicisinde saklanır.

Burada dikkat ederseniz çarpma işlemiyle birlikte AX kaydedicisindeki "çarpılan" da kaybedilecektir. Benim tavsiyem bu tür çarpmalarda çarpan ve çarpılanı birer değişken olarak programınızın data segmentinde tanımlamanızdır. Aşağıdaki örnekleri inceleyin,

100h ile 2345h değerlerini çarpalım;

```
mov    ax, 2345h
mov    bx, 100h
mul    bx
```

bu işlemden sonra DX=0023h ve AX=4500h olur. Yani asıl sonuç olan 234500h değerinin yüksek değerlikli word'u DX'te düşük değerlikli kısmıda AX'te görülür. Fakat çarpılan değer yani 2345h bu işlemden sonra kaybolacaktır. Şayet bu çarpılan değer sizin için önemliyse;

```
carpilan    db    2345h
..
..
..
mov    ax, carpilan
mov    bx,100h
mul    bx
```

böylece 2345h değeri daima "carpılan" ismi ile hafızada korunur. Aynı şeyi tabi ki çarpan için yani 100h değeri içinde yapabilirsiniz.

```
carpilan    db    2345h
carpan      db    100h
..
..
..
mov    ax, carpilan
mov    bx, carpan
mul    bx
```

Double Word Çarpma

Double word boyutundaki verilerin çarpımında da word dekine benzer bir yapı kullanılır. Çarpılan değer EAX kaydedicisine yerleştirilip, MUL veya IMUL komutunun peşinden gelen operand ile bu değer çarpılır. Daha sonra elde edilen sonucun yüksek değerlikli doubleword'u EDX'te düşük değerlikli doubleword'ü ise EAX'te saklanır. Yani sonuç 64 bitliktir.

	EDX	EAX
Çarpma işleminden Önce	İçinde ne olduğu önemli değil	Çarpılan
Çarpma işleminden sonra	Sonuç (Yüksek değerlikli)	Sonuç (Düşük değerlikli)

```
carpilan    dd    12345678h
carpan      dd    34522344h
..
..
.386
..
..
mov     ax, carpilan
mov     bx, carpan
mul     bx
```

Yukarıdaki örnekte sonuç olarak işlemci 03B878D610295FE0h değerini hesaplar. Bu çarpma işleminden sonra EDX=03B878D6h ve EAX=10295FE0h olur.

Buradaki .386 32 bitlik kaydedicileri kullanmak için assembler'a verilen bir direktif (talimat) dır. EDX ve EAX gibi 32 kaydedicilerin 32 bitlik alanlarını kullanmak için nu talimatı vermeniz gerekir. 32 bitlik programlama, 16 bitlik programlama nedir bunlar? Şimdilik sadece 32 bitlik programların 16 bitliklere göre daha avantajlı olduğunu görebilirsiniz. Çünkü 32 bitlik programlama ile kaydedici boyutlarımı 2 katına çıkıyor ve bir kaydedicide hesaplayabileceğimiz değerlerde aynı oranda artıyor, bu işlemi 16 bitlik bir programlama ile de halledebilmemize rağmen 2 katı daha fazla komut yazmamız gerekir.

MUL komutu bayrak kaydedicisinin C ve O bitlerini etkiler. Bu bayraklar beraber değerlendirildiğinde aşağıdaki sonuçlar çıkartılır.

1- Byte boyutundaki bir operand AL ile çarpılırsa sonuç AX'te görülür. AH=0 ise C ve O sıfır olur, aksi durumlarda bu bayraklar set (1) olur.

2- Word çarpmada C ve O sıfır ise DX'te sıfır demektir, aksi durumlarda bu bayraklar set olur.

3- Double word çarpmada ise C ve O sıfır ise EDX'te sıfır demektir, aksi durumlarda bu bayraklar yine set olur.

Yukarıdaki üç durum soldaki sıfırların çarmada bir değeri olmadığından, sonucu optimize etmenize yardımcı olacaktır.

IMUL ile Diğer Çarpma Formatları

IMUL (Integer Multiplication) komutu ile yukarıdaki MUL komutu için verilmiş kalıpları kullanabilirsiniz, bununla beraber IMUL komutuna özel çok operandlı kullanım formatlarıda mevcuttur. Tüm kullanım formatları bu makalenin başında verildiği gibidir fakat kaydedicilerin 8 16 ve 32 bitlik durumları da göz önüne almamız gerekir. Şimdi henüz açıklamadığımız 286 ve 386 sonrası işlemcilerde kullanılabilen komut formatlarını aşağıdaki örneklerle inceleyelim.

```
imul operand1, operand2, imm      ;Genel kullanım formatı

    imul    reg16, reg16, imm8
    imul    reg16, reg16, imm16
```

```
imul    reg16, mem16, imm8
imul    reg16, mem16, imm16
imul    reg16, imm8
imul    reg16, imm6
imul    reg32, reg32, imm8      (*)
imul    reg32, reg32, imm32     (*)
imul    reg32, mem32, imm8      (*)
imul    reg32, mem32, imm32     (*)
imul    reg32, imm8             (*)
imul    reg32, imm32            (*)
```

* Sadece 80386 ve sonrası işlemcilerde kullanılabilir

Yukarıdaki komut formatlarının 3 operandlı olanları

```
operand1 := operand2 x imm
```

ve 2 operandlı olanlarıda

```
operand1 := operand1 x imm
```

şeklinde çalışır. Her zaman son kullanılacak olan operandın "imm" yani sayısal bir değer olduğuna dikkatinizi çekerim.

```
mov    bx, 4        ; BX = 0004h
imul   ax, bx, 3    ; AX = 4 * 3 = 000Ch
```

Bu komutlar ile 8x8 bit çarpım söz konusu değildir, imm8 olarak yukarıda gördüğünüz operand sadece komutunun makine kodunun olmasını sağlar. Ayrıca bu çarpma işlemlerinde sonucun boyutu operandta belirtilen kaydedicilerin boyutuyla aynıdır, yani makalenin başında anlattığımız mul komutu gibi sonuç operandın 2 katı olmaz. Bu durumda sonucun hedef kaydediciye sığmaması durumuna karşı C ve O bitleri birlikte kontrol edilmelidir, bu durumlara Intel'in komut setinden bakabilirsiniz.

Bununla beraber bu formattaki çarpma komutları Z bitini her zaman doğru bir şekilde etkilemeyebilir, şayet sonucun sıfır olup olmadığı sizin için önemli ise ancak sonucu sıfır ile karşılaştırdıktan sonra Z bitini kontrol etmelisiniz. Aynı şekilde sonucun işaretini öğrenmek için işaret bayrağı yerine C ve O bitlerinin sıfır olup olmadığı kontrol edilmelidir.

IMUL komutu için Intel'in 80286 ve sonrası işlemcilere koyduğu bu adresleme biçimleri çok boyutlu diziler ile yapılan işlemleri hatırı sayılır biçimde kolaylaştırmıştır. Bu konuya çok boyutlu dizileri ve karmaşık veri yapılarını anlatırken bir daha değinmeyi düşünüyorum.

DIV ve IDIV Komutları

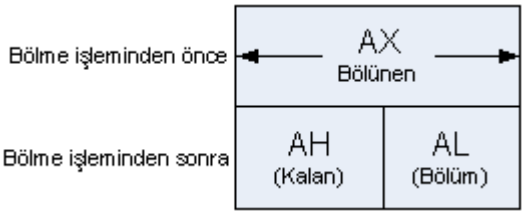
DIV division yani bölme kelimesinin kısaltmasıdır. Bölme işlemi çarpmanın tersine bölünen'e göre küçük sonuç üretir, bu yüzden bu komutları kullanırken bölünen'in boyutu bölen'in boyutunun iki katı olmak zorundadır, an azından Intel bu komutlar için böyle bir form öngörmüştür. Bu yüzden bölünenin boyutu en az word türünde olmalıdır, çünkü x86 Assembly dilinde en küçük veri tipi byte'dır. Bu durumda byte türünden bir değeri bölmek isterseniz bunu CBW komutu ile word boyutuna dönüştürmeniz gerekir, hatırlarsanız bu tür komutları daha önceki makalelerimizde açıklamıştık.

Div ve idiv komutlarının genel kullanım formatları aşağıdaki gibidir.

```
div    reg      ; İşaretsiz çarpma
div    mem
```

```
idiv    reg      ; İşaretli çarpma
idiv    mem
```

Word'u Byte'a Bölmek

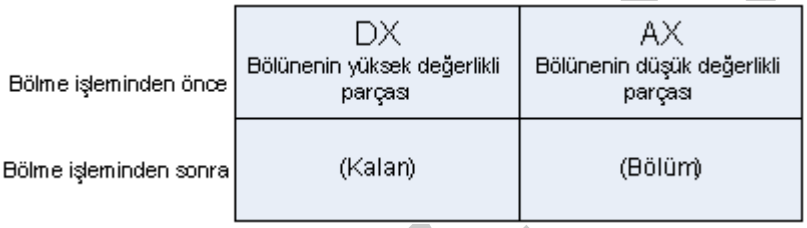


Örnek: 10h:3h işlemini yapmak istiyoruz, bu değerlerin ikisinin de byte türünden olduğunu varsayalım;

```
mov    al, 10h
mov    bl, 03h ; bl yerine başka bir kaydedicide olabilir!
cbw            ; 10h şimdi 0010h ve AX'te
div    bl      ; ax, bl'ye bölündü
```

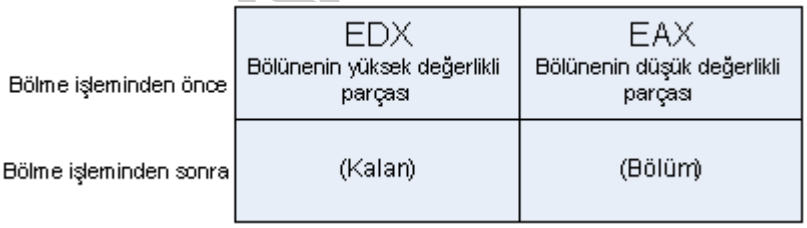
bu işlemden sonra AX, 0105h olur. AH'taki 01h kalan ve AL'deki 05h ise bölümdür. Nitekim 16'nın (yani 10h) 3'e bölümü ile de bu sonuç üretilir. Bu tür bölme işleminde elde edebileceğiniz en büyük bölüm 255 (FFh işaretsiz değerler için) yada 127 (7Fh işaretli değerler için) değerleridir.

Doubleword'u Word'e Bölmek



Bu tür bölme işleminde elde edebileceğiniz en büyük bölüm 32767 (FFFFh işaretsiz değerler için) yada 16383 (7FFFFh işaretli değerler için) değerleridir.

Quadword'u Doubleword'e Bölmek



Peki ya sonuç tam değilse?

Mesela 11h'ı (yani 17'yi) 3'e bölerseniz sonuç normalde 5,666... şeklinde olur, bu işlemi div komutu ile yaptığınızda ise AX'te 0205h değerini görürsünüz, yani bölüm 5 ve kalan 2. Özetle 17'nin içinde 5 tane 3 ve 1 tane'de 2 vardır ve div ve idiv komutları tamsayı bölme işlemlerini gerçekleştirebilir. Ondalıklı bölme işlemleri için floating point kaydedicileri kullanılır ve bu kaydediciler matematik işlemcisinin içindedir. 80486 DX işlemcisine kadar matematik işlemci normal işlemcinin yanına opsiyonel olarak konulurdu mesela işlemci 80386 ise matematik işlemcisi 80387 olurdu. Artık matematik işlemcisi normal işlemcinin içine gömülü olarak geliyor. Bu arada floating point ünitelerini kullanmak için yüksek seviyeli assembly kodları yazmak gerekir ve bu iş bizim için henüz çok erken. Fakat kalanı 10 ile çarpıp sonrada

bölünen ile karşılaştırıp şayet bölünenden büyükse tekrar bölene bölme işlemine gidebilirsiniz, aynı kağıt üzerinde normal bölme işlemi yapar gibi, fakat floating point ünitelerini kullanmak inanın bu işten daha pratiktir ve daha kolay sonuç verir.

Malesef Tüm Sonuçlar Binary

Toplama, çıkartma, çarpma ve bölme komutlarını gördük, artık bu komutları kullanarak basit bir hesap makinesi programı yazmak isteyebilirsiniz, böyle bir programı yazmaya başladığınızda karşınıza sonuçları ekranda desimal formatta göstermek gibi bir problem çıkacaktır. Evet ekrana yazdırmak için daha önce programlar yazdık fakat bunu sadece stringler ile gerçekleştirdik. DB direktifi ile deklare edilen kelime katarları (stringler) hafızada byte-byte ve ardışık olarak saklanır aynı zamanda bu byte'lar elbette harflerin veya sayıların ASCII kod karşılıklarıdır. 8086 komut seti (bölüm 1) başlıklı makalemizde saat programı yapmıştık ve kaydedicilerde elde ettiğimiz sonuçları bir tablo vasıtasıyla ASCII karakterlerini ekranda göstermiştik. Bu yöntem sıkça kullanılmaz, hatta daha önce bu iş için bu yöntemi kullanan bir program görmedim diyebilirim.

Representation yani gösterme veya sunum işlemi çok geniş bir yelpazede incelenebilir, bu tamamen kullanıcının hayal gücüne kalmış bir olaydır. Benim burada anlatmaya çalışacağım olay ise ekran text modunda iken hafızadaki binary ifadelerin ASCII karakter karşılıklarını ekranda göstermek olacaktır.

Diyelim ki bir işlem yaptınız ve sonucunu 20h (32) olarak AL kaydedicisinde saklamayı başardınız ve bunu ekranda göstereceksiniz, bunu direk olarak ekrana yazdırırsanız sadece imleci 1 kez ilerletmiş olursunuz çünkü 20h ascii kod tablosunda space yani boşluk karakterine karşılık gelir, klavyedeki en büyük tuş yani. Peki 32yi nasıl yazdıracağız? Unutmayın ki ascii kod tablosunda sadece rakamların, harflerin ve bir dizi kontrol karakterinin kod karşılıkları vardır sayıların kod karşılıkları yoktur. Bu bağlamda biz 32 yi değil 3 ve 2 yi yan yana ekranda göstermeyi düşünmeliyiz. Ama şu anda AL'de ne 3 ne 2 var sadece 20h var. Diğer bir gerçek 3'un ascii kod karşılığı 33h ve 2 nin ki ise 32h dır. 30h ile 39h arası ascii kod tablosunda rakamlar için ayrılmıştır.

şimdi ekrana sırasıyla 33h'ı ve 32h'ı gönderirsek kullanıcı 32 yi görecektir. Problemi özetleyelim elimizde bir baytlık 20h var ve bizim bunu 2 byte lık 3332h dizisine dönüştürmemiz gerekiyor. Keşke bir komut bu işlemi bizim yerimize yapsa!

Böyle bir komut varmı yokmu oraya geleceğiz ama 20h ile 3332h arasında güzel bir bağ var. 20h ı 0ah yani 10'a bölsek zaten hex'den decimal'e dönüşüm işlemi yapmış oluruz;

```
mov    ax, 20h
mov    bl, 0Ah
div    bl          ;AX=0203 yapar.
```

Keşke AX'teki 0203h değerinde 0'ların yerine 3 gelseydi;

```
or    ax,3030h    ;AX'teki 0203h artık 3233h oldu.
```

birde al ile ah'ı yer değiştirsek! Acaba buna gerçekten gerek var mı? Hatırlarsanız ekrana bir string yazdırmak için DOS kesmelerinden 09h nolu fonksiyonunu kullanmıştık ve bu fonksiyon hafızadaki stringleri yazdırıyordu, yani biz AX'teki bu 3233h değerini önce bir hafızaya atalım sonra 32h ile 33h'ın yerini değiştirmek gerekiyor mu düşünürüz;

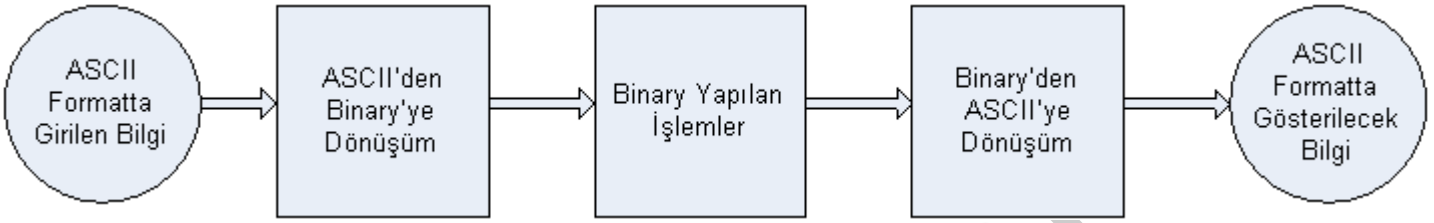
```
mov    sonuc, ax    ; sonuc=3332h olur.
```

x86 tabanlı işlemciler hafızayı adreslerken little endian byte sıralamasını kullanırlar, bu yüzden al'deki düşük değerlikli byte hafızanın düşük numaralı adresine ax'teki yüksek değerlikli byte'ta hafızanın yüksek numaralı adresine yerleşmiş olur, böylece ax ile al'nin içeriklerini takas etmemize de gerek kalmaz. Artık sonuc değişkenini referans göstererek ekrana yazdırma kesmesini kullanabilirsiniz.

```
lea     dx,sonuc
mov     ah,9
int     21h
```

AAA, AAS Komutları

Klavyeden giriş yapıldığında, basılan tuşa ait kod hafızada ascii formatta saklanır, mikroişlemci hesaplamaları binary yapar bu yüzden ascii'den binary'ye dönüşüm yapmak gerekir. Şayet ekrana bir karakter basılacaksa bu karakter ekrana gönderilmeden önce ascii forma dönüştürülmelidir.



Bizler günlük yaşamımızda desimal değerleri kullanırız, bu desimal değerleri hafızada binary rakamlar olan 1 ve 0'ları kullanarak gösterebiliriz, buna BCD (Binary Coded Decimal) kodlama diyoruz. Desimal numaralar hafızada BCD olarak gösterilmenin yanında ASCII olarakta gösterilebilir. İşte AAA, DAA, AAS gibi komutları bunun için kullanıyoruz.

Örneğin klavyeden girilen 1234 hafızada 31 32 33 34 olarak ASCII formda saklanır. BCD gösterimin ise iki farklı çeşidi vardır, bunlar packed BCD (paketlenmiş BCD) ve unpacked (paketlenmemiş) BCD'dir.

1234 hafızada 01 02 03 04 byte dizisi olarak saklanırsa bu paketlenmemiş BCD'dir, şayet aynı değer hafızada 12 34 byte dizisi olarak saklanırsa bu da paketlenmiş BCD'dir. AAA, AAS, AAD ve AAM komutlarının hepsi ascii değerlere dönüşüm için kullanılır. Bu komutların operandı yoktur AL kaydedicisindeki değerleri dönüştürürler.

AAA (Ascii Adjust After Addition) komutu toplama komutundan sonra sonucu ascii ye ayarlar. Aşağıdaki örnekleri inceleyin;

```
  34h = 0011 0100b
  35h = 0011 0101b
+_________________
  69h = 0110 1001b
```

Sonuç 09 olması gerekir AAA komutu 6 değerini siler. Sonuç AL'de 09 olarak görülür.

```
  36 = 0011 0100
  35 = 0011 0101
+_________________
  6B = 0110 1011
```

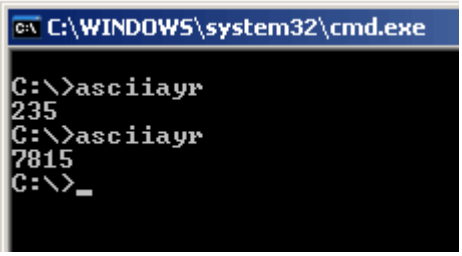
Sonuç 11 olması gerekir AAA komutu B değerinin yerine 1 koyar ve toplama sonucu 9 değerini aştığından AH kaydedicisine de 1 koyar, yani AX = 0101 olur.

Tüm bu işlemlerden sonra OR komutunu kullanarak sonucu ASCII gösterim için hazırlayabiliriz.

```
SUB    AH, AH       ; AH temizleniyor
MOV    AL, 6        ; ilk değer AL'de
ADD    AL, 8        ; AL + 08h işlemi yapıldı sonuç 0Eh
AAA                 ; AX = 0104h
OR     AL,30h       ; AL = 34h
```

34h artık ascii olarak 4 demektir. Tabi burada AH kaydedicisini de kontrol edip şayet 01 ise bu değeri de 30h ile OR işlemine tabii tutmalıyız. Bu tür işlemler için genelde ekrana tek bir karakter basma fonksiyonu kullanılır yani AL'deki değer teker teker ekrana bastırılır. Henüz mantıksal komutları görmedik bu yüzden aşağıdaki programı tam olarak anlamayabilirsiniz, bu yüzden açıklamalara dikkat edin.

```
            .model small
            .stack 32
            .data

            sayi1   DB ?            ; 1. sayı için alan
            sayi2   DB ?            ; 2. sayı için alan
            sonuc   DW ?            ; Sonuç için ayrılan alan

            .CODE

    ANA PROC

            MOV     AX,@DATA
            MOV     DS,AX

            MOV     AH,01h          ; klavyeden bir karakter isteniyor
            INT     21h             ; girilen karakter şimdi AL'de
            MOV     sayi1,AL        ; girilen karakter şimdi hafızada

            MOV     AH,01h          ; klavyeden ikinci karakter isteniyor
            INT     21h             ; girilen karakter şimdi AL'de
            MOV     sayi2,AL        ; girilen karakter şimdi diğer bir hafızada konumunda

            SUB     AH,AH           ; AH temizlendi
            ADD     AL,sayi1        ; sayi1+sayi2 = AL
            AAA                     ; ascii çevrim için AL yada AX pocked BCD
            MOV     sonuc,AX        ; AX hafızada saklandı
            CMP     AH,01h          ; sonuç 9'u aşmışmı?
            JE      SNCWORD         ; aşmışsa 2 karakter yazdırmak için dallan
            OR      AL,30h          ; aşmamışsa AL'yi ascii karakter yap

            MOV     DL,AL           ; 1 byte'lık sonucu yazdırmak için,
            MOV     AH,02h          ; ekrana tek karakter yazdırma,
            INT     21h             ; kesmesini çağır.
            JMP     BITIR           ; Programı sonlandır.
            ;-------------------------------------------------------------------------
SNCWORD:    MOV     AL,AH           ; Önce AH'taki değeri yazdırmak için AL'ye al
            OR      AL,30h          ; ascii koda dönüştür

            MOV     DL,AL           ; bu değeri,
            MOV     AH,02h          ; ekrana,
            INT     21h             ; bas.
            MOV     AX,sonuc        ; hafızadan BCD sonucu tekrar al
            OR      AL,30h          ; düşük değerlikli kısmını ascii yap

            MOV     DL,AL           ; bu değeride,
            MOV     AH,02h          ; ekrana,
            INT     21h             ; bas.

BITIR:      MOV     AH,4Ch
            INT     21h

ANA ENDP
    END ANA
```



Şekil - Makine dilinden insan diline dönüşüm

Pekte güzel bir ekran çıktısı olmasa da bu program dönüşümleri anlamak için iyi bir örnektir. Yukarıdaki şekilde program ilk çakıştırıldığında klavyeden sırayla 2 ve 3 girilmiş ve sonuç 5 olarak ekrana basılmış. Daha sonra 7 ve 8 değerleri girilmiş ve sonuç 15 olarak ekrana basılmış. Gerçektende sonuçlar doğru :) Unutulmaması gereken bir nokta AAA komutunun C bitini etkilediğidir, şayet AAA komutundan önceki bayrakların durumu sizin için önemliyse bayrakların durumunu saklamanız gerekir.

AAS (Ascii Adjust after Subtract) komutuda AAA gibi çalışır, çıkartma komutundan sonra sonucu ASCII'ye ayarlamak için kullanılır. AAS komutu sekiz bitlik AL kaydedicisinin yüksek değerlikli 4 bitini kontrol eder, şayet AF bayrağı 1 ise (başka bir deyişle bu 4 bit Ah...Fh arasında ise) AL'den 6, AH'tan 1 çıkartılır. AH'tan 1 çıkartmak normalde 00 olan AH'ı FF yapmak yani rakamın negatifliğini ayarlamak demektir.

```
mov   al,35h   ; ascii 5
sub   al,31h   ; 5 - 1 = 4
aas            ; AF=0 olduğundan bir değişiklik olmaz sonuç hala 04h
or    al,34h   ; ascii 4
```

sonucun negatif çıktığı bir örnek;

```
mov   al,34h   ;ascii 4
sub   al,38h   ;ascii 8 -- sonuç FCh (negatif)
aas            ; AX = FF06 (yanlış sonuç)
```

yukarıdaki gibi bir durumda SUB komutu ile A ve C bayrakları set (1) olur. Bu durumda sonucu doğrudan 30h ile OR işlemine tabi tutmak hatalı olacaktır. FCh sonucu desimal -4 değerine eşittir bu yüzden burada programcı OR komutu ile doğru ascii değeri ekrana yazdırmadan önce NEG komutu ile tersini alabilir. Bu durumda sonuc 4 olacaktır. Tabiki bu 4 değeri ekrana yazdırılmadan önce önüne - işareti konulmalı.

```
      mov   al,34h
      sub   al,38h
      jnc   devam
      neg   al
devam: aas
      or    al,30h
```

yukarıdaki kod parçası sonuç negatif olsa da pozitif olsa da, sonucun mutlak değerini doğru bir şekilde ascii değere dönüştürür.

Unutmayalım ki klavye bize ekran text moundayken daima ascii değerler verecektir, bu çarpma ve bölme işlemlerinde de problem oluşturur. Klavyeden girilen değerlere sanki 30 eklenmiş gibi geleceğinden çarpma işleminin sonucu girilen değerde bir modifikasyon yapmadığımız sürece yanlış hesaplarız. Bu tür modifikasyonları yapmak için başka mantıksal komutlara ihtiyacımız olacak, bu yüzden bu komutları anlatmadan DAA, DAS, AAD ve AAM gibi komutları açıklamak istemiyorum.

Gelecek makalemde Intel'in komut setini anlatmaya devam edeceğim, bir sonraki makaleye kadar hoşçakalın.

Bu makalemde 80x86 mantıksal (logical) komutlarından bazılarını anlatmaya çalışacağım. Ayrıca VGA 80x25 text mode ekranın hafız alanına direk erişim yapacağız.

80x86 KOMUT SETİ (Bölüm 4)

Mantıksal (logical) komutlar AND, OR, XOR ve NOT adıyla bilinen ve matematiksel hesaplamalarda çok kullanılan komutlardır.

AND komutu

Yapı olarak AND (VE) mantığı 1 ve 0'lar ile ifade edilirse;

1 ve 1 = 1
1 ve 0 = 0
0 ve 1 = 0
0 ve 0 = 0

sonuçlarını üretir.

Bu komutu assembly programcıları genelde maskeleme işlerinde kullanırlar. Örneğin 1 byte'lık değerin 7, 6, 5, ve 4. bitlerini göz ardı etmek için aşağıdaki gibi bir program parçası yazılabilir.

```
mov al, A5h
and al, 0Fh
```

bu işlemlerden sonra AL'in yüksek değerli 4 biti (nibble) sıfırlanacaktır yani AL binary olarak ifade edersek 0000 0101 olacaktır. Buna düşük değerlikli 4 bite dokunmadan diğer bitleri sıfırlamakta denilebilir.

Bu komutu elektronikçiler çok kullanırlar, mesela paralel porttan alınacak olan verinin sadece 5 bitini kontrol etmek için;

```
        mov  dx,378h          ; paralel port adresi
        in     al, dx         ; bu adresten bilgiyi al
        and  al, 0010 0000b   ; 5. biti kontrol et (maskele)
        jnz   devam           ; 5. bit 1 ise "devam" a dallan
        .
        .
        .
        .
devam:   .
        ;devam komutları
        .
```

Diyelim ki AL'ye IN komutu ile alınan byte 1011 0001 olsun, bu durumda;

```
  1011 0001
  0010 0000
v
 ___________
  0010 0000
```

yukarıdaki işlem yapılır ve AND komutu bayrak kaydedicisinin Z bitini 0 olarak kurar, çünkü sonuç 0'dan farklıdır. İşte burada maske olarak sadece 5. biti 1 olan bir byte seçilmiştir.

Not: Windows XP altında paralel port'a doğrudan erişim işletim sisteminin kerneli tarafından engellendiğinden bu programı windows XP öncesi işletim sistemlerinde çalıştırabilirsiniz. IN ve OUT komutlarının kullanımı sonraki makalelerin konusudur.

AND komutunun kullanım formatları;

```
and    hedef, kaynak     ;hedef := hedef & kaynak

and    reg, reg
and    mem, reg
and    reg, mem
and    reg, imm
and    mem, imm
and    eax/ax/al, imm
```

OR komutu

Mantıksal veya işlemini gerçekleştirir,

```
1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0
```

doğruluk tablosu yukarıdaki gibidir. Kullanım formatları AND komutundaki gibidir.

XOR komutu

XOR olmadan şifreleme işlemleri sanırım çok zor yapılırdı. Çok fazla kullanım alanı olmakla beraber veri paketleme ve şifreleme işlemleri için hayat kurtarıcı bir komuttur. eXclusiveOR (özel veya) kelimesinin kısaltmasıdır. Doğruluk tablosu;

```
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0
```

Ben genelde öğrencilerime bu komutun yaptığı işi anlatırken "aynılarda 0 farklılarda 1" sonucunu veren mantıksal ifade derim.

XOR komutu ile şifreleme ve paketleme işlemleri için örnekler şu anda belki de sizin için biraz ağır kaçabilir, bunun yerine swap (yer değiştirme) ve kaydedici sıfırlama örneklerini XOR komutunu ile neler yapılabildiğine örnek teşkil edeceğini düşünüyorum.

```
xor    ax,ax    ; ax=0 olur ve "mov ax,0" dan kat kat hızlı çalışır


;AX = 1234h ve BX=9876h olsun

xor    ax, bx
xor    bx, ax
xor    ax, bx
```

; bu 3 komut sonrasında AX = 9876h ve BX = 1234h olur.

XOR komutunun kullanım formatları AND ve OR komutlarınınkiyle aynıdır.

Bayrakların Durumu

Yukarıda anlattığım bu üç komut bayrak kaydedicisini aşağıdaki gibi etkiler;

Carry bayrağını 0 yaparlar,
Overflow bayrağını 0 yaparlar
Zero bayrağını şayet sonuç 0 ise 1 yaparlar ve aksi durumlarda bu bayrağı 0 yaparlar

En Yüksek değerlikli bit'i Sign bayrağına kopyalarlar
Parity bayrağını (sonuçtaki 1'lerin sayısı çift ise) 1 yaparlar
Auxiliary carry (Ara Elde) bayrağının durumunu değiştirirler.

NOT komutu

NOT komutu operandının mantıksal tersini alır.

Kullanım formatları aşağıdaki gibidir;

```
not reg
not mem
```

; BL kaydedicisinin değeri 15h olsun;

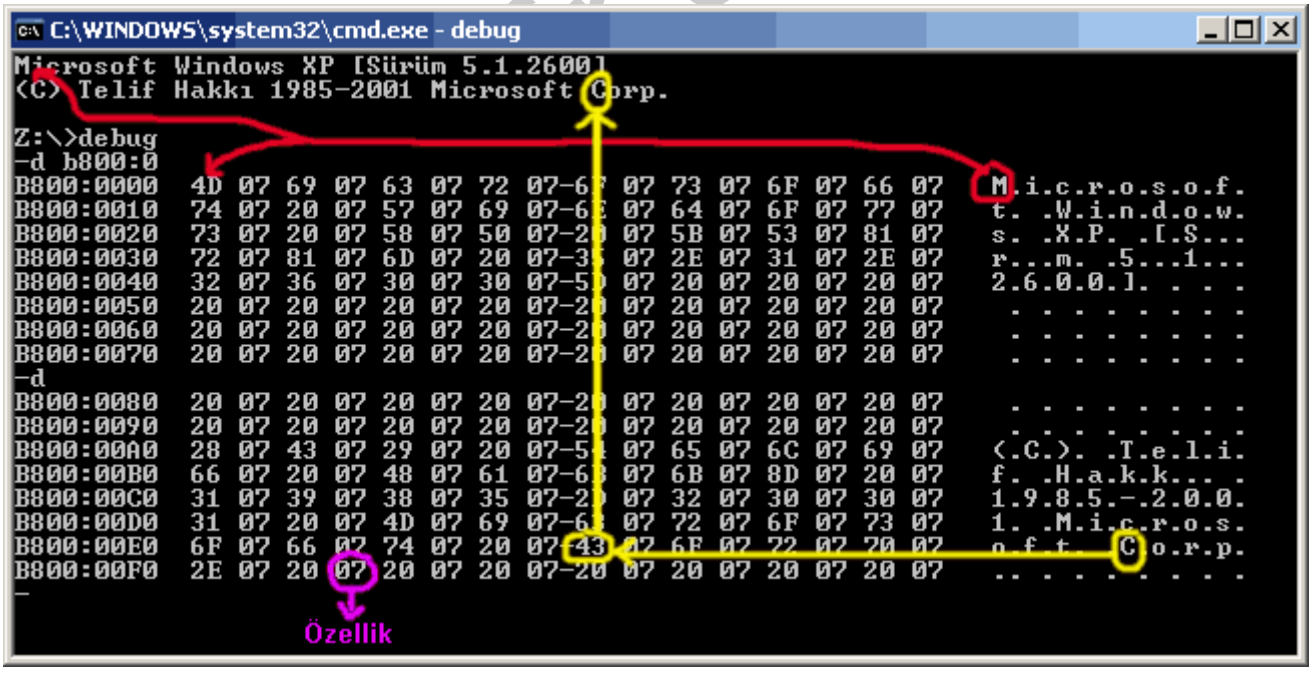
```
not bl
```

;bl = 0001 0101b iken bu komuttan sonra 1110 1010b olur.

NOT komutu hiçbir bayrağı etkilemez.

Şimdi Uygulama Zamanı

Bu makalemde sizlere örnek teşkil etmesi için konsol ekranındaki görüntüyü (tüm yazıları) önce karma karışık bir hale getiren daha sonrada tekrar orijinal haline çeviren bir programı anlatmayı düşünüyorum. Öncelikle Windows'un cmd.exe yada komut istemi ile açılan pencerenin görüntüsünün aslında B800:0000 adresinin bir yansıması olduğunu hatırlatmak isterim. Debug ile bu alana göz atalım.



Şekil 1 - Aslında ekran yansıması da hafızadaki byte'lardan ibarettir.

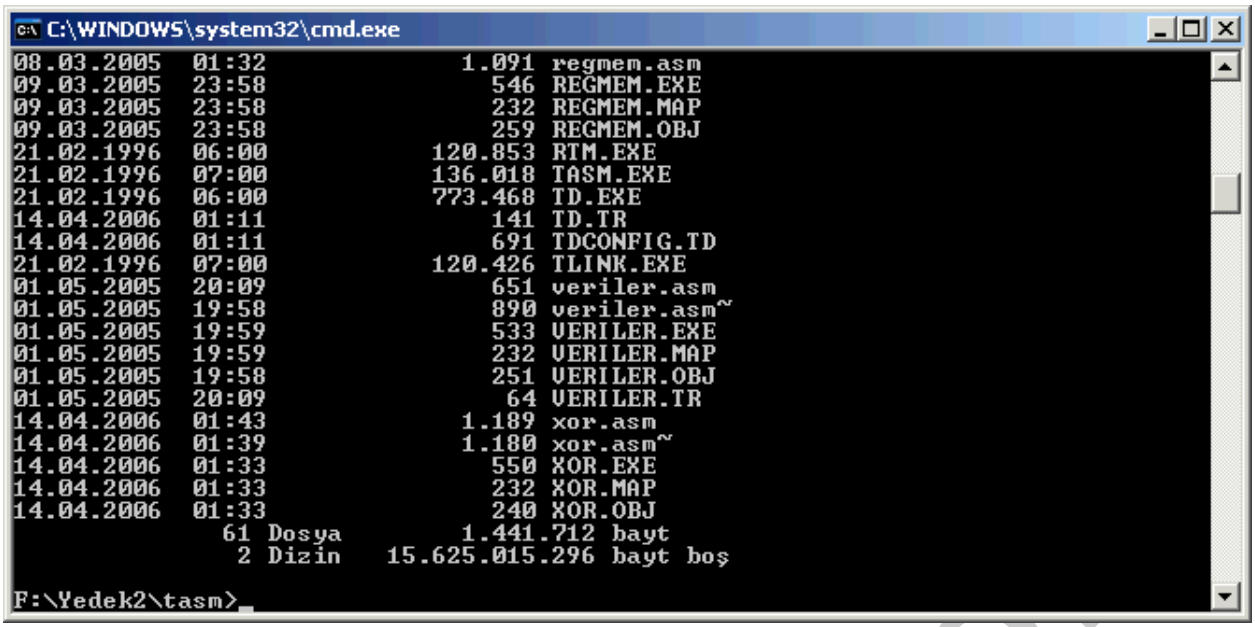
Konsol penceresi açılınca şayet işletim sisteminiz Windows XP ise "Microsoft Windows XP....." ile başlayan bir yazı görülür. Debug'ı çalıştırıp b800:0000 adresini ekrana döktüğümüzde bu yazı hala aynı yerindedir. B800h segmenti de burada görünen yazının bulunduğu text video alanıdır. Şekilde B800:0000 adresindeki 4Dh değerinin aslında M harfi olduğunu ve C karakterininde yine aynı segmentte bir değer olduğunu görüyorsunuz. Yukarıdaki resimdeki hafıza dökümü, en üst satırın tamamını ve sonraki satırın yarısından biraz fazlasını gözler önüne seriyor. Hafıza dökümünde tek numaralı her offsette (0001h,0003h,00F3h gibi) 07h

değerini görüyorsunuz. Bu text mode özelliği olup siyah zemin üzerine beyaz yazı anlamına gelir. Gerçektende ekrandaki karakterlerin hepsi beyaz ve zemin siyah değil mi?

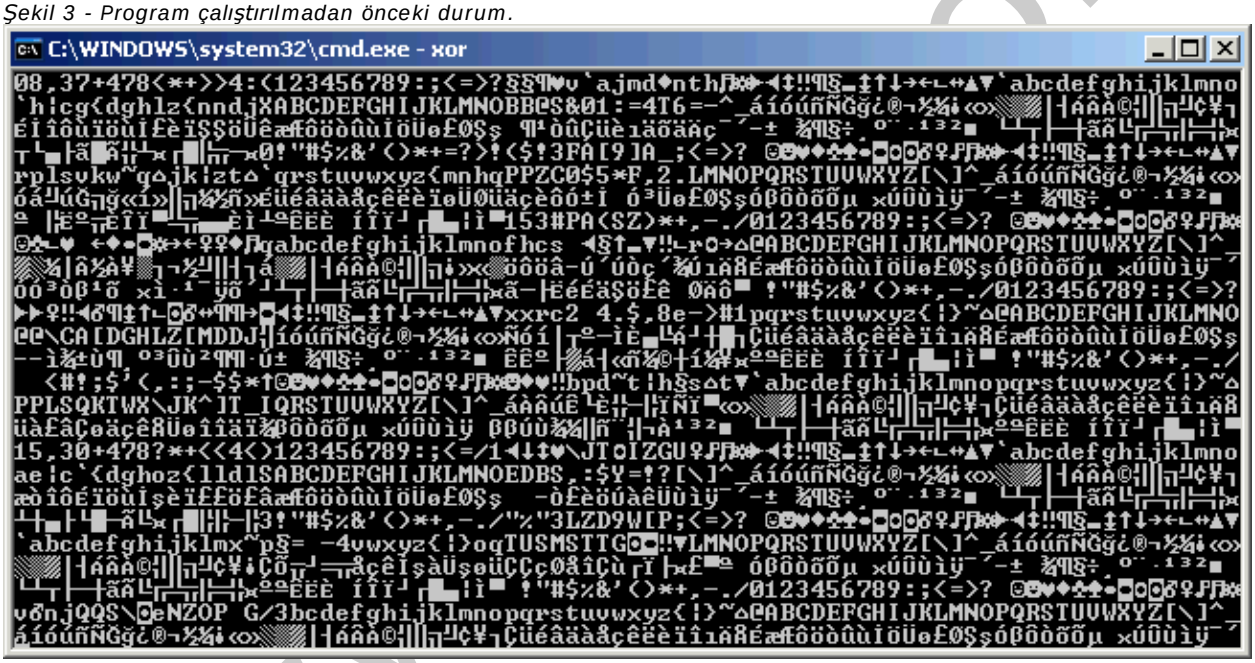
Konsol ekranının tamamı karakterlerle dolsa, ekranda kaç karakter görürüz? Tabi ki 80x25=2000 adet. Şimdi bu karakterlerin tek tek adreslerine erişip (özellik içeren 07h byte'larını atlayarak) bunları lojik bir işlemden geçirsem ekranın o anki görüntüsü değişir, hem de bizim boş olarak tabir ettiğimiz ama aslında 20h olan yerler bile. Böyle bir durumda ekranda karman çorman anlaşılmaz bir görüntünün belirmesi olası bir durumdur.
Peki değiştirdiğim her byte'ı tekrar eski haline getirebilir miyim? Bütün bunların cevabı aşağıdaki programda saklı.

```
 TITLE XOR    (xor.asm)
;##########################################
;# Ekranı karıştır sonra tekrar düzelt  #
;# Son Güncelleme: 13/04/06             #
;# Yazan --> Eren ERENER                #
;##########################################

.MODEL SMALL
.STACK 32

.CODE

    ANA PROC

        mov     ax, 0B800h      ;80x25 text video alanını,
        mov     ds, ax          ;adreslemek için hazırlık.

        mov     al,2            ;80x25 text moda,
        int     10h             ;geçiş yap.

        mov     dh,1            ;1. sayaç
TEKRAR: xor     bx,bx           ;bx=0 (text video alanınının başlangıç adresi)
        xor     dl,dl           ;dl=0
        mov     cx,2000         ;text video alanının görünen kısmı (ilk pencere)
DEVAM:  xor     ds:[bx],dl      ;text video alanı dl ile xor yapılıyor
        inc     dl              ;dl=dl+1
        inc     bx              ;bir sonraki,
        inc     bx              ;karakterin konumunu ayarla.
        loop    DEVAM           ;tum ekrana bu işlemi uygula

        mov     ah,7            ;bir tuşa basılmasını.
        int     21h             ;bekle.
        dec     dh              ;dh=0 (aynı programı 1 defa
                                ;daha çalıştırmak için kontrol)
        je      TEKRAR          ;aynı programı yine çalıştır.
                                ;2. çalıştırma karakterleri orijinal hallerine
                                ;çevirir.
        mov     ah,4ch          ;DOS'a
        int     21h             ;dönüş

    ANA ENDP
        END     ANA
```

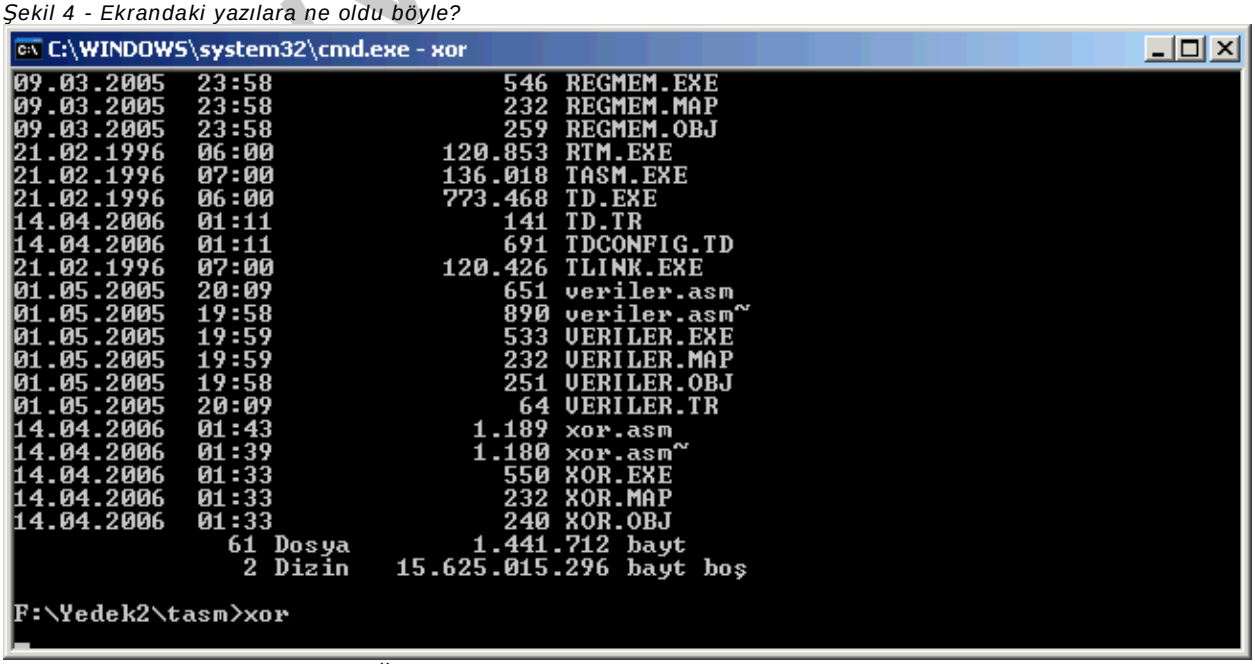
Şekil 2 - 80x25 text mode da görünen ekrana müdahele.



Şekil 3 - Program çalıştırılmadan önceki durum.



Şekil 4 - Ekrandaki yazılara ne oldu böyle?



Şekil 5 - Klavyeden bir tuşa basıldığında görüntü tekrar normale döner.

Aslında video işlemleri için daha pratik interrupt'lar mevcuttur, burada anlatmaya çalıştığım, "şayet assembly dili ile hafızanın her yerine erişebilecek deneyime sahipseniz yapamayacağınız şey yoktur" gerçeğiydi.

Bu programda özellikte, text video alanının segment adresini DS kaydedicisine yüklenişini ve bu alanın kaydedici dolaylı (register indirect) adresleme modu ile sanki kendi tanımladığımız bir data alanı gibi kullanılışını incelemenizi tavsiye ederim.

Düşük seviyede sağlam ve hızlı kodlar yazmanız dileğiyle... :)

Makalemizde, 80X86 komut sistemine ait aritmetik ve lojik komutları açıklamaya devam edeceğiz.

80x86 KOMUT SETİ (BÖLÜM 5)

AAM ve AAD Komutu:

AAM komutu, çarpma işlemi sonucunda, AX registerinde oluşan değeri ASCII formata dönüştürme amaçlı kullanılır. Daha önce anlatılan AAA ve AAS komutlarında olduğu gibi AAM ve AAD komutlarıda operandsız yazılırlar. AH ve AL registerlerini gizli operand olarak kulanmaktadırlar.
AAM komutu, çarpma işlemi ile ax registerinde oluşan değerin ardından kullanılır. Al'nin değerini desimal 10 ile böler, elde edilen değeri ah'a aktarır. Bölümden kalan değeride tekrar al'ye aktarır.Yapılan işlemin sonucunda, elde edilen sayının sol dijiti ah'da, sağ dijiti ise al'de yer alır.AAM komutu, parity, sign ve zero flaglarını al'ye aktardığı değere göre uygun şekilde etkiler.

Örneğimizde mul komutu, çarpım sonucu olan 00 51 değerini ax registerine aktarırken, aam komutuda bu değeri 00 81 olarak ASCII formatına dönüştürmüştür.

```
mov al,09      ;al'nin değeri 9
mov bl,09      ;bl'nin değeri 9
mul bl         ;ax = 00 51h
aam            ;ax = 08 01h
```

I. adım al registerinin değerini 0Ah'a böl 0Ah=d'10'	51h = 0101 0001b _____ = 08h 0Ah = 0000 1010b
II.adım bölümün sonuç değerini ah'a , kalan'ı al'ye aktar	ah = 08h al = 01h
III.adım Sonuç	ax = 0801h

AAD komutu ise bölme işleminde bölünenin sonucunda, AX registerinde oluşan değeri ASCII formata dönüştürme amaçlı kullanılır.AAD komutu, bölünen değerin yüksek seviyeli kısmını (ah registerinin içeriğini) desimal 10 ile çarpar ve al (düşük seviyeli kısmı) ile toplayarak al içerisine aktarır.
Yapılan işlem sonucunda ah registeri sıfırlanır, al registerinde bulunan BCD sayısı, başka bir BCD sayısı ile bölünebilir duruma getirilmiş olur. AAD komutu, parity, sign ve zero flaglarını

al'ye aktardığı değere göre uygun şekilde etkiler.

Örneğimizde desimal 35 sayısı, desimal 5'e bölünmekte ve sonuç yine desimal olarak elde edilmektedir. AAD komutunu kullanarak, ax üzerine paketlenmemiş olarak aktarılan sayıyı, bölme işlemi için uygun hale getiriyoruz.

```
mov ax, 0305    ;ax = 03 05
aad             ;ax = 00 23
mov bl,05       ; bl'ye al değerini koy
div bl          ;ax = 00 07
```

I. adım ah registerinin içeriğini 0Ah ile çarp 0Ah=d'10'	03h = 0000 0011b 0Ah = 0000 1010b *_____ 1Eh = 0001 1110b
II.adım çarpım sonucundaki değere al'nin değerini ekle ve al'ye yaz	1Eh = 0001 1110b 05h = 0000 0101b +_____ 23h = 0010 0011b
III.adım Sonuç	23h = 0010 0011b _____ = 7 05h = 0000 0101b

DAA ve DAS komutu:

DAA komutu ile yapılan toplama işlemi sonucunda her bir byte'da iki adet desimal dijit bulunur. ADD komutunun arkasından kullanılan DAA komutu, al'nin içeriği olan değere ve AF (auxilary flag) ile CF (carry flag)'nin durumuna bağlı olarak iki farklı şekilde işleme sokulur.
I. Durum: al registerinin düşük seviyeli 4 biti 9'dan büyük veya AF set ise, al registeri 6 ile toplanarak, AF set edilir.
II.Durum: al registerinin yüksek seviyeli 4 biti 9'dan büyük veya CF set ise, al registeri ile 60h toplanarak CF set edilir.

Örneğimizde 9 ve 5 değerlerini toplayarak komutumuzla istenilen sonucu elde edeceğiz.

```
mov al,09  ; al = 9
add al,05  ;  al = 0E
daa        ;  al =14
```

I. adım al registerlerini topla	09h = 0000 1001b 05h = 0000 1001b +_____ 0Eh = 0000 1110b
II.adım al registerinin düşük	0Eh = 0000 1110b 06h = 0000 0110b +_____ 14h = 0001 0100b

seviyeli biti 0'dan küçük olduğu için 6 değerini ekledik ve sonuç değere ulaştık	

DAS komutu ile yapılan çıkartma işleminin al registeri üzerindeki sonucuna ve auxuiliary ile carry flaglarının durumuna bağlı olarak iki farklı şekilde kullanılır.
I. Durum: al registerinin düşük seviyeli 4 biti 9'dan büyük veya AF set ise, al registerinden 6 çıkartılarak, AF set edilir.
II.Durum: al registerinin yüksek seviyeli 4 biti 9'dan büyük veya CF set ise, al registerinden 60h çıkartılarak CF set edilir.
DAS komutu, auxiliary, carry, parity, sign ve zero flaglarını etkiler.

Örneğimizde, desimal 13 'den 5 'i çıkartma işlemini al registerinde yaptığımız için sonuç 0E oluyor. al registerindeki değerin desimal sayı olması için çıkartma komutunun ardından DAS düzenleyici komutu kullanılıyor.

```
mov al,13   ;al=13h
mov bl,05   ;bl=05h
sub al,bl   ;al = 0Eh
das         ;al = 08
```

I. adım al registerindeki değerden bl registerindeki değeri çıkart	13h = 0001 0011b 05h = 0000 1001b - _____ 0Eh = 0000 1110b
II.adım al registerinin düşük seviyeli biti 0'dan küçük olduğu için 6 değerini çıkardık ve sonuç değere ulaştık	0Eh = 0000 1110b 06h = 0000 0110b - _____ 08h = 0000 1000b

CBW ve CWD Komutu:

CBW:Byte'ın worde çevrilmesi.
CWD:Word'un double worde çevrilmesi

Aritmetik işlemler sırasında operandlar farklı uzunluklara (byte,word veya double word) sahip olabilirler. Bu gibi durumlarda, işlem öncesinde operandların uzunluklarının düzenlenmesi gerekir.Küçük uzunluğa sahip operandın uzunluğunu, büyük uzunluğa sahip operandın uzunluğuna denkleştirilmektedir.İşaretsiz sayılarla temsil edilen operandlar, rahat bir şekilde

düzenlenir. Fakat işaretli sayılarla temsil edilen operandlarda değişim o kadar rahat olmamaktadır.
İşaretli sayıları içeren operandların uzunluklarının düzenlenmesi için CBW ve CWD komutları geliştirilmiştir. CBW komutu al ve ah, CWD komutu ise ax ve dx'i gizli operand olarak kullanırlar.
CBW komutu, al içerisindeki işaretli sayının işaret bitini ah'ın tüm bitlerine, CWD komutu da ax içindeki sayının işaret bitinin değerini dx'in tüm bitlerine aktararak sonuca ulaşmamızı sağlar.

TEST Komutu:

Test komutunun çalışma prensibi ve flagları etkileme biçimide dahil olmak üzere And ile aynıdır. Farklı olarak, komut ile birlikte kullanılan operandların değerlerini değiştirmez. Diğer lojik komutlarında olduğu gibi, carry, owerflow bayrakları test işleminin sonrasında reset edilirler, auxiliary dışındaki diğer bayraklarda işlem sonucuna uygun şekilde etkileneceklerdir.

Farkı olmamasına rağmen, hangi durumda and, hangi durumda test komutunu kullanmalıyız sorusu aklımıza gelecektir. Örneğimizi inceleyelim;

I. Durum	and al,01 jz * ... ...
II.Durum	test al,01 jz * ... ...

I. ve II. Durumda, al registerinin değeri desimal 1 ile karşılaştırılmaktadır; al registerindeki değer desimal 1'e eşit ise
I.Durumda: al'nin değeri korunacak ve zero bayrağının reset durumuna gelmesine sebeb olacak, bu sayede jz'yi izleyen komut ile program devam etmiş olacaktır. Fakat değer 1'den farklı ise sonuç 0 olarak al'ye yerleştirilecek ve zero bayrağı set edilecektir, sonucunda ise * yerine kullanılan adrese dallanılacak ve program bu adreste bulunan komut ile sürecektir. Fakat al'nin değeri 1'e eşit değil ve başka işlemler için bize gerekli ise sorun çıkacaktır.Sonuçtan da anlaşıldığı gibi, al registerinin karşılaştırma öncesi değeri, karşılaştırma sonrası adımlarda da gerekli ise test komutunu tercih etmeliyiz.

SHR ve SHL Komutu:

SHR Komutu ile, soldaki operandın en düşük seviyeli biti carry bayrağına kopyalanır ve operandın tüm bitleri sağa doğru 1'er bit kayar. En soldaki bitin değeri 0 yapılır. Bu işlem SHR ile verilen ikinci operandın değeri kadar tekrarlanır.

Örneğimizde, 30h değerini iki kere 1'er biti sağa kaydırarak sonucun nasıl değiştiğini inceleyeceğiz.

```
mov al,30h    ;al=30h
shr al,01     ;al=18h
shr al,01     ;al=0Ch
```

Şekil 1 - Shr komutu ile bitlerin sağa kaydırılması

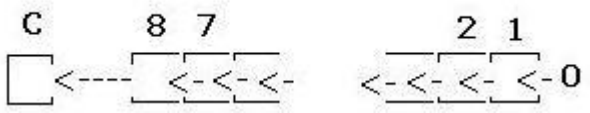
Adımlar	Hexadesimal	Binary	Desimal	CF
I. adım: al'ye 30h değerini koy	al=30h	00110000	48	-
II.adım: değerleri 1 bit sağa kaydır, en düşük seviyeli biti CF'na taşı	al=18h	00011000	24	0
III.adım: değerleri 1 bit sağa kaydır, en düşük seviyeli biti CF'na taşı	al=0Ch	00001100	12	0

Programda da görüldüğü gibi her bir sağa kaydırma işlemi ile sayıyı ikiye bölmektedir.Her bir shr işlemi sonrasında carry bayrağı 0 ise çift sayı, 1 ise tek sayıdır.

SHL Komutu ile, soldaki operandın en yüksek seviyeli biti carry bayrağına kopyalanır ve operandın tüm bitleri sola doğru 1'er bit kayar. En soldaki bitin değeri 0 yapılır. Bu işlem SHL ile verilen ikinci operandın değeri kadar tekrarlanır.

Örneğimizde 0Ch değerini iki kere 1'er bit sola kaydırarak sonuç üzerindeki değişiklikleri inceleyeceğiz.

```
mov al,0Ch    ;al=0Ch
shr al,01     ;al=18h
shr al,01     ;al=30h
```



Şekil 2 - Shl komutu ile bitlerin sola kaydırılması

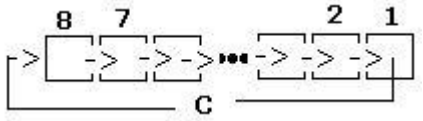
Adımlar	Hexadesimal	Binary	Desimal	CF
I. adım: al'ye 0Ch değerini koy	al=0Ch	00001100	12	-
II.adım: değerleri	al=18h	00011000	24	0

1 bit sola kaydır, en yüksek seviyeli biti CF'na taşı				
III.adım: değerleri 1 bit sola kaydır, en yüksek seviyeli biti CF'na taşı	al=0Ch	00110000	48	0

Programda da görüldüğü gibi her bir sola kaydırma işlemi ile sayıyı iki ile çarpmış oluyoruz.

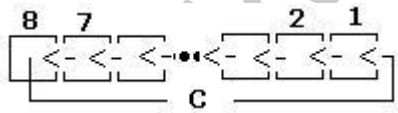
RCR ve RCL Komutu:

RCR Komutu, işleme tabi tutulan bitlerin herbirini bir pozisyon sağa kaydırır. Düşük seviyeli bit pozisyonundaki değer carry bayrağına, carry bayrağının değeride yüksek seviyeli bit pozisyonuna aktarılır. İşlem ikinci operand değeri kadar tekrarlanır.

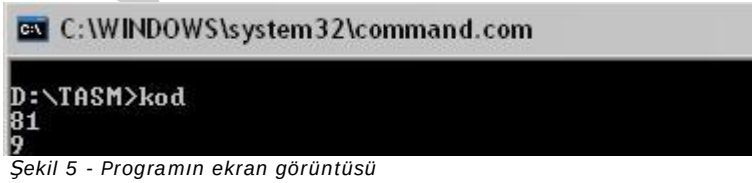


Şekil 3 - Rcr komutu ile bitlerin sağa kaydırılması

RCL Komutu, işleme tabi tutulan bitlerin herbirini bir pozisyon sola kaydırır. Carry flağının değeri düşük seviyeli bit pozisyonuna, yüksek seviyeli bit pozisyonundaki değer ise carry flağına aktarılır. İşlem ikinci operand değeri kadar tekrarlanır.



Şekil 4 - Rcl komutu ile bitlerin sola kaydırılması



Şekil 5 - Programın ekran görüntüsü

Bu bölümde anlattığımız aam,das,shr ve rcl komutlarını kullanarak örnek bir program hazırladım. aam komutu kullanılan prosedürde shr komutunu işleme katmak istediğimiz al registerinin içeriğini bir bit sağa kaydırmak için, das ile ilgili prosedürde ise rcl komutuyla al registerinin içeriğini bir bit sola kaydırdım.

```
.model small
.stack 64
.data

aamsonuc db '00$',10,13
dassonuc db '00$'
knm db 10,13,'$'

;.................................................
.code
mov ax,@data
mov ds,ax
mov es,ax

call aamornek
call dasornek

;.................................................
aamornek proc

mov al,12h
shr al,01                       ;al registerinin içeriğini bir bit sağa kaydır, al=9h

mov bl,9h                       ;bl=9h
mul bl                          ;ax=51h
aam                             ;ax=81h

add ax,3030h                    ;al ve ah değerlerini ascii karakterlere çevir.

mov byte ptr [aamsonuc],ah
mov byte ptr [aamsonuc+1],al    ;ax registerini sonuc dizisi içerisine yerleştir.

mov ah,09h
lea dx,aamsonuc                 ;sonucu yaz
int 21h

mov ah,09h
lea dx,knm                      ;alt satıra geç
int 21h

aamornek endp

;.................................................
dasornek proc

mov al,28h
rcr al,01                       ;al registerinin içeriğini bir bit sola kaydır, al=14h

mov bl,05h                      ;bl=5h
sub al,bl                       ;al=Fh
das                             ;al=9h

add al,30h                      ;al değerini ascii karaktere çevir.

mov ah,02h
mov dl,al
int 21h                         ;sonucu yaz

dasornek endp
;.................................................
mov ah,4ch
```

```
int 21h
end                                    ;programdan çıkış
;.....................................................
```

Programımız, anlatılan komutların sağlaması olarak da değerlendirilebilir. Ben kodu çalıştırdım ve gördüm ki kullandığım komutlar işini başarıyla gerçekleştiriyor.

Makalemizde aritmetik lojik komutların çalışma şekillerini, bayraklar üzerindeki etkilerini anlatmaya çalıştım. Benzer özelliklere sahip komutların kullanılması sizlerin tercihine kalmış, gerçekleştireceğiniz uygulamalar ve elde etmek istediğiniz performans doğrultusunda ihtiyacınıza en uygun olanı belirleyebilirsiniz.

Makalemizde, 80X86 komut sistemine ait, CMPXCHG, CMPXCHG8B, SAL, SAR, SHLD, SHRD, ROL, ROR, BT, BTS, BTR, BTC, BSF, BSR aritmetik lojik komutlarını açıklamaya devam edeceğiz.

80x86 KOMUT SETİ (BÖLÜM 6)

CMPXCHG ve CMPXCHG8B Komutu:

CMPXCHG Komutu

Komut	Anlamı	İşlem
CMPXCHG	CMPXCHG r/m8,al	al ile r/m8'i karşılaştır. Eşitse ZF değeri 1 olur ve al değeri r/m8'e yüklenir. Değilse ZF değeri 0 olur ve r/m8 değeri al'ye yüklenir.
CMPXCHG	CMPXCHG r/m16,ax	ax ile r/m16'yı karşılaştır. Eşitse ZF değeri 1 olur ve ax değeri r/m16'ya yüklenir. Değilse ZF değeri 0 olur ve r/m16 değeri ax'e yüklenir.
CMPXCHG	CMPXCHG r/m32,eax	eax ile r/m32'i karşılaştır. Eşitse ZF değeri 1 olur ve eax değeri r/m32'ye yüklenir. Değilse ZF değeri 0 olur ve r/m32 değeri eax'e yüklenir.

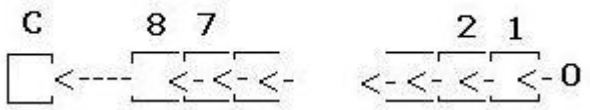
CMPXCHG8B Komutu, ile 64 bitlik sayılar üzerinde işlemler yapabilmek için kullanıyoruz, kendisine gelen adresteki değeri EDX:EAX içindeki değer ile karşılaştırır. Eşit ise, ECX:EBX'deki değeri 64 bitlik operanda yerleştirilir. Eşit olmaması halinde ise, hedef operand değeri EDZ:EAX'e yerleştirilir. Hedef operand 8 byte hafıza alanına sahiptir. EDX:EAX ve ECX:EBX 64 bitlik registerlerinin; EDX ve ECX yüksek seviyeli 32 bitini, EAX ve EBX ise düşük seviyeli 32 bitini paylaşır.Hedef operand ve EDX:EAX'in değerleri eşit olursa zero bayrağı etkilenir.

Komut	Anlamı	İşlem
CMPXCHG8B	CMPXCHG8B m64	m64 ile EDX:EAX değerini karşılaştır. Eşitse ZF değeri 1 olur ve ECX:EBX'teki değer m64'e yüklenir. Değilse ZF değeri 0 olur ve m64 değeri EDX:EAX'e yüklenir.

SAL ve SAR Komutu:

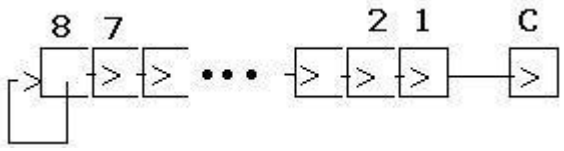
SAL Komutu ,ile soldaki operandın en yüksek seviyeli biti carry bayrağına kopyalanır ve

operandın tüm bitleri sola doğru 1'er bit kayar. En soldaki bitin değeri 0 yapılır. Bu işlem SAL ile verilen ikinci operandın değeri kadar tekrarlanır.



Şekil 1 - Sal komutu ile bitlerin sola kaydırılması

SAR komutu ,ile soldaki operandın en düşük seviyeli biti carry bayrağına kopyalanır ve operandın tüm bitleri sağa doğru 1'er bit kayar. En soldaki bitin değeri de kaydırılır fakat eski değeri korunur. Bu bit işaretli sayılarda sign biti olarak kullanılır. Bu işlem SAR ile verilen ikinci operandın değeri kadar tekrarlanır.



Şekil 2 - Sar komutu ile bitlerin sağa kaydırılması

SHLD ve SHRD Komutu:

SHLD Komutu hedef operand değeri, 3. operand ile belirlenen değer kadar sola kaydırılır. 2. operand sağdan gelen bitleri tutar. Hedef operand, register yada hafıza alanı, 3. operand ise register olabilir. Sayaç operandı byte olarak tanımlanan işaretsiz tamsayı yada CL registerinin değeridir. Eğer sayaç değeri operand değerinden fazla ise hedef operand içerisinde sonuç tanımlanamaz. Sign, Zero, Auxiliary, Parity ve Carry bayraklarını etkiler.

Komut	Anlamı	İşlem
SHLD	SHLD r/m16,r16,opr3	r/m16 değerini, opr3 ile belirlenen değer kadar sola kaydırırken, r16 sağdan gelen bitleri tutar.
SHLD	SHLD r/m16,r16,CL	r/m16 değerini, CL ile belirlenen değer kadar sola kaydırırken, r16 sağdan gelen bitleri tutar.
SHLD	SHLD r/m32,r32,opr3	r/m32 değerini, opr3 ile belirlenen değer kadar sola kaydırırken, r32 sağdan gelen bitleri tutar.
SHLD	SHLD r/m32,r32,CL	r/m32 değerini, CL ile belirlenen değer kadar sola kaydırırken, r32 sağdan gelen bitleri tutar.

SHRD Komutu hedef operand değeri, 3. operand ile belirlenen değer kadar sağa kaydırılır. 2. operand soldan gelen bitleri tutar.Hedef operand, register yada hafıza alanı, 3. operand ise register olabilir. Sayaç operandı byte olarak tanımlanan işaretsiz tamsayı yada CL registerinin değeridir. Eğer sayaç değeri operand değerinden fazla ise hedef operand içerisinde sonuç tanımlanamaz. Sign, Zero, Auxiliary, Parity ve Carry bayraklarını etkiler.

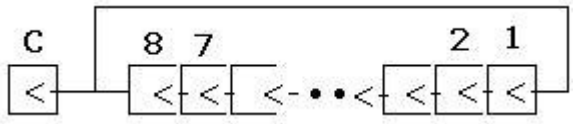
Komut	Anlamı	İşlem

SHRD	SHRD r/m16,r16,opr3	r/m16 değerini, opr3 ile belirlenen değer kadar sağa kaydırırken, r16 soldan gelen bitleri tutar.
SHRD	SHRD r/m16,r16,CL	r/m16 değerini, CL ile belirlenen değer kadar sağa kaydırırken, r16 soldan gelen bitleri tutar.
SHRD	SHRD r/m32,r32,opr3	r/m32 değerini, opr3 ile belirlenen değer kadar sağa kaydırırken, r32 soldan gelen bitleri tutar.
SHRD	SHRD r/m32,r32,CL	r/m32 değerini, CL ile belirlenen değer kadar sağa kaydırırken, r32 soldan gelen bitleri tutar.

SHL ve SHR ile aynı şekilde çalışmalarına rağmen, bit alanları 64 bite kadar çıkabildiklerinden dolayı SHLD ve SHRD komutları kullanılır.

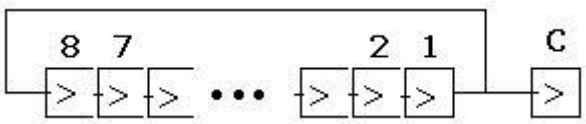
ROL ve ROR komutu:

ROL komutu ile, operandın tüm bitleri sola doğru 1'er bit kayar. En soldaki bitin değeri, carry bayrağına ve düşük seviyeli bit pozisyonuna aktarılır. Bu işlem ROL ile verilen ikinci operandın değeri kadar tekrarlanır.



Şekil 3 - Rol komutu ile bitlerin sola kaydırılması

ROR komutu ile, operandın tüm bitleri sağa doğru 1'er bit kaydırılır. En sağdaki bitin değeri, carry bayrağına ve yüksek seviyeli bit pozisyonuna aktarılır. Bu işlem ROR ile verilen ikinci operandın değeri kadar tekrarlanır.



Şekil 4 - Ror komutu ile bitlerin sağa kaydırılması

Genel olarak kaydırma komutları gibi çalışırlar, fakat adında anlaşılacağı gibi bit değerleri kaybolmaz, dönerek yer değiştirirler.

BT ve BTS Komutu:

BT komutu ile, bit ofset operandı ile gösterilen pozisyonda, bit dizisi içerisinde bulunan bit seçilir ve carry bayrağına depolanır. Temel bit operandı register ya da hafıza alanı, ofset operand ise register yada registere yakın bir değer olabilir. Eğer temel bit operand register ise, registerin boyutuna bağlı olarak komutun değeri, ofset operandın 16 veya 32'ye göre mod'u alınarak bulunur. Eğer temel bit operand hafıza alanı ise, temel bit değerinin olduğu hafıza içerisinde baytın adresi gösterilir. Carry bayrağı etkilenir.

Komut	Anlamı	İşlem

BT	BT r/m16, r16	r16 dizisi içinde, r/m16 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola.
BT	BT r/m32, r32	r32 dizisi içinde, r/m32 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola.
BT	BT r/m16, opr2	opr2 dizisi içinde,r/m16 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola.
BT	BT r/m32, opr2	opr2 dizisi içinde,r/m32 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola.

BTS komutu ile, bit ofset operandı ile gösterilen pozisyonda, bit dizisi içerisinde bulunan bit seçilir, carry bayrağına depolanır ve 1 olarak ayarlanır. Temel bit operandı register ya da hafıza alanı, ofset operand ise register yada registere yakın bir değer olabilir.Eğer temel bit operand register ise, registerin boyutuna bağlı olarak komutun değeri, ofset operandın 16 veya 32'ye göre mod'u alınarak bulunur. Eğer temel bit operand hafıza alanı ise, temel bit değerinin olduğu hafıza içerisinde baytın adresi gösterilir. Carry bayrağı etkilenir.

Komut	Anlamı	İşlem
BTS	BTS r/m16, r16	r16 dizisi içinde, r/m16 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve değerini 1 olarak ayarla.
BTS	BTS r/m32, r32	r32 dizisi içinde, r/m32 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve değerini 1 olarak ayarla.
BTS	BTS r/m16, opr2	opr2 dizisi içinde, r/m16 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve değerini 1 olarak ayarla.
BTS	BTS r/m32, opr2	opr2 dizisi içinde, r/m32 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve değerini 1 olarak ayarla.

BTR ve BTC Komutu:

BTR komutu ile, bit ofset operandı ile gösterilen pozisyonda, bit dizisi içerisinde bulunan bit seçilir, carry bayrağına depolanır ve 0 değeri atanır.Temel bit operandı register ya da hafıza alanı, ofset operand ise register yada registere yakın bir değer olabilir.Eğer temel bit operand register ise, registerin boyutuna bağlı olarak komutun değeri, ofset operandın 16 veya 32'ye göre mod'u alınarak bulunur. Eğer temel bit operand hafıza alanı ise, temel bit değerinin olduğu hafıza içerisinde baytın adresi gösterilir. Carry bayrağı etkilenir.

Komut	Anlamı	İşlem
BTR	BTR r/m16, r16	r16 dizisi içinde, r/m16 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve değerini 0 olarak ayarla.
BTR	BTR r/m32, r32	r32 dizisi içinde, r/m32 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve değerini 0 olarak ayarla.
BTR	BTR r/m16, opr2	opr2 dizisi içinde, r/m16 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve değerini 0 olarak ayarla.
BTR	BTR r/m32, opr2	opr2 dizisi içinde, r/m32 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve değerini 0 olarak ayarla.

BTC komutu ile, bit ofset operandı ile gösterilen pozisyonda, bit dizisi içerisinde bulunan bit seçilir, carry bayrağına depolanır ve tersi alınır.Temel bit operandı register ya da hafıza alanı, ofset operand ise register yada registere yakın bir değer olabilir.Eğer temel bit operand register ise, registerin boyutuna bağlı olarak komutun değeri, ofset operandın 16 veya 32'ye göre mod'u alınarak bulunur. Eğer temel bit operand hafıza alanı ise, temel bit değerinin olduğu hafıza içerisinde baytın adresi gösterilir. Carry bayrağı etkilenir.

Komut	Anlamı	İşlem
BTC	BTC r/m16, r16	r16 dizisi içinde, r/m16 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve tersini al.
BTC	BTC r/m32, r32	r32 dizisi içinde, r/m32 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve tersini al.
BTC	BTC r/m16, opr2	opr2 dizisi içinde, r/m16 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve tersini al.
BTC	BTC r/m32, opr2	opr2 dizisi içinde, r/m32 ile gösterilen pozisyondaki biti seç ve Carry bayrağına depola ve tersini al.

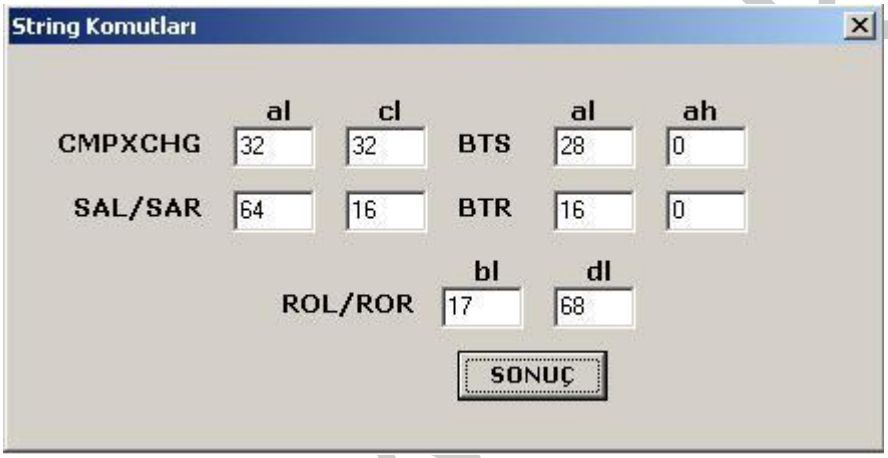
BSF ve BSR Komutu:

BSF komutu , kaynak (2.) operand içerisindeki en anlamsız biti, 1 yapmak için arar. Eğer bulursa, hedef (1.) operand içerisinde index biti olarak kullanılır. Kaynak operand register ya da hafıza alanı, hedef operand ise register olabilir. Eğer kaynak operandın değeri 0 ise, hedef operandın değeri tanımlanamaz.

Komut	Anlamı	İşlem
BSF	BSF r16, r/m16	r/m16 içerisinde en anlamsız biti ara ve değerini 1 olarak ayarla.
BSF	BSF r32, r/m32	r/m32 içerisinde en anlamsız biti ara ve değerini 1 olarak ayarla.

BSR komutu ,kaynak (2.) operand içerisindeki en anlamlı biti, 1 yapmak için arar. Eğer bulursa, hedef (1.) operand içerisinde index biti olarak kullanılır. Kaynak operand register ya da hafıza alanı, hedef operand ise register olabilir. Eğer kaynak operandın değeri 0 ise, hedef operandın değeri tanımlanamaz.

Komut	Anlamı	İşlem
BSR	BSR r16, r/m16	r/m16 içerisinde en anlamlı biti ara ve değerini 1 olarak ayarla.
BSR	BSR r32, r/m32	r/m32 içerisinde en anlamlı biti ara ve değerini 1 olarak ayarla.



Şekil 5 - Programın Ekran Görüntüsü

Örneğimizde; CMPXCHG komutuyla al=14h ve cl=20h registerindeki değerler karşılaştırılıyor, değerler farklı oluğu için, Şekil 6'da CMPXCHG komutunun sonrasında cl registerin'deki değerin al'ye aktarıldığını görüyoruz.

Register	16'lı	Des	Bin	İşlem	16'lı	Des	Bin
al	14	20	0001 0100	CMPXCHG	20	32	0010 0000
cl	20	32	0010 0000		20	32	0010 0000

Şekil 6 - CMPXCHG komutu öncesi ve sonrasında al,cl registerlerinin almış olduğu değerler

SAL/SAR komutu ile al ve cl registerlerinde sola ve sağa doğru ikinci operand değeri kadar kaydırıyoruz, Şekil7'de SAL komutu sonrasında al'nin , SAR komutunun ardından cl registerinin yeni değeri görülmektedir.

Register	16'lı	Des	Bin	İşlem	16'lı	Des	Bin
al	20	32	0010 0000	SAL	40	64	0100 0000
cl	20	32	0010 0000	SAR	10	16	0001 0000

Şekil 7 - SAL ve SAR komutları öncesi ve sonrasında al,cl registerlerinin almış olduğu değerler

ROL/ROR komutu ile bl ve dl registerlerinde sola ve sağa doğru ikinci operand değeri kadar kaydırıyoruz. Şekil8'de ROL komutu sonrasında en soldaki bitin değerini carry bayrağına ve düşük seviyeli bit pozisyonuna,ROR komutu sonrasında en sağdaki bitin değerini carry bayrağına ve yüksek seviyeli bit pozisyonuna aktarıyoruz.

Register	16'lı	Des	Bin	İşlem	16'lı	Des	Bin
bl	88	136	1000 1000	ROL	11	17	0001 0001
dl	88	136	1000 1000	ROR	10	68	0100 0100

Şekil 8 - ROL ve ROR komutları öncesi ve sonrasında bl,dl registerlerinin almış olduğu değerler

BTS komutu ile ax registerinin 3. bitini set ediyoruz. Şekil9'da BTS komutu sonrasında seçilen bitin değerini carry bayrağına aktarıp 1 yapıyoruz.

Register	16'lı	Des	Bin	İşlem	16'lı	Des	Bin
al	14	20	0001 0100	BTS	1C	28	0001 1100
ah	0	0	0000 0000		0	0	0000 0000

Şekil 9 - BTS komutu öncesi ve sonrasında al,ah registerlerinin almış olduğu değerler

BTR komutu ile ax registerinin 2. bitini seçiyoruz. Şekil10'da BTR komutu sonrasında seçilen bitin değerini carry bayrağına aktarıp 0 yapıyoruz.

Register	16'lı	Des	Bin	İşlem	16'lı	Des	Bin
al	14	20	0001 0100	BTR	10	16	0001 0000
ah	0	0	0000 0000		0	0	0000 0000

Şekil 10 - BTR komutu öncesi ve sonrasında al,ah registerlerinin almış olduğu değerler

```
asm {

mov al, 0X14          //desimal 20'ye eşit
mov cl, 0X20          //desimal 32'ye eşit

cmpxchg cl, al        //al değeri ile cl değerlerini karşılaştır.
                      //değerler farklı olduğu için, cl değerini, al'ye at

mov cmpal,al          //cmpxchg komutu için değerleri sakla
mov cmpcl,cl

//-------------------------------------------------------------
sal al,1              //al değerini sal komutu ile sola doğru kaydır
sar cl,1              //cl değerini sar komutu ile sağa doğru kaydır

mov slal,al           //sal ve sar komutu için değerleri sakla
mov srcl,cl

//-------------------------------------------------------------
mov bl,0X88           // desimal 136'ya eşit
mov dl,0X88

rol bl,1              //al değerini rol komutu ile sola doğru kaydır
ror dl,1              //cl değerini ror komutu ile sola doğru kaydır

mov rlal,bl           //rol ve ror komutu için değerleri sakla
mov rrcl,dl

//-------------------------------------------------------------
mov ax,0X14     // desimal 20'ye eşit
```

```
bts ax,3        //bts komutu ile ax registerinin 3. bitini 1 yap

mov btal,al     //bts komutu için ax (al , ah) değerleri sakla
mov btah,ah

//-------------------------------------------------------------
mov ax,0X14     // desimal 20'ye eşit

btr ax,2        //btr komutu ile ax registerinin 2. bitini 0 yap

mov btral,al     //btr komutu için ax (al , ah) değerleri sakla
mov btrah,ah


}
```

C++ Builder içerisinde assembly kullanarak gerçekleştirdiğim programın kodlarını indirebilirsiniz. Programı bilgisayara aktarmadan, kendi belirlediğiniz değerler ile kağıt üzerinde işlemleri gerçekleştirirseniz komutların çalışma mantığı daha kalıcı olacaktır. Herkese iyi çalışmalar...

(NOT: Bu makaleler csharpnedir.com dan alınmıştır.
Bu makaleleri hazırladıkları için Eren ERENER ve Mustafa ARKAN 'a teşekkür ederiz.)