Assembly

Assembly Nedir?

Assembly dili en alt seviye programlama dilidir. Assembly dili ile bellek gözeneklerine ve işlemci yazmaçlarına direkt müdahale ederiz. Daha üst seviye programlama dillerinde kaynak kodu makina koduna dönüştürmek için son derece karmaşık işlemler yapılır. Halbuki Assembly dili, makina kodlarının akılda tutulması zor olduğu için türetilmiş, İngilizce sözcüklerin kısaltmasından ibaret olan bir dildir. Her makina kodunun Assembly dilinde bir karşılığı vardır.

X86 ve X64 ne demek?

- **X86**

X86 Intel'in 8086 işlemcisini merkez alan bir işlemci mimarisidir. Intel'in işlemci ailesinin son iki rakamı daima 86 ile biterdi. X86 ise bu geçmiş nesildeki işlemcilere atıfta bulunmanın bir kod adı haline geldi. Bu işlemcilerin hepsi 32-bit'ti. Bundan dolayı x86 mimarisi zaman içerisinde 32-bit olarak yeniden adlandırıldı.

- **X64**

X64 kısaca 64-bit işlemci mimarisi anlamına gelir.

- **Temel Farkları**

32-Bit boyutunda bir sayı 2^32 ya da 4,294,967,267 olası adrese dayanır. Buna karşılık 64-Bit boyutunda bir sayı, 2^64 yani 12,446,744,073,709,551,616 olarak karşılık bulur. 32-Bit mimaride 4 milyar byte yani yaklaşık 4 gigabayt veri bulunurken, 64-Bit mimaride 18 quintillion byte yani 18 milyar gigabayt veri bulunabilir. Temel olarak x64 mimarisi daha fazla hız ve hafıza sağlar.

Neden Assembly?

Kuşkusuz son derece gelişmiş ve işlerimizi son derece kolaylaştıran birçok programlama dili varken Assembly gibi çok alt seviye bir programlama diliyle program geliştirmek son derece mantıksız gibi görünüyor. Ancak Assembly dili ile diğer programlama dilleriyle yapamayacağımız birçok şeyi yapabiliriz.

Örneğin:

- Bilgisayar sistemlerini yakından tanımak için,
- Aygıt sürücüleri yazmak için,
- İşletim sistemleri yazmak için,
- Program ya da işletim sistemlerindeki güvenlik açıklarını görmek için,

- Şifre kırma ve hacking işlemleri için,
- Virüs ve antivirüs programları yazmak için.

Bunların yanına bütün programların kaynak kodlarını Assembly olarak görebileceğimizi de eklememiz gerek. Yani bir Assembly programcısı için bütün programlar açık kaynak kodludur.

Assembler ve Assembly kelimeleri ne anlama gelir?

Assembly bir programlama dilidir. Assembler ise “Assembly” dilinde yazdığımız kaynak kodlarımızı makina kodlarına dönüştüren programlardır.

Assembly Registerları (Kaydediciler)

➢ Genel Kaydediciler

Bu kaydedicilerin 8,16 ve 32 bitlik kullanımı mümkündür. Örneğin AL ve AH Accumulator’ün 8 bitlik kullanımını AX 16 bitlik EAX ise 32 bitlik kullanımını simgeler. Genel amaçlı kaydedicilerin hepsi verileri geçici olarak üzerlerinde barındırabilir fakat bazı x86 komutları buradaki kaydedicilere özeldir. Mesela loop komutu ile CX kaydedicisinin değeri azalır.

Accumulator (EAX, AX, AH, AL): En sık kullanacağınız kaydedicidir. Çok genel kullanım alanına sahiptir, daha önceki makalelerimizde yazdığımız kodlara bakarsanız çok değişik amaçlarla kullanıldığını görebilirsiniz. Bu kaydedici birçok giriş/çıkış işleminde ve aritmetik işlemlerde (çarpma, bölme ve taşıma gibi) kullanılır. Ayrıca bazı komutlar işlenmeden önce accumulator’den parametre alır veya bu kaydediciye işlemin sonucunu kaydeder.

Base (EBX, BX, BH, BL): Accumulator gibi genel amaçlı ve hafıza erişiminde indexleri göstermede kullanılır. Bir başka kullanım alanıda hesaplamalardır.

Counter (ECX, CX, CH, CL): Özel amaçlar ve hesaplamalarda kullanılacağı gibi genellikle bu kaydediciyi sayıcı olarak kullanılır, daha önce loop komutunun CX kaydedicisini otomatik olarak değiştirdiğini söylemiştik.

Data (EDX, DX, DH, DL): Bazı giriş/çıkış komutlarında bu kaydedicinin kullanılması gerekir, ayrıca çarpma ve bölme işlemlerinde accumulator ile birlikte büyük sonuçları bu kaydediciden okuruz.

➢ Segment Kaydedicileri

Segment kaydedicilerinin hepsi 16 bitliktir ve hafızanın segment olarak adlandırılan kısımlarını adreslemede kullanılır.

Code Segment Kaydedicisi (CS): DOS işletim sisteminde programları oluşturan kodlar code segment’e yüklenir. CS kaydedicisi ise IP kaydedicisi ile birlikte programın çalışma sürecinde,

programın oluşturan kodların adreslerini gösterirler.

Data Segment Kaydedicisi (DS): .exe türündeki bir programda kullanılacak olan veriler data segment denilen hafıza bölümünde tutulur. DS kaydedicisi ise bu bölgedeki verilerin konumlarını gösterir.

Stack Segment Kaydedicisi (SS): Tüm programlar stack segment denilen bir hafıza alanını geçici depolama alanı olarak kullanmak zorundadırlar (örneğin dallanma işlemlerinde). SS kaydedicisi ise SP kaydedicisi ile birlikte bu verilerin adreslerini referans eder.

Extra Segment Kaydedicisi (ES): Bazı string işlemlerinde DI kaydedicisi ile birlikte karakterlerin bulunduğu hafıza adreslerini tutarlar.

FS ve GS Kaydedicileri: 80386 ve sonrası CPU'larda bulunurlar ve diğer segment kaydedicilerinin yetersiz kaldığı durumlarda kullanılırlar.

➢ Özel kaydediciler

IP ve EIP kaydedicileri: IP 16 bitlik DOS programlarının EIP ise 32 bitlik programların işlenmesi sürecinde, işlenecek olan bir sonraki komutun offset adresini gösterir.

FLAG ve EFLAG kaydedicileri: Flag kaydedicisi 16 Eflag kaydedicisi ise 32 bitten oluşur. Bildiğiniz gibi mikroişlemci matematiksel işlem yapar, bu kaydedicilerde her işlemden sonra o işleme özel sonuçları gösterirler. İşlemci durum kaydedicisi olarakta bilinen bu kaydediciler sonucun sıfır, pozitif veya negatif olduğunu veya işlemin sonucunda elde üretilip üretilmediği gibi birçok önemli veriyi bitsel olarak programcıya bildirirler.

➢ Index Kaydedicileri

Bu kaydedicilerin E ile başlayanlar 32 bitlik programlarda, diğerleride 16 bitlik programlarda kullanılır. Hepsi de verilerin offset adreslerini tutmada kullanılır. SP ve BP, SS kaydedicisi ile birlikte SI ve DI, DS ve ES kaydedicileri ile birlikte hafıza adreslerine erişmek için kullanılır.

Segment ve Ofset Adresleri

14F3:0102 bir adres örneğidir ve " : " solunda yer alan sayılar bellek gözeneğinin segment adresini, sağında yer alan sayılar ise ofset adresini belirtir.

Real mod'da hafıza 64Kb'lık segmentlerden oluşur ve bu segment içindeki her byte'a erişmek için offset adresleri kullanılır. Bu durumda hafızanın tamamı bir kitap, segmentler sayfalar ve adreslenebilen her byte ise satır olara düşünülebilir.
Mikroişlemci hafızanın herhangi bir konumuna erişirken segment adresini segment kaydedicilerinden, offset adresini de şayet erişilecek olan bir komut kodu ise IP kaydedicisinden alır. Erişilecek olan veri (değişken, karakter vs.) ise verinin türüne göre offset adresini tutacak olan kaydedici değişir, örneğin

erişilecek veri stack'ta (yığın bellekte diyebilirsiniz) ise buradaki verinin offset adresini SP (stack pointer) tutar.

Segment kaydedicileri 16 bitliktir, real mod'da offset adresleri index kaydedicilerinin 16 bitlik kısımlarında (SP,BP,SI,DI) veya 16 bitlik olan IP kaydedicisinde tutulur. Bu durum xxxx:xxxx formatında gösterilir. Örneğin 3456:76A3 real modda bir hafıza adresini gösterir, burada 3456 segment adresi iken 76A3 ofset adresidir. 3456:76A3 gibi bir adres gösteriminde 3456'yı sayfa numarası olarak düşünürsek bu sayfada 0000-FFFF arasında 65536 adet offset adresi (satır) mevcuttur diyebiliriz ve bu konumların her biri 1 byte'a denk gelir.

Hafızanın gerçek yapısını göz önüne alırsak örneğin 8086 işlemcisi en fazla 1MB lık hafızayı adresleyebilir fakat xxxx:xxxx şeklinde bir gösterim 4GB'lık bir alana denk gelir (FFFF kere FFFF). Hal böyle olunca işlemci real modda çalışırken gerçek adresi bulmak için bir hesaplama yapar çünkü gerçekte hafıza real mod'da 00000H-FFFFFH arasında anlamlıdır.

Assembly Temel Komutları

- **Giriş**

- **Yorumlar / Açıklama Satırı**
- **Code Segment (CS)**
  - **Aritmetik Komutlar**
- **ınc**
- **dec**
- **add**
- **sub**
- **mul**
- **ımul**
- **dıv**
- **ıdıv**
- **neg**
  - **Mantıksal Komutlar**
- **and**
- **or**
- **xor**
- **test**
- **not**
  - **Koşullar**
- **cmp**
- **jmp**
  - ✓ **je**
  - ✓ **jne**
  - ✓ **jg**
  - ✓ **jge**
  - ✓ **jl**
  - ✓ **jle**
    - **Döngüler**
- **djnz**
- **loop**

▪ Giriş

➢ Yorumlar / Açıklama Satırı

Açıklamalar program satırlarının başına noktalı virgül konularak yapılır. Açıklama satırları assembler tarafından dikkate alınmaz. Program içinde daha detaylı bilgi vermek, kullanılan komutları izah etmek için kullanılır.

Örnek:

```
; MOV ES, AX    //Bu komut başta " ; " işareti ile yazıldığı için yorum satırı haline gelmiştir.
```

➢ Code Segment (CS)

Code segment (CS) ("text segment", executable code)

Hafızada, talimat kodlarını saklayan bir alan tanımlar, bu sabit bir alandır.

```
Section .text
   global _start

_start:
   ...
```

▪ Aritmetik Komutlar

➢ Inc

Bir yazmaçtaki değeri bir artırır.

Örnek:

```
inc   al
```

Burada al'nin önceki değerini 5 sayarsak artık al'nin değeri 6 olur.

➢ Dec

Bir yazmaçtaki değeri bir azaltır.

Örnek:

```
inc   al
```

Burada al'nin önceki değerini 5 sayarsak artık al'nin değeri 4 olur.

➢ Add

Herhangi bir yazmaçtaki değerle herhangi sabit bir değer veya yazmaçtaki değeri toplayıp sonuç ilk belirtilen yazmaca koyulur.

Örnek:

```
add   al,1
add   al,cl
```

Birincisinde al ile 1 toplanıp sonuç al'ye atanır. Yani sonuçta al bir artırılmış olur. İkincisinde ise al ile cl toplanıp sonuç al'ye atanır. Yani sonuçta al'ye cl eklenmiş olur. Add komutu argüman olarak yalnızca eşit kapasitedeki yazmaçları alabilir. Örneğin aşağıdaki komut hatalıdır.

```
add   ax,cl
```

➢ Sub

Herhangi bir yazmaçtan sabit bir değeri ya da başka bir yazmaçtaki değeri çıkarır.

Örnek:

```
sub   al,10
sub   ax,cx
```

Birincisinde al yazmacındaki değer 10 eksilir. İkincisinde de ax yazmacındaki değer cx'teki değer kadar eksilir. Sub komutu da tıpkı add gibi eşit kapasiteli yazmaçlarla işlem yapar.

➢ Mul

Belirtilen yazmaçtaki değerle al yazmacındaki değeri çarpıp sonucu ax yazmacına koyar.

Örnek:

```
mov  ax, 5     ; ax = 5
mov  cx, 10    ; cx = 10
mul  cx        ; ax = ax * cx
```

Burada ax yazmacındaki değerle cx yazmacındaki değer çarpılıp sonuç ax yazmacına konur.

➢ Imul

Tam anlamadım ☹ ama 8 bit veya 16 bitlik değerlerin çarpımlarının sonucunu 32 bitte tutmayı sağlıyor.Ayrıca (-,+) işaretlerini tutuyor.

Örnek:

```
mov     eax ,  9          ; eax = 9
mov     edx ,  66666667h   ; edx = 0x66666667
imul    edx              ; edx: eax = 0x66666667 * 9 = 0x39999999f
                         ; => edx = 0x3
                         ; => eax = 0x9999999f
```

➢ Dıv

Bölme işlemini gerçekleştirir.

➢ Idıv

Bölme işlemini 32 bitlik ve işaret vererek yapıyor ayrıca bölmede ondalık kısım için bişeler oluyor.Yine tam anlamadım ☹

➢ Neg

Değeri “-1” ile çarpar. Eğer değer pozitif ise negatife döner, negatif ise pozitif olur.

```
mov    al , 1
neg       al ; al içindeki 1 değeri -1 e dönüştü.
```

▪ Mantıksal Komutlar

➢ And

AND komutu, bitsel AND işlemini gerçekleştirir. Her iki işlenenden eşleşen bitler 1 ise, bitsel AND işlemi 1 döndürür, aksi takdirde 0 döndürür.

➢ Or

OR komutu, bitsel VEYA işlemini gerçekleştirir. Bir veya her iki işlenenden eşleşen bitler bir ise, bitsel VEYA operatörü 1 değerini döndürür. Her iki bit de sıfırsa 0 döndürür.

➢ Xor

XOR komutu, bitsel XOR işlemini uygular. XOR işlemi, yalnızca işlenenlerden bitler farklıysa, sonuçtaki biti 1 olarak ayarlar. Eğer işlenenlerden bitler aynı ise (her ikisi de 0 veya her ikisi de 1), elde edilen bit 0 olur.

➢ Test

TEST komutu AND işlemiyle aynı şekilde çalışır, ancak AND komutunun aksine ilk işleneni değiştirmez. Dolayısıyla, bir kayıttaki bir sayının çift mi yoksa tek mi olduğunu kontrol etmemiz gerekirse, bunu orijinal sayıyı değiştirmeden TEST talimatını kullanarak da yapabiliriz.

➢ Not

NOT komutu bitsel NOT işlemini uygular. NOT işlemi, bir işlenendeki bitleri tersine çevirir. İşlenen bir kayıtta veya bellekte olabilir.

▪ Koşullar

➢ jmp

JMP, programı belirtilen etiketin olduğu yere dallandırmakta ve program buradan çalışmaya devam etmektedir. Yani koşul için kullanılmaz fakat jmp ile aynı görevi koşullu yapan komutlar aşağıdadır.

(Go-to gibi)

- ✓ **je**
- ✓ **jne**
- ✓ **jg**
- ✓ **jge**
- ✓ **jl**
- ✓ **jle**

Bu komutlar koşullu bir şekilde zıplama işlemini gerçekleştiriyor. Genellikle cmp komutu ile kullanılır.

- jg → büyüktür.
- jl → küçüktür.
- je → eşittir.
- jne → eşit değilse.
- jge → büyük eşittir.
- jle → küçük eşittir.

➢ cmp

İki yazmaçtaki değeri ya da bir yazmaçtaki değerle sabit bir değeri kıyaslar. Tek başına kullanmak bir işe yaramaz bu yüzden koşullu zıplama komutları ile kullanılır. (if gibidir)

Örnek:

```
14F7:100 cmp   al,bl
14F7:102 jl    10e
```

Bu örnekte cmp kıyaslama yapar ardından koşullu zıplama komutumuz olan jl küçük ise anlamına gelir ve yukardaki karşılaştırmada al bl den küçük ise 10e ye zıplar ve akış oradan devam eder.

Döngüler

djnz

DJNZ komutu, montajcıda bir for döngüsü oluşturmanın en basit yoludur. DJNZ "azaltma, sıfır değilse atlama" anlamına gelir.

```
    LD   B, 5   ; B 5 olarak tanımlanmış
Loop:
    INC A       ; A ise döngü boyunca 1 er artacak
    DJNZ Loop   ; B değerini 1 er azaltır ve ardından 0 mı diye kontrol eder
                                  0'a ulaştığı zaman döngü bitmiş olucak.
    ...         ; döngüden sonraki kodlar.
```

Loop

Komutun argümanındaki ofset adresi ile loop komutunun bulunduğu satır arasında cx'in değeri kadar gidip gelinir. Yani bir döngü kurulmuş olur.

Örnek:

```
14F7:100 mov   cx,5
14F7:103 inc   al
14F7:105 add   dl,al
14F7:107 loop  103
14F7:109 int   20
```

Burdaki kod "14F7:100" ile "14F7:107" arasındaki komutları "mov cx,5" yani cx 5 olarak tanımlanmış olduğu için 5 kere çalışır.

Örnek2:

```
mov ECX,10
l1:
<loop body>
loop l1
```

Bu örnekte loop komutu label ile kullanılmış ve döngü 10 kere dönücek.

Notlar ve Eklenecekler

- **Int** önceden yazılmış bir servisi çalıştırmak için kullanılır. Windows'un içine gömülü çeşitli int servisleri bulunmaktadır. int 21 servisinin 9. fonksiyonu ekrana yazı yazdırmaya yarar.

  Bu fonksiyon dx yazmacındaki ofsetten itibaren $ karakterine kadar olan bütün karakterleri ekrana yazdırmaya yarar. Int 21, fonksiyonunu ah yazmacından alır. Int 20 servisi programı sonlandırır.

- **Db** talimatı ise bellek gözeneklerine komut değil de karakter yazılmasını sağlar.

Operands

Bir x86 komutunda sıfır ila üç işlenen olabilir. İşlenenler virgül (,) (ASCII 0x2C) ile ayrılır. İki işlenenle ilgili talimatlar için, ilk (sol el) işlenen **kaynak** işlenen ve ikinci (sağ el) işlenen **hedef** işlenendir (yani, **kaynak** -> **hedef**).İşlenenler **anında** (yani satır içi bir değeri değerlendiren sabit ifadeler), **kayıt** (işlemci numarası kayıtlarındaki bir değer) veya **bellek** (**bellekte** depolanan bir değer) olabilir. Bir **dolaylı** işlenen fiili işlenen değerin adresini içerir. Dolaylı işlenenler, işlenenin bir yıldız işareti (*) (ASCII 0x2A) önekiyle belirtilir. Sadece atlama ve çağrı talimatları dolaylı işlenenleri kullanabilir.

- **Anında** işlenenlere dolar işareti ($) gelir (ASCII 0x24)
- **Kayıt** adlarının önüne yüzde işareti (%) gelir (ASCII 0x25)
- **Bellek** işlenenleri ya bir değişkenin adıyla ya da bir değişkenin adresini içeren bir yazmaçla belirtilir. Değişken adı, değişkenin adresini belirtir ve bilgisayara bu adresteki bellek içeriğine başvurmasını bildirir. Bellek referansları şu sözdizimine sahiptir: *segment* : *ofset* (*taban* , *dizin* , *ölçek*) .
  - *Segment* , x86 mimari segment kayıtlarından herhangi biridir. *Segment* opsiyoneldir: belirtilirse, bu ayrılmalıdır *ofset* iki nokta üst üste (:). Eğer *kademeli* atlanırsa, değeri % ds (varsayılan segment yazmacı) varsayılır.
  - *Ofset* , istenen bellek değerinin *segmentinden* yer değiştirmedir . *Ofset* isteğe bağlıdır.
  - *Taban* ve *dizin* , genel 32 bit sayı kayıtlarından herhangi biri olabilir.
  - *Ölçek* , işlenenin adresini belirtmek için *tabana* eklenmeden önce *dizinin* çoğaltılacağı bir faktördür . *Ölçek* ise 1, 2, 4, veya 8 değerine sahip olabilir *ölçekli* belirtilmemişse, varsayılan değer 1'dir.

➢ Verables(Değişkenler)

Burada, *değişken adı* her depolama alanı için tanımlayıcıdır. Montajcı, veri segmentinde tanımlanan her değişken adı için bir ofset değerini ilişkilendirir.

Temel tanımlama yöntemleri aşağıdaki tabloda gösterilmiştir.

Direktif	Amaç	Depolama alanı
DB	Bayt Tanımla	1 bayt ayırır
DW	Kelimeyi Tanımla	2 bayt ayırır
DD	Doubleword'ü tanımlayın	4 bayt ayırır
DQ	Dört Kelimeyi Tanımla	8 bayt ayırır
DT	On Bayt Tanımla	10 bayt ayırır

➢ Labels(Etiket)

- Sembolik Etiketler

Bir **sembolik** bir plak oluşur **tanımlayıcısı** (ya da **sembol** iki nokta üst üste (:) (ASCII 0x3A) takip eder). Sembolik etiketler sadece bir kez tanımlanmalıdır. Sembolik etiketlerin **genel** kapsamı vardır ve nesne dosyasının sembol tablosunda görünür.Nokta (.) (ASCII 0x2E) ile başlayan tanımlayıcılara sahip sembolik etiketlerin **yerel** kapsamı olduğu kabul edilir ve nesne dosyasının sembol tablosuna dahil edilmez.

- Sayısal Etiketler

Bir **sayısal** etiket dokuz (9) arasında bir değer olarak sıfır (0) tek bir basamak oluşur (:) iki nokta üst üste. Sayısal etiketler yalnızca yerel başvuru için kullanılır ve nesne

dosyasının sembol tablosuna dahil edilmez. Sayısal etiketlerin kapsamı sınırlıdır ve tekrar tekrar tanımlanabilir.

➢ Instruction (required)

b -- Bayt (8 bit)
w -- Kelime (16 bit)
l -- Uzun (32 bit) (varsayılan)
q -- Quadword (64 bit)
l -- ("uzun") Komut işlenenleri 64 bit
s -- ("kısa") Komut işlenenleri 32 bit

➢ Proc Tanımı (Prosüdür)

Bir yordamı tanımlamak için sözdizimi aşağıdadır.

```
proc_name:
   procedure body
   ...                    ; Yordam içindeki kodlar
   Ret                    ; Yordamın sonunu belirtiyor.

   ...                    ; Yordam dışındaki kodlar
CALL proc_name
```