80X86 Sembolik Makine Dili
Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

Güncelleme Tarihi: 13/10/2020

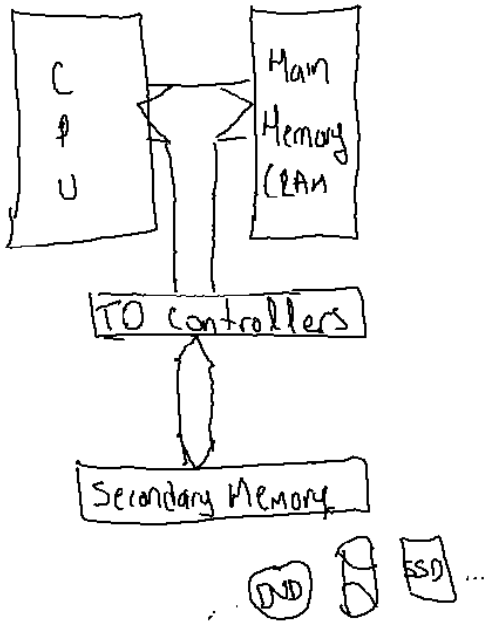
---

1. Temel Bilgisayar Mimarisi

Bu bölümde giriş niteliğinde bilgisayar mimarisi temel düzeyde ele alınacaktır.

1.1. Bilgisayar Mimarisinin Temel Bileşenleri

Tipik bir bilgisayar sistemini oldukça basitleştirerek bir blok diyagramı biçiminde aşağıdaki gibi gösterebiliriz:

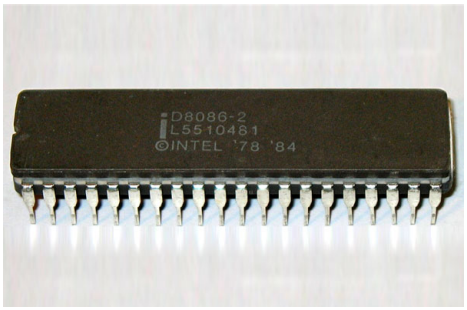


Bilgisayarlardaki asıl hesaplama işlemleri CPU tarafından yapılmaktadır. CPU Merkezi (Central) bir işlemcidir. Aslında bir bilgisayar sisteminde merkezi işlemcinin dışında başka işlemciler de vardır. Komutları alarak anlamlandırıp onları çalıştıran elektronik devrelere işlemci (processor) denilmektedir. Bir bilgisayar sisteminde yerel işlemlerden sorumlu pek çok yardımcı işlemci de vardır. CPU tüm bu yardımcı işlemcilere de ne yapması gerektiğini söylediği için ona "merkezi" işlemci denilmektedir.

Eskiden bilgisayarın ilk devirlerinde CPU'lar vakum tüplerle yapılıyordu. Daha sonra tarnsistörler icat edilince 50'li yıllarda CPU'lar transistörlerle yapılmaya başlanmıştır. Nihayet 70'lerle birlikte artık CPU'lar tek bir chip biçiminde entegre devre (integrated circuit) olarak imal edilmiştir. CPU'ların entegre devre biçiminde imal edilmiş biçimlerine mikroişlemci (microprocessor) denilmektedir. (Yani CPU mikroişlemcilerin daha kavramsal bir ismidir.)

CPU ile elektriksel olarak bağlantılı belleklere “Ana Bellekler (Main Memories)” ya da “Birincil Bellekler (Primary Memories)” denilmektedir.

Bir mikroişlemci elektronik bir devre olarak uçlara (pin'lere) sahiptir. Bu uçlar onun başka elektronik birimlere bağlanması için kullanılır. Örneğin Intel 8086 (elimizdeki PC'lerin ilk örneklerinde kullanılan işlemci) işlemcisinin fiziksel görünümü şöyledir:

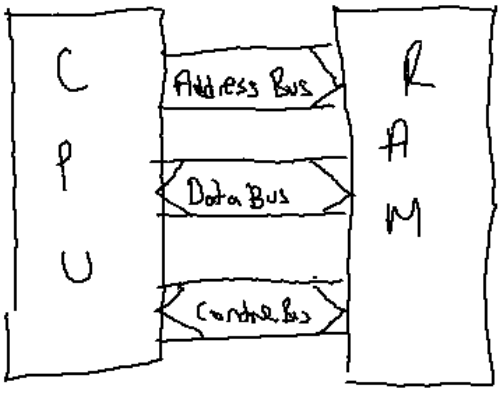


CPU'ların uçları bilgilerin dijital olarak (yani binary biçimde) aktarılacağı biçimde gerçekleştirilmiştir. Yani CPU'nun uçları analog uçlar değildir. Bu uçlardaki gerilim belli bir düzeydeyse mantıksal 1 olarak (örneğin tipik 5 V), yine bir düzeydeyse mantıksal 0 olarak (örneğin 0V) değerlendirilir. Böylece CPU dış dünyayla “binary düzeyde” haberleşmektedir. Bu biçimde dijital devre elemanları birbirlerine bağlanarak daha yetenekli bir organiszyon oluşturabilmektedir.

RAM'ler de entegre devre biçiminde üretilen ve uçları olan elektronik birimlerdir. Mikroişlemcinin bazı uçları RAM'le bağlantı için kullanılır. Yani Mikroişlemci ile RAM arasında binary düzeyde fakat elektriksel bir bağlantı söz konusudur. Mikroişlemci ile dış dünya arasındaki bağlantı uçlarına “yol (bus)” denilmektedir. Tipik olarak bir mikroişlemci için üç tür yol (yani uç grubu) söz vardır:

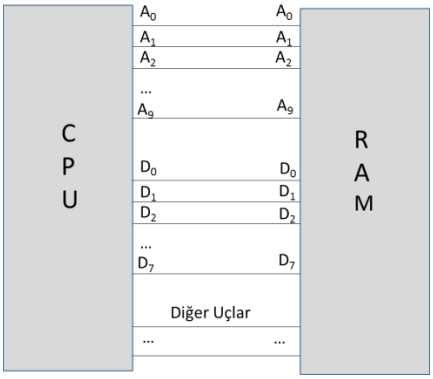
1) Adres Yolu (Address Bus)
2) Veri Yolu (Data Bus)
3) Kontrol Yolu (Control Bus)

Bu bilgi eşliğinde CPU ile RAM arasındaki bağlantıyı aşağıdaki gibi detaylandırabiliriz:



RAM de aslında akıllı bir birimdir. RAM'deki her bir byte'ın bir adresi vardır. Buna ilgili byte'ın fiziksel adresi denir. RAM devresinin de adres seçmek için, veri aktarmak için uçları vardır. RAM devrelerinin çalışmasını bir örnekle açıklayabiliriz.Eelimizde 1K'lık bir RAM entegresi olduğunu varsayalım. Bu RAM'deki herhangi bir byte'la ilgilendiğimizi RAM'e kaç uçla iletebiliriz? Yanıt: 10 uçla (2^{10} = 1K). Buna RAM'in adres uçları denir ve genellikle A_0, A_1, ... A_9 biçiminde gösterilir. Bu uçlara dışarıdan 5V-0V biçiminde gerilim uygulandığında RAM ilgili adresteki byte'ı seçer. Örneğin biz RAM'in 512'inci byte'ı ile ilgilenmek isteyelim. Bu durumda A_0-A_9 uçlarına 512 sayısını ikilik sistemde elektriksel olarak uygularız. Pekiyi söz konusu bu RAM'e ya da bu RAM'den 1 byte bilgi aktarımı için kaç uç gerekir? Yanıt: 8 uç (1

byte 8'bittir). İşte bu uçlara da RAM'in veri uçları denir ve genellikle D_0, D_1...D_7 biçiminde gösterilir.



Böylece dış dünyadan RAM'e 1 byte bilgi yazmak şöyle bir protokolle yapılabilmektedir:

1) Yazacağın byte'ın adresini RAM'in A0-A9 uçlarına elektriksel işaret olarak uygula
2) Yazma yapmak istediğini RAM'e bir kontrol ucu ile (R/W uçları) elektriksel olarak bildir.
3) O adrese yazacağın byte'ı da D_0-D_7 uçlarına elektriksel olarak uygula

Benzer biçimde RAM'in bir adresinden bilgi okumak için de şöyle bir protokol uygulanabilmektedir:

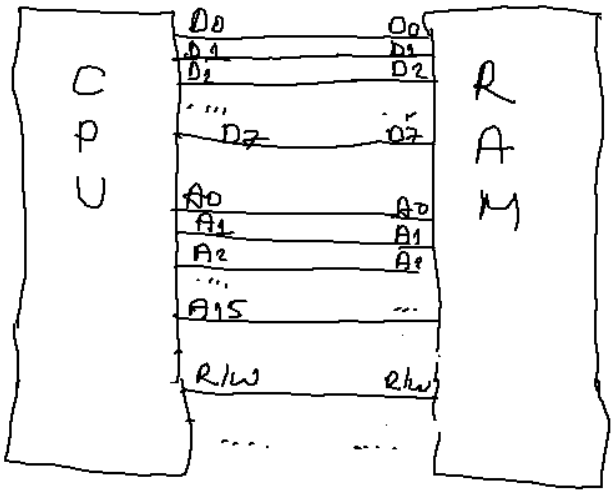
1) Okunacak byte'ın adresini RAM'in A_0-A_9 uçlarına elektirksel olarak uygula
2) Okuma yapmak istediğini RAM'e bir kontrol ucu ile (R/W uçları) elektriksel olarak bildir.
3) Biraz bekle ve RAM'in D_0-D_7 uçlarını oku

Şimdi bir CPU'nın c = a + b işlemini nasıl yaptığına bakalım:

Burada a + b işlemi CPU içerisinde yapılacağına göre önce a ve b'nin CPU'ya çekilmesi gerekir. Bu durumda CPU RAM'den a'yı b'yi isteyecektir. a'nın adresini CPU RAM'in adres uçlarına elektriksel olarak uygular. Sonra RAM'in data uçlarından a'yı alır kendi içerisine çeker. Sonra aynı şeyi b için de yapar. Kendi devrelerinde bu iki değeri toplar. Sonucu elde eder. Onu da yazacağı yerin adresini RAM'e vererek sonra da değeri data uçlarına uygulayarak gerçekleştirir.

Pekiyi RAM kendisinden okuma mı yapılacağını yoksa kendisine yazma mı yapılacağını nasıl anlamaktadır? İşte bunun için de RAM'in R/W biçiminde uçları da vardır. CPU RAM'in adres uçlarına adresi yerleştirirken aynı zamanda yapacağı işlemi de elektriksel olarak R/W uçlarıyla bildirir. RAM'de R/W ucuna bakarak okuma mı yoksa yazma mı yapıldığını anlar.

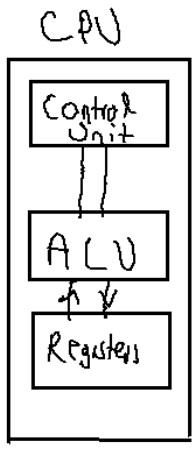
Nasıl RAM'in adres ve data uçları varsa CPU'ların da adres ve data uçları vardır. CPU'nun adres uçları RAM'in adres uçlarına CPU'nun data uçları RAM'in data uçlarına bağlanmaktadır. Örneğin 8 bitlik, 64K belleği adresleyebilen bir mikroişlemci ile RAM bağlantısı aşağıdakine benzemektedir:



Pekiyi mikroişlemci ve RAM fiziksel olarak iki ayrı entegre devre olmak zorunda mıdır? Esnekliği artırmak için parçaların ayrı ayrı üretilmesi daha anlamlıdır. Böylece örneğin CPU'yu satın alan ona istediği marka ve özellikte RAM bağlayabilir ve RAM'i yükseltebilir. Fakat bazı uygulamalarda CPU ile RAM'in ve hatta diğer bazı birimlerin tek bir entegre devre biçiminde üretildiği görülmektedir. Bu tür üretimlere SoC (System on Chip) denilmektedir. Örneğin Raspberry Pi karı üzerinde SoC biçiminde (Broadcom 2835 ya da Broadcom 2836) CPU, RAM ve GPU tek bir entegre devre olarak üretilmiştir.

Mikroişlemci kavramı ile mikrodenetleyici (microcontroller) kavramları da bazen birbirlerine karıştırılmaktadır. Mirodenetleyiciyi en güzel "single chip computer" sözcükleri tanımalamaktadır. Mikrodenetleyici kendi içerisinde CPU'su olan, RAM'i olan, IO birimleri olan ve hemen dış dünyayla bağlantı kurabilecek biçimde uçlara sahip olan entegre devrelerdir. Genellikle mikrodenetleyicilerin hızları düşük ve bellek kapasitesi küçük olma eğilimindedir. Ancak fiyatları çok daha ucuzdur. Bu nedenle düşük fiyatlı gömülü sistemlerde mikrodenetleyiciler çok tercih edilirler. Mikrodenetleyicilerin güç gereksinimleri diğer CPU'lara göre daha azdır.

SoC kavramı ile mikrodenetleyici kavramları git gide birbirlerine yaklaşmaktadır. Ancak SoC terimi tek başına kullanılabilen bir entegre devreyi belirtmez. Ayrıca SoC'ların çoğunda ikincil bellekler dışarıdadır. Fakat mikro denetleyiciler komple her şeyi içerisinde olan hemen devrelere bağlanabilecek entegre devrelerdir.



1.2. CPU'ların İçsel Yapısı ve Çalışma Mekanizmaları

Bir CPU'nun kabaca blok diyagramı şöyledir:

ALU (Arithmetic Logic Unit) aritmetik ve karşılaştırma işlemlerinin ve diğer işlemlerin yapıldığı mantık devrelerinin bulunduğu CPU bölümüdür. Örneğin a = b + c gibi bir işlemde b + c işlemi CPU içerisindeki ALU tarafından yapılmaktadır. Yazmaçlar (registers) ALU'nun iskeleleri (ya da portları) görevindedir. Yani ALU işleme sokulacak değerleri yazmaçlardan alır ve sonucu yazmaçlara yerleştirir.

Aslında CPU ne yapacağını da (yani program komutlarını da) birincil belleklerden (neden RAM dememiş olabiliriz?) almaktadır. Aslında bunun için iki mimari kullanılmaktadır. Eğer program komutları ile data'lar (yani a, b, c'ler) aynı chip üzerindeyse başka bir deyişle program komutları ile data'lar aynı chip'te bulunuyorsa bu mimariye "Von Neumann" mimarisi, eğer program komutları ile data'lar farklı chiplerdeyse bu mimariye de "Harward" mimarisi denilmektedir. Bugün kullandığımız bilgisayarların hemen hepsi "Von Neumann" mimarisine uygundur. Fakat örneğin PIC mikrodenetleyicileri Harward mimarisini kullanırlar.

Bir CPU kabaca şöyle çalışmaktadır: CPU'nun içerisindeki bir yazmaca "PC (Program Counter)" ya da IP (Instuction Pointer)" denilmektedir. CPU bu yazmacın gösterdiği adresten komutu çekip çalıştıracak biçimde tasarlanmıştır. CPU'nun çalıştırdığı komutlara makina komutları (machine command, machine instruction) denir. Her mikroişlemcinin makina komut kümesi (instruction set) birbirlerinden farklı olabilmektedir. Makina komutları ikilik sistemde byte toplulukları biçimindedir. Makina komutları CPU tasarlanırken tasarımcı firma tarafından tasarlanır. Programcılar onları değiştiremez. Bir mikroişlemcinin tüm işlem yeteneği onun komut kümesiyle sınırlıdır. Makina komutları bir mikroişlemciye yaptırılacak en yalın işlemleri tanımlar. Bir sembolik makina dili programcısının ilk bilmesi gereken şeylerden biri çalıştığı işlemcinin makina komut kümesidir.

İşlemci makina komutunu RAM'den çekince önce onu anlamlandırır. Yani bu komut ne komutudur? Makşina komutunun RAM'den çekilmesi sürecine İnglizce "fetch", onun anlamlandırılması sürecine "decode" denilmektedir. Pekiyi makina komutları kaç byte uzunluktadır? Tarihsel gelişim süreci içerisinde 1 byte, 2 byte, 4 byte ve hatta 8 byte uzunluğunda makina komut kümelerinin kullanıldığını görmekteyiz. Pekişyi her makina komutu aynı uzunlukta mıdır? İşte eskiden belleğin daha verimli kullanılması için komutlar farklı uzunluklarda tasarlanmıştır. Ancak modern sistemlerde artık komutların hepsi aynı uzunlukta olma eğilimindedir. Komutların hepsinin aynı uzunlukta olması "decode" işlemini kolaylaştırmaktadır. Intel sisteminde komutlar farklı uzunluklardadır fakat örneğin ARM işlemcilerinde komutların hepsi aynı uzunluktadır.

Mikroişlemci komutun ne komutu olduğunu anladıktan sonra onu ALU birimine vererek çalıştırır. Bu aşamaya da İngilizce "execute" denilmektedir. Komut çalıştırıldıktan sonra artık mikroişlemci sonraki komutun çalıştırılması için PC yazmacını komut uzunluğu kadar artırır. Bu işlemler bu biçimde hep devam eder. Görüldüğü gibi çalışma "fetch-decode-execute" döngüsü biçiminde devam eder. Mikroişlemcilerin çalışma mekanizması bir pseudo kodla aşağıdaki gibi götserilebilir:

PC = başlangıç adresi

```
PC = starting address;
for (;;) {
```

```
    instr = memory[PC];

    PC = PC + 1;
    instr_type = decode(instr);
    data_loc = find_data(instr, instr type);
    if (data_loc >= 0)
        data = memory[data_loc];
    execute(instr_type, data);
}
```

Buradaki PC program sayacı yazmacını belirtir. Görüldüğü gibi PC'nin gösterdiği yerden komut byte'ları instr isimli değişkene çekilmiştir. decode fonksiyonu decode işlemini temsil etmektedir. Eğer komut RAM'e erişmeyi gerektiriyorsa bu kez RAM'den komutun ilgili operandı çekilmiştir. Nihayet execute işlemi de komutun çalıştırılmasını temsil eder.

Pekiyi bir mikroişlemci nasıl çalışmaya başlar? Yanıt: İşlemcinin GND ve Vcc uçlarına güç kaynağı uygulanarak. Pekiyi mikroişlemci reset edildiğinde PC yazmacı hangi değerdedir? Yani mikroişlemciyi biz reset ettiğimizde RAM'in neresindeki program çalışmaya başlayacaktır? İşte mikroişlemciyi reset ettiğimizde çalışmanın başlatıldığı adrese reset vektörü denilmektedir. Her mikroişlemcinin reset vektörü üretici firmaya bağlı olarak değişebilmektedir. O halde mikroişlemci reset edildiğinde bir programın kalıcı bir bellekte (non-volatile memory) reset vektöründe hazır olarak bulunması bulunması gerekir. Eskiden bu amaçla EPROM bellekler kullanılıyordu. Artık EPROM'lar EEPROM tarzı belleklere yerini bırakmıştır. Normal RAM'ler güç kaynağı kesildiğinde içindeki bilgiyi kayberderler. Güç kaynağı kesildiğinde içindeki bilgiyi kaybetmeyen belleklere aile olarak ROM (Read Only Memory) denilmektedir. Bu tür bellekler tarihsel olarak ROM, PROM, EPROM ve nihayet EEPROM (E^2PROM) biçiminde evrimleşmiştir. EEPROM'ların hem içerisine bilgi yerleştirilebilmekte hem de güç kaynağı kesilince içindeki bilgiler kaybolmamaktadır. İşte örneğin elimizdeki PC'lerde reset vektöründe EEPROM tarzı bir bellek bulunur. Buraya üretici firma bir kod yerleştirmiştir. İşletim sisteminin yüklenmesi buradaki program (bootstrap program) tarafından başlatılır.

Pekiyi EEPROM ve RAM nasıl beraber mikroişlemciye bağlanmıştır? Aslında bir mikroişlemciye teorik olarak bağlanabilecek bellek miktarı mikroişlemcinin adres uçlarının sayısı ile ilişkilidir. Örneğin 8086 mikroişlemcisinin 20 tane adres ucu vardı. Bu durumda 8086 işlemcisine en fazla 1MB bellek bağlanabiliyordu. Bir mikroişlemciye teorik olarak bağlanabilecek bellek miktarına mikroişlemcinin adres alanı (address space) denilmektedir. Şüphesiz biz bir mikroişlemcinin teorik adres alanı kadar fiziksel belleği ona takmak zorunda değiliz. Örneğin Intel X64 ve AMD64 işlemcilerinin teorik adres alanı çok büyüktür. Fakat biz bu işlemcilere 1GB bellek takarak da onları kullanabiliriz. Ayrıca mikroişlemcinin teorik adres alanına takacağımız bellek çipleri de ardışıl olmak zorunda değildir. Örneğin biz isetersek işlemcinin adres alanının (bunun anlamı olabilir ya da olmayabilir) ilk 1GB'sine bir RAM bağlayıp sonraki 10GB'sini boş bırakıp sonraki 1GB'sine de yeniden RAM bağlayabiliriz. Ya da adres alanının bazı yerlerine EEPROM bağlayıp bazı yerlerine de normal RAM bağlayabiliriz. Örneğin Intel işlemcileri reset edildiğinde çalışma (reset vektörü) teorik adres alanının sonundan başlamaktadır. Oraya da EEPROM bir bellek yerleştirilmiştir. O belleğin içerisindeki kod ve datalara BIOS (Basic Input Output System) denilmektedir.

Pekiyi mikroişlemciyi reset ettiğimizde BIOS'taki kod ne yapar? Bu tür kodların ilk yaptığı şeyler mikroişlemciye hangi kaynakların (örneğin ne kadar RAM'in) bağlı olduğunu tespit etmektir. Bu işleme POST (Power On Self Test) denilmektedir. Genellikle POST programları elde ettikleri sonucu RAM'de bir bölgeye yazarlar. İşletim sistemleri bunlardan faydalanabilmektedir.

1.3. Makine Komutları ve Mikro Kodlar

Bir işlemcinin ALU'su belli işlemleri yapacak biçimde tasarlanmıştır. İşlemcinin çalıştırabileceği en yalın komutlara makine komutları (machine command ya da machine instruction) denilmektedir. Her işlemcinin bir makine komut kümesi (instruction set) vardır. Derleyiciler (ya da makine dili programcıları) tüm programı o işlemcinin kabul edeceği makine komutlarına göre organize ederler.

Makine komutları işlemciyle yazılımlar arasındaki en aşağı seviyeli arayüzdür. İşlemcilerin makine komutlarının sayıları ve biçimleri birbirlerinden farklı olabilmektedir. Bazı işlemcilerde çok sayıda makine komutu bulunurken bazılarında daha az sayıda makine komutu bulunuyor olabilir. Bazı mikroişlemci aileleri aynı makine komut kümesini kullanıyor olabilirler. (Örneğin Intel ve AMD işlemcileri model numaraları ilerlese de aynı komut kümesini kullanmaya devam etmektedir. Tabii işlemciler ilerledikçe onlara özgü yeni komutlar da ekleniyor olabilir. Ancak geri doğru uyumluluk için eski komutların desteklenmesine devam edilmektedir.)

Makine komutları ikilik sistemde sayılar biçimindedir. Yukarıda da sözü edildiği gibi bazı işlemcilerde makine komutlarının byte uzunlukları hep aynıyken bazılarında farklı olabilmektedir. Bazı kaynaklarda makine komutlarının ikilik sistemdeki haline "operation code (kısaca opcode)" denilmektedir. Tabii ikilik sistemdeki her byte dizilimlerinin o işlemci için anlamlı bir makine komutu oluşturması garanti değildir. (Pek çok işlemci geçersiz bir makine komutuyla karşılaştığında içsel bir kesme oluşturmaktadır.) Şu an için dünyada yüzlerce farklı mikroişlemci ailesi vardır. Bunların makine komutları hep birbirinden farklı olma eğilimindedir. Dolayısıyla sembolik makine dili programlaması da genel olmaktan ziyade büyük ölçüde işlemciye özgüdür. Tabii bir ailenin sembolik makine dilini öğrenmek başka aileleri öğrenmede kavramsal bakımdan kolaylık sağlar. (Örneğin C bilen kişinin C#'ı kolay öğrenmesi gibi) Şu anda dünyada en fazla kullanılan iki mikroişlemci ailesi Intel 80X86 ve ARM aileleridir. ARM işlemcileri akıllı telefon, tablet ve diğer taşınabilir cihazlarda en tercih edilen ailedir.

Bazı mikroişlemci tasarımlarında herbir makine komutuna ALU'da ayrı bir matık devresi karşı gelmemektedir. Bu tür işlemcilerde ALU'daki mantık devreleri daha kompakt ve genel amaçlı olarak tasarlanmıştır. Bu i işlemcinin içerisinde ismine mikrokod denilen bir yazılım bulunur. Bu yazılım bellekten alınan (fetch edilen) komutu anlamlandırıp onu birden fazla mantık devresine sokarak çalıştırır. Mikrokod yazma faaliyetine mikroprogramlama (microprogramming) denilmektedir. Yani mikrokod temelinde bakıldığında aslında makine komutları en yalın komutlar değildir. Bunlar da parçalara ayrılıp ALU tarafından çalıştırılmaktadır. Pek çok işlemcide mikrokod dışarıdan "firmware update" mekanizamasıyla değiştirilebilmektedir.

Her türlü mikroişlemcide mikrokod mekanizması yoktur. Genel olarak mikrokod mekanizması CISC tabanlı işlemcilerde hala kullanılmaktadır. Örneğin Intel işlemcilerinde mikrokod mekanizması eskiden çok yoğun olarak kullanılıyordu. Fakat şimdilerde çok daha az kullanılmaktadır. RISC tarzı tasarımlarda mikrok mekanizması neredeyse hiç kullanılmamaktadır.

Mikrokod programlama büyük ölçüde üretici firma tarafından korunan bir bilgidir. Yani örneğin Intel resmi dokümanlarında kendi ürünlerinin mikrokodları hakkında bir bilgi vermemektedir. Mikrokod programlama sembolik makine dili programcısının etkili olabileceği bir alan değildir.

Mikrokod mekanizmasının önemli bir faydası da esnekliğin artırılmasıdır. Şöyle ki: Bu sayede bir mikroişlemcinin makine komutları değiştirilebilir, başka emülasyonlar işlemciye donanım düzeyinde yaptırılabilir. Komut eklemeleri daha kolay gerçekleştirilebilir. Ancak şüphesiz mikrokod çalıştırmanın performans üzerinde olumsuz bir etkisi vardır.

1.4. CISC ve RISC Mimarileri

Mikroişlemci tasarımında iki önemli mimari vardır: CISC (Complex Instruction Set Computing) ve RISC (Reduced Instruction Set Computing). İlk mikroişlemciler CISC mimarisine uygun tasarlanmışlardır. CISC ve RISC mimarileri arasındaki farklılıklar aşağıda özetlenmektedir. Tabii belli bir mikroişlemci her iki mimariden de özellikleri barındırabilir. (Örneğin Intel'in son dönem işlemcileri pek çok RISC özelliğini de barındırmaktadır. Bu nedenle CISC ve RISC mimarilerini var yok biçiminde değil bir derece biçiminde düşünmek daha uygundur:

1) CISC mimarisinde daha fazla makine komutu RISC mimarisinde daha az makine komutu bulunma eğilimindedir. RISC mimarisinde daha az komut daha huzlı ve etkin çalışacak biçimde mantık devreleriyle oluşturulmuştur. Dolayısıyla komutların hemen hepsi tek bir mantık devresiyle doğrudan çalıştırılır. Halbuki CISC mimarisinde mikrokod programlamayla karmaşık komutlar daha yalın parçalara ayrılarak çalıştırılır.

2) CISC mimarisinde makine komutları farklı uzunluklarda olma eğilimindedir. Halbuki RISC mimarisinde tüm makine komutları aynı uzunluktadır. CISC mimarisinde çok kullanılam makine kamutları az byte'la az kullanılan makine komutları çok byte'la ifade edilmeye çalışılmıştır. İlk zamanlar bunun iyi bir teknik olduğu sanılmışsa da daha sonraları bazı problemler göze çarpmıştır. Komutlar farklı uzunluklarda olursa genel olarak komutu alıp yorumlama (fetch ve decode) işlemi yavaş yapılmakatdır. Ayrıca komut seviyesinde pipeline mekanizması komutlar eşit uzunluktaysa daha etkin yapılabilmektedir.

3) RISC mimarisinde çok sayıda yazmaç (register) bulunma eğilimindedir. Halbuki CISC mimarisinde daha az sayıda yazmaç vardır. Ayrıca RISC mimarisinde her yazmaçla her şey yapılabilmektedir. Halbuki CISC mimarisinde bazı işlemler ancak bazı özel yazmaçlarla yapılabilmektedir.

4) RISC mimarisinde belleğe erişen makine komutları ile işlem yapan makine komutları biribirlerinden ayrılmıştır. Bu nedenle RISC işlemcilerine Load/Store işlemcileri de denir. RISC mimarisinde toplama, çıkartma, çarpma gibi tüm işlemler operandlarını yazmaçlardan alacak biçimde tasarlanmıştır. Halbuki CISC mimarisinde komutların bir operandı yazmaç iken diğer operandı bellek adresi olabilmektedir. Ayrıca RISC mimarisinde Load/Store komutları dışındaki jomutlar üç operandlı olma eğilimindedir. Halbuki CISC mimarisinde neredeyse tüm komutlar iki operandlıdır. Örneğin aşağıdaki gibi bir C program parçası olsun:

```
int a = 10, b = 20, c;
c = a + b
```

Bu işlemi CISC tabanlı bir mikroişlemcide aşağıdaki sembolik makine komutları ile yaptırabiliriz:

```
MOV reg1, a
ADD reg1, b
MOV  c, reg1
```

Aynı işlem RISC mimarisinde aşağıdaki gibi yapılabilmektedir:

```
MOV reg1, a
MOV reg2, b
ADD reg1, reg2, reg3
```

Burada CISC'teki ADD komutunun yapısıyla RISC'teki ADD komutunun yapısı arasındaki farka dikkat ediniz. CISC'te toplama yaparken operandlardan biri yazmaç diğeri bellek adresi olabilmektedir. Halbuki RISC'te her iki operandın yazmaç olması zounludur. Toplamda RISC'teki tasarımın daha faydalı ve etkin olduğu ispatlanmıştır. Bu nedenle artık yeni işlemcilerin hepsi RISC mimarisinde tasarlanmaktadır.

5) RISC işlemcileri pipeline işleminin etkinliğiyle ünlüdür. Pipeline işlemci,nin bir komutu yaparken

diğerleri üzerinde de bazı hazırlık işlemlerini yapabilmesi anlamına gelir. Şüphesiz Intel gibi CISC işlemcileri de pipeline mekanizmasına sahiptir. Ancak RISC tasarımı pipeline mekanizmasının daha etkin yapılabilmesine yol açmaktadır.

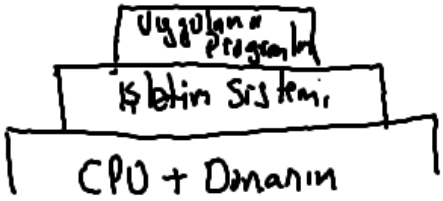
1.5. Bir Program CPU Tarafından Nasıl Çalıştılma Durumuna Getirilir?

Bir program yazılıp derlendikten sonra nasıl çalışma aşamasına getirilmektedir? Eğer çalıştırma ortamı bir işletim sistemine sahip olmayan bir ortamsa bizim programı bir biçimde reset vektöründen itibaren o ortamın belleğine yüklememiz gerekir. Tipik olarak mikrodenetleyici (microcontroller) ile çalışmalar bu biçimde yürütülmektedir.

Bir mikrodenetleyicinin içerisinde bir CPU'nun yanı sıra bir RAM ve EEPROM bellek de vardır. Bu EEPROM belleğe mikrodenetleyicinin program belleği denilmektedir. Bu ortamlarda programcı programını PC'ler üzerinde yazar. Onu derler ve "programlayıcı denilen bir devre yoluyla mikrodenetleyicinin içerisine yerleştirir.

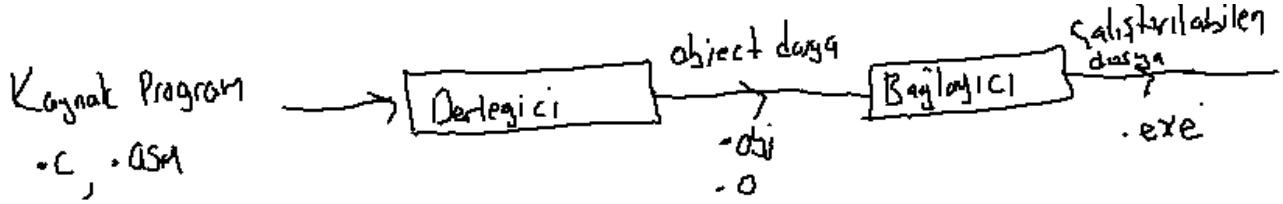
Anahtar Notlar: Eğer biz derleme işlemini derleyicinin çalıştığı CPU için değil de başka bir CPU için yapıyorsak bu derleme işlemine "çapraz derleme (cross compiling)", bu işi yapanderleyicilere de "çapraz derleyiciler (cross compilers)" denilmektedir. Örneğin biz PIC mikrodenetleyicileri için derlemeyi 80X86 tabanlı PC'lerde yapıyor olabiliriz. Fakat derleme işleminin sonucunda üretilen kod PIC'in makine kodlarıdır. O halde burada bir çapraz derleme faaliyeti söz konusudur.

Mikrodenetleyici dünyasının dışında biz genellikle bir işletim sisteminin üzerinde çalıyor durumdayızdır. (Hatta bazı mikrodenetleyiciler de işletim sistemi yüklenerek kullanılabilmektedir.)



Biz bir işletim sisteminin üzerinde çalışıyorsak programı yüklemek ve onu CPU'ya vermek tamamen artık işletim sisteminin bir görevi durumundadır. İşletim sistemlerinin programı yükleyip çalışır hale getiren kısmına kavramsal olarak yükleyici (loader) denilmektedir.

İşletim sistemi üzerinde çalışılan bir ortamda tipik olarak program bir editörle (ya da IDE ile) yazılır. Derleyici ile derlenerek amaç dosya (object file) oluşturulur. Bu amaç dosya da bağlama (link) işlemine sokularak çalıştırılabilen (executable) dosya elde edilir.



Pekiyi amaç dosyaların (object file) ve çalıştırılabilen dosyaların (executable file) içerisinde ne vardır? Amaç dosyalar kursumuzda daha ileride ele alınacaktır. Ancak çalıştırılabilen bir dosyanın içerisinde en azından derlenmiş ve ikilik sisteme dönüştürülmüş kodlar ve bu kodların kullandığı data'lar bulunmak zorundadır. Tabii bunların dışında çalıştırılabilen dosyaların içerisinde dosyanın işletim sistemi tarafından yüklenebilmesi için gereken başka birtakım meta data bilgiler de bulunmak zorundadır. Örneğin yükleyici programı RAM'e yükledikten sonra IP (ya da PC) yazmacının ilkdeğerini nasıl verecektir. Bilindiği gibi C'de programın akışı main gibi bir fonksiyondan başlar. İşte program akışının başlatılacağı adrese "enrty point" denilmektedir. Bu adres çalıştırılabilen dosyanın içerisinde belli bir yerde bulunur. Yükleyici de programı

yükledikten sonra IP (ya da PC) yazmacına bu değeri atar. Sonra da program kendi kendine çalışmaya devam eder. Tabii çalıştırılabilen dosyaların içerisinde "entry point" dışında daha gerekli olan pek çok yükleme bilgisi de bulunmaktadır.

Anahtar Notlar: Neredeyse her dosyanın (saf metin dosyaların dışında) bir dosya formatı vardır. Dosya formatları o dosyanın neresinde hangi bilgilerin olduğunu belirtir. Dosya formatlarının başında genellikle bir başlık kısmı (header) bulunur. Bu başlık kısmında meta data (yani dosya içerisindeki bilgileri tanıtan ve anlamlandıran bilgiler) alanı bulunmamkatdır. Bu durum .bmp, .jpeg, .xls gibi dosyalar için de .exe gibi çalıştırılabilen dosyalar için de böyledir. Bir dosya formatıyla karşılaştığımızda onun başlık kısmı bizim ilgimizi çekmeli ve orada hangi bilgilerin bulunduğunu merak etmeliyiz.

O halde çalıştırılabilen bir dosya kabaca şu biçimde çalıştırılma durumuna getirilmektedir:

1) Yükleyici çalıştırılabilen dosyanın başlık kısımlarına bakarak onun içindeki kod ve data bilgilerinin nerelerde olduğunu anlar. Çalıştırılabilen dosyanın kod ve data bölümlerini bellekte uygun bir yere yükler.

2) Yükleyici çalıştırılabilen dosyanın başlık kısımlarından programın "entry point" adresini belirler ve CPU'nun IP (ya da PC) yazmacına kodun başlangıç adresini yükler. Sonra da CPU'yu serbest bırakır.

1.6. Sembolik Makine Dili Programlamasına Neden Gereksinim Duyulmaktadır?

1) Sembolik makine dili ve bunun çevresindeki konular bilgisayar sistemlerinin çalışma biçimini aşağı seviyeli olarak öğrenmemize katkı sağlamaktadır. Yani biz gerçek anlamda hiç sembolik makine dilinde program yazmayacak olsak bile bu kurstaki bilgiler pek çok olayı yorumlayabilmemiz için bize zemin oluşturacaktır. Başka bir deyişle bu kurs bilgisayar sistemlerinin aşağı seviyeli çalışma biçimi hakkında "oldukça besleyici" bilgiler sunmaktadır. Bazı mekanizmalar ancak sembolik makine dili ve çevre konularını belli düzeyde bilmekle anlaşılabilir. (Örneğin virüslerin çalışma mekanizması, hacking işlemleri, debugger'lar vs gibi)

2) Bazı aşağı seviyeli işlemlerde mecburen sembolik makine dilleri kullanılmak zorundadır. Örneğin:

- Boot programları
- Firware'ler
- Virüs kodları
- İşletim sistemlerinin bazı kısımları (yükleyici, çizelgeleyici alt sistemler, aygıtların programlanması, aygıt sürücülerde vs.)
- Derleyiciler, debug programları ve bağlayıcı programların bazı kısımlarının yazımı

Pekiyi neden bu tür kodları biz C'de yazamıyoruz? İşte C derleyicisi bu tür kodlarda yetersiz kalmaktadır. Çünkü bu tür kodlarda programcının özel makine komutlarını seçerek ve etkin kullanması gerekir. C derleyicileri bu kadar özel kodları bizim için üretemezler.

3) Sembolik makine dili çok özel kodların hızlandırılması için de kullanılabilmektedir. Her ne kadar derleyicilerin kod optimzasyonları çok gelişmiş olsa da yine de iyi bir sembolik makine dili programcısı kodu çok daha etkin bir biçimde düzenleyebilmektedir.

4) İşlemcilerin çeşitli modelleri bulunabilmektedir. C derleyicilerinin çoğu bir ailenin en düşük elemanına yönelik kod üretmektedir. Bazen durumlarda sembolik makine dili programsıcı derleyicinin hiç kullanmak istemediği makine komutlarından faydalanabilmektedir.

5) Az yer kaplayan kodların sembolik makine dilinde özel bir ihtimamla yazılması gerekebilir. Her ne kadar C derleyicilerinin "size optimizasyonu" varsa da o kadar gelişkin değildir. Az yer kaplaması gereken kodlar özel makine komutları seçilerek sembolik makine dili programcıları tarafından daha etkin yazılabilir.

6) Bazı sistemlerde hiç C derleyicisi yoktur. Bu sistemlerde biz mecburen sembolik makine dilleri ile çalışırız.

Pekiyi C yerine hep sembolik makine dili kullansak olur mu? Bunun da bazı dezavantajları vardır:

1) Sembolik makine dili programlamada hata riski daha fazladır

2) Sembolik makine dilleri taşınabilir diller değildir. Yalnızca o sisteme özgü kodlamalar yapılabilmektedir.

3) Sembolik makine dilinde kod yazmak zahmetlidir ve daha uzun süre gerektirir

4) Sembolik makine dili programları kolaylıkla değiştirilip, ilerletilemez. Kodların anlamlandırılması kodu inceleyen kişi tarafından çok daha zordur.

5) Semblik makine dilinde belli bir programlama modelinin uygulanması zordur. (Örneğin nesne yönelimli teknik burada kullanılamaz. Ancak prosedürel teknik kullanılabilir.)

Pekiyi C ile sembolik makine dilini birlikte kullanabilir miyiz? Aslında uygulamada en çok kullanılan yöntem bir kodu saf olarak sembolik makine dilinde kodlaamak değil onun belirli kısımlarını sembolik makne dilinde kodlayıp o kısımları C'den çağırarak kullanmak biçimindedir. Yani tıpkı bir işletim sisteminin kodlanmasında olduğu gibi. Gerekmeyen kısımlar C'de ya da yüksek seviyeli bir dilde yazılabilir. Gereken kısımlar sembolik makine dilinde yazılıp C'den çağrılabilir.

1.7. Sembolik Makine Dillerinde Çalışma Modeli

Eğer hedef sistemde bir işletim sistemi varsa (kursumuzda Windows ve Linux sistemleri kullanılacaktır) tipik çalışma modeli şöyledir:

1) Program bir editör kullanılarak sembolik makine dili kuralları ile bir dosyaya yazılır. (Dosya uzunatısı olarak .asm, .s gibi isimler tercih edilmektedir.)

2) Program sembolik makine dili derleyiciyle derlenir ve amaç dosya (object file) elde edilir.

3) Bu amaç dosya bağlayıcı program ile bağlanır (link edilir) ve çalıştırılabilen dosya (executable file) elde edilir. Artık çalıştırılabilen dosyanın içerisinde derleyicinin seçtiği kodlar değil bizim kendi belirlediğimiz makine kodları doğrudan bulunmaktadır.

4) Çalıştırılabilen dosya işletim sisteminin kabuğu yoluyla (örneğin komut satırından ya da grafik arayüz kullanılarak) çalıştırılır.

Sembolik makine dilleri IDE ile çalışmaya çok uygun değildir. Bu nedenle sembolik makine dili programcılarının çoğu derlemeyi komut satırından komut satırı derleyiciyle yaparlar. Fakat yine de sembolik makine dilleri için bazı IDE'ler de kullanılabilmektedir. Bazı sembolik makine dili IDE'lerinin içerisine debugger'lar da entegre edilmiştir. Kursumuzda SASM isimli IDE tercih edilecektir.

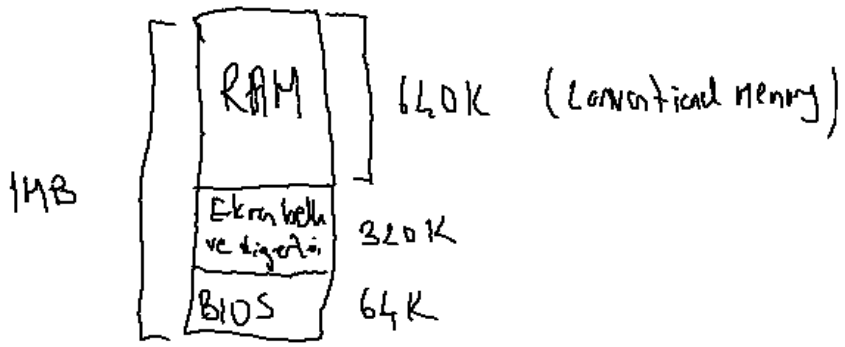
2. Intel 80X86 İşlemcileri

Bu bölümde Intel'in 80X86 serisi mikroişlemcileri konusunda temel bilgiler verilecektir.

2.1. Intel 80X86 Serisi Mikroişlemcilerin Tarihsel Gelişimi

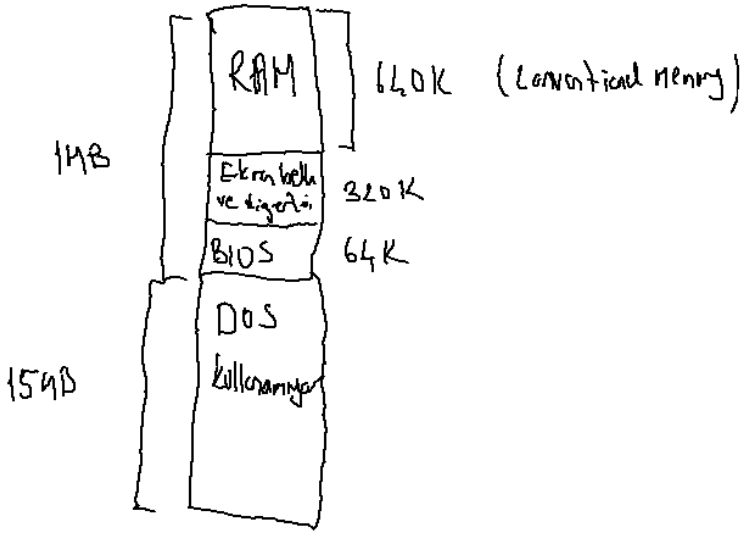
Intel'in 80X86 serisi mikroişlemcilerinin tarihi 70'li yıllara dayanmaktadır. Intel 8086'dan önce 4004, 4008, 8008 ve 8080 gibi (bu serilere ilişkin başka işlemcilerde var) işlemcileri tasarlamıştır. 40 serisi işlemciler tam olarak bir CPU görünümünde değildi. Bu nedenle pek çokları dünyanın ilk mikroişlemcisi olarak 8080'i kabul etmektedir. 8080 8 bit bir mikroişlemciydi (zaten 70'li yıllar 8 bit mikroişlemci yıllarıdır. Aslında hala 8080 işlemcisi kullanılmaktadır.)

80x86 ailesinin ilk üyesi 8086 işlemcisidir. 8086 1978 yılında tasarlanmış ve üretilmiştir. 8086 piyasaya çıktığında çok fazla rağbet görmedi ancak. IBM bugün kullandığımız PC'lerin atası olan ilk bilgisayarı bu işlemciyi kullanarak tasarlamıştır. İşletimini Microsoft'un yazdığı donanımı IBM tarafından tasarlanmış olan ilk PC'ler 1980 sonlarına doğru piyasaya sürülmüştür. 8086 16 bit bir mikroişlemciydi 20 adres ucu, 16 veri ucu vardı. Yani toplamda 1MB belleği kullanabilecek potansiyeldeydi. Ancak PC mimarisinde 1 MB belleğin 384K'sı donanım aygıtlarına ve EPROM'a yönelndirilmiştir. Dolayısıyla programların yüklenebileceği alan 640K kadardı. Gerçekten de DOS işletim sistemi 640K belleği idare edebilecek biçimde tasarlanmıştır.



Bu 640'lık alana hala "geleneksel bellek (conventional memory)" denilmektedir. 8086 işlemcisi 8 MHz hızındaydı. Intel daha sonra 8086'nın 8088 ismiyle daha ucuz bir versiyonunu piyasaya sürmüştür. 8088'in veri yolunun 8 ucu adres yoluyla multiplex yapılmıştı. Bu çalışma hızını biraz düşürüyordu. Ancak 8088 10 MHz hıza yükseltilmişti.

1982 yılında Intel 80286 mikroişlemcisini piyasaya sürdü (bundan önce 80186'da üretilmiştir.) 80286 24 adres ucuna sahipti ve 16 MB belleği adresleyebiliyordu. Fakat 80286 da 16 bit bir mikroişlemciydi. Ayrıca 80286 segment tabanlı sanal bellek kullanımını da mümkün hale getirmişti. Korumalı mod da ilk kez 80286 ile aileye sokuldu. Tabii DOS işletim sistemi 640K belleği kullanacak biçimde tasarlandığı için bu işlemciler DOS ile kullanıldığında bunların 16 MB adresleyebilme yeteneklerinden faydalanılamıyordu.



80286 işlemcilerini kullanan ilk DOS PC makinaları 1985-1986 yıllarında piyasaya sürüldü. Bunlara "AT (Advanced Technology)" ismi verilmişti. Ayrıca AT'lere geçişte PC mimarisinde de bazı küçük değişiklikler yapılmıştır. Örneğin klavyeler 81 tuştan 101 tuşa yükseltilmiştir. 5.25 inch flopy disketlerin yerine AT'lerde 3.5 inch floppy disketlere geçilmiştir. Gerçek zaman saatleri de ilk kez AT'ler zamanında PC'lere dahil edilmiştir. 80286 işlemcileri 16 MHz hızındaydı. Bu nedenle DOS AT'lerde daha hızlı çalışıyordu.

1985 yılında Intel 80386 işlemcisini piyasaya sürdü. Ancak piyasa henüz bu kapasitedeki bir işlemciye hazır değildi. Bu işlemcinin kullanıldığı ilk PC'ler 80 yılların sonlarında doğru piyasaya çıkmıştır. Fakat bu işlemcilerle yapılan PC'lerde de ağırlıklı DOS kullanımı devam etmiştir. 80386 32 bit bir mikroişlemciydi. Bu işlemcilerin adres yolu da 32 uçluydu. Yani teorik olarak bu işlemciler 4 GB belleği adresleyebiliyordu. 16 bitten 32 bite geçiş bçok ciddi bir hız kazancı sağlamıştır. Ancak DOS programları 16 bit olduğu için DOS bu 32 bitlik çalışmanın nimetlerinden faydalanamamıştır. 8036 bugün kullandığımız bilgisayarlardaki işlemcilere en çok benzeyen ilk işlemcidir. Gerçekten de bugün hala uygulamarın büyük bölümü 32 bit uygulamalardır.

80386 için çeşitli işletim sistemleri yazılmış ve port edilmiştir. Örneğin IBM tarafından geliştirilen OS/2, Microsoft tarafından Santa Cruz Operation firması için yazılmış olan XENIX böyle sistemlerdi.

Microsoft artık DOS ile daha fazla yürünemeyeceğini anladı ve grafik arayüzü de olan (Machintosh sistemleri zaten o yıllarda grafik arayüzlü işletim sistemlerine sahipti) Windows sistemleri üzerinde çalışmaya başladı. İlk Windows sistemi aslında bir utility program gibi idi ve 1985 yılında piyasaya çıkmıştı. Bunu Windows 2.0, 3.0 ve 3.1 versiyonları izledi. Windows işletim sisteminin 3.1 versiyonu çok uzun süre kullanılmıştır. Ancak bu 16 bit Windows sistemleri ayrı bir işletim sistemi gibi değil daha çok DOS altında çalışan bir grafik arayüz gibiydi. Microsoft'un 80386 işlemcilerinin kapasitesini tam olarak kullanabilen ilk işletim sistemi Windows NT'dir (1992-1993) bunu 1995 yılında Windows 95 izlemiştir. Bu işletim sistemleri Intel işlemcilerinin 32 bitlik modunu tam kapasiteyle kullanabilmiştir.

80386 geriye doğru uyumlu olarak tasarlanmıştı. Yani bu işlemci hem 8086 gibi çalışabiliyordu hem de 32 bit bir işlemci gibi de çalışabiliyordu. 80386 işlemcisinin üç çalışma modu vardı:

Gerçek mod (real mode): Bu mod tamamen işlemcinin 8086 gibi çalıştığı moddur. 80386 reset edildiğinde bu modda çalışmaya başlar.

Korumalı mod (protected mode): Korumalı mod işlemcinin tam kapasiteyle kullanıldığı sayfalama (dolkayısıyla da sanal bellek() mekanizmasının aktive edilebildiği en ileri çalışma modudur.

Virtual 86 modu (V86 Mode): Bu mod adeta koruma mekanizmasının aktif olduğu gerçek mod gibidir. Intel DOS programlarıyla 32 bit korumalı mod programlarının zaman paylaşımlı çalışabilmesi için bu modu devreye sokmuştur.

80386 işlemcilerinin çalışma modları aşağıdaki tabloyla özetlenebilir:

Mode	Özellik
Gerçek Mod	İşlemci reset edildiğinde bu modda çalışmaya başlar. Temel çalışma modeli 16 bittir. Ancak istenirse 32 bit yazmaçlar ve 32 bit bellek erişimleri de yapılabilir. Bu modda koruma mekanizması etkin değildir.
V86 Modu	Koruma mekanizmasının etkin olduğu 32 bit programlarla 16 bit programların zaman paylaşımlı çalışmasının mümkün hale getirildiği ara bir moddur. Bu moddaki temel çalışma modeli yine 16 bittir. Ancak istenirse 32 bit yazmaçlar ve 32 bit bellek erişimleri de yapılabilir.
Korumalı Mod	Koruma mekanizmasının etkin durumda olduğu tam 32 bit moddur. 32 bit işletim sistemleri 32 bit programları bu modda çalıştırmaktadır. Bu moddaki temel çalışma 32 bittir.

8036 işlemcileri SX ve DX olmak üzere iki modelle piyasaya sürülmüştür.

80486 işlemcisi 1989 yılında Intel tarafından üretildi. Ancak bunların PC'lere girmesi doksanlı yılların ilk yarısında olmuştur. 80486'nın 80386'dan çok önemli farkları yoktur. Çalışma frekansı yükseltilmiştir ve

yeni bazı makine komutları da eklenmiştir. 80486 tipik olarak 50 Mhz hızındaydı. Bunların da SX ve DX modelleri vardı. Intel ilk kezx 80486 DX modeli ile birlikte matemetik işlemciyi de anal işlemci ile aynı entegre devre içerisine dahil etmiştir. Bundan sonraki modeller de böyle devam etmiştir.

1993 yılında Intel Pentium işlemcilerini piyasaya sürmüştür. Pentium aslında 80586 işlemcisidir. Temel çalışma modları 80486 gibidir. Fakatçalışma frekansı artırılmıştır. İşlemciye bazı alt komut kümeleri de eklenmiştir. Bunu daha sonra Pentium Pro, Pentium II, Pentium III ve Pentium IV izlemiştir. Pentium serisi ilerledikçe bazı makine komutları işlemcilere eklenmiş ve işlemcilerin adres alanları 4 * 32GB'ye kadar yükseltilmiştir. Ancak işlemcinin temel çalışma modları 80386 gibidir. Pentium işlemcileri de 32 bit işlemcilerdir ve bunların saat frekansları gittikçe artırılmıştır.

Intel Pentium'larla sonra önce hyper-threading içeren sonra da birden fazla çekirdek içeren modeller piyasaya sürmüştür. Intel çıkarttığı 32 bitlik iki CPU'dan oluşan ilk mikroişlemci "Dual Core" işlemcisidir.

Daha sonraları artık 64 bit Intel işlemcileri devri başlamıştır. Aslında bu aileyi 64 bite ilk taşıyan AMD firmasıdır. Buna AMD64 denilmektedir. Intel AMD64 mimarisini alarak bazı küçük değişiklikler yapmıştır. Buna da X64 denilmektedir. AMD64 ile X64 mimarileri küçük farklılıklar dışında aynıdır. Intel'in ilk 64 bit işlemcisi "Dual Core 2" işlemcisidir. Sonra artık Intel ve AMD hep 64 biti içeren işlemciler yapmıştır.

AMD'nin ve Intel'in 64 bit işlemcileri aslında 52 bit adres yoluna sahiptir. Yani 4 Peta byte fiziksel belleği kullanabilmektedir. Bu işlemcilerin sanal bellek alanı 48 bit yani 256 Tera byte'tır.

AMD64 ve X64 temelde 64 bit mikroişlemcilerdir. Bunların yazmaç uzunlukları 64 bittir. Bunlar tek hamlede 64 bitlik (8 byte'lık) sayılar üzerinde işlem yapabilmektedir. Ancak gerek AMD64 gerekse X64 mimarileri geriye doğru da uyumludur. Yani bunlar Pentium (80486 ya da 8036) gibi de çalışabilmektedir. Örneğin biz bu işlemcilerin kullanıldığı bilgisayarlara 32 bitlik işletim sistemlerini yükleyerek onları 32 bitlik hızlı bir Pentium işlemci gibi de kullanabiliriz.

AMD64 ve X64 işlemcileri "long mode" isminde bir çalışma moduna daha sahiptir. "Long mode" işlemcinin 64 bit çalışma kapasitesinin kullanılabildiği en gelişmiş modudur. AMD64 ve X64 işlemcilerinin "long mode"u iki alt moda ayrılmaktadır: "64 bit long mode" ve "compatibility long mode". "64 bit long mode" 64 bit işlemcinin tüm özelliklerinin tam kapasiteyle kullanılabildiği moddur. "Comaptibility long mode" ise 64 bit işlemcinin 32 bit ve 16 bir korumalı mod programlarını çalıştırabildiği moddur. 64 bit Windows ve 64 bit Linux işletim sistemleri AMD64 ve X64 işlemcilerini "long mode"da çalıştırmaktadır. Bu sistemlerde 64 bit uygulamalar "64 bit long mode"da 32 bit uygulamalar ise "compatibility long mode"da zaman paylaşımlı olarak çalıştırılırlar. (Örneğin Windows ve Linux gibi 64 bit işletim sistemleri de 32 bit programları da 64 bit programlarla birlikte zaman paylaşımlı olarak çalıştırabiliyor.) 64 bit işlemcilerin çalışma modları aşağıdaki tabloyla özetlenebilir:

Mod	Anlamı
Gerçek Mod (Real Mode)	Bu mod işlemcinin 16 bit 8086 gibi çalıştığı moddur. 64 bitlik Intel işlemcileri reset edildiğinde çalışma gerçek moddan başlatılmaktadır.
32 Bit Modu (Legacy Mode) Alt Modlar: - V86 modu (V86 Mode) - Korumalı Mod (Protected Mode)	Bu mod 80386 ve Pentium uyumlu moddur. Bu modda işlemci 32 bit bir Pentium işlemcisi gibi kullanılır. AMD64 ve X64 işlemcilerine 32 bit işletim sistemi yüklendiğinde bu işletim sistemleri 64 bit işlemcileri bu modda kullanır.
Long Mod (Long Mode) Alt Modlar: - "Compatibility Long Mode"	"Compatibility Long mode" 64 bit uygulamalarla 32 bit uygulamaların zaman paylaşımlı biçimde beraber çalıştırılmaları için kullanılan moddur. 64 bit işletim sistemleri 32 bit uygulamaları çalıştırırken işlemciyi "compatibility long mode"da çalıştırmaktadır. "64 bit long mode" ise 64 bit işlemcilerinin 64 bit yeteneğinin kullanılabildiği

- “64 Bit Long Mode”	çalışma modudur. “long mode”da 16 bit programlar çalıştırılamamaktadır. (Böylece örneğimn biz 64 bit Windows sistemlerinde 16 bitlik MS-DOS programlarını çalıştıramayız.)

Buraya kadar açıkladığımız tüm Intel işlemcileri reset edildiğinde hala gerçek modda çalışmaya başlatılmaktadır. Yani elimizdeki 64 bitlik çok çekirdekli işlemciler bile reset edildiğinde hala 1978 yılında tasarlanan 8086’nın çok hızlı bir biçimi gibi çalışmaya başlamaktadır.

Anahtar Notlar: HP ile Intel’in işbirliği ile tasarlanan Itanium işlemcisi de 64 bit bir RISC işlemcisidir. Itanium bazı server makinelerde tercih edilmektedir. Intel’in Itanium mimarisine IA64 denilmektedir. X64 ile IA64 terimleri bazen birbirlerine karıştırılabilmektedir.

2.2. Matematik İşlemciler

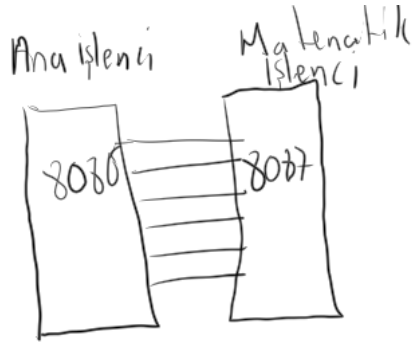
İlk üretilen mikroişlemciler (örneğin 8 bitlik işlemciler ve 16 bitlik işlemciler) yalnızca tamsayı işlemleri yapabiliyordu. Çünkü gerçek sayılarla işlemler için büyük mantık devreleri gerekiyordu. O zamanın teknolojileri buna fazlaca müsait değildi. O yıllarda matematiksel işlemler aslında arka planda tamsayı işlemleriyle, yani bir kod çalıştırılarak (başka bir deyişle fonksiyon çağrılarak) yapılıyordu. Örneğin 80’li yıllarında başında 8086 için zamanlarında aşağıdaki gibi bir kod yazmış olalım:

```
double a = 3.4, b = 67.8, c;

c = a + b;
```

İşte C derleyicileri bu tür işlemleri kendi kütüphane fonksiyonlarınu kullanarak adeta aşağıdaki gibi yapıyorlardı:

```
double a = 3.4, b = 67.8, c;

c = fadd(a, b);
```

Tabii gerçeksayı işlemlerinin tamsayı aritmetiğiyle emülasyon yoluyla yapılması yavaşlığa yol açıyordu. İşte Intel işlemleri hızlandırmak için 8087 isminde, 8086 işlemcisi ile bağlanarak koordineli çalışacak bir matematik işlemci (math coprocessor) de tasarladı. (Örneğin eski ana kartlarda ana işlemcinin yanında genellikle boş bir soket bulunuyordu. Bu soket 8087 matematik işlemcisi için ayrılmıştı). 8087 matematik işlemcisi gerçek sayı işlemlerini donanım yoluyla, yani elektrik devreleriyle yapıyordu.



Böylece bu eski günlerde noktalı sayılarla bir işlem yaptığımızda eğer sistemimizde matematik işlemci yoksa bu işlemler emülasyon yoluyla, eğer sistemimizde matematik işlemci varsa matematik işlemcinin elektrik devreleriyle yapılmaktaydı. Intel 286’yı çıkartınca matematik işlemcisini de 80287 ismiyle güncelledi. Sonra 80386 çıktığında matematik işlemci 8037 oldu. İşte nihayet 80486 DX modeliyle birlikte artık Intel matematik işlemciyi de ana işlemciyla aynı entegre devre içerisine yerleştirmeye başladı. Bugünkü kullandığımız Intel işlemcilerinde yine matematik işlemci ayrı bir birim olarak vardır fakat bunlar aynı entegre devrenin içerisindedir.

Intel'in matematik işlemci sistemi şöyle çalışmaktadır: Ana işlemci (yani tamsayı işlemcisi) komutu çektiğinde (fetch) onun başındaki byte'a bakar. O byte özel bir değerdeyse (bu tür byte'lara Intel terminolojisinde "prefix" denilmektedir). O komutun gerçeksayı işlemi yapan makine komutu olduğunu anlar. Komutu yardımcı işlemciye verir. Onu artık yardımcı işlemci çalıştırır. Intel bu konuda zaman içerisinde bazı optimizasyonlar yaptıysa da temel çalışma biçimi hala böyledir.

Artık günümüzde tasarlanan yeni işlemciler kendi içlerinde gerçek sayı işlem birimini de içeriyorlar. Yani yeni tasarımlarda artık ayrı bir matematik işlemci diye kavram yoktur. Tasarımcı zaten işlemciyi gerçek sayı işlemlerini de yapacak biçimde tek parça olarak tasarlamaktadır.

Anahtar Notlar: Tabii bu anlatımdan bugün her işlemcinin gerçek sayı işlemlerini yapan birime de sahip olduğu gibi bir anlam çıkmasınn. Bugün için hala küçük mikrodenetleyicilerin ve mikroişlemcilerin matematik işlemci modülleri yoktur. (Örneğin Microchip'in PIC16 mikrodenetleyicilerinin matematik işlemci birimleri yoktur. Bu mikrodenetleyicilerde noktalı sayılarla işlemler yine emülasyon yoluyla yapılmakatadır.)

Anahtar Notlar: Noktalı sayılarla işlemler özellikle IEEE 754 gibi kayan noktalı (floating point) formatlarda zahmetlidir. Bu nedenle küçük mikrodenetleyicilerde programcılar sabit noktalı (fixed point) formatları tercih etmektedir. Çünkü sabit noktalı formatlarla işlemler tamsayılarla çok kolay emüle edilebilmektedir.

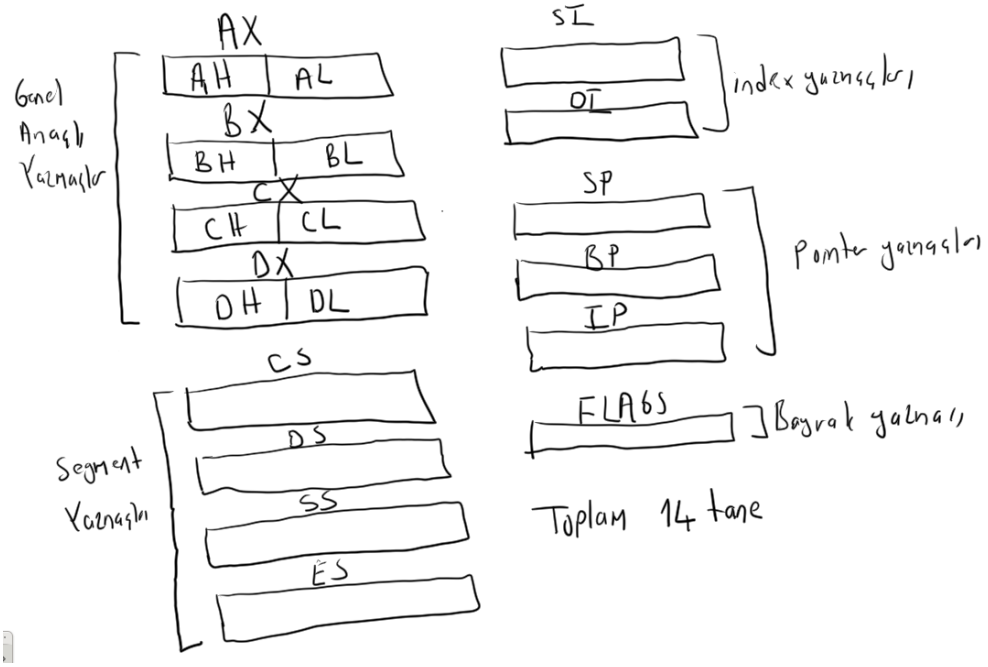
Anahtar Notlar: Bugün modern kapasiteli mikroişlemcilerin hemen hepsi IEEE 754 kayan noktalı formatı kullanmaktadır. Kayan noktalı formatlar daha dinamik olduğu için daha verimlidir. Bu formatlarda sayı nokta yokmuş gibi ikilik sistemde saklanır. Sonra noktanın yeri sayı içerisinde bazı bitlerde tutulmaktadır.

2.3. Intel 80X86 İşlemcilerinin Yazmaç Yapısı

Bugün kullandığımız 64 bit Intel işlemcilerinin yazmaç yapısı 8086 işlemcisinden evrilerek bu halini almıştır. Bu nedenle her ne kadar biz kursumuzda 32 bit ve 64 bit sembolik makine dili programlamasını görecek olsak da işin başında 16 bit 8086'nın yazmaç yapısına da değineceğiz. Tarihsel arka plan ve geriye doğru uyumluluk bizim konuyu daha iyi anlamamıza yardımcı olacaktır.

2.3.1. 16 Bit 8086 İşlemcisinin Yazmaç Yapısı

8086 işlemcisinin programcı tarafından kullanılabilen yazmaç yapısı şöyleydi:



Görüldüğü gibi 16 bit 8086 işlemcisinde toplam 14 yazmaç vardır. AX (Accumulator Register), BX (Base Register), CX (Count Register) ve DX (Data Register) yazmaçlarına genel amaçlı (general purpose) yazmaçlar denir. Çünkü temel artimetik ve bit işlemleri –bazı istisnalar olmakla birlikte- bu yazmaçların

hepsiyle yapılabilmektedir. Şekilden de gördüğünüz gibi genel amaçlı yazmaçların her biri 8 bitlik iki parçaya ayrılmış durumdadır. Bu parçalardan yüksek anlamlı olanlara H (high) soneki, düşük anlamlık olanlara L (low) soneki verilmiştir. Örneğin AX yazmacının yüksek anlamlı 8 biti AH, düşük anlamlı 8 biti AL'dir. 8 bitlik bu yazmaçlar bağımsız yazmaçlarmış gibi makine komutlarında kullanılabilmektedir. Örneğin:

```
ADD AX, BX
```

Burada biz 16 bitlik bir toplama yapmış oluyoruz. Halbuki:

```
ADD AH, BL
```

Burada biz 8 bitlik iki sayıyı toplamış oluyoruz. Örneğin:

```
MOV [1FC0], AX
```

Burada AX'teki değer belleğin 1FC0 adresinden itibaren aktarılmaktadır. (Yani bu işlemden 1FC0 ve 1FC1 byte'ları etkilenecektir). Halbuki:

```
MOV [1FC0], AL
```

Burada 1FC0 adresine 8 bitlik bir değer aktarılmaktadır. (Yani bu işlemden yalnızca 1FC0 byte'ı etkilenecektir.) Ayrıca şunu da belirtmek gerekir: Biz örneğin AH ve AL yazmaçlarına ayrı ayrı 8 bit yerleştirip AX'i kullandığımız zaman yüksek anlamlı 8 biti AH, düşük anlamlı 8 biti AL olan değeri kullanmış oluruz. Yani bu biçimde biz parçalardan bütünü, bütünden de parçaları oluşturabilmekteyiz. Şu noktaya da dikkat ediniz: 8086 işlemcisinin 16 bit olması her işlemin 16 bit olarak yapıldığı anlamına gelmemektedir. En fazla 16 bit işlemin tek hamlede yapılabileceği anlamına gelmektedir. Özetle 16 bitlik 8086'da 16 bit işlemler için AX, BX, CX ve DX 8 bitlik işlemler için bunların H ve L parçaları kullanılmaktadır.

8086'nın SI (Source Index Register) ve DI (Destination Index Register) yazmaçları DS'yi indeksleyen yazmaçlardı. Artık 32 bit sistemde genel amaçlı yazmaçlar da bu indekslemeyi yapabildiği için bu yazmaçların diğerlerine göre önemi kalmamıştır. Bu yazmaçlarla yine aritmetik ve bitsel işlemler yapılabilmektedir. Ancak bu yazmaçların yüksek ve düşük anlamlı 8 bitlik kısımlarına ayrıca erişilememektedir.

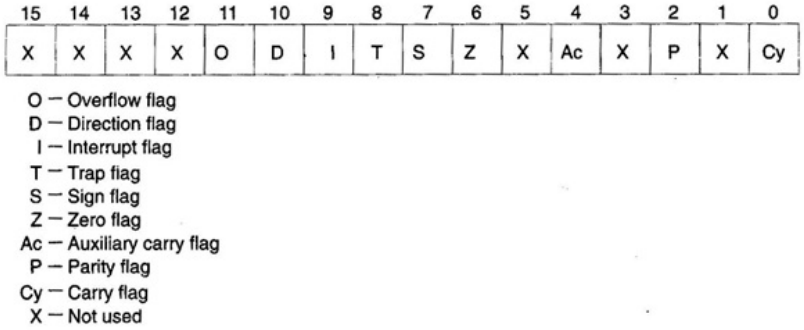
IP (Instruction Pointer) yazmacı bilindiği gibi o anda çalıştırılan makine komutunun bellekteki adresini tutar. SP (Stack Pointer) stack'in aktif noktasını (yani tepesini) tutan önemli bir yazmaçtır. BP (Base Pointer) stack'ten veri almak ve oraya veri aktarmak için kullanılmaktadır.

8086'da dört tane segment yazmacı vardır: CS (Code Segment Register), DS (Data Segment Register), SS (Stack Segment Register) ve ES (Extra Segment Register). ES dışındaki diğer üç yazmaç 1 MB belleği adresleyebilmek için taban adres görevini yapar. Ancak 32 bit korumalı mod programlamada bu segment yazmaçlarının işlevi değişmiştir. 32 bit korumalı modda artık bu yazmaçlara "selector" denilmektedir. Biz burada 16 bit segment yazmaçlarının işlevini görmeyeceğiz. Çünkü 16 bit programlama artık çok az kullanılmamaktadır.

Anahtar Notlar: Tabii 16 bir programlama hiç kullanılmıyor da değildir. Bugün kullandığımız PC'leri reset ettiğimizde çalışmanın 16 bit gerçek modda başlatıldığını anımsayınız. PC'lerimizdeki BIOS'un önemli bölümü 16 bit gerçek mod için yazılmış kodlardan oluşmaktadır. Ayrıca hala MS-DOS ya da FreeDOS kullanan bilgisayarlar da vardır.

FLAGS yazmacı bitlerden oluşan bir yazmaçtır. Bu yazmacın her bitinin anlamı farklıdır. FLAGS yazmacının her bitine bayrak (flag) denir. Bir bayrak bitinin 1 olmasına İngilizce ilgili bayrağın "set" edilmiş olması, 0 olmasına ise "reset" edilmiş olması denilmektedir. Her bayrağa ayrıca bir isim de verilmiştir. (Örneğin Zero Flag, Carry Flag, Overflow Flag gibi). Makine komutlarının birçoğu bir işlem

sonucunda bu bayrak yazmacının bazı bitleri üzerinde değişikliğe yol açar. Böylece sembolik makine dili programcısı yaptığı işlemin sonucu hakkında bu bayraklara bakarak bilgi edinmektedir. (Özellikle sembolik makine dilinde jump işlemleri bayraklara dayalı olarak yapılıyor. Örneğin JZ komutu "Jump (if) Zero" biçiminde okunur ve "eğer ZF bayrağı set edilmişse jump işlemi yap" anlamına gelir. Benzer biçimde örneğin JNC komutu da "carry flag set edilmemişse (ya da reset durumdaysa) jump işlemi yap" anlamına gelir. Aşağıda 8086'nın FLAGS yazmacının bitlerini (bayraklarını) görüyorsunuz:

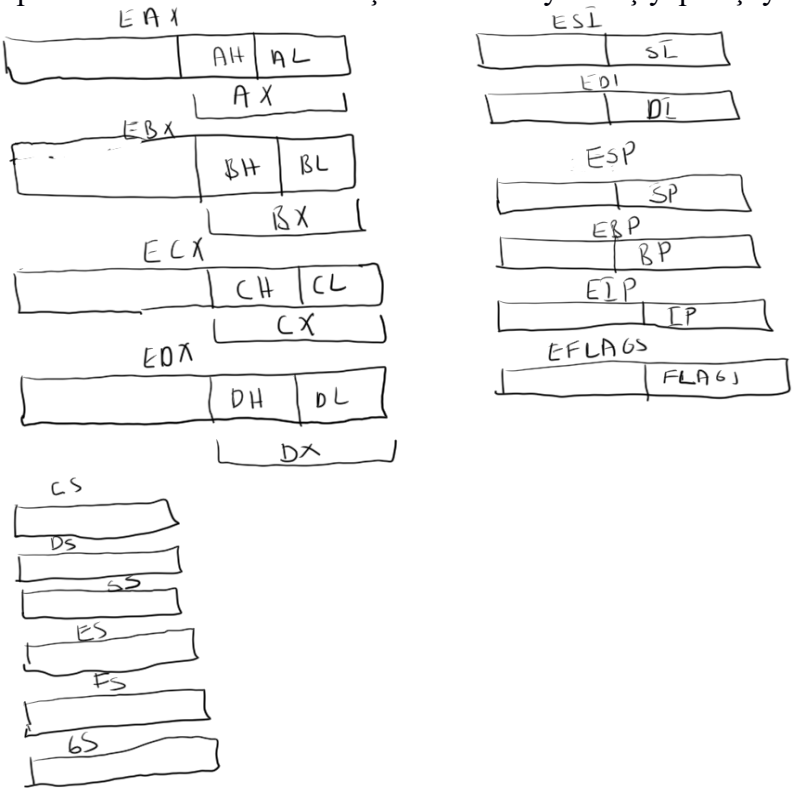


Görüldüğü gibi 8086'nın FLAGS yazmacının bazı bitleri hiç kullanılmamaktadır. FLAGS yazmacının bitleri ve onların anlamları ileride ayrıntılarıyla ele alınacaktır.

2.3.2. 32 Bit 80X86 İşlemcilerinin Yazmaç Yapısı

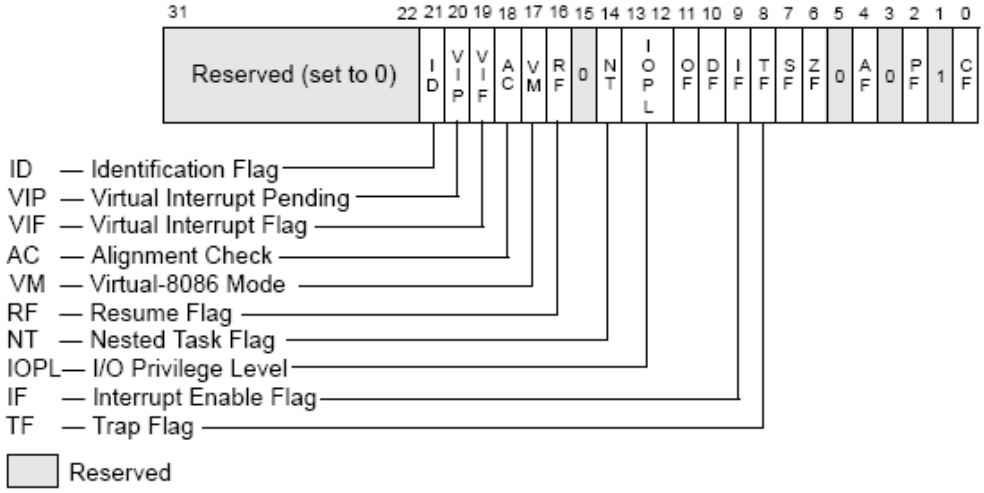
Bugün bile işletim sistemlerimiz 64 bit olsa da pek çok programcı hala 32 bit programlar yazmaktadır. Intel işlemcilerinde 32 bit programlamanın halen en yoğun kullanılan programlama modeli olduğunu söyleyebiliriz.

Anımsanacağı gibi Intel'in ilk 32 bit işlemcisi 80386'dır. Bunu 80486 ve Pentium işlemcileri izlemiştir. Tüm bu işlemcilerin yazmaçları 32 bittir. Yani bunlar tek hamlede 32 bitlik tamsayılar üzerinde işlemler yapabilmektedir. 32 bit Intel işlemcilerinin yazmaç yapısı şöyledir:



Görüldüğü gibi 32 bit sisteme geçildiğinde 16 bit yazmaçların başına E harfi getirilmiştir ve bunlar 32 bite yükseltilmiştir. Ayrıca FS ve GS isimli iki segment yazmacının daha eklendiğini görüyorsunuz.

Genel amaçlı yazmaçlar için dikkat edilmesi gereken bir nokta vardır: Bunların yüksek anlamlı 16 biti bağımsız bir yazmaç gibi kullanılamamaktadır. Örneğin biz EAX yazmacını kullandığımızda 32 bit, AX yazmacını kullanırsak 16 bit işlem yaparız. AH ve AL yazmaçlarıyla da 8 bit işlemler yapabiliriz. ESI, EDI, EBP, ESP, EIP yazmaçları da 32 bit yazmaçlardır. Bayrak yazmacı da EFLAGS ismiyle 32 bite yükseltilmiştir. Şekle dikkat edilirse CS, DS, SS ve ES yazmaçları 16 bitte bırakılmıştır. Bunlara ek olarak yine 16 bitlik FS ve GS isimli iki segment yazmacı daha eklenmiştir. EFLAGS yazmacının bayrakları aşağıdaki duruma getirilmiştir:



Yukarıda da belirttiğimiz gibi 32 bit Intel işlemcileri 32 bit, 16 bit ve 8 bit işlemler yapabilmektedir. Ancak Intel işlemcilerinin makine komutlarının oluşturulma sistematiği değiştirilemediği için 32 bitte eklenen yeni yazmaçların kodlanması sorun oluşturmuştur. Intel bu sorunu 0x66 öneki (prefix) ile çözmeye çalışmıştır. 32 bit Intel işlemcilerinin korumalı modunda 16 bit yazmaçların kullanılması opcode olarak bir byte'lık maaliyetle yapılmaktadır. Yani biz 32 bit korumalı modda çalışıyorsak 16 bit yazmaçları kullanırken makine komutlarımız ekstra 1 byte büyür. Benzer biçimde aslında biz bu işlemcilerde gerçek modda ya da V86 modunda çalışıyorsak yine de 32 bit yazmaçları kullanabiliriz. Fakat bu sefer tam tersine bu 32 bit yazmaçlar için 0x66 öneki gerekmektedir. (Yani tam ters olarak 16 bit modda 32 bit yazmaçları kullanırken ekstra bir byte gerekmektedir.) Buradan özetle şu sonuç çıkmaktadır: Biz 32 bit Intel işlemcilerinde 32 bitlik, 16 bitlik ve 8 bitlik yazmaçları kullanabiliriz. Ancak gerçek mod ve V86 modunda (yani 16 bit modunda) 32 bit yazmaçların, korumalı modda (yani 32 bit modda) ise 16 bit yazmaçların kullanılması ek bir byte'lık 0x66 önek maaliyeti getirmektedir. 32 bit modda ise 8 bitlik yazmaçlar maaliyetsiz kullanılabilmektedir. Bunu aşağıdaki şekille de ifade edebilirz:

Mod	16 Bit Yazmaç Kullanımı	32 Bit Yazmaç Kullanımı
Gerçek Mod	Öneksiz	0x66 Önekli
V86 Modu	Öneksiz	0x66 Önekli
Korumalı Mod	0x66 Önekli	Öneksiz

32 bit Intel işlemcileri belleğe erişirkemn de 16 bit ya da 32 bit offset kullanabilmektedir. Bu konuda ileride açıklanacak olsa da burada bazı ip uçlarını vermek istiyoruz. 16 bit modda belleğe 32 bit adres ile erişim makine kodunda 0x67 önekiyle (yani 1 byte maaliyetle) yapılmaktadır. Benzer biçimde 32 bit korumalı modda da belleğe 16 bit adresle erişimler için 0x67 öneki gerekmektedir. Bunu da şöyle özetleyebiliriz:

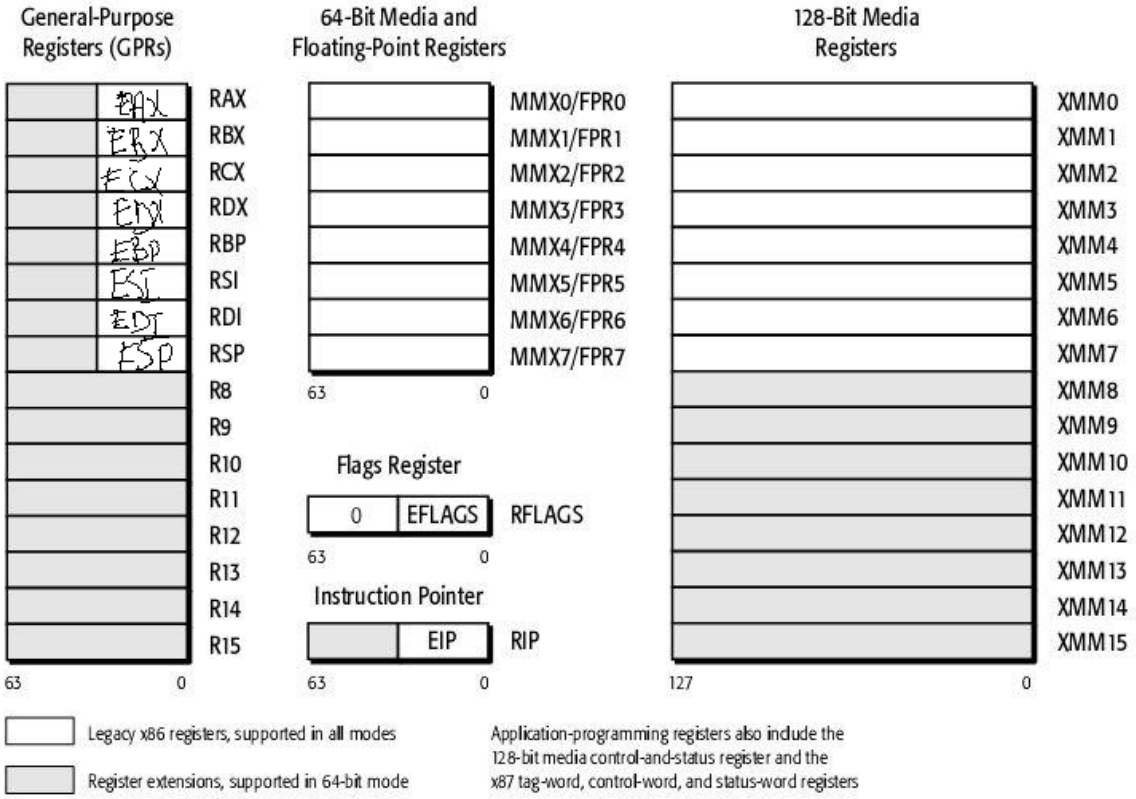
Mod	16 Bit Adresle Belleğe Erişim	32 Bit Adresle Belleğe Erişim
Gerçek Mod	Öneksiz	0x67 Önekli
V86 Modu	Öneksiz	0x67 Önekli

Korumalı Mod	0x67 Önekli	Öneksiz

Bir makine komutunda 0x66 ve 0x67 öneklerinin her ikisi de bulunuyor olabilir. 0x66 ve 0x67 öneklerinin anlamlarını tam olarak kavrayamamış olabilirsiniz. Bunun için endişelenmeyin. İleride başka konular ele alındıktan sonra bu öneklerin anlamları üzerinde yeniden durulacaktır.

2.3.3. 64 Bit 80X86 İşlemcilerinin(AMD64 ve X64) Yazmaç Yapısı

64 bite geçildiğinde Intel yukarıda gördüğümüz yazmaçları da 64 bite yükseltmiştir. Bunun için yazmaç isimlerinin başındaki E öneki 64 bit için R haline (R muhtemelen "Register" sözcüğünden geliyor) getirilmiştir. Örneğn RAX, RBX, RCX, RDX gibi. 64 Bit Intel işlemcilerinin yazmaç yapısı şöyledir:



Şekilden de görüldüğü gibi 64 bit sistemde şu ek farklılıklar söz konusudur:

- 64 bite geçildiğinde Intel genel amaçlı yazmaçlara 8 tane daha eklemiştir. Bunlar R8-R15 yazmaçlarıdır.

- 64 bit modda genel olarak önceki modlardaki 32 bit yazmaçlar, 16 bit yazmaçlar ve 8 bit yazmaçlar kullanılabilmektedir. (Yani örneğin biz 64 bit modda RAX yazmacını da EAX yazmacını da AX yazmacını da AL yazmacını da kullanabiliriz)

- Intel'in Pentium serisine 64 bitlik MMX komut kümesi eklenmişti. Bu komutlar MMX yazmaçlarını kullanıyordu. 64 bitte özel 128 bitlik XMM komut kümesi ve yazmaçları da mimariye dahil edilmiştir. (Ayrıca 64 bit işlemcilerin sonraki modellerde vektörel işlem yapan yine 16 tane 256 bit YMM yazmaçları da mimariye eklendi. YMM yazmaçlarının düşük anlamlı 128 biti XMM yazmaçları ile çakışmaktadır. Yukarıdaki şekile YMM yazmaçları dahil edilmemiştir.)

255	127	0
	XMM0	YMM0
	XMM1	YMM1
	XMM2	YMM2
	XMM3	YMM3
	XMM4	YMM4
	XMM5	YMM5
	XMM6	YMM6
	XMM7	YMM7
	XMM8	YMM8
	XMM9	YMM9
	XMM10	YMM10
	XMM11	YMM11
	XMM12	YMM12
	XMM13	YMM13
	XMM14	YMM14
	XMM15	YMM15

Daha önceden de sözünü ettiğimiz gibi AMD64 ve X64 işlemcileri "32 bit" çalışma modunun yanı sıra bir de "long" moda sahiptir. Bu long modun da "64 bit long" mod ve "compatibility long mod" biçiminde iki alt modu olduğunu anımsayınız. İşte 64 bitlik yazmaçlar yalnızca bu işlemcilerin "64 bit long" modunda kullanılabilmektedir.

"64 bit long" modda 64 bit yazmaçların kullanılabilmesi için bir byte'lık REX isimli öneklere gereksinim duyulmaktadır. Yani 64 bit işlemcimiz "64 bit long" moddaysa biz 64 bit yazmaçları kullanrıken makine komutları fazladan bir byte artar. Ayrıca bu modda yine 0x66 ve 0x67 önekleri kullanılabilmektedir. Bu öneklerin işlevleri ileride ele alınacak olsa da biz yine de bunun hakkında şimdiden bazı bilgiler vermek istiyoruz. Aşağıdaki önek açıklamalarını tam olarak anlayamazsanız telaşa kapılmayın. Bunlar sonrak bölümlerde zaten yeniden ele alınacak.

"64 bit long" modda 16 bit, 32 bit yazmaçlara erişim ve 32 bit, 64 bit adreslerle belleğe erişim için 0x66 ve 0x67 öneklerinin kullanımı şöyledir:

8 Bit Yazmaçlara Erişim	**16 Bit Yazmaçlara Erişim**	**32 Bit Yazmaçlara Erişim**	**32 Bit Adresle Belleğe Erişim**	**64 Bit Adresle Belleğe Erişim**
Öneksiz	0x66 Önekli	Öneksiz	0x67 Önekli	Öneksiz

Tabii bu öneklerin biri ya da birden fazlası kullanılabilmektedir. (Yani örneğin komutun başında yalnızca 0x66, yalnızca 0x67 ya da hem 9x66 hem de 0x67 önekleri bulunuyor olabilir.)

"64 bit long" modda ayrıca komut kümesine 4 tane REX isimli önek de dahil edilmiştir. Bix bu 4 REX önekine kısaca "REX önekleri" diyeceğiz. REX öneklerinin işlevleri şöyledir:

- REX önekleri ile biz 8 bitlik yazmaçlardan AL, CL, DL, BL yazmaçlarına erişebiliriz. Ancak AH, CH, DH, BH yazmaçlarına erişemeyiz. Bunun yerine SPL, BPL, SIL ve DIL (yani SP'nin, BP'nin, SI'nın ve DI'nin düşük anlamlı byte'larına) erişebiliriz.

- REX önekleri ile biz 16 bit yazmaçlara ve ayrıca R8'den R15'e kadar yeni eklenen sekiz tane 64 bit yazmacın düşük anlamlı 16 bitlik kısmına R8W, R9W, R10W, R11W, R12W, R13W, R14W ve R15W isimleriyle erişebiliriz.

- REX önekleri ile biz 32 bit yazmaçlara ve ayrıca R8'den R15'e kadar yeni eklenen sekiz tane 64 bit yazmacın düşük anlamlı 32 bitlik kısmına R8D, R9D, R10D, R11D, R12D, R13D, R14D ve R15D isimleriyle erişebiliriz.

- REX önekleri ile biz 64 bit yazmaçlara ve ayrıca R8'den R15'e kadar yeni eklenen sekiz 64 bit yazmaca erişebiliriz.

REX önekleri ile 0x66 önekleri birlikte kullanılamamaktadır. Ancak REX önekleri ile 0x67 önekleri birlikte kullanılabilir. Bu durumda "64 bit long" modda 32 bit bellek erişimi yapılabilmektedir.

REX önekleri kullanılsa bile 64 bit yazmaçların yüksek anlamlı 32 bitlik kısımlarına erişilemediğine dikkat ediniz.

Burada bir kez daha bir uyarıda bulunmak istiyoruz: Yukarıdaki "64 bit long" moddaki önekler şimdilik kafanızı kurcalamasın. Sizin bu aşamada bilmeniz gereken şey şudur: 64 bit işletim sistemleri 64 bit programları 64 bit long modda çalıştırırlar. Bu moddaki programlar da 64 bit yazmaçları, 32 bit yazmaçları, 16 bit yazmaçları ve 8 bitlik yazmaçları kullanabilmektedir. Ancak bu kullanım sırasında komutun önünde bazı öneklerin (0x66, 0x67 ve REX) bulunması gerekebilmektedir.

2.4. 32 Bit Intel İşlemcilerinde Komut Kalıpları

İşlemcilerdeki yazmaç ve bellek kullanım kalıplarına "adresleme modları (addressing mode)" da denilmektedir. Intel'in 32 bit işlemcilerinde komut kalıplarını aşağıda maddeler halinde tek tek ele alacağız. Daha önceden de belirtildiği gibi Intel sisteminde komutlar iki operandlıdır. Ayrıca örneklerimizde pek çok assembly derleyicisinin kabul ettiği gibi komutların hedef operandları sol taraftaki operandla belirtilecektir (AT&T yazım tarzında sağ taraftaki operandla belirtilmektedir.)

Sembolik makine dillerinde genel olarak komutlarda belirtilen işlemlerin kaç bit düzeyinde yapılacağı eğer komutta yazmaç varsa yazmacın uzunluğuyla tespit edilir. Ancak komutta yazmaç yoksa işlemlerin kaç bit düzeyinde yapılacağı onların yanına getirilen qword, dword, word, byte gibi anahtar sözcüklerle belirlenmektedir. Tabii bu anahtar sözcükler assembly derleyicisinden derleyicisine farklılık gösterebilmektedir.

Aslında bir işlemcinin tüm makine komutları bütün kombinasyonlarıyla tek tek bir liste ile de gösterilebilir. Ancak komutları bir liste biçiminde tek tek ele almak öğrenmeyi zorlaştırmaktadır. Bunun yerine komutları ortak özellikler bağlamında öğrenmek çok daha pratiktir. İşte komutların öğrenmeyi kolaylaştıracak bu ortak özelliklerine "komut kalıpları" diyoruz.

32 Bit Intel İşlemcilerinin komut kalıpları özet olarak şöyledir:

1) Yazmaçlarla sabitler işleme sokulabilirler.

2) Yazmaçlarla yazmaçlar işleme sokulabilirler.

3) Yazmaçlarla bellekteki değerler işleme sokulabilirler.

4) Bellekteki değerlerle sabitler işleme sokulabilirler.

Bellek operandı genellikle sembolik makine dillerinde köşeli parantezlerle gösterilmektedir. Bellek operandı herhangi bir biçimde oluşturulamaz. Köşeli parantezlerin içerisine nelerin yazılabileceği önceden belirlenmiştir. Aşağıda da göreceğiniz gibi köşeli parantezler içerisinde yazmaçlar bulundurulabilmektedir. Ancak 32 bit Intel işlemcilerinde hangi yazmaçların kçşeli parantezler içerisinde bulundurulabileceği önceden belirlenmiştir. 16 bit yazmaçlardan yalnızca SI, DI, BX ve BP yazmaçları, 32 bit yazmaçlardan da EAX, EBX, ECX, EDX, ESI, EDI, EBP ve ESP yazmaçları köşeli parantezler içerisinde bulundurulabilmektedir.

Şimdi komut kalıplarını daha ayrıntılı olarak tek tek ele alalım:

1) Yazmaçlarla sabitler işleme sokulabilirler. Örneğin:

```
ADD     EAX, 100
ADD     AH, 20
MOV     EAX, 512
```

Sembolik makine dillerinde doğrudan yazılan sayılara İngilizce "immediate" denilmektedir. "Constant" ya da "literal" dennilmemektedir. Ancak biz kursumuzda "immediate" terimi yerine "sabit" terimini kullancağız.

2) Yazmaçlarla yazmaçlar işleme sokulabilirler. Ancak yazmaç uzunluk uyumunun sağlanmış olması gerekir. Örneğin:

```
ADD     EAX, EBX
SUB     AX, CX
XOR     AH, BL
```

Fakat örneğin aşağıdaki komutlar yazmaç uzunlukları aynı olmadığı için geçersizdir:

```
ADD EAX, BX
XOR BX, AL
```

Bu konuda bazı istisnalar vardır.

3) Yazmaçlarla köşeli parantez içerisindeki sabit değerler işleme sokulabilirler (Burada sabit değerler adres belirtmektedir). Genel olarak bütün işlemciler bellekteki değerlere onların adreslerini alarak erişirler. Sembolik makine dillerinin çoğunda bir bellek adresindeki değerler köşeli parantezlerle belirtilmektedir. Bu durumda o yazmaçtaki değer ile köşeli parantez içerisindeki bellek adresinden başlayan değer işleme sokulmuş olur. İşlemin kaç bit düzeyinde yapılacağı yazmacın uzunluğuna bağlıdır. Örneğin:

```
ADD EAX, [1FC14A]
```

Burada EAX yazmacındaki değer ile belleğin 1FC14A adresinden başlayan 32 bitlik değer toplanmıştır. Sonuç EAX yazmacındaki değer bozularak oraya aktarılmaktadır. Örneğin:

```
MOV BX, [1FC14A]
```

Burada BX'e 1FC14A adresinden başlayan 16 bit değer aktarılmaktadır. Örneğin:

```
ADD AH, [1FC14A]
```

Burada AH içerisindeki değerle 1FC14A adresinden başlayan 8 bitlik değer toplanmış, sonuç yine AH'ya atanmıştır. Örneğin:

```
ADD [1FC14A], EAX
```

Burada EAX yazmacı ile bellekte 1FC14A adresinden başlayan 32 bit değer işleme sokulmuştur. Fakat sonuç belleğin yine 1FC14A adresinden itibaren 32 biti etkileyecek biçimde aktarılmaktadır.

Kalıptaki köşeli parantezin işlevine dikkat ediniz. Köşeli parantezler olmasaydı komut tamamen asssembly derleyicisi tarafından farklı yorumlanırdı. Örneğin:

```
ADD EAX, 1FC14A
```

Bu komutta EAX yazmacındaki değer doğrudan 1FC14A sabiti ile (immediate) ile toplanmıştır. Sonuç EAX'te bulunacaktır. Fakat örneğin:

```
ADD EAX, [1FC14A]
```

Burada EAX yazmacındaki değer ile 1FC14A adresinden başlayan 32 bit değer toplanmıştır.

RISC işlemcilerinde yazmaç ile bellek bölgesinin işleme sokulamadığını anımsayınız. Yazmaç-bellek işlemleri tipik olarak CISC tarzı eski işlemcilerde karşılaşılan komut kalıplarıdır.

Normal olarak köşeli parantez içerisindeki sabit adresler 32 bit korumalı modda 32 bit uzunluğundadır. Ancak köşeli parantez içerisindeki bellek adresi 16 bitten de oluşabilir. Ancak komutlardaki 16 bit adresler 32 bit korumalı modda 0x67 öneki gerektirmektedir. Bu nedenle böyle komutlarla 32 bit programlamada pek karşılaşmayız.

4) Köşeli parantez içerisindeki adresler ile sabitler işleme sokulabilirler. Bu durumda sembolik makine dili derleyicileri komutta yazmaç olmadığı için işlemin kaç bit üzerinden yapılacağını ek anahtar sözcükler yardımıyla anlarlar. Örneğin:

```
ADD dword [1FC14A], 100
```

Burada belleğin 1FC14A adresinden başlayan 32 bitlik değer 100 ile toplanmıştır, sonuç yine 1FC14A adresinden başlayarak oraya aktarılmıştır. Komuttaki dword işlemin 32 bit (double word = 4 byte = 32 bit) olduğunu assembly derleyicisine anlatmak için gerekmektedir. (Değişik derleyicilerde bu amaçla değişik anahtar sözcükler kullanılabilmektedir.) RISC işlemcilerinde bellekteki değerlerle sabitler işlemlere sokulamamaktadır. Bellekteki değerlerle sabitler işleme sokulurken köşeli parantez içerisindeki bellek adresi sonraki maddelerde olduğu gibi yazmaçlarla da oluşturulabilmektedir.

5) Bir yazmaçla köşeli parantez içerisindeki bir yazmaç işleme sokulabilir (örneğin [EAX], [EBX] gibi). Bu durumda o yazmacın içerisindeki değerle belirtilen adresteki değer işleme sokulur. Örneğin:

```
MOV EAX, 1FC10A
MOV EBX, [EAX]
```

Burada EAX yazmacının içerisinde 1FC10A değeri vardır. O halde bellekteki 1FC10A adresinden başlayan 32 bit değer EBX yazmacına aktarılmıştır. Örneğin:

```
MOV EAX, 1FC10A
MOV EBX, [EAX]
ADD EAX, 4
MOV EBX, [EAX]
ADD EAX, 4
MOV EBX, [EAX]
...
```

Bu adresleme modunun neden kullanıldığı yukarıdaki örnek kodla anlaşılabilir. Biz bu sayede örneğin bir yazmaca bir dizinin adresini atayıp sonra o yazmacı köşeli parantez içerisinde kullanarak dizinin tüm elemanlarına erişebiliriz. Benzer biçimde gösterici işlemleri de derleyici tarafından bu komut kalıbıyla yapılmaktadır. Örneğin:

```
int *pi = (int *) 0x1FC41A;
*pi = 20;
```

Bu işlem aşağıdaki gibi makine komutlarına dönüştürülebilir (örneğimizdeki pi, pi göstericisinin adresini belirtiyor olsun):

```
MOV dword [pi], 1FC41A
MOV EAX, [pi]
MOV dword [EAX], 20
```

Anahtar Notlar: Yukarıdaki işlem aslında C'de tanımsız davranışa yol açmaktadır. Ancak ne olursa olsun geçerlidir. Burada onun tanımsız davranışa yol açmasını dikkate almayınız.

Anahtar Notlar: Sembolik makine dillerinde (ve tabii doğal makine dillerinde) değişkenlerin isimleri yoktur. Dolayısıyla biz onlara isimleriyle erişmeyiz. Biz ancak onlara onların adresleriyle erişebiliriz. Yani yüksek seviyeli dillerdeki değişken isimleri aslında programcılar tarafından uydurulmuş isimlerdir. Kaynak program derlendikten sonra artık çalıştırılabilen program bir değişken ismi içermez. Yüksek seviyeli dillerin derleyicileri belli adreslerden başlayan bilgileri bize belli bir isimle sunmaktadır. Tabii yazmaçlar bir adres belirtmezler. Onlar CPU içerisinde yeri belli olan küçük bellek bölgeleridir. Makine komutları bile yazmaçları bilmektedir. İşin aslı yazmaçların da makine komutlarında isimleri yoktur. Yazmaçlar ikilik sisteme dönüştürülmüş makine komutlarında numaralarla temsil edilirler.

Köşeli parantez içerisindeki yazmaçlar 16 bitlik yazmaçlar olabilir. Ancak 8 bitlik yazmaçlar olamaz. Örneğin:

```
ADD EAX, [BX]
```

komutu geçerlidir. Ancak:

```
ADD EAX, [BL]
```

komutu geçersizdir. Köşeli parantez içerisine 16 bit yazmaç yerleştirmek 32 bit programlamada çok nadir görülebilecek bir durumdur. Çünkü 16 bit yazmaç ile ancak belleğin tepesindeki 64K'ya erişebiliriz. (Ayrıca önceki bölümlerde de belirtildiği gibi 32 bit korumalı modda köşeli parantez içerisine 16 bit yazmaç yerleştrmek için komutun başında 0x67 önekinin bulunması gerektiğini anımsayınız. Bu önek komutu bir byte büyütmektedir.)

32 bit Intel işlemcilerinde her yazmacın köşeli parantezler içerisinde bulundurulamayacağını yukarıda belirtmiştik. 16 bit yazmaçlardan yalnızca SI, DI, BX ve BP yazmaçları, 32 bit yazmaçlardan da EAX, EBX, ECX, EDX, ESI, EDI, EBP ve ESP yazmaçları köşeli parantezler içerisinde bulundurulabildiğini anımsayınız.

6) Bir yazmaçla "köşeli parantez içerisindeki bir yazmaç + 8 bit sabit" ya da bir yazmaçla "köşeli parantez içerisindeki bir yazmaç + 32 bit sabit" işleme sokulabilir. Bu gösterimi Intel [base + $disp_8$] ve [base + $disp_{32}$] ile belirtmektedir. Buradaki disp "displacement" sözcüğünden gelmektedir. Örneğin:

```
ADD EAX, [EBX + 1F]           ; yazmaç + 8 bit
ADD EAX, [EBX + 1001FC01]     ; yazmaç + 32 bit
```

Burada köşeli parantezin içindeki ifadeden bir bellek adresi elde edilmektedir. Bu bellek adresi oradaki yazmacın içerisindeki değerle o sabit değerin (displacement) toplamıyla oluşturulmaktadır. Örneğin:

```
MOV EBX, 1FC010
MOV EAX, [EBX + 1C]
```

Burada EBX değerinin içerisindeki değerle 1C değeri toplanarak bellek adresi elde edilmiştir. Yani işlemci 1FC010 + 1C = 1FC02C adresinden başlayan 32 bit değeri EAX yazmacına yerleştirir.

Köşeli parantez içerisinde 16 bit yazmaç kullanılırsa sabit değer (displacement) 8 bit ya da 16 bit olabilmektedir. (Fakat bu biçimdeki komutların önüne 00x67 öneki getirildiğini anımsayınız. Bu da makine komutunu bir byte büyütecektir. Zaten 32 bit ptogramlamada bu biçimdeki komutlarla fazlaca karşılaşmayız). Örneğin:

```
MOV     EAX, [BX + 1FC0]
```

Burada BX yazmacının içerisindeki değerle 16 bitlik 1FC0 değeri toplanarak bellek adresini oluşturmaktadır. (Komutun başında 0x67 öneki bulunacaktır.)

7) Bir yazmaçla "köşeli parantez içerisindeki iki yazmaç toplamı" işleme sokulabilir. Örneğin:

```
ADD EAX, [EBX + ECX]
```

Burada EAX yazmacının içerisindeki değer bellekte EBX ve ECX yazmaçlarının içerisindeki değerlerin toplamıyla belirtilen adresteki 32 bit değer ile toplanmaktadır. Sonuç EAX yazmacına aktarılmaktadır. Örneğin bir dizinin başlangış adresi EBX'te olsun. ECX'te de 0 değerinin bulunduğunu varsayalım:

```
MOV EAX, [EBX + ECX]
ADD ECX, 4
...
MOV EAX, [EBX + ECX]
ADD ECX, 4
...
MOV EAX, [EBX + ECX]
ADD ECX, 4
```

(Yine 32 bit korumalı modda bellek operandı ve bit düzeyini belirten yazmaçlar 16 bit olabilir. Bu durumda 0x66 ve 0x67 önekleri gerekecektir.)

8) Bir yazmaçla "köşeli parantez içerisindeki "yazmaç + yazmaç + 8 bit sabit" ya da "yazmaç + yazmaç + 32 bit sabit" işleme sokulabilir. Örneğin:

```
MOV EAX, [EBX + ECX + 1C] ; yazmaç + yazmaç + 8 bit sabit
```

Burada EAX yazmacına EBX, ECX yazmacının içindeki değerlerin toplamıyla 1C değerinin toplanması sonucunda elde edilen bellek adresindeki 32 bit değer aktarılmaktadır.

ya da örneğin:

```
MOV EAX, [EBX + ECX + 001A121C]   ; yzmaç + yazmaç + 32 bit sabit
```

9) Yukarıdaki 7'inci ve 8'inci kalıplarda toplama işlemine sokulan yazmaçlardan yalnızca bir tanesi iki ile, 4 ile ya da 8 ile çarpılabilir. Örneğin:

```
MOV EAX, [EBX + ECX * 4]
```

ya da örneğin:

```
MOV     EAX, [EBX + ECX * 4 + 1C]
```

ya da örneğin:

```
MOV EAX, [EBX + ECX * 4 + 001a121C]
```

Burada ECX'in içeindeki değer 4 ile çarpılıp toplama işlemine sokulmuştur.

Yukarıdaki kalıplarda köşeli parantezler içerisinde hangi yazmaçların bulunabileceği konusunda da bazı kısıtlar vardır. Tüm ve kesin kurallar daha ileride komutların ikilik sistemdeki karşılıkları (operation codes) ele alınırken açıklanacaktır. Genel olarak EIP ve EFLAGS yazmaçlarının dışındaki 32 bit yazmaçların hepsi

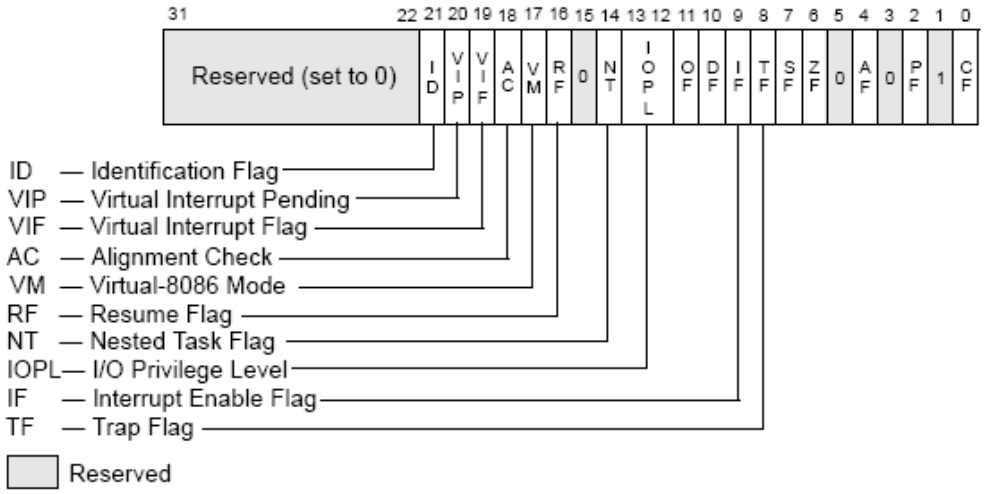
köşeli parantez içerisinde kullanılabilir. Fakat köşeli parantez içerisinde hem 32 bit yazmaçlar hem de 16 bit yazmaçlar toplama işlemine sokulamazlar. Ayrıca köşeli parantez içerisinde 16 bit yazmaçların kullanımı konusunda bazı ayrıntılar vardır.

Son olarak 32 bit sistemlerdeki komut kalıplarının tüm listesini vermek istiyoruz. Burada CMD herhangi bir komutu temsil ediyor olsun. Komutların operandlarının genel olarak yer değiştirebileceğini varsayabilirsiniz:

```
CMD         reg32, reg32
CMD         reg8, reg8
CMD         reg16, reg16        ; 0x66 öneki gerektirir

CMD         reg32, [sabit32]
CMD         reg32, [sabit16]    ; 0x67 öneki gerektirir
CMD         reg16, [sabit16]    ; 0x66 ve 0x67 önekleri gerektirir
CMD         reg16, [sabit32]    ; 0x66 öneki gerektirir
CMD         reg8, [sabit16]         ; 0x67 önekleri gerektirir
CMD         reg8, [sabit32]         ; 0x66 öneki gerektirir

CMD         reg32, [reg32]
CMD         reg32, [reg16]          ; 0x67 öneki gerektirir
CMD         reg16, [reg32]          ; 0x66 öneki gerektirir
CMD         reg16, [reg16]          ; 0x66 ve 0x67 önekleri gerektirir
CMD         reg8, [reg32]
CMD         reg8, [reg16]       ; 0x67 önekleri gerektirir

CMD         reg32, [reg32 + sabit8]
CMD         reg32, [reg32 + sabit32]
CMD         reg32, [reg16 + sabit8]     ; 0x67 öneki gerektirir
CMD         reg32, [reg16 + sabit16]        ; 0x67 öneki gerektirir
CMD         reg16, [reg32 + sabit8]     ; 0x66 öneki gerektirir
CMD         reg16, [reg32 + sabit32]    ; 0x66 öneki gerekitir
CMD         reg16, [reg16 + sabit8]     ; 0x66 ve 0x67 öneki gerektirir
CMD         reg16, [reg16 + sabit16]    ; 0x66 ve 0x67 öneki gerektirir
CMD         reg8, [reg32 + sabit8]
CMD         reg8, [reg32 + sabit32]
CMD         reg8, [reg16 + sabit8]      ; 0x67 öneki gerektirir
CMD         reg8, [reg16 + sabit16]     ; 0x67 öneki gerektirir
```

2.5. EFLAGS Yazmacının Bayrakları

Önceki konularda da belirtildiği gibi CPU içerisindeki bayrak yazmacının bitlerine bayrak (flag) denilmektedir. Bazı makine komutları bu bitler üzerinde değişiklik yaparlar. Bu bitlerin 1 durumuna çekilmesine "set" edilmesi, 0 durumuna çekilmesine "reset" edilmesi denilmektedir. EFLAGS yazmacının bitlerinin konumları şöyledir:



Hangi makine komutlarının hangi bayraklar üzerinde etkili olduğu Intel tarafından dokümante edilmiştir. Kursumuzda makine komutları incelenirken onların hangi bayrakları etkilediği üzerinde de durulacaktır. Burada önemli bir nokta şudur: Bir işlem eğer bir bayrağı etkiliyorsa ya onu set eder ya da reset eder. (Örneğin ADD makine komutu işlemin sonucu 0 ise ZF bayrağını set, ssıfır değilse reset etmektedir. İşlemin sonucu sıfır değilse onu önceki değerinde bırakmamaktadır.)

Bu bölümde yalnızca EFLAGS yazmacının bütün bayrakları değil yalnızca en önemli bayrakları (hangilerinin daha önemli olduğu da tartışılır tabii) ele alınacaktır:

ZF (Zero Flag): Son yapılan işlemin sonucu sıfır ise bu bayrak set edilir, sıfır değilse reset edilir. Örneğin:

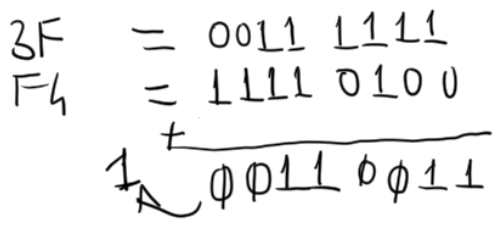
```
MOV EAX, 1
DEC EAX
```

Buradaki DEC makine komutu EAX yazmacının içerisindeki değeri 1 eksiltir. EAX yazmacındaki değer 1 olduğuna göre DEC komutundan ZF bayrağı set edilecektir.

CF (Carry Flag): İşaretsiz tamsayılar üzerinde işlemler yapılırken işlem sonucunda taşma ya da borç oluşursa oluşursa bu bayrak set edilmektedir.

```
MOV AH, 3F
MOV AL, F4
ADD AH, AL
```

Burada 3F ile F4 toplandığında sonuç 8 bite sığmamaktadır:



İşaretsiz sayıların çıkartılması durumunda borç oluşursa da (yani soldaki operand işaretsiz olarak sağdaki operand'tan küçükse) bu bayrak set edilmektedir. Başka bir deyişle CF bayrağı işaretsiz düzeydeki çıkartma işleminde borç oluşuyorsa set edilir, borç oluşmuyorsa reset edilir.

Örneğin:

```
MOV AH, 3F
MOV AL, F4
SUB AH, AL
```

Burada 8 bitlik toplama işlemi yapılmıştır. İşaretsiz düzeyde AH'taki değer AL'deki değerden (işaretsiz olarak) daha küçük olduğu için borç oluşur. Dolayısıyla işlem sonucunda CF bayrağı set edilecektir.

Toplama ve çıkartma işlemlerinin dışında diğer bazı makine komutları da CF bayrağı üzerinde etkili olmaktadır. Bu etki o komutların anlatıldığı bölümde ele alınacaktır.

OF (Overflow Flag): Bu bayrak sıklıkla CF bayrağı ile karıştırılmaktadır. OF işaretli tamsayılarda bir taşma ya da borç oluştuğunda set edilmektedir. Bu bayrağın resmi tanımı şöyle yapılabilir: Eğer bir işleme sokulan sayıların en soldaki bitleri (bildiğiniz gibi buna işaret biti de deniyor) aynı ise fakat işlem sonucunda elde edilen değerin en soldaki biti bunlardan farklı ise OF set edilir değilse reset edilir. (İşleme sokulan sayıların işaret bitleri farklı ise OF bayrağının her durumda reset edildiğine dikkat ediniz.) Örneğin:

```
mov eax, 0xFFFFFFF0       ;     1111 1111 1111 1111 1111 1111 1111 0000
mov ebx, 0x000000FF       ;     0000 0000 0000 0000 0000 0000 1111 1111
add eax, ebx           ; (1) 0000 0000 0000 0000 0000 0000 1110 1111
```

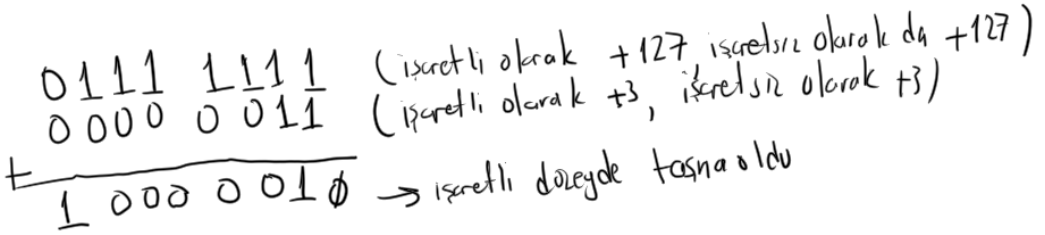
Burada add işleminde OF bayrağı reset edilecektir. Çünkü sayıların işaret bitleri farklıdır. Ancak CF bayrağının set edileceğine dikkat ediniz. Örneğimizdeki 0xF0000000 sayısı işaretli olarak 10'luk sistemde -16'dır. `0x000000FF` ise işaretli olarak 10'luk sistemde +255'tir. Toplama işleminin sonucunda 10'luk sistemde 239 sayısı elde edilecektir. Toplama işleminde işaretli düzeyde bir taşma olmadığına dikkat ediniz.

Fakat örneğin:

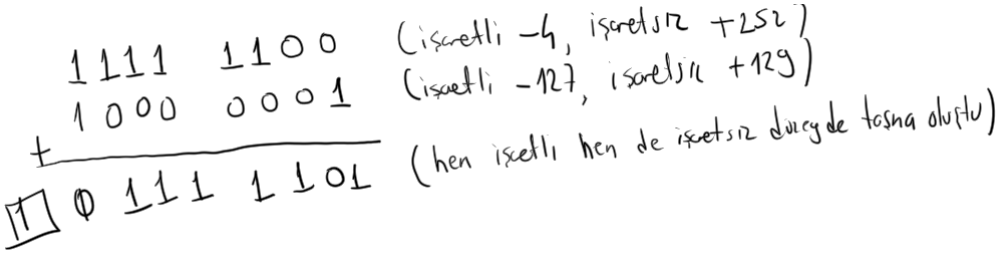
```
mov eax, 0xFFFFFFF0       ;     1111 1111 1111 1111 1111 1111 1111 0000
mov ebx, 0x80000000       ;     1000 0000 0000 0000 0000 0000 0000 0000
add eax, ebx           ; (1) 0111 1111 1111 1111 1111 1111 1111 0000
```

Buradaki add komutunda sayıların işaret bitleri (yani en soldaki bitleri) 1'dir (yani aynıdır). Fakat işlem sonucunda elde edilen değerin işaret biti 0'dır. Bu durumda OF set edilecektir. (Ayrıca CF bayrağının da set edileceğine dikkat ediniz.)

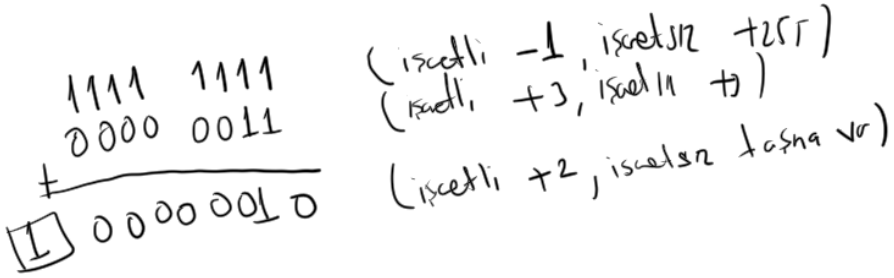
Pekiyi OF bayrağının programcı için anlamı nedir? OF bayrağı sayıların işaretli (signed) olduğu fikriyle işleme sokulması durumunda işlem sonucunda işaretli taşma ya da borç oluştuğunda set edilmektedir. OF bayrağı ile CF bayrağının birbirlerine benzediğine dikkat ediniz. CF bayrağı sayıların işaretsiz kabul edildiği durumda taşma ya da borç oluştuğunda set edilirken OF bayrağı sayıların işaretli kabul edildiği durumda taşma ya da borç olduğunda set edilmektedir. Örneğin 1 byte uzunluğunda aşağıdaki iki değeri toplayalım:



Görüldüğü gibi burada sayıları işaretli kabul ettiğimizde işaretli bir taşma oluşmuştur. Bu nedenle OF bayrağı set edilecektir. Ancak sayıları işaretsiz kubul ettiğimizde işaretsiz taşma oluşmamıştır. Bu nedenle CF bayrağı reset edilecektir. Örneğin:

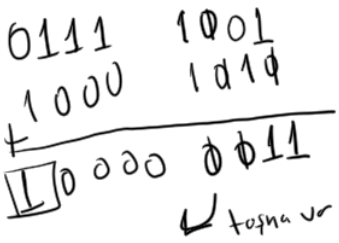


Burada syalar toplandığında hem işaretli düzeyde hem de işaretsiz düzeyde taşma oluşmaktadır. O halde hem OF hem de CF bayrakları set edilecektir. Örneğin:



Burada işaretli bir taşma oluşmamış ancak işaretsiz taşma oluşmuştur. O halde OF reset edilecek, CF ise set edilecektir.

AF (Auxilary Carry Flag): Bu bayrak 3'üncü bitten 4'üncü bite taşma oluşmuşsa set edilir, oluşmamışsa reset edilir. Yani bu bayrak düşük anlamlı 4 bitteki taşmaya bakmaktadır. Örneğin:



Burada 3'üncü bitten 4'üncü bite bir taşma oluşmuştur. O halde AF bayrağı set edilecektir. AF bayrağı BCD (Binary Coded Decimal) sayılarla işlem yaparken önemli olmaktadır. Onun dışında bu bayrağın bizim için bir önemi yoktur. Intel işlemcilerinde BCD sayılar üzerinde işlem yapan bazı makine komutları bulunmaktadır.

Anahtar Notlar: 10'luk sistemdeki sayılar her basamak 4 bit ile açılarak yazılırsa buna BCD kodlama denilmektedir. Örneğin 194 sayısını BCD olarak 0001 1001 0100 biçiminde kodlayabiliriz.

Parity Flag (PF): Bir işlemin sonucundaki 1 olan bitlerin sayısı çift ise PF bayrağı set edilmektedir, tek ise reset edilmektedir. PF bayrağı "parity" denilen hata kontrol (error check) mekznizması için düşünülmüştür. Bunun dışında bu bayrağın ciddi bir kullanım alanı yoktur.

Sign Flag (SF): Bu bayrak işlem sonucunda elde edilen değerin en soldaki bitini (işaret bitini) tutar. Başka bir deyişle işlem sonucunda elde edilen değerin işaret biti (en soldaki biti) 0 ise bu bayrak reset edilir, 1 ise set edilir. Örneğin bu bayrak sayesinde biz son yapılan işlemden elde edilen değerin negatif olup olmadığını anlayabiliriz.

Direction Flag (DF): Bu bayrak Intel'in bazı makine komutları tarafından set ve reset edilmektedir. Aynı zamanda string komutları denilen bir grup makine komutu bu bayrağa bakarak işlemin yönüne karar vermektedir.

3. Sembolik Makine Dili Derleyicileri ve Debugger'ları

Genel olarak sembolik makine dilleri sentaks ve semantik bakımlardan standardize edilmemektedir. Bu nedenle sembolik makine dili derleyicileri arasında önemli farklılıklar bulunabilmektedir. Örneğin tüm 80x86 işlemcileri aynı komut yapısına sahip olsa da bu komutların yazılış biçimleri ve çeşitli direktiflerin sentaksları ve anlamları sembolik makine dili derleyicisinden derleyicisine farklılık gösterebilmektedir.

Şimdi 80X86 işlemcileri için çok kullanılan sembolik makine dili derleyicileri hakkında bazı özet bilgiler verelim:

Microsoft MASM (Macro Assembler): Microsoft'un assembly derleyicisine MASM denilmektedir. Bu derleyici Visual Studio IDE'sinin (ya da Windows SDK'sının) bir parçası olarak Windows sistemlerine yüklenebilmektedir. Microsoft'un 32 bit assembly komut satırı derleyicisi "ml.exe", 64 bit komut satırı derleyicisi ise "ml64.exe" isimli programlardır. MASM DOS zamanlarında çok kullanılıyordu. Fakat son zamanlarda popülaritesi oldukça düşmüştür. Ancak yine 80X86 sistemleri için ana derleyicilerden biri olarak kabul edilmektedir.

Netwide Assembler (NASM): NASM özellikle son 10 yıldır çok popülarite kazanmıştır. Bunun en büyük nedenlerinden biri NASM'nin "cross platform" olmasıdır. (Yani örneğin NASM'nin hem Windows, hem Linux, hem BSD hem de MAC OS X sistemleri için derleyicisi vardır.) Biz de kursumuzda NASM derleyicisini kullanacağız. (Halbuki örneğin eskiden bu kursta MASM ve TASM derleyicileri kullanılıyordu.)

Borland Turbo Assembler (TASM): Borland DOS zamanlarında çok güçlü bir firmaydı. Onun C derleyicileri çok yaygın kullanılıyordu. TASM de Borland'ın assembly derleyicisi olarak MASM ile rekabet halindeydi. Ancak TASM artık programcılar tarafından tercih edilmemektedir. Zaten Borland TASM'yi artık başka bir ürün paketi içerisinde paralı olarak dağıtmaktadır. TASM sentaks bakımından neredeyse MASM'ye çok benzemektedir.

Flat Assembler (FASM): FASM de sentaks bakımından daha çok NASM'ye benzemektedir. Bu derleyici de "cross platform" özelliğe sahiptir. Ancak NASM kadar yaygın kullanılmamaktadır.

GNU Assembler (GASM): GASM GNU projesi kapsamında geliştirilmiş olan sembolik makine dili derleyicisidir. Bu nedenle Linux ve UNIX tabanlı sistemlerin ana assembly derleyicisi durumundadır. Ancak gerek sentaks yapısı bakımından gerekse özellik bakımından GASM pek çok kesim tarafından eleştirilmektedir. Kursumuzda temel düzeyde GASM sentaksı da gösterilecektir. Bu derleyicinin program ismi "as" biçimindedir.

3.1. NASM ve GASM Derleyicilerinin Yüklenmesi

NASM "sourceforge.net"te host edilen (https://sourceforge.net/projects/nasm/) açık kaynak kodlu bir derleyicidir. NASM'nin Windows için kurulum programı mevcuttur. Dolayısıyla Windows için yüklenmesi çok kolaydır. Yüklemedee sonra PATH çevre değişkeninin ayarlanması uygun olur. Böylelikle biz derleeyiciyi komut satırında herhangi bir dizinden çalıştırabiliriz.

NASM'yi Linux sistemlerinde kurmak için Debian tabanlı sistemlerde (Ubuntu, Mint gibi) aşağıdaki komut uygulanabilir:

```
sudo apt-get install nasm
```

Eğer dağıtım apt-get'i desteklemiyorsa kullanılan paket yöneticisine bağlı olarak komut değişebilir. Ya da ilgili dağıtımın "Software Manager" GUI arayüzü ile de yükleme yapılabilir.

NASM Linux sistemlerinin doğal bir parçası değildir. Ancak GASM (yani "as" derleyicisi") Linux

sistemlerinin doğal bir parçasıdır. Dolayısıyla uç bazı durumlar dışında GNU sembolik makine dili derleyicisi zaten Linux sistemlerinde hazır olarak bulunmaktadır.

3.2. Çok Kullanılan Debugger'lar

Bir programı çalışırken incelemek için kullanılan yazılımlara debugger denilmektedir. Debugger'lar genellikle hata bulma amacıyla kullanılırlar. (Etimolojik olarak "debug" "hata ayıklama" anlamına gelmektedir. "Bug" sözcüğünden türetilmiştir.) Debugger'lar dinamik analiz araçlarındandır. Bir programı çalıştırmadan analiz eden araçlara "statik analiz araçları", çalıştırarak analiz eden araçlara "dinamik analiz araçları" denilmektedir. Örneğin lint bir statik analiz aracıdır. Halbuki debugger'lar ve profiler'lar dinamik analiz araçlarıdır.

Debugger'lar çeşitli bakımlardan sınıflandırılabilmektedir. Örneğin bazı debugger'lar yüksek seviyeli dillerdeki kodları o dillere göre debug ederlerken (bunlara "source level debugger" da denilmektedir) bazıları makine kodu düzeyinde debug (bunlara "machine level debugger" da denilebilmektedir) yapabilmektedir. Bazı debugger'lar yalnızca "user mode" programları debug ederken bazıları "kernel mode" programları da debug edebilmektedir (bunlara "kernel mode debugger"lar da denilmektedir). Bazı debugger'lar komut satırından yönetilirler, bazıları GUI arayüzüne sahiptir. Bazıları uzaktaki makinelerdeki programları debug edebilirlern (bunlara "remote debugger" da denilmektedir) bazıları yalnızca o makinedeki programları debug edebilmektedir (bunlara "local debugger"lar da denilebilmektedir.) Bazı debugger'larda debug faaliyeti sırasında kod üzerinde değişiklik yapılabilmektedir.

Intel 80x86 işlemcileri için en yaygın kullanılan debugger'lar şunlardır:

Microsost Visual Studio Debugger: Visual Dtudio IDE'sisn içerisinde hem kaynak kod düzeyinde hem de makine kodu düzeyinde debug işlemi yapan bir debugger vardır. Aslında Microsoft eskiden Numega şirketinin "Code View" debugger'ını kullaıyordu. Sonra onu Visual Studio içerisine entegre etti. Visual Studio Debugger'ı "user level" bir debugger'dır.

Borland Turbo Debugger: Borland firmasının eski debugger'ı idi. Hala devam ettirilse de artık pek çok bakımdan demode olmuştur. Fakat DOS zamanlarında çok kullanılıyordu.

GNU Debugger (GDB): GNU projesi kapsamında gcc ile birlikte geliştirilmiş en önemli debugger'lardan biridir. GDB UNIX/Linux sistemlerindeki temel debugger'dır. Windows ve MAC OS X sistemleri için de port edilmiştir. Aslında pek çok GUI araç arka planda GDB kullanmaktadır. (Bunlara GDB'nin frontend'leri de denilmektedir.) Örneğin Netbeans, Eclipse, Qt-Creator, SASM arka planda gdb debugger'ını kullanmaktadır. GDB'de "user level" bir debugger'dır. Hem kaynak kod düzeyinde hem de makine kodu düzeyinde debug yapabilmektedir.

IDAPRO Debugger: IDAPRO profesyonel hem user level hem de kernel level debugger'dır. Pek çok hacking işleminde birincil debugger olarak tercih edilmektedir. Kaynak düzeyinde ve makine kodu düzeyinde debug yapabilmektedir. IDAPRO arka planda çeşitli debugger'lardan da faydalanmaktadır. IDAPRO eski SoftIce Debugger'ınıngeliştirilmiş biçimidir. Çok fazla özelliğe sahiptir. Bu nedenle kullanımı biraz zordur. IDAPRO bir GUI debugger'dır.

Microsoft WinDbg: Windows'un "kernel level debugger"ıdır. DDK (ya da yeni ismi ile WDK) paketi içerisinde onn bir parçası olarak bulundurulmaktadır. WinDbg ile programların kernel moddaki çalışması hakkında analizler yapılabilmektedir.

KDB ve KGDB: Bu debugger'lar temelde Linux için düşünülmüş "kernel level debugger"lardır.

Biz kursumuzda Visual Studio, IDAPRO ve GDB (frontend'leri dahil) debugger'larını kullanacağız.

3.3. NASM ile 32 Bit Windows Merhaba Dünya Programı

Tamamen sembolik makine diliyle yazılmış ekrana “Merhaba Dunya” yazısını çıkartan temel program şöyle yazılabilir:

; HelloWorld.asm

```
[BITS 32]

SECTION .data

msg        db  'Merhaba Dunya', 10
msg.written    dd  0

SECTION .text
    global _start
    extern _GetStdHandle@4, _WriteFile@20, _ExitProcess@4

_start:
    push    -11
    call    _GetStdHandle@4

    push    0
    push    msg.written
    push    14
    push    msg
    push    eax
    call    _WriteFile@20

    push    0
    call    _ExitProcess@4
```

Programı nasm ile aşağıdaki gibi derlenebilir:

```
nasm –f win32 HelloWorld.asm
```

Buradan ürün olarak HelloWorld.obj dosyası elde edilecektir. Komuttaki –f win32 seçeneği amaç kodun 32 bit Windows sistemleri için COFF formatında olmasını sağlar. Nasm “cross platform” bir derleyici olduğu için pek çok object module formatına göre derleme yapabilmektedir.

Program Microsoft’un link.exe programı ile aşağıdaki gibi link edilmelidir:

```
link /entry:start /subsystem:console HelloWorld.obj kernel32.lib
```

Komuttaki /entry:start seçeneği programın başlangıç noktasını belirleemek için kullanılır. HelloWorld.asm programının başlangıç noktası _start etiketinin bulunduğu yerdedir. Windows uygulamaları “GUI” ve “Console” olmak üzere ikiye ayrılmaktadır. Console uygulamalarında işletim sistemi programı yüklendiğinde bir console ekranını kendisi oluşturmaktadır. Console uygulaması için /subsystem:console seçeneği kullanılmalıdırç Link edilecek dosya HelloWorld.obj dosyasıdır. Merhaba Dunya programında kernel32.dll içerisindeki çeşitli API fonksiyonları (sistem fonksiyonları) kullanılmıştır. Bunun link aşamasına “kernel32.lib” isimli import kütüphanesinin dahil edilmesi gerekmektedir.

Merhaba Dünya programının tepesindeki [BITS 32] direktifi derlemenin 32 bit sistem için yapılacağını belirtmektedir. Bu programda üç API fonksiyonu çağrılmıştır. Önce GetStdHandle API fonksiyonuyla console ekranının handle değeri elde edilmiş, sonra WriteFile API fonksiyonu ile oraya yazma yapılmıştır. WriteFile aslında dosyaya yazma yapan bir API fonksiyonudur. Ancak console ekranı da sanki bir dosyaymış gibi ele alınmaktadır. Prosesin sonlanması ExitProcess API fonksiyonuyla yapılmak zorundadır.

C'nin standart exit fonksiyonu zaten ExitProcess API fonksiyonu çağırır. Merhaba Dünya programının eşdeğer C karşılığı şöyledir:

```
#include <stdio.h>
#include <windows.h>

char msg[] = "Merhaba Dunya\n";
DWORD msg_written;

int main(void)
{
    HANDLE hConsoleOutput;

    hConsoleOutput = GetStdHandle(STD_OUTPUT_HANDLE);
    WriteFile(hConsoleOutput, msg, 14, &msg_written, NULL);

    ExitProcess(0);

    return 0;
}
```

Görüldüğü gibi Windows'ta ekrana bir yazının yazdırılması iki API fonksiyonu ile yapılabilmektedir. Halbuki pek çok uygulamada biz ekrana hiçbirşey yazdırmak istemeyebiliriz. Fakat yine ExitProcess API fonksiyonuyla düzgün bir çıkışı yapmamız gerekir. İşte ekrana birşey basmayan minimal bir 32 bit Windowsnasm programı şöyle yazılabilir:

```
; Minimal.asm

[BITS 32]

SECTION .text
    global _start
    extern _ExitProcess@4

_start:

    push    0
    call    _ExitProcess@4
```

Program aşağıdaki derlenerek link edilmelidir:

```
nasm -f win32 Minimal.asm
link /entry:start /subsystem:console Minimal.obj kernel32.lib
```

3.4. NASM ile 32 Bit Linux Merhaba Dünya Programı

nasm ile Linux sistemlerinde 32 bit Merhaba Dünya programı şöyle yazılabilir:

```
; helloworld.asm

[BITS 32]

SECTION .data

msg         db  "Merhaba Dunya", 10

SECTION .text
    global _start
```

```
_start:
    mov     eax, 4
    mov     ebx, 1
    mov     ecx, msg
    mov     edx, 14
    int     80h

    mov     eax, 1
    mov     ebx, 0
    int     80h
```

Derleme işlemi şöyle yapılmalıdır:

```
nasm –f elf32 helloworld.asm
```

Link işlemi de şöyle yapılmalıdır:

```
ld –m elf_i386 –o helloworld helloworld.o
```

Linux'taki Merhaba Dünya programı Windows'takine göre daha sadedir. Bunun nedeni Linux'ta sistem fonksiyonlarının dinamik kütüphaneden değil kesme yoluyla çağrılıyor olmasıdır. Merhaba Dünya programında ekrana yazı yazdırmak için "sys_write", prosesi sonlandırmak için de sys_exit isimli sistem fonksiyonları kullanılmıştır. Linux'taki sistem fonksiyonları 80h kesmesi ile çağrılır. Sistem fonksiyonları çağrılmadan önce onların parametreleri yazmaçlara yerleştirilmektedir. Her sistem fonksiyonunun bir numarası vardır. Çağrılacak sistem fonks,yonunun numaraları 32 bit sistemde EAX yazmacına yerleştirilir. Sonra sırasıyla EBX, ECX, EDX yazmaçlarına da fonksiyonun parametreleri yerleştirilmektedir. Örneğin sys_exit fonksiyonu şöyle çağrılmıştır:

```
mov     eax, 1
mov     ebx, 0
int     80h
```

```
void sys_exit(int exitcode);
```

sys_exit fonksiyonunun numarası 1'dir. Fobksiyonun tek parametresi vardır. O da prosesin exit kodunu belirtir. Bu argüman EBX yazmacına yerleştirilmiştir.

GNU projesi kapsamında geliştirilmiş olan Linux'un temel linker programı "ld" isimli programdır. ld programı kullanılırken –o seçeneği ile çalıştırılabilen dosyaya isim verilmiştir. Eğer link sırasında çalıştırılabilen dosyay isim verilmezse default olarak a.out ismi kullanılır. Ayrıca Linux sistemlerinde "ld" programı ile link işlemi yapılırken "entry point" verilmediğine dikkat ediniz. "ld" linker'ı default olarak "_start" adresini "entry point" olarak almaktadır.

Yine Linux için de ekrana birşey yazmayan çalışır çalışmaz düzgün bir biçimde sonlanan minimal bir nasm sembolik makine dili programı da şöyle yazılabilir:

```
; minimal.asm

[BITS 32]

SECTION .text
    global _start

_start:
    mov     eax, 1
    mov     ebx, 0
```

```
    int     80h
```

Derleme ve link işlemi de aşağıdaki gibi yapılmalıdır:

```
nasm -f elf32 minimal.asm
ld -m elf_i386 -o minimal minimal.o
```

4. 32 Bit Intel İşlemcilerindeki Temel Makine Komutları

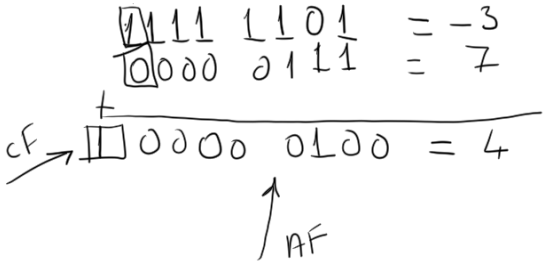
Bu bölümde 32 bit Intel işlemcilerinde çok kullanılan makine komutları ele alınacaktır. Giriş konularında da bahsedildiği gibi Intel 80x86 işlemcilerinin makine komutları zaten geriye doğru uyumludur. Yani burada ele alınacak makine komutlarının çoğu aslında 16 bit Intel işlemcilerinde de 64 bit Intel işlemcilerinde de aynı biçimde bulunmaktadır. Bu bölümde biz yalnızca tamsayılar üzerinde işlem yapan temel makine komutlarını göreceğiz. Intel'in gerçek sayılar üzerinde işlem yapan makine komutları -daha önceden de belirtildiği gibi- matematik işlemci tarafından yapılmaktadır. Matematik işlemcinin çalışma biçimi ana işlemciden oldukça faklıdır. Bu nedenle kursumuzda matematik işlemci komutları ayrı bir bölümde ele alınacaktır.

4.1. Temel Artimetik Komutlar

ADD Komutu: Bu komut toplama işlemi yapar. İşlemden OF, SF, ZF, AF, CF ve PF bayrakları etkilenmektedir. Toplama işleminin işaretli ya da işaretsiz biçimleri yoktur. ADD komutu her iki durumda da çalışır. Çünkü işaretli sayılar da işaretsiz sayılar da (2'ye tümleyen aritmetiğini anımsayınız) aynı biçimde toplanıp çıkartılırlar. Örneğin:

```
mov     ah, -3
mov     al, 7
add     ah, al
```

Burada add komutu sonucunda ah yazmacında 4 değeri bulunacaktır.



Burada sayıların işaretsiz olduğu varsayılarak yapılan toplama işlemi sonocunda "elde" oluştuğuna, fakat işaretli düzeyde taşma oluşmadığına dikkat ediniz. Bu nedenle işlem sonucunda CF set edilecek OF ise reset edilecektir.

Anahtar Notlar: Yukarıdaki örnekte de gördüğünüz gibi biz sembolik makine dillerinde sayıları 10'luk sistemde de belirtebilmekteyiz. Sembolik makine dili derleyicileri o sayıları ikilik sisteme dönüştürüp komutun ikilik sistem karşılığını oluşturmaktadır.

ADC Komutu: Bu komutun ADD komutundan tek farkı ayrıca sonuca bir de CF bayrağındaki 1'i ya da 0'ı eklemesidir. Yani komut şöyle çalışmaktadır:

DEST ← DEST + SRC + CF;

Bu komut da OF, SF, ZF, AF, CF ve PF bayraklarını etkilemektedir. ADC komutu özellikle işlemcinin yazmaç uzunluğundan fazla toplama işlemi yapmakta kullanılır. Örneğin 32 bit bir işlemcide 64 bit iki sayıyı önce düşük anlamlı 32 bitini ADD komutuyla sonra yüksek anlamlı 32 bitini de ADC komutuyla toplayarak toplayabiliriz. Örneğin 32 bit Intel işlemcilerinde toplanacak sayılar şunar olsun:

```
0x1FFF FFC3 1463 FF12
0x2100 3F12 4000 1000
```

Toplama işlemi şöyle yapılabilir:

```
mov ecx, 0x1463FF12
mov ebx, 0x40001000
add ebx, ecx
mov ecx, 0x1FFFFFC3
mov eax, 0x21003F12
adc eax, ecx
```

Toplama işlemi sonucunda elde edilen değerin yüksek anlamlı 32 biti eax, düşük anlamlı 32 biti ebx yazmacındadır.

SUB Komutu: Bu komut çıkarma işlemi yapar. Çıkartma işleminin de işaretli ve işaretsiz biçimleri yoktur. İşlemin işaretli olup olmadığı programcının varsayımına bağlıdır. Tabi programcı EFLAGS yazmacındaki bayraklara bakarak sonuç hakkında bazı sonuçlar çıkartabilir. İşlem sonucunda OF, SF, ZF, AF, PF ve CF bayrakları etkilenmektedir. Örneğin:

```
MOV eax, 0x80000000; işaretli olarak -2147483648, işaretsiz olarak +2147485648
SUB ebx, 20
```

Burada işaretsiz düzeyde bir borç oluşmadığı için CF bayrağı reset edilecek, işaretli düzeyde taşma oluştuğu için de OF bayrağı set edilecektir. Çıkartma işlemi sonucunda elde edilen değerin işaretli sayı sınırları içerisinde ifade edilemediğine (yani taşma oluştuğuna) dikkat ediniz.

SBB Komutu: Bo komut ADC komutunun tersini yapmaktadır. Bu komut önce normal çıkarma işlemini yapar, sonra ondan bir de CF bayrağındaki değeri çıkartır. Yani komut şöyle çalışmaktadır:

DEST ← DEST - SRC - CF;

Örneğin bu komut sayesinde biz 32 bit sistemde 64 bit iki sayıyı çıkartabiliriz. (Bunun için önce düşük anlamlı dört byte'ı SUB komutuyla, sonra da yüksek anlamlı dört byte SBB komutuyla çıkarırız). SBB işleminden OF, SF, ZF, AF, PF ve CF bayrakları etkilenmektedir.

MUL Komutu: Bu komut işaretsiz tamsayıları çarpmak amacıyla kullanılır. (Intel işlemcilerinde çarpma ve bölme işlemlerinin işaretli ve işaretsiz biçimleri farklı makine komutlarıyla yapılmaktadır) MUL komutunun bir operandı 8 bit çarpma yapılacaksa AL yazmacında, 16 bit çarpma yapılacaksa AX yazmacında, 32 bit çarpma yapılacaksa EAX yazmacında bulunmak zorundadır. Bu nedenle komutta bu yazmaç hiç belirtilemeyebilir. (Tabii komutta tek bir operandın belirtilmesi yanlış anlaşılmalara yol açabilmektedir. Pek çok assembly derleyicisi default operand olan akümülatörün belirtilmesini hata olarak değerlendirmemektedir) Bir byte'lık çarpmanın sonucu AX yazmacına, iki byte'lık çarpmanın sonucu DX:AX yazmaçlarına (yani yüksek anlamlı word DX yazmacına, düşük anlamlı word AX yazmacına) ve 32 bit çarpmanın sonucu da EDX:EAX yazmaçlarına (yani yüksek anlamlı dword eax yazmacına, düşük anlamlı dword edx yazmacına) aktarılır. Komutun işleyişi sembolik olarak şöyle gösterilebilir:

```
IF (Byte operation)
    THEN
         AX ← AL * SRC;
    ELSE (* Word or doubleword operation *)
         IF OperandSize = 16
              THEN
```

```
                    DX:AX ← AX ∗ SRC;
               ELSE IF OperandSize = 32
                    THEN EDX:EAX ← EAX ∗ SRC; FI;
          FI;
FI;
```

Örneğin:

```
mov     ebx, 10
mov     eax, 2
mul     ebx     ; eax'teki değer ebx'teki değerle ile çarpılıyor
```

Burada 32 bit çarpma işlemi yapılmıştır. Sonucun yüksek anlamlı 4 byte'ı EDX'te düşük anlamlı 4 byte'ı EAX'te bulunacaktur.

Örneğin:

```
mov ebx, 2
mov eax, 3000000000
mul ebx     ; eax'teki değer ebx'teki değerle çarpılıyor
```

Burada sonuç 6000000000 çıkacaktır. Ancak bu değer EAX içerisine sığmaz. İşte EDX 6000000000 sayısının yüksek anlamlı dört byte'ını EAX ise düşük anlamlı dört byte'ını tutar.

MUL komutundan SF, ZF, AF ve PF bayrakları etkilenir. Ayrıca eğer sonucun yüksek anlamlı biti 1 ise OF ve CF bayrakları set, 0 ise reset edilmektedir.

MUL komutunda diğer operand sabit olamaz. Diğer operandın yazmaç ya da bellek operandı olması zorunludur.

IMUL Komutu: Bu komut tamsayıları işaretli olduğu varsayımıyla çarpar. Genel işleyiş yukarıdaki gibidir. Ancak IMUL komutu akümülatör dışında herhangi iki operandlı olarak da çalışabilmektedir. Bu komut da her zaman CF ve OF bayraklarını sıfıra çeker. MUL komutundan SF, ZF, AF ve PF bayrakları etkilenir ve eğer sonucun yüksek anlamlı biti 1 ise OF ve CF bayrakları set, 0 ise reset edilir.

Örneğin:

```
mov     al, -2
mov     bl, 4
imul    bl
```

Burada sonuç AX yazmacında bulunacaktır ve işaretli olarak -8 değeri (0xFFF8) elde edilecektir.

IMUL komutunda diğer operand sabit olamaz. Diğer operandın yazmaç ya da bellek operandı olması zorunludur.

DIV Komutu: Bu komut işaresiz tamsayılarda bölme işlemi yapar. Intel işlemcilerinde ayrıca mod alma makine komutu yoktur. Mod alma işlemi DIV ve IDIV makina komutlarının yan ürünü olarak elde edilir. 8 bit bölme için bölünecek değerin AX yazmacında, 16 bit bölme için DX:AX yazmaçlarında ve 32 bit bölme için ise EDX:EAX yazmaçlarında bulunuyor olması gerekir. İşlem sonucunda bölüm değeri 8 bit bölmede AL yazmacına, 16 bit bölmede AX yazmacına ve 32 bit bölmede de EAX yazmacına yerleştirilir. Bölümden elde edilen kalan da 8 bit bölmede AH yazmacına, 16 bit bölmede DX yazmacına ve 32 bit bölmede de EDX yazmacına yerleştirilmektedir. Örneğin:

```
mov     edx, 0
mov     eax, 5
```

```
mov     ebx, 2
div     ebx
```

Bu işlem sonucunda EAX yazmacında 2 değeri EDX yazmacında da 1 değeri bulunacaktır. Komutun işleyişi aşağıdaki sembolik kodla da ayrıntılı açıklanabilir:

```
IF SRC = 0
    THEN #DE; FI; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
    THEN
         temp ← AX / SRC;
         IF temp > FFH
              THEN #DE; (* Divide error *)
              ELSE
                   AL ← temp;
                   AH ← AX MOD SRC;
         FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
         THEN
              temp ← DX:AX / SRC;
              IF temp > FFFFH
                   THEN #DE; (* Divide error *)
              ELSE
                   AX ← temp;
                   DX ← DX:AX MOD SRC;
              FI;
         FI;
    ELSE IF Operandsize = 32 (* Quadword/doubleword operation *)
         THEN
              temp ← EDX:EAX / SRC;
              IF temp > FFFFFFFFH
                   THEN #DE; (* Divide error *)
              ELSE
                   EAX ← temp;
                   EDX ← EDX:EAX MOD SRC;
              FI;
         FI;
FI;
```

Aslında yukarıdaki sembolik koddan da görüldüğü gibi DIV komutunda bölüm değeri ilgili yazmaca sığmazsa exception oluşmaktadır. Ancak exception konusu kursumuzda çok ileride ele alınacaktır.

Bu komut CF, OF, SF, ZF, AF ve PF bayraklarını rastgele değiştirebilmektedir. Yani komuttan sonra bu bayrakların durumundan herhangi özel bir anlam çıkartılmamalıdır.

DIV komutunda diğer operand sabit olamaz. Diğer operandın yazmaç ya da bellek operandı olması zorunludur.

IDIV Komutu: Bu komut DIV komutunun işaretli biçimidir. Yani işaretli bölme yapmaktadır. Ancak burada önemli bir nokta kalan değerin negatif olabileceğidir. Yani örneğin biz -3 değerini 2'ye böldüğümüzde bölüm -1 ve kalan -1 elde edilir. (C ve C++'taki % operatörünün de böyle çalıştığına dikkat ediniz)

Örneğin:

```
mov         edx, -1             ; negatif sayının yüksek anlamlı bitleri 1 olmalı
mov         eax, -3
mov     ebx, 2
idiv    ebx
```

Burada işlem sonucunda EAX yazmacında -1 ve EDX yazmacında da -1 değeri görülecektir.

IDIV komutunda da DIV komutunda olduğu gibi CF, OF, SF, ZF, AF ve PF bayrakları rastgele değişebilmektedir. Yani komuttan sonra bu bayrakların durumundan herhangi özel bir anlam çıkartılmamalıdır.

Ayrıca IMUL'da olduğu gibi IDIV'in iki operandlı bir biçimi yoktur.

MOV Komutu: MOV komutu genel bir aktarım komutudur. Aktarımın hedefi bir CPU yazmacı olabilir ya da bir bellek bölgesi olabilir. Eğer hedef bellekteki bir bölge ise bunun köşeli parantez içerisinde belirtilmesi gerekir. MOV komutu ile sabit atamaları da yapılabilir. Kursun giriş bölümünde gördüğümüz gibi bellek ile bellek bölgesi arasında MOV ile atama yapılamaz. Tabii komutların genel kalıplarından da anımsanacağı gibi bellek bölgesine doğrudan sabit atanabilir. Örneğin:

```
mov eax, 10
mov ah, 20
mov dword [ebx], 20
mov [ebx], eax
mov [1FC1020], eax
mov eax, ebx
```

MOV makine komutu bayrakları etkilememekktedir.

Bazı işlemci ailelerinde aktarımın hedefi yazmaç ise aktarım komutları "LOAD" biçiminde, bellek ise "STORE" biçiminde isimlendirilmektedir. Genellikle bu ailelerdeki sembolik makine dillerinde komutlar LD ve ST biimde bulunur.

XCHG Komutu: Bu komut iki yazmaç ya da bir yazmaç ile bir bellek operandı alır. İki değeri yer değiştirir. Örneğin:

```
xchg    eax, ebx
```

Burada artık EAX'teki değer EBX'te, EBX'teki değer de EAX'te olacaktır. Örneğin:

```
xchgeax, [ebx]
```

Burada [ebx]'teki değer ile eax yer değiştirmektedir. Yani komutun sonucunda EBXadresiyle belirtilen yerde EAX'teki değer, EAX'te de EBX adresindeki değer bulunur. Tabii XCHG komutunun her iki operandı da bellek olamaz. Operandlardan herhangi birinin de sabit olması mümkün değildir.

LEA Komutu: En çok kullanılan komutlardan biri de budur. Bu komut köşeli parantez içerisindeki adresin sayısal değerini bir yazmaca aktarmak için kullanılır. Bu komutta hedef operand her zaman bir yazmaçtır. Kaynak operand ise köşeli parantezli olmak zorundadır. C dilindeki adres almak için kullanılan & operatörü çoğu kez bu komutla gerçekleştirilmektedir. Örneğin:

```
mov eax, [ebx + ecx]
```

Burada ebx ile ecx içerisindeki değerler toplanıp bir adres elde edilmiştir. O adresteki 4 byte bilgi eax'e atanmıştır. Fakat örneğin:

```
lea eax, [ebx + ecx]
```

Burada ebx ile ecx içeriisndeki değerler toplanmış, bu toplam değer eax'e atanmıştır. Aşağıdaki komut geçersizdir. Ancak geçerli olsaydı yukarıdaki komut bununla eşdeğer olurdu:

```
mov eax, ebx + ecx
```

Intel'de iki yazmaç toplamı ancak köşeli parantez içerisinde bulunabilir. Örneğin:

```
lea  eax, [ebx]
```

Burada ebx değeri eax' atanmıştır. Bu komut aşağıdaki komutla işlevsel olarak eşdeğerdir:

```
mov eax, ebx
```

Örneğin:

```
lea  eax, [ebx + 0x1F]
```

Burada eax'e ebx ile 0x1F değerinin toplamı aktarılmaktadır.

LEA komutunun bir bellek erişimi yapmadığına özellikle dikkat ediniz. Örneğin köşeli parantez içerisindeki adres tahsis edilmemiş bir bölgenin adresi olsa bile LEA oraya erişim yapmadığı için bir sorun oluşmaz.

LEA komutu da tıpkı MOV komutu gibi bayrakları etkilememktedir.

INC ve DEC komutları:

INC ve DEC tek operandlı bir makine komutudur. INC operandını bir artırır, DEC de bir eksiltir. Operand yazmaç olabilir ya da köşeli parantezli bellek bölgesi olabilir. Örneğin:

```
inc eax
dec ebx
inc ah
dec dword [ebx]
inc byte [ebx + ecx]
```

Komut OF, SF, ZF, AF ve PF bayrakları etkilenmektedir. Bu komutlar CF bayrağını etkilemezler.

CMP Komutu: CMP komutu yapılan iş olarak SUB komutunun aynısıdır. Ancak CMP komutu hedef operandı değiştirmez yalnızca çıkartma yapılmış gibi bayrakları etkiler. Örneğin:

```
sub eax, ebx
```

burada eax'ten ebx değeri çıkartılmıştır. Sonuç eax'te saklanır. Fakat örneğin:

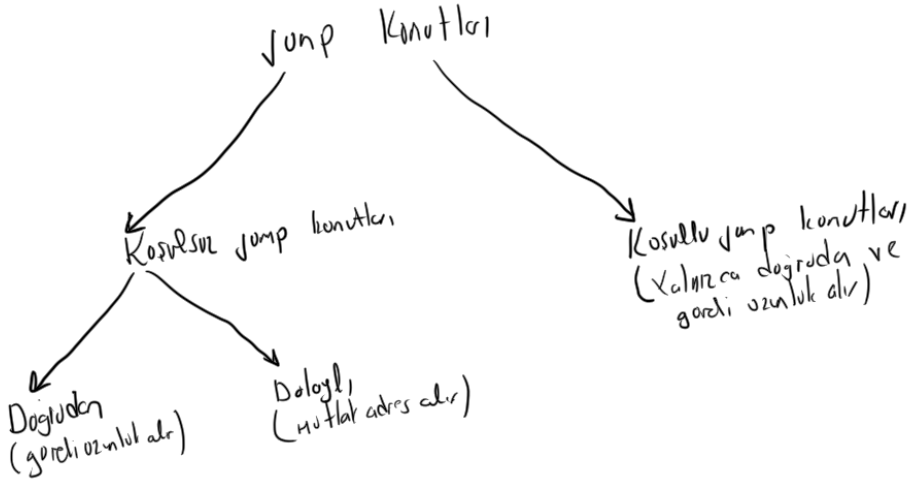
```
cmp eax, ebx
```

burada sanki çıkartma işlemi yapılmış gibi bayraklar etkilenir. Ancak gerçek bir çıkartma yapılmamaktadır. Dolayısıyla eax yazmacında bir değişiklik de yapılmayacaktır.

Sembolik makine dillerinde çıkartma çok önemlidir. Çünkü çıkartma işlemi sonucunda bayrakların konumuna bakılarak iki operand arasındaki büyüklük küçüklük ilişkisi anlaşılabilir. İki işaretsiz tamsayıyı karşılaştırmak isteyelim. SUB ya da CMP komutlarından sonra CF bayrağı set edilmişse bu durum birinci operand ikinci operand'tan küçük olduğu anlamına gelir. Tabii karşılaştırma işlemi için çıkartma yapıyorsak biz aslında operand'ların değerlerini de değiştirmek istemeyiz. İşte bu durumda SUB yerine CMP komutu tercih edilmektedir.

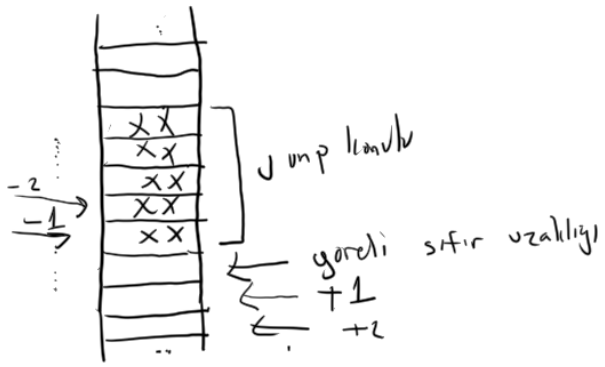
4.2. JUMP Komutları

Jump komutları sembolik makine dillerinin mutlaka gereken komutlarındandır. Bazı işlemci ailelerinde bu komutlar "branch (dallanma) komutları olarak isimlendirilmektedir. Jump komutları olmadan yüksek seviyeli dillerdeki if, switch, while for gibi deyimler gerçekleştirilemez. Jump komutları koşulsuz (unconditional) ve koşullu (conditional) olmak üzere ikiye ayrılmaktadır. Koşulsuz jump komutları C'deki goto deyimi gibidir. Koşulsuz olarak EIP yazmacını belli bir değere çeker. Koşulsuz jump komutlarının "doğrudan (direct)" ve dolaylı (indirect)" biçimleri de vardır. Koşulsuz jump komutlarının doğrudan biçimleri sonraki konuda ele alınacağı gibi "göreli (relative) uzaklık" değerini, dolaylı biçimleri ise "mutlak (absolute)" adres değerini operand olarak alır. Koşulu jump komutları bazı bayraklara bakarak jmp işlemini yapmaktadır. Koşulu jump komutlarının yalnızca doğrudan (direct) ve göreli uzaklık alan biçimleri vardır:



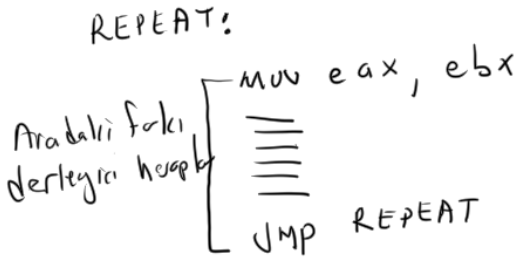
4.2.1. Intel'in Koşulsuz JMP Komutları

Koşulsuz jump komutları bayraklara bakmadan doğrudan EIP yazmacına belli bir değeri atayarak programın akışını başka bir adrese aktarmakta kullanılır. Koşulsuz jump komutları da kendi aralarında "doğrudan (direct)" ya da "dolaylı (indirect)" olmak üzere ikiye ayrılmaktadır. Doğrudan koşulsuz jump komutları pek çok işlemcide olduğu gibi göreli uzaklık değerini operand olarak alır. Doğrudan koşulsuz jump komutlarının son byte'larından sonraki ilk byte göreli uzaklık için sıfır orijini belirtir. Negatif uzaklık "yukarıya", pozitif uzaklık "aşağıya" jump yapılacağı anlamına gelmektedir.



Görüldüğü gibi koşulsuz doğrudan jump komutları operand olarak "göreli uzaklık" miktarını almaktadır.

Göreli uzaklıkların sembolik makine dili programcısı tarafından hesaplanması çok zordur. Assembly derleyicileri "etiket (label)" yöntemi ile bu hammaliyeyi bizim üzerimizden almaktadır. Biz sembolik makine dillerinde JMP komutlarının yanına bir etiket ismi veriririz. Sembolik makine dili derleyicileri de jmp komutunun sonundan o etiketin bulunduğu uzaklığı hesaplayarak makine komutu oluşturur. Örneğin:



Pekiyi JMP komutları operand olarak neden mutlak adres yerine göreli uzunluk almaktadır? Çünkü bu sayede biz kodu bellekte başka yere yüklesek de o jump komutları yine aynı yere atlamayı sağlayacaktır.

Koşulsuz jump komutlarının 8 bit, 16 bit ve 32 bit göreli uzunluk alan biçimleri de vardır. Tabii programcı özel bir belirleme yapmakdıktan sonra derleyici zıplama miktarını hesaplayarak en uygun jump komutunu üretir. Intel sisteminde 8 bit göreli uzunluk alarak yapılan jump işlemlerine "short jump", 16 bit ve 32 bit göreli uzunluk alarak yapılan jump işlemlerine de "near jump" denilmektedir. Pek çok sembolik makine dili derleyicisinde short ya da near anahtar sözcüğü ile bunu isterse programcı belirleyebilmektedir. Örneğin:

```
    jmp  near NEXT
        ; ...
NEXT:
       call  _ExitProcess@4
```

Eğer "short" ya da "near" anahtar sözcüklerinin hiçbiri kullanılmamışsa default durumda derleyici en uygun jump komutunu hesaplamaktadır. 32 bit sistemde 16 bit göreli uzunluk için komutta 0x66 öneki gerekmektedir.

Intel'deki koşulsuz jump komutlarının yazmaç ve bellek operandı alan biçimleri de vardır. Yazmaç operandı alan biçimi mutlak jump işlemi yapar. Yani yazmacın içerisindeki değer göreli uzunluk değil, bizzat jump edilecek adrestir. Örneğin:

```
mov eax, 0x123456
jmp eax
```

Burada koşulsuz olarak 0x123456 adresine jump yapılmaktadır. Koşulsuz jump komutlarının bellek operandı alan biçimleri de vardır. Bu durumda önce o bellek bölgesindeki 32 bit değer çekilir. Oraya mutlak jump uygulanır. Örneğin:

```
jmp    [eax]
```

Burada eax yazmacının içerisinde bulunan adresten 32 bit çekilerek o adrese jump yapılmaktadır. Intel jmp komutlarının operandları yazmaç ya da bellek ise böyle jump'ları "indirect jump" demektedir. Örneğin:

```
; Minimal.asm

[BITS 32]

SECTION .data

jmpPoint dd      EXIT

SECTION .text
    global _start
    extern _ExitProcess@4

_start:
    jmp     [jmpPoint]
```

```
EXIT:
    xor     eax, eax
    pusheax
  call  _ExitProcess@4
```

Burada EXIT etiketinin adresi jmpPoint adresindeki bellek bölgesine yazılmıştır:

```
SECTION .data

jmpPoint dd      EXIT
```

Sonra oraya dolaylı jump işlemi yapılmıştır:

```
jmp     [jmpPoint]
```

4.2.2. Koşullu Jump Komutları

Koşullu jump komutları bayraklara bakarak jump işlemi yapmaktadır. Intel'deki bütün koşullu jump komutları doğrudandır ve "göreli uzunluk" değerini operand olarak alırlar. Bayrakların uygun karşılaştırma sonuçlarını içermesi SUB ya da CMP komutlarıyla sağlanmaktadır. Bu nedenle önce bayrakların karşılaştırma için set ya da reset edilmesi gerekir. Yani koşullu jump komutları tipik olarak SUB ya da CMP komutlarından sonra uygulanmaktadır. Zaten onların isimlendirmeleri de SUB ya da CMP komutlarından sonra kullanılacağı fikriyle yapılmıştır. Intel'de çok fazla koşullu jump komutu varmış gibi görülse de aslında bazı komutlar diğerleriyle aynı işlemi yaparlar. Yani bu komutlar aynı makine kodunun farklı isimleridirler. Örneğin JA ile JNBE aslında aynı komutlardır. Bunlar tek bir komutun iki farklı isimleridir.

Koşullu jump komutlarının isimlendirilmeleri SUB ya da CMP komutlarının birinci operandına göre yapılmıştır. Örneğin:

```
cmp eax, ebx
jb  REPEAT
```

Burada jb (jump below) eax'teki değer ebx'teki değerden küçükse anlamına gelmektedir. İsimlendirmede "A (Above)" ve "B (Below)" işaretsiz tamsayılar için "G (Greater)" ve "L (Less)" de işaretli sayılar için kullanılmaktadır. Eşitlik "E (Equal)" ya da "Z (Zero flag set)" ile ifade edilebilmektedir.

Koşullu jump komutlarında eğer koşul sağlanmışsa jump işlemi yapılır. eğer koşul sağlanmamışsa sonraki komuttan devam edilir.

Aşağıda tüm koşullu jump komutlarının listesi verilmiştir:

Komut İsimleri	Anlamı	Bayrak Koşulları
JA/JNBE	İşaretsiz olarak birinci operand ikinci operand'tan büyüktür	CF = 0 ve ZF = 0
JB/JNAE/JC	İşaretsiz olarak birinci operand ikinci operand'tan daha küçük	CF = 1
JAE/JNB/JNC	İşaretsiz olarak birinci operand ikinci operand'tan büyük ya da eşit	CF = 0
JBE/JNA	İşaretsiz olarak birinci operand ikinci operand'tan küçük ya da eşit	CF = 1 veya ZF = 1
JE/JZ	İşaretli ya da işaretsiz olarak iki operand birbirine eşitse	ZF = 1
JNE/JNZ	İşaretli ya da işaretsiz olarak iki operand birbirine eşitse	ZF = 0

JG/JNLE	İşaretli olarak birinci operand ikinci operand'tan büyüktür	ZF = 0 ve SF = OF
JL/JNGE	İşaretli olarak birinci operand ikinci operand'tan daha küçük	SF ≠ OF
JGE/JNL	İşaretli olarak birinci operand ikinci operand'tan daha büyük ya da eşitse	SF = OF
JLE/JNG	İşaretli olarak birinci operand ikinci operand'tan küçük ya da eşitse	ZF = 1 veya SF ≠ OF
JO	Bir artimetik işlemde işaretli olarak taşma oluşmuşsa	OF = 1
JNO	Bir artimetik işlemde işaretli olarak taşma oluşmamışsa	OF = 0
JS	İşlem sonucu negatif çıkmışsa	SF = 1
JNS	İşlem sonucu pozitif ya da sıfır çıkmışsa	SF = 0
JP / JPE	Parity biti set edilmişse	PF = 1
JNP / JPO	Parity biti reset edilmişse	PF = 0
JCXZ	CX yazmacındaki değer sıfır ise	Bayraklara değil CX'e bakar
JECXZ	ECX yazmacındaki değer sıfır ise	Bayraklara değil ECX'e bakar

Sembolik makine dilinde döngüler ve if deyimleri koşullu ve koşulsuz jump komutlarının kullanılmasıyla gerçekleştirilirler. Örneğin aşağıdaki gibi bir C kodunun sembolik makine dili karşılığını yazmak isteyelim:

```
int g_x = 0;
/* ... */
while (g_x < 10) {
    printf("test\n");
   ++g_x;
}
```

Böyle bir while döngüsü aşağıdaki gibi oluşturulabilir:

```
[BITS 32]

SECTION .data

msg                 db  'test', 10
msg.written      dd  0

g_x     dd  0

SECTION .text
    global _start
    extern _GetStdHandle@4, _WriteFile@20, _ExitProcess@4

_start:
REPEAT:
    cmp     dword [g_x], 10
    jge     EXIT

    push -11
    call _GetStdHandle@4

    push 0
    push msg.written
    push 5
    push msg
    push eax
    call _WriteFile@20

    inc     dword [g_x]
```

```
    jmp     REPEAT

EXIT:
    xor     eax, eax
    push eax
  call   _ExitProcess@4
```

Döngüdeki en önemli nokta şurasıdır:

```
cmp     dword [g_x], 10
jge     EXIT
```

Burada g_x ile 10 değeri karşılaştırılmıştır. Eğer g_x 10'dan büyükse ya da 10'a eşitse while koşulu sağlanmadığı için döngüden çıkılmıştır. Eğer bu koşul sağlanmıyorsa akış aşağıdan devam eder. Orada da mesaj ekrana yazdırılmıştır. Dönünün devamının sağlanması için yukarıya jump yapıldıığına dikkat ediniz:

```
; ...
inc     dword [g_x]
jmp     REPEAT
```

Şimdi de aşağıdaki while döngüsünü sembolik makine dilinde yazmaya çalışalım:

```
unsigned g_x = 10;

while (g_x != 0) {
    /* ... */
    --g_x;
}
```

Kodun eşdeğer assembly karşılığı şöyle olabilir:

```
REPEAT:
   cmp      dword [g_x], 0
   je           EXIT
   ;  ...
   dec      dword [g_x]
   jmp      REPEAT
EXIT:
    ; ...
```

Tabii bu döngüyü şöyle de oluşturabilirdik:

```
cmp     dword [g_x], 0
je          EXIT

REPEAT:
   ; ...
   dec      dword [g_x]
   jnz      REPEAT
EXIT:
    ; ...
```

Burada dec komutuyla g_x değeri sıfıra düştüğünde ZF bayrağı set edileceğine dikkat ediniz. do-while döngüleri de benzer biçimde yapılabilir. Örneğin:

```
unsigned g_x = 10;

do {
   /* ... */
    --g_x;
} while (g_x != 0);
```

İşlemi sembolik makine dilinde şöyle yapılabilir:

```
REPEAT:
   ; ...
   dec   dword [g_x]
   jnz   REPEAT
```

for döngüleri de benzer biçimde sembolik makine dilinde oluşturulabilir. (Örneklerimizde stack görmediğimiz için hep global değişkenleri kullanıyoruz). Örneğin:

```
int g_i;
/* ... */
for (g_i = 0; g_i < 10; ++g_i) {
    /* ... */
}
```

Bu işlemin sembolik makine dili karşılığı şöyle oluşturulabilir:

```
   MOV           dword [g_i], 0
@2:
   cmp           dword [g_i], 10
   jge           @1

   ; ...

   inc       dword[g_i]
   jmp       @2
@1:
    ; ...
```

Anahtar Notlar: jmp işlemlerinde etiket kullanırken isim uydurmak zordur. Bu nedenle fabrikasyon etiket isimleri kullanılabilir. Sembolik mekine dilinde @ karakteri geçerli bir isimlendirme karakteridir. Biz örneklerimizde fabrikasyon etiket isimlerini @ karakterlerini kullanrak vereceğiz. Fonksiyonlara geçtiğimizde @ karakterlerini ayrıca fonksiyon isimleriyle de kombine edeceğiz. Pek çok C derleyicisi programın sembolik makine dili karşılığını oluştururken bu biçimdeki fabrikasyon etiketleri kullanmaktadır.

if deyimleri de yine koşullu ve koşulsuz jump komutlarıyla gerçekleştirilir. Örneğin yalnızca doğruysa kısmı olan bir if deyimi düşünelim:

```
if (g_i > 100) {
    /* ... */
}
/* ... */
```

Bu işlem aşağıdaki gibi sembolik sembolik makine dilinde ifade edilebilir:

```
   cmp      dword [g_i], 100
   jle          @1

    ; doğruysa kısmı

@1:
    ; if deyimin dışı
```

Şimdi de else kısmı olan bir if deyimini sembolik makine dili ile ifade etmeye çalışalım:

```
if (g_i > 10) {
    /* ... */
}
```

```
else {
    /* ... */
}
```

Bu işlemin eşdeğer sembolik makine dili karşılığı şöyle olabilir:

```
    cmp    dword [g_i], 10
    jle    @1

    ; if'in doğruysa kısmı

    jmp @2
@1:

    ; if'in yanlışsa kısmı

@2:
```

Şimdi de else-if örneği üzerinde duralım:

```
int g_a;

if (g_a > 0) {
    /* .... */
}
else if (g_a < 0) {
    /* ... */
}
else {
    /* ... */
}
```

Bu işlemin sembolik makine dili karşılığı şöyle oluşturulabilir:

```
  cmp      dword [g_a], 0
  jle          @1

  ; g_a > büyükse sıfır

  jmp      @3
@1:
  cmp      dword [g_a], 0
  jge      @2

  ; g_a < 0

  jmp      @3
@2:

  ; g_a == 0

@3:
```

4.2.3. Stack Kullanımı

Stack mikroişlemci tarafından yönetilen deposal bir alandır. Çalışan her programın (aslında her thread'in)

bir stack alanı vardır. Programların stack'lerinin belleğin neresinde ve hangi büyüklükte oluşturulacağı işletim sisteminin kontrolündedir. İşletim sistemi programı yüklediğinde onun için belli uzunlukta bir stack alanı tahsis eder.

Stack doğrudan mikroişlemci tarafından desteklenen LIFO (Last In First Out) prensibiyle çalışan bir kuyruk sistemidir. Stack'e eleman yerleştirmeye geleneksel olarak "push" işlemi, stack'ten eleman almaya da "pop" işlemi denilmektedir. Mikroişlemcilerin çoğunda bu işlemleri yapan PUSH ve POP komutları vardır.

Stack'in aktif noktası (buna stack'in tepesi (top of the stack) de denilmektedir) mikroişlemcilerde bir yazmaç tarafından tutulur. 32 bit Intel işlemcilerinde bu yazmaç ESP (SP "stack pointer" sözcüklerinden kısaltma) yazmacıdır. Stack'e eleman ekleme ve stackten eleman alma işlemleri genel olarak mikroişlemci kaç bitlikse o büyüklükte yapılır. Örneğin 32 bit Intel işlemcilerinde genellikle stack'e 32 bit yani 4 byte bilgi yerleştirilerek alınmaktadır. (Gerçi 64 bit Intel işlemcilerinde stack'e 16 bit, 32 bit ve 64 bit; 32 bit Intel işlemcilerinde de stack'e 16 bit ve 32 bit bilgi yerleştirip almak mümkün olsa da tipik durum işlemci kaç bitse stack'e o büyüklükte bilginin yerleştirilip alınmasıdır)

Intel'deki PUSH makine komutu tek operand'lıdır. Bu komut önce ESP yazmacını 4 geriye alır, sonra ESP'nin gösterdiği yere operand'ı ile belirtilen değeri yerleştirir. Örneğin:

```
PUSH        EAX
```

Bu komutla EAX içerisindeki değer stack'e yerleştirilecektir. ESP'nin komut öncesindeki değeri 0x123456 olsun. PUSH işlemi ile ESP önce 0x123452 değerine çekilir. Sonra oraya 4 byte olarak EAX yazmacındaki değer yerleştirilir. Bu biçimde PUSH yaptıkça ESP azalacaktır. PUSH işlemi sırasında ESP'nin yönü aşağıdan yukarıya doğrudur (bizim çizimlerimizde belleğin aşağısı yüksek adreste, yukarısı düşük adrestedir).

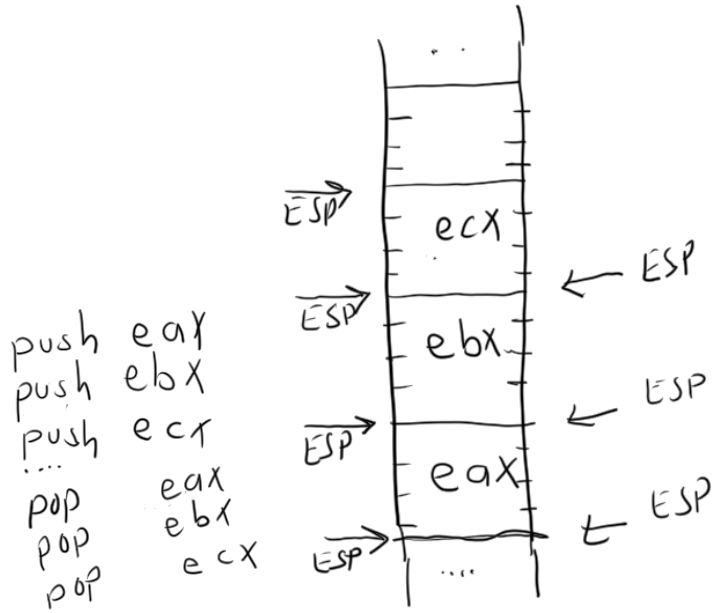
POP makine komutu tam tersi bir işlemi yapar. Önce ESP'nin gösterdiği yerden 4 byte değeri alarak operandına yerleştirir, sonra ESP'yi 4 byte artırır. Örneğin:

```
POP     EBX
```

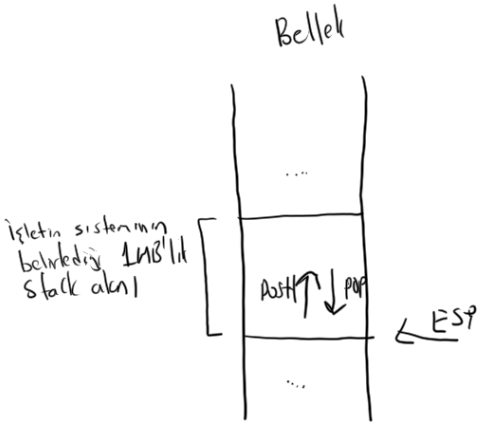
Komut öncesinde ESP'nin değeri 0x123452 olsun. Burada artık belleğin 0x123452 adresinden başlayan 4 byte'lık değer EBX'e yerleştirilir, sonra ESP 4 artırılarak 0x123456 değerine getirilir. Şimdi bu iki makine komutunu peşi sıra yazalım:

```
PUSH        EAX
POP     EBX
```

Bu işlem soncında ESP aynı değere geri gelir. Ancak EAX'teki değer EBX'e atanmış olmaktadır. Örneğin bir dizi PUSH ve POP yapalım:



Pekiyi işletim sisteminin program için stack alanını 1MB olarak belirlediğini düşünelim. İşin başında ESP yazmacı işletim sistemi tarafından nereye konumlandırılmalıdır? Yanıt: Tabii ki en sona! Çünkü başlangıçta stack'te hiçbir değer olmadığına göre bizim maksimum miktarda PUSH yapabilmemeiz gerekir.



Eğer biz stack için ayrılan alana dikkat etmeden aşırı derecede PUSH işlemi yaparsak stack alanı yukarıdan taşar. Buna İngilizce "Stack Overflow" denilmektedir. Eğer biz PUSH etmeden POP etmeye çalışırsak ya da PUSH ettiğimizden daha fazla POP etmeye çalışırsak bu kez bize ayrılan alanı aşağıdan taşırırız. Buna da İnglizce "Stack Underflow" denilmektedir.

32 bit Intel işlemcilerinde PUSH ve POP komutlarının tek operand'lı komutlar olduğuna dikkat ediniz. PUSH komutunun operand'ı bir sabit (immediate), bir yazmaç ya da bellek olabilir. (Bellek operandının köşeli parantez içerisinde hangi kalıplarla oluşturulabildiğini anımsaynız.) POP komutunun operand'ı da bir yazmaç ya da bellek olabilir. Örneğin:

```
PUSH        0x12345678
```

Burada 0x12345678 değeri stack'e push edilmiştir. Örneğin:

```
POP     EAX
```

Burada stack'in tepesindeki 4 byte'lık değer EAX'e yerleştirilmiştir. Örneğin:

```
PUSH      dword [EAX]
```

Burada EAX'in gösterdiği bellek adresindeki 4 byte oradan alınarak yine bellekteki stack'e push edilmiştir. Bu işlemin aslında iki bellek bölgesi arasında yapıldığına dikkat ediniz. Bu Intel'de bellek-bellek arası işlemin yapıldığı istisna durumlardan biridir. Örneğin:

```
POP    dword [EBX + ECX]
```

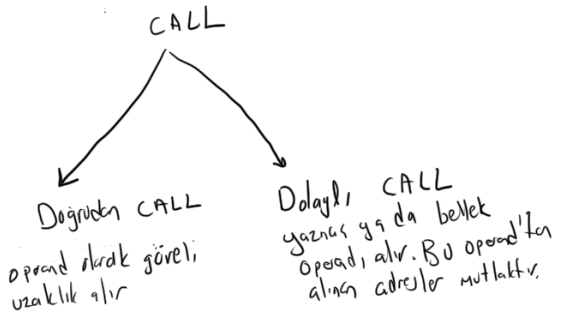
Burada stack'in tepesindeki 4 byte alınarak EBX + ECX ile belirtilen adrese yerleştirilmektedir.

4.2.3.1. Stack'in Anlamı Nedir ve Ne Amaçla Kullanılmaktadır?

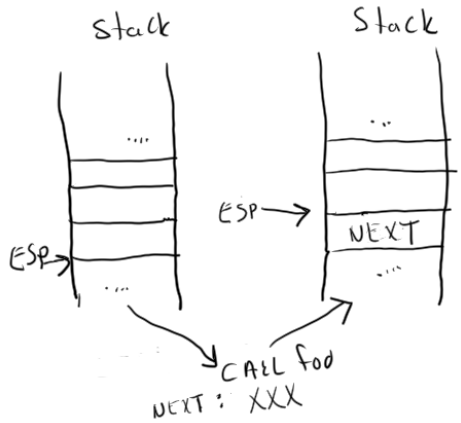
Stack son derece zekice düşünülmüş bir mekazizmadır. Bu sayede bilgilerin geçici olarak saklanıp geri alınabilmesi çok basit bir biçimde sağlanabilmektedir. Örneğin EAX yazmacının içerisindeki değeri kaybetmek istemeyelim. Fakat EAX'i de çarpma için kullanmak zorunda kalalım. İşte biz EAX değerini stack'e PUSH edebiliriz, çarpmayı yaptıktan sonra onu yeniden POP edebiliriz. Burada stack geçici bir saklama alanı olarak kullanılmıştır. Stack aynı zamanda sonraki başlıkta ele alınacağı gibi fonksiyon çağrılarında da işlemci tarafından kullanılır. Ayrıca fonksiyonlara parametre aktarımı, yerel değişkenlerin yaratılması ve yok edilmesi de stack mekanizmasıyla yapılmaktadır. Tüm bu kullanımlar sırası geldikçe ele alınacaktır.

4.2.4. Fonksiyonların Çağrılması CALL ve RET Makine Komutları

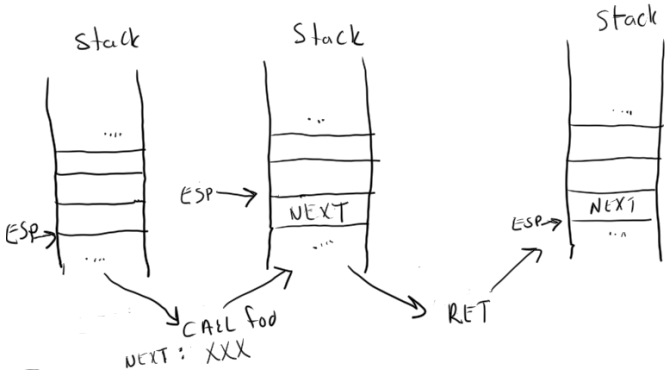
Anımsanacağı gibi yüksek seviyeli dillerde bir fonksiyon çağrıldığında akış fonksiyona gider. Fonksiyon sonlandığında çağrıldığı yerden akış devam etmektedir. İşte bu işlem sembolik makine dillerinde CALL ve RET makine komutlarıyla yapılmaktadır. CALL makine komutunun tıpkı koşulsuz JMP komutu gibi doğrudan ve dolaylı biçimleri vardır. Komutun doğrudan biçimi göreli uzaklık değerini operand olarak almaktadır. Yine komutun son byte'ından sonraki byte göreli uzaklıkta orijin (yani 0 noktası) kabul edilir. Tabii biz sembolik makine dillerinde CALL makine komutunun operandı olarak göreli uzaklığı değil call edilecek yerin etiketini veririz. Sembolik makine dili derleyicileri bu etiketi göreli uzaklığa dönüştürmektedir:



CALL makine komutu önce kendisinden sonraki ilk makine komutunun adresini stack'e push eder. Sonra hedeflenen adrese dallanır. Örneğin:



CALL makine makomutuyla akış hedeflenen adrese aktarıldıktan sonra oaradaki kod çalıştırılır. İşte geri dönüş için RET makine komutu kullanılmaktadır. Aslında RET makine komutunun yaptığı şey POP EIP gibidir. (POP EIP biçiminde bir komut geçerli değil). Yani stack'te ESP'nin gösterdiği yerdeki değeri POP eder ve onu EIP yazmacına yerleştirir. Böylece RET işlemiyle akış CALL makine komutundan sonraki komutla devam edecektir.



O halde CALL makine komutunun eşğdeğeri şöyle yazılabilir:

```
push        NEXT_CMD_ADDRESS
jmp     foo
```

Örneğin:

```
[BITS 32]

SECTION .data

msg          db   'foo', 10
msg.written       dd   0

SECTION .text
    global _start
    extern _GetStdHandle@4, _WriteFile@20, _ExitProcess@4

_start:
    call foo

    push 0
    call _ExitProcess@4
foo:
    push -11
    call _GetStdHandle@4

    push 0
    push msg.written
```

```
    push 4
    push msg
    push eax
    call _WriteFile@20

    ret
```

Yukarıdaki programa test.asm isminin verildiğini varsayalım. Derleme ve link işlemi şöyle yapılmalıdır:

```
nasm -fwin32 Test.asm
link /entry:start /subsystem:console Test.obj kernel32.lib
```

Burada foo fonksiyonu (yani foo adresinden başlayan kod) ekrana foo yazısını basmaktadır. Aynı işlemi Linux sistemlerinde şöyle yapabiliriz:

```
[BITS 32]

SECTION .data

msg         db  "foo", 10

SECTION .text
    global _start

_start:
  call    foo

  mov       eax, 1
    mov     ebx, 0
    int     80h
foo:

    mov     eax, 4
    mov     ebx, 1
    mov     ecx, msg
    mov     edx, 4
    int     80h
  ret
```

Derleme ve link işlemi de şöyle yapılabilir:

```
nasm -felf32 test.asm
ld -m elf_i386 -o test test.o
```

CALL makine komutunun dolaylı biçimi mutlak adresle işlemini yapar. Örneğin:

```
CALL    [EBX]
```

Burada bellekte EBX yazmacıyla belirtilen adresten 4 byte çekilecek ve o dört byte EIP yazmacına yüklenerek JMP işlemi yapılacaktır. Tabii bunun öncesinde yine sonraki komutun adresi stack'e atılacaktır.

Yukarıdaki örneklerde RET makine komutunun operandsız olduğu görülmektedir. Ancak bu komutun sabit bir değeri operand olarak alan RET n biçiminde bir versiyonu da vardır. (RET n makine komutu fonksiyonlardan dönüşlerde oldukça tercih edilmektedir ve bu komutun kullanımı ileride ele alınmaktadır.) RET n önce bir kez pop işlemi yaparak elde ettiği değeri EIP yazmacına yerleştirir fakat aynı zamanda ESP yazmacını n kadar da artırır. Bu durumda RET komutu ile RET 0 komutu işlevsel olarak eşdeğerdir.

Bir fonksiyon CALL ile çağrıldığında onun içerisinde de yine CALL ile başka fonksiyon çağrılabilir. Bu duurmda bir karışıklık olmaz. İlk RET işlemi sonraki çağrılan fonksiyondan geri dönüşü sonraki ret işlemi de ilk çağrılan fonksiyondan geri dönüşü sağlar.

Bir fonksiyonun içerisinde biz istediğimiz kadar PUSH işlemi yapabiliriz. Ancak RET işleminden önce ne kadar PUSH yapmışsak o kadar POP yapmış olmalıyız. Böylece Stack yazmacı (ESP) RET işlemi için yeniden geçerli durumuna geri dönebilsin.

4.2.5. Bit Düzeyinde İşlemler Yapan Makine Komutları

Sembolik makine dilleri alçak seviyeli diller oldukları için bu dillerde bit işlemlerine çok sık gereksinim duyulmaktadır. Bu bölümde temel bit işlemlerini yapan makine komutları ele alınacaktır.

4.2.5.1. AND ve OR Komutları

AND v e OR komutları iki tamsayı değerin karşılıklı bitlerini AND ve OR işlemlerine sokmaktadır. Örneğin:

```
and eax, ebx
or      eax, [ebx + ecx]
and eax, 1
```

Komut sonucunda her zaman OF ve CF bayrakları reset'lenir. AF bayrağının durumu tanımsızdır (undefined). İşlemden SF, ZF, PF bayrakları etkilenir.

AND işleminin operandlı etkilemeyen yalnızca bayrakları etkileyen TEST isimli bir biçimi de vardır. AND ile TEST arasındaki ilişki SUB ile CMP arasındaki ilişkiye benzetilebilir. TEST işlemi AND işlemi yapar fakat bu işlemden yalnızca bayraklar etkilenir. Pekiyi TEST işlemine neden gereksinim duyulmaktadır? Bazen AND işlemi sonucundaki bayrak değerlerini merak edebilirsiniz ancak operandı daa değiştirmek istemeeyebilirsiniz. Örneğin EAX'teki değeri bozmadan onun içerisindeki değerin tek mi çift mi olduğunu anlamak istediğinizi düşünün:

```
test  eax, 1
jz        EVEN        ; çift ise jump et
```

Bir değeri kendisiyle AND işlemine soktuğumuzda onunla aynı değeri elde ederiz. Fakat bu işlemden bayraklar etkileneceği için artık koşullu jump işlemi yapabiliriz. Örneğin EAX'teki değer negatifse jump etmek isteyelim. Henüz bir işlem yapmadığımıza göre bayraklara bakamayız. Bayrakları etkileyecek bir işleme ihtiyacımız vardır. Bu AND işlemi ya da TEST işlemi olabilir:

```
and     eax, eax
js          NEGATIVE        ; negatifse jump et
```

Bu işlemle EAX'teki değer negatif ise jump yapılmaktadır. Benzer biçimde:

```
test        eax, eax
jnz     NOTZERO
```

Burada da EAX'teki değer sıfır değilse jump yapılmıştır.

Anahtar Notlar: gcc'de test.c isiml C programını yalnızca derleyerek (yani link etmeyerek) ondan sembolik makine dili çıktısı şöyle elde edilir:

```
gcc -c -S -masm=intel test.c
```

Burada –c yalnızca derleme yapmak için, -S assembly çıktısı elde etmek için, -masm=intel ise sembolik makine dili çıktısının Intel sentaksına göre düzenlenmesi için kullanılmıştır. Üretilen dosya default olarak "test.s" olacaktır.

Yukarıdaki işlemin aynısı Microsoft'un cl.exe derleyicisi ile şöyle yapılabilir:

```
cl /c /FA test.c
```

Burada /c yalnızca derleme için /FA sembolik makine dili çıktısı için kullanılmaktadır. Üretilen dosya "test.asm" olacaktır.

4.2.5.2. XOR Komutu

XOR komutu iki değerin karşılıklı bitlerini EXOR işlemine sokar. EXOR (Exclusive OR) işlemi iki bit aynıysa 0 değerini farklıysa 1 değerini veren bir işlemdir. EXOR işlemi geri dönüşümlüdür. Bu nedenle EXOR şifreleme gibi işlemlerde çok tercih edilmektedir. (Yani biz bir değeri bir değerle EXOR çekmiş olalım. Sonucu yine aynı değerle EXOR işlemine sokarsak orijinal değeri elde ederiz.)

Bir değer kendisiyle EXOR işlemine sokulursa sıfır elde edilir. Bu nedenle assembly programcıları bir yazmacı sıfırlamak için bu komutu sık kullanmaktadır:

```
xor     eax, eax
```

Tabii aynı işlemin MOV makine komutuyla da yapılabilirdi:

```
mov     eax, 0
```

Bu durumda komutun daha uzun olacağına dikkat ediniz. (Sabit değerlerin makine komutunun bir parçası olarak koda dahil olduğunu anımsayınız)

Eskiden XOR işlemi SUB işleminden daha hızlıydı. Ancak uzunca bir süredir artık bunların arasında bir hız farkı kalmamıştır:

```
sub eax, eax
```

Komut sonucunda her zaman OF ve CF bayrakları reset'lenir. AF bayrağının durumu tanımsızdır (undefined). İşlemden SF, ZF, PF bayrakları etkilenir.

4.2.5.3. Öteleme (Shift) Komutları

C/C++, C# ve Java'daki << ve >> operatörleri aslında işlemcinin sola ve sağa öteleme komutlarını kullanmaktadır. Öteleme işlemleri Intel işlemcilerinde SAL, SAR, SHL ve SHR makine komutları ile yapılmaktadır. SAL (Shift Arithmetic Left) ve SAR (Shift Arithmetic Right) komutlarına aritmetik öteleme komutları SHL (Shift Logical Left), SHR (Shift Logical Right) komutlarına da mantıksal öteleme komutları denilmektedir. SAL ve SHL komutları arasında farklılık yoktur. SAR ile SHR komutları arasında ise küçük bir farklılık vardır.

Diğer yüksek seviyeli dillerden de bilindiği gibi sola bir kez ötelemede bütün bitler bir sola kaydırılır ve sağdan sıfır ile besleme yapılır. Sağa bir kez ötelemede ise bütün bitler bir sağa kaydırılır ancak en soldan 0 ile mi bir ile mi besleme yapılacağı SAR ve SHR komutlarında değişmektedir. SAR komutunda besleme işaret bitiyle, SHR komutunda ise her zaman 0 ile yapılmaktadır. SAL ve SHL komutları arasında aslında hiçbir fark yoktur. Mantıksal bütünlük oluşturmak için sanki iki farklı komut varmış gibi isimlendirme yapılmıştır. (Yani aslında SAL ve SHL iki ayrı makine komutu değil aynı komutun iki farklı ismidir.) Bir'den fazla kez ötelemede aynı işlemler birden fazla yapılmaktadır.

Öteleme komutlarında ötelenecek operand yazmaç ya da bellek olabilir. Bir kez öteleme için 2 byte'lık bir komut versiyonu (opcode) bulundurulmuştur. Bir'den fazla öteleme yapılmak isteniyorsa öteleme sayısı ya sabit olarak verilmek zorundadır ya da CL yazmacına yerleştirilmek zorundadır. Öteleme miktarının sabit olarak verilmesi durumunda ise komut uzunluğu 3 byte olur. Eğer komut uzunluğu CL yazmacına yerleştirilirse bu durumda komut uzunluğu yine 2 byte'tır. Örneğin bazı geçerli ötleme komutları şöyle olabilir:

```
sal     eax, 1                      ; komut uzunluğu 2 byte
```

```
shl     dword [ebx], 5              ; komut uzunluğu 3 byte
sar     byte [ebx + ecx], cl        ; komut uzunluğu 2 byte
```

Aşağıdaki komutlar ise geçersizdir:

```
sal    eax, bl       ; geçersiz!
sal    ebx, ecx      ; geçersiz!
```

Komutların bayrakları etkilemesi şöyle olmaktadır: Her zaman kaybedilen bit CF bayrağına yerleştirilir. Yani örneğin biz sola bir kez öteleme yaptığımızda en soldaki bit CF bayrağına yerleşecektir. Sağa bir kez öteleme yaptığımızda da en sağdaki bir CF bayrağına yerleşir. Birden fazla öteleme yapıldığında son ötelemede kaybedilen bit CF'de kalır. SHL ve SHR komutlarında öteleme sayısı ötelenmek istenen değerin bit uzunluğununun bir eksiğini aşıyorsa bu durumda CF bayrağı tanımsız durumdadır. OF bayrağı yalnızca 1 kez ötelemede etkili olur. Birden fazla kez ötelemede OF de tanımsız durumdadır. Bir kez ötelemede OF bayrağı bize sayıda işaretli taşma olup olmadığını bildirmektedir. Ayrıca SF, ZF ve PF bayrakları normal biçimde işlemden etkilenirler. Sıfır kere öteleme geçerlidir. Ancak bu durumda işlemden bayraklar etkilenmez.

Sola öteleme bilindiği gibi iki ile çarpma anlamına, sağa öteleme ise iki ile bölme anlamına gelmektedir. İşaretli sayıların sağa ötelenmesi için SAR komutu işaretsiz sayıların sağa ötelenemesi için SHR komutu kullanılmaktadır. İşaretli ya da işaretsiz sola öteleme için aslında yukarıda da belirtildiği gibi tek bir komut vardır. Bu komuta SAL ve SHL isimleri verilmiştir.

Mantıksal sağa ötelemede en soldan beslemenin 0 ile artimetik sağa ötelemede ise 1 ile yapıldığını anımsayınız. Örneğin AL yazmacında aşağıdaki değerin bulunduğunu düşünelim:

```
AL: 1011 0111
```

Şimdi AL yazmacındaki değeri SAR AL, 1 ile artimetik olarak bir kez sağa öteeleyelim. AL'deki değer şu hale gelecektir.

```
AL: 1101 1011
```

Oysa AL'ye SHR al, 1 komutuyla mantıksal sağa öteleme uygulasaydık AL'deki değer şu hale gelecekti:

```
AL: 0101 1011
```

Bildiğiniz gibi sağa öteleme sayıyı tamsayısal olarak (yani nokta oluşmayacak biçimde) ikiye bölme anlamına gelir. Fakat aritmetik sağa ötelemede eğer ötelenecek değer negatifse sonuç küçültülecek biçimde (yani eksi sonsuza doğru) tamsayı olarak elde edileceğine dikkat ediniz. Yani örneğin biz -3 değerini bir kez sağa aritmetik ötelemek isteyelim:

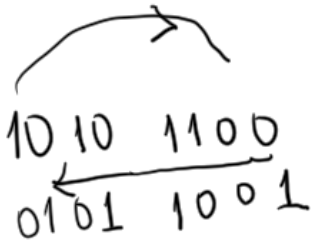
```
1111 1101       -3
```

Sağa bir kez artimektik ötelendiğinde sayının -2 olduğuna dikkat ediniz:

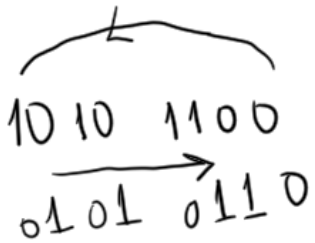
```
1111 1110       -2
```

4.2.5.4. Döndürme (Rotate) Komutları

Döndürme işlemi için C/C++ dillerinde (Tabii Java ve C#'ta da) bir operatör bulundurulmamıştır. (Bu nedenle bu işleme öteleme işlemlerindeki gib bir aşinalığınız bulunmayabilir.) Döndürme işlemi ötelemeye benzemektedir. Ancak ötelemede kaybedilen bit diğer tarafı beslemede kullanılır. Örneğin sola döndürme işlemini şöyle gösterilebiliriz:



Sağa döndürme işlemini de şöyle gösterebiliriz:

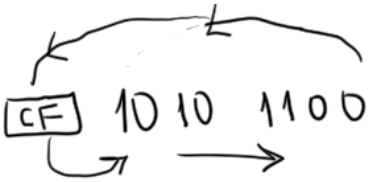


Döndürme işlemi bir sayının belli kısımlarının yer değiştirilmesi için kullanılabilmektedir. Ayrıca sayıyı ötelemek yerine döndürdüğümüzde biz onu yeniden ters döndürerek eski haline de getirebiliriz. Diğer özel bazı durumlarda da döndürme işleminden faydalanılmaktadır.

Intel işlemcilerinde dört döndürme (rotate) komutu vardır. Komutların ikisi carry'li döndürme için diğer ikisi de carry'siz döndürme için kullanılır:

ROR (Rotate Right)
RCR (Rotate Carry Right)
ROL (Rotate Left)
RCL (Rotace Carry Left)

Carry'li döndürmede sanki CF bayrağı sayının en yüksek anlamlı ekstra biti gibi davranmaktadır. Dolayısıyla döndürmeye o da dahil edilir. Örneğin sağa bir kez carry'li döndürmeyi şöyle gösterebiliriz:



Rotate komutlarının biçimleri de tamamen öteleme komutları gibidir. Yani bir kez döndürme için ayrı bir makine komutu vardır. Birden fazla kez döndürme sabitle ya da CL yazmacıyla yapılabilmektedir. Komut uzunlukları da yine öteleme komutlarında olduğu gibidir. Örneğin bazı geeçerli döndürme komutları şöyledir:

```
ror    eax, 1
rcl    eax, 4
rcr    eax, cl
```

CF bayrağı her zaman (ister carry'li döndürme ister carry'siz döndürme söz konusu olsun) son döndürmede kaybedilen biti tutar (tıpkı ötelemede olduğu gibi). Ancak döndürme döndürülecek değerin bit uzunluğunun bir eksiğinden fazla olursa bu bayrak tanımsız durumda olur. OF bayrağı yine yalnızca bir kez öteleme söz konusu olduğunda etkilenir. SF, ZF, AF, PF bayrakları ise normal biçimde etkilenmektedir.

4.2.5.5. BT (Bit Test) BTS (Bit Test Set), BTR (Bit Test Reset) ve BTC (Bit Test Complement)

Komutları

Bu komutlar Intel ailesine 80386 ile eklenmiştir. Dolayısıyla 80386'dan sonraki tüm işlemcilerde bu komutlar bulunmaktadır. Bilindiği gibi bazı olgular var-yok biçiminde ikil değer alırlar. Örneğin 256 tane dosyanın açık olup olmadığı bilgisi, 512 tane makinenin çalışır durumda olup olmadığı bilgisi gibi. Bu biçimdeki bilgiler bit düzeyinde saklanmaya çok uygundurlar. İşte BTS komutu bir bit dizisinin herhangi numaralı bir bitini set etmek (1'lemek) için, BTR komutu reset etmek (0'lamak) için BTC komutu da terslemek (yani o bit 0 ise 1 yapmak, 1 ise 0 yapmak) için kullanılmaktadır. Bazen biz bir bit dizisinin belli bir bitini değiştirmek istemeyiz. Yalnızca o bitin durumunu (yani 0 mı 1 mi olduğunu) öğrenmek isteriz. BT (Bit Test) komutu da bunu yapmaktadır.

Yukarıdaki komutlarının hepsinin genel biçimi aynıdır. Bu komutların birinci operandları yazmaç ya da bellek, iki perand'ları yazmaç ya da sabit olabilir. Örneğin:

```
bts     eax, 14
btr     [ebx], eax
btc     [eax + ebx], 19
```

Komutların ilk operand'ları bit işleminin yapılacağı hedefi, ikinci operand'ları ise bit numarasını belirtmektedir. Bitlere ilk bit 0 olmak üzere artan sayıda bir numaraları karşılık getirilmiştir. Eğer ikinci operand sabit ise ya da birinci operand yazmaç ise en son bit numarası birinci operand'ın bit uzunluğunun bir eksiği kadar olabilir. (Bu durumda eğer ikinci operanda daha büyük bir bit numarası verilirse bit numarası olarak birinci operandın bit uzunluğuna bölümünden elde edilen kalan değeri kullanılır.) Örneğin:

```
bts     eax, 32
```

komutu aslında,

```
bts     eax, 0
```

ile aynı işleve sahiptir. Ya da örneğin:

```
bts     byte [eax], 12
```

komutu aslında,

```
bts     byte [eax], 4
```

ile aynı işleve sahiptir.

Eğer bu komutların birinci operandı bellek, ikinci operandı yazmaç ise komutun davranışı ilginçtir. Bu durumda işlem sanki birinci operand ile belirtilen adresten itibaren sınırsız bir bit dizisi varmış gibi yapılır. İkinci operand büyük bir pozitif değer ya da büyük bir negatif değer olabilir. Örneğin:

```
mov eax, 1345
bts [data], eax
```

Burada bellekte data adresinden başlayan bölge bir bit dizisi olarak kullanılmaktadır. Bu bit dizisinin 1345'inci biti set edilmiştir. Burada birinci operand için byte, word, dword gibi bir belirleyici getirilmediğine dikkat ediniz. Bu komutların bit numarasını sabit olarak verdiğimiz biçimlerinde birinci operandın uzunluğunun belirtilmesi gerektiğine de dikkat ediniz. Örneğin:

```
bts dword [data], 29
```

BTS, BTR ve BTC komutları işlem yapılan bitin değişmeden önceki değerini CF bayrağına yerleştirmektedir. BT komutu da ilgili bitin değerini CF bayrağına yerleştirir. Bu komutların hiçbiri ZF

bayrağını etkilemez. OF, SF, AF ve PF bayraklarının durumu bu komutlardan sonra tanımsızdır (undefined).

4.2.5.6. BSF (Bit Scan Forward) ve BSR (Bit Scan Reverse) Komutları

Bu komutlar bir yazmaç ya da bellek bölgesindeki bitlerden ilk 1'olanın ya da son 1 olanın indeksini elde etmek için kullanılmaktadır. Komutların birinci operandları yazmaç ikinci operand'ları ise yazmaç ya da bellek bölgesi olmak zorundadır. İkinci operand'lar 1 olan bitin araştırılacağı kaynak değeri, ikinci operand'lar ise bulunan indeks numarasının yerleştirileceği yazmacı belirtmektedir. BSF ilk 1 olan bitin numarasını, BSR ise son 1 olan bitin numarasını birinci operanda yerleştirir. Örneğin:

```
mov ebx, 0x70
bsf eax, ebx
```

Burada EBX yazmacındaki değerin içerisindeki ilk 1 olan bitin numarası 4'ür (0x70 = ...0111 0000). Dolasyıyla bu 4 değeri komut sonucunda birinci opearand'a (yani EAX'e) yerleştirilecektir. Örneğin:

```
mov ebx, 0x7F000000
bsr eax, ebx
```

Burada EBX içerisindeki son 1 olan bit 30 numaralı bittir. Komut sonucunda EAX yazmacına 30 değeri yerleştirilir.

Bu komutlarda kaynak operand 0 ise hedef operand'taki değer tanımsızdır. Bu durumda ZF bayrağı set edilmektedir. CF, OF, SF, AF, PF bayrakları bu komutlardan sonra tanımsız durumda olur.

4.2.5.7. NOT ve NEG Komutları

Bu komutlar tek operand'lıdır. NOT bir sayının 1'e tümleyenini (1'e tümleme sayı içerisindeki 0'ların 1, 1'lerin 0 yapılmasıdır) NEG ise 2'ye tümleyenini elde eder. Komutların operandları yazmaç ya da bellek olabilir. Örneğin:

```
xor eax, eax       ; eax = 0
not eax               ; eax = 0xFFFFFFFF
neg eax               ; eax = 1
```

Burada önce EAX yazmacı XOR komutuyla 0'a çekilmiştir. Sonra NOT komutuyla EAX'in içerisindeki değer 0xFFFFFFFF durumuna getirilmiştir. Bu değer işaretli olarak -1'dir. NEG değerin negatifini elde ettiğinden NEG sonrasında EAX'te 1 değeri bulunacaktır.

4.2.6. Bayraklarla İlgili Bazı Makine Komutları

Komut kalıplarındaki yazmaç kavramı bayrak yazmacını içermemektedir. Yani biz MOV gibi bir makine komutuyla EFLAGS yazmacına değer atayamayız ya da onun değerini alamayız. Ancak Intel'de bayrak yazmacını stack'e atan ve onu stack'ten alan PUSHF ve POPF isimli iki özel komut bulunmaktadır. Bu komutlar EFLAGS yazmacının düşük anlamlı 16 bitini alıp set etmek amacıyla kullanılabilmektedir. Çünkü EFLAGS yazmacının yüksek anlamlı (yani sonradan eklenen) bitleri işlemcinin modları gibi özel işlemlerle ilgilidir. PUSHF komutu EFLAGS yazmacındaki değeri yüksek anlamlı 2 byte'ı sıfır olacak biçimde stack'e atar. PUSHF ve POPF makine komutları için ayrıca eş anlamlı PUSHFD ve POPFD isimleri de bulundurulmuştur. Aslında biz 32 bit modda 16 bit olacak biçimde bayrak yazmacını stack'e push edip pop edebiliriz. Tabii bunun için komut ekstra öneke sahip olur. Ancak pek çok 32 bit assembly derleyicisi bayrakları 16 bit olarak stack'e atan ve aalan komutlara sahip değildir. (Zaten bu işlemin 32 bit modda bir anlamı olduğu da söylenemez.)

PUSHF ve POPF komutları sayesinde EFLAGS yazmacının düşük anlamlı 2 byte'ındaki bayrakları elde edip değiştirebiliriz. Örneğin:

```
pushf
pop     eax
```

Burada biz EFLAGS yazmacının değerini eax'e atamış olduk. Şimdi eax'in bitleri üzerinde AND, OR gibi komutlarla değişiklik yaptıktan sonra ters işlemi yapabiliriz:

```
push    eax
popf
```

Bayraklarla ilgili ilginç iki makine komutu da LAHF ve SAHF komutlarıdır. LAHF komutu EFLAGS yazmacının düşük anlamlı 8 bitini AH yazmacına atar. SAHF ise tam tersi biçimde AH yazmacındaki değeri EFLAGS yazmacının düşük anlamlı byte'ına yerleştirir. EFLAGS'in düşük anlamlı byte'ında CF, PF, AF, ZF, SF gibi önemli bayraklarının bulunduğunu anımsayınız.

Yukarıdaki bayrak komutlarından başka ayrıca belli bayrakları (fakat hepsini değil) set ve reset eden ayrı makine komutları da vardır. Bunların listesi aşağıda verilmiştir:

CLAC (Auxiliary Carry bayrağını reset eder)
STAC (Auxiliary Carr bayrağını set eder)
CLC (Carry bayrağını reset eder)
STC (Carry bayrağını set eder)
CLD (Direction bayrağını reset eder)
STD (Direction bayrağını set eder)
CLI (Interrupt bayrağını reset eder)
STI (Interrupt bayrağını set eder)
CMC (CF bayrağını tersiyle değiştirir)

5. 80X87 Matematik İşlemcilerinin Kullanımı ve Gerçek Sayılarla İşlemler

Anımsanacağı gibi Intel işlemcilerinde birleşik tasarlanmış bir gerçek sayı birimi yoktur. Intel eskiden gerçek sayı işlemleri için 80X87 matematik işlemcilerini üretmişti. 80486 DX modeliyle birlikte bu matematik işlemci ana tamsayı işlemcisiyle aynı entegre devreye yerleştirilmiştir. Aynı entegre devreye yerleştirilmiş olsalar da tamsayı işlemlerini yapan ana işlemciyle gerçek sayı işlemlerini yapan matematik işlemci birbirine bağlı iki farklı birim gibi çalışmaya devam etmektedir.

Intel'in matemitik işlemci komutlarının başı F harfiyle başlar. (Örneğin FADD, FSIN, FCOS gibi.) Matemetik işlemci birimi stack makinesi (stack machine) gibi çalışmaktadır. (.NET'in arakodu olan CIL ve Java platformunun arakodu olan "Java Byte Code" un da stack makinesi gibi çalışan bir sanal makineye dayalı olarak oluşturulduğunu anımsayınız.)

Matematik işlemcinin içerisinde 8 elemanlık bir içsel stack bulunur. Bu stack'in ana işlemcinin kullandığı RAM'deki stack'le bir ilgisi yoktur. Matematik işlemcinin stack'i bellekte değil matematik işlemcinin kendi içerisindedir. Bu stack sistemi aslında yazmaçlardan oluşturulmuş durumdadır. Bu nedenle matematik işlemci içerisindeki bu stack'e "veri yazmaçları (data registers) da denilmektedir. Intel'in matematik işlemcilerindeki stack elemanları 10 byte uzunluğundadır. Bu 10 byte'lık stack elemanları "IEEE 754 Extended Real Format"ına uygun bir biçimde gerçek sayıları tutar. Yani Intel'in matematik işlemcileri kendi içerisinde gerçek sayıları hep 10 byte duyarlılığında tutarak işleme sokmaktadır. (Örneğin işleme sokulacak operandlar 4 byte (float) olsa bile matematik işlemci bunları 10 byte olarak stack yazmaçlarında saklar ve işlemlere de 10 byte duyarlılıkla sokar.) Intel'in matematik işlemcilerinin yazmaç yapısı aşağıdaki şeekille özetlenebilir:

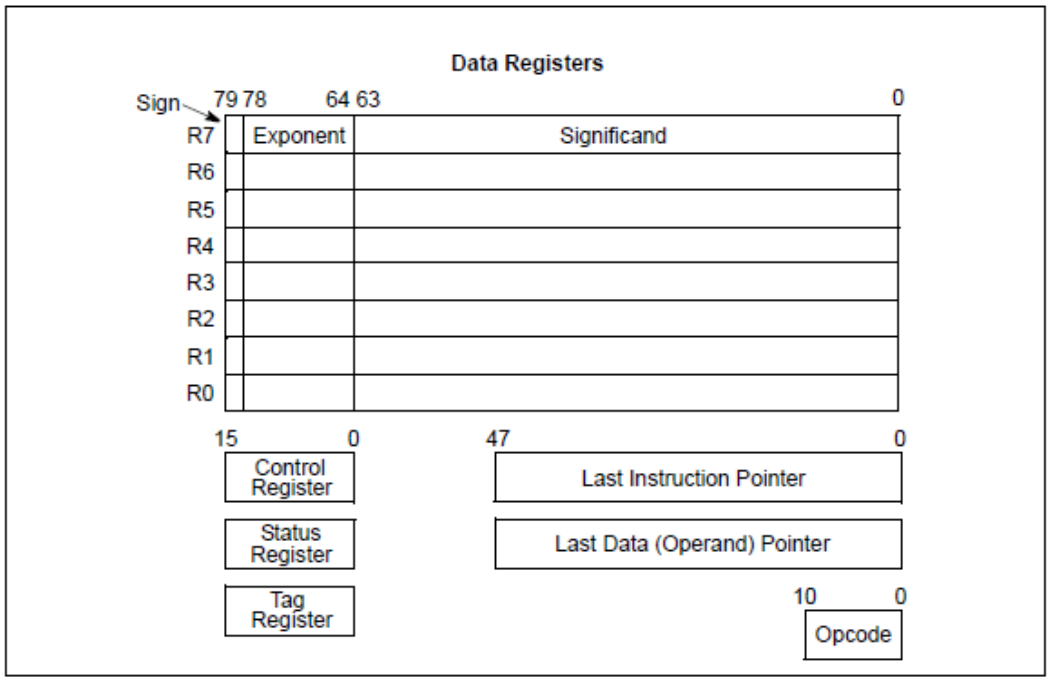
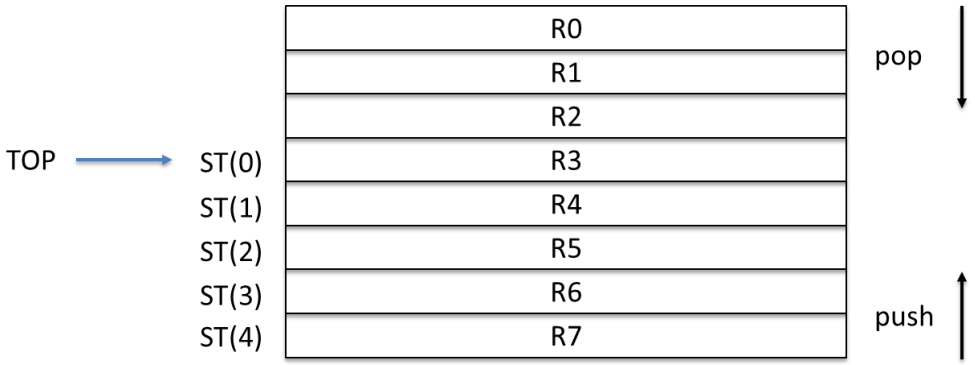


Figure 8-1. x87 FPU Execution Environment

Gördüğünüz gibi matematik işlemcinin içerisinde stack yazmaçlarının yanı sıra bir durum yazmacı (status register) bir de kontrol yazmacı (control register) da bulunmaktadır.

Matematik işlemcideki stack'in tepesi durum yazmacındaki TOP (Top Of the Stack) isimli 3 bit ile gösterilmektedir. Yani durum yazmacının TOP bitleri adeta stack göstericisi gibi işlem görür. Stack'in yönü R7'den R0'a doğrudur. Yani push işlemi sırasında yazmaç değeri bir azaltılır ve o yazmaca değer aktarılır.

Matematik işlemcinin stack'i döngüsel biçimdedir. Yani push işlemi (FLD işlemi) ile R0 yazmacına gelindiğinde yeniden R7'den devam edilmektedir. Stack'in tepesi ST(0) ile tepenin altındaki yazmaçlar da (yani daha önce push edilmiş olanlar) sırasıyla ST(1), ST(2), ... ST(7) biçiminde temsil edilmektedir.

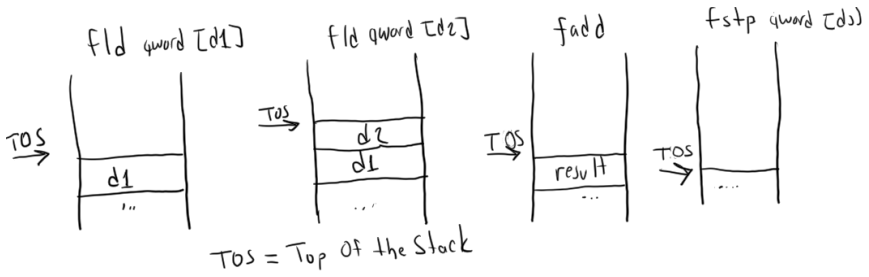


Burada durum yazmacındaki TOP bitleri R3 yazmacını göstermektedir. Yani stack'in tepesi R3'tür. Stack'in tepesi her zaman ST(0) ile temsil edilmektedir.

Anahtar Notlar: Yukarıdaki çizimi biz düşük numaralı stack yazmacı yukarıda olacak biçimde çizdik. Halbuki Intel her zaman düşük adreesleri ve değerleri şekillerde daha aşağıda olacak biçimde çizmektedir. Matematik işlemcinin yazmaçlarını gösteren önceki şekil Intel dokümanlarından alınmıştır. Orada çizimin ters yönde olduğuna dikkat ediniz.

Gerçek sayı işlemleri matematik işlemci tarafından kabaca şöyle yapılmaktadır: İşleme sokulacak değerler programcı tarafından önce matematik işlemcinin stack'ine push edilir. Sonra gerçek sayı komutu uygulanır. Eğer komut operandsızsa her zaman stack'in tepesindeki değerler pop edilerek işleme sokulur ve sonuç yeniden stack'e push edilir. Örneğin iki double değer toplanacak olsun. Bu işlem şöyle yapılabilir:

```
fld     qword [d1]
fld     qword [d2]
fadd
fstpqword[d3]
```



FLD (f load) komutu stack'e eleman push etmek için, FST (f store) komutu ise stack'teki elemanın değerini belleğe yerleştirmek için kullanılmaktadır. FSTP (f store pop) elemanı stack'ten alarak aynı zamanda pop işlemi de yapar. Programcının stack dengisine dikkat etmesi gerekir. Yani biz stack'e ne kadar değer push etmişsek o kadar miktarda pop yapılmış olmasına dikkat etmeeliyiz. Matematik işlemcinin stack'i de "overflow" ve underflow" olabilmektedir.

Anahtar Notlar: Bugün işlemci ve sanal makine gerçekleştirimleri için iki temel model vardır. Bunlardan birine "register machine" diğerine "stack machine" denilmektedir. İşlemcilerin çoğu "register machine" modeline uygundur. Bu modelde komutların operandları yazmaçlarda tutulur. Komutlarda da hangi yazmaçların işleme sokulacağı belirtilir. "Stack machine"de ise komutlarda operandlar belirtilmez. İşleme sokulacak operandlar önce stack'e push edilir. Komutlar zaten stack'in tepesindeki değerleri pop ederek işleme sokarlar. Sonucu da yeniden stack'e atarlar. .NET'in çalıştırma ortamı ve ara kodu, Java'nın çalıştırma ortamı ve arakodu stack makinesi olarak tasarlanmıştır.

Matematik işlemcinin komutlarında (birkaç istisna dışında) ana işlemcinin yazmaçları doğrudan kullanılamaz. Ana işlemcinin yazmaçları yalnızca bellek operandını oluştururken kullanılabilmektedir. Çünkü Intel tarihsel olarak ana işlemciyi ve matematik işlemciyi iki farklı birim olarak tasarlamıştır.

5.1. 80X87 Matematik İşlemcilerinin Temel Matematiksel Komutları

Yukarıda da belirttiğimiz gibi Intel'in tüm matematik işlemci komutları F harfi ile başlamaktadır. Bu komutların her biri tipik olarak birkaç biçimde bulunur:

1) Operandsız Biçim: Bu biçimde komutun hiçbir operandı yoktur. Örneğin:

```
FADD
```

Bu durumda komut matematik işlemci stack'inin tepesindeki iki değeri (ST(0) ve ST(1)) pop ederek alır onları işleme sokar ve sonucu da yeniden stack'e push eder.

2) Tek Bellek Operandlı Biçim: Bu biçimde matematik işlemci stack'inin tepesindeki değer (ST(0)) pop edilerek operand olan değerle işleme sokulur. Sonuç yeniden stack'e push edilir. Örneğin:

```
FADD    qword [EBX]
```

İşlem 4 byte, 8 byte ya da 10 byte duyarlılığında yapılabilmektedir.

3) İki Stack Operandlı Biçim: Bu biçimde operand'lardan biri stack'in tepesi (yani ST(0)) diğeri ise herhangi bir stack operandı (ST(i) biçiminde gösterebiliriz) olabilmektedir. Stak'in tepesindeki değer ile belirtilen stack elemanındaki değer işleme sokulup sonuç sol taraftaki stack elemanına yeniden yerleştirilir. Herhangi bir pop işlemi yapılmaz. Örneğin:

```
FADD    ST3, ST0
```

Burada ST(3) ile ST(0) işleme sokularak sonuç ST(3)'te bırakılmaktadır.

5.1.1. FLD, FST ve FSTP Komutları

FLD komutu bellekteki 4 byte (float), 8 (double) ve 10 byte (long double)'lık gerçek sayıları matematik işlemcinin stack'ine push etmek için kullanılmaktadır. FLD komutunun operand'ı bir bellek adresi ya da stack elemanıdır. Örneğin:

```
fld qword [d1]  ; d1 adresinden başlayan double değeri push et
fld dword [d2]  ; d2 adresinden başlayan float değeri push et
fld st(5)       ; stack'in tepesinden itibaren 5'inci elemanı yeniden stack'e push et
```

FST komutu matematik işlemcinin stack'indeki değeri operandına aktarmak için kullanılır. Komut yine bellek ya da stack elemanını operand olarak almaktadır. Örneğin:

```
FST qword [result]
```

Bu komutla matematik işlemcinin stack'inin tepesindeki (ST(0)'daki) değer result adresinden itibaren belleğe aktarılacaktır. Örneğin:

```
FST ST(5)
```

Bu komutla da stack'in tepesindeki eleman stack'in tepesinden itibaren 5'inci elemana aktarılır. (Yani artık stack'in 5'inci elemanında stack'in tepesindeki elemanın değeri bulunur.) FST komutları pop işlemi yapmaz. Yani stack'in tepesinden alınan değer yine stack'in tepesinde kalmaya devam eder. İşte FST komutunun FSTP biçimi tamamen FST gibidir. Ancak stack'in tepesindeki değeri aynı zamanda oradan alarak pop eder (yani atar).

5.1.2. FILD, FIST ve FISTP Komutları

Bu komutlar FLD, FST ve FSTP komutlarına oldukça benzemektedir. Ancak operand olarak tamsayı alırlar. Yani örneğin biz bir tamsayıyı gerçek sayı formatına dönüştürerek matematik işlemcinin stack'ine push etmek istiyorsak bu komutları kullanmalıyız. Örneğin:

```
FLD     dword  [data]
```

komutu ile,

```
FILD    dword [data]
```

komutu arasında ne fark vardır? FLD komutu operandı olan adresteki değerin zaten gerçek sayı formatıyla orada bulunduğunu varsayar. Halbuki FILD komutu operandı olan adresteki değerin tamsayı olduğunu kabul ederek onu gerçek sayı formatına dönüştürüp matematik işlemcinin stack'ine atmaktadır. Örneğin C/C++ gibi dillerde int bir değerin double gibi bir nesneye atanması işlemi derleyiciler tarafından şöyle yapılmaktadır:

```
FILD    dword [int_object]
FSTP    qword [double_object]
```

FIST ve FISTP komutları tam ters bir işlemi yapmaktadır. Yani bu komutlar matematik işlemcinin stack'inin içerisinde bulunan gerçek sayı formatındaki değeri tamsayıya dönüştürerek belleğe yerleştirirler. Örneğin C/C++ gibi dillerde biz bir double değeri int bir nesneye atamak istediğimizde derleyici bunun aşağıdaki gibi bir kod üretecektir:

```
FLD     qword [double_object]
```

```
FISTP   dword [int_object]
```

Bu komutlar 64 bit tamsayı üzerine de işlem yapabilmektedir. Örneğin:

```
fild    qword [long_long_object]
fistp   qword [long_long_object]
```

5.1.3. FLD1, FLDZ, FLDPI, FLDL2T, FLDL2E, FLDLG2, FLDLN2 Komutları

Bazı özel değerleri matemetik işlemcinin stack'ine push etmek için özel FLD komutları bulundurulmuştur. Bu komutların operandları yoktur. FLD1 komutu 1 sayısını, FLDZ 0 komutu 0 sayısını push eder. FLDPI pi sayısını push etmektedir. FLD2T komutu $\log_2 10$ değerini, FLDL2E komutu $\log_2 e$ değerini, FLDLG2 komutu $\log_{10} 2$ değerini, FLDN2 komutu da $\log_e 2$ değerini push etmektedir.

5.1.4. FADD, FSUB, FMUL, FIMUL, FDIV ve FIDIV Komutları

Bu komutlar gerçek sayılar üzerinde klasik 4 işlemi yapmaktadır. Komutların genel biçimleri aynıdır. Bu komutların birkaç biçimi vardır:

1) Operandsız biçim: Bu biçimde her zaman stack'in tepesindeki iki değer pop edilip işleme sokulur ve sonuç yine işlemcinin stack'ine push edilir. Örneğin:

```
FLD     qword [d1]
FLD     qword [d2]
FMUL
FSTP    qword [d3]
```

2) Bellek operandlı biçim: Bu biçimde stack'in tepesindeki değer pop edilerek doğrudan bellekteki değerle işleme sokulur, sonuç yine stack'e push edilir. Örneğin:

```
fld     qword [d1]
fmul        qword [d2]
fstp        qword [d3]
```

Bu işlem yukarıdakiyle eşdeğerdir.

3) Stack elemanını operand olarak alan biçim: Burada komutun operandı bir stack elemanıdır. Komut stack'in tepesindeki eleman ile operand olarak verilen elemanı işleme sokar. Sonuç operand olan elemanda bulunur. Örneğin:

```
FMULST1
```

Burada ST(1) ile ST(0) yani stack'in tepesindeki değer işleme sokulmuştur. Sonuç ST(0)'a atanmıştır. Bu işlem uzun olarak şöyle de belirtilebilir:

```
FMULST0, ST1
```

Operandlar yer de değiştirebilmektedir. Örneğin:

```
FMULST1, ST0
```

Burada artık sonuç ST(1)'e atanmaktadır. Bu biçimdeki kullanımda operandların biri mutlaka stack'in tepesi yani ST(0) olmak zorundadır. Komutların stack operandlı olanlarına genel olarak pek gereksinim duyulmamaktadır. Ancak bazı durumlarda bu komutlarla daha optimize kod üretilmesi mümkün olabilmektedir.

FIMUL stack'in tepesindeki değeri pop edip operandı olan tamsayı değerle çarpar ve sonucu yeniden stack'e

push eder. FIDIV de benzer işlemi yapmaktadır.

5.1.5. FSIN, FCOS, FSINCOS ve FPTAN Komutları

FSIN stack'in tepesindeki değeri pop edip onun sinüsünü hesaplayarak yeni değeri push eder. Örneğin sinüs 30 derece şöyle hesaplanabilir:

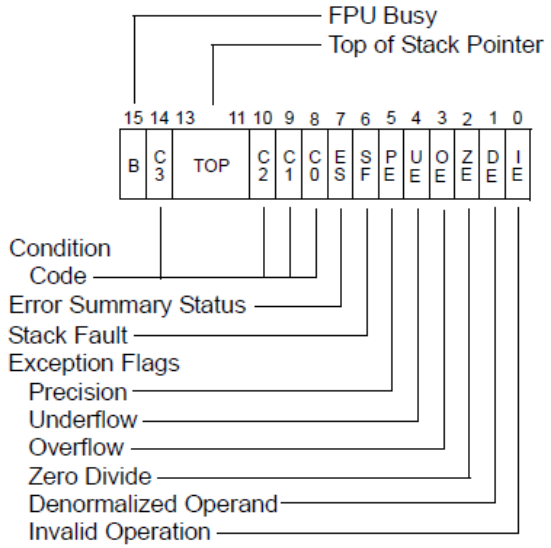
```
fldpi
fdiv        qword [d1]
fsin
fstp        qword [d2]
```

FCOS benzer biçimde cosinüs işlemi yapmaktadır. FSINCOS stack'in tepesindeki değeri pop eder sonra onun sinüsünü ve kosinüsünü alır. Önce kosinüsü sonra da sinüsü stack'e push eder. FPTAN stack'in tepesinde değeri pop eder. Önce onun tanjantını sonra da 1 sayısını stack'e push eder.

5.2. Matematik İşlemcide Karşılaştırma İşlemleri

İki gerçek sayıyı nasıl karşılaştırabiliriz? Anımsanacağı gibi ana tamsayı işlemcilerinde karşılaştırma için önce SUB ya da CMP komutu uygulanıyor sonra da bayraklara bakılarak koşullu jump yapılıyordu. Matematik işlemcide de karşılaştırma komutları vardır. Ancak bu karşılaştırma komutları ana tamsayı işlemcisinin bayraklarını etkilememektedir. Pekiyi bu durumda gerçek sayı karşılaştırmaları nasıl yapılmaktadır?

Matematik işlemcide karşılaştırma yapan komutlar matematik işlemcinin içerisindeki durum yazmacının (status register) C0, C2 ve C3 bitlerini etkilemektedir. Matematik işlemci içerisindeki durum yazmacının bitleri aşağıdaki gibidir:



Yukarıdaki şekilden de görüldüğü gibi matematik işlemcinin durum yazmacı 16 bittir. Durum yazmacının C0, C2 ve C3 bitlerine "durum kodu (conditin code)" da denilmektedir. İşte karşılaştırma komutları C1 dışındaki durum kodlarını etkilemektedir.

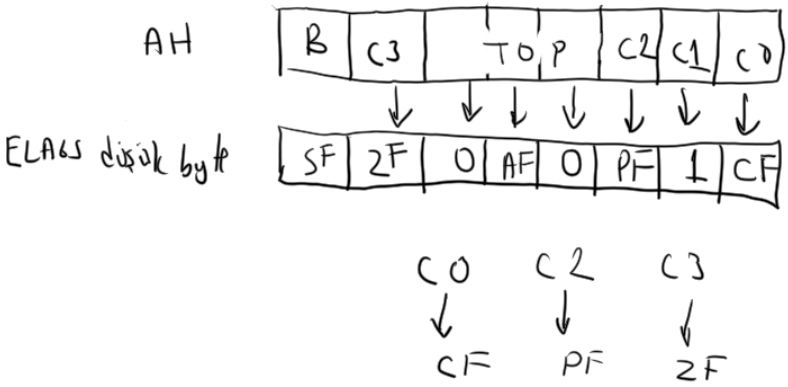
Karşılaştırma için tipik olarak FCOM, FCOMP v e FCOMPP komutları kullanılmaktadır. FCOMPP komutu stack'e karşılaştırma amacıyla push edilmiş iki değeri de pop etmektedir. FCOMP ise yalnızca stack'teki tek değeri (stack'in tepesindeki değeri) pop etmektedir. FCOM ise hiçbir değeri pop etmez. Bu komutların parametresiz biçimleri stack'in tepesindeki iki değeri karşılaştırmaktadır (yani ST(0) ile ST(1)'i). Bu komutların ayrıca bellek operandı alan biçimleri de vardır. Komutların bu biçimleri ST(0) ile doğrudan o bellek operandını karşılaştırmaktadır. Matematik işlemcinin karşılaştırma komutları C0, C2 ve C3 bitlerini

şöyle etkilemektedir (Aşağıdaki şekilde ST(0) karşılaştırmanın solundaki operandı, diğeri ise ST(1) ya da bellek operandını belirtmektedir):

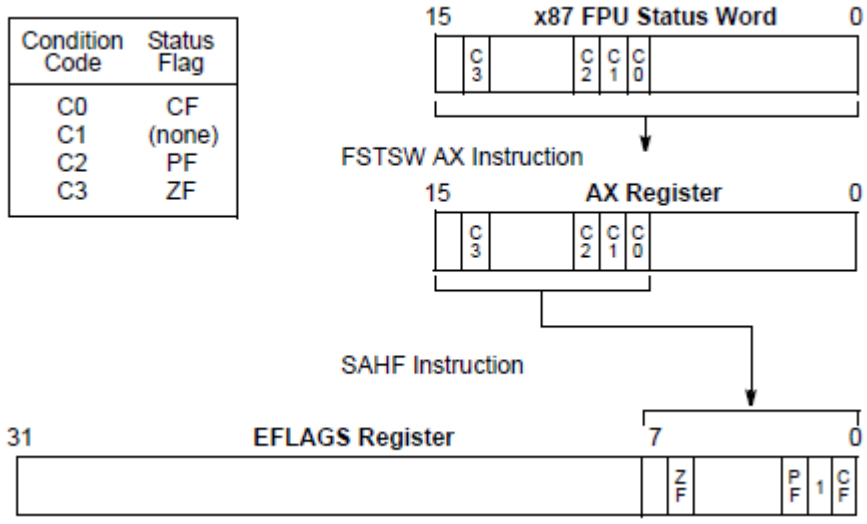
Condition	C3	C2	C0
ST(0) > Source Operand	0	0	0
ST(0) < Source Operand	0	0	1
ST(0) = Source Operand	1	0	0
Unordered	1	1	1

Tabii karşılatırma komutlarının C0, C2 ve C3 bitlerini set etmesi bizim yeterli değildir. İşte matematik işlemcide durum yazmacını ana işlemcinin AX yazmacına taşıyan FSTSW komutu vardır. Bu komut sonucunda yukarıdaki durum yazmacının 16 biti AX yazmacına taşınır. Bu işlemden sonra da SAHF komutu uygulanırsa bu komut da AH yazmacını (yani artık durum yazmacının yüksek anlamlı byte'ını) EFLAGS yazmacının düşük anlamlı byte'ına yerleştirecektir. Böylece iki komutun peş peşe verilmesiyle aşağıdaki gibi ilginç bir durum oluşur:

```
fstw   ax
sahf
```



Condition Code	Status Flag
C0	CF
C1	(none)
C2	PF
C3	ZF



İşaretsiz jump komutlarının CF ve ZF baprağına baktığını anımsayınız:

JA/JNBE	İşaretsiz olarak birinci operand ikinci operand'tan büyüktür	CF = 0 ve ZF = 0
JB/JNAE/JC	İşaretsiz olarak birinci operand ikinci operand'tan daha küçük	CF = 1

JAE/JNB/JNC	İşaretsiz olarak birinci operand ikinci operand'tan büyük ya da eşit	CF = 0
JBE/JNA	İşaretsiz olarak birinci operand ikinci operand'tan küçük ya da eşit	CF = 1 veya ZF = 1
JE/JZ	İşaretli ya da işaretsiz olarak iki operand birbirine eşitse	ZF = 1
JNE/JNZ	İşaretli ya da işaretsiz olarak iki operand birbirine eşitse	ZF = 0

O halde biz gerçeksayıları karşılaştırma işlemine sokup sırasıyla fstw ax ve sahf komutlarını uyguladıktan sonra yukarıdaki işaretsiz jump komutları ile dallanma yapabiliriz. Örneğin:

```
   fld       qword [a]
   fld       qword [b]
   fcompp

   fstsw     ax
   sahf
   ja        @1

   ; b <= a

   jmp       @2
@1:
   ; b > a
@2:
```

fcom, fcomp ve fcompp komutlarının stack'in tepesindeki değerle diğer operandı karşılaştırdığına dikkat ediniz. Yukarıdaki örnekte b stack'in tepesinde (yani ST(0)'da) a ise onun aşağısındadır (yani (ST(1)'de). Dolayısıyla fcompp komutu b ile a'yı b solda a sağda olacak biçimde karşılaştırmaktadır.

Karşılaştırma sırasında önce fstsw ax sonra sahf komutlarını kullanmak biraz zahmetlidir. Intel bu yüzden 80386 ile birlikte bu işlemi otomatik yapan fcomi, fcomip komutlarını tasarlamıştır. Bu komutlar karşılaştırmayı yapıp durum yazmacının C0, C2 ce C3 bitlerini sırasıyla EFLAGS yazmacının CF, PF ve ZF bayraklarına yerleştirmektedir. Yani yukarıdaki karşılaştırma işleminin işlevsel olarak eşdeğeri şöyle de yazılabilir:

```
   fld       qword [a]
   fld       qword [b]
   fcomip
   fstp      st0

   ja        @1

   ; b <= a

   jmp       @2
@1:
   ; b > a
@2:
```

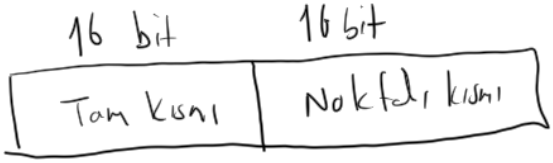
fcomip komutunun fcomipp biçiminde bir versiyonun olmadığına dikkat ediniz. Bu nedenle yukarıdaki örnekte fstp st0 komutu ile boş bir pop işlemi yapılmıştır. Karşılaştırma komutlarının "unordered" versiyonları da vardır. Bunlar fucom, fucomp, fucompp, fucomi, fucomip komutlarıdır. "Unordered" komutlarla "ordered" komutlar arasındaki tek fark "underored" komutlarda operandlardan biri ya da her ikisi NAN ise "unordered" komutların "invalid aritmetik operand exception" oluşturmasıdır. Kesmeler konusu ilerideki bölümlerde ele alınmaktadır.

5.3. IEEE 754 Gerçek Sayı Formatlarına Özet Bir Bakış

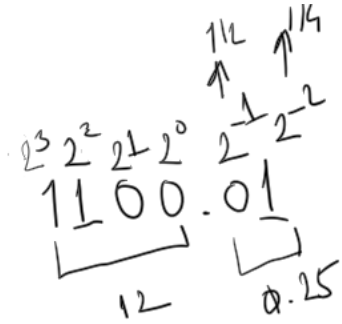
Bilindiği gibi Intel'in matematik işlemcileri (ve neredeyse floating point işlemi yapan işlemcilerin hepsi) IEEE 754 numaralı gerçeksayı formatını kullanmaktadır. Bu nedenle C, C++ ve diğer dillerin hemen hepsindeki gerçek sayı türleri hep bu formata sahiptir.

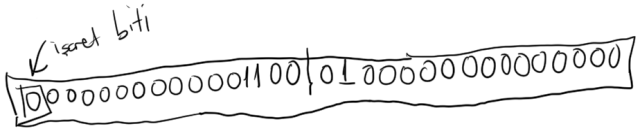
Gerçek sayılar yalnızca 1 ve 0'larla nasıl ifade edilmektedir? İkilik sistemde +, – ve nokta gibi semboller yoktur. İşte noktalı sayıların ikilik sistemde ifade edilmesi için iki temel format ailesi kullanılmaktadır: "Sabit noktalı (fixed point) formatlar ve "kayan noktalı (floating point)" formatlar.

Sabit noktalı formatlar bugün mikrodenetleyicilerde DSP'lerde hala kullanılmaktadır. Bu formatlar verimli olmamasına karşın noktalı sayı işlemlerini tamsayı işlemleriyle kolayca yapmaya olanak sağlar. Sabit noktalı formatlarda noktanın yeri bellidir. Böylece noktanın yeri sayı içerisinde ayrıca saklanmaz. Örneğin 32 bit sabit noktalı formatta noktanın tam ortada bulunduğunu varsayalım:

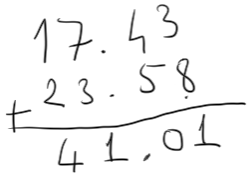


Örneğin bu formatta 12.23 sayısını tutmak isteeyelim. Önce bu sayı ikilik sisteme dönüştürülür.





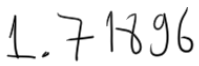
Noktanın sağ tarafının 2'nin negatif kuvvetleriyle çarpıldığına dikkat ediniz. Sabit noktalı formatlarda aritmetik işlemler sanki tamsayı işlemleri gibi yapılabilmektedir. Örneğin:



Bu işlemin sanki nokta yokmuş gibi tamsayılarla yapılıp noktanın daha sonra da yerleştirilebileceğine dikkat ediniz.

Sabit noktalı formatlar yukarıda da belirtildiği gibi düşük güçlü gerçek sayı birimi olmayan ya da zayıf olan işlemciler için çok uygundur. Ancak sabit noktalı formatlar verimsizdir. Nedeni oldukça basittir. Bu formatlarda noktanın yeri sabit olduğu için onun sağı ya da solu az basamaktan oluşuyorsa orası boşuna yer

kaplıyor durumdadır. Örneğin:



gibi bir sayıda noktanın sağı belki de 16 bite sığmayacaktır. Fakat solunda boş alan vardır. Bundan faydalanılamamktadır. İşte bu nedenle kayan noktalı (floating point) formatlar tasarlanmıştır.

Kayan noktalı formatlarda nokta olmadan tüm sayıya "mantis (mantissa)" ya da "fraction" denilmektedir. Noktanın yeri sayının içerisinde bir yerde tutulmaktadır. İşte IEEE'nin 754 numaralı formatı da bu biçimdedir. Bu formatta mantisin ve üstel kısmın sayının hangi bitlerinde tutulduğu açıkça dokümante edilmiştir.

Anlatımı devam ettirmeden önce test amaçlı kullanılmak üzere float ve double sayıların bitlerini ekrana yazdıran bir C programı yazalım:

```
#include <stdio.h>

void printfp(void *val, int size)
{
    unsigned char *puc = (unsigned char *)val + size - 1;
    int i;

    while (size-- > 0) {
        for (i = 7; i >= 0; --i)
            putchar((*puc >> i & 1) + '0');
        --puc;
    }
    putchar('\n');
}


int main(void)
{
    float f = 12.3f;
    double d = 12.3;

    printfp(&f, sizeof(f));
    printfp(&d, sizeof(d));

    return 0;
}
```

IEEE 754 numaralı formatlar her zaman işaretlidir. Sayının en yüksek biti işaret bitidir. Bu bit 1 ise sayı negatif 0 ise pozitiftir. IEEE 4 byte'lık (short real) ve 8 byte'lık (long real) gerçek sayı formatları şöyledir:

Floating Point Components

	Sign	Exponent	Fraction
Single Precision	1 [31]	8 [30–23]	23 [22–00]
Double Precision	1 [63]	11 [62–52]	52 [51–00]

```
Single: SEEEEEEE EMMMMMMM MMMMMMMM MMMMMMMM
Double: SEEEEEEE EEEEMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM
```

Tablodaki "Fraction" mantis anlamındadır. Onun aşağısındaki şekilde de şekilde de S işaret bitini, E üstel kısmı oluşturan bitleri M de mantis bitleerini göstermektedir.

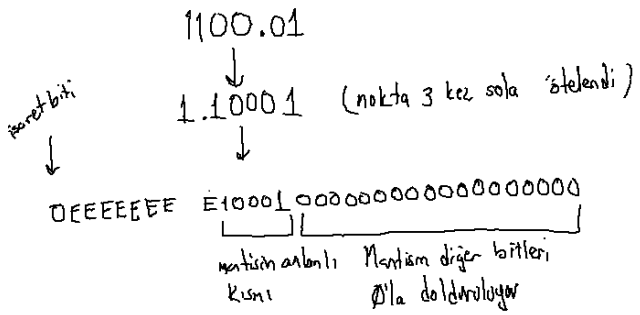
Noktalı sayının mantis kısmını elde etmek için sayı noktalı biçimde (noktanın sağı 2'nin negatif kuvvetleri olacak biçimde) ikilik sisteme dönüştürülür. Sonra nokta kaldırılır. Örneğin 12.25 float sayısını ikilik sistemde yazmak isteyelim. Bunun için önce sayıyı ikilik sistemde noktanın sağ tarafı ikinin negatif kuvvetleri ile oluşturulur:

1100.01

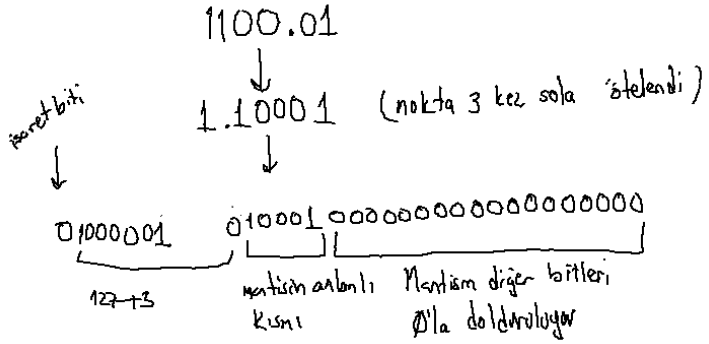
Bu durumda sayının mantisi 110001 biçimindedir. Bu mantisin yukarıdaki formatta MM...M kısmına oturtulması şöyle yapılmaktadır: IEEE formatında sayının her zaman 1.XXXX biçiminde mantise sahip olduğu düşünülür. Bu nedenle formattaki mantis kısmındaki ilk 1 biti boşuna tutulmaz. Bu durumda önce sayı 1.XXXX biçimine getirilir. Sayıyı bu biçime getirmek için kaç kere sola ya da sağa noktanın hareket ettirildiği de not alınır. Örneğin 12.25 sayısı yukarıda da belirtildiği gibi 1100.01 biçimindedir. Bu sayıyı 1.XXX biçimine getirmek için noktayı 3 kez sola ötelememiz gerekir:

1.10001

İşte artık 1.XXXX biçimine dönüştürülmüş olan noktalı sayının noktadan sonraki kısmı formattaki mantisi oluşturmaktadır. Eğer bu mantis formatta ayrılan alandan daha küçükse geri kalan kısım sağdan sıfır ile doldurulur (ikilik sistemdeki noktalı bir sayının soluna ya da sağına koyacağımız sıfırların sayının değerini değiştirmeyeceğine dikkat ediniz):



Pekiyi üstel kısım (yani E ile belirtilen bitler) nasıl oluşturulmaktadır? İşte float ve double formatları için "bias" değerleri vardır. "Bias" derleri float için 127 double için 1023'tür. Eğer sayı noktadan sonra n defa sola ötelenmişse +n değeri n defa sağa ötelnmişse –n değeri "bias" değerle toplanır ve formattaki üstel kısma (E'lerle belirtilen kısma) yerleştirilir. Yukarıdaki sayıyı biz 3 kere sola öteledik float için "bias" değer 127 olduğuna göre buraya yazacağımız değer 127 + 3 = 130 olacaktır. Böylece örnekteki 12.25 sayısının float formatındaki bitsel karşılığı şöyle olur:



Şimdi de -0.125 sayısını double formatına dönüştürelim. Önce sayıyı noktanın sağında ikinin negatif kuvvetleri olacak biçimde ikilik sisteme dönüştürelim:

```
-0.0010000...
```

Bu sayıyı daha sonra 1.XXX biçimine dönüştürelim:

```
-1.0000...
```

Böylece mantisin anlamlı kısmı sıfır olacaktır. Yani tüm mantis sıfırlardan oluşacaktır. Double sayılar için "bias" değeri 1023'tür. Dolayısıyla sayının üstel kısmı 1023 – 3 = 1020'dir. O halde sayının bitsel karşılığı şöyle olacaktır:

```
1 01111111100 0000000000000000000000000000000000000000000000000000
```

Biz yine özet olarak 10'luk sistemdeki ondalık bir sayının nasıl float ve double formatlarına dönüştürüleceğini madde madde açıklayalım:

1) Sayı önce noktanın sağ tarafı 2'nin negatif kuvvetleri olacak biçimde ikilik sisteme dönüştürülür.

2) Sonra sayı 1.XXXX biçimine getirilir. Noktanın ne kadar ötelendiği not alınır (sola ötelemeler pozitif, sağa ötelemler negatif olacak biçimde)

3) Sonra öteleme miktarı "bias" değerle (yani 127 ya da 1023 değeriyle) toplanır.

4) İşaret biti olarak sayı pozitifse 0, negatifse 1 kullanılır.

5) Sonra elde edilen ikilik sistemdeki değerler formattaki yerlerine yerleştirilir. Sayı yerleştirilirken mantisin en başındaki 1 atılır. Mantisin sağı 0'larla doldurulur.

IEEE'nin 10 byte'lık gerçek sayı formatında işaret biti için 1 bit, üstel kısım için 15 bit ve mantis için 64 bit ayrılmıştır. "Bias" değeri ise 16383'tür. Bu formatta mantisin en yüksek anlamlı biti her zaman 1 olmalıdır.

6. NASM Derleyicisinde Sembolik Makine Dili Direktifleri

Bu bölümde NASM derleyicisinde kullanılan sembolik makine dili direktifleri ele alınacaktır. Konunun daha iyi anlaşılması için önce PE ve ELF formatlarındaki "bölüm (section)" kavramını açıklayacağız.

6.1. PE (Portable Executable) ve ELF (Executable and Linkable Format) Çalıştırılabilen Dosyalarındaki Bölüm (Section) Kavramı

Bugün için en çok kullanılan çalıştırılabilir (executable) dosya formatları PE ve ELF'tir. PE formatını Microsoft 32 bit Windows sistemleri ilk çıktığında tasarlamıştı. Microsoft daha önce 16 bit Windows 3.X sistemlerinde NE (New Executable) denilen bir format kullanıyordu. Microsoft'un DOS'ta kullandığı çalıştırılabilen dosya formatı da MZ (Mark Zibikoovsky'nin isminden geliyor) formatıydı. UNIX/Linux dünyasında da pek çok çalıştırılabilir dosya formatı denemiştir. "a.out" isimli format uzun süre pek çok UNIX türevi sistemde kullanılmıştır. Linux da başlangıçta bu formatı kullanıyordu. Artık UNIX türevi sistemlerin büyük kısmı ELF formatını birincil çalıştırılabilen format olarak desteklemektedir. Ayrıca bir işletim sistemi birden fazla çalıştırılabilen dosya formatını destekliyor olabilir. Örneğin Linux sistemleri ELF formatının yanı sıra hala klasik "a.out" formatını da desteklemektedir. Windows'un pek çok versiyonu NE ve MZ formatlarını da destekliyordu.

PE ve ELF formatlarının 32 bitlik ve 64 bitlik birbirine çok benzeyen biçimleri de vardır. Böylece bazen bu formatlar PE32, PE64, ELF32, ELF64 isimleriyle de belirtilmektedir.

PE ve ELF formatları genel tasarım olarak aslında birbirlerine benzemektedir. Her iki formatta da önemli bilgilerin dosyanın neresinde bulunduğunu gösteren bir başlık (header) kısmı vardır. Her iki format da bölümlerden (sections) oluşmaktadır. Bu formatlara sahip bir program çalıştırılmak istendiğinde işletim sistemi çalıştırılabilen dosyayı açar formattaki bölümleri inceler ve bölümleri belleğe (RAM'e) yükler.

İşletim sisteminin çalıştırılabilen dosyayı okuyarak çalıştırmak üzere belleğe yükleyen kısmına kavramsal olarak "yükleyici (loader)" denilmektedir.

Bölümler aynı özelliklere sahip ardışıl sayfalardan (page) oluşmaktadır. Bölümlerin birer isimleri vardır. ELF ve PE formatında geleneksel olarak bölümler başında "." olacak biçimde isimlendirilmektedir. (Örneğin, ".text", ".data" gibi). Tabii aslında böyle bir zorunluluk yoktur. İşletim sistemi bölümler için bellekte yer ayırıp onları çalıştırılabilen dosyadan okuyarak belleğe (RAM'e) yüklemektedir.

6.1.1. PE ve ELF Formatlarındaki Çok Karşılaşılan Bölümler

Bölümlerin işlevlerini açıklamadan önce açıklamalarda kullanmak için aşağıdaki gibi bir C programı yazalım:

```
#include <stdio.h>

int g_a = 10;
int g_b = 20;

int g_c;
double g_d[100];

char *g_name = "CSD";


void foo(void)
{
    static int count = 1;
    /* ... */
}

void bar(void)
{
    /* .. */
}

void tar(void)
{

}

int main(void)
{
    /* ... */

    return 0;
}
```

PE ve ELF formatlarında en çok karşılaşılan bölümler şunlardır:

.text Bölümü: Bir programın bütün makine kodları (yani fonksiyon kodları) bu bölümde bulunur. Yani yukarıdaki C programında programdaki main, foo ve bar fonksiyonlarının kodları .text bölümüne yerleştirilmektedir.

.data Bölümü: Bu bölümde ilkdeğer verilmiş global değişkenler ve static yerel değişkenler tutulmaktadır. Yani örneğin yukarıdaki C programında g_a, g_b, g_name ve count değişkenleri derleyici tarafından tipik olarak ".data" bölümünde tutulacaktır. Derleyiciler ilkdeğer verilmiş global değişkenleri ilkdeğerleriyle birlikte çalıştırılabilen dosyanın ".data" bölümüne yerleştirirler. İşletim sisteminin yükleyicisi de onları bu bölümden alıp blok olarak fiziksel belleğe yüklemektedir. Bu durumda biz ".data" bölümündeki değişkenlerin çalıştırılabilen dosyada yer kaplayacağını söyleyebiliriz. Ancak bazı çalıştırılabilen dosya formatları bölümler içerisinde hangi ilkdeğerden ne miktarda olduğunu tutma yeteneğine sahiptir (örneğin Windows'un PE formatı böyledir). Böylece aşağıdaki gibi global bir dizi bu sistemlerdeki çalıştırılabilen

dosyalarda çok fazla yer kaplamayabilir:

```
int g_x[1000000] = {1, 2, 3};
```

Ancak ELF gibi bazı formatların bu yeteneği yoktur. Dolayısıyla bu formatlarda yukarıdaki dizinin hepsi ".data" bölümünde ilkdeğerleriyle bulunacak, dolayısıyla bu da çalıştırılabilen dosyanın uzunluğunu büyütecektir.

.bss Bölümü: Bu bölümde ilkdeğer verilmemiş global değişkenler ve static yerel değişkenler tutulmaktadır. Bunlara ilkdeğer verilmediği için bunların çalıştırılabilen dosyalarda boşuna yer kaplamasına gerek de yoktur. PE ve ELF formatlarında bu bölümün yalnızca uzunluğu çalıştırılabilen dosya içerisinde tutulur. İşletim sisteminin yükleyicisi bu uzunluğa bakarak ".bss" bölümünü bellekte (RAM'de) tahsis eder ve orayı sıfırlar.

.rdata ve .rodata Bölümleri: PE formatındaki ".rdata", ELF formatındaki ".rodata" bölümleri global read-only verileri tutmak için düşünülmüştür. Örneğin string ifadeleri genellikle bu sistemlerdeki derleyiciler tarafından bu bölümlerde saklanmaktadır. Windows ve Linux'un yükleyicileri bu bölümlerdeki bilgileri "read-only" sayfalara yüklerler. Dolayısıyla programın çalışma zamanı sırasında buradaki değerler değiştirilmek istenirse "exception (page fault)" oluşur.

PE ve ELF formatında başka bölümler de vardır. Bu bölümler gerek görüldüğünde başka konuların içerisinde ele alınacaktır.

6.2. Microsoft'un DUMPBIN Programı

"Dumpbin" Microsoft'un C/C++ derleyici paketinde bulunan bir utility programdır. Bu program "object module (.obj)" dosyalarını, dinamik kütüphane dosyalarını (dll), statik kütüphane dosyalarını ve çalıştırılabilen (executable) dosyaları incelemek için kullanılmaktadır. Visual Studio IDE'si kurulduğunda "dumpbin" de kurulmuş olmaktadır.

"dumpbin" hiç seçeneksiz çalıştırıldığında yalnızca dosyadaki bölümlerin isimlerini ve uzunluklarını gösterir. Örneğin:

```
D:\Dropbox\Kurslar\80X86-ARM-Assembly\Src\C\Sample\Debug>
D:\Dropbox\Kurslar\80X86-ARM-Assembly\Src\C\Sample\Debug>dumpbin sample.exe
Microsoft (R) COFF/PE Dumper Version 14.00.23506.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file sample.exe

File Type: EXECUTABLE IMAGE

  Summary

        1000 .00cfg
        1000 .data
        1000 .gfids
        1000 .idata
        2000 .rdata
        1000 .reloc
        1000 .rsrc
        5000 .text
       10000 .textbss
```

"Dumpbin" /HEADERS seçenceği ile çalıştırılırsa PE dosyasının başlık kısımlarını görüntüler. "/SECTION:<isim>" seçeneği ile dumpbin istediğimiz bir bölümü de bize gösterebilmektedir. "/DISASM" seçeneği ".text" bölümünü "assembly" sentaksıyla bize gösterir. Diğer seçenekler için MSDN yardım sistemine

başvurabilirsiniz.

“Dumpbin” programının başkaları tarafından yazılmış “wumpbin” biçiminde bir GUI versiyonu da vardır. Aslında “wumpbin” programı “dumpbin” programını çalıştırıp sonucu GUI penceresinde göstermektedir.

6.3. readelf ve objdump Programları

“readelf” ve “objdump” programları “dumpbin” programının UNIX/Linux’taki karşılığı gibi düşünülebilir. “readelf” ELF object modüllerini, executable dosyalarını ve kütüphane dosyalarını incelemekte kullanılır. “objdump” daha genel amaçlı ve daha kapsamlı bir utility’dir.

“readelf” programında “–h” ELF formatının ana başlığını “–e” bütün başlaıklarını görüntüler. “–S” ise bölümleri görüntülemektedir. Örneğin:

```
csd@csd-VirtualBox ~/Study/Assembly $ readelf -h helloworld
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048080
  Start of program headers:          52 (bytes into file)
  Start of section headers:          220 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 3
csd@csd-VirtualBox ~/Study/Assembly $
```

“readelf” ile yapılan pek çok işlem “objdump” ile de yapılabilmektedir. “objdump” programının başka yetenekleri de vardır. “objdump”ta “–f” dosya başlıklarını, “–h” dosyadaki bölümleri görüntülemektedir. “–D” ise dosyanın kod bölümünü bize “assembly” kodu olarak göstermektedir. “–M” seçeneği ile gösterim formatını (Intel ya da AT&T) belirleyebiliriz. Örneğin:

```
csd@csd-VirtualBox ~/Study/Assembly $ objdump -D -M intel helloworld

helloworld:     file format elf32-i386


Disassembly of section .text:

08048080 <_start>:
 8048080:       b8 04 00 00 00          mov    eax,0x4
 8048085:       bb 01 00 00 00          mov    ebx,0x1
 804808a:       b9 a4 90 04 08          mov    ecx,0x80490a4
 804808f:       ba 0e 00 00 00          mov    edx,0xe
 8048094:       cd 80                   int    0x80
 8048096:       b8 01 00 00 00          mov    eax,0x1
 804809b:       bb 00 00 00 00          mov    ebx,0x0
 80480a0:       cd 80                   int    0x80

Disassembly of section .data:

080490a4 <msg>:
 80490a4:       4d                      dec    ebp
 80490a5:       65                      gs
 80490a6:       72 68                   jb     8049110 <_end+0x5c>
 80490a8:       61                      popa
 80490a9:       62 61 20                bound  esp,QWORD PTR [ecx+0x20]
 80490ac:       44                      inc    esp
 80490ad:       75 6e                   jne    804911d <_end+0x69>
 80490af:       79 61                   jns    8049112 <_end+0x5e>
 80490b1:       0a                      .byte 0xa
```

6.4. NASM Sembolik Makine Dili Derleyicisinde Temel Direktifler

Bir NASM dosyasında (aslında her sembolik makine dili programında) iki çeşit komut vardır:

1) Assembly Direktifleri: Bunlara sahte komutlar (pseudo instructions) da denilmektedir. Assembly direktifleri gerçek makine komutları değildir. Sembolik makine dili derleyicisine ne yapması gerektiğini belirten, kodu organize etmek için kullanılan komutlardır.

2) Gerçek Makine Komutları: Bunlar sembolik makine dili derleyicisi tarafından ikilik sisteme dönüştürülen makine komutlarını oluşturmaktadır. Yani bunlar işlemcinin çalıştıracağı makine komutlarını oluşturan komutlardır.

6.4.1. NASM'de Program Satırlarının Genel Yapısı

NASM programını oluşturan satırların genel biçimi şöyledir:

```
[etiket] [:] [komut] [; yorum]
```

Örneğin:

```
EXIT:       xor eax, eax    ; exit kod oluşturuluyor
```

Komutta bir etiketin olması onu kullanmayı zorunlu hale getirmez. Etiketler NASM'de adres belirtirler. Bunlar makine komutlarında kullanıldığında o adresi belirten düz sabitler gibi ele alınırlar. Etiketlerden sonra ':' atomu bulunmak zorunda değildir. Örneğin yukarıdaki komut ile aşağıdaki komut eşdeğerdir:

```
EXIT    xor eax, eax    ; exit kod oluşturuluyor
```

Veri bildirimlerindeki etkitelerde genellikle ':' tercih edilmemektedir. Ancak kod bölümündeki etiketlerde genellikle ':' kullanılmaz. Örneğin:

```
count               dd      10
```

ile,

```
count:          dd      10
```

eşdeğerdir.

NASM'de etiketlerin diğer dillerdeki değişkenlere benzediğine dikkat ediniz. Bir etiketin köşeli parantez içerisine alınmasıyla alınmaması arasında fark vardır. Örneğin:

```
mov     eax, count
```

Burada eax yazmacına count ile belirtilen adres değeri (aslında orada bir bilgi varsaa onun adresi) atanmaktadır. Halbuki:

```
mov     eax, [count]
```

Burada eax yazmacına count adresindeki değer (örneğimizde 10) atanacaktır.

Köşeli parantezler assembly'deki bir sentakstır. Yoksa ikilik sisteme dönüştürülen makine komutlarında köşeli parantezler bulunmaz. Makine dillerinde bellekteki nesneler her zaman komut içerisinde onların adresleri belirtilerek kullanılırlar. Örneğin aşağıdaki iki komutun da ikilik karşılığında count adresi bulunacaktır:

```
mov     eax, count
mov     eax, [count]
```

Ancak bu iki komutun ikilik sistem karşılığında ilgili adresin bir sabit değer mi olduğu yoksa o adresteki bilginin mi kastedildiği komutun bazı bitlerinde kodlanmaktadır. Şuraya dikkat ediniz: İşlemcide zaten bir adresteki bilgiye erişip onu kullanma yeteneği vardır. Köşeli parantezler yalnızca bunu sembolik olarak belirtmek için kullanılmaktadır.

Anahtar Notlar: MASM ve TASM sentaksında etiketlerin kendisi zaten köşeli parantezli olarak ele alınırlar. Örneğin:

```
mov eax, count
```

Bu komut MASM ve TASM'de "count adresinden başlayan dört byte'ı eax yazmacına yerleştir" anlamına gelir. Yani bunun NASM eşdeğeri şöyledir:

```
mov eax, [count]
```

MASM ve TASM'de offset anahtar sözcüğü etiketin adresini elde etmek için kullanılmaktadır. Yani MASM ve TASM'deki:

```
mov eax, offset count
```

komutunun NASM'deki eşdeğeri:

```
mov eax, count
```

biçimindedir.

6.4.2. NASM'de Veri Bildirim Direktifleri

NASM'de veri bildirimi için db, dw, dd, dq, dt direktifleri kullanılmaktadır. Bu direktifleri bir değer listesinin izlemesi gerekir. (Eğer bu direktifleri bir değer listesi izlemezse NASM derleyicisi bu durum için uyarı mesajı verir.). Değer listesi tek elemandan ya da virgül atomu ile ayrılmış birden fazla elemandan oluşabilir. Diğer assembly direktiflerinde olduğu gibi veri bildirim direktiflerinin başında da etiketler bulunabilir. Ancak bu zorunlu değildir. Örneğin:

```
counts    dd    100, 200, 300       ; geçerli
          dd    200                 ; geçerli
number    dd                        ; NASM default sıfır atayacaktır ancak uyarı mesajı oluşur
```

Veri bildirim direktiflerindeki etiketler NASM'de ilgili değerlerin bellekteki adreslerini belirtmektedir. Değerler sembolik makine dili derleyicisi tarafından belleğe ardışıl bir biçimde yerleştirilirler. Bu tür bildirimleri C/C++'taki global dizi tanımlamalarına benzetebiliriz. Örneğin:

```
int g_numbers[5] = {1, 2, 3, 4, 5};
```

Tanımlamasının eşdeğer NASM karşılığı şöyle düşünülebilir:

```
g_numbers       dd      1, 2, 3, 4, 5
```

Tabii bu direktiflerin en normal bulunduğu yer ".data" bölümüdür. ".bss" bölümünün ilkdeğer verilmemiş global nesneleri tuttuğunu anımsayınız. Ancak programın ".text" bölümünde de bu direktifler kullanılabilir. Örneğin:

```
  ;...
  jmp CONTINUE

  db 0x10
  db 0x20

CONTINUE:
```

```
    ;...
```

Bu direktiflerin yanındaki değer listesinde etiketler de kullanılabilir. Örneğin:

```
count       dd      100
pcount  dd      count           ; pcount count'un adresini tutuyor
```

Bazen ilkdeğer vermeden derleyicinin yalnızca belli uzunlukta yer ayırmasını isteyebiliriz. Bunun için resb, resw, resd, resq, rest direktifleri kullanılmaktadır. Bu direktiflerin sağ tarafında kaç tane eleman için yer ayrılacağına ilişkin bir değerin bulunması gerekir. Örneğin:

```
count       resd10
```

Burada count adresinden itibaren 10 tane 4 byte'lık yer ayrılmıştır. count ayrılan yerin başlangıç adresini temsil etmektedir. Ayrılan yerdeki elemanlara ilkdeğer verilmediğine dikkat ediniz. Örneğin:

```
numbers     resq    100
```

Burada da her biri 8 byte uzunluğunda 100 eleman için yer aayrılmıştır. Burada da numbers ayrılan yerin başlangıç adresini temsil etmektedir. RESX direktifleriyle yer ayırma işlemini C/C++'taki ilkdeğer verilmemiş global tanımlamalara benzetebiliriz. Örneğin:

```
int g_a[100];
```

bildiriminin NASM eşdeğeri şöyle oluşturulabilir:

```
g_a resd    100
```

İlkdeğer vermeden yer ayırma tipik olarak ".bss" bölümünde yapılan bir işlemdir. Gerçekten de NASM derleyicisi ".data" bölümünde bu direktiflerin kullanılması durumunda uyarı mesajı vermektedir.

Anahtar Notlar: İlkdeğer verilmeyen yer ayırmalar MASM ve TASM'de '?' sembolüyle belirtilmektedir. Örneğin NASM'deki:

```
number      resd    1
```

direktifinin MASM ve TASM karşılığı şöyledir:

```
number      dd      ?
```

Örneğin NASM'deki:

```
array       resd10
```

direktifinin MASM ve TASM'deki karşılığı:

```
array       dd  10 dup(?)
```

biçimindedir.

6.4.3. NASM'de Sabitler

NASM'de sabitler 10'luk, 16'lık, 8'lik ve 2'lik sistemlerde belirtilebilmektedir. Diğer pek çok dilde olduğu gibi NASM'de de sayılar için default sistem 10'luk sistemdir. Örneğin:

```
mov     eax, 123
```

Buradaki, 123 sayısı 10'luk sistemdeki 123 sayısıdır. Küçük 'd' ya da 'D', küçük 't' ya da 'T' sonekleri de 10'luk sistemi vurgulamak için kullanılabilir. Örneğin:

```
mov     eax, 123D   ; Geçerli fakat zaten default 10'luk sistem söz konusu, D'ye gerek yoktu
```

Büyük harf ya da küçük harf 'H' ya da 'X' soneki 16'lık sistemi, 'Q' ya da 'O' sonekleri 8'lik sistemi, 'B' ya da 'Y' sonekleri ise ikilik sistemi belirtmektedir. Örneğin:

```
mov     ah, 10001010b
```

ile,

```
mov     ah, 8ah
```

eşdeğerdir.

16'lık sistemde sayıları yazarken eğer ilk karakter alfabetik ise başına 0 (sıfır) getirilmesi zorunludur. Aksi takdirde derleyici onu sayı yerine isim olarak ele alacağına dikkat ediniz. Örneğin:

```
mov ah, abh
```

Burada abh 16'lık sistemde belirtilmiş bir sayı değil bir isim (örneğin bir etiket) olarak ele alınacaktır. Bu komut şöyle yazılmalıydı:

```
mov ah, 0abh
```

Genel olarak her türlü sabitin karakterleri arasına okunabilirliği artırmak amacıyla '_' (alt tire) karakteri getirilebilir. Sayı bu alt tire karakterleri sanki yokmuş gibi derleyici tarafından ele alınmaktadır. Örneğin:

```
mov eax, 100_000_000
```

Bu komutla aşağıdaki eşdeğerdir:

```
mov eax, 100000000
```

Tabii sayıları alt tire karakteri ile başlatamayız. Çünkü baştaki baştaki alt tire karakterleri alfabetik bir karakter olarak ele alınmaktadır. Örneğin:

```
mov eax, _100_000       ; _100_000 bir sayı gibi ele alınmaz
```

NASM'de H, X, O, Q, D, T, B, Y sonekleri önek biçiminde de kullanılabilmektedir. Ancak bu ekler önek olarak kullanılacaksa isimlerle karışmasın diye başına 0 (sıfır) getirilmek zorundadır. Örneğin:

```
mov     ah, 0q12                    ; geçerli 12 ocatal sistemde belirtilmiş
mov     eax, 0x12345678             ; geçerli 12345678 sayısı hex sistemde belirtilmiş
```

Ayrıca NASM'de yukarıdaki eklerin dışında $ sembolü de yalnızca önek olarak 16'lık sistemi belirtmek için de kullanılabilmektedir. Örneğin:

```
mov eax, $12345678
```

'$' sembolünün yalnızca önek olarak kullanılabildiğine ve onun önüne 0 getirilemediğine dikkat ediniz.

Anahtar Notlar: NASM'de sabit önekleri ve sonekleri konusunda neden bu kadar çok seçenek vardır? Aslında bu önek ve soneklerin çoğu başka sembolik makine dillerinde bulunmaktadır. NASM o dillerden geçmiş olan kişilerin alışkanlarını devam ettirmesi için seçeneği bol tutmuştur.

NASM'de karakterlerin sayısal kodlaarını belirtmek için tek tırnak ile iki tırnak arasında hiçbir farklılık yoktur. Tek tırnak ya da iki tırnak içerisine birden fazla karakter yerleştirilebilir. Bu durumda bunlar bu

karakterlerin ASCII tablosundaki sıra numaralarını anlatan sayılar olarak değerlendirilir. Örneğin:

```
mov eax, 'abcd'
```

Burada EAX'e 0x64636261 değeri yerleştirilir. Yerleştirmenin "little endian" formatına göre yapıldığına dikkat ediniz. Aşağıdaki komut da yukarıdakiyle eşdeğerdir:

```
mov eax, "abcd"
```

Tek tırnak ve iki tırnak içerisindeki karakter string'leri veri bildirim direktiflerinde daha çok karşımıza çıkar. Örneğin:

```
alpha db    "abcd"
```

Bu bildirim aşağıdakilerle eşdeğerdir:

```
alpha db    "a", "b", "c", "d"
alpha db    'a', 'b', 'c', 'd'
```

Örneğin sonu null karakter ile biten bir yazı için etiket aşağıdaki gibi oluşturulabilir:

```
namedb      'ali', 0
```

Pekiyi string kullanırken direktifin DB olması zorunlu mudur? Yanıt hayır. Örneğin:

```
betadd      'abcdef'
```

Bunun DB'den tek farkı burada derleyicinin her zaman direktifte belirtilen uzunluğun katı kadar yer tahsis etmesi ve bunun sonunu da 0 ile doldurmasıdır. Yani yukarıdaki direktifin eşdeğeri şöyledir:

```
betadb  'abcdef, 0, 0
```

'abcdef' karakterlerinin 6 byte yer kapladığına bunun 8 byte'a tamamlandığına dikkat ediniz.

Örneğin:

```
alpha dw    "a", "b"
```

Burada NASM alpha adresinden başlayarak toplam 4 byte yer ayıracaktır. Bu bildirim aşağıdaki ile eşdeğerdir:

```
alpha db    "a", 0, "b", 0
```

NASM'de string'ler içerisinde C'deki "ters bölü karakter sabitlerini" kullanabilmek için string'in `backquote` karakterleriyle oluşturulmuş olması gerekir. Bunun dışında string oluşturmak için backquote karakteri ile tek tırnak ve iki tırnak karakterleri arasında farklılık yoktur. Örneğin:

```
message         db      'merhaba dunya\0'
```

Buradaki \0 null karakter anlamına gelmez. Ters bölü karakteri ve 0 karakterleri anlamına gelir. Halbuki:

```
message         db      `merhaba dunya\0`
```

Burada \0 gerçekten null karakter anlamına gelmektedir. Yani yukarıdaki bildirimin eşdeğeri şöyledir:

```
message         db      'merhaba dunya', 0
```

Gerçek sayı sabitleri '.' karakteri kullanılarak oluşturulurlar. Örneğin:

```
weight  dd      82.3
```

Burada NASM nokta karakterinden dolayı sayıyı IEEE 754 float formatına göre kodlayacaktır. Aşağıdaki iki direktifin farklı anlamlara geldiğine dikkat ediniz:

```
number      dd      10              ; 10 tamsayı olarak kodlanacaktır
number      dd      10.0        ; 10 IEEE float formatına göre kodlanacaktır.
```

Noktanın sağı yine boş bırakılabilir:

```
number      dd      10.         ; 10 IEEE 754 float formatına göre kodlanacaktır.
```

Yine diğer dillerde olduğu gibi gerçek sayılar üstel formatta belirtilebilirler. Örneğin:

```
number dq       1e20
```

Eğer üst karakteri olarak 'e' yerine 'p' kullanılırsa 2'nin kuvveti anlaşılır. Ancak burumda sayının başına 0x ya da 0h getirilmelidir. Örneğin:

```
number  dd 0x2p10       ; 2 * 2^10 değerin float olarak yorumlanacak
```

6.4.3.1. EQU Direktifi

EQU direktifi pek çok assembly derleyicisinde çok benzer biçimde bulunmaktadır. Direktifin genel biçimi şöyledir:

```
<isim>      EQU     <sabit ifadesi>
```

Bu direktiften sonra artık direktifin başındaki isim sonundaki sabiti temsil eder hale gelmektedir. Direktifteeki sabit ifadesinin hesaplanması derleme aşamasında yapılmaktadır. Bu nedenle EQU direktifini #define gibi önişlemci direktifleriyle karıştırmayınız. Örneğin:

```
[BITS 32]

SECTION .data

msg             db          'Merhaba Dunya', 10
msg.len         equ     14
stdhandle       equ     -11
msg.written  dd         0

SECTION .text
    global _start
    extern _GetStdHandle@4, _WriteFile@20, _ExitProcess@4

_start:
    push stdhandle
    call _GetStdHandle@4

    push 0
    push msg.written
    push msg.len
    push msg
    push eax
    call _WriteFile@20

    push 0
    call _ExitProcess@4
```

EQU direktifi bellekte bir yer ayrılmasına yol açmamaktadır. EQU direktifi ile oluşturulan isim bir sabit biçiminde derleme işlemine sokulur. Örneğin:

```
yearequ 2016
;....
mov eax, year
```

Burada eax yazmacına 2016 sabit değeri atanmıştır. EQU direktifinin sağ tarafındaki ifadenin derleme aşamasında değeri hesaplanabilen bir ifade olması gerekir. Etiketlerin adres değerleri derleme aşamasında bilinmektedir. Örneğin:

```
message     db      ‘ankara’, 0
msgend      equ     message + 6     ; geçerli
```

Burada msgend bir sabit bir sayı belirtir. Bu sayı message etiketinin belirttiği adres değerinden 6 fazlasıdır.

6.4.4. Direktiflerdeki $ Karakterinin Koddaki Anlamı

Sabitler konusunda ‘$’ karakterinin hex sistemi belirten bir önek olarak kullanıldığını görmüştük. Ancak bu karakterin başka bir işlevi daha vardır. ‘$’ karakteri sanki bu karakterin kullanıldığı satırın başındaki etiketmiş gibi işleme sokulmaktadır. Yani ‘$’ bulunulan satırın etiketi anlamına gelmektedir. Örneğin:

```
message     db          'ankara', 10
message.len equ      $ - message
```

Burada message.len message yazısının byte uzunluğunu belirten bir sabit değer belirtir. Örneğin:

```
numbers         dd  10, 20, 30, 40, 50
numbers.size    equ ($ - numbers) / 4
```

Burada ise numbers.size söz konusu int dizinin eleman sayısını belirtmektedir.

6.4.5. TIMES Direktifi

Bu direktif bir komutu ya da direktifi çoklamak için kullanılır. Genel biçimi şöyledir:

```
times <sayı> <komut ya da direktif>
```

Örneğin 100 elemanlı ve tüm değerleri sıfır olan “.data” bölümünde bir dizi yaratmak isteyelim:

```
array       times 100   dd      0
```

Örneğin 512 byte uzunluğundaki boot sektör için bir program yazmış olalım. Program 512 byte’ı doldrumamış olsun biz de geri kalan kısmı sıfırlamak isteyelim. Ancak boot sektörün sonundaki iki byte’ın 0x55, 0xAA (boot signature) olması gereksin:

```
; program kodları var

times (510 – $)     db   0
signature                   db      0x55, 0xAA
```

Burada derleyici kodun geri kalan kısmına 0 yerleştirecektir. 512 byte’ın son iki byte’ında da 0x55 ve 0xAA değerleri bulunacaktır.

TIMES direktifi kod bölümünde de kullanılabilir.

```
times   10      NOP
```

Burada koda 10 tane NOP komutu yerleştirilmiştir.

Tabii TIMES direktifinin sağında birden fazla öğe de bulunabilir. Örneğin:

```
numbers times 100 dd 1, 2, 3, 4, 5
```

Burada 100 tane her biri 1, 2, 3, 4, 5 olan değerler yan yana bulunacaktır.

7. Sembolik Makine Dilinde Prosedürel Programlama Tekniği

Sembolik makine dilinde de program yazarken kodu tek parça değil fonksiyonların birbirlerini çağırması yoluyla yazmak isteyebiliriz. Programların tek parça halinde değil fonksiyonların birbirlerini çağırmaası biçiminde organize etmenin üç faydası vardır:

1) Kod tekrarı engellenmiş olur. Aynı işlemin birden fazla kez yapıldığı durumda o kodun tekrar tekrar yazılması yerine fonksiyon olarak ifade edilip CALL edilmesi kod tekrarını engellemektedir.

2) Kodların yeniden kullanılabilirliği (reeusability) sağlanmış olur. Böylelikle bir kez yazmış olduğumuz fonksiyonları değişik projelerde kullanabiliriz.

3) Kodun daha kolay algılanması sağlanmış ve okunabilirliği artırılmış olur. Belli işlerin fonksiyonlara yaptırılması kodun daha kolay anlaşılmasını ve yönetilmesini sağlamaktadır.

Sembolik makine dillerinde fonksiyon yazmak çok kolaydır. Örneğin NASM'de bunun için fonksiyonun başına onun başlangıç adresini temsil eden bir etiket yerleştirilir. Fonksiyonun sonu da ret makine koduyla sonlandırılır. Örneğin:

```
foo:
    ;.......
    ret

bar:
    ;.......
    ret
```

Bu fonksiyonların programın ".text" bölümünde alt alta bulunmasında hiçbir sakınca yoktur. Çünkü bu fonksiyonlar CALL makine komutuyla çağrılacağından zaten bunların sonundaki RET makine komutu da geri dönüşü sağlayacaktır. (Eğer bu fonksiyonların sonunda RET makine komutu olmasaydı akış aşağıya doğru devam ederdi değil mi?). Yukarıdaki fonksiyonların C karşılığı şöyle düşünülebilir:

```
void foo()
{
    /* ... */
}

void bar()
{
    /* ... */
}
```

Ancak sembolik makine dilinde prosedürel teknikle çaalışabilmek için bizim şunları bilmemiz gerekir?

1) Fonksiyonlara parametre aktaarımı nasıl yapılmaktadır?

2) Fonksiyonların geri dönüş değerleri nasıl oluşturulmaktadır?

3) Fonksiyonların yerel değişkenleri nasıl oluşturulup kullanılmaktadır?

Aşğıda sırasıyla bu konular ele alınmaktadır.

7.1. Sembolik Makine Dilinde Fonksiyonlara Parametre Aktarımı

Sembolik makine dillerinde fonksiyonlara parametre aktarımı üç yolla yapılabilmektedir:

1) Yazmaç Yoluyla
2) .data ya da .bss Alanı Kullanılarak Global Yolla
3) Stack Yoluyla

Sembolik makine dili terminolojisinde fonksiyonu çağıran fonkisyona İngilizce "caller", çağrılan fonksiyona da "callee" denilmektedir. Çağıran ve çağrılan fonksiyonların her ikisini de sembolik makine dilinde biz yazıyor olabiliriz ya da bunların yalnızca birini biz yazıyor olabiliriz. Örneğin programın bir kısmını belli bir C derleyicisinde yazıp oradan sembolik makine dilinde yazdığımız bir fonksiyonu çağırmak isteyebiliriz. Bu durumda "çağıran (caller)" o C derleyicisi tarafından oluşturulan fonksiyon "çağrılan (callee)" ise bizim yazdığımız fonksiyon olacaktır. Tabii tam tersi de olabilir. Biz C derleyicisi tarafından derlenmiş olan bir fonksiyonu sembolik makine dilinden çağırmak isteyebiliriz. Eğer çağıran ve çağrılan fonksiyonları farklı kişiler yazıyorlarsa bunların parametre aktarımı konusunda anlaşmış olmaları gerekir.

Şimdi bunları tek tek ele alalım.

7.1.1. Yazmaç Yoluyla Parametre Aktarımı

Bu aktarımda çağıran ve çağrılan fonksiyonlar parametrelerin hangi yazmaçlar yoluyla aktarılacağı konusunda anlaşırlar. (Örneğin birinci parametre EAX, ikinci parametre EBX, üçüncü parametre ECX ile aktarılabilir.) Böylece çağıran taraf fonksiyonu CALL etmeden önce parametreleri bu yazmaçlara yerleştirir. Sonra CALL işlemini yapar. Fonksiyon da o parametreleri o yazmaçlardan alarak kullanır. Örneğin iki int sayının toplamını ekrana yazdıran dispadd isimli bir fonksiyonu sembolik makine dilinde bu yöntemi kullanarak yazmak isteyelim. Yazmak istediğimiz fonksiyonu aşağıdaki gibi bir C prototipi ile temsil edebiliriz:

```
void dispadd(int a, int b);
```

Fonksiyon NASM'de aşağıdaki gibi yazılabilir:

```
dispadd:
    add eax, ebx
    ; eax'teki sonucu ekrana yazdır.
ret
```

Bu fonksiyonu şöyle çağırabiliriz:

```
mov     eax, 10
mov     ebx, 20
call    dispadd
```

Şimdi de sonu '\0' karakter ile biten bir yazıyı ekrana basan dispstr isimli bir fonksiyonu yazmak isteyelim. Fonksiyonun C'deki prototipi aşağıdaki gibi olsun:

```
void dispstr(const char *str);
```

Fonksiyonu NASM'de şöyle yazabiliriz:

```
dispstr:
    ; adresi eax'ten al ve uygun bir yöntemle yazdır
    ret
```

Bu fonksiyonu da aşağıdaki gibi çağırabiliriz:

```
section .data

messgae     db      'this is a test', 0
;....

mov eax, message
calldispstr
```

Yazmaç yoluyla aktarım aslında en hızlı aktarım biçimidir. Ancak sınırlılıkları vardır. Bazı işlemcilerde (örneğin Intel'de) az sayıda genel amaçlı yazmaç bulunur. Bunların parametre aktarımı için kullanılması fonksiyonun gerçekleştirilmesi sırasında yazmaç kıtlığı yaratabilir. (Tabii çağrılan fonksiyon aktarımın yapıldığı yazmaçları stack'e push edip sonra onları pop ederek de kullanabilir. Fakat bunlar ek makine komutlarını gerektirecektir.) Ayrıca bu biçimde yazmaçları izlemek programcı için zor da olabilmektedir.

Yazılım kesmelerinde aktarım hemen her zaman yazmaç yoluyla yapılmaktadır. (Kesmeler konusu ileride ele alınacaktır. Ancak yazılım kesmelerini şimdilik bir çeşit CALL işlemi gibi düşünebilirisiniz.) Örneğin Linux'un sistem fonksiyonları 80h kesmesine yerleştirilen tuzak kapısı (trap gate) yoluyla tetiklenir. Böylece Linux'ta biz bir sistem fonksiyonunu çağırmak istediğimizde önce parametreleri yazmaçlara yerleştiririz. Sonra INT makine komutuyla 80h kesmesini uygularız. Daha önce Linux için yazmış olduğumuz "Merhaba Dünya" programını bu bakımdan yeniden inceleyiniz:

```
; helloworld.asm

[BITS 32]

SECTION .data

msg         db  "Merhaba Dunya", 10

SECTION .text
    global _start

_start:
    mov     eax, 4
    mov     ebx, 1
    mov     ecx, msg
    mov     edx, 14
    int     80h

    mov     eax, 1
    mov     ebx, 0
    int     80h
```

Linux'ta her sistem fonksiyonunun bir numarası vardır. Örneğin write sistem fonksiyonun numarası 4, exit sistem fonksiyonun numarası 1'dir. Linux'ta sistem fonksiyonun numarası eax yazmacına yerleştirilir. Diğer parametreler de sırasıyla ebx, ecs, edx yazmaçlarına yerleştirilmektedir.

Şimdi Windows'ta yazmaç yoluyla parametre aktarımına bir örnek verelim. Örneğimizde dispstr fonksiyonu adresiyle aldığı bir yazının karakterlerini '\0' görene kadar ekrana basmaktadır. Bu fonksiyondaki ayrıntılara şimdilik dikkat etmeyiniz. Windows API fonksiyonlarında parametre aktarımını stack yoluyla yapmaktadır. Aşağıdaki kodda fonksiyonun içerisinde eax yazmacına gereksinim duyulduğu için önce o stack'e push edilmiştir.

```
[BITS 32]

SECTION .data

message1    db      'this is a test', 13, 10, 0,
message2    db      'this is another test', 13, 10, 0
message3    db      'this is the last test', 13, 10, 0

stdhandle       equ     -11

SECTION .text
    global _start
    extern  _GetStdHandle@4, _WriteFile@20, _ExitProcess@4

_start:

    mov     eax, message1
    calldispstr

    mov     eax, message2
    calldispstr

    mov     eax, message3
    calldispstr

    push0
    call_ExitProcess@4

    ret

dispstr:
    xor     ecx, ecx
    pusheax
REPEAT:
    inc     ecx
    mov     bl, [eax]
    inc     eax
    test    bl, bl
    jnz     REPEAT
    dec     ecx
    push    ecx

    pushstdhandle
    call_GetStdHandle@4

    pop     ecx
    pop     edx
    sub     esp, 4          ; yerel değişken için yer ayrılıyor
    push    0
    lea     ebx, [esp + 4]
    push    ebx
    push    ecx
    push    edx
    push    eax
    call    _WriteFile@20

    add     esp, 4
    ret
```

7.1.2. .data ya da .bss Alanı Kullanılarak Global Yolla Aktarım

Bu yöntemde programcı ".data" ya da ".bss" alanı içerisinde tahsis ettiği alanları parametre aktarımı için kullanır. Fonksiyonu CALL etmeden önce bu alanlara parametreleri yerleştirir. Sonra CALL işlemi yapar.

Fonksiyon da parametreleri o alanlardan alarak kullanır. Örneğin böyle bir aktarımda dispadd fonksiyonun çağrılması şöyle yapılabilir:

```
section .bss

param1      resd    1
param2      resd    1

.text

mov     dword [param1], 10
mov     dword [param2], 20
call    dispadd
```

Fonksiyon da şöyle şöyle yazılabilir:

```
dispadd:
    ; birinci parametreyi param1'den, ikinci aparemtreyi param2'Den al topla
    ; sonucu yazdır
    ret
```

Bu yöntem çok az tercih edilen bir yöntemdir. Çünkü iki önemli dezavantajı vardır:

1) Böyle bir tasarımda fonksiyon global sembollere bağımlı hale gelir. (Yani bu durumu C'de global değişken kullanan fonksiyonlara benzetilebilir.) Biz böyle bir fonksiyonu programdan programa kolay taşıyamayız. Taaşıma sırasında o fonksiyonun kullandığı etiketleri de taşıdığımız yerde yeniden oluşturmamız gerekir.

2) İç içe fonksiyon çağrılarında dıştaki fonksiyonun o global alandaki parametreleri saklaması gerekir. Aksi takdirde iki fonksiyon aynı etiketleri kullanıyorsa bozulma oluşur. Eğer her fonksiyon farklı etiketleri kullanıyorsa bu da hepten gereksiz alan harcanmasına yol açar.

7.1.3. Stack Yoluyla Parametre Aktarımı

Stack yoluyla aktarım en çok tercih edilen parametre aktarım yöntemidir. Bu yöntem çok sayıda parametrenin yazmaç kullanılmadan düzenli bir biçimde aktarılmasını mümkün kılarken iç içe fonksiyon çağrılarında da organizasyonel bir zorluğa yol açmamaktadır. Pek çok C derleyicisi default olarak parametre aktarımını stack yoluyla yapmaktadır.

Stack yoluyla aktarımda fonksiyonu çağıran fonksiyon (caller) önce parametreleri sırasıyla stack'e push eder, sonra CALL komutunu uygular. Bu yöntemde parametrelerin soldan sağa ya da sağdan sola stack'e push edilmesi arasında genel olarak bir fark yoktur. Ancak değişken sayıda argüman alan (örneğin printf ve scanf gibi) fonksiyonların yazılabilmesi için push işlemlerinin sağdan sola yapılması gerekir. Bu nedenle pratikte sağdan sola push işlemi soldan push işlemine tercih edilmektedir.

Stack yoluyla aktarımın nasıl yapıldığını yine iki sayının toplamını ekrana yazdıran dispadd gibi bir fonksiyon yoluyla açıklayalım. Fonksiyonun C'deki prototipini yeniden anımsatmak istiyoruz:

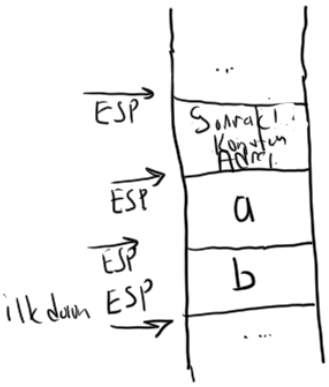
```
void dispadd(int a, int b);
```

Yukarıda da belirttiğimiz gibi stack yoluyla parametre aktarımında fonksiyonu çağıran taraf (caller) önce parametreleri sağdan sola stack'e push eder sonra da CALL işlemini yapar:

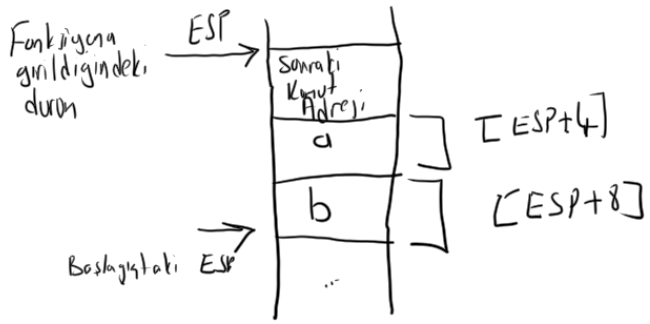
```
pushb
pusha
calldispadd
```

Akış çağrılan fonksiyona geldiğinde parametreler stack'tedir. Çağrılan fonksiyon (callee) onları stack'ten alarak kullanır. Pekiyi çağrılan fonksiyona göre parametreler stack'in neresindedir?..

Akış çağrılan fonksiyona geldiğinde stack'in durumu çok önemlidir. Buna "stack çerçevesi (stack frame)" denilmektedir. Örneğimizde akış dispadd fonksiyonuna geldiğinde stack'in (ve stack göstericisinin) durumu şöyle olacaktır:



Bu durumda çağrılan fonksiyon ilk parametreyi [ESP + 4] bellek operandı ile, ikinci parametreyi de [ESP + 8] bellek operandı ile stack'ten alabilir. Şekli biraz daha ayrıntılandıralım:



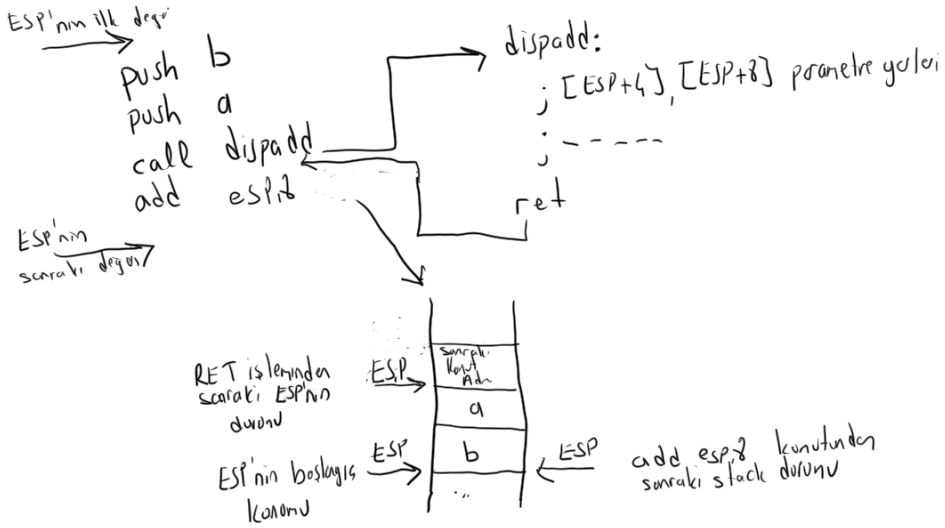
Görüldüğü gibi dispadd fonksiyonu hangi durumda çağrılırsa çağrılsın fonksiyon paarametrelere [ESP + 4] ve [ESP + 8] komutlarıyla erişebilmektedir. Intel işlemcilerinde 32 bit modda ESP yazmacının bellek erişimlerinde köşeli parantezler içerisinde kullanılabildiğini anımsayınız.

Stack yoluyla parametre aktarımında stack'in dengelenmesi önemli bir konudur. Fonksiyon çağrılmadan önce stack hangi durumdaysa (yani stack göstericisi neredeyse) çağırma işlemi bittiğinde onun yine aynı yerde olması gerekir. Yukarıdaki aktarımda parametreleri stack'e çağıran fonksiyon push etmiştir. CALL işlemi için fonksiyonda bir RET komutu bulunacağına göre CALLişlemi ile stack'e atılan değer RET işlemiyle geri alınmış olacaktır. Ancak çağıranın stack'e push ettiği parametreler stack dengesini bozmuştur. İşte stack yoluyla parametre aktarımında parametreler için stack dengelemesi çağıran fonksiyon tarafından ya da çağrılan fonksiyon tarafından yapılabilmektedir.

Stack dengelemesi çağıran fonksiyon tarafından CALL işleminden sonra ESP yazmacınının push edilen parametrelerin uzunluğu kadar artırılmasıyla yapılabilir. Örneğin:

```
push    b
push    a
call    dispadd
add     esp, 8
```

Bunu şekilsel olarak da aşağıdaki gibi gösterebiliriz:

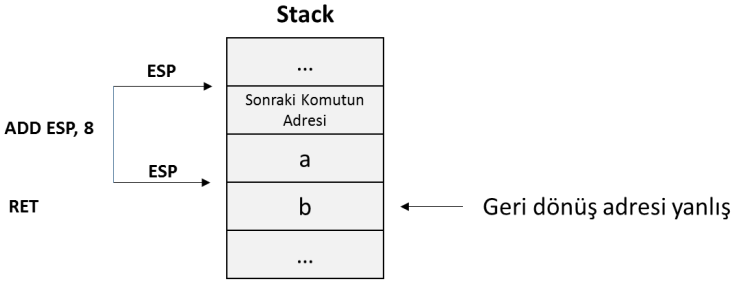


Kuşkusuz örneğimizde çağıran fonksiyon dengeleme işlemini iki boş POP komutuyla da yaapabilirdi:

```
push    b
push    a
call    dispadd
pop     eax
pop     eax
```

Tabii POP komutlarıyla stack'i dengelemeye çaalışmak çok sayıda argümanın push edildiği durumda ESP yazmacının artırılmasına göre çok daha maaliyetli olacaktır.

Stack denegelemesini çağrılan fonksiyon da yapabilir. Fakat nasıl? Burada sorun RET işleminden sonra ADD ESP, 8 gibi bir işlemi çağrılan fonksiyonun yapamamasıdır. (RET işleminden sonra akışın çağıran fonksiyona döndüğüne dikkat ediniz) Eğer bu işlemi çağrılan fonksiyon RET'ten önce yapsa bu durumda da RET komutunun stack'ten aldığı adres yanlış olacaktır. Bunu aşağıdaki şekilden kolaylıkla görebilirsiniz:



Tabii şöyle bir yol da düşünülebilir: Önce RET adresini POP etmek sonra ESP'yi artırmak ve POP edilen adresi PUSH ederek RET uygulamak. Örneğin:

```
dispadd:
    ;------------
    pop eax         ; geri dönüş adresi
    add esp, 8      ; şu anda stack dengede
    pusheax
    ret
```

Ancak bu çözüm çok fazla makine komutu gerektirdiği için etkin bir yöntem değildir. İşte bu nedenlerle Intel stack'i çağrılan fonksiyonun dengeleyebilmesi için RET makine komutunun "RET N" şeklinde kullanılan tek operandlı bir biçimini de bulundurmuştur. Eğer RET komutunun yanına bir değer yazılırsa atomik bir biçimde hem RET işlemi yapılmakta hem de ESP bu değer kadar artırılmaktadır. Başka bir

deyişle, örneğin:

```
ret     8
```

işleminin mantıksal eşdeğeri:

```
ret
add esp, 8
```

gibidir. Tabii RET işlemi ile ESP'nin artırılması atomik bir biçimde aynı komutta yapılmaktadır. Yani RET N makine komutu önce geri dönüş adresini stack'ten alır, sonra ESP'yi N kaadar artırır. Sonra da geri dönüş işlemi için jump yapar. Böylece eğer stack'in dengelemesini çağrılan fonksiyon yapacaksa geri dönüş için tipik olarak RET N makine komutu kullanılmaktadır. Bu durumda stack dengelemesinin çağrılan fonksiyon tarafından yapıldığında dispadd fonksiyonu şöyle çağrılacaktır:

```
push    b
push    a
call    dispadd
```

Fonksiyonun geri döndürülmesi de şöyle yapılacaktır:

```
dispadd:
    ; -----
    ret     8
```

Parametre aktarımında parametreleri bellekten çekmek için ESP'nin kullanılmasının bazı zorlukları söz konusu olabilmektedir. (Anımsanacağı gibi çağrılan fonksiyon ilk parametreyi [ESP + 4]'ten alıyordu.) Örneğin çağrılan fonksiyonun içerisinde PUSH işlemi yapılırsa parametrelerin yerleri de değişir. Tabii programcı (ya da derleyici) bu durumu izleyebilir. Ancak programcı için bu durum kavramsal zorluk yaratır. Ayrıca C99'daki gibi değişken uzunlukta yerel dizilerin tanımlanabildiği bir durumda ya da stack üzerinde programın çalışma zamanı sırasında tahsisatın yapıldığı bir durumda (alloca gibi) parametrelere ESP yoluyla erişmek mümkün olamamaktadır. (Bunun nedeni ileri ele alınacaktır.) Bunların yanı sıra 16 bit mod söz konusu olduğunda ESP'nin 16 bit karşılığı olan SP yazmacı da bellek operandı oluşturmak için kullanılamamaktadır. (SP'nin ESP olarak köşeli parantez içerisinde kullanılması 80386 ile başlamıştır.) İşte tüm bu gerekçelerle çağrılan fonksiyonun parametreleri [ESP + N] gibi bir komutla alması her zaman uygun olmayabilir. Bunun için parametrelere erişmede daha çok EBP yazmacı tercih edilmektedir.

EBP yazmacı yoluyla parametrelere erişim şöyle gerçekleştirilmektedir: Çağrılan fonksiyon hemen işin başında ESP'nin değerini EBP yazmacına yerleştirmek ister. (Artık hep parametrelere [EBP + N] bellek operandıyla ve aynı N değeriyle erişecektir). Ancak çağıran fonksiyon da EBP'yi kullanabiliyor olacağı için onun değerinin fonksiyon geri döndüğünde bozulmaması gerekir. İşte bunun için çağrılan fonksiyon ESP'yi EBP'ye atamadan önce EBP'yi push ederek stack'te saklar. Fonksiyondan çıkarken de EBP'yi pop ederek geri alır. Sonra da RET işlemini uygular. Böylece çağrılan fonksiyonun girişine ve çıkışına ilişkin tipik kalıp şöyle olacaktır:

```
foo:
    pushebp
    mov ebp, esp

  ;-----

    pop ebp
  ret
```

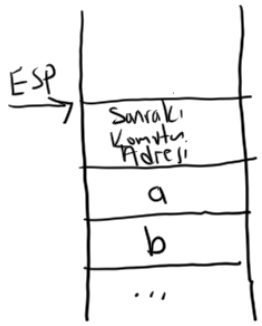
Pekiyi parametrelere EBP yazmacı yoluyla erişilmek istendiğinde parametrelerin yerleri çağrılan fonksiyona göre nerede olacaktır? Bunu aşağıdaki gibi iki parametre alan foo fonksiyonu üzerinden açıklayalım:

```
void foo(int a, int b);
```

Fonksiyonu şöyle çağırırız:

```
push    b
push    a
call    foo
add     esp, 8
```

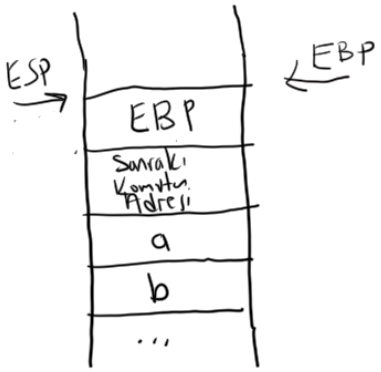
Bu noktada stack'in durumu şöyle olacaktır:



foo fonksiyonunun girişi aşağıdaki gibidir:

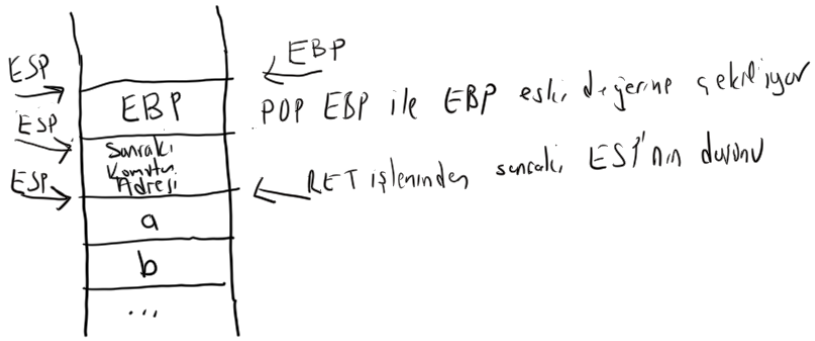
```
foo:
    push    ebp
    mov     ebp, esp
    ;------
```

PUSH EBP işleminden sonraki stack'in durumu ise şöyle olacaktır:



Böylece ilk parametreye [EBP + 8], sonraki parametreye ise [EBP + 12] bellek operandıyla erişilebilecektir. Pekiyi fonksiyondan çıkılırkenki stack'in durumu nasıldır?

```
;----------
pop ebp
ret
```



Çağrılan fonksiyon (callee) içerisnde neden EBP'nin saklandığını tam anlamamış olabilirsiniz. Bunu aşağıdaki gibi iç içe bir çağrı örneği ile açıklamaya çalışalım:

```
void bar(int a, int b)
{
    .....
    foo(10, 20);
    ....
}

void foo(int a, int b)
{
    ....
}
```

Burada EBP yazmacı hem bar içerisinde hem de foo içerisinde parametreleri almak için konumlandırılmış durumda olacaktır. İşte bar fonksiyonunun foo'yu çağırdıktan sonra kendi parametrelerini yine [EBP + 8] ve [EBP + 12]'den alabilmesi için foo'nun EBP'yi değiştirmemesi gerekir. Yukarıdaki C kodunun sembolik makina dili karşılığı da şöyle olacaktır:

```
bar:
    push    ebp
    mov     ebp, esp

  ; --------

  push      20
  push      10
  call      foo
  add       esp, 8

  ;-------

    pop     ebp
    ret


foo:
    push    ebp
    mov     ebp, esp

  ;-------

  pop       ebp
    ret
```

Pekiyi EBP'yi neden çağıran fonksiyon değil de çağrılan fonksiyon saklamaktadır? EBP'yi çağıran fonksiyon da saklayamaz mıydı? Şüphesiz bu durum da olabilirdi. Ancak EBP'yi çağrılan fonksiyonun saklaması daha uygundur. Böylece çağrılan fonksiyon parametreleri kullanmıyorsa EBP'yi hiç

saklamayabilir. Çünkü bu durumda fonksiyonun başındaki PUSH EBP ve MOV EBP, ESP komutlarına hiç gerek kalmayacaktır. Oysa EBP'yi çağıran fonksiyon (caller) saklasaydı bunu her durumda (fonksiyonun ne yaptığını bilemeyeceği için) yapması gerekirdi.

Stack yoluyla parametre aktarımına ilişkin tipik soru-cevaplar (faq'lar) şöyle olabilir:

Soru: Stack dengelemesi nedir?
Yanıt: Fonksiyonu CALL etmeden önce ve CALL ettikten sonra stack göstericisinin aynı yerde olması durumudur.

Soru: Stack yoluyla parametre aktarımında stack dengeleme sorunu neden ortaya çıkmaktadır?
Yanıt: Parametrelerin stack'e push edilmesi ile stack göstericisinin değeri azaltılmış olur. Bunun yeniden eski değerine getirilmesi gerekmektedir.

Soru: Stack'i çağıran fonksiyon dengeliyorsa bunu nasıl yapmaktadır?
Yanıt: Çağıran fonksiyon CALL işleminden sonra ya push işlemi kadar POP yaparak ya da ESP'yi push işlemi kadar artırarak dengelemeyi yapar.

Soru: Stack'i çağrılan fonksiyon dengeliyorsa bunu nasıl yapmaktadır?
Yanıt: Çağrılan fonksiyon RET işleminden önce ya da sonra ESP'yi artıramaz. Bunun için Intel'in RET N makine komutundan faydalanılmaktadır.

Soru: Parametrelere ESP yoluyla erişilecekse sağdan sola push işlemine göre parametrelerin yerleri nasıldır?
Yanıt: İlk parametre [ESP + 4], ikincisi [ESP + 8] biçimindedir.

Soru: Parametrelere EBP yoluyla erişilecekse sağdan sola push işlemine göre parametrelerin yerleri nasıldır?
Yanıt: İlk parametre [EBP + 8], ikincisi [ESP + 12] biçimindedir.

Soru: Neden ESP yerine EBP yoluyla parametre erişimi tercih edilmektedir?
Yanıt: Bunun üç nedeni vardır: Birincisi rahat çalışmak (yani parametrelerin hep aynı yerlerde aynı bellek operandıyla elde edilmesinin verdiği rahatlık). İkincisi stack'te programın çalışma zamanı sırasında tahsisat yapılması durumuyla başa çıkmak. Üçüncüsü ise geçmişe doğru uyumluluğu korumak. (Eskiden zaten ESP (SP) yazmacı bu amaçla kullanılamıyordu.)

Soru: C derleyicileri parametrelere ESP yoluyla mı, EBP yoluyla mı erişmektedir?
Yanıt: C derleyicileri her iki yöntemi de kullanabilmektedir. Optimizasyon seçenkleri açıksa ESP ile erişim daha az makine komutu ile yapılacağından durum da uygunsa derleyici tarafından tercih edilebilmektedir. Ancak pek çok durumda C derleyicileri parametrelere EBP yoluyla erişmektedir.

Soru: Fonksiyon parametrelere EBP yoluyla erişiyorsa EBP'yi neden saklamaktadır?
Yanıt: Onu çağıran fonksiyon da EBP'yi kullanıyor olabilir. Bu nedenle EBP fonksiyona girişte hangi değerdeyse çıkışta da aynı değerde olmalıdır.

Parametrelerin sağdan sola push edilmesi değişken sayıda argüman alan fonksiyonların yazılmasını mümkün hale getirmektedir. Bilindiği gibi C'de fonksiyonun değişken sayıda argümanla çağrılabileceği "... (ellipsis)" sentaksıyla belirtilmektedir. Örneğin:

```
void foo(int a, ...);
```

Örneğin değişken sayıda argüman alabilen printf fonksiyonunun da prototipi şöyledir:

```
int printf(const char *format, ...);
```

Değişken sayıda argüman alan fonksiyonları yazan kişi fonksiyonun kaç argümanla çağrılmış olduğunu anlamak zorundadır. İşte bu genellikle ilk argümanda açık ya da gizli bir biçimde kodlanır. Örneğin:

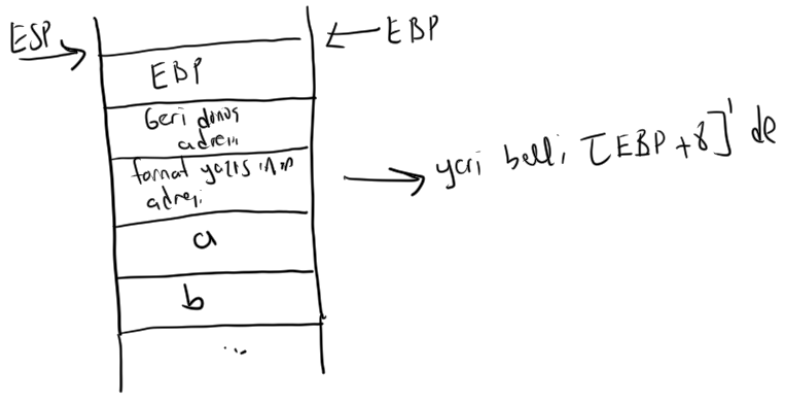
```
printf(“%d, %d\n”, a, b);
```

printf fonksiyonu birinci parametresindeki % karakterlerini sayarak fonksiyonun kaç argümanla çağrıldığını belirleyebilmektedir. UNIX/Linux sistemlerindeki execl POSIX fonksiyonun prototipini anımsayınız:

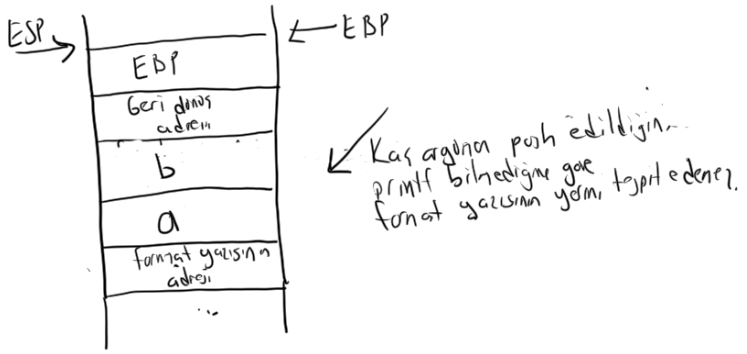
```
int execl(const char *path, ...);
```

Burada execl fonksiyonu argümanların sayısını son argümanın NULL adres olmasından anlamaktadır.

İşte mademki fonksiyonu yazan kişi fonksiyonun kaç argümanla çağrıldığını genellikle ilk parametreye bakarak anlıyor, o halde ilk parametrenin yerinin belli olması gerekir. Yukarıdaki örnekteki printf çağrısında parametrelerin stack’e sağdan sola push edildiği durumda stack’in durumu şöyle olacaktır:



Eğer parametreler soldan sağa push edilseydi ilk parametrenin yeri belli olmayacaktı. Örneğin:



Pekiyi şöyle bir tasarım olamaz mıydı? Parametreleri soldan sağa stack’e push etmek, fakat argümanların sayısını son argümandan tespit etmek. (Örneğin bu durumda printf fonksiyonunun format parametresi parametre listesinin sonunda olacaktır) Bu durum mümkün olsa da pek okunabilir değildir. Örneğin böyle bir tasarımda printf çağrısı da şöyle olacaktır:

```
printf(a, b, “%d, %d”);
```

Prototip ifadesinin de de biraz tuhaflaşacağına da dikkat ediniz (C ve C++!t “…” parametresi parametre listesinin sonunda bulunmak zorundadır):

```
int printf(..., const char *format);
```

İşte tüm bunlar değerlendirildiğinde parametrelerin sağdan sola push edilmesi daha anlamlı gözükmektedir. Fakat Pascal gibi bazı programlama dillerinde parametreler eskiden soldan sağa push ediliyordu. (Artık Pascal ve Delphie dillerinde de parametreler uzun süredir sağdan sola push edilmektedir.)

7.2. Fonksiyonların Geri Dönüş Değerlerinin Oluşturulması ve Kullanılması

Fonksiyonların geri dönüş değerleri tıpkı parametrelerde olduğu yazmaç yoluyla, stack ya da ".data" ve ".bss" yoluyla oluşturulup aktarılabilir. Ancak geri dönüş deperlerinin oluşturulması ve aktarılması için hemen her zaman yazmaçlar tercih edilmektedir. Örneeğinn C derleyicileri geri dönüş değerlerini yazmaçlar yoluyla aktarmaktadır.

Yazmaç yoluyla geri dönüş değerinin aktarımında önce çağrılan fonksiyon geri dönüş değerini oluşturur. Onu önceden üzerinde anlaşılan bir yazamaca yerleştirdikten sonra RET işlemi ile fonksiyonu sonlandırır. Çağıran fonksiyon da CALL işleminden sonra geri dönüş değerini ilgili yazmaçtan alır.

32 bit Intel mimarisinde genellikle 8 bitlik geri dönüş değerleri AL yazmacı ile, 16 bitlik geri dönüş değerleri AX yazmacı ile (yani EAX'in düşük anlamlı WORD kısmı ile), 32 bitlik geri dönüş değerleri EAX yazmacı ile ve 64 bitlik geri dönüş değerleri de EDX:EAX yazmaçları ile aktarılmaktadır. Adresler için de yine EAX yazmacı kullanılmaktadır. float, double ve long double geri dönüş değerleri ise matematik işlemcisinin stack yazmacı yoluyla aktarılırlar. (Yani fonksiyon FLD komutu ile geri dönüş değerini stack'te bırakır. Çağıran taraf da FSTP ile onu stack'ten alıp stack'i dengeler.) Geri dönüş değeri yapı türünden olan fonksiyonlarda genellikle aktarım çağıran tarafın tahsisatı stack üzerinde yapması ve onun adresini fonksiyona geçirmesi, fonksiyonun da geri dönüş değerini o adrese aktarması yoluyla gerçekleştirilmektedir.

Örneğin iki sayının en büyüğü ile geri dönen aşağıdaki fonksiyonu sembolik makine dilinde yazacak olalım:

```
int max(int a, int b);
```

Fonksiyonun sembolik makine dilindeki karşılığı şöyle olabilir:

```
;----------
   mov        eax, [b]
   push       eax
   mov        eax, [a]
   push       eax
   call       max
   add        esp, 8

; geri dönüş değeri eax'te, oradan alınarak kullanılabilir

; ----------

max:
   push      ebp
   mov       ebp, esp
   mov       eax, [ebp + 8]
   cmp       eax, [ebp + 12]
   jg        @1
   mov       eax, [ebp + 12]
@1:
   pop       ebp
ret
```

Örneğin aşağıda C prototipi verilmiş olan mysqrt fonksiyonunu yazmak isteyelim:

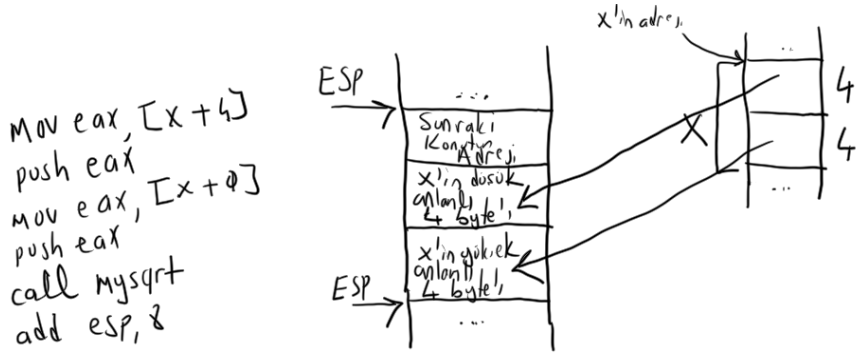
```
double mysqrt(double x);
```

Fonksiyonun sembolik makine dili çıktısı şöyle olabilir:

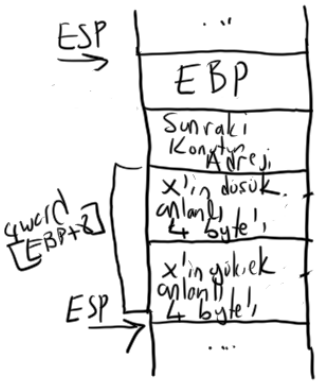
```
section.data
x         dq     9.0
result    dq     10
```

```
section.text
;-----------
   mov       eax, [x + 4]
   push    eax
   mov       eax, [x + 0]
   push    eax
   call    mysqrt
   add       esp, 8
   fstp    qword [result]

   ; geri dönüş değeri result'ta
   ret

mysqrt :
   push   ebp
   mov    ebp, esp
   fld    qword[ebp + 8]
   fsqrt          ; sonuç matematik işlemcinin ST(0) yazmacında bırakılıyor
   pop    ebp
   ret
```

Burada mysqrt fonksiyonuna double değerin stack yoluyla nasıl aktarıldığına dikkat ediniz:



8 byte'lık double değer stack yoluyla aktarılırken "little endian" notasyona göre yine stack'te düşük adreste onun düşük anlamlı kısmı bulunmalıdır. 32 bit sistemde 64 bitlik değerler tek hamlede push edilemez. Bu nedenle x'in aktarımı iki ayrı PUSH işlemi ile yapılmıştır. Double değerin (yani x'in) önce yüksek anlamlı 4 byte'lık kısmının sonra düşük anlamlı 4 byte'lık kısmının push edildiğine dikkat ediniz. Böylelikte stack'te double değerin yine düşük anlamlı 4 byte'ı düşük adreste bulunacaktır. Akış mysqrt fonksiyonuna geldiğinde EBP push edildikten sonraki stack durumu şöyle olacaktır:



mysqrt fonksiyonunun geri dönüş değerini matematik işlemcinin stack'inde bıraktığına dikkat ediniz. (FSQRT komutunun operandını matematik işlemcinin stack'inden alarak sonucu yeniden matematik işlemcinin stack'ine push ettiğini anımsayınız). mysqrt fonksiyonunu çağıran kod geri dönüş değerini aşağıdaki gibi matematik işlemcinin yazmacından alarak result adresine yerleştirmiştir:

```
fstpqword [result]
```

7.3. Fonksiyon Çağrılarında Yazmaç Değerlerinin Korunması Sorunu

Sembolik makine dilinde bir fonksiyonu CALL ettikten sonra akış geri döndüğünde yazmaçların durumu ne olacaktır? CALL edilen fonksiyon yazmaçların değerlerini değiştirmiş olabilir. Bu durumda CALL etmeden önce çağıran fonksiyon yazmaçlarda belli değerleri saklamışsa CALL işleminden sonra artık o değerlerin yazmaçlarda olmayabileceğini göz önüne almalıdır. İşte bu konuda da çağıran fonksiyonla (caller) çağrılan fonksiyon (callee) bir anlaşma yapabilirler. Bu anlaşmaya göre çağrılan fonksiyon bazı yazmaçları koruyabilir, bazılarını korumayabilir. Örneğin C derleyicilerinde pek çok çağırma biçiminde çağrılan fonksiyonun EAX, ECX ve EDX yazmaçlarını bozmasına izin verilmiştir. Ancak diğer yazmaçları çağrılan fonksiyon korumalıdır. Yani onların değerleri akış fonksiyona girdiğinde neyse çıktığında da aynı olmalıdır. Tabii bazı yazmaçların korunmasında bir anlaşma yapılmışsa bu durum çağrılan fonksiyonun o yazmaç değerlerini hiç değiştirmeyeceği amlamına gelmez. Çağrılan fonksiyon eğer bu yazmaçların değerlerini değiştirecekse önce onların değerlerini stack'e push saklayabilir. Fonksiyondan çıkmadan önce de pop eder geri yükleyebilir. Ancak onları bu biçimde koruma sorumluluğu çağrılan fonksiyona aittir.

Çağıran ve çağrılan fonksiyonların her ikisini de biz yazacaksak hangi yazmaçların çağrılma sırasında çağrılan fonksiyonlar tarafından korunucağını yine biz kendimiz belirleyebiliriz. Eğer biz yalnızca çağrılan fonksiyonu yazacaksak bu durumda çağıran fonksiyonun hangi yazmaçların korunacağı konusundaki beklentisini karşılamamız gerekebilir. Eğer biz yalnızca çağıran fonksiyonu yazacaksak çağrılan fonksiyonun hangi yazmaçları koruduğunu bilmek yine bize fayda sağlayabilir. Örneğin biz programın büyük kısmını C'de yazmış olalım ve oradan sembolik makine dilinde yazmış olduğumuz fonksiyonu çağırmak isteyelim. Bu durumda bizim C derleyicisinin yazmaç koruması konusundaki beklentilerini (kurallarını) karşılamamız gerekir. Çünkü derleyici bazı yazmaçların fonksiyon tarafından bozulmayacağı beklentisiyle fonksiyon çağrısından sonra o yazmaçlardaki değerleri kullanıyor olabilir.

7.4. Yerel Etiketler

Anımsanacağı gibi NASM'de her komutun başına bir etiket getirilebilmekteydi. Ayrıca etiket isimlerinden sonra ':' atomu da zorunlu değildi. NASM'de etiketlerin faaliyet alanları tüm sembolik makine dili dosyasını kapsamaktadır (file scope). Yani aynı isimli birden fazla etiket aynı NASM kaynak dosyasında bulunamaz. Öte yandan etiketler için isim uydurmak da bir sorundur. İşte etiketlerin faaliyet alanını bir fonksiyonla sınırlandırmak için "yerel etiket" kavramı düşünülmüştür.

Yerel etiketler '.' karakteriyle başlatılır. Aslında bir yerel etiket ondan önceki ilk normal etiketin isimsel olarak kombine edilmiş biçimidir. Şöyle ki:

```
foo:
    ;....
    .REPEAT:
    ;....
    jmp .REPEAT
    ;....
bar:
    ;....
    .REPEAT:
    ;....
    jmp .REPEAT
    ;....
```

Burada aslında foo etiketinden sonraki .REPEAT yerel etiket ismi "foo.REPEAT" etiket ismi ile edşdeğerdir. Benzer biçimde bar etiketinden sonraki .REPEAT yerel etiketi de aslında "bar.REPEAT" etiket ismiyle eşdeğerdir. Başka bir deyişle bizim foo fonksiyonun içerisinde ".REPEAT" ismini kullanmamızla "foo.REPEAT" ismini kullanmamız tamamen eşdeğerdir. Örneğin:

```
foo:
    ;....
```

```
    .REPEAT:
    ;....
    jmp foo.REPEAT      ; geçerli!
    ;....
```

Tabii başka bir fonksiyondan ".REPEAT" yerel etiketine JMP ya da CALL işlemi de yapabiliriz. Ancak bu etiketi orada uzun ismiyle (yani "foo.REPEAT" biçiminde) belirtmemiz gerekir.

7.5. 32 Bit C ve C++ Derleyicilerinde Fonksiyon Çağırma Biçimleri (Calling Conventions)

Fonksiyonların çağrılması ve geri dönüş değerlerinin alınması konusundaki belirlemelere "çağırma biçimi (calling convention)" denilmektedir. Fonksiyon çağırma biçimleri şu konulardaki belirlemeleri içerir:

1) Çağıran fonksiyon ile çağrılan fonksiyon arasında parametre aktarımı nasıl yapılacaktır?

2) Çağrılan fonksiyonun geri dönüş değeri çağıran fonksiyona nasıl aktarılacaktır?

3) Çağrılan fonksiyon hangi yazmaçları bozma hakkına sahiptir, hangi yazmaçları korumak zorundadır?

Çağırma biçimi C standartlarında olan bir konu değildir. Çünkü C standartları böylesi aşağı seviyeli belirlemeleri derleyicilere bırakmıştır. Dolayısıyla çağırma biçimlerini oluşturmak için gereken anahtar sözcükler de derleyicilerde bir eklenti (extension) biçiminde bulunurlar. Çağırma biçimlerine ilişkin anahtar sözcükler genel olarak tür belirten sözcük ile dekleratörün arasına yerleştirilmektedir. Örneğin:

```
void __cdecl foo(int a, int b)
{
    ...
}
```

Microsoft derleyicilerinde çağırma biçimleri yukarıdaki örnekte olduğu gibi iki alt tire ile başlayan anahtar sözcüklerle temsil edilmektedir. gcc derleyicilerinde ise "fonksiyon özellikleri (function attributes)" biçimindeki bir sentksla temsil edilir. Örneğin:

```
void __attribute__((cdecl)) foo(int a, int b)
{
    ...
}
```

Şimdi Microsoft ve gcc derleyicilerindeki çağırma biçimlerini tek tek ele alacağız. Ancak burada bir noktayı vurgulamak istiyoruz: Gerek Microsoft gerekse gcc derleyici ailelerinde 32 bit uygulamalardaki çağırma biçimleriyle 64 bit uygulamalardaki çağırma biçimleri tamamen farklıdır. Aşağıda yalnızca 32 bit uygulamalardaki çağırma biçimleri ele alınmaktadır. 64 bit uygulamalardaki çağırma biçimleri 64 bit çalışmanın anlatıldığı bölümde ele alınacaktır (64 bit sistemlerdeki gcc derleyicilerinde default derlemenin 64 olduğunu anımsayınız. Bu sistemlerdeki gcc derleyicilerinde 32 bit derleme yaparken -m32 seçeneğini kullanmayı unutmayınız. 64 bit Windows sistemlerinde iki ayrı cl.exe derleyicisi vardır. Default olarak hangisinin devreye girdiği PATH çevre değişkenine bağlı olmaktadır.)

7.5.1. cdecl (C Declaration) Çağırma Biçimi

Bu çağırma biçimi Microsoft ve gcc derleyicilerinde C Programlama Dili için default durumdur. (Yani bu derleyicilerde fonksiyon bildirimlerinde çağırma biçimi hiç belirtilmezse sanki bu çağırma biçimi belirtilmiş gibi işlem işlem yapılır.) Her iki derleyici ailesinde de C++'taki global ve static üye fonksiyonlar için de yine default olarak bu çağırma biçimi kullanılmaktadır. Ancak sınıfların static olmayan üye fonksiyonları için Microsoft derleyicilerindeki default çağırma biçimi "thiscall" iken gcc derleyicilerindeki cdecl biçimindedir.

cdecl çağırma biçiminin 32 bit Intel sistemindeki kuralları şunlardır:

1) Parametre aktarımında stack kullanılır ve parametreler sağdan sola stack'e push edilirler. Parametreler için stack çağıran fonksiyon (caller) tarafından dengelenmektedir.

2) Geri dönüş değerleri yazmaçlar yoluyla aktarılmaktadır. 8 bitlik geri dönüş değerleri AL yazmacı ile, 16 bitlik geri dönüş değerleri AX yazmacı ile (yani EAX'in düşük anlamlı WORD değeri ile), 32 bitlik tamsayı geri dönüş değerleri EAX yazmacı ile ve 64 bitlik tamsayı geri dönüş değerleri de EDX:EAX yazmacı ile aktarılır. float, double ve long double geri dönüş değerlerinin aktarımı için matematik işlemcinin ST(0) yazmacı kullanılmaktadır. Geri dönüş değeri adres türünden olan fonksiyonlarda aktarım için yine EAX yazmacı kullanılır. Geri dönüş değeri yapı türünden olan fonksiyonlarda ise aktarımda önce çağıran fonksiyon geri dönüş değeri için gereken yapı alanını stack'te tahsis eder, onun adresini fonksiyona gönderir, fonksiyon da geri dönüş değerini bu adrese yerleştirir.

3) EAX, ECX ve EDX yazmaçları çağrılan fonksiyon tarafından bozulabilir. Fakat diğer yazmaçlar çağrılan fonksiyon tarafından korunmalıdır.

Eğer biz sembolik makine dilinde bu çağırma biçimine uygun bir fonksiyon yazmışsak, C'den çağırırken onun prototipinde -cedecl default biçim olduğu için- çağırma biçimini hiç belirtmeyebiliriz. Ya da bunu aşağıdaki gibi belirtebiliriz:

```
void __cdecl foo(int a, int b);                          /* Microsoft derleyicileri için
prototip */
void __attribute__((cdecl)) foo(int a, int b);       /* gcc derleyicileri için prototip */
```

7.5.2. fastcall Çağırma Biçimi

Bu çağırma biçiminin kuralları Microsoft ve gcc derleyicilerinde yine aynıdır:

1) Parametre aktarımı hem yazmaç hem de stack yoluyla yapılmaktadır. Şöyle ki: Bu çağırma biçiminde ilk iki parametre sırasıyla ECX ve EDX yazmaçlarıyla aktarılır. Eğer parametre sayısı ikiden fazlaysa diğer parametrelerin aktarımı için cdecl çağırma biçimindeki gibi stack kullanılır. (İlk iki parametreden sonraki parametreler yine stack'e sağdan sola push edilir ve stack yine çağrılan fonksiyon tarafından (callee) dengelenir.)

2) Geri dönüş değeri tamamen cdecl'de olduğu gibi yazmaç yoluyla yapılmaktadır.

3) Yine EAX, ECX ve EDX yazmaçları çağrılan fonksiyon tarafından bozulabilir. Fakat diğer yazmaçlar çağrılan fonksiyon tarafından korunmalıdır.

fastcall çağırma biçimi için prototip ifadesi şöyle oluşturulabilir:

```
void __fastcall foo(int a, int b);           /* Microsoft derleyicileri için prototip */
void __attribute__((fastcall)) foo(int a, int b);    /* gcc derleyicileri için prototip */
```

7.5.3. stdcall Çağırma Biçimi

Bu çağırma biçimi Windows sistemlerinde çok yaygın kullanılmaktadır. Windows'un bütün API fonksiyonları ve adresleri bizden alınarak bunlar tarafından çağrılan "callback" fonksiyonlar bu çağırma biçimine sahiptir. Örneğin ExitProcess fonksiyonunun prototipi şöyledir:

```
void WINAPI ExitProcess(UINT uExitCode);
```

API fonksiyonlarının isimlerinin önündeki WINAPI <windows.h> dosyası içerisinde şöyle define edilmiştir:

```
#define WINAPI  __stdcall
```

stdcall çağırma biçimi Linux sistemlerinde seyrek kullanılmaktadır.

stdcall çağırma biçiminin kuralları şöyledir:

1) Parametre aktarımında stack kullanılır ve parametreler sağdan sola stack'e push edilirler. Stack çağrılan fonksiyon (callee) tarafından dengelenir. (cdecl çağırma biçiminde stack'i çağıran fonksiyonun dengelediğini anımsayınız.)

2) Geri dönüş değerlerinin aktarımı tamamen cdecl çağırma biçimindeki gibi yazmaç yoluyla yapılmaktadır.

3) Yine EAX, ECX ve EDX yazmaçları çağrılan fonksiyon tarafından bozulabilir. Fakat diğer yazmaçlar çağrılan fonksiyon tarafından korunmalıdır.

Intel sisteminde stack'i çağrılan fonksiyonun dengelemesinin değişken sayıda argüman alan fonksiyonların yazılabilmesini engellediğini anımsayınız. Bu nedenle Windows'ta değişken sayıda parametreye sahip olan bir API fonksiyonu yoktur.

```
void __stdcall foo(int a, int b);        /* Microsoft derleyicileri için prototip */
void __attribute__((stdcall)) foo(int a, int b);     /* gcc derleyicileri için prototip */
```

7.5.4. thiscall Çağırma Biçimi

thiscall çağırma biçimi C++'ta sınıfların static olmayan üye fonksiyonlarının çağrılmasında kullanılmaktadır. Bu çağırma biçiminde static olmayan üye fonksiyonlara this göstericisi ECX yazmacıyla aktarılır. Diğer parametrelerin aktarımı ise tamamen __stdcall çağırma biçimindeki gibidir. Yani parametreler sağdan sola stack'e push edilirler. Parametreler için stack'i çağrılan fonksiyon (callee) dengeler. Geri dönüş değerinin aktarımı da yazmaçlar yoluyla yapılmaktadır. Yine EAX, ECX ve EDX yazmaçlarını çağrılan fonksiyon bozabilir fakat diğerlerini korumak zorundadır.

7.6. C ve C++'tan Sembolik Makine Dilinde Yazılmış Fonksiyonların Çağrılması

Bir programın tamamının sembolik makine dilinde yazılması çok özel durumlar dışında tercih edilen bir yol değildir. Bunun nedenleri şöyle sıralanabilir:

1) Sembolik makine dilleri taşınabilir (portable) değildir. Makine komutları işlemciden işlemciye değişebildiği gibi genel sentaks da aynı işlemci söz konusu olduğunda bile derleyiciden derleyiciye değişebilmektedir.

2) Sembolik makine dilleri alçak seviyeli olduğu için bu dillerde programcıların hata yapması daha kolaydır.

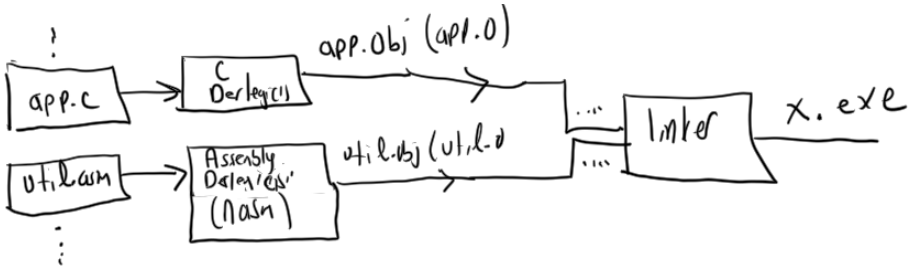
3) Sembolik makine dillerinde kod yazım hızı daha düşüktür. Bu da üretkenliği (birim zamanda yazılan kod miktarını) düşürmektedir.

4) Sembolik makine dillerinde programın debug edilmesi daha zordur.

İşte bu nedenlerden dolayı pratikte kodun büyük bölümünün C, C++ gibi yüksek seviyeli ve taşınabilir dillerde, yalnızca kritik yerlerin sembolik makine dilinde yazılması tercih edilmektedir. (Örneğin Linux gibi bir işletim sistemi çekirdeğinde bile kodların %97'si C ile yazılmıştır. Linux çekirdeğinin yalnızca %3'lük bir kısmı sembolik makine dilinde yazılmıştır.) Bunlar göz önüne alındığında sembolik makine dilinde yazılan kodların C ve C++ gibi dillerden çağrılmasının önemi anlaşılabilir. Bu bölümde bu işlemin nasıl yapıldığı ele alınmaktadır.

Sembolik makine dilinde yazılmış bir fonksiyonun C ya da C++'tan çağrılabilmesi için önce onun sembolik makine dili derleyicisiyle derlenmesi gerekir. Bu derleme işleminden bir amaç dosya (object module) elde

edilir. (Amaç dosya uzantılarının Windos'ta ".obj" biçiminde, UNIX/Linux sistemlerinde ".o" biçiminde olduğunu anımsayınız.) Sonra bu fonksiyonu çağıran C ya da C++ kodu da bu dillerin derleyicileri ile derlenir. Bu işlemden de bir amaç dosya (object module) elde edilir. Nihayet bu amaç dosyalar birlikte link işlemine sokularak çalıştırılabilir (executable) dosya olurulur. Örneğin "app.c" isimli C programından "util.asm" isimli sembolik makine dilinde yazılmış olan fonksiyonlar çağrılmak istensin. Yapılacak işlemleri aşağıdaki şekille özetleyebiliriz:



Yukarıda özetlediğimiz adınmlardan sorunsuz geçilebilmesi için şu noktalara dikkat edilmesi gerekir:

1) Sembolik makine dilinde fonksiyonu yazarken onun için NASM'de "global" bildirimi yapmak gerekir. ("global" bildirimine neden duyulduğu ileride ele alınacaktır.)

2) Derleyiciler global fonksiyonların ve değişkenlerin isimlerini amaç dosyaya yazarken değiştirebilmektedir. Bu duruma "isim dekorasyonu (name decoration)" denilmektedir. İsim dekorasyonu ileride ele alınacaktır. Ancak burada şunu ifade etmek istiyoruz: İsim dekorasyonu çağırma biçimine göre, derleyiciye göre ve hatta dile göre değişebilmektedir. Örneğin Windows'ta Microsoft derleyicileri "cdecl" çağırma biçiminde global sembollerin (fonksiyon ve değişkenlerin) başına bir alt tire (underscore) eklemektedir. Halbuki Linux'ta gcc derleyicileri bu isimleri hiç değiştirmeden amaç koda yazmaktadır. (Dolayısıyla Windows'ta bizim de sembolik makine dilinde yazdığımız fonksiyonların başına alt tire eklememiz gerekir. Aksi takdirde linker iki modüldeki isimleri eşleyemez.)

3) C ya da C++'ta belirlenen çağırma biçimine sembolik makine dilinde uyulmalıdır. (Yani örneğin C'de "cdecl" çağırma biçimine sahip olarak çağırdığımız add fonksiyonunu sembolik makine dilinde yazarken parametrelerin sağdan sola stack'e atılacağını bilerek fonksiyonu yazmalıyız ve geri dönüş değerini de EAX yazmacında bırakmalıyız.)

4) Çağrılan fonksiyonun hangi yazmaçları saklaması gerektiği bilinmelidir ve fonksiyonu yazarken bu kurala uyulmalıdır. Örneğin cdecl ve stdcall çağırma biçimlerinde çağrılan fonksiyon EAX, ECD ve EDX yazmaçlarını bozma hakkına sahiptir. Fakat diğerlerinin değerlerini değiştirecekse önce onları stack'te saklamalı fonksiyon sonlanmadan önce geri almalıdır.

5) Semboik makine dilindeki kodlarla C ya da C++'taki kodların aynı bölüm (section) içerisinde bulunması gerekir. (Aslında bu konunun biraz ayrıntıları vardır. Farklı bölümlere yerleştirilmiş kodlar platforma bağlı olarak sorun oluşturmayabilir.) Örneğin C ve C++ derleyicileri Windows ve Linux sistemlerinde kodu ".text" isimli bölüme yerleştirmektedir. O halde bizim de fonksiyonları ".text" isimli bölümde yazmamız gerekir. Aynı isimli bölümler linker tarafından birleştirilmektedir. (Yani birleştirme işlemi sonucunda çalıştırılabilen dosyada tek bir ".text" bölümü bulunacaktır)

6) C ya da C++ kaynak programında sembolik makine dilinden çağrılan fonksiyonun prototip bildirimi çağırma biçimi aynı olacak biçimde yapılmalıdır.

Örneğin Windows'ta iki sayının toplamına ve çarpımına geri dönen add ve multiply isimli fonksiyonları sembolik makine dilinde yazıp C'den çağırmak isteleyim. Bu işlemlerin hepsi komut satırından yapılacak olsun. Bunlar için NASM kaynak dosyası şöyle oluşturulabilir:

```
; util.asm

[BITS 32]

SECTION .text
    global _add, _multiply

_add:
    push   ebp
    mov    ebp, esp
    mov    eax, [ebp + 8]
    add    eax, [ebp + 12]
    pop    ebp
    ret

_multiply:
    push   ebp
    mov    ebp, esp
    mov    eax, [ebp + 8]
    mul    dword [ebp + 12]
    pop    ebp
    ret
```

C kaynak dosyası da şöyle oluşturulabilir:

```
/* app.c */

#include <stdio.h>

int add(int a, int b);
int multiply(int a, int b);

int main(void)
{
    printf("%d\n", add(10, 20));
    printf("%d\n", multiply(10, 20));

    return 0;
}
```

NASM kaynak dosyası şöyle derlenmelidir:

```
nasm -fwin32 util.asm
```

Buradan ürün olarak "util.obj" dosyası elde edilecektir. C kaynak dosyası da komut satırından şöyle derlenebilir:

```
cl -c app.c
```

Buradan da ürün olarak "app.obj" dosyası elde edilir. Microsoft'un linker'ı ile link işlemi de şöyle yapılabilir:

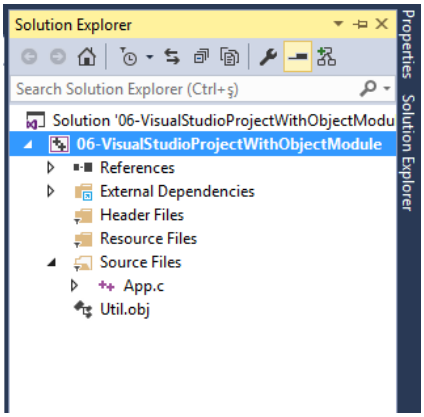
```
link /out:app.exe app.obj util.obj
```

Buradan elde edilen ürün "app.exe" dosyası olacaktır.

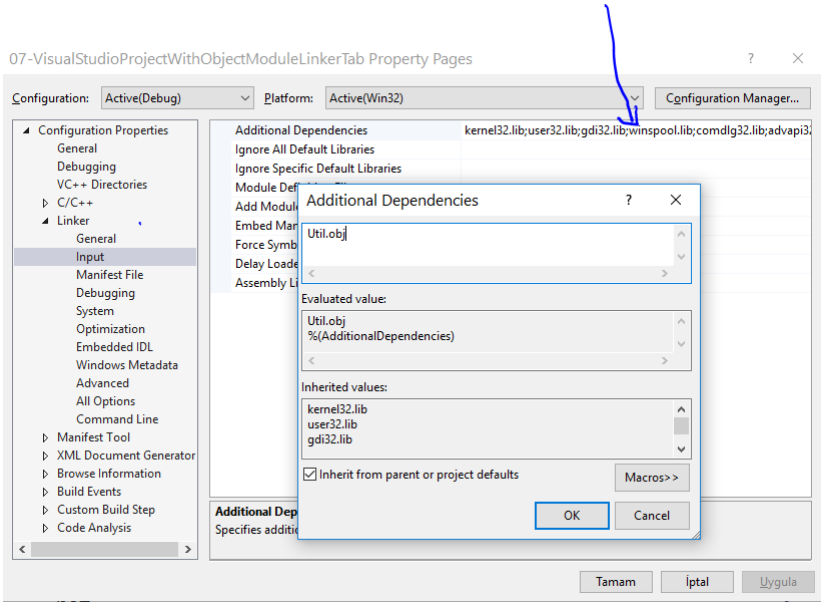
Pekiyi Visual Studio IDE'sinde oluşturulmuş olan bir C projesinde sembolik makine dilinde yazılan bir kod nasıl eklenir? Bunun üç yolu vardır:

1) Visual Studio IDE'sinde projeye bir ".obj" dosyası eklendiğinde (örneğin NASM ile derlenmiş bir dosya

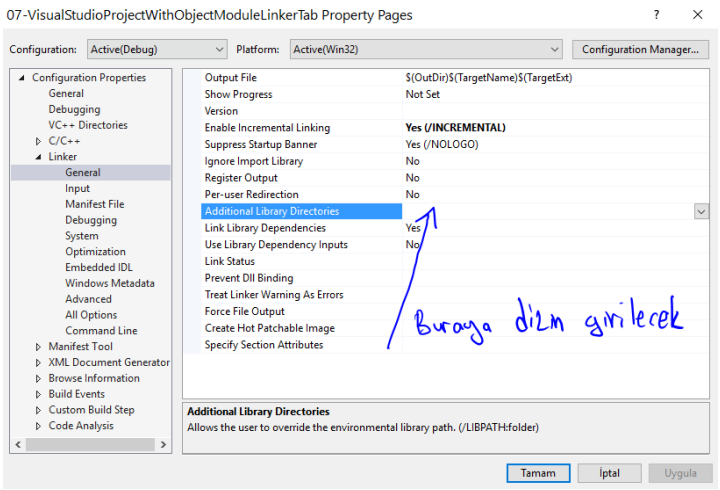
olabilir) zaten IDE bunu link işlemine dahil etmektedir. Böylece komut satırında NASM ile kod derlenip amaç dosya proje eklenebilir.



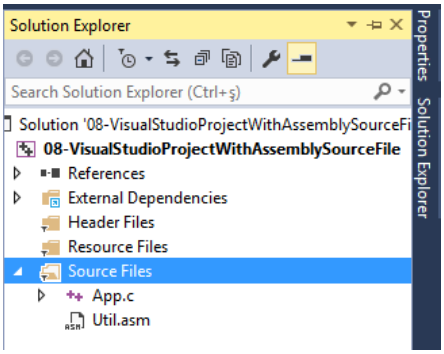
2) Amaç dosyayı proje eklemek yerine proje seçeneklerinden Linker/Input sekmesine gelinir ve "Additional Dependencies" kısmına amaç dosyanın yalnızca ismi (uzantı dahil fakat tam yol ifadesi değil) girilir.



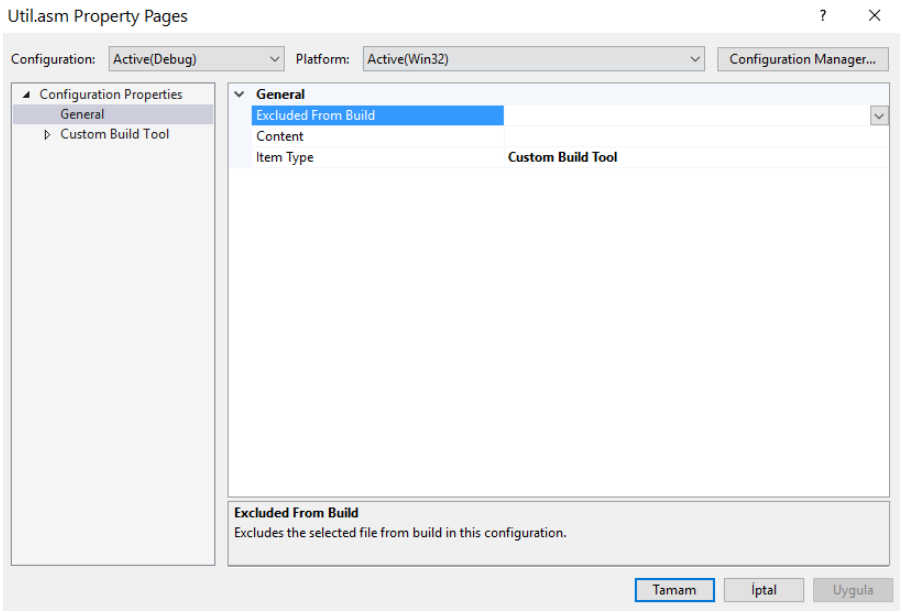
Burada önemli bir noktayı belirtmek istiyoruz: Eğer bu biçimde eklenecek olan amaç dosya başka bir dizindeyse bu sekmede yalnızca dosyanın ismi yazılmalıdır. Amaç dosyanın bulunduğu dizinin ise "Linker/Generel/Additional Library Directories" sekmesinde belirtilmesi gerekir.



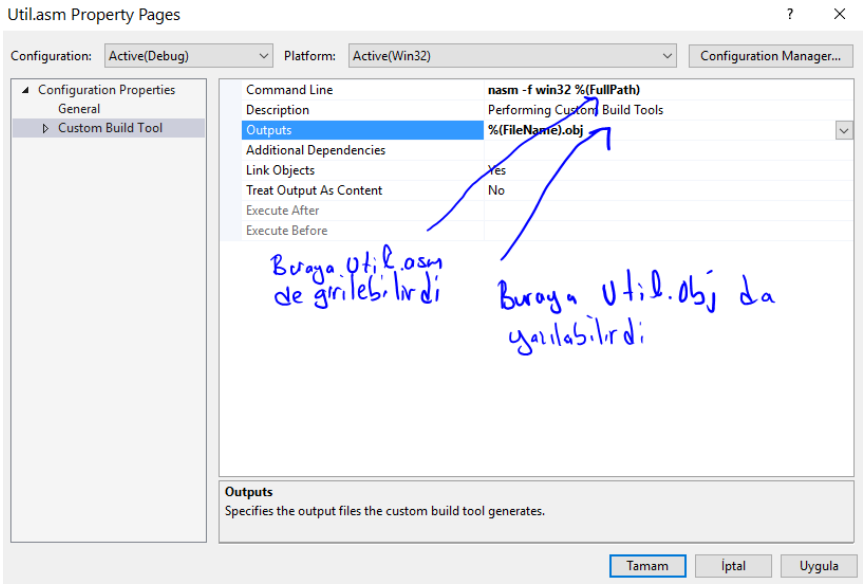
3) Bu seçenekte projeye ".obj" uzantılı amaç dosya değil, ".asm" uzantılı kaynak dosya eklenir.



Fakat Visual Studio IDE'si ".asm" uzantılı dosya için ne yapacağını bilmemektedir. Bunu ona anlatmak gerekir. İşte bu da iki aşamada yapılmaktadır. Birinci aşamada "Solution Explorer"da projedeki ".asm" dosyasının üzerine gelinir ve farenin sağ tuşu ile bağlam menüsünden "Properties" seçilir. Buradaki dialog penceresinde "Configuration Properties/General/Item Type" "Custom Build Tool" olarak girilir.



Artık "Custom Build Tool" seçeneği de dialog penceresinde görülür. İşte ikinci aşamada "Custom Build Tool" sekmesinde ilgili dosyanın hangi program tarafından derleneceği ve çıktısının ne olacağı belirtilir:



Her ne kadar örneğimizde zaten NASM "util.obj" dosyası üretecekse de Visual Studio bunu bilememektedir.

Sekmedeki “Outputs” girişi Visual Studio’nun hangi dosyayı link işlemine sokacağını belirtir. %(FullPath) işlem uygulanan dosyanın tam yol ifadesini %(FileName) ise yalnızca ismini (uzantı dahil değil) belirtmektedir.

Artık proje build edildiğinde bu “.asm” dosyası NASM ile derlenip çıktısı da link işlemine sokulacaktır. Bu üçüncü yöntemin en önemli avantajı doğrudan sembolik makine dili kaynak dosyasında değişiklik yapıp hemen programı build edebilmemizdir. Visual Studio IDE’si bizim kaynak dosya üzerinde değişilik yaptığımızı anlayarak onu yeniden NASM ile derleme işlemine sokup elde edilen amaç dosyayı link aşamasına dahil edip yeniden çalıştırılabilir dosya oluşturacaktır.

Şimdi aynı uygulamayı Linux’un komut satırında yapalım. NASM kaynak dosyası şöyle olacaktır:

```
; util.asm

[BITS 32]

section .text
    global add, multiply

 add:
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp + 8]
    add     eax, [ebp + 12]
    pop     ebp
    ret

 multiply:
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp + 8]
    mul     dword [ebp + 12]
    pop     ebp
    ret
```

Burada fonksiyon isimlerinin alt tire başlamadığına dikkat ediniz. gcc derleyicileri default durumda “cdecl” çağırma biçimininde fonksiyonun ismine alt tire eklemeden onu amaç dosyaya yazmaktadır. “app.c” dosyası yine aşağıdaki gibi olacaktır:

```
/* app.c */

#include <stdio.h>

int add(int a, int b);
int multiply(int a, int b);

int main(void)
{
    printf("%d\n", add(10, 20));
    printf("%d\n", multiply(10, 20));

    return 0;
}
```

Yukarıdaki util.asm Linux’ta aşağıdaki gibi derlenir:

```
nasm -felf32 util.asm
```

app.c programı da 32 bit olarak aşağıdaki gibi derlenmelidir:

```
gcc -c -m32 app.c
```

Linux'ta "app.o" ve "util.o" doğrudan ld linker'ıyla link edilebilir. Ancak "ld" Micsrosoft'un "link.exe" programının yaptığı gibi standart C kütüphanelerini ve başlangıç modüllerini (start up modules) link işlemine dahil etmemektedir. Linux'ta maalesef standart C kütüphanelerinin ve özellikle de başlangıç modüllerinin link aşamasına dahil edilmesi oldukça zahmetlidir. Bu nedenle "app.o" ve "util.o" dosyalarının doğrudan "ld" linker'ıyla link edilmesi yerine "gcc" ile link edilmesi daha pratiktir. Biz "gcc"ye yalnızca ".o" dosyalarını girdi olarak verdiğimizde "gcc" bunları standart C kütüphanelerini ve başlangıç modüllerini de dahil ederek link işlemine sokmaktadır. Bu biçimde "gcc" ile link işlemi şöyle yapılabilir:

```
gcc -m32 -o app app.o util.o
```

Anahtar Notlar: 64 bit Linux sistemlerinde 32 bit kütüphane dosyaları ve diğer modüller sistemimizde kurulu durumda olmayabilir. Bu durumda bağlama aşamasında sorun oluşacaktır. 32 bit kütüphane ve modüller C için "gcc-multilib" paketindei C++ için "g++-multilib" paketinde bulunmaktadır. Bu paketleri debian türevi sistemlerde aşağıdaki komutla yükeleyebilirsiniz:

```
sudo apt-get install gcc-multilib
sudo apt-get install g++-multilib
```

Anahtar Notlar: Biz yine de zahmete katlanarak link işlemini "ld" ile yapmak istersek bunu nasıl başarabiliriz? Öncelikle standart C fonksiyonlarının "libc" kütüphanesi içerisinde bulunduğunu söylemek gerekir. Bu kütüphanenin static ve dinamik biçimleri vardır. ld programında "-lc" seçeneği bu kütüphanenin link dahil edilmesini sağlayacaktır. Pekiyi başlangıç modüllerinin neler olduğu ve nerede bulunduğu nasıl tespit edilebilir? Öncelikle bunun sorunlu konu olduğunu belirtmek gerekir. Çünkü "gcc" ve "libc" versiyonları arasında başlangıç modülleri konusunda da çokça değişilik yapılmıştır ve gelecekte de yapılmaya devam edilebilecektir. İşte hangi başlangıç modüllerinin gerektiğini anlamak için "gcc" --verbose" seçeneği ile çalıştırılabilir. Örneğin:

```
gcc -m32 --verbose -o app app.o util.o
```

7.6.1. Sembolik Makine Dilinde Yazılan Fonksiyonların C 'den Çağrılmasına Çeşitli Örnekler

Bu bölümde C'den çağrılacak biçimde sembolik makine dilinde fonksiyon yazımına ilişkin çeşitli örnekler verilecektir. Örneklerdeki fonksiyon isimlerinin başına Windows'taki "cdecl" isim dekorasyonuna uygun olacak biçimde "_ (underscore)" karakteri gfetirilmiştir. Eğer bu örnekleri Lişnux sistemlerinde çalıştıracaksanız bu "_" karakterlerin. kaldırmalısınız. Aşağıdaki örneklerde sembolik makine dili kaynak dosyasının ismi "util.asm" ve çağrımın yapıldığı C dosyasının ismi de "app.c" biçimindedir. Bu dosyaları Windows'ta aşağıdaki gibi derleyip link edebilirsiniz:

```
nasm -fwin32 util.asm
cl -c app.c
link /out:app.exe app.obj util.obj
```

Linux'ta ise derleme ve link işlemi aşağıdaki gibi yapılabilir:

```
nasm -felf32 util.asm
gcc -c -m32 app.c
gcc -m32 -o app app.o util.o
```

Şimdi örneklere geçelim.

1) Bir yazının uzunluğunu bulan mystrlen isimli fonksiyonun yazımı: Bu örnekte nul karakter ('\0) görene kadar bir yazıdaki karakterlerin sayısını bulan ve onunla geri dönen mystrlen isimli fonksiyonu yazacağız. (Ancak burada özel olarak şunu belirtmek istiyoruz: Aslında yazıdaki karakterlerin sayısını hesaplama işlemi Intel işlemcilerindeki "string komutları" denilen bazı özel makine komutlarıyla daha etkin yapılabilmektedir. Fakat bu komutları henüz görmediğimiz için burada bu komutları kullanmayacağız.)

Fonksiyon şöyle yazılabilir:

```
; util.asm

[BITS 32]

SECTION .text
    global _mystrlen

_mystrlen:
    push    ebp
    mov     ebp, esp

    xor     ecx, ecx
    mov     eax, [ebp + 8]
.@2:
    mov     dl, [eax]
    test    dl, dl
    jz          .@1
    inc     ecx
    inc     eax
    jmp     .@2
.@1:
    mov     eax, ecx

    pop     ebp
    ret
```

Burada [EBP + 8]'den alınan parametre yazının başlangıç adresidir. Dolayısıyla yazının karakterlerine daha sonra [EAX] bellek operandıyla erişilmiştir. Nul karakter testi TEST komutuyla yapılmıştır. Anımsanacağı gibi TEST komutu operand'ların etkilenmediği AND işlemi yapar. Karakterlerin sayısı ECX yazmacında tutulmaktadır. Tabii aslında örneğimizde EAX ile ECX yazmaçlarını yer değiştirseydik en sondaki döngü çıkışındaki MOV makine komutunu elimine edebilirdik. Aynı kod tek jump içerecek biçimde (do-while optimizasyonu) ve parametreye erişmek için ESP uyazmacı kullanılarak şöyle de yazılabilirdi:

```
; util.asm

[BITS 32]

SECTION .text
    global _mystrlen

_mystrlen:

    mov     eax, -1
    mov     ecx, [esp + 4]
.@1:
    mov     dl, [ecx]
    inc     eax
    inc     ecx
    test    dl, dl
    jnz     .@1

    ret
```

C'den çağrım şöyle yapılabilir:

```
/* app.c */

#include <stdio.h>

unsigned mystrlen(const char *str);
```

```
int main(void)
{
    char *str = "ankara";
    unsigned len;

    len = mystrlen(str);
    printf("%u\n", len);

    return 0;
}
```

2) int türden bir dizinin en büyük elemanıyla geri dönen aşağıdaki fonksiyonu sembolik makine dilinde yazacak olalım:

```
int getmax(const int *array, int size);
```

Fonksiyonun sembolik makine dilindeki karşılığı şöyle olabilir:

```
; Util.asm

[BITS 32]

SECTION .text
   global _getmax

_getmax:
    push ebp
    mov     ebp, esp

    mov     ecx, [ebp + 12]
    mov     eax, [ebp + 8]
    dec     ecx

    mov     edx, [eax]
.@2:
    add     eax, 4
    dec     ecx
    jz          .@1
    cmp     edx, [eax]
    jg          .@2
    mov     edx, [eax]
    jmp     .@2
.@1:
    mov     eax, edx
    pop     ebp
    ret
```

Burada en büyük eleman EDX yazmacında tutulmuştur. EAX yazmacı adresleme için kullanılmıştır. (Tersi yapılsaydı aslında bir makine komutundan kazanırdık). Algoritmada önce ilk eleman en büyük kabul edilip EDX yazmacına yerleştirilmiş sonra dizinin her elemanıyla EDX'teki değer karşılaştırılmıştır. Duruma göre yer değiştirme yapılmıştır. ECX yazmacı döngü sayacını tutmaktadır. Döngü ECX'teki değer 0 olana kadar yinelenmektedir. Fonksiyon aşağıdaki gibi bir kodla test edilebilir:

```
/* app.c */

#include <stdio.h>

int getmax(const int *array, int size);

int main(void)
{
    int a[] = { 2, 6, 34, 239, 123, 54, 37, 76, 53, 11 };
    int max;
```

```
    max = getmax(a, 10);
    printf("%d\n", max);

    return 0;
}
```

3) double bir dizinin elemanlarının toplamına geri dönen aşağıdaki C prototipine sahip fonksiyonu sembolik makine dilinde yazmak isteyelim:

```
double gettotal(const double *array, int size);
```

Fonksiyon şöyle yazılabilir:

```
; util.asm

[BITS 32]

SECTION .text
   global _gettotal

_gettotal:
    push ebp
    mov      ebp, esp

    mov      eax, [ebp + 8]
    mov      ecx, [ebp + 12]
    test     ecx, ecx
    jz           .@1
    fldz
.@2:
    fadd     qword [eax]
    add      eax, 8
    dec      ecx
    jnz      .@2
.@1:
    pop      ebp
    ret
```

Burada önce matematik işlemcinin stack'ine sıfır değeri push edilmiştir. Sonra dizini her elemanı fadd komutu kulanılarak ST(0) ile toplanmıştır. Böylece toplanan değerler her zaman ST(0)'da kalmaktadır. Gerçek sayı türünden geri dönüş değerlerinin matematik işlemcinin stack'inde (ST(0)'da) bırakıldığını anımsayınız. Fonksiyonu aşağıdaki gibi kodla test edebiliriz:

```
/* app.c */

#include <stdio.h>

double gettotal(const double *array, int size);

int main(void)
{
    double a[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 1 };
    double result;

    result = gettotal(a, 6);
    printf("%f\n", result);

    return 0;
}
```

4) Şimdi de int bir diziyi kabarcık sıralaması (boubble sort) yöntemiyle sıraya dizen aşağıdaki prototipe sahip bsort isimli fonksiyonu yazmak isteyelim:

```
void bsort(int *array, int size);
```

Fonksiyon şöyle yazılabilir:

```
; util.asm

[BITS 32]

SECTION .text
   global _bsort

_bsort:
    push    ebp
    mov     ebp, esp
    push    ebx

    xor     ebx, ebx
    jmp     .@2
.@1:
    mov     eax, [ebp + 8]
    xor     ecx, ecx
    jmp     .@4
.@3:
    mov     edx, [eax]
    cmp     edx, [eax + 4 ]
    jle     .@5
    xchg    edx, [eax + 4]
    mov     [eax], edx
.@5:
    add     eax, 4
    inc     ecx
.@4:
    mov     edx, [ebp + 12]
    dec     edx
    sub     edx, ebx
    cmp     ecx, edx
    jl          .@3
    inc     ebx
.@2:
    mov     edx, [ebp + 12]
    dec     edx
    cmp     ebx, edx
    jl          .@1

    pop     ebx
    pop     ebp
    ret
```

Buradaki algoritma aşağıdaki C kodundaki gibidir:

```
for (ebx = 0; ebx < size – 1; ++ebx)
    for (ecx = 0; ecx < size – 1 – ebx; ++ecx)
        if (a[ecx] > a[ecx + 1) {
            edx = ecx;
       xchg(edx, a[ecx + 1]);
       a[ecx + 1] = edx;
     }
```

Sembolik makine dilinde yan yana dizi elemanlarına [EAX] ve [EAX + 4] gibi bellek operandlarıyla erişildiğine dikkat ediniz. Bunun yerine alternatif olarak köşeli parantez içerisinde çarpansal faktör de kullanabilirdik:

```
; util.asm
```

```
[BITS 32]

SECTION .text
   global _bsort

_bsort:
   push      ebp
     mov     ebp, esp
     push    ebx

     xor     ebx, ebx
     jmp     .@2
     mov     eax, [ebp + 8]
.@1:
     xor     ecx, ecx
     jmp     .@4
.@3:
     mov     edx, [eax + ecx * 4]
     cmp     edx, [eax + 4 + ecx * 4 ]
     jle     .@5
     xchg    edx, [eax + 4 + ecx * 4]
     mov     [eax + ecx * 4], edx
.@5:
     inc     ecx
.@4:
     mov     edx, [ebp + 12]
     dec     edx
     sub     edx, ebx
     cmp     ecx, edx
     jl           .@3
     inc     ebx
.@2:
     mov     edx, [ebp + 12]
     dec     edx
     cmp     ebx, edx
     jl           .@1

     pop     ebx
     pop     ebp
     ret
```

Test kodu da şöyle olabilir:

```
/* app.c */

#include <stdio.h>

void bsort(int *array, int size);

int main(void)
{
     int a[10] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
     double result;
     int i;

     bsort(a, 10);

     for (i = 0; i < 10; ++i)
          printf("%d ", a[i]);
     printf("\n");

     return 0;
}
```

5) int türden iki adresi alarak oradaki değerleri yer değiştiren aşağıda prototipi verilmiş swap fonksiyonunu yazmaya çalışalım:

```
void swap(int *a, int *b);
```

Bu fonksiyon şöyle yazılabilir:

```
; util.asm

[BITS 32]

SECTION .text
   global _swap

_swap:
    push    ebp
    mov     ebp, esp

    mov     eax, [ebp + 8]
    mov     edx, [eax]
    mov     ecx, [ebp + 12]
    xchg    edx, [ecx]
    mov     [eax], edx

    pop     ebp
    ret
```

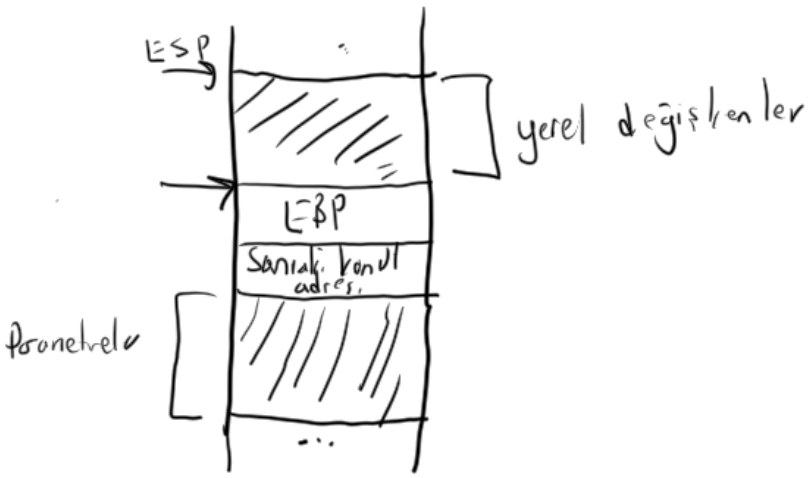
Fonksiyon aşağıdaki kodla test edilebilir:

```
/* app.c */

#include <stdio.h>

void swap(int *a, int *b);

int main(void)
{
    int a = 10, b = 20;

    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

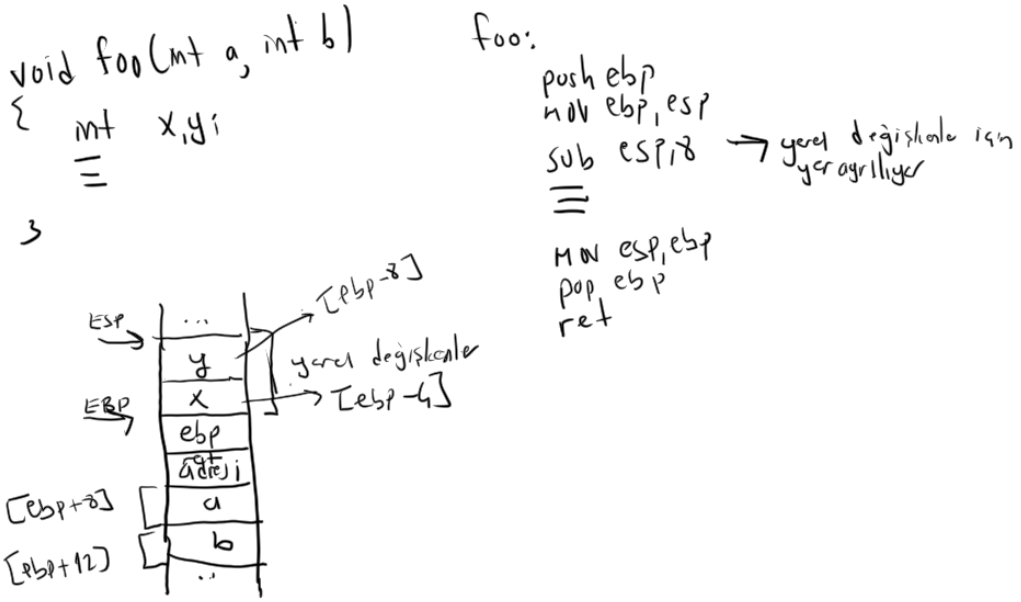
7.7. Yerel Değişkenlerin Kullanımı

Diğer kurslarda çeşitli konular içerisinde programlama dillerindeki yerel değişkenlerin "stack" üzerinde yaratıldığından söz etmiştik. Yine bu kurslarda bunların yaratılmasının ve yok edilmesinin çok hızlı bir biçimde yapıldığını belirtmiştik. Yerel değişkenlerin faaliyet alanlarının ilgili blokla sınırlı olduğunu biliyorsunuz. Yine anımsanacağı gibi yerel değişkenler programın akışı onların bildirildikleri noktaya geldiğinde yaratılıyor, akış onların bildirildiği bloğun sonuna geldiğinde yok ediliyordu. Pekiyi bütün bunlar nasıl gerçekleştirilmektedir?..

Yerel değişkenler için yer EBP yazmacı fonksiyon tarafından ayarladıktan sonra ESP yazmacının eksiltilmesiyle ayrılmaktadır. Böylece nasıl parametre değişkenlerine [EBP + N] bellek operandıyla erişiliyorsa yerel değişkenlere de [EBP - N] bellek operandıyla erişilir. Yerel değişken kullanan fonksiyonların stack çerçeveleri (stack frames) aşağıdaki şekille gösterilebilir:



Daha somut bir örnek verebiliriz:



Görüldüğü gibi yerel değişkenler için yer ayrılması aslında SUB ESP, N gibi tek bir makine komutuyla yapılmaktadır. Yerel değişkenler MOV ESP, EBP komutuyla aslında ESP yazmacının eski durumuna getirilmesi yolıuyla (ya da ESP'nin artırılması yoluyla) yok edilirler.

Derleyiciler genellikle yerel değişkenler için stack'te hizalı (align edilmiş) ve ardışıl yerler ayırırlar. Fakat hangi yerel değişkenin diğerlerine göre nerede olacağı derleyiciden derleyiciye değişebilmektedir. Bazı derleyiciler ilk tanımlanan yerel değişken yüksek adreste olacak biçimde (yani EBP'nin hemen yukarısında olacak biçimde) bir düzenleme yaparken bazıları bunun tam tersi düzenleme yapabilmektedir. (Tabii bilindiği gibi C ve C++ standartlarında yerel değişkenlerin ardışıllığı hakkında ya da bunların birbirlerine göre durumları hakkında zaten bir belirlemede bulunulmamıştır.) Microsoft ve gcc derleyicileri tanımlama sırasına göre yerel değişkenler için düşük adresten yüksek adrese doğru (yani işlk tanımlanan değişken düşük adreste bulunacak biçimde) yer ayırmaktadır.

Örneğin aşağıdaki gibi yerel değişken kullanan bir C fonksiyonuınu sembolik makine dilinde yazmaya çalışalım:

```
int foo(int a, int b)
{
    int result;

    result = a + b;
```

```
    return result;
}
```

Fonksiyonu şöyle yazabiliriz:

```
[BITS 32]

SECTION .text
    global _foo

_foo:
    push    ebp
    mov     ebp, esp
    sub     esp, 4              ; result için yer ayrılıyor

    mov     eax, [ebp + 8]
    add     eax, [ebp + 12]     ; a + b
    mov     [ebp - 4], eax      ; result = a + b
    mov     eax, [ebp - 4]      ; return result

    mov     esp, ebp            ; result yok ediliyor
    pop     ebp
    ret
```

Şüphesiz yukarıdaki kodu daha optimize bir biçimde yazabilşirdik. (Örneğin aslında bu kodda yerel değişken kullanmaya bile gerek yoktur.) Fakat biz burada hiç optimizasyon yapmadan ilgili C fonksiyonunun sembolik makine dilindeki tam karşılığını yazmaya çalıştık.

Test kodu da şöyle olabilir:

```
/* app.c */

#include <stdio.h>

int foo(int a, int b);

int main(void)
{
    int result;

    result = foo(10, 20);
    printf("%d\n", result);

    return 0;
}
```

Şimdi de aşağıdaki gibi bir fonksiyonu bire bir hiç optimizasyon yapmadan sembolik makine dilinde yazmaya çalışalım:

```
int gettotal(const int *array, int size)
{
    int total = 0;
    int i;

    for (i = 0; i < size; ++i)
        total += array[i];

    return total;
}
```

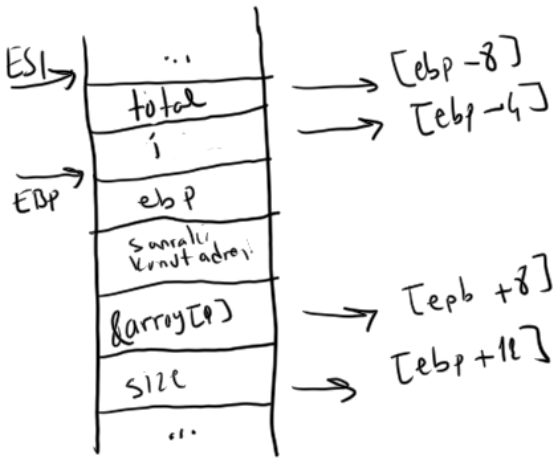
Fonksiyon şöyle yazılabilir:

```
[BITS 32]

SECTION .text
    global _gettotal

_gettotal:
    push ebp
    mov      ebp, esp
    sub      esp, 8

    mov      ecx, [ebp + 8]                ; ecx = array
    mov      dword [ebp - 8], 0            ; total = 0
    mov      dword [ebp - 4], 0;           ; i = 0;

.@2:
    mov      eax, [ebp -4]
    cmp      eax, [ebp + 12]               ; i < size
    jge      .@1

    mov      eax, [ebp - 8]
    mov      edx, [ebp - 4]                ; edx = i
    add      eax, [ecx + edx * 4]          ; array[i]
    mov      [ebp - 8], eax                ; total += array[i]

    inc      dword [ebp - 4]               ; ++i
    jmp      .@2

.@1:
    mov      eax, [ebp - 8]

    mov      esp, ebp
    pop      ebp
    ret
```

Test şu kodla yapılabilir:

```
#include <stdio.h>

int gettotal(const int *array, int size);

int main(void)
{
    int result;
    int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    result = gettotal(a, 10);
    printf("%d\n", result);

    return 0;
}
```

gettotal fonksiyonundaki stack çerçevesi aşağıdaki gibi oluşturulmuştur:



C'de henüz değer atanmamış yerel değişlerin içerisinde çöp değerler bulunduğunu anımsayınız. Pekiyi bunun sembolik makine dilindeki anlamı ne olabilir? İşte yerel değişkenler için stack'te yer ayrıldığında orada rastgele değerler bulunmaktadır. Ayrıca bir noktaya daha dikkatiniz çekmek istiyoruz. C'de biz bir yerel değişkene aşağıdaki gibi ilkdeğer vermiş olalım:

```
int a = 10;
```

Bu ilkdeğer verme işlemi derleme aşamasında yapılamamaktadır. Yerel değişkenlere verilen ilk değerler ancak MOV makine komutuyla gerçekleştirilebilmektedir. Halbuki global değişkenlere verilen ilkdeğerler programın derleme aşamasında bu değişkenlere yerleştirilebilmektedir.

Şimdi aşağıdaki gibi bir fonksiyonu sembolik makine dilinde yazmaya çalışalım:

```
void foo(void)
{
    int a[10];
    int i;

    for (i = 0; i < 10; ++i)
        a[i] = i;
    ....
}
```

Bu kodun optimize edilmemiş bire bir sembolik makine dili karşılığı şöyle yazılabilir:

```
_foo:
    push ebp
    mov     ebp, esp
    sub     esp, 44

    mov     dword [ebp - 4], 0        ; i = 0
.@2:
    mov     eax, [ebp - 4]
    cmp     eax, 10                   ; if (i < 10)
    jge     .@1
    mov     eax, [ebp - 4]
    lea     edx, [ebp - 44]
    mov     [edx + eax * 4], eax      ; a[i] = i
    inc     eax
    mov     [ebp - 4], eax
    jmp     .@2
.@1:
    ; .....
    mov     esp, ebp
    pop     ebp
    ret
```

Burada dizinin ilk elemanı EBP – 44 adresindedir. LEA komutu köşeli parantezin içerisindeki offset değerini elde etmektedir. Böylece aşağıdaki komutla dizinin başlangıç adresi EDX yazmacına çekilmiş olur:

```
lea     edx, [ebp - 44]
```

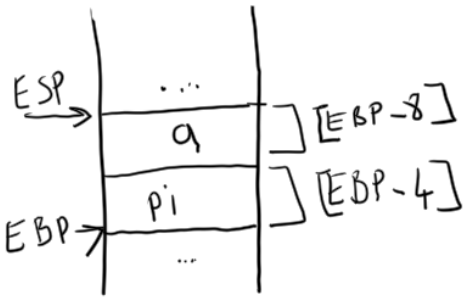
i değişkenine ise [ebp – 4] bellek operandıyla erişilebilir.

7.7.1. Yerel Nesnelerin Adreslerinin Elde Edilmesi

Aşağıdaki gibi bir C kodu söz konusu olsun:

```
void foo(void)
{
   int a = 10;
   int *pi;

   pi = &a;
   ...
}
```

Burada a ve pi olmak üzere iki yerel değişken vardır. a'ya [EBP – 8] bellek operandıyla pi'ye de [EBP – 4] bellek operandıyla erişiriz.



Görüldüğü gibi burada aslında a'nın adresi EBP – 8 değeridir. Bizim de yapmamız gereken şey bu değeri [EBP – 4] bellek operandına yerleştirmektir. Şüphesiz EBP – 8 değeri EBP'nin kendisini bozmadan aşağıdaki gibi elde edilebilir:

```
mov     eax, ebp
sub     eax, 8
```

Fakat bu iki komut yerine LEA komutu ile bu işlem tek hamlede yapılabilmektedir:

```
lea     eax, [ebp - 8]
```

LEA komutunun köşeli parantez içerisindeki değeri elde ettiğini anımsayınız. O halde yukarıdaki C işlemin sembolik makine dili karşılığı aşağıdaki gibi olabilir:

```
_foo:
    push ebp
    mov     ebp, esp
    sub     esp, 8

    mov     dword [ebp - 8], 10          ; a = 10
    lea     eax, [ebp - 8]               ; eax = &a
    mov     [ebp - 4], eax               ; pi = eax
    ; .....

    mov     esp, ebp
```

```
    pop     ebp
    ret
```

O halde yerel nesnelerin adreslerini almak için aklımıza LEA makine komutu gelmelidir. LEA komutunu adeta C'deki & (address of) operatörüne benzetebiliriz. Tabii global nesnelerin adreslerini almak için LEA komutuna gerek yoktur. Çünkü global nesneler birer etiket ile bildirilebilirler. Etiketler de birer adres belirtmektedir.

Pekiyi yerel bir dizinin adresini nasıl elde edebiliriz? Örneğin aşağıdaki C kodunun sembolik makine dili karşılığını yazmak isteyelim:

```
void foo(void)
{
   int a[10];
   int *pi;

   pi = &a;
   ...
}
```

Bu kod eşdeğer olarak sembolik makine dilinde şöyle ifade edilebilir:

```
_foo:
    push    ebp
    mov     ebp, esp
    sub     esp, 44

    lea     eax, [ebp - 44]             ; eax = a
    mov     [ebp - 4], eax              ; pi = eax
    ; .....

    mov     esp, ebp
    pop     ebp
    ret
```

Burada 32 bit sistemlerde 10 elemanlı int türden dizi bellekte 40 byte, int türünden gösterici de 4 byte yer kaplar. İlk bildirilen yerel nesne düşük adrese yerleştirildiğinde dizi EBP - 44', int türden gösterici de [EBP – 4] adresinde bulunacaktır.

C'de bir diziye (yapıya ve birliğe de) küme parantezleri içerisinde ilkdeğer verildiğini anımsayınız. Pekiyi aşağıdaki gibi yerel bir dizi bildirmi sembolik makine diliyle nasıl oluşturulmaktadır?

```
void foo(void)
{
   int a[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
   ...
}
```

Burada derleyici tipik olarak dizi elemanlarına tek tek MOV komutuyla değer atar. Örneğin:

```
_foo:
    push    ebp
    mov     ebp, esp
    sub     esp, 40

    mov     dword [ebp-40], 10
    mov     dword [ebp-36], 20
    mov     dword [ebp-32], 30
    mov     dword [ebp-28], 40
    mov     dword [ebp-24], 50
```

```
    mov     dword [ebp-20], 60
    mov     dword [ebp-16], 70
    mov     dword [ebp-12], 80
    mov     dword [ebp-8], 90
    mov     dword [ebp-4], 100
    ; ....

    mov     esp, ebp
    pop     ebp
    ret
```

Pekiyi ya ilkdeğer verilen elemanların sayısı fazla olursa ne olur? Yine derleyici MOV komutlarıyla ilkdeğerleri dizi elemanlarına tek te mi atar? Örneğin:

```
void foo(void)
{
   int a[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
                110, 120, 130, 140, 150, 160, 170, 180, 190, 200 };
   ...
}
```

İşte derleyiciler bu tür durumlarda kodu optimize etme eğilimindedirler. Genellikle ilkdeğer olarak verilen bu sabit değerleri .rdata ya da .rodata gibi const bir bölüme (section) yerleştirip buradan kopyalama yaparlar. Bu durumda bu ilkdeğer verme işlemi de sembolik makine dilinde aşağıdakine benzer bir biçimde olacaktır:

```
array_values:
    dd  10
    dd  20
    dd  30
    dd  40
    dd  50
    dd  60
    dd  70
    dd  80
    dd  90
    dd  100
    dd  110
    dd  120
    dd  130
    dd  140
    dd  150
    dd  160
    dd  170
    dd  180
    dd  190
    dd  200
;   ....

_foo:
    push ebp
    mov     ebp, esp
    sub     esp, 80

    ; Burada array_values adresinden EBP - 80 adresine 20 int değeri kopyalayan bir kod olacak

    mov     esp, ebp
    pop     ebp
    ret
```

Burada kopyalama işlemi henüz görmediğimiz string komutlarıyla etkin bir biçimde yapılabilmektedir. Ya da bazı derleyiciler bunun için "başlangıç modüllerinde ("start up modules) bulundurdukları fonksiyonları da kullanabilmektedir. Fakat hangi yöntem söz konusu olursa olsun dizi elemanlarına verilen bu ilkdeğerlerin bir biçimde programda yer kapladığına dikkat ediniz. İlkdeğer verme işlemi MOV komutlarıyla yapıldığında bu ilkdeğerler komutun bir parçası biçiminde ".text" bölümünde, kopyalama

tekniği kullanıldığında ise ".rdata" ya da ".rodata" gibi bir bölümde bulunacaktır. O halde derleyiciler hangi yöntemi kullanacaklarına n tane elemanı diziye yerleştirmek için gereken MOV makine komutlarının uzunluğu ile kopyalama kodunun uzunluğuna bakarak karar verebilirler.

7.8. C'deki Yapıların Sembolik Makine Dilindeki Karşılıkları

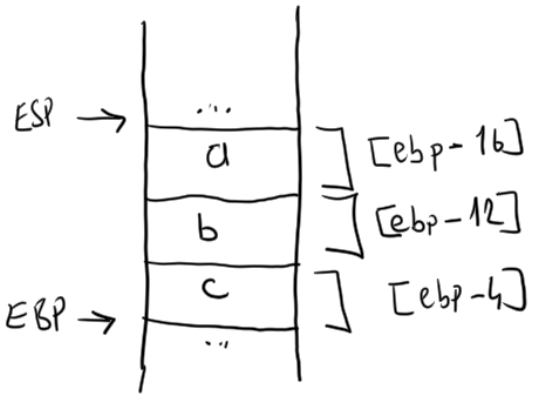
Bilindiği gibi C'de dizilerle yapılar birbirlerine oldukça benzemektedir. (C standartlarında bunlara topluca "aggregate" denilmektedir.) Diziler "elemanlarını aynı türden olan ve bellekte ardışıl bir biçimde bulunan" veri yapılarıdır. Yapılar ise "elemanları farklı türlerden olabilen ve bellekte ardışıl biçimde bulunan veri yapılarıdır. Görüldüğü gibi her iki veri yapısında da "ardışıllık" durumu vardır. Ayrıca C'de dizi ve yapılarda ilk eleman düşü adreste bulunacak biçimde bir yerleşim öngörülmüştür. Bu durumda yerel bir yapı nesnesinin elemanları bellekte peşi sıra bulunan ayrı ayrı nesneler gibi düşünülebilir. Örneğin aşağıdaki gibi bir C kodu bulunuyor olsun:

```
struct SAMPLE {
    int a;
    double b;
    int c;
};

void foo(void)
{
    struct SAMPLE s;

    s.a = 10;
    s.b = 0;
    s.c = 20;
    /* ... */
}
```

Bu kodun sembolik makine dili karşılığı şöyle olabilir:

```
_foo:
    push ebp
    mov     ebp, esp
    sub     esp, 16

    mov     dword [ebp - 16], 10      ; s.a = 10
    mov     dword [ebp - 12], 0       ; s.b = 0 (düşük dword)
    mov     dword [ebp - 8], 0        ; s.b = 0 (yüksek dword)
    mov     dword [ebp - 4], 20       ; s.c = 20
    ; ...
    mov     esp, ebp
    pop     ebp
    ret
```

foo fonksiyonunun stack çerçevesini aşağıdaki şekille gösterebiliriz:



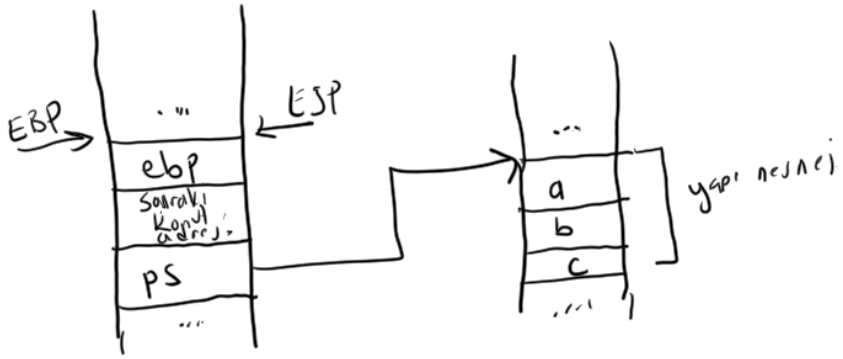
Bilindiği gibi C'de yapılar fonksiyonlara genellikle adres yoluyla aktarılmaktadır. Aşağıdaki gibi bir C kodu bulunuyor olsun:

```
struct SAMPLE {
    int a;
    double b;
    int c;
};

void foo(struct SAMPLE *ps)
{
    ps->a = 10;
    ps->b = 0;
    ps->c = 20;
    /* ... */
}
```

Bu kodun sembolik makine dili karşılığı da şöyle oluşturulabilir:

```
_foo:
    push    ebp
    mov     ebp, esp

    mov     eax, [ebp + 8]          ; yapı nesnesinin adresi eax'te

    mov     dword [eax], 10         ; ps->a = 10
    mov     dword [eax + 4], 0      ; ps->b = 0 (düşük dword)
    mov     dword [eax + 8], 0      ; ps->b = 0 (yüksek dword)
    mov     dword [eax + 12], 20    ; ps->c = 20

    ; ...

    mov     esp, ebp
    pop     ebp
    ret
```

Burada [ebp + 8]'den çekilen parametre yapı nesnesinin adresidir. Bu adres EAX yazmacına yerleştirilmiş ve sonra yapı elemanlarına [EAX + 0], [EAX + 4], [EAX + 12] bellek operandlarıyla erişilmiştir.



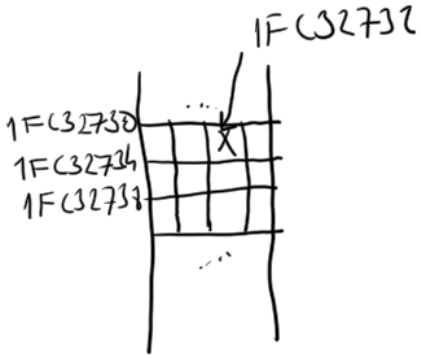
7.9. Yapı Elemanlarının Hizalanması (Alignment)

Yapı elemanlarının ve yerel nesnelerin belli değerlerin katlarına yerleştirilmesi durumuna hizalama (alignment) denilmektedir. Pek çok işlemci bellekte belli değerlerin katlarına daha hızlı bir biçimde erişmektedir. Örneğin 32 bit Intel işlemcilerinde işlemci ile bellek arasındaki bağlantı nedeniyle 4 byte'lık değerlere eğer onlar bellekte 4'ün katlarındaysa (dword alignment) daha hızlı erişmektedir. Bunun nedeni bazı ayrıntılar göz ardı edilerek şöyle açıklanabilir: 32 bit Intel işlemcilerinde RAM ile CPU arasındaki adres yolu 32 değil 30 yolludur ve CPU tek hamlede bellekten her zaman 4 byte'lık bilgiyi çekmektedir. Bellekte erişilecek bilgi 4 byte olmasa bile işlemci önce o bilginin bulunduğu 4'byte'ın katından itibaren 4

byte'ı çeker, o 4 byte'ın içerisinden ilgili kısmı alır. Örneğin aşağıdaki gibi bir makine komutu düşünelim:

```
mov al, [0x1FC32732]
```

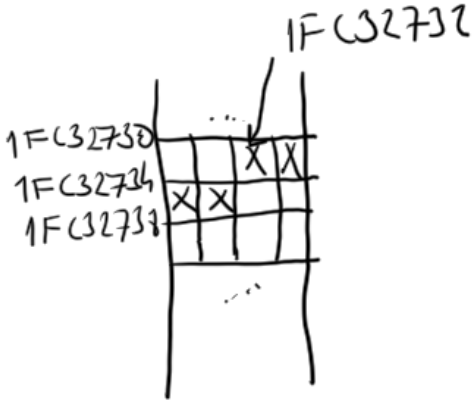
Burada 32 bit Intel işlemcisi RAM'in 4'ün katı olan 0x1FC32730 adresinden itibaren 4 byte'ını çekip onun içerinden ilgili kısmı AL yazmacına yerleştirmektedir. Tabii bu işlem MOV komutunun kendi içerisinde yani tek bir koımut ile gerçekleşmektedir.



Şimdi biz 32 bit Intel işlemcilerinde bellekte 4'ün katlarına hizalanmamış 4 byte'lık bir bilgiye erişmek isteyelim. Örneğin:

```
mov eax, [0x1FC32732]
```

Burada 4'ün katlarına hizalanmamış 4 byte'lık bir erişim söz konusudur. İşte her ne kadar bu erişim tek bir makine komutuyla yapılıyorsa da bu makine komutu iki kez RAM'e eriştiğinden dolayı göreli olarak biraz daha yavaş çalışmaktadır:



Yukarıdaki örnekte işlemci önce 1FC32730 adresinden 4 byte'ı çeker onun yüksek anlamlı 2 byte'ını alır, sonra 1FC32734 adresinden 4 byte'ı çekerek onun düşük anlamlı 2 bte'ını alıp birleştirdiktren sonra EAX yazmacına yerleştirir. Çok çekirdekli sistemlerde bu iki RAM erişimi sırasında RAM (yani "bus") serbest bırakılmaktadır. Bu da başka bir çekirdeğin o anda aynı bellek bölgesine erişmesi durumunda geçersiz bir değerin oluşmasına yol açabilir. İşte bunu engellemek iç.in Intel işlemcilerinde LOCK isimli bir komut öneki bulundurulmuştur:

```
lock mov eax, [0x1FC32732]
```

İşte bu nedenle 4 byte'lık nesnelerin bellekte 4'ün katlarına yerleştirilmesi hız kazancı sağlayabilmektedir. Örneğin:

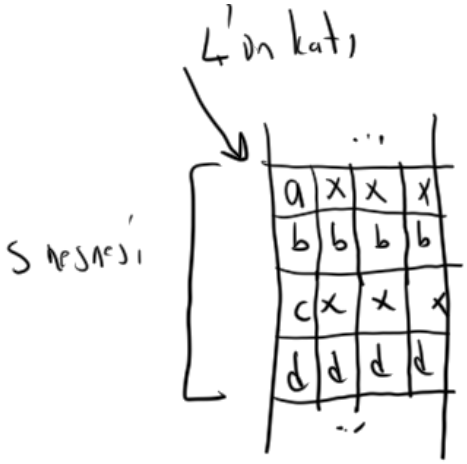
```
void foo(void)
{
    char a;
```

```
    int b;
    ...
}
```

Derleyici burada a'yı 4'ün katına yerleştirdiktem sonra arada 3 byte boşluk bırakarak b'nin 4'ün katında bulunmasını sağlayabilir. (Tabii yerel değişkenlerde bir ardışıllık garanti edilmediği için derleyici yerleşimi ters de yapabilirdi). Hizalama yapılar söz konusu olduğunda daha önemli hale gelebilmektedir. Çünkü standartlara göre C'de yapı elemanları ilk bildirilen eleman düşük adreste olacak biçimde ardışıl bulunmak zorundadır. Ancak derleyici elemanlar arasında hizalama amaçlı ekstra boşluklar bırakabilir. Bu ekstra boşluklar derleyici tarafından yerleştirildiği için ardışıllığı bozmazlar. Örneğin:

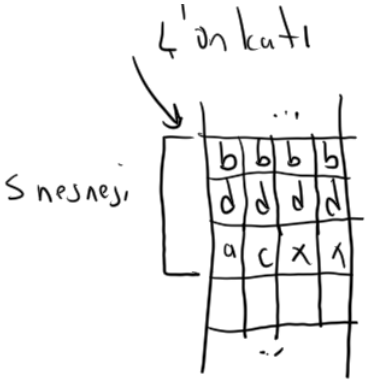
```
struct SAMPLE {
    char a;
    int b;
    char c;
    int d;
};

struct SAMPLE s;
```

Şimdi derleyici s nesnesini 4'ün katına yerleştirecektir. Böylece s'in a parçası da 4'ün katında bulunacaktır. Ancak derleyici bu a parçasından sonra hemen yapının b parçasını yerleştirirse b parçası hizalanmamış olur. Bu yüzden derleyici a'dan sonra 3 byte boşluk bırakarak b'yi yerleştrebilir. Bu durumda c de 4'ün katında olacaktır. Ondan sonra yine 3 byte boşluk bırakarak d'yi yerleştirecektir. Toplamda s nesnesinin kendisi 16 byte yer kaplamış olacaktır:



Biz yapı elemanlarınn yerlerini değiştirerek yapı nesnesinin daha az yer kaplamasını sağlayabiliriz:

```
struct SAMPLE {
    int b;
    int d;
    char a;
    char c;
};

struct SAMPLE s;
```



1 byte'lık nesnelerin bir hizalanma gereksiniminin olmadığına dikkat ediniz. Ancak 2 byte'lık nesneler için bir hizalama gereksinimi söz konusu olabilir. Intel işlemcilerinde 8 byte'lık double değerlerin 8'in katlarında bulunması da benzer biçimde hız kazancı sağlamaktadır.

Hizalama genellikle belli değerlerin katlarına göre isimlendirilmektedir:

- Byte Hizalaması (1 Byte Hizalama): Burada nesneler ve yapı elemanları 1'in katlarında olacak biçimde yerleştirilir. Başka bir deyişle nesneler ve yapı elemanları hizalanmaz yani aralarına boşluk bırakılmaz. Nesnelerin ve veri elemanlarının 1'in katlarına hizalanmasının aslında bir hizalama olmadığına dikkat ediniz.

- Word Hizalaması (2 Byte Hizalama): Burada 2 byte ya da 2 byte'tan büyük nesneler ve yapı elemanları 2'nin katlarına hizalanır. 1 byte'lık nesneler ve yapı elemanları 1'in katlarına hizalanır.

- DWord Hizalaması (4 byte'lık Hizalama): Burada 4 byte ya da 4 byte'tan büyük nesneler ve yapı elemanları 4'ün katlarına, 2 byte'lık nesneler ve yapı elemanları 2'nin katlarına, 1 byte'lık nesneler ve yapı elemanları da 1'in katlarına hizalanır.

- Burada 8 byte ya da 8 byte'tan büyük nesneler ve yapı elemanları 8'in katlarına, 4 byte'lık nesneler ve yapı elemanları 4'ün katlarına, 2 byte'lık nesneler ve yapı elemanları 2'nin katlarına, 1 byte'lık nesneler ve yapı elemanları da 1'in katlarına hizalanır.

- Paragraph Hizalaması (16 byte hizalama): Burada 16 byte ya da 16 byte'tan büyük nesneler ve yapı elemanları 16'nın katlarına, 8 byte ya da 8 byte'tan büyük nesneler ve yapı elemanları 8'in katlarına, 4 byte'lık nesneler ve yapı elemanları 4'ün katlarına, 2 byte'lık nesneler ve yapı elemanları 2'nin katlarına, 1 byte'lık nesneler ve yapı elemanları da 1'in katlarına hizalanır.

Yukarıda açıkladığımız n byte'lık hizalama kurallarını aslında hepsini içerecek biçimde tek bir cümleyle de özetleyebiliriz: "n byte'lık hizalamada n'den daha küçük olan elemanları kendi katlarına, n ve n'den büyük olanları n'in katlarına hizalanırlar".

Yerel ve global nesnelerin ve yapı elemanlarının hizalanması pek çok C derleyicisinde derleyici ayarlarıyla, özel pragma direktifleriyle ya da anahtar sözcüklerle kontrol altında bulundurulabilmektedir. Örneğin Microsoft derleyicilerinde /Zpn seçeneği ile (burada n bir sayı belirtmelidir) yapı elemanları için hizalama biçimi ayarlanabilmektedir. (Bu işlem Visual Studio IDE'sinde proje seçenklerine gelinip "C/C++/Code Generation/Struct Member Alignment" menüsüyle de yapılabilir.) gcc derleyicilerinde de benzer biçimde yapı elemanları için hizalama ayarlaması komut satırından "-fpack-struct=n" seçeneği ile yapılmaktadır. Microsoft ve gcc derleyicilerinde default hizalama durumu QWord (8 byte) hizalamasıdır.

Yapı hizalamaları bütünsel olarak değil de belli bir kod bölgesini etkileyecek biçimde de "#pragma pack" önişlemci direktifleriyle yapılabilmektedir. Örneğin:

```
#include <stdio.h>
```

```
#pragma pack(1)

struct SAMPLE1 {
    char a;
    int b;
};

#pragma pack(4)

struct SAMPLE2 {
    char a;
    int b;
};

int main(void)
{
    printf("%u\n", sizeof(struct SAMPLE1));
    printf("%u\n", sizeof(struct SAMPLE2));

    return 0;
}
```

Bir pragma pack komutu diğerine kadar etki göstermektedir.

Son olarak C11 ile "_Alignas" ve C++11 ile de "alignas" anahtar sözcüklerinin bildirimde belirtilen nesnelerin hizalanması için C ve C++ dillerine eklendiğini anımsatalım. Bunların dışında ayrıca çeşitli derleyicilerde hizalama için eklenti biçiminde başka anahtar sözcükler de bulunabilmektedir.

7.10. Gerçek Sayı Türlerine İlişkin Nesnelere Değer Atanması

4, 8 ve 10 byte uzunluktaki (yani C'deki float, double ve long double türleri) nesnelere değer atanması sembolik makine dilinde nasıl yapılmaktadır? Öncelikle 32 bit işlemcilerde MOV işleminin en fazla 32 bit olabildiğini anımsatmak istiyoruz. Ancak daha önceden de gördüğümüz gibi gerçek sayı işlemlerinde 4 byte'lık, 8 byte'lık ve 10'lık bellek işlemleri yapılabilmektedir. 32 bit Intel işlemcilerinde nesnelere gerçek sayı değerlerini atamanın birkaç yolu olabilir:

1) Atama işlemi dd, dq ve dt sembolik makine dili direktifleriyle ilkdeğer vererek yapılabilir. Bu durumda verilen ilkdeğerin gerçek sayı formatına dönüştürülüp ilgili adrese yerleştirilmesi sembolik makine dili derleyicisi tarafından yapılacaktır. Biz de oluşturulmuş olan bu değeri ilgili adresten alarak istediğimiz başka bir yere aktarabiliriz. Örneğin -20.5 gibi double bir değeri [EBP – 8]'den başlayarak bir yerel değişkene yerleştirmek istediğimizi düşünelim:

```
SECTION .data

val     dq      -20.5

;.....

mov     eax, [val]
mov     [ebp - 8], eax

mov     eax, [val + 4]
mov     [ebp - 4], eax
```

8 byte uzunluğundaki değeri 32 bit işlemcilerde tek bir MOV işlemi ile başka bir yere atayamayacağımızı biliyorsunuz. Bu nedenle yukarıdaki örnek kodda bu işlem iki aşamada yapılmıştır. Tabii bu atama işlemi fld ve fstp matematik işlemci komutlarıyla tek hamlede de yapılabilirdi:

```
fld  qword [val]
fstp qword [ebp - 8]
```

Ayrıca bazı özel değerlerin (örneğin 0 değeri, 1 değeri, pi değeri gibi) tek bir makine komutuyla yüklenebildiğini de anımsaynız. Örneğin:

```
fldpi
fstpqword [ebp - 8]
```

2) Yerleştirilmek istenen gerçek sayı değeri hesaplanıp MOV komutlarıyla sabit ataması biçiminde de yapılabilir. Örneğin -20.5 double değeri hex sistemde 0xC034800000000000 biçimindedir. Bu değerin yüksek anlamlı 4 byte'ı 0xC0348000, düşük anlamlı 4 byte'ı da 0x00000000 biçimindedir. O halde atama şöyle de yapılabilir:

```
mov     dword [ebp - 8], 0
mov     dword [ebp - 4], 0xC0348000
```

7.10. Intel İşlemcilerinde String Komutları

Intel'de bir bellek bloğu üzerinde işlem yapan bazı özel komutlara "string komutları" denilmektedir. String komutlarının sonu S harfiyle bitmektedir. Birkaç string komutu IO işlemleriyle ilgili olduğu için burada ele alınmayacaktır. Burada ele alınacak string komutları şunlardır: LODS, MOVS, STOS, SCAS ve CMPS. String komutları genellikle REP önekleriyle kullanılır. İki REP öneki vardır. Ancak sembolik makine dillerinde bu iki REP komutuyla eşdeğer olan (yani bunların farklı isimleri biçiminde olan) REP komut isimleri de bulunmaktadır. Örneğin REPE ile REPZ, REPNE ile de REPNZ tamamen eşdeğerdir. Yalnızca REP denildiğinde ise default olarak REPE/REPZ anlaşılmaktadır.

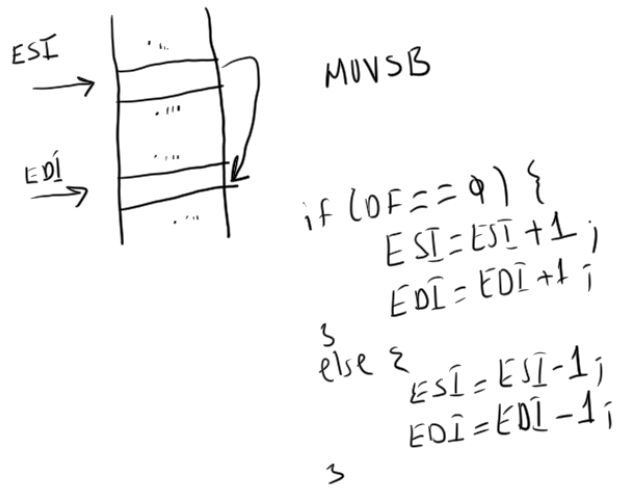
String komutları genel olarak DS:ESI ve ES:EDI yazmaçları ile gösterilen adresteki bilgiler üzerinde işlem yapmaktadır. 32 bit mimaride işletim sistemleri segment (selector de denilmektedir) yazmaçlarını belleğin tepesine ayarladığı için (bu sisteeme "flat" model denilmektedir) ESI ve EDI yazmaçlarının ilişkin olduğu segment yazmaçlarının bir önemi kalmamaktadır. String komutları işlem sonucunda aynı zamanda ESI ve/veya EDI yazmaçlarını da artırmakta ya da eksiltmektedir. İşlem sonucunda artırım mı eksiltim mi yapılacağı EFLAGS yazmacındaki DF (Direcition Dlag) bayrağına bağlıdır. DF = 0 ise artırım, DF = 1 ise eksiltim yapılır. Artırım ileriye doğru hareketi, eksiltim de geriye doğru hareketi belirtmektedir. DF bayrağını reset etmek için CLD, set etmek için de STD komutlarının bulunduğunu anımsayınız.

REP önekleri ("repitition" sözcüğünden kısaltmadır) string komutlarını belli sayıda devam ettirmek için kullanılmaktadır. Bu sayı da ECX (16 bit modda CX, 64 bit modda RCX) yazmacıyla ayarlanır. REP komutları her yinelemede ECX yazmacındaki değeri bir eksiltir. ECX'teki değer sıfıra düştüğünde ya da diğer başka koşullar oluştuğunda REP öneki string komutlarını yinelemeyi durrdurur.

String komutları byte düzeyinde, word düzeyinde, dword düzeyinde ya da qword düzeyinde uygulanabilmektedir. İşlemin hangi düzeyde yapılacağı komutun sonuna getirilen B, W, D, Q harfleriyle belirtilir (örneğin MOVSB, MOVSW, MOVSD, MOVSQ).

7.10.1. MOVS Komutları

Bu komut DS:ESI adresindeki bilgiyi ES:EDI adresine atamak için kullanılmaktadır. Eğer komut REP öneksiz kullanılırsa bu işlem yalnızca bir kez yapılır. Aktarımdan sonra DF'nin durumuna göre ESI ve EDI artılır ya da eksiltilir. Artırım mı eksiltim mi yapılacağı DF bayrağına bağlıdır. DF = 0 ise artırım DF = 1 ise eksiltim yapılmaktadır. Artırım ya da eksiltim komutun işlem genişliğine bağlıdır. Yani örneğin eğer komut MOVSB ise (yani atama 1 byte ise) ESI ve EDI bir artırılır ya eksiltilir, komut MOVSD ise (yani atama 4 byte ise) ESI ve EDI dört artırılır ya da eksiltilir. MOVS komutları bayrakları etkilememektedir.



Örneğin x adresindeki 1 byte'ı y adresine atamak isteyelim. Bu işlemi MOVS komutuyla şöyle yapabiliriz:

```
mov     esi, x
mov     edi, y
cld
movsb
```

İşlem sonucunda DF = 0 olduğu için ESI ve EDI bir artırılacaktır. MOVS komutlarının bellek-bellek işlemi yaptığına dikkat ediniz.

MOVS komutları ilk bakışta size anlamsız gelebilir. Çünkü bu işlemi yapmanın açık başka yolları da vardır. Örneğin:

```
mov al, [x]
mov [y], al
```

Evet, MOVS komutlarının REP öneksiz kullanımının genellikle ek bir faydası yoktur. Bu komut hemen her zaman REP önekiyle kullanılmaktadır.

Yukarıda da belirttiğimiz gibi gibi iki ayrı REP öneki vardır: REP/REPE/REPZ önek isimleri aslında aynı öneki, REPNE/REPNZ önek isimleri de aslında aynı öneki belirtmektedir. MOVS komutlarının bu öneklerin hangisiyle kullanıldığının bir önemi yoktur. Yani biz MOVS komutlarını bu iki önekten herhangi birisiyle kullanırsak komutta bir davranış değişikliği oluşmaz. REP önekleri MOVS komutlarıyla kullanılırken 16 bit modda CX, 32 bit modda ECX ve 64 bit modda RCX yazmaçlarına bakmaktadır. REP öneki MOVS işlemini yaptıktan sonra ECX yazmacını bir eksiltir ve işlemi yineler. Ta ki ECX yazmacındaki değer sıfır olana kadar. Başka bir deyişle biz işlemin ne kadar yinelenmesini istiyorsak REP MOVS işlemine başlamadan önce bu yinelenme sayısını ECX yazmacına yerleştirmemiz gerekir.

REP MOVS komutları tipik olarak etkin blok kopyalaması için tercih edilmektedir. Örneğin x adresinden başlayan ve her elemanı 4 byte olan 5 elemanlı bir diziyi (yani C'de 5 elemanlı int türden bir diziyi) y adresine şöyle kopyalayabiliriz:

```
mov     esi, x
mov     edi, y
cld
mov     ecx, 5
rep movsd
```

Örneğin C'deki memcpy fonksiyonu MOVS komutlarıyla daha etkin bir biçimde gerçekleştirilebilir:

```
[BITS 32]

SECTION .text
    global _mymemcpy

_mymemcpy:
    pushebp
    mov     ebp, esp
    pushesi
    pushedi

    mov     edi, [ebp + 8]
    mov     esi, [ebp + 12]
    mov     ecx, [ebp + 16]
    cld
    rep movsb

    mov     eax, [ebp + 8]

    pop     edi
    pop     esi
    pop     ebp
    ret
```

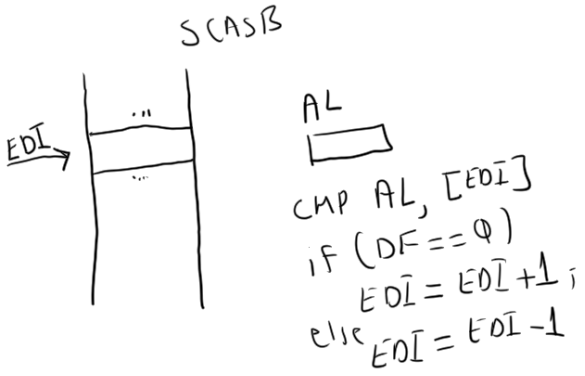
Çakışık blokların duruma göre sondan başa kopyalanması gerekebilmektedir. Bu işlemin ESI ve EDI yazmaçlarının bloğun sonuna ayarlanıp STD komutuyla DF bayrağı set edilerek yapılabileceğine dikkat ediniz.

Fonksiyon aşağıdaki gibi bir kodla test edilebilir:

```
#include <stdio.h>

void *mymemcpy(void *dest, const void *source, size_t size);

int main(void)
{
    int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int b[10];
    int i;

    mymemcpy(b, a, sizeof(int) * 10);
    for (i = 0; i < 10; ++i)
        printf("%d ", b[i]);
    printf("\n");

    return 0;
}
```

7.10.2. SCAS Komutları

SCAS komutları bir dizi içerisinde belli bir değeri bulmak için kullanılmaktadır. REP öneksiz olarak bu komutlar AL, AX, EAX ya da RAX içerisindeki değeri ES:EDI adresindeki değerle karşılatırır. Yine işlem 1 byte olarak, 2 byte olarak, 4 byte olarak ya da 8 byte olarak yapılabilmektedir. Karşılaştırma işlemi akümülatördeki değerden ES:EDI adresindeki değerin çıkartılmasıyla yapılır. (Tabii bu çıkartmadan akümülatör etkilenmemektedir. Yani işlem CMP gibi yapılmaktadır). Bayraklar bu çıkartma işleminden etkilenirler. SCAS işleminden sonra DF bayrağının durumuna göre yine EDI yazmacı işlem genişliği kadar artırılır ya da eksiltilir.



SCASB için kaynak operand AL, SCASW için AX, SCASD için EAX ve SCASQ için de RAX'tir. Örneğin:

```
mov     edi, x
mov     al, 123
cld
scasb
```

Burada x adresindeki değerle AL yazmacındaki değer karşılaştırılmak istenmiştir. Eğer bu değerler eşit ise çıkartma işleminin sonucu 0 vereceğinden ZF bayrağı set edilecektir.

SCAS komutları nadiren tek başlarına kullanılmaktadır. Bu komutlar genellikle REP/REPE/REPZ ya da REPNE/REPNZ önekleriyle kullanılırlar. Ancak bu iki önek grubunun SCAS komutlarındaki davranışları farklıdır. REPNE/REPNZ öneki ECX yazmacı 0 olmayana kadar ya da ZF bayrağı set edilmediği sürece (reset olduğu sürece) işlemi devam ettirir. Halbuki REP/REPE/REPZ öneki ise ECX sıfır olmayana kadar ya da ZF bayrağı reset edilmediği sürece (yani set olduğu sürece) SCAS işlemini devam ettirir. Yine her iki REP öneki de ECX yazmacını işlem sonrasında 1 eksiltmektedir.

Örneğin bir dizide belli bir değeri aramak için REPNE SCAS kalıbını, belli bir değerden farklı olan ilk değeri aramak için ise REPE SCAS kalıbını kullanırız.

Tipik olarak strlen gibi fonksiyon SCAS komutlarıyla gerçekleştirilebilmektedir:

```
[BITS 32]

SECTION .text
    global _mystrlen

_mystrlen:
    push ebp
    mov     ebp, esp
    push edi

    xor     ecx, ecx
    dec     ecx     ; ecx = 0xFFFFFFFF
    mov     edi, [ebp + 8]
    xor     al, al
    cld
    repnz   scasb

    neg     ecx
    lea     eax, [ecx - 2]

    pop     edi
    pop     ebp
    ret

    ret
```

Burada REPNE SCASB komutu sonlandığında ECX içerisindeki değerin işaretli olarak negatif biçimde olacağına dikkat ediniz. 0XFFFFFFFF işaretli olarak -1'dir. SCASB ECX'i azalttığı için arama sonucundaki karakter sayısı mutlak değer olarak 2 fazla olur. Biz de bu nedenle ECX'teki değeri önce NEG komutuyla pozitife dönüştürüp sonra bundan 2 çıkarttık. LEA EAX, [ECX – 2] komutunun ECX – 2 değerini EAX'e yerleştirdiğine dikkat ediniz. Tabii uzunluk hesaplama işlemi son durumdaki EDI'den dizinin başlangıç adresinin çıkartılmasıyla da yapılabilirdi:

```
[BITS 32]

SECTION .text
    global _mystrlen

_mystrlen:
    push ebp
    mov     ebp, esp
    push edi

    xor     ecx, ecx
    dec     ecx     ; ecx = 0xFFFFFFFF
    mov     edi, [ebp + 8]
    xor     al, al
    cld
    repnz   scasb

    sub     edi, [ebp + 8]
    lea     eax, [edi - 1]

    pop     edi
    pop     ebp
    ret

    ret
```

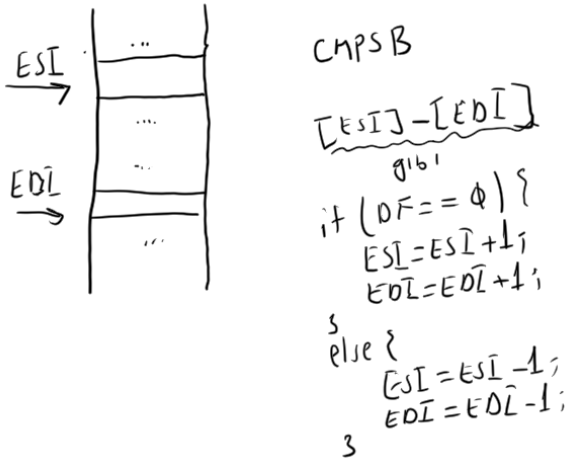
Test kodu şöyle olabilir:

```
#include <stdio.h>

size_t mystrlen(const char *str);

int main(void)
{
    size_t n;

    n = mystrlen("a");
    printf("%u\n", n);

    return 0;
}
```

7.10.3. CMPS Komutları

CMPS komutları adeta SCAS komutlarının iki bellek bölgesi ile yapılan biçimi gibidir. Yani örneğin, SCASB komutu AL yazmacındaki değer ile ES:EDI adresindeki değeri karşılaştırırken CMPSB komutu DS:ESI adresindeki değerle ES:EDI adresindeki değeri karşılaştırmaktadır.

CMPS komutu aslında DS:ESI adresindeki değerden ES:EDI adresindeki değeri çıkartır. Tabii çıkartma işleminden operandlar etkilenmez bu işlem CMP gibi düşünülmelidir. Bu çıkartma işleminden yine bayraklar etkilenecektir. Diğer string komutlarında olduğu gibi ESI ve EDI işlem sonrasında DF = 0 ise işlem genişliği kadar artırılır, DF = 1 ise eksiltilir.

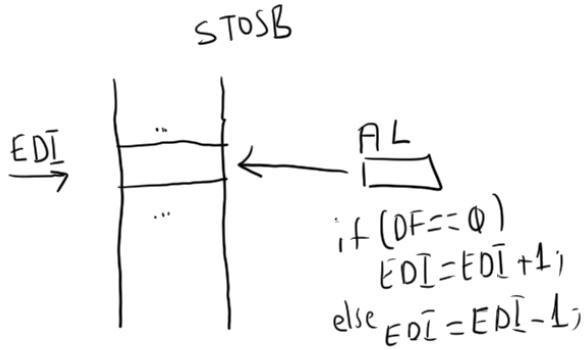


Tabii CMPS komutları da nadiren tek başlarına kullanılmaktadır. Bunlar genellikle REP/REPE/REPZ ya da REPNE/REPNE önekleriyle kullanılırlar. Bu önekler ECX yazmacı sıfır olana kadar ya da ZF bayrağı set ya da reset edilene kadar işlemin devam etmesini sağlarlar.

Örneğin iki yazının aynı olup olmadığına bakmak isteyelim. Bunun için önce yazıdan küçük olanın uzunluğunu ECX'e, yazıların adreslerini de ESI ve EDI yazmaçlarına yerleştirip CLD komutunu uygularız. Sonra da REPE SCASB işlemi ile karşılaştırmayı yapabiliriz. İşlem sonucunda ZF bayrağının durumuna baktığımzda eğer ZF set edilmişse komut ECX'in sıfır olması dolayısıyla sonlanmıştır. Demek ki yazılar birbirine eşittir. Eğer ZF reset edilmişse komut iki yazının karakterlerindee biri farklı olduğundansonlanmıştır. Demek ki yazılar birbirine eşit değildir.

7.10.4. STOS Komutları

Bu komut tipik olarak belli bir bellek bloğunu belli bir değerle doldurmak için kullanılmaktadır. STOS komutları işlem genişliğine göre AL, AX, EAX ya da RAX'teki değerleri ES:EDI adresine yerleştirirler. Sonra yine DF bayrağının durumuna göre EDI yazmacındaki değer işlem genişliği kadar artırılır ya da eksiltilir.



STOS komutları da hemen her zaman REP önekiyle kullanılmaktadır. Bu komutlarda hangi REP önekinin kullanıldığının bir önemi yoktur. (Yani örneğin REPE STOSB ile REPNE STOSB arasında bir fark yoktur.) STOS komutlarının REP önekleriyle kullanılması durumunda ECX yazmacı sıfır olana kadar aynı işlemler yinelenir. Yani örneğin REP STOSB komutu AL'deki değerin EDI adresinden itibaren ECX kadar sayıda kopyalanmasına yol açacaktır.

Tipik olarak C'deki memset fonksiyonu STOS komutlarıyla gerçekleştirilebilir:

```
[BITS 32]

SECTION .text
    global _mymemset
```

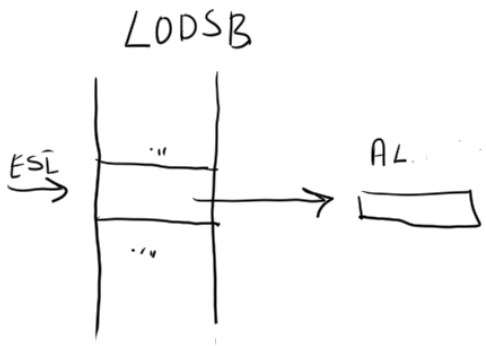
```
_mymemset:
    push ebp
    mov     ebp, esp
    push edi

    mov     edi, [ebp + 8]
    mov     al, [ebp + 12]
    mov     ecx, [ebp + 16]
    cld
    rep     stosb

    mov     eax, [ebp + 8]

    pop     edi
    pop     ebp
    ret

    ret
```

Test kodu şöyle olabilir:

```
#include <stdio.h>

void *mymemset(void *ptr, int ch, size_t n);

int main(void)
{
    int a[10];
    int i;

    mymemset(a, 0, sizeof(a));

    for (i = 0; i < sizeof(a) / sizeof(*a); ++i)
        printf("%d ", a[i]);
    printf("\n");

    return 0;
}
```

7.10.5. LODS Komutları

LODS komutları çok seyrek kullanılmaktadır. Bu komutlar STOS komutlarının tam tersini yaparlar. Yani LODS komutları ile DS:ESI adresindeki değer işlem genişliğine göre AL, AX, EAX ya da RAX yazmaçlarına yerleştirilmektedir. LODS komutlarının REP önekleriyle kullanılmasının bir anlamı yoktur. LODS işlemi sonrasında yine ESI yazmacındaki değer DF = 0 ise ESI işlem genişliği kadar artırılır, DF = 1 ise ESI işlem genişliği kadar eksiltilir.



Örneğin:

```
mov    esi, x
cld
lodsb
```

Bu örnekte x adresindeki 1 byte AL yazmacına aktarılmaktadır.

7.11. C ve C++ Derleyicilerinin İsim Dekorasyonları (Name Decoration / Name Mangling)

C ve C++ derleyicileri global nesne ve fonksiyon isimlerini amaç koda (object file) değiştirerek yazabilmektedir. Sembolik makine dilinde yazılan fonksiyonların C ve C++'tan çağrılması sürecinde programcının uygulanan isim dekorasyonu hakkında bilgi sahibi olması gerekir. İsim dekorasyonu derleyiciden derleyiciye, platformdan platforma değişebilmektedir. 32 bit derleyicilerle 64 bit derleyiciler arasında da isim dekarosayonu bakımından farklılıklar olabilmektedir. Fakat genel olarak 32 bit derleyicilerle 64 bit derleyicilerin isim dekorasyonları birbirlerine çok benzediğini söyleyebiliriz.

Pekiyi neden C ve C++ derleyicileri global nesne ve fonksiyon isimlerini olduğu gibi değil de değiştirerek (dekore ederek) amaç koda yazmaktadır? Bunun birkaç nedeni vardır:

- C++'ta farklı isim alanlarında aynı isimli global değişkenler ve fonksiyonlar bulunabilmektedir.

- C++'ta farklı parametrik yapılara ilişkin aynı isimli fonksiyonlar (function overloading) bulunabilmektedir.

- Bağlayıcılar bazı bilgileri isim dekorasyonundan elde edebilmektedir.

- Bazı isimler sembolik makine dilindeki bazı direktiflerle ya da makine komutlarıyla çakışabilmekedir. C ve C++ derleyicileri sembolik makine dili çıktısı üretirken bunun derlenmesi bir sorun olabilmektedir. (Örneğin add isimli bir fonksiyon olsun. Biz böyle bir C programı için sembolik makine dili çıktısı elde ettiğimizde bu isim ADD makine komutuyla karışabilir. Bazı sembolik makine dili derleyicileri kendi içerisinde bu sorunu çözüyor olsa da bazıları için bu durum sorun yaratabilmektedir.)

İsim dekorasyonu uygulamak yalnızca C ve C++ derleyicileri için söz konusu olan bir özellik değildir. Amaç kod üreten diğer diller için yazılmış derleyiciler de isim dekorasyonları uygulayabilmektedir. Ancak biz bu başlıkta C ve C++ derleyicilerinin uyguladıkları isim dekorasyonları üzerinde duracağız. Burada nir noktayı da vurgulamak istiyoruz: Genel olarak sembolik makine dili derleyicileri hiçbir isim dekorasyonu uygulamazlar. Programcı sembollerin isimlerini nasıl vermişse sembolik makine dili derleyicileri onları hiç değiştirmeden amaç dosyaya yazmaktadır.

7.11.1. Amaç Dosya İçerisindeki İsimlerin Görüntülenmesi

Sembolik makine dili programcısının isim dekorasyonlarını çok iyi bilmesine gerek yoktur. Ancak böyle bir olgunun varlığını bilmelidir. Bir ismin derleyici tarafından nasıl dekore edildiğini bazı araçlarla görebilir. Burada bu araçlardan birkaçını tanıtmak istiyoruz.

Windows sistemlerinde Microsoft'un "dumpbin" isimli aracı "/symbols" seçeneğiyle kullanılırsa amaç dosyaların ve çalıştırılabilir dosyaların sembol tablosunu görüntülemektedir. Buradan biz ilgili ismin nasıl dekore edildiğini bulabiliriz. Örneğin "test.cpp" isimli dosyada aşağıdakitanımlamalar olsun:

```
int g_a;

int Foo(int a, int b)
{
    return a + b;
}
```

Biz bu dosyayı derleyerek "test.obj" dosyasını elde etmiş olalım:

```
dumpbin /symbols test.obj
```

komutunu uyguladığımızda aşağıdaki gibi bir çıktı elde ederiz:

```
Microsoft (R) COFF/PE Dumper Version 14.00.23506.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file decorationCpp.obj

File Type: COFF OBJECT

COFF SYMBOL TABLE
000 01055BD2 ABS    notype       Static       | @comp.id
001 80000191 ABS    notype       Static       | @feat.00
002 00000000 SECT1  notype       Static       | .drectve
    Section length   41, #relocs    0, #linenums    0, checksum        0
    Relocation CRC 00000000
005 00000000 SECT2  notype       Static       | .debug$S
    Section length  89C, #relocs    2, #linenums    0, checksum        0
    Relocation CRC 2B25DD3C
008 00000000 SECT3  notype       Static       | .debug$T
    Section length   64, #relocs    0, #linenums    0, checksum        0
    Relocation CRC 00000000
00B 00000000 SECT4  notype       Static       | .bss
    Section length    4, #relocs    0, #linenums    0, checksum        0
    Relocation CRC 00000000
00E 00000000 SECT4  notype       External     | ?g_a@@3HA (int g_a)
00F 00000000 SECT5  notype       Static       | .text$mn
    Section length   2B, #relocs    0, #linenums    0, checksum C30536D7, selection    1 (pick no
duplicates)
    Relocation CRC AABCEB83
012 00000000 SECT6  notype       Static       | .debug$S
    Section length   DC, #relocs    5, #linenums    0, checksum        0, selection    5 (pick
associative Section 0x5)
    Relocation CRC 303411D6
015 00000000 SECT5  notype ()    External     | ?Foo@@YAHHH@Z (int __cdecl Foo(int,int))
016 00000000 UNDEF  notype ()    External     | __RTC_InitBase
017 00000000 UNDEF  notype ()    External     | __RTC_Shutdown
018 00000000 SECT7  notype       Static       | .rtc$IMZ
    Section length    4, #relocs    1, #linenums    0, checksum        0, selection    2 (pick any)
    Relocation CRC 5D907A9E
01B 00000000 SECT7  notype       Static       | __RTC_InitBase.rtc$IMZ
01C 00000000 SECT8  notype       Static       | .rtc$TMZ
    Section length    4, #relocs    1, #linenums    0, checksum        0, selection    2 (pick any)
    Relocation CRC 4C2E11CC
01F 00000000 SECT8  notype       Static       | __RTC_Shutdown.rtc$TMZ

String Table Size = 0x68 bytes

  Summary

           4 .bss
         978 .debug$S
          64 .debug$T
          41 .drectve
           4 .rtc$IMZ
           4 .rtc$TMZ
          2B .text$mn
```

UNIX/Linux sistemlerinde benzer biçimde "objdump" aracı "-t" seçeneğiyle amaç dosyaların ve çalıştırılabilen dosyaların sembol tablosunu görüntülemektedir. Örneğin Linux sistemlerinde biz "test.cpp" dosyasını derleyerek "test.o" dosyasını oluşturmuş olalım:

```
objdump –t test.o
```

Çıktı aşağıdaki gibi olacaktır:

```
test.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.cpp
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame   0000000000000000 .eh_frame
0000000000000000 l    d  .comment    0000000000000000 .comment
0000000000000000 g     O .bss   0000000000000004 g_a
0000000000000000 g     F .text  0000000000000014 _Z3addii
```

Yine UNIX/Linux sistemlerinde "nm" isimli utility'si de benzer işlemi yapmaktadır:

```
nm test.o
```

Programın çıktısı şöyle olacaktır:

```
0000000000000000 B g_a
0000000000000000 T _Z3addii
```

Dekore edilmiş isimler doğrudan ilgili C/C++ programının sembolik makine çıktısı üretecek biçiminde derlenmesiyle de gözlemlenebilir. Üretilen sembolik makine dili programının içerisinde dekaore edilmiş isimler bulunacaktır.

7.11.2. C Derleyicilerinde İsim Dekorasyonu

C'de isim alanları olmadığı için ve farklı parametrik yapılara ilişkin aynı isimli fonklsiyonlar (function overloading) tanımlanamadığı için C derleyicilerinin uyguladığı isim dekorasyonları da oldukça yalındır. Ancak bazı derleyicilerde fonksiyonların isim dekorasyonları çağırma biçimine göre farklılıklar gösterebilmektedir.

7.11.2.1. Microsoft C Derleyicisinin İsim Dekorasyonu

Microsoft C derleyicilerinde tüm global nesne isimlerinin başına '_' karakteri getirilmektedir. Ancak fonksiyonların isim dekorasyonu çağırma biçimine bağlıdır. (Bilindiği gibi __cdecl, __stdcall, __fastcall ve __thiscall 32 C derleyicileri için kullanılan çağırma biçimleridir.)

__cdecl çağırma biçiminde fonksiyon isimlerinin başına '_' getirilmektedir. Örneğin:

```
int add(int a, int b);
int g_a;
```

Global isimleri sırasıyla Microsoft'un C derleyicileri tarafından _add ve _g_a biçiminde dekore edilir.

__stdcall çağırma biçiminde fonksiyon isimlerinin başına önce bir '_' karakteri sonra ismin kendisi, sonra bir '@' karakteri ve sonra da fonksiyonun parametrelerinin byte sayısı getirilmektedir. Örneğin:

```
int __stdcall add(int a, int b);
```

Fonksiyonun isim dekorasyonu şöyle yapılmaktadır:

```
_add@8
```

__fastcall çağırma biçiminde fonksiyon isminin başına önce bir '@' karakteri getirilir. Sonra bunu fonksiyonun ismi ve bir '@' karakteri izler. Bundan sonra fonksiyonun parametre değişkenlerinin byte

uzunluğu getirilir. Örneğin:

```
int __fastcall add(int a, int b);
```

İsim dekorasyonu şöyle yapılmaktadır:

```
@add@8
```

7.11.2.2. GNU C Derleyicisinde (GCC) İsim Dekorasyonu

GCC derleyicileri geleneksel olarak global değişken isimlerini doğrudan amaç koda yazmaktadır. Bunlar isimlerin başına Microsoft'taki gibi '_' karakteri getirmezler. Ayrıca GCC'de fonksiyon isimlerindeki dekorasyonda da herhangi bir şey yapılmamaktadır. Çağırma biçiminin de dekorasyonda bir önemi yoktur. Örneğin:

```
int g_a;
int add(int a, int b);
```

İsimlerinin dekorasyonu GCC C derleyicileri tarafından şöyle yapılır:

```
g_a
add
```

7.11.3. C++ Derleyicilerinde İsim Dekorasyonu

C++ derleyicilerinde isim dekorasyonları biraz ayrıntılıdır. Burada biz bu ayrıntılara girmeyeceğiz. Ancak kurs dokümanlarındaki "Doc/Others/CallingConventions" isimli makalenin 8. Bölümünde konu ayrıntılarıyla açıklanmıştır. Ayrıntılar için bu dokümanlara başvurulabilirsiiniz. Ayrıca Wikipedia'da "Name Mangling" sayfasının "External Links" kısmında belirtilen makalaler de ayrıntılar için yardımcı olabilir.

Anımsanacağı gibi C++'ta extern "C" bildirimi C'de yazılmış fonksiyonları çağırmak için kullanılmaktadır. Bu bağlama özelliğine sahip fonksiyonlar C++ derleyicisini yazan şirket ya da kurumun ilgili C derleycisinin kurallarına göre dekore edilmektedir. Örneğin:

```
extern "C" void foo(void);
```

Microsoft C++ derleyicileri bu fonksiyonu "_foo" biçiminde, GCC C++ (G++) derleyicileri ise "foo" biçiminde dekore edecektir.

Şimdi Microsoft ve GCC C++ (G++) derleyicilerindeki isim dekorasyonlarını ele alacağız. Bu dekorasyonlardaki genel biçimlerin EBNF tarzı bir notasyonla betimlendiğini görecekseniz. Bu notasyondaki açısal parantezler zorunlu öğeleri, köşeli parantezler zorunlu olmayan öğeleri belirtiyor.

7.11.3.1. Microsoft C++ Derleyicilerindeki İsim Dekorasyonu

Microsoft'un C++ derleyicilerindeki isim dekorasyonu biraz karmaşıktır. Dekorasyon ismin bir nesne ismi mi, global bir fonksiyon ismi mi yoksa bir üye fonksiyon ismi mi olduğuna göre değişmektedir. Burada biz tüm ayrıntılara girmeyeceğiz.

Global bir nesne dekorasyonu şöyle yapılmaktadır:

<public name> ::= ? <name> @ $\left[< \text{namespace} > @ \right]_0^\infty$ @ 3 <type> <storage class>

Buna göre dekore edilmiş isim bir '?' karakteri ile başlar. Sonra bunu dekore edilmemiş isim, ve '@'

karakteri izler. Bunu da isimin bulunduğu isim alanı isimleri a@b@c... biçiminde bunu izlemektedir. Bundan sonra bir '@' karakteri daha bulunmaktadır. Sonra bunu bir '3' karakteri ve nesnenin türünü belirten büyük harf bir karaketer izlemektedir. Dekorasyonn sonunda da "storage class specifier"ları (auto, extern, register, static gibi) belirten karakterler bulunmaktadır. Türler ve "storage class specifier"lar için belirlenen önekler "Doc/Others/CallingConventions" makalesinde dokümante edilmiştir. Tür belirten harflerden bazıları şunlardır:

void	X
bool	_N
char	D
signed char	C
unsigned char	E
short int	F
unsigned short int	G
int	H
unsigned int	I
long int	J
unsigned long int	K
long long (__int64)	_J
unsigned long long (unsigned __int64)	_K
wchar_t	_W (G)
float	M
double	N
long double	O, _T, _Z [3]

"Storage Class Specifier" belirten harfler de şunlardır:

(default)	A
near	A
const	B
volatile	C
const volatile	D

Şimdi bir örnek verelim. Örnekler için aşağıdaki programı kullanıyor olalım:

```
double pi = 3.1415;

namespace X
{
    namespace Y
    {
        static int count = 10;
        //...
    }

    const int *ptr;
}
```

Buradaki isimlerin dekore edilmiş karşılıkları şöyledir:

Değişken İsmi	Dekore Edilmiş İsim
pi	?pi@@3NA
count	?count@Y@X@@3HA
ptr	?ptr@X@@3PBHB

Global fonksiyonların dekore edilme kuralı da şöyledir:

<public name> ::= ? <function name> @ $\left[< \text{namespace} > @ \right]_0^\infty$ @ <near far>
<calling conv> [<stor ret>] <return type> $\left[< \text{parameter type} > \right]_1^\infty$ <term> Z

Burada near için eğer fonksiyon global ise 'Y', üye fonksiyon ise 'Q' kullanılmaktadır. Çağırma biçimi için kullanılan harfler şunlardır:

__cdecl	A[17]
__pascal	C
__fortran	C
__thiscall	E
__stdcall	G
__fastcall	I[17]
__regcall	E
__vectorcall	
interrupt	A

Global fonksiyonlar için şöyle bir örnek verebiliriz:

```
int Add(int a, int b)
{
    return a + b;
}

namespace X
{
    namespace Y
    {
        double Multiply(double a, double b)
        {
            return a * b;
        }
    }

    void Sort(int *pi, int size)
    {
        //...
    }
}
```

Programdaki global fonksiyonların dekore edilmiş isimleri şöyle olacaktır:

Fonksiyon İsmi	Dekore Edilmiş İsim
Add	?Add@@YAHHH@Z
Multiply	?Multiply@Y@X@@YANNN@Z
Sort	?Sort@X@@YAXPAHH@Z

Sınıfların üye fonksiyonları için uygulanan dekorasyon da şöyledir:

<public name> ::= ? <function name> @ $\left[< \text{class name} > @ \right]_1^\infty$ @ <modif> [<const vol>]
<calling conv> [<stor ret>] <return type> $\left[< \text{parameter type} > \right]_1^\infty$ <term> Z

Örnek için aşağıdaki programı kullanabiliriz:

```
namespace X
{
```

```
    namespace Y
    {
        class Sample
        {
        public:
            Sample();
            void Set(int a, int b);
            void Disp();
        private:
            int m_a, m_b;
        };
    }
}

X::Y::Sample::Sample()
{}

void X::Y::Sample::Set(int a, int b)
{
    //...
}

void X::Y::Sample::Disp()
{
    //...
}
```

Programdaki global fonksiyonların dekore edilmiş isimleri şöyle olacaktır:

Üye Fonksiyon İsmi	Dekore Edilmiş İsim
Sample	??0Sample@Y@X@@QAE@XZ
Set	?Set@Sample@Y@X@@QAEXHH@Z
Disp	?Disp@Sample@Y@X@@QAEXXZ

7.11.3.2. GCC C++ (G++) Derleyicilerindeki İsim Dekorasyonu

GCC'nin C++ derleyicisi G++ olarak bilinmektedir. Fakat G++'nın 3.4 versiyonuna kadarki isim dekorasyonuyla 3.4 ve sonrasındaki isim dekorasyonu arasında farklılıklar vardır. Biz burada 3.4 ve sonrası tarafından uygulanan dekorasyonu ele alacağız. G++ tarafından uygulanan dekorasyonun genel yapı olarak Microsoft'a benzediği söylenebilir.

G++'da eğer global isim global isim alanı içerisindeyse hiç dekore edilmez. Dekore edilmiş isim aynen kullanılır. Eğer isim bir isim alanının ya da bir sınıfın içerisindeyse dekorasyonu şöyle yapılır:

```
<public name> ::= _Z <qualified name>
<qualified name> ::= N [<simple name>]+ E
<simple name> ::= <name length> <name>
```

Örnek için yine aynı programı kullanabiliriz:

```
double pi = 3.1415;

namespace X
{
    namespace Y
    {
        static int count = 10;
        //...
    }
```

```
    const int *ptr;
}
```

Programdaki global fonksiyonların dekore edilmiş isimleri şöyle olacaktır:

Değişken İsmi	Dekore Edilmiş İsim
pi	Pi
count	_ZN1X1YL5countE
ptr	_ZN1X3ptrE

Global fonksiyonlar ve üye fonksiyonlar için isim dekorasyonları da şöyledir:

```
<public name> ::= _Z <simple or qualified name> [<parameter type>]∞1

<simple or qualified name> ::= <simple name> | <qualified name> | <operator name>
```

G++ derleyicilerinin isim dekorasyonlarının ayrıntıları için konun başında önerdiğimiz makaleleri inceleyebilirsiniz.

7.12. C++'taki Üye Fonksiyonların Sembolik Makine Dilinde Kodlanması

Çok seyrek de olsa bazen biz bir sınıfın belli bir üye fonksiyonunu sembolik makine dilinde yazmak isteyebiliriz. Bu yazım sırasında dikkat edilmesi gereken noktalar şunlardır:

1) Üye fonksiyonların Microsoft C++ derleyicilerindeki default çağırma biçimleri __thiscall, G++ derleyicilerindeki default çağırma biçimleri ise cdecl şeklindedir. (Anımsanacağı gibi __thiscall çağırma biçiminde this göstericisi ECX yazmacı yoluyla, cdecl çağırma biçimine göre ise ilk argüman olarak aktarılmaktadır.)

2) Üye fonksiyonları sembolik makine dilinde yazarken isim dekorasyonuna dikkat etmek gerekir.

3) C++ derleyicileri genel olarak sınıfın static olmayan veri elemanlarını bir yapı gibi peş peşe yerleştirmektedir. Ancak bu durum C++ standartlarında garanti altına alınmamıştır. (Standartlar iki erişim belirleyici arasındaki veri elemanlarının ardışıl olacağı konusunda garanti vermektedir.) Derleyicinizin static olmayan veri elemanlarını nesne içerisinde organize etme biçiminden emin olmalısınız.

Şimdi Microsoft C++ derleyicisi için aşağıdaki sınıfın Set üye fonksiyonunu sembolik makine dilinde yazacak olalım. Microsoft derleyicileri sınıfın static olmayan veri elemanlarını yapılarda olduğu gibi peşi sıra dizmektedir:

```
#include <iostream>

using namespace std;

class Sample {
public:
    void Set(int a, int b);
    void Disp() const;
private:
    int m_a, m_b;
};

void Sample::Disp() const
{
    cout << m_a << ", " << m_b << endl;
}
```

```
int main()
{
    Sample s;

    s.Set(10, 20);
    s.Disp();

    return 0;
}
```

Burada main fonksiyonunda çağırma şöyle yapılacaktır.:

```
_main:
    push    ebp
    mov     ebp, esp
    sub     esp, 8          ; s nesnesi için yer ayrıldı

    lea     ecx, [ebp - 8]      ; ecx = &s
    push    20
    push    10
    call    ?Set@Sample@@QAEXHH@Z

    lea     ecx, [ebp - 8]
    call    ?Disp@Sample@Y@X@@QAEXXZ

    pop     ebp
    ret
```

Yazım şöyle yapılabilir:

```
[BITS 32]

SECTION .text
    global ?Set@Sample@@QAEXHH@Z

?Set@Sample@@QAEXHH@Z:
    push ebp
    mov     ebp, esp

    mov     eax, [ebp + 8]    ; eax = a
    mov     [ecx], eax        ; m_a = eax

    mov     eax, [ebp + 12]   ; eax = b
    mov     [ecx + 4], eax    ; m_b = eax

    pop     ebp
    ret     8
```

Aynı C++ programının G++ derleyicisinde yazılmış olduğunu varsayalım:

```
#include <iostream>

using namespace std;

class Sample {
public:
    void Set(int a, int b);
    void Disp() const;
private:
    int m_a, m_b;
};

void Sample::Disp() const
{
```

```
    cout << m_a << ", " << m_b << endl;
}

int main()
{
    Sample s;

    s.Set(10, 20);
    s.Disp();

    return 0;
}
```

Buradaki main fonksiyonunda çağrımlar da şöyle yapılacaktır:

```
_main:
    push    ebp
    mov     ebp, esp
    sub     esp, 8          ; s nesnesi için yer ayrıldı

    push    20
    push    10
    mov     eax, [ebp - 8]
    push    eax
    call    _ZN6Sample3SetEii
    add     esp, 12

    mov     eax, [ebp - 8]
    push    eax
    call    _ZNK6Sample4DispEv
    add     esp, 4

    pop     ebp
    ret
```

Bu durumda Set üye fonksiyonu sembolik makine dilinde şöyle yazılabilir:

```
[BITS 32]

SECTION .text
    global _ZN6Sample3SetEii

_ZN6Sample3SetEii:

    pushebp
    mov     ebp, esp

    mov     ecx, [ebp + 8]    ; ecx = this

    mov     eax, [ebp + 12]    ; eax = a
    mov     [ecx], eax         ; m_a = eax

    mov     eax, [ebp + 16]    ; eax = b
    mov     [ecx + 4], eax     ; m_b = eax

    pop     ebp
    ret
```

8. Yeniden Konumlandırma (Relocation) İşlemleri

Bir programın doğal makine diline derlenmesi sonucunda makine kodlarında görülen adresler nihai adresler değildir. Programın düzgün çalışabilmesi için derleyiciler tarafından üretilmiş olan bu adreslerin belli biçimlerde değiştirilmesi gerekmektedir. İşte üretilmiş makine kodlarının içerisindeki adreslerin

değiştirilmesi sürecine yeniden "konumlandırma (relocation)" denilmektedir. Yeniden konumlandırma işlemleri bağlayıcılar tarafından ya da işletim sistemlerinin yükleyicileri tarafından yapılabilmektedir. Bölüm içerisinde bunları sırasıyla ele alacağız.

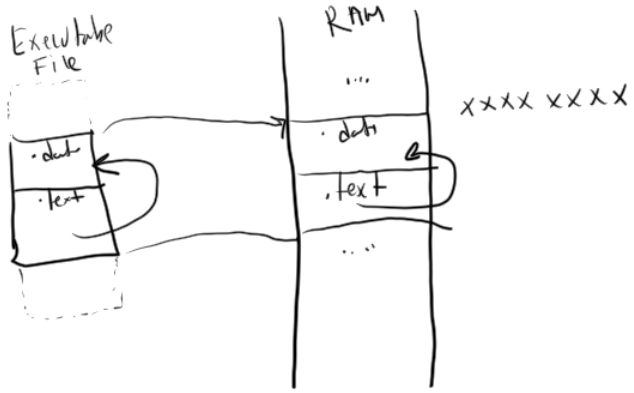
8.1. Çalıştırılabilen Dosyaların Yüklenmesi Sırasında Yükleyiciler Tarafından Uygulanan Yeniden Konumlandırma İşlemleri

Çalıştırılabilen dosya içerisindeki adresler gerçek doğrusal adresler olmayabilirler. Çünkü çalıştırılabilen programın RAM'in neresine yükleneceği pek çok durumda önceden bilinememektedir. Örneğin sembolik makine dilinde yazılmış aşağıdaki gibi bir program bulunuyor olsun (programı dikkatlice incelemeniz gerekmiyor):

```
[BITS 32]

SECTION .data

message1      db       'this is a test', 13, 10, 0,
message2      db       'this is another test', 13, 10, 0
message3      db       'this is the last test', 13, 10, 0

stdhandle     equ      -11

SECTION .text
     global _start
     extern  _GetStdHandle@4, _WriteFile@20, _ExitProcess@4

_start:

     mov      eax, message1
     call dispstr

     mov      eax, message2
     call dispstr

     mov      eax, message3
     call dispstr

     push 0
     call _ExitProcess@4

     ret      0

dispstr:
     xor      ecx, ecx
     push eax
REPEAT:
     inc      ecx
     mov      bl, [eax]
     inc      eax
     test bl, bl
     jnz      REPEAT
     dec      ecx
     push     ecx

     push     stdhandle
     call     _GetStdHandle@4

     pop      ecx
     pop      edx
     sub      esp, 4            ; yerel değişken için yer ayrılıyor
     push     0
     lea      ebx, [esp + 4]
     push     ebx
     push     ecx
     push     edx
     push     eax
     call     _WriteFile@20
```

```
    add     esp, 4
    ret
```

Bu programda bazı etiketlerin kullanıldığını görüyorsunuz. Pekiyi bu etiketlerin belirttiği adresler henüz program çalışmadan nasıl belirlenmektedir? Normal olarak programın düzgün bir biçimde çalışabilmesi için bu adreslerin RAM'in tepesine göre bir yer belirtmesi gerekir. (RAM'in tepesine göre yer belirten adreslere doğrusal (linear) adresler de denilmektedir.) Fakat pek çok durumda işletim sistemlerinin yükleyicileri (loaders) programları RAM'de o anda boş buldukları yerlere yüklerler. İşte yükleyiciler çalıştırılabilen dosyayı RAM'in herhangi bir yerinw yüklediklerinde program içerisindeki adresler artık geçerli olmazlar.

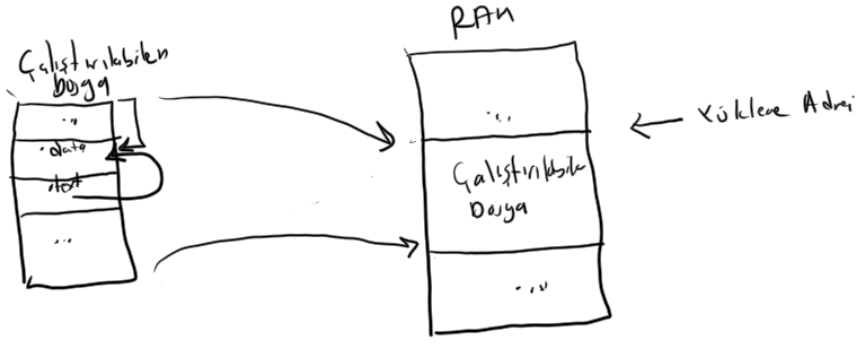


Programın düzgün çalışabilmesi için çalıştırılabilen dosyanın içerisindeki adreslerin yükleme işleminden sonra geçerli olması gerekir. Bu ise iki biçimde sağlanabilmektedir:

1) Çalıştırılabilen dosyadaki adresler programın RAM'de belli bir adrese yükleneceği fikriyle oluşturulmuş olabilir. Bu durumda aslında çalıştırılabilen dosyadaki adresler gerçek doğrusal adreslerdir. Tabii bu biçimde oluşturulmuş olan çalıştırılabilen dosyaların RAM'e belirlenen adresten itibaren yüklenmesi gerekir. Bu durumda da doğal olarak bir yeniden konumlandırma işlemi gerekmeyecektir.

2) Çalıştırılabilen dosyanın içerisindeki derleyici ve bağlayıcı tarafından üretilmiş adresler RAM'in tepesinden itibaren yer belirten gerçek doğrusal adresler değildir. Çalıştırılabilen dosyanın başından itibaren yer belirten göreli adreslerdir. Bu durumda işletim sisteminin yükleyicisi çalıştırılabilen dosyayı RAM'de herhangi bir yere yükleyebilir. Fakat bunun için çalıştırılabilen dosyanın içerisindeki tüm göreli adresleri dosyanın yüklendiği adres ile toplayarak çalıştırılacak koddaki adresleri düzeltmesi gerekir. İşte yukarıda da belirtildiği gibi bu sürece yeniden konumlandırma (relocation) denilmektedir.

Yeniden konumlandırma işleminin yükleyici tarafından yapılabilmesi için program içerisindeki tüm göreli adreslerin dosyadaki yerlerinin çalıştırılabilen dosyanın bir yerinde tutulması gerekir. Çalıştırılabilen dosyadaki düzeltilecek göreli adreslerin dosya içerisindeki offset'lerinin tutulduğu tabloya "yeniden konumlandırma tablosu (relocation table)" denilmektedir.



Yeniden konumlandırma işleminin sistemden sisteme bazı ayrıntıları vardır. Fakat biz bu noktada bu

ayrıntılar üzerinde durmayacağız. Amacımız yalnızca yeniden konumlandırma kavramı üzerinde bir bilinç oluşturmak.

Yüklenen programdaki adreslerin geçerliliğini sağlamak için kullanılan iki tekniği yukarıda açıkladık. Özetlersek birinci teknik programın zaten belli bir yere yükleneceği fikriyle oluşturulması, ikinci teknik de program içerisindeki adreslerin yükleme işlemi sırasında düzeltilmesiydi. Aslında yukarıdaki açıkladığımız bu birinci ve ikinci yöntemler birlikte de kullanılabilmektedir. Yani çalıştırılabilen dosya belli bir adrese yüklenecek biçimde gerçek doğrusal adresler içerebilir ve bunun yanı sıra ayrıca çalıştırılabilen dosya bir "yeniden konumlandırma tablosuna (relocation table)" da sahip olabilir. Bu durumda eğer yükleyici çalıştırılabilen dosyayı bu önerilen adrese (preferred addres) yükleyebilirse dosyanın yeniden konumlandırılmasına gerek kalmaz. Ancak yükleyici dosyayı birtakım nedenlerden dolayı bu adres yerine başka bir yere yüklerse artık yeniden konumlandırma işleminin uygulanması gerekecektir. Şüphesiz yükleyiciler tarafından uygulanan bu yeniden konumlandırma işlemleri programların yüklenme zamanını uzatıcı bir etken oluşturmaktadır.

Windows'ta ".exe" uzantılı çalıştırılabilen dosyalar genellikle belli bir adrese yüklenecek biçimde oluşturulmaktadır. Böylece Windows'un yükleyicisi de hiç yeniden konumlandırma yapmadan çalıştırılabilen dosyaları belleğe yükleyebilmektedir (birinci yöntem). Ancak bu sistemlerde DLL'lerin yeniden konumlandırma işlemi yapılmadan yüklenmesi çoğu zaman mümkün olmamaktadır. Windows sistemlerindeki DLL'ler de belli bir adrese yüklenecek biçimde oluşturulurlar ancak birden fazla DLL'in adres alanına yüklenmesi durumunda ilk DLL'in dışındaki DLL'ler için yeniden konumlandırma işlemi gerekmektedir.

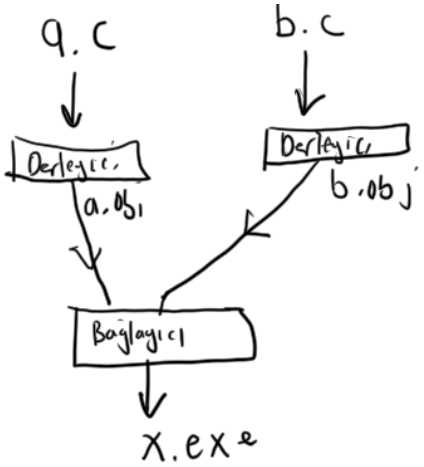
Çalıştırılabilen dosyaların yüklenmesi sırasında yükleyiciler tarafından uygulanan yeniden konumlandırma işlemi hakkında sıkça sorulan sorular şunlardır:

Soru: Bir program içerisindeki hangi adresler yeniden konumlandırma işlemi sırasında düzeltilmektedir?
Yanıt: Global nesnelerin ve fonksiyonların adresleri yeniden konumlandırma işlemi sırasında düzeltilirler. JMP ve CALL komutlarının büyük bölümünün mutlak adresleri değil göreli uzaklık değerini komut operandı olarak aldığını anımsayınız. Bu nedenle JMP ve CALL komutlarındaki adreslerin relocation işlemi sırasında düzeltilmesi gerekmemektedir. Benzer biçimde yerel değişkenlerin ve parametre değişkenlerinin adresleri de mutlak değil ESP yazmacına (ya da EBP yazmacına) göre görelidir. Dolayısıyla bu adresler için de yeniden konumlandırma işleminin yapılması gerekmez.

Soru: Yeniden konumlandırma tablosu çalıştırılabilen dosyanın neresindedir?
Yanıt: Çalıştırılabilen dosya formatlarının içerisinde bu formatların başlık kısımlarında yeniden konumlandırma tablosunun dosya içerisindeki yeri ve uzunluğu bulunmaktadır. Örneğin Windows'un PE formatında yeniden konumlandırma tablosunun yeri ve uzunluğu "Image Optional Header" başlığındaki "Data Directory" alanında belirtilmektedir. Windows sistemlerinde tipik olarak yeniden konumlandırma tablosu ".reloc" isimli bölümde (section) bulunmaktadır.

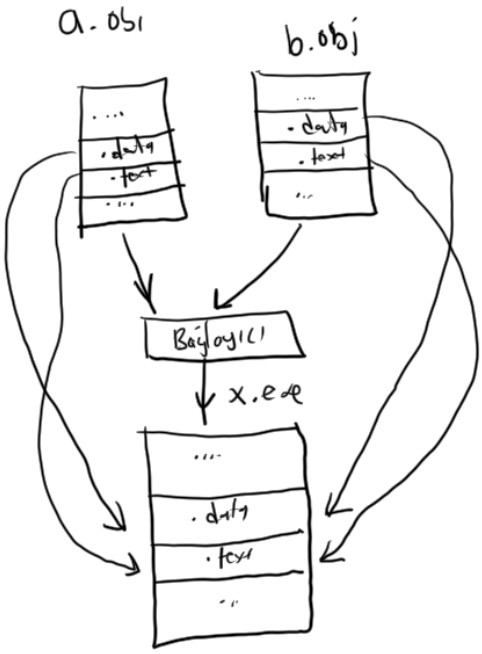
8.2. Amaç Dosyaların (Object Files) Birleştirilmesi Sırasında Bağlayıcılar Tarafından Uygulanan Yeniden Konumlandırma (Relocation) İşlemi

Bilindiği gibi çalıştırılabilen dosyalar tipik olarak iki aşamadan geçilerek elde edilmektedir: Önce bir grup kaynak dosya derlenip amaç dosyalar (object files) oluşturulur. Sonra da bu amaç dosyalar birlikte bağlama işlemine sokularak çalıştırılabilen dosya elde edilir. Örneğin bir program iki "a.c" ve "b.c" isimli iki kaynak dosyasından oluşuyor olsun. Önce bu iki dosya bağımsız olarak derlenip "a.obj" ve "b.obj" biçiminde (UNIX/Linux sistemlerinde "a.o" ve "b.o" biçimimde) amaç dosyalar elde edilir, sonra bu amaç dosyalar birlikte bağlanarak çalıştırılabilen dosya oluşturulur:



Yukarıda da belirttiğimiz gibi elde edilen çalıştırılabilen dosyadaki mutlak adresler ya dosyanın başından itibaren göreli yer belirtilecekler ya da belli bir yükleme adresine göre gerçek doğrusal adresler olacaklardır. Biz burada bu amaç dosyaların nasıl birleştirildikleri üzerinde duracağız.

Birden fazla modülle proje geliştirirken bu modüller bağımsız olarak derlendiğinden derleyici bu modüllerdeki global nesnelerin ve fonksiyonların adreslerini dosyanın tepesinden itibaren bile oluşturamaz. Çünkü modüller diğer modüllerle birleştirildiğinde modüllerdeki bu göreli adresler bile geçersiz duruma gelir.



Şimdi basit bir biçimde çalıştırılabilen dosyadaki global nesne ve fonksiyon adreslerinin çalıştırılabilen dosyanın başından itibaren yer belirttiğini düşünelim. Pekiyi bu durumda amaç dosyaların içerisindeki global nesnelerin ve fonksiyonların göreli adresleri nereye göre bir yer belirtecektir? Eğer bunlar kendi amaç dosyalarının başından itibaren yer belirtirse bağlayıcı tarafından bu dosyalar birleştirildiğinde bu adresler geçersiz durumda olmaz mı?

İşte derleyiciler tipik olarak kodu derlerken global nesne ve fonksiyonların adreslerini boş bırakırlar (örneğin bunlar için 00000000 değerini yerleştirirler). Bağlayıcılar bu modülleri birleştirdikten sonra bu adresleri düzeltmektedir. Bu da bir çeşit yeniden konumlandırma işlemidir. Ancak burada yeniden kınumlandırma işlemini yapan yükleyici değil bağlayıcıdır. Ancak bu noktada hemen bir uyarıda da bulunalım: Modül birleştirmeleri sırasında yeniden konumlandırma sürecinin ayrıntıları sistemden sistemde farklılıklar gösterebilmektedir. Ayrıca yeniden konumlandırma işlemleri amaç dosya (object file) ve çalıştırılabilen dosya (executable file) formatlarına da belli ölçülerde bağlıdır. Biz burada yalnızca genel bir

fikir vermek istiyoruz.

Yeniden konumlandırma işleminin uygulanabilmesi için pek çok amaç dosya formatında amaç dosya içerisindeki iki tablodan faydalanılmaktadır. Bunlardan biri yine "yeniden konumlandırma tablosu (relocation table)", diğeri de "sembol tablosudur (symbol table)".

Ayrıntıları bir tarafa bırakırsak tipik olarak amaç dosyalara yönelik yeniden konumlandırma işlemleri şöyle yürütülmektedir:

1) Global bir nesne ya da fonksiyon kullanıldığında derleyici bunların adreslerini boş bırakır (tipik olarak bu adresler için derleyiciler 00000000 değerini yazmaktadır.)

2) Derleyici düzeltilecek adreslerin yerlerini (dosya içerisindeki offset'lerini) ve bu adreslerin hangi global değişken ya da fonksiyona ilişkin olduğunu amaç dosyanın yeniden konumlandırma tablosunda toplar. Yani amaç dosyanın yeniden konumlandırma tablosu kabaca "kodun neresi düzeltilecek ve bu düzeltilecek yer hangi değişkene ilişkindir" bilgilerini içermektedir. Program içerisindeki tüm global değişken ve fonksiyonlara ilişkin bilgiler (örneğin onların isimleri ve uzunlukları) "sembol tablosu (symbol table)" denilen bir tablonun içerisindedir. Yeniden konumlandırma tablosundaki adresin hangi değişken ya da fonksiyona ilişkin olduğu o değişken ya da fonksiyonun sembol tablosundaki indeks numarasıyla kodlamaktadır.

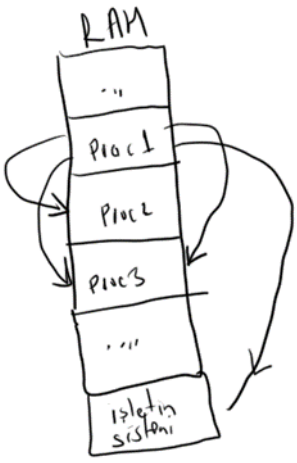
3) Bağlayıcı düzeltilecek adrese ilişkin değişkeni tüm amaç dosyaların sembol tablolarında arar. Onun tanımlamasının bulunduğu (yani onun yerinin ayrıldığı) amaç dosyayı bulur. Değişken için ayrılan yerin o amaç dosyanın neresinde olduğunu belirler. Sonra modülleri peşi sıra getirerek birleştirir ve adresleri de uygun biçimde düzeltir.

9. 32 Bit Intel İşlemcilerinde Koruma Mekanizması

Çokişlemli (multiprocessing) sistemlerde kullanılan modern ve güçlü mikroişlemcilerin çoğu bir koruma mekanizmasına (protection mechanisms) sahiptir. Intel 80286 ile birlikte segment tabanlı, 80386 ile birlikte de sayfa tabanlı koruma mekanizmasına sahip olmuştur. ARM işlemcilerinin pek çok modelinde, PowerPC, Itanium, SPARC gibi RISC tabanlı modern işlemcilerde de koruma mekanizmasına vardır.

Koruma mekanizmasının üç yönü vardır:

1) Bellek Koruması (Memory Protection): Çokprosesli sistemlerde tüm prosesler aynı fiziksel bellek üzerinde çalışırlar. İşte böyle bir çalışma sırasında bir prosesin (yani çalışan programın) kendi alanı dışına çıkarak başka proseslerin kullandığı bellek bölgelerine erişememesi gerekir. Aksi takdirde bir proses başka bir prosesin bellek alanını bozabilir ya da oradaki verileri çalabilir.



2) Komut Koruması (Instruction Proctection): Her prosesin her makine komutunu kullanması sistem

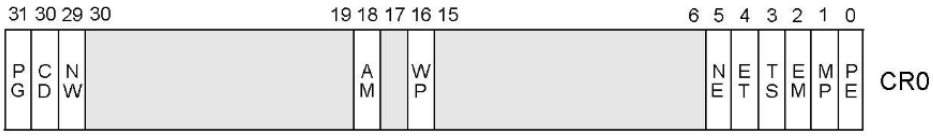
güvenliğini tehlikeye atabilmektedir. Çünkü bazı makine komutları eğer rastgele bir biçimde ya da özensiz olarak kullanılırsa sistemin çökmesine yol açabilir. Örneğin Intel işlemcilerindeki CLI (Clear Interrupt Flag) makine komutu işlemcinin kesme bayrağını resetelemektedir. Bu durumda işlemci donanım kesmelerine yanıt vermez. Bu ise tüm sistemin hemen çökmesine yol açabilecek bir durum oluşturur. CLI komutunun dışında tehlikeli olabilecek başka makine komutları vardır. Bu komutların yetkisiz ve sıradan prosesler tarafından kullanılmaması gerekir.

3) IO Koruması (IO Protection): Merkezi işlemci (yani CPU) pek çok yerel işlemciye bağlıdır ve onlara elektriksel olarak komutlar gönderebilmektedir. Yetkisiz ve sıradan bir proseslerin önemli IO portlarına komutlar göndermesi de sistemi çökertebilir. Bu nedenle sıradan bir prosesin önemli olabilecek IO portlarına erişiminin engellenmesi gerekir.

Şüphesiz koruma mekanizması bazı proseslere uygulanıp bazılarına uygulanmayacak biçimde esnek olmalıdır. Örneğin işletim sisteminin kodları koruma mekanizmasının denetiminden muaf olmak zorundadır. Çünkü işletim sistemi bir kaynak yöneticisidir ve kaynakları yönetirken de her türlü işlemi yapabilecek durumda olmalıdır. Benzer biçimde aygıt sürücüleri ve çekirdek modülleri de yaptıkları işin gereği olarak koruma mekanizmasından muaf olmak durumundadır.

Intel işlemcileri koruma mekanizması için 4 dereceli bir yetkilendirme modeline sahiptir. Ancak 4 dereceli yetkilendirmenin pratikte pek kullanışlığı olduğu söylenemez. Bu nedenle Intel işlemcilerini kullanan Windows gibi Linux gibi sistemler 4 yetki derecesi yerine yalnızca iki yetki derecesini kullanmaktadır. Benzer biçimde Intel dışındaki diğer işlemci aileleri de 2 dereceli bir yetki sistemine sahiptir. Bu yetki derecelerinin birine "çekirdek modu (kernel mode)" diğerine ise "kullanıcı modu (user mode)" denilmektedir. Pek çok ayrıntı söz konusu olsa da kabaca çekirdek modunda çalışan kodların hiçbir koruma engeline takılmadığını söyleyebiliriz. Ancak kullanıcı modunda çalışan kodlar için işlemciler katı bir koruma denetimi uygulamaktadır. İşletim sistemlerinin kodları, aygıt sürücüler, çekirdek modülleri "çekirdek modunda" çalışan kodlardır. Bunların dışındaki tüm programlar (örneğin Excel, Word gibi programlar ya da bizim yazdığımız programlar) "kullanıcı modunda" çalışırlar. Intel işlemcilerinde koruma mekanizmasının pek çok ayrıntısı vardır. Biz burada bu ayrıntıları belli bir derinlikte inceleyeceğiz.

Daha önceden de belirtildiği gibi Intel işlemcileri reset edildiğinde "gerçek mod (real mode)" denilen bir moddan çalışmaya başlar. Gerçek mod işlemcinin 1978 yılında tasarlanmış olan 8086 işlemcisi gibi çalıştığı moddur. (DOS işletim sisteminin ilk kez 8086 işlemcisi için yazıldığını anımsayınız.) Gerçek modda koruma mekanizması kullanılamamaktadır. Koruma mekanizmasının kullanılabilmesi için işlemcinin korumalı moda (protected mode) geçirilmesi gerekir. Intel işlemcilerinin korumalı moda geçirilmesi CR0 isimli bir kontrol yazmacının en düşük anlamlı bitinin 1 yapılmasıyla sağlanır. Biz bugüne kadar CR0 yazmacından hiç bahsetmedik. Çünkü bu yazmaç tamamen koruma mekanizmasıyla ilgili işlemlere yönelik bitlere sahiptir. CR0 yazmacının bitleri şöyledir:



Intel'de CR0 ve diğer kontrol yazmaçları diğer yazmaçlar gibi aritmetiksel ve bitsel işlemlere sokulamazlar. Bu yazmaçlar ancak başka genel amaçlı yazmaçlar ile MOV işlemine sokulabilmektedir. O halde işlemciyi korumalı moda geçirme işlemini aşağıdaki gibi bir kodla yapabiliriz:

```
mov     eax, cr0
or      eax, 1
mov     cr0, eax
```

Intel işlemcileri korumalı moda geçirildiğinde çalışma biçimlerinde önemli farklılıklar oluşmaktadır. Bu nedenle bunları korumalı moda geçirmeden önce bizim bazı hazırlıkları yapmış olmamız gerekir. Ayrıca Intel işlemcilerinde koruma mekanizmasını yalnızca koruma amacıyla kullanılan bir mekanizma olarak

düşünmek de doğru değildir. Tasarım gereği (geçmişe doğru uyumun korunması ile de ilgili olarak) bu işlemcilerin bazı özellikleri ancak koruma mekanizması aktive edildiğinde kullanılabilmektedir.

9.1. Segment Yazmaçlarının Korumalı Moddaki Anlamı

Anımsanacağı gibi 32 bir Intel işlemcilerinde CS, DS, SS, FS ve GS olmak üzere 16 bitlik beş segment yazmacı vardır. Segment yazmaçları gerçek modda ve V86 modunda 1 MB belleğe erişmek için offset'in yüksek anlamlı kısmını tutan bir görevdedir. Kursumuzda 16 bit gerçek moddaki çalışma ileride kısaca ele alınacaktır. Ancak bu segment yazmaçları işlemci korumalı geçirildiğinde tamamen farklı bir amaca hizmet eder hale gelirler. Bu nedenle korumalı modda segment yazmaçlarına ve onlara atanan değerlere "selector" de denilmektedir.

Segment yazmaçlarının korumalı moddaki genel yapısı şöyledir:

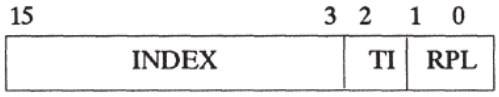
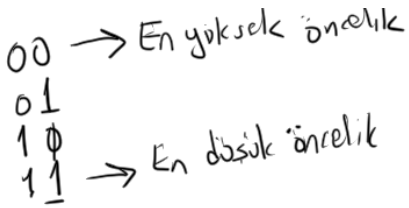


Figure 3.2 Segment Selector Format

Şekilden de segment yazmaçları bitsel olarak anlamlı üç bölümden oluşmaktadır. Bunları kısaca açıklayalım:

RPL (Requested Privilage Level): Bu segment yazmaçlarının düşük anlamlı iki bitidir. RPL bitleri öncelik belirtir. Genel olarak Intel sisteminde düşük numara daha yüksek öncelik belirtmektedir:

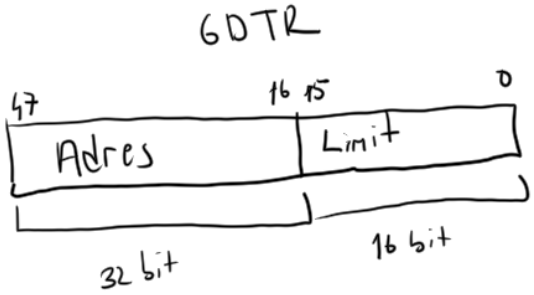


TI (Table Indicator): Segment yazmaçlarının 2 numaralı bitine TI biti denilmektedir. Bir selektör aslında "Global Betimleyici Tablosunda (GBT)" ya da "Yerel Betimleyici Tablosunda (YBT)" bir indeks belirtmektedir. İşte TI biti 0 ise bu selektör "Global Betimleyici Tablosunda", 1 ise "Yerel Betimleyici Tablosunda" bir indeks belirtir.

Index: Index alanı için 13 bit ayrılmıştır. 13 bit ile belirtilebilecek sayı sınırı [0, 8191] aralığındadır. İşte bu index TI ile belirtilen tabloda bir "betimleyici (descriptor)" belirtmektedir.

9.2. Global ve Yerel Betimleyici Tabloları

Korumalı moddaki en önemli tablolardan biri GBT'dir. İşlemci GBT'yi GDTR (Global Descriptor Table Register) yazmacının gösterdiği yerde aramaktadır. Yani sistem programcısı önce GBT'yi oluşturur, sonra onun adresini GDTR yazmacına yerleştirerek GDTR'nin GBT'yi göstermesini sağlar. Korumalı geçildiğinde GBT'nin bellekte hazır durumda olması ve GDTR'nin de GBT'yi gösterir durumda olması gerekir. GDTR yazmacı 48 bitlik bir yazmaçtır.



GDTR yazmacının düşük anlamlı 16 biti GBT'nin uzunluğunu belirtir. Yüksek anlamlı 32 biti ise tablonun bellekteki başlangıç adresini belirtmektedir. (Eğer sayfalama mekanizması aktif hale getirilmişse GDTR yazmacında belirtilen adres fiziksel adres değil doğrusal adres belirtmektedir.)

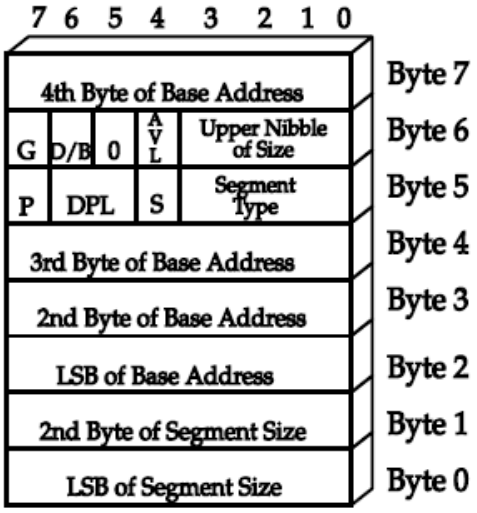
GBT'nin içerisinde ne vardır? GBT betimleyicilerden (descriptors) oluşmaktadır. Bir betimleyici 8 byte uzunluğundadır. Selektörün (yani segment yazmacının) yüksek anlamlı 13 bitinin betimleyici tablolarda offset değil indeks belirttiğine dikkat ediniz. Yani örneğin selektördeki indeks 3 ise aslında bu selektörün gösterdiği betimleyici GDTR yazmacının belirttiği adresten 3 * 8 = 24 byte ileridedir. GBT'de şu türden betimleyiciler bulunmaktadır:

- TSS (Task State Segment) betimleyicisi
- LDT (Local Descriptor Table) betimleyicisi
- Code betimleyicisi
- Data/Stack betimleyicisi
- Çağırma Kapısı (Call Gate) betimleyicisi
- Görev Kapısı (Task Gate) Betimleyicisi

Biz şimdiye kadar yalnızca Global Betimleyici Tablodan (GBT) bahsettik. Pekiyi Yerel Betimleyici Tablo (YBT) nedir ve nerede bulunmaktadır? YBT'yi LDTR (Local Descriptor Table Register) isimli bir yazmaç göstermektedir. Ancak LDTR yazmacı YBT'yi doğrudan göstermez. YBT'nin yeri aslında GBT içerisinde bulunan YBT betimleyicilerindedir. LDTR yazmacı YBT'nin adresini değil GBT içerisindeki ilgili LDT betimleyicisinin indeksini gösterir. Yani LDTR bir selektör gibidir:

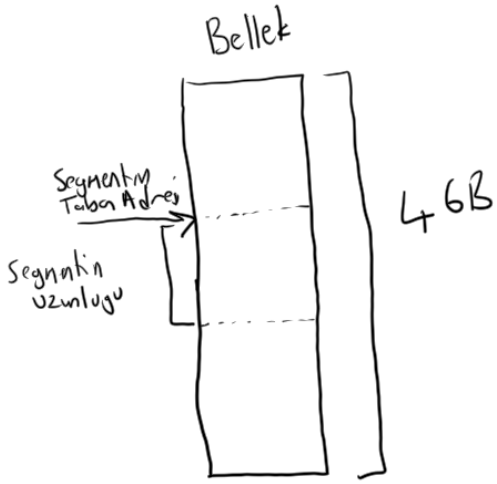
Bazı sistemlerde YBT hiç kullanılmaz yalnızca GBT kullanılır. Aslında Intel YBT'nin prosese özgü olmasını öngörmüştür. Yani GBT bir tanedir ancak her prosesin ayrı bir YBT'si olabilir. Proseslerrası geçiş işleminde LDTR yazmacının da değeri değiştirilerek prosesin kendi LDT'sini göstermesi sağlanabilmektedir. Biz şimdilik YBT kullanımı üzerinde durmayacağız.

GDT ve LDT içerisindeki betimleyicilerin genel formatı şöyledir:



Anahtar Notlar: Yukarıdaki şekil Intel gösterimine göre çizilmiştir. Intel bellek çizimlerinde düşük adresi aşağıda göstermektedir. (Düşük adresin fiziksel olarak da düşük yükseklikte olmasından hareketle). Halbuki biz kursumuzda bunun tam tersini yapıyoruz. Bizim çizdiğimiz bellek haritalarında düşük adresin daha yukarıda bulunduğuna dikkat ediniz.

Betimleyicinin düşük anlamlı 2 byte'ı ile 6'ıncı byte'ının düşük anlamlı 4 biti 20 bitlik segment uzunluğunu belirtmektedir. Buradaki uzunluk betimleyici içerisindeki G biti 0 ise byte cinsindendir, 1 ise sayfa (page) cinsindendir. G = 0 durumunda limitin 1 MB, G = 1 durumunda ise 4 GB olacağına dikkat ediniz. Segmentin uzunluğu limit kontrolü sırasında etki göstermektedir. Bu konuda ileride bilgi verilecektir. Betimleyicinin düşük anlamlı 2'inci, 3'üncü, 4'üncü ve 7'inci byte'ları segmentin taban adresini (base address) belirtir. Segmentin taban adresi segmentin bellekteki başlangıç adresini göstermektedir. Taban adres ve uzunluğun ardışıl bir bellek bölgesi elirttiğine dikkat ediniz.

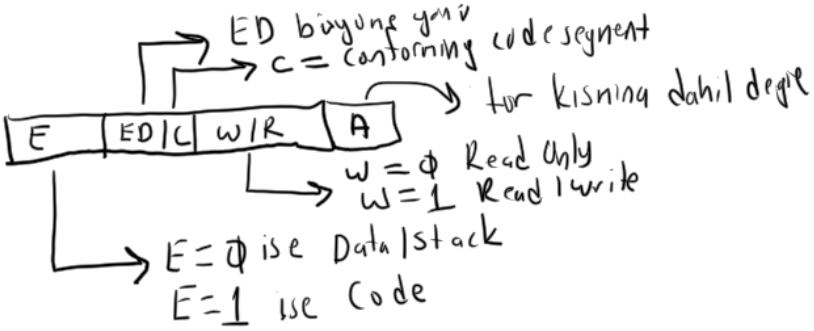


Betimleyicinin düşük anlamlı 5'inci byte'ının düşük anlamlı 4 biti betimleyicinin türünü belirtmektedir. Betimleyici türü ise kategori olarak S bitine bağlıdır. Eğer S = 0 ise bu durum betimleyicinin sistem betimleyicisi olduğunu gösterir. Eğer S = 1 ise betimleyici kod ya da data/stack betimleyicisidir.

S = 0 olması durumunda segment türleri 4 bit içerisinde şöyle kodlanmıştır:

Hex Value	Type Field (Bits 11:8)	Description
0	0000	Reserved (Illegal)
1	0001	Available 16-bit TSS
2	0010	LDT
3	0011	Busy 16-bit TSS
4	0100	16-bit Call Gate
5	0101	Task Gate
6	0110	16-bit Interrupt Gate
7	0111	16-bit Trap Gate
8	1000	Reserved (Illegal)
9	1001	Available 32-bit TSS
A	1010	Reserved (Illegal)
B	1011	Busy 32-bit TSS
C	1100	32-bit Call Gate
D	1101	Reserved (Illegal)
E	1110	32-bit Interrupt Gate
F	1111	32-bit Trap Gate

S = 1 durumunda 4 bitlik tür belirten alanın formatı da değişmektedir.



Biz bu tür konusunu burada daha fazla ayrıntılandırmayacağız. Ancak özet olarak şunları söyleyebiliriz: Türler sistem türleri (kapı türleri) ya da code ve data/stack türleri biçiminde iki sınıfa ayrılırlar. Eğer betimleyici code ve data/stack sınıfına ilişkinse o segmentin read/write özelliği de tür alanı içerisinde bulunmaktadır.

Betimleyicinin en önemli alanlarından biri de DPL'dir. DPL (Descriptor Privilege Level) alanı iki bitten oluşmaktadır. Bu iki bit hedef segmentin öncelik derecesini belirtir. Ve ileride ele alınacak olan bazı kontrollerde DPL kullanılmaktadır.

Betimleyicideki P (Present) biti segmentin o anda bellekte olup olmadığını belirtmektedir. Segment bellekte değilse işlemci içsel kesme oluşturur. Bu P biti segment tabanlı sanal bellek mekanizmasında kullanılmak üzere betimleyiciye dahil edilmiştir. 80286 işlemcilerinde sayfa tabanlı sanal bellek kullanılamıyordu fakat segment tabanlı sanal bellek kullanılabiliyordu. Bugün artık segment tabanlı sanal bellek kullanan sistemler kalmamıştır. Dolayısıyla buradaki P biti işletim sistemi geliştiricileri tarafından hep 1 durumunda tutulmaktadır.

Betimleyicideki AVL biti sistem programcısının kullanımına ayrılmıştır. Sistem programcısı bu bitleri istediği amaçla kullanabilmektedir.

9.3. Komutların Bellek Operandlarının Segment Yazmaçlarıyla İlişkisi

Bilindiği gibi sembolik makine dillerinde komutlardaki bellek operandları köşeli parantez ile belirtilmektedir. Köşeli parantez içerisinde nelerin bulunabileceği önceki konularda belirtilmişti. Burada kısa bir anımsatma yapmak istersek, köşeli parantezler içerisinde kabaca şunlar bulunabiliyordu:

- Sabit bit sayı. Örneğin [0x1FC1220] gibi.
- Bir yazma. Örneğin [EAX] gibi.
- Bir yazmaç ve sabit bir sayı toplamı. Örneğin [EAX + 0x1F] gibi.
- İki yazmaç toplamı. Örneğin [EAX + EBX] gibi.
- İki yazmaç ve bir sabit toplamı. Örneğin [EAX + EBX + 0x1F] gibi.
- Yukarıdaki yazmaçlı biçimlerde ayrıca yazmaçlardan biri 2, 4, 8 ile çarpılabilir. Örneğin [EAX + 2 * EBX] gibi.

Tabii buradaki yazmaçların her türlü yazmaç olamayacağını anımsatalım. 32 bit Intel işlemcilerinde ancak EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI yazmaçları köşeli parantez içerisinde bellek operandını belirlemek için kullanılabilmektedir.

Intel'de 16 bit mimariden beri her bellek operandı bir segment yazmacıyla ilişkildir. İlişki kuralı şöyledir:

1) Eğer bellek operandında EBP ve ESP varsa bu bellek operandı SS segment yazmacıyla ilişkilidir.
2) Eğer bellek operandında EBP ya da ESP yoksa bu bellek operandı DS segment yazmacıyla ilişkilidir.
3) Bazı string komutları ES segment yazmacıyla ilişkili olabilmektedir (string komutlarını anımsayınız).
4) PUSH ve POP komutları içsel olarak SS segment yazmacıyla ilişkilidir.

Şimdi bazı bellek operandlarının hangi segment yazmaçlarıyla ilişkili olduğuna yönelik birkaç örnek verelim:

```
[EAX]               --> DS
[EBP + EAX]         --> SS
[0x1FC1245]         --> DS
[ESP + EAX + 12]    --> SS
PUSH EAX            --> SS
```

Her ne kadar bellek operandlarının ilişkin olduğu default segment yazmaçları varsa da Intel'de "segment overriding" denilen önekleme ile bu default durum değiştirilebilmektedir. Segment öneklemesi sembolik makine dillerinde ':' ile belirtilmektedir. Örneğin:

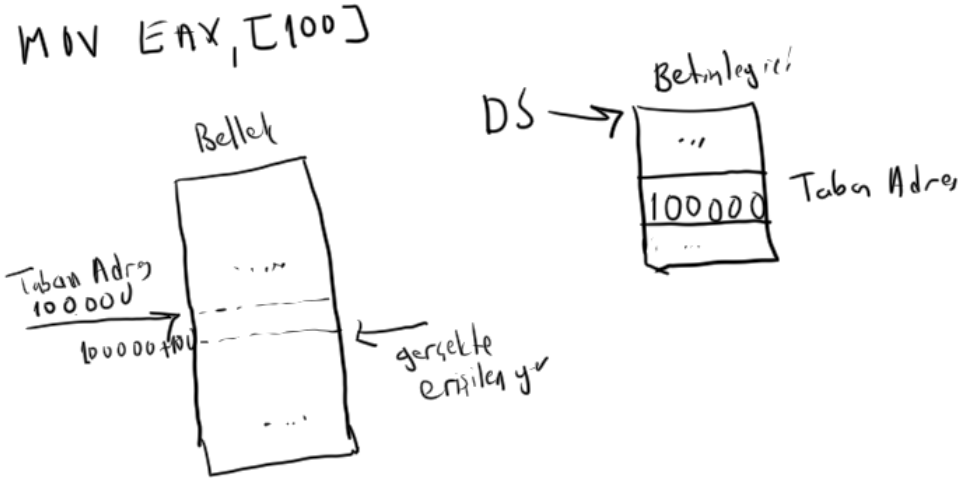
```
MOV EAX, SS:[EBX + 10]
```

Burada [EBX + 10] bellek operandının lişkin olduğu segment yazmacı default durumda DS'dir. Ancak segment öneklemesi (segment overriding) yapılarak bunun SS olması sağlanmıştır. Örneğin:

```
MOV FS:[EAX + EBP], EBX
```

Burada da default segment yazmacı SS iken bu erişim için FS yapılmıştır.

Şimdi gelelim bellek operandlarıyla segment yazmaçlarının ilişkisine. Aslında Intel'de 16 bit sistemden beri köşeli parantez içerisindeki bellek adresleri belleğin tepesinden itibaren bir yer belirtmemektedir. Belli bir taban adresten itibaren bir yer belirtmektedir. Başka bir deyişle aslında köşeli parantez içerisindeki efektif adresler (yani toplam durumundaki sonuç adresler) bir taban değerden itibaren offset belirtirler. İşte bu taban değer o bellek operandının ilişkin olduğu betimleyici içerisindeki taban adrestir. Örneğin MOV EAX, [100] gibi bir komutun uygulandığını düşünelim. Bu komuttaki default segment yazmacı DS'dir. DS'nin de gösterdiği betimleyicinin içerisindeki taban adresin 100000 olduğunu varsayalım. Bu durumda bu bellek operandı belleğin tepesinden itibaren 100'ün byte'a değil 100000'den itibaren 100'üncü byte'a (yani 100100 byte'ına) erişecektir.



Pekiyi betimleyici içerisindeki uzunluk (limit) alanı ne anlama gelmektedir? İşte betimleyicideki taban adres bir başlangıç noktası belirtirken uzunluk da o segment yazmacı ile o taban adresten itibaren en fazla hangi uzaklığa erişilebilineceğini belirtir. Örneğin betimleyicideki uzunluk (limit) bilgisinin 1 MB olduğunu varsayalım. Biz artık köşeli parantez içerisinde 1 MB'den daha büyük bir efektif adres kullanamayız. Eğer kullanırsak işlemci bir kesme oluşturarak ihlal yapan prosesi işletim sistemine bildirecektir. Görüldüğü gibi betimleyicideki uzunluk (limit) belli bir taban adresten istenildiği kadar gidilebilmesini engellemektedir.

Her ne kadar betimleyicideki taban ve limit alanlarının işlevi yukarıda açıklandığı gibiyse de Windows gibi Linux gibi işletim sistemleri oluşturdukları betimleyicilerin hep taban adreslerişni sıfır ve uzunluklarını da 4 GB yapmaktadır. Böylece Windows ve Linux gibi sistemlerinde segment yazmaçlarının taban adreslerinin ve limitlerinin bir etkiye yol açmadığı söylenebilir. Bu yönteme Intel terminolojisinde "flat model" denilmektedir. Yani biz Windows ve Linux sistemlerinde hangi segment yazmacını kullanırsak kullanalım bunun taban adres ve limit üzerinde bir etkisi olmayacaktır. Başka bir deyişle bu sistemlerde bellek operandları her zaman belleğin tepesinden itibaren bir yer belirtir durumdadır.

Pekiyi Windows gibi Linux gibi işletim sistemlerinde bir program çalıştırıldığında segment yazmaçlarının başlangıçtaki değerleri nedir? Bu sistemlerde işletim sisteminin yükleyicileri prosesi oluştururken onlar için GBT'de birer betimleyici oluşturup bu segment yazmaçlarının bu betimleyicileri göstermesini sağlarlar. Biz sıradan bir kullanıcı modu programcısı olarak bir daha bu segment yazmaçlarının değerini zaten değiştiremeyiz. Ayrıca GBT ya da YBT tablolarına da yeni bir betimleyici ekleyemeyiz. Hatta bu sistemler her proses için GBT'de ayrı birer segment betimleyicisi bile tutmazlar. Kod ve data/stack için birer segment betimleyicisi bulundurup tüm proseslerde bunları kullanırlar. Yani bu sistemlerde değişik proseslerde segment yazmaçlarının aynı değerde olduğunu görürseniz şaşırmayın.

9.4. CPL (Current Privilege Level) Kavramı

Anımsanacağı gibi programın o andaki çalıştığı kod EIP yazmacı tarafından gösterilmektedir. Aslında CALL gibi, JMP gibi komutlar EIP yazmacının değerini değiştirirler. EIP yazmacının da CS segment yazmacı ile ilişkili olduğunu belirtmiştik. İşte CS yazmacının düşük anlamlı 2 bitine CPL denilmektedir. (Diğer segment yazmaçlarının düşük anlamlı 2 bitine RPL denildiğini anımsayınız.)

CPL bazı test işlemlerine çalışmakta olan kodun öncelik derecesini belirten bir değer olarak sokulmaktadır. Genel olarak söylersek, CPL değeri 0 olan kodlar herhangi bir koruma engeline takılmazlar. Bu kodlar her

makine komutunu kullanabilirler ve doğrusal adres alanındaki her bellek sayfasına erişebilirler. Bu nedenle CPL değeri 0 olan kodlara "çekirdek modunda (kernel mode)" çalışan kodlar denilmektedir. Daha önceden de belirtildiği gibi Intel mimarisi çalışan kodlar için 4 öncelik derecesi sunsa da Windows gibi, Linux gibi sistemler yalnızca iki öncelik derecesini kullanmaktadırlar: 0 ve 3. CPL değeri 3 olan programlara "kullanıcı modunda (user mode)" çalışan programlar denilmektedir. Windows ve Linux sistemlerinde yalnızca işletim sisteminin kendi kodları ve aygıt sürücülerin kodları CPL = 0 değeri ile (yani çekirdek modunda) çalıştırılmaktadır. Bunların dışındaki bütün kodlar CPL= 3 değeriyle (yani kullanıcı modunda) çalıştırılırlar.

9.5. Segment Yazmaçlarına Yüklemelerinde Öncelik Kontrolleri

Bir segment yazmacına MOV komutuyla yeni bir değer (selektör) yüklenirken bazı kontroller yapılmaktadır. Eğer böyle olmasaydı bu segment yazmaçlarının RPL ve betimleyicilerin DPL bitlerinin bir önemi kalmazdı. Ayrıca CS yazmacına MOV komutu ile atama yapılamadığını anımsatalım. CS yazmacı stack'e push edilebilir, ancak POP edilemez. İleride de bahsedileceği gibi CS yazmacının değerinin değiştirilmesinin yalnızca üç yolu vardır.

DS, SS, ES, FS ve GS yazmaçlarına atama yapılabilmesi için o anda çalışmakta olan kodun CPL değeriyle segment yazmacına atanacak selektörün RPL değerinin en düşük önceliklisi (yani en yüksek değeri) atanacak selektörün belirttiği DPL değerinden ya daha öncelikli ya da onunla eşit öncelikli olmak zorundadır. Aksi durumda "GP (General Protection Fault)" isimli içsel kesme ile atama başarısızlıkla sonuçlanır. Erişim kontrolünü nümerik olarak (öncelik olarak değil) şöyle ifade edebiliriz:

```
max {CPL, Yüklenecek selektörün RPL'si} <= Yüklenecek betimleyicnin DPL'si
```

Örneğin aşağıdaki gibi iki komutla DS yazmacının değerini değiştmek isteyelim:

```
mov ax, 0x003B
mov ds, ax
```

Bu işlemi yapan kodun CPL değeri 3 olsun. İşte işlemci öncelikle 0x3B ile belirtilen selektörün geçerliliğini kontrol eder. Bu selektörün var olan bir betimleyiciyi gösteriyor olması gerekir. Bundan sonra işlemci DS'ye atanacak selektör olan 0x3B'nin RPL bitlerine bakar. 0x3B selektörü için RPL = 3'tür. Şimdi bu 0x3B selektörünün gösterdiği betimleyicinin DPL bitlerinin 0 olduğunu düşünelim. İşte CPL ve RPL'nin en düşük önceliklisi 3 olduğuna göre bunun yüklenecek selektörün DPL'sinden daha öncelikli ya da eşit öncelikli olması gerekir. Böyle olmadığı için GP hatası oluşacaktır.

Şimdi 0x38 numaralı selektörü DS yazmacına yüklemek isteyelim. Bu selektörün gösterdiği betimleyicinin DPL'si 3 olsun. DS'yi yüklemek isteyen kodun da CPL'sinin 3 olduğunu varsayalım. Yükleme sırasında GP oluşur mu? Yanıt hayır, oluşmaz. Şöyle ki:

max {CPL = 3, Yüklenecek selektöre ilişkin RPL = 0} <= Yüklenecek selektöre ilişkin betimleyicinin DPL'si = 3

Görüldüğü gibi koşul sağlanmaktadır. Pekiyi bu durumda yüklenecek selektöre ilişkin RPL'nin test işlemine girmesinin bir anlamı var mıdır? Şöyle ki: Bizim CPL'miz 0 olsun, biz de DPL'si 0 olan bir betimleyiciyi RPL'si 3 olan bir selektörle yüklemek isteyelim. Bu durumda GP (General Proctection Fault) kesmesi oluşacaktır. Fakat biz yüklemek istediğimiz selektörün tablo indeksi ve TI biti aynı kalacak biçimde yalnızca onun düşük anlamlı 2 bitini kendimize uydurarak yüklemeyi sağlayabiliriz. İşte bu nedenden dolayı aslında yüklenecek selektöre ilişkin RPL değerinin kontrolde ciddi bir anlamı yoktur. Ancak Intel güvenliği artırmak için (yani bozulmuş kodların koruma engeline takılması için) bu ek kontrolü sisteme dahil etmiştir. Fakat buradan basit bir sonuç çıkmaktadır: Bizim CPL'miz yüklenecek selektörün belirttiği DPL'den düşük öncelikli ise biz RPL'yi ayarlasak bile segment yazmaçlarını bu selektörle yükleyemeyiz. DS, SS, ES, FS ve GS segment yazmaçlarının yüklenmesinde önemli olan unsur o andaki kodun CPL'si ve yüklenecek selektörün belirttiği betimleyicinin DPL'sidir.

Aslında bugün yoğun kullandığımız Windows ve Linux sistemlerinde zaten çalışmakta olan programın

segment yazmaçlarının yüklenme gerekliliği yoktur. Intel'in segment yazmaçlarına yükleme yapılırken uyguladığı bu kontrol segment tabanlı modeller için düşünülmüştür. Oysa Windows ve Linux sistemleri zaten “flat model” kullanmaktadır. Bu modelde zaten belleğin her yerine mevcut betimleyicilerle erişilebilmektedir. Yani bu sistemlerde bir yere erişmek için segment yazmaçlarının yüklenmesi gibi bir durum söz konusu değildir. Bugün Windows ve Linux sistemlerinde segment tabanlı değil sayfa tabanlı bir koruma modeli uygulanmaktadır. Sayfalama mekanizması ve sayfa tabanlı koruma modeli ileride ele alaınacaktır.

CS yazmacının değiştirilmesi ister segment tabanlı isterse sayfa tabanlı koruma modeli söz konusu olsun kritik önemdedir. Yukarıda da belirttiğimiz gibi Intel sisteminde CS segment yazmacının değerini MOV komutuyla ya da POP komutuyla değiştiremeyiz. (CS yazmacı push edilebilmektedir ancak POP edilememektedir.) CS yazmacının değiştirilmesinin dört yolu vardır:

1) Segmentli CALL komutlarıyla. unlara Intel uzak (far) call komutları da demektedir.

2) RETF ya da IRET komutları yoluyla

3) Segmentli JMP komutlarıyla. Bunlara Intel uzak (far) jump komutları demektedir.

4) Kapılar (gates) yoluyla

Segment belirtilerek yapılan CALL ve JMP işlemlerine uzak (far) CALL ve JMP işlemleri denilmektedir. Assembly’de uzak CALL ve JMP komtları segment ve offset arasına ‘:’ getirilerek gösterilirler. Örneğin:

```
CALL    0x002B : 0x10000
```

gibi. Tabii "dolaylı uzak (indirect far)" CALL ve JMP işlemleri de uygulanabilir:

```
CALL    FAR     [EBX]
```

Bu komutta EBX yazmacının gösterdiği yerdeki düşük anlamlı 4 byte offset (yani EIP’ye yerleştirilecek değer), yüksek anlamlı 2 byte ise selektör (yani CS’ye yerleştirilecek değer) belirtir.

Segment’li uzak CALL ve JMP işlemleri CS’yi dolayısıyla da CPL’yi değiştirme iddiasındadır. Öncelikle kod segment betimleyicilerinin “conforming” ya da “non-conforming” biçiminde ikiye ayrıldığını belirtelim. Çünkü uzak dallanmalar sırasında yapılan kontroller dallanmada kullanılan selektörün gösterdiği betimleyicinin “conforming" kod segment betimleyicisi mi, yoksa “non-conforming" kod segment betimleyicisi mi olduğuna göre değişmektedir. "Conforming" ya da "non-conforming" olma durumunun selektörün gösterdiği betimleyicinin içerisinde kodlandığına dikkat ediniz.

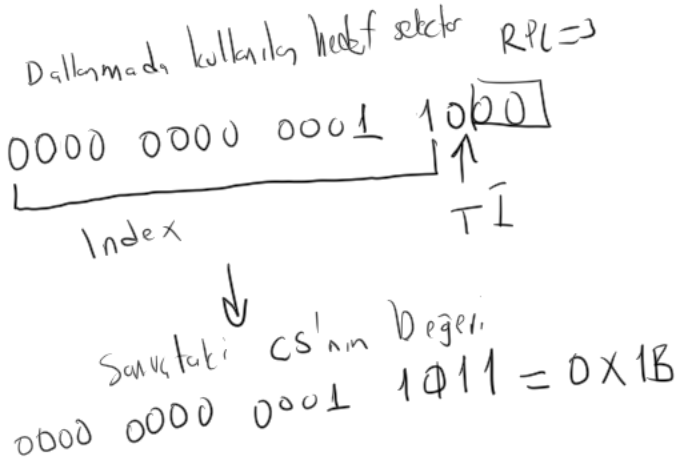
“Non-conforming” kod segment betimleyicilerini gösteren selektörler ile yapılan JMP ve CALL işlemlerinde şu kontroller uygulanmaktadır:

1) Çalışmakta olan kodun CPL değerinin dallanmada kullanılan hedef selektörün belirttiği DPL değerine eşit olması gerekir (ondan büyük ya da küçük olamaz).

2) Dallanılacak selektöre ilişkin RPL değerinin CPL’den daha öncelikli olması ya da eşit öncelikli olması gerekir. (Yani nümerik olarak ifade edersek dallanılacak selektöre ilişkin RPL’nin CPL’den küçük ya da ona eşit olması gerekir.)

Eğer bu koşullar sağlanıyorsa JMP ya da CALL işlemi yapılır. Ancak CS segment yazmacının CPL değeri değişmez, yalnızca dallanılacak selektörün "TI" ve "Index" bitleri CS’ye geçer. Örneğin bizim kodumuzun CPL’si 3 olsun. Biz de 0x0018 numaralı selektörü kullanarak uzak CALL işlemi yapmak isteyelim. 0x0018 numaralı selektörün belirttiği RPL değerinin 0 olduğunu görüyorsunuz. Bu selektörün gösterdiği kod segment betimleyicisinin DPL değerinin de 3 olduğunu varsayalım. Bu durumda yukarıdaki iki madde de

sağlanacaktır. Bu işlem sonucunda CS yazmacının düşük anlamlı 2 biti durum değiştirmeyecektir. Ancak "TI" ve "Indeks" bitleri 0x0018 selektöründen alınacaktır.



Dallanma sonrasında CS'nin değeri 0x001B olacaktır.

"Conforming" kod segment betimleyicisi ile yapılan JMP ve CALL işlemlerinde dallanılacak hedef selektöre ilişkin RPL değeri kontrol işlemine sokulmamaktadır. Dallanmanın başarılı olabilmesi için CPL'nin dallanmada kullanılan hedef selektörün gösterdiği betimleyicinin DPL değerinden daha düşük öncelikli ya da onunla eşit öncelikli olması gerekir. Yani nümerik olarak dallanmanın yapılabilmesi için CPL >= DPL olmak zorundadır. Görüldüğü gibi "conforming" kod segmente dallanılabilmesi için CPL'nin dallanılacak segmentin DPL'sinden daha az öncelikli ya da ona eşit öncelikli olması gerekir. Fakat kontrolden geçildiğinde yine kodun CS selektöründeki CPL değeri değiştirilmemektedir. Yani özetle biz yüksek öncelikli "conforming" bir kod segmente dallanma yapabiliriz ancak bu dallanma sorasında kodumuzun önceliği değiştirilmemektedir.

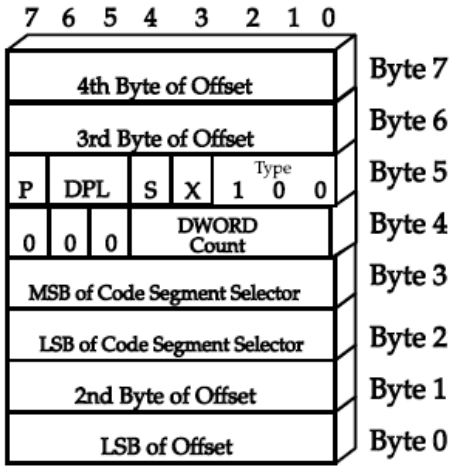
Yukarıdaki "non-conforming" ve "conforming" kod segment dallanmalarında hiçbir biçimde çalışmakta olan kodun CPL değerinin değiştirilmediğini bir kez daha vurgulamak istiyoruz. Çalışmakta olan kodun önceliğinin (yani CPL'sinin) değiştirilmesinin tek yolu kapı (gate) kullanmaktır.

9.6. Kapılar (Gates)

Intel işlemcilerinde çalışmakta olan kodun öncelik derecesini (yani CS yazmacının CPL bitlerini) değiştiren mekanizmaya kapı (gate) denilmektedir. Kapıların pek çok türü vardır. Ancak burada biz şimdilik "çağırma kapısı (call gate)" üzerinde duracağız. Çağırma kapılarına CALL ve JMP komutları ile dallanılabilmektedir.

Çağırma kapısı GBT (Global Betimleyici Tablosu) ya da YBT (Yerel Betimleyici Tablosu) içerisinde bir betimleyici biçiminde bulunmaktadır. Bu betimleyiciye "çağırma kapısı betimleyicisi (call gate descriptor)" denilmektedir. Çağırma kapısı uzak CALL ya da JMP komutlarıyla devreye sokulmaktadır. Çağırma kapısına uzak CALL ya da JMP işlemi yapılırken komutta belirtilen selektörün GBT ya da YBT'deki bir bir çağırma kapısı betimleyicisini göstermesi gerekir. Komuttaki offset herhangi bir değerde olabilir. Komuttaki offset komut tarafından kullanılmamaktadır. Çağırma kapısı betimleyicisinin genel formatı şöyledir:

Call Gate Descriptor Format



Çağırma kapısı betimleyicisinin içerisinde şu alanlar vardır:

- Hedef kod segment selektörü (CS).
- Hedef komut yazmacı offseti (EIP değeri).
- Betimleyicinin DPL'si (öncelik derecesi).
- Stack değişimi için (stack switch) kullanıcı stack'inden (user stack) kopyalanacak DWORD miktarı.

Çağırma kapısına uzak CALL ya da JMP komutlarıyla dallanma işlemi sırasında bazı kontroller yapılmaktadır. Yapılan kontroller CALL ve JMP komutları arasında ve çağırma kapısındaki CS selektörünün "conforming" olup olmamasına göre bazı küçük farklılıklar içermektedir. Genel olarak çağırma kapısına dallanılırken kontrole giren öğeler şunlardır:

- Dallanma işlemini yapan kodun CPL değeri.
- Kapıya uzak CALL ya da uzak JMP yapılırken kullanılan selektörün RPL değeri.
- Çağırma kapısı betimleyicisinin DPL değeri.
- Çağırma kapısı betimleyicisinin içerisindeki CS selektörünün belirttiği kod segment betimleyicisinin DPL değeri.

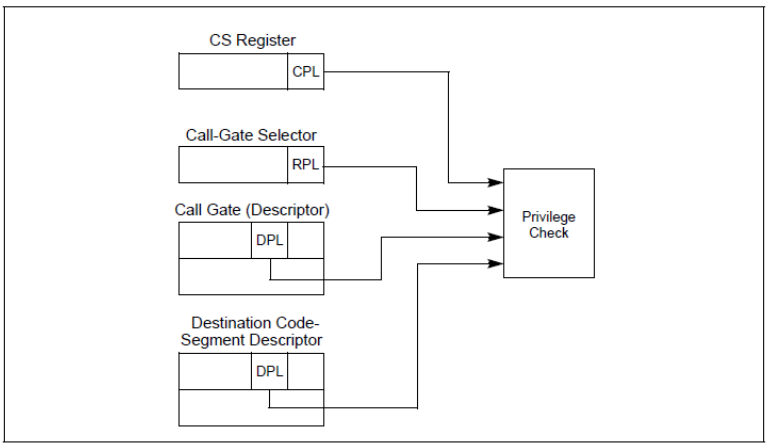


Figure 5-11. Privilege Check for Control Transfer with Call Gate

Çağırma kapısına yapılan uzak CALL işlemi sırasında çağırma kapısı betimleyicisi içerisindeki selektörün gösterdiği kod segment betimleyicisinin "conforming" olup olmamasına bakılmaksızın şu kontroller uygulanmaktadır.

1) Çağırma işlemini yapan kodun CPL değerinin ve CALL komutunda kullanılan selektörün RPL değerinin

her ikisinin de komuttaki selektörün gösterdiği çağırma kapısı betimleyicisinin DPL değerinden daha yüksek öncelikte ya da onunla aynı öncelikte olması gerekir. Yani nümerik olarak kontrolü şöyle ifade edebiliriz:

(CALL işlemini uygulayan kodun CPL değeri) <= (çağırma kapısı betimleyicisinin DPL değeri) && (çağırmada kullanılan selektörün RPL değeri) <= (Çağırma kapısı betimleyicisinin DPL değeri)

Bu kontrol bize şunu anlatmaktadır: Bizim çağırma kapısına dallanabilmemiz için çağırma kapısı betimleyicisinin DPL değerinden daha öncelikli olmamız ya da onunla eşit öncelikli olmamız gerekir. Başka bir deyişle biz daha yüksek önceliğe sahip bir çağırma kapısına dallanamayız.

2) Çağırma kapısı betimleyicisinin içerisindeki selektörün (yani CS'ye yüklenecek değerin) gösterdiği betimleyicinin DPL değerinin o anda çalışmakta olan kodun CPL değerinden daha öncelikli ya da onunla eşit öncelikli olması gerekir. Nümerik olarak kontrol şöyle ifade edilebilir:

(CALL işlemini uygulayan kodun CPL değeri) >= (çağırma kapısı betimleycisinin içerisindeki selektörün (yani hedef CS'nin) belirttiği kod segment betimleyicisinin DPL değeri)

Çağırma kapısına uzak JMP ile dallanılırken birinci maddede belirtilen kontrol yine aynı biçimde uygulanır. Ancak ikinci maddede küçük bir kontrol değişikliği söz konusudur:

2) Eğer çağırma kapısı içerisindeki selektörün gösterdiği kod segment betimleyicisi "conforming" bir kod segment betimleyicisi ise bu durumda bu betimleyicinin DPL değerinin CPL değerinden daha öncelikli olması gerekir. Ancak kod segment betimleyicisi "non-conforming" ise bu durumda hedef betimleyicinin DPL değerinin çalışmakta olan kodun CPL değerine eşit olması gerekir.

Eğer uzak CALL işlemi ile çağırma kapısına yapılan dallanmalarda kontrollerden başarılı olarak geçilirse CS yazmacı çağırma kapısı içerisindeki selektör ile yüklenir. Ancak CS yazmacının düşük anlamlı iki biti olan CPL değeri buradaki selektörden değil çağırma kapısı betimleyicisi içerisindeki bu selektörün gösterdiği kod segment betimleyicisinin DPL'sinden alınarak oluşturulmaktadır. (Çağırma kapısı içerisindeki CS selektörünün RPL değerinin herhangi bir kontrole sokulmadığına dikkat ediniz. Dolayısıyla bu selektörün RPL değerinin bir önemi yoktur). CS yüklendikten sonra EIP yazmacı da çağırma kapısı içerisindeki dört byte'lık EIP alanınındaki adres ile yüklenir. Bundan sonra ileride ele alınacağı gibi stack değişimi yapılıp akış bu yeni yüklenen CS:EIP adresinden devam edecektir.

Intel'de uzak JMP komutu ile çağırma kapısına dallanma yapıldığında öncelik yükseltmesi yapılmamaktadır. Bu nedenle uzak JMP ile çağırma kapısına dallanmanın pratikte öncelik yükseltme için bir etkisi yoktur.

Biz burada yalnızca çağırma kapısı (call gate) üzerinde durduk. Çağırma kapısı yalnızca CALL (ve JMP) makine komutlarıyla tetiklenmektedir. Halbuki "kesme kapısı (interrupt gate)" ve "tuzak kapısı (trap gate)" isimli iki kapı türü daha vardır. Bu kapılar kesme işlemleriyle tetiklenirler. Ancak kullanım amaçları ve kullanım sırasında uygulanan kontroller çok benzerdir.

Kapı konusunu tek bir cümleyle özetleyecek olursak şunları söyleyebiliriz: Biz az öncelikli bir kod olarak bir kapıya dallandığımızda kendimizi önceliğimiz yükseltilmiş olarak o kapı betimleyicisinin içerisinde belirtilen adreste buluruz.

9.6.1. Kapılardan Geri Dönüş

Çağırma kapısından geriye uzak RET (RETF) komutuyla dönülür. Kapıya uzak CALL ile dallanıldığında stack'e yalnızca EIP yazmacının değeri değil aynı zamanda CS yazmacının değeri de push edilmektedir. RETF makine komutu stack'teki eski CS ve EIP değerini alarak kodun kalınan yerden devamını sağlar. Çağırma kapısından dönüş için RETF uygulandığında RETF yüksek bir öncelikten düşük önceliğe geçiş yapıldığını doğrular ve eski CS'yi (dolayısıyla CPL'yi) geri yükler. İleride de görüleceği gibi kesme ve tuzak kapılardan geri dönüş de IRET makine komutuyla yapılmaktadır.

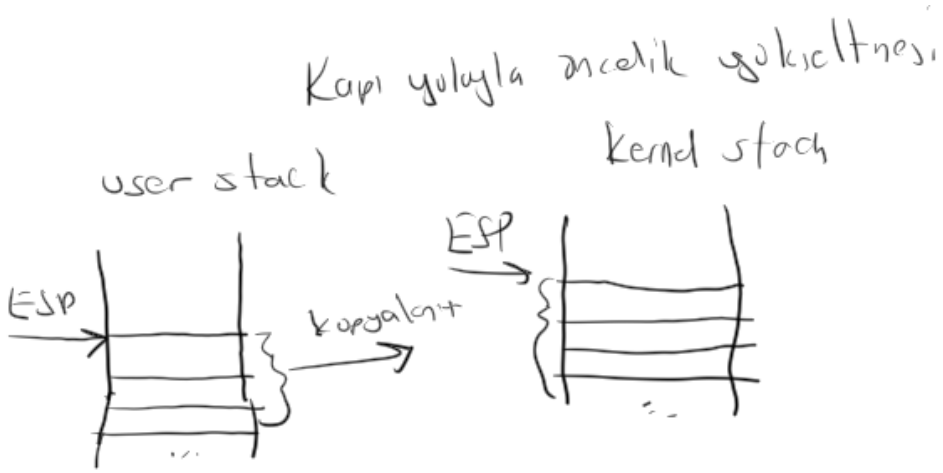
9.6.2. Kapılara Dallanma Sırasında Stack Değişimi (Stack Switch)

Kapı yoluyla yüksek öncelikli bir biçimde güvenli kodlara dallanıldığında stack'in düşük öncelikli kod bölgesinde kalmasının bazı potansiyel tehlikeleri vardır. Bunu anlamak için kapı yoluyla işletim sisteminin bir sistem fonksiyonunu çağırdığımızı düşünelim. Eğer işletim sisteminin CPL değeri 0 ile çalıştırılan güvenli kodlarının kullandığı stack "kullanıcı modunda (user mode)" kalırsa şu tehlikeler söz konusu olabilmektedir:

1) Kullanıcı modundaki (CPL = 3) prosesin başka bir thread'i bu stack'i yanlışlıkla bozabilir. Bu durumda sistem fonksiyonları da aynı stack'i kullandığı için onların çalışması bozulur. Bundan da tüm sistem etkilenebilir.

2) Kullanıcı modunda (CPL = 3) kullanılan stack bir biçimde taşabilir. Bu da yine sistem fonksiyonun kesilmesine ve tüm sistemin çökmesine yol açabilir.

İşte bunun engellenmesi için kapı yoluyla öncelik yükseltmesi sırasında otomatik "stack değişimi (stack switch)" yapılmaktadır. Yani biz bir kapı yoluyla kodumuzun önceliğini yükselterek bir noktaya dallandığımızda artık kullanılan stack de otomatik olarak değiştirilmektedir. Başka bir deyişle kapıya dallanma sırasında SS ve ESP yazmaçları artık güvenli bir stack'i gösterecek hale getirilmektedir. Örneğin biz kapı yoluyla bir sistem çağırması yapmış olalım. Sistem fonksiyonu çalışırken artık onun kullandığı stack bizim stack'imiz olmayacaktır. İşletim sisteminin sağladığı güvenli bir stack olacaktır. Tabii stack değişimi otomatik olarak yapılırken düşük öncelikli koda ilişkin stack'in tepesindeki belli bir bölümün yüksek öncelikli stack'e kopyalanması da gerekir. Çünkü sistem fonksiyonları gibi yüksek öncelikli fonksiyonları çağıran kodlar parametre aktarımını stack yoluyla yapıyor olabilirler. Bu durumda o parametrelerin de yeni stack'e taşınması gerekecektir.



Pekiyi otomatik stack değişiminde yeni stack'in yeri nasıl belirlenmektedir? İşte henüz açıklamadığımız bir veri yapısı daha bu noktada devreye girmektedir. Buna TSS (Task State Segment) denilmektedir. Bir kod çalışırken o kodun ilişkin olduğu bir TSS alanı vardır. Kapıya dallanma yapıldığında yeni stack'in yeri o kodun ilişkin olduğu TSS alanında belirtilmektedir. Eski stack'ten yeni stack'e kaç byte kopyalanacağı bilgisinin çağırma kapısı betimleyicisinin içerisinde bulunduğunu anımsayınız.

9.6.3. İşletim Sisteminin Sistem Fonksiyonları ve Kapılar

Korumalı modda çalışan Windows, Linux ve Mac OS X gibi işletim sistemlerinde sıradan proseslerin kodları CPL = 3 önceliğinde çalışmaktadır. Bu kodlar işletim sisteminin yüksek öncelikle çalışması gereken sistem fonksiyonlarını kapılar yoluyla çağırırlar. Böylece işletim sisteminin sistem fonksiyonları çalışırken kodun önceliği CPL = 0'a yükseltilmiş olur. Bu sürece "prosesin kullanıcı modundan çekirdek moduna

geçmesi (user mode to kernel mode transition)" denilmektedir. Yani bu sistemlerde bizim programlarımız aslında sürekli olarak CPL = 3 ile kullanıcı modunda çalışmamaktadır. Sistem fonksiyonları ya da aygıt sürücülerdeki kodlar çağrıldığında programımızın öncelik seviyesi geçici olarak CPL = 0'a yükseltilmektedir. İşte kapılar Intel işlemcilerindeki bu geçişi sağlayan mekanizmalardır.

Linux, BSD ve Mac OS X sistemlerinde sistem fonksiyonları geleneksel olarak 80h kesmesi yoluyla çağrılmaktadır. (Yeni sistemler 64 bit Intel işlemcilerindeki SYSENTER ve SYSEXIT makine komutlarını da bu amaçla kullanabiliyorlar. Bu komutlar 64 bit çalışmanın anlatıldığı bölümde ele alınmaktadır.) Bu 80h kesmesi bir tuzak kapısını tetikler. Bu kapı da kodun önceliğini CPL = 0'a çekerek kodun işletim sisteminin belirlediği bir noktaya aktarılmasını sağlar. İşte o noktada çağrılan sistem fonksiyonunun numarasına göre akış ilgili sistem fonksiyonun koduna aktarılmaktadır. Örneğin Linux sistemlerinde sistem fonksiyonu 80h kesmesi ile çağrılmadan önce onun numarası EAX yazmacına yerleştirilir. Böylece akış çekirdek moduna geçtiğinde buradaki kod EAX yazmacının değerine bakarak akışı uygun yere aktarır. Bu süreci aşağıdaki kodla temsil edebiliriz:

```
SYS_ENTER:  // kapıya girildiğinde akışın aktarıldığı yer. Artık kod için CPL = 0'dır
    switch (eax) {
        case 1:
            sys_exit();
            break;
        case 2:
            sys_fork();
            break;
        case 3:
            sys_read();
            break;
        ...
    }
```

Tabii biz bu sözde kodu (pseudo code) yalnızca kafanızda bir fikir oluşsun diye verdik. Aslında Linux'ta uygun sistem fonksiyonuna dallanma işlemi EAX yazmacı switch içerisine sokularak değil bir diziye index yapılarak bir "look up" tablosu yoluyla gerçekleştirilmektedir. Yani bu sistemlerde sistem fonksiyonlarının adresleri bir dizide tutulmaktadır. Sistem fonksiyonlarının numarası da (EAX yazmacı içerisindeki değer) bu diziye indeks yapılarak dolaylı CALL işlemi ile çağrılmaktadır.

Linux sistemleriyle BSD ve Mac OS X arasında sistem fonksiyonlarının çağrılması arasında küçük bir farklılık vardır. Linux'ta sistem fonksiyonlarının parametreleri yazmaçlarla aktarılırken BSD ve Mac OS X sistemlerinde -tıpkı C'deki gibi- stack yoluyla aktarım yapılmaktadır. (Ayrıca Linux sistemlerindeki sistem fonksiyonlarının numaralarının ve parametrik yapılarının BSD ve Mac OS X sistemleriyle bire bir aynı olduğunu da düşünmemelisiniz.) Windows sistemlerinde ise çekirdek moduna geçiş genel olarak 2EH kesmesiyle yapılmaktadır. Fakat genel mekanizma Linux, BSD ve Mac OS X sistemlerine oldukça benzemektedir.

9.6.4. Korumalı Moddaki Çalışmanı Özeti

Korumalı mod proseslerin bir arada çalıştığı çok prosesli sistemlerde sistem güvenliğini artırmak için düşünülmüştür. Intel'in koruma mekanizmasında dört öncelik derecesi vardır. Ancak işletim sistemleri genellikle yalnızca iki dereceyi kullanmaktadır: 0 ve 3. Intel sisteminde düşük numara daha yüksek, yüksek numara ise daha düşük öncelik belirtmektedir. 0 önceliğine "çekirdek modu (kernel mode)" önceliği, 3 önceliğine de "kullanıcı modu (user mode)" önceliği denilmektedir.

32 bit Windows ve Linux gibi sistemler düz model (flat model) kullanmaktadır. Bu modelde tüm segment yazmaçlarının gösterdiği betimleyicilerin taban adresleri 0 ve limit değerleri de 4 GB'dir. Düz modelde segment tabanlı bir koruma uygulanmamaktadır. Yani bu modelde bir prosesin CS yazmacı dışındaki segment yazmaçlarının değerlerini değiştirmesi için bir gerekçe yoktur. Zaten düz model uygulayan

sistemlerde genellikle tüm kullanıcı mod programları için tek bir kod ve data/stack betimleyicisi kullanılmaktadır. Bu betimleyicilerle de terorik olarak belleğin her yerine erişilebilmektedir.

Korumalı modda o anda çalışmakta olan kodun öncelik derecesi CS yazmacının düşük anlamlı iki bitiyle belirlenmektedir. Bu bitlere CPL (Current Privilege Level) denir. Intel'de özel makine komutlarını ancak CPL değeri 0 olan kodlar kullanabilirler. Böylece sıradan prosesler CPL = 3 değeriyle çalıştıkları için bu makine komutlarını kullanamamaktadır. Ayrıca CPL değeri sayfalama mekanizması aktifken bellekte sayfalara erişirken de kontrol işlemlerine sokulmaktadır. Şöyle ki: Her sayfanın iki öncelik derecesi vardır. Önceliklerden birine "kullanıcı (user)", diğerine ise "yönetici (supervisor)" önceliği denir. CPL değeri 1, 2 ve 3 olan kodlar ancak kullanıcı önceliğindeki sayfalara erişebilirler. CPL değeri 0 olan kodlar ise tüm sayfalara erişebilmektedir. İşletim sisteminin çekirdek kodları "yönetici (supervisor)" önceliğindeki sayfalarda tutulur. Böylece bu alanlara yalnızca işletim sisteminin kodları erişebilmektedir.

Korumalı modda programcı DS, ES, SS, FS ve GS segment yazmaçlarındaki değerleri MOV komutlarıyla değiştirmek isterse bazı kontroller uygulanmaktadır. Özet olarak programcı CPL değerinden daha yüksek önceliğe sahip bir betimleyiciyi gösteren selektörü bu segment yazmaçlarına yükleyememektedir. Zaten yukarıda da ifade ettiğimiz gibi Windows gibi Linux gibi sistemler "düz bellek modeli (flat model)" kullanmaktadır. Düz bellek modelinde de DS, ES, SS, FS ve GS segment yazmaçlarını yüklemek istemenin pratikte bir amacı yoktur.

Korumalı modda çalışmakta olan kodun önceliğinin yükseltilmesi uzak CALL ve JMP işlemleriyle yapılamamaktadır. Bunu yapmanın tek yolu "kapı (gate)" denilen özel bir mekanizmayı kullanmaktır. Düşük öncelikli kodlar kapı ile belirtilen adresteki kodları yüksek öncelikle çalıştırabilmektedir.

9.7. 32 Bit Intel İşlemcilerinde Sayfalama (Paging) İşlemleri

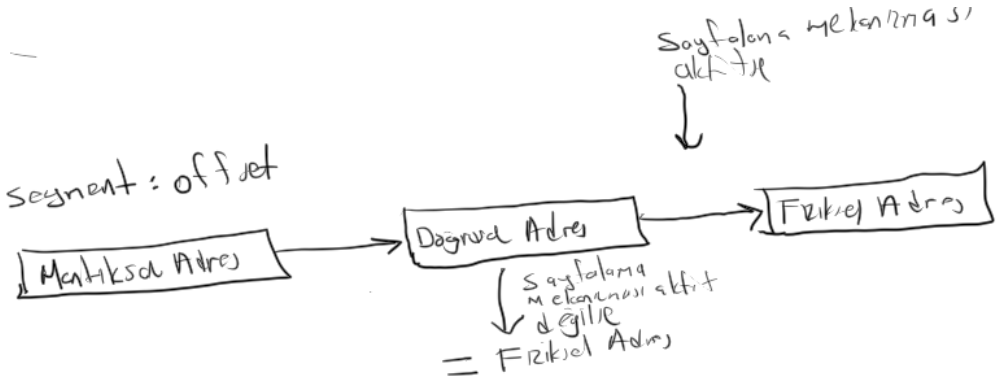
Sayfalama mekanizması modern ve geniş kapasiteli pek çok işlemcide bulunmaktadır. Intel işlemcileri 80386 ile birlikte sayfalama mekanizmasına sahip olmuşlardır. Intel 64 bite geçildiğinde sayfalama mekanizmalarında da birtakım eklentiler yapmıştır. Biz burada şimdilik 32 bit Intel işlemcilerindeki sayfalama mekanizması üzerinde duracağız.

Intel terminolojisinde adres kavramı üç gruba ayrılmaktadır:

1) Mantıksal Adresler (Logical Addresses)
2) Doğrusal Adresler (Linear Adresses)
3) Fiziksel Adresler (Physical Addresses)

Segment ve Offset'ten oluşan "segment : offset" biçiminde belirtilen adreslere mantıksal adresler denir. Anımsanacağı gibi aslında korumalı modda her adres bir mantıksal adrestir. İşlemci segment ile belirtilen selektörün gösterdiği yerdeki betimleyicinin içerisindeki taban adresi offset ile toplar ve buradan doğrusal adresi elde eder. Yani doğrusal adres segment-offset işlemi yapıldıktan sonra elde edilen adres değeridir. Anımsanacağı gibi Windows, Linux, Mac OS X gibi "düz model (flat model)" kullanan işletim sistemlerinde zaten segment yazmaçlarının gösterdiği betimleyicilerin taban kısımları sıfırdır. Dolayısıyla düz bellek modeli kullanan sistemleri sanki hiç segment yokmuş gibi düşünebiliriz. Bu sistemlerde offset zaten doğrusal adres belirtiyor durumdadır.

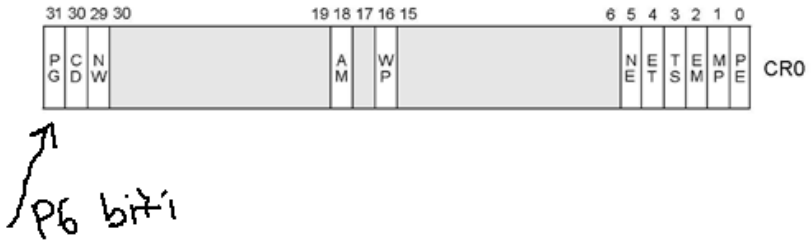
Intel işlemcilerinde sayfalama (paging) denilen mekanizma etkin hale getirilebilir ya da kapatılabilir. Bu sistemlerde eğer sayfalama mekanizması etkin değilse (yani kapalıysa) doğrusal adreslerle fiziksel adresler arasında bir fark yoktur. Yani doğrusal adresler aynı zamanda RAM'de yer belirten gerçek fiziksel adreslerdir. Ancak eğer sayfalama denilen bu mekanizma etkinse (yani açıksa) doğrusal adresler RAM'de yer belirten fiziksel adresler değildir. Gerçek fiziksel adresler doğrusal adreslerin bir işleme sokulmasıyla elde edilmektedir. Bu durumu şekilsel olarak şöyle gösterebiliriz:



Intel işlemcilerinde doğrusal adresleri fiziksel adreslere dönüştüren bölüme "sayfalama birimi (paging unit)" denilmektedir.

Anımsanacağı gibi Intel işlemcileri reset edildiğinde çalışma “16 bit gerçek moddan” başlamaktadır. Bu modda sayfalama etkin değildir. Gerçek modda adreslerin offset kısımları 16 bittir, segment yazmaçları selektör belirtmez. Mantıksal adresler segmentin 16 ile çarpılıp offset ile toplanmasıyla elde edilir. Gerçek modda sayfalama mekanizması etkin olmadığı için de doğrusal adres aynı zamanda fiziksel adres anlamına gelmektedir. (Gerçek moddaki çalışma ana hatlarıyla ileri bölümlerde ele alınmaktadır.)

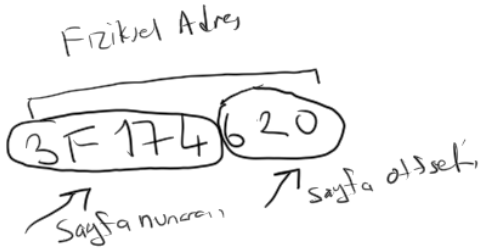
Intel işlemcilerinde sayfalama mekanizması ancak korumalı modda etkin hale getirilebilmektedir. Sayfalama mekanizmasını etkin hale getirmek için CR0 yazmacının en son biti olan 31 numaralı bit set edilir. Bu bite PG (Paging) biti denilmektedir.



Aardışıl belli uzunluktaki byte byte topluluğuna sayfa (page) denilmektedir. Bir sayfanın kaç byte uzunlukta olacağı işlemci ailesine bağlı olarak bağlı olarak değişebildiği gibi aynı işlemci ailesinde modelden modele de değişebilmektedir. 32 bit Intel işlemcilerinde bir sayfa 4 KB (4096 byte) ya da 4 MB (4194304 byte) büyüklüğünde olabilmektedir. 4 MB’lik sayfalara "büyük sayfalar" denir. Büyük sayfalar şimdilik işletim sistemleri tarafından tercih edilmemektedir. O halde 32 bit Intel işlemcilerinde ağırlıklı kullanılan sayfa uzunluğunun 4 KB (4096 byte) olduğunu söyleyebiliriz. Sparc ve Alpha işlemcilerinde sayfalar 8 KB uzunluğundadır. Biz buradaki notlarımızda aksi belirtilmediği sürece sayfa uzunluklarının 4 KB olduğunu varsayacağız.

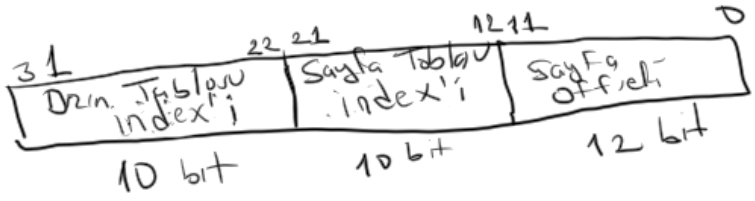
Sayfalama mekanizmasında fiziksel bellekteki her 4 KB’ye bir sayfa numarası karşı düşürülmüştür. Örneğin fiziksel belleğin tepesindeki ilk 4 KB 0 numaralı sayfayı, ikinci 4 KB 1 numaralı sayfayı, üçüncü 4 KB 2 numaralı sayfayı oluşturmaktadır. Fiziksel bellek bu biçimde sayfalar temelinde numaralandırılmıştır.

Bilindiği gibi 32 bit sistemlerde adresler 32 bit uzunluğundadır. Biz 32 bitlik adresleri 8 hex digit ile gösterebiliriz. Bu durumda 32 bit fiziksel bir adresin hangi fiziksel sayfada olduğu adres değerinin 4096'ya bölünmesiyle elde edilebilir. Hex sistemde bir değeri 4096'ya bölmek onun düşük anlamlı 3 hex digit'ini atmakla yapılabilir. 32 bitlik bir fiziksel adresin hangi fiziksel sayfaya karşı geldiğini belirledikten sonra o fiziksel sayfadan ne kadar ileride olduğunu da belirleyebiliriz. Adresin onun içinde bulunduğu fiziksel sayfanın neresinde olduğu bilgisine "sayfa offset'i" denilmektedir. Adresin sayfa offset'i onun 4096'ya bölümünden elde edilen kalanla elde edilebilir. Hex sistemde bir değerin 4096'ya bölümünden kalan o değerin düşük anlamlı 3 hex digitidir. Bu durumda 32 bitlik fiziksel bir adresin yüksek anlamlı 5 hex digit'i onun fiziksel sayfa numarasını, düşük anlamlı 3 hex digiti'i de sayfa offset'ini verecektir. Örneğin:

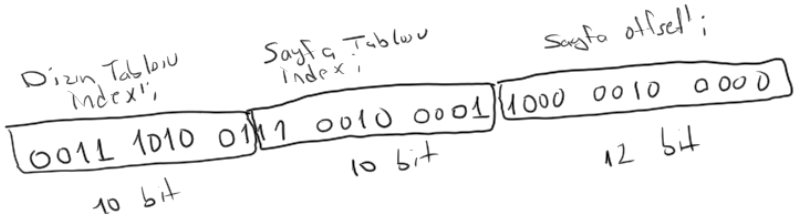


9.7.1. Sayfalama Birimi Tarafından Doğrusal Adreslerin Fiziksel Adreslere Dönüştürülmesi

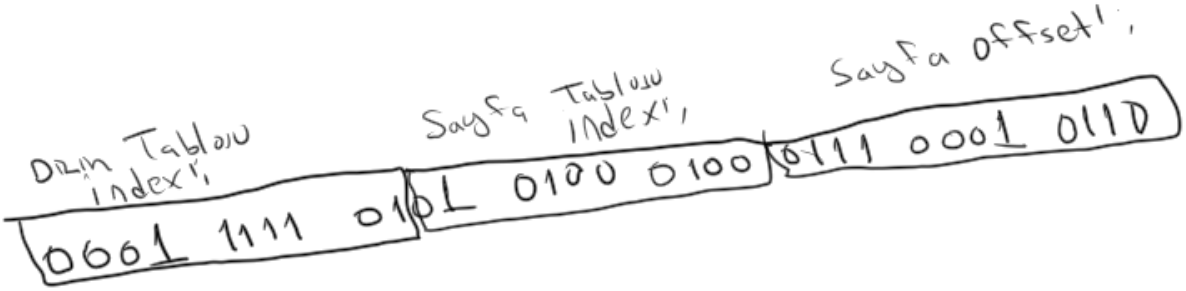
4 KB'lik sayfaların kullanıldığı 32 bit Intel işlemcilerinde doğrusal adresler kendi içerisinde üç parçaya ayrılmaktadır: "Dizin Tablosu Index'i", "Sayfa Tablosu Index'i" ve "Sayfa Offset'i".



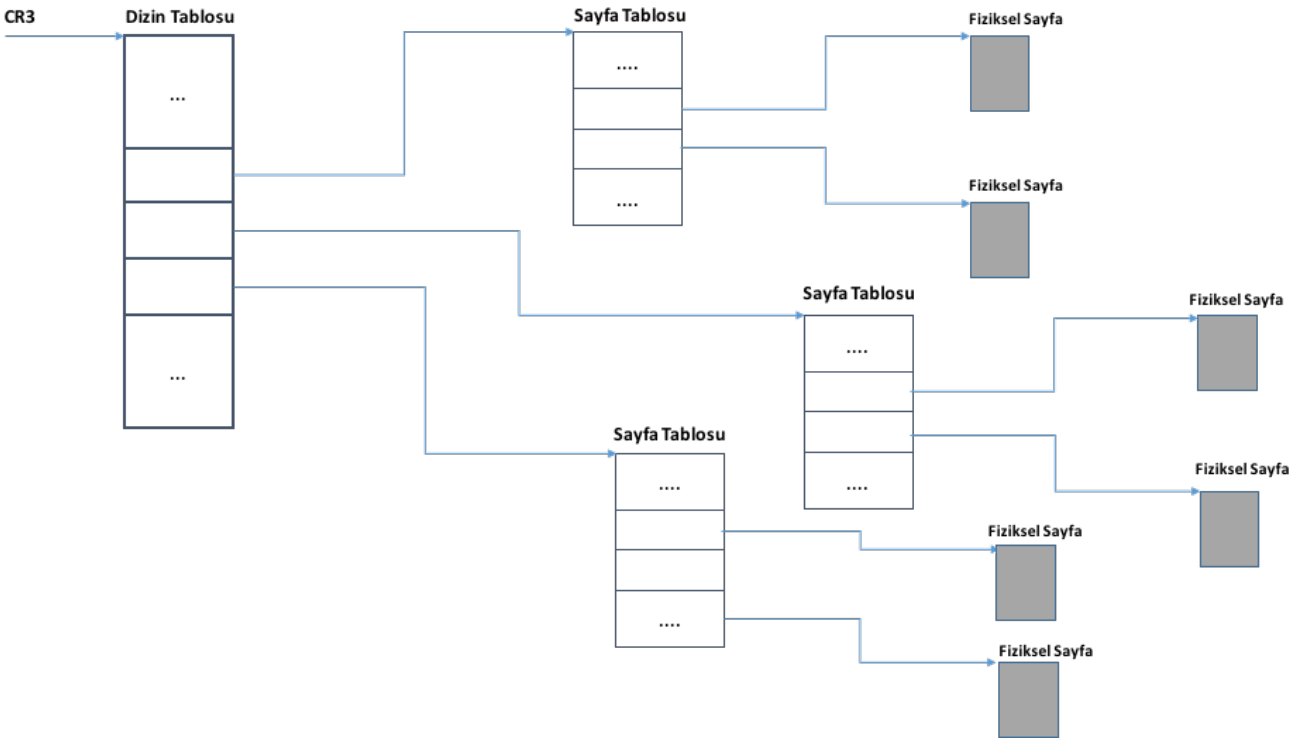
Şimdi birkaç örnek üzerinde duralım. Doğrusal adres hex sistemde 3A721820 biçiminde olsun. Biz bu değeri 2'lik sistemde ifade edip parçalarına şöyle ayrıştırabiliriz:



Şimdi de aynı işlemi 1F544716 adresi için yapalım:



Doğrusal adreslerin fiziksel adreslere dönüştürülmesi işleminde iki grup tablodan faydalanılmaktadır: "Dizin tablosu (page directory)" ve "sayfa tablosu (page table)". Dizin tablosu toplamda bir tanedir ancak sayfa tabloları fiziksel RAM büyüklüğüne bağlı olarak birden fazla sayıdadır. Dizin tablosu sayfa tablolarının fiziksel sayfa numaralarını gösteren bir dizi gibi düşünülebilir. Dizin tablosu dizin elemanlarından (page directory entry) oluşmaktadır. Her dizin elemanı bir sayfa tablosunun fiziksel sayfa numarısını ve özelliklerini tutmaktadır. Sayfa tabloları da sayfaların fiziksel sayfa numaralarını tutan bir dizi gibi düşünülebilir. Sayfa tablosu "sayfa elemanlarından (page table entry)" oluşmaktadır. Her sayfa elemanı bir fiziksel sayfanın numarasını ve onun bazı özelliklerini tutmaktadır. Dizin elemanları ve sayfa elemanları 4 byte uzunluğundadır.



Doğrusal adresin yüksek anlamlı 10 biti (dizin tablo indeksi) dizin tablosu denilen tabloda bir indeks belirtmektedir. Dizin tablosundan bu indeksteki eleman çekilerek doğrusal adresin ilişkin olduğu sayfa tablosunun fiziksel sayfa numarası elde edilir. (Bu değer 4096 ile çarpılıp sayfa tablosunun fiziksel adresi hesaplanabilir.) Doğrusal adresin ortadaki 10 biti de (sayfa tablosu indeksi) sayfa tablosunda bir indeks belirtmektedir. Bu değer de dizin tablosundan elde edilen sayfa tablosuna indeks yapılarak doğrusal adresin

ilişkin olduğu fiziksel sayfa numarası elde edilmektedir. (Bu değer 4096 ile çarpılarak fiziksel sayfanın adresi hesaplanabilir.) Böylece doğrusal adresin yüksek anlamlı 20 bitinden bu doğrusal adrese karşılık gelen fiziksel sayfa adresi elde edilmiş olur. İşte bu fiziksel sayfa adresi offset ile toplanarak doğrusal adrese karşı gelen nihai fiziksel adres elde edilmektedir. İşlemci her adres işleminde yukarıdaki algoritmayı uygulayarak o doğrusal adresi fiziksel adrese dönüştürmekte ve ondan sonra RAM erişimini yapmaktadır. Bu işlem aşağıdaki gibi bir şekille gösterilebilir:

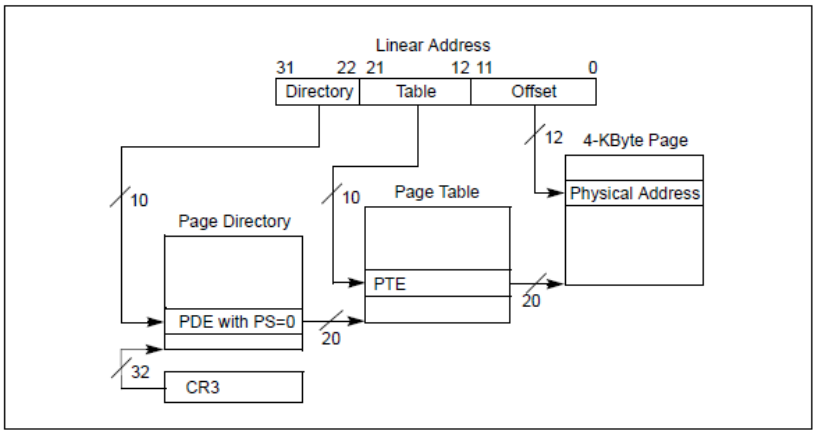
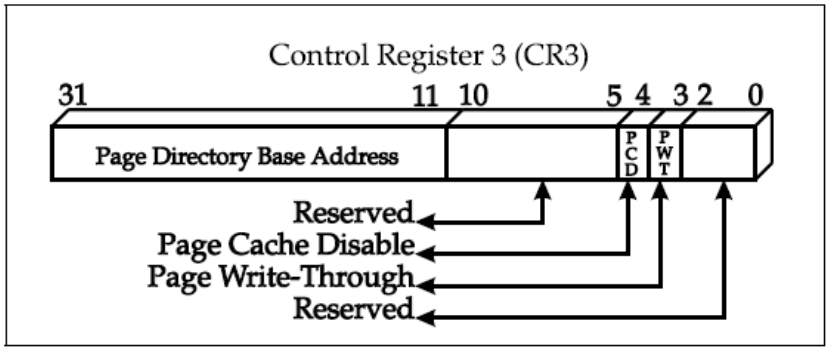


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Bu şekil Intel'in orijinak dokümanlarından alınmıştır. Intel'in çizimlerde bizim yaptığımızın aksine düşük adresi daha aşağıda gösterdiğini anımsatalım.

Görüldüğü gibi doğrusal adresler iki tabloya bakılarak fiziksel adreslere dönüştürülmektedir. Önce dizin tablosundan sayfa tablosunun fiziksel sayfa numarası bulunmakta sonra da sayfa tablosundan doğrusal adresin ilişkin olduğu fiziksel sayfanın numarası elde edilmektedir. Pekiyi dizin tablosu nerededir? İşte Intel işlemcileri dizin tablosunu CR3 yisimli kontrol yazmacının gösterdiği yerde aramaktadır. Bu durumda sayfalama mekanizmasının çalışabilmesi için önce dizin tablosunun ve sayfa tablolarının oluşturulması ve CR3 yazmacının da dizin tablosunun adresini gösterir hale getirilmesi gerekmektedir. CR3 yazmacının bitleri aşağıdaki gibidir:

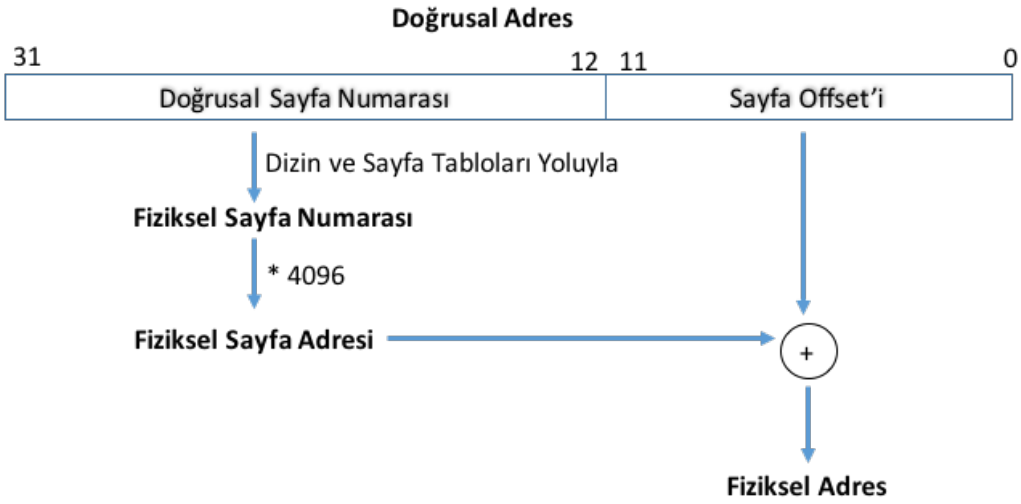


Şekilden de anlaşılabileceği gibi CR3 yazmacı dizin tablosunun fiziksel adresini değil, fiziksel sayfa numarasını tutmaktadır. Çünkü Intel'de dizin tabloları ve sayfa tabloları 4 KB'ye hizalanmış olmak zorundadır. CR3 yazmacının yüksek anlamlı 20 biti dizin tablosunun fiziksel adres numarasını tutmaktadır. Bu değer 4096 ile çarpılarak dizin tablosunun fiziksel adresi elde edilebilir. CR3 yazmacının diğer bitleriyle biz şimdilik bu aşamada ilgilenmeyeceğiz.

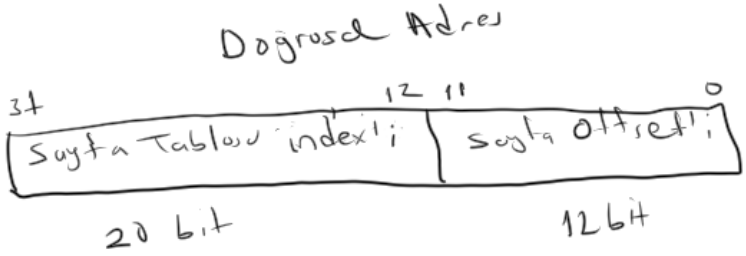
Dizin tablosu 4 KB uzunluktadır. Nereden mi anladık? Doğrusal adresteki dizin indeksi 10 bittir. Bu da tabloda toplam 1024 girişin bulunduğu anlamına gelir. Bir giriş 4 byte olduğuna göre dizin tablosunun toplam uzunluğu 4 KB olacaktır. Aynı nedenden dolayı sayfa tabloları da 4 KB uzunluğundadır. Doğrusal adreslerdeki sayfa tablo indeksinin de 10 bit olduğuna dikkat ediniz.

Intel işlemcilerindeki sayfalama mekanizmasını şöyle de düşünebilirsiniz: Doğrusal adresin yüksek anlamlı

20 biti bir doğrusal sayfa numarası belirtmektedir. Bu doğrusal sayfa numarası dizin tablosu ve sayfa tabloları yoluyla bir fiziksel sayfa numarasına dönüştürülür. Bu fiziksel sayfa numarasının belirttiği sayfa adresine doğrusal adresin düşük anlamlı 12 biti toplanarak fiziksel sayfa elde edilmektedir.



Pekiyi doğrusal adresin fiziksel adrese dönüştürülmesi sürecinde doğrusal adres için neticede bir fiziksel sayfa bulunduğuna göre bu işlem neden iki aşamada ve iki tablo kullanılarak yapılmaktadır? Örneğin doğrusal adres aşağıdaki gibi üç parçaya değil de iki parçaya ayrılsaydı sistem daha basit olmaz mıydı?



Yani sistemde yalnızca tablosu olsaydı ve doğrusal adresin yüksek anlamlı 20 biti bu tabloya indeks yapılıp oradan fiziksel sayfa numarası çekilseydi sistem daha basit olmaz mıydı? Evet bu durumda belki sistem daha basit olurdu. Ancak bu durumda sayfa tablosu yaklaşık 1 milyon elemandan oluşurdu. Böyle bir sayfa tablosunun kaplayacağı alan da 4 MB olurdu. Aslında ileride de ele alınacağı gibi sayfa tablolarının hepsi doldurulmak zorunda değildir. Yalnızca gerektiği miktarda sayfa tablosunun doldurulması yeterli olabilmektedir. İşte bunu dikkate aldığımızda iki kademeli dönüştürme toplamda çok daha az meta-data alanı gerektirir. Adres alanının daha geniş olduğu 64 bit sistemlerde genellikle iki değil üç kademeli bir tablo tercih edilir. Gerçekten de X64 işlemcilerinde üç kademeli sayfa tabloları kullanılmaktadır.

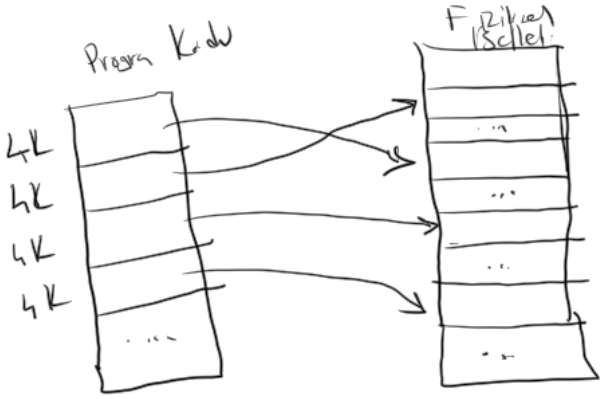
Doğrusal adreslerin dizin ve sayfa tablolarına bakılarak fiziksel adreslere dönüştürülmesi pek çok bellek okumasına yol açacak potansiyeldedir. Pekiyi böyle olduğu halde sistem neden yine de çok hızlı çalışmaktadır? İşte Intel tasarımcıları dizin ve sayfa tablolarına sık sık başvurmamak için orada son okunan girişleri işlemci içerisindeki bu amaçla oluşturulmuş bir cache sistemine aktarmaktadır. Bu cache sistemine "TLB (Translation Lokkaside Buffer)" denilmektedir. Böylece işlemci belli süreden sonra okuduğu dizin ve sayfa tablo girişlerini bir daha okumak için belleğe başvurmamaktadır. TLB isimli cache'i işlemcinin içerisindeki L1 cache ile karıştırmayınız. TLB yalnızca dizin ve sayfa girişlerini tutan çok hızlı bir tampon bellektir. Halbuki L1 cache genel amaçlı bir cache sistemidir.

9.7.2. Sayfalama Mekanizmasının Anlamı

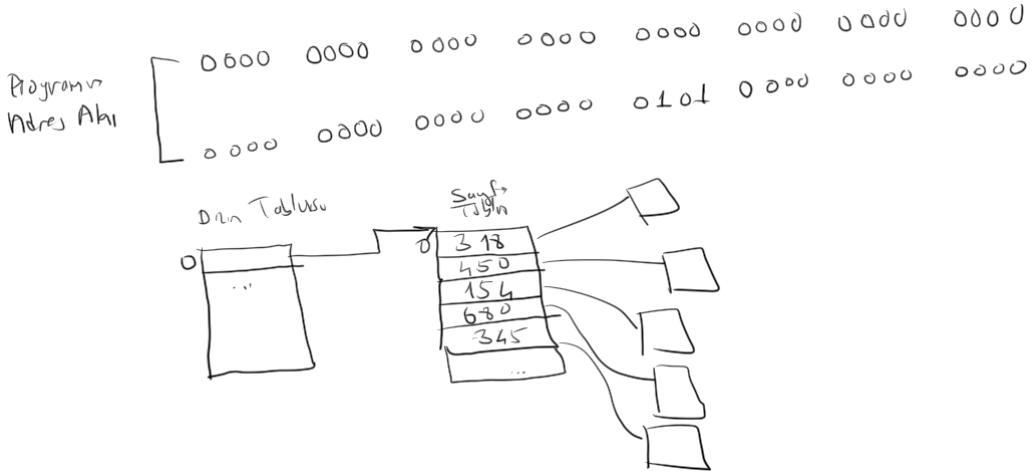
Sayfalama mekanizması sayesinde bir programın fiziksel belleğe ardışıl yüklenme zorunluluğu ortadan kaldırılır. Böylece fiziksel bellek bölünmesinin (fragmentation) belli ölçülerde önüne geçilebilmektedir. Bunun dışında sayfalamadan "sanal bellek mekanizmasının (virtual memory)" gerçekleştirilmesinde de

faydalanılmaktadır. Sanal bellek mekanizması sonraki bölümde ele alınmaktadır.

Yukarıda da belirttiğimiz gibi sayfalama mekanizması sayesinde bir programın 4KB'lik kısımları fiziksel belleğe ardışıl yüklenmek zorunda değildir. Bunu aşağıdaki gibi bir şekille gösterebiliriz:



Sayfalama mekanizmasının kullanıldığı sistemlerde programlarımız belleğe ardışıl yüklenecekmiş gibi derlenir ve bağlanır. Ancak işletim sistemi programlarımızı aslında fiziksel belleğe ardışıl yüklememektedir. Programımızın 4KB'lik kısımları birbirleriyle ilgisiz fiziksel sayfalarda bulunuyor olabilir. Ancak sanki o fiziksel sayfalar sayfa tablosu yoluyla ardışılmış gibi gösterilebilmektedir. Örneğin 20K uzunluğunda (5 sayfalık) bir programımız sanki belleğin tepesinden itibaren ardışıl yüklenecekmiş gibi derlenip bağlanmış olsun. Biz programımıza baktığımızda sanki onun fiziksel belleğe ardışıl yüklenmesi gerektiğini düşünebiliriz. Halbuki sayfalama mekaznizması sayesinde programımızın mantıksal ardışıllığı bozulmadan onun 4 KB'lik parçaları fiziksel bellekte farklı yerlere yüklenebilir.



Gerçekten de Windows, Linux ve Mac OS X gibi sistemlerde aslında programlar sanki tek parça ve ardışıl olarak belleğe yüklenecekmiş gibi derlenip bağlanmaktadır. Ancak bu işletim sistemleri onları farklı fiziksel sayfalara yükleyip bu ardışıllığı dizin ve sayfa tablolarını organize ederek yapay biçimde sağlarlar. Bu sistemlerde kodları debugger'lar altında görüntülerken hiçbir zaman bu debugger'lar bize gerçek fiziksel adresleri göstermezler. Onların gösterdikleri adresler her zaman henüz dönüştürülmemiş olan doğrusal adreslerdir.

Yeni bir program yüklendiğinde işletim sistemi onun kod, data/bss ve stack alanları için fiziksel sayfalar tahsis etmektedir. Bir program sonlandığında da benzer biçimde işletim sistemi o program için tahsis ettiği sayfaları boşaltacaktır. Şüphesiz bu işlemleri yürütebilmek için sayfalama mekanizmasını kullanan işletim sistemlerinin hangi fiziksel sayfaların boş olduğunu, hangilerinin hangi programlar tarafından kullanıldığını izlemesi gerekir.

Sayfalama mekanizması genellikle sanal bellek mekanizmasını oluşturmak için sanal bellek mekanizmasıyla

birlikte kullanılmaktadır. Sanal bellek (virtual memory) konusu ileride ele alınacaktır.

9.7.3. Sayfa Düzeyinde Koruma İşlemleri

Intel işlemcileri (diğer işlemcilerin büyük bölümünde de böyle) safta tabanlı bir koruma mekanizmasına da sahiptir. Özellikle "düz model (flat model)" kullanan Windows, Linux ve Mac OS X gibi sistemler bellek korumasını prosesleri izole ederek sayfa tabanlı olarak gerçekleştirmektedir. Sayfa tabanlı bellek korumasının üç özelliği vardır: Sayfaya erişim ancak "çekirdek modunda (CPL = 0)" yapılabilir. Bir sayfa yazmaya karşı korunabilir ve sayfadan program çalıştırma engellenebilir.

Öncelikle dizin tablosundaki girişlerin (directory table entries) ve sayfa tablosundaki girişlerin formatlarına bakalım:

Figure 13-7: Page Directory Entry Format

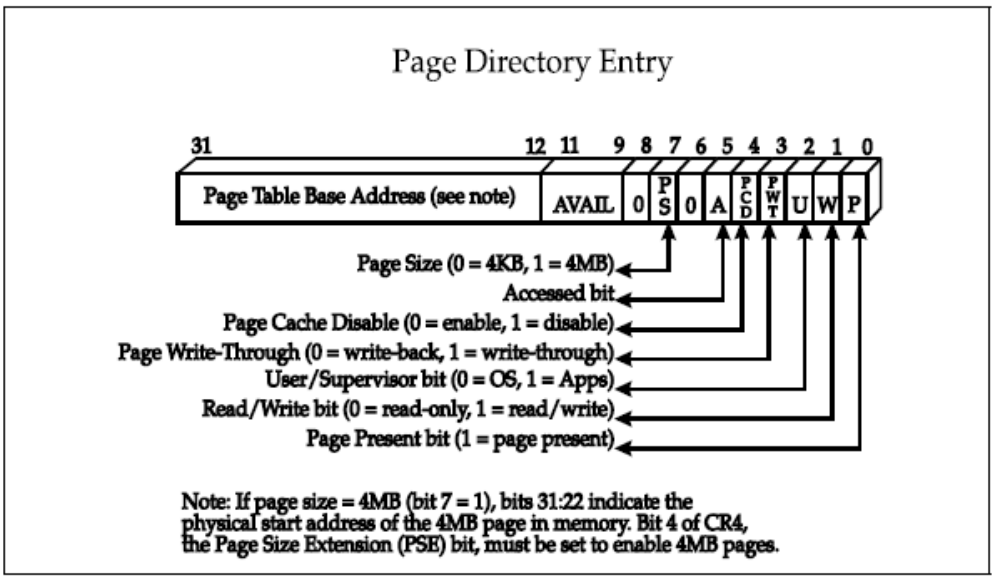
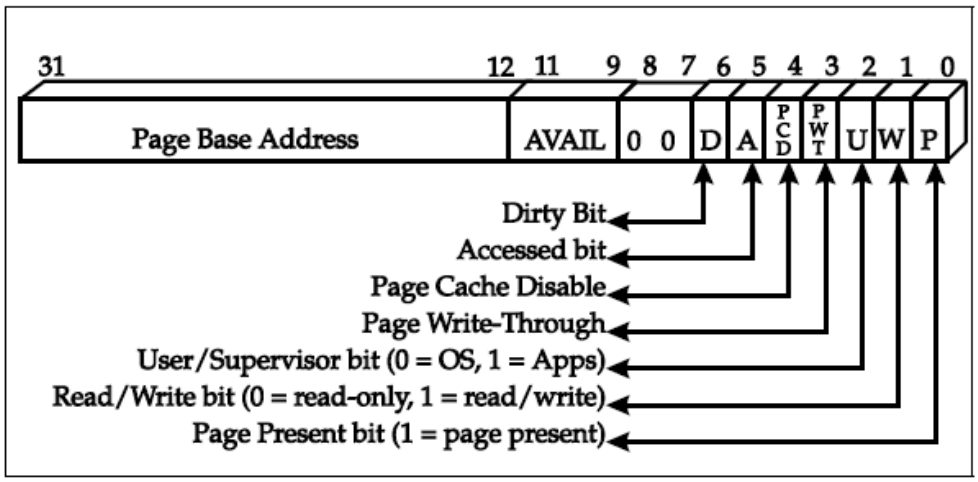


Figure 13-11: Page Table Entry Format



Dizin tablosu girişleriyle (elemanlarıyla) sayfa tablosu girişleri (elemanları) yapı olarak birbirlerine çok benzemektedir. Her iki girişte de 0 numaralı bite P (present) biti denilmektedir. Bu bit 1 ise ilgili sayfa tablosu ya da fiziksel sayfa mevcut değildir. Yani bu doğrusal sayfa için bir fiziksel sayfa yönlendirmesi yapılmamıştır. Dizin tablosu girişinde ya da sayfa tablosu girişinde P biti 0 ise işlemci "sayfalama hatası (page fault)" kesmesi oluşturmaktadır. Dizin ve sayfa girişlerindeki P biti sanal bellek mekanizmasında kullanılmaktadır. Bu konu ileride ele alınacaktır.

Her iki giriş türünde de 1 numaralı bit W bitidir. Bu bitin 1 olması ilgili sayfaya yazma yapılabileceği anlamına gelmektedir. Bu bit 0 ise ilgili sayfaya yazma yapmak istenirse "sayfalama hatası (page fault)" kesmesi oluşur. Dizin tablosu elemanı bu bakımdan ana şalter görevi yapar. Yani dizin tablosunun ilgili elemanının W biti 0 ise o dizin tablosunun gösterdiği sayfa tablosu içerisindeki tüm fiziksel sayfalar (o sayfaların sayfa tablosundaki W bitleri 1 olsa bile) "read-only" durumdadır. Default durumda CPL = 0 olan çekirdek modundaki kodlar için bu W bitinin bir etkisi yoktur. Yani çekirdek mod programlar W biti 0 olsa bile ilgili sayfaya yazma yapabilirler. Ancak bu durum ileride ele alınacağı gibi çekirdek modundayken sanal bellek mekanizmasında "copy on write" özelliğinin uygulanamamasına yol açmaktadır. İşte CPL = 0 olan çekirdek modunda çalışan kodların, W biti 0 olan sayfalara yazamaması isteniyorsa CR0 yazmacının 16'ıncı biti olan WP biti 1'lenmelidir. Yukarıda da belirtildiği gibi bu bit default durumda 0'dır.

Yine her iki giriş türünün de 3 numaralı olan U biti sayfanın "spuervisor" modda mı, yoksa "user" modunda mı olduğunu belirtmektedir. Eğer bu bit 1 ise sayfa "user moddadır" ve sayfaya tüm CPL değerli kodlar erişebilir. Eğer bu bit 0 ise sayfa "supervisor" modundadır. (Intel burada "kernel mode" yerine daha genel olacak biçimde "supervisor mode" terimini kullanmayı yeğlemiştir.) Bu durumda sayfaya yalnızca CPL = 0 olan çekirdek modda çalışan kodlar erişebilir. Bu sayede işletim sistemi kendi kod ve verilerinin yüklü olduğu fiziksel sayfaların U bitlerini 0 yaparak kendini CPL = 1, CPL = 2 ve CPL = 3 öncelikli kodlardan korumuş olur. Gerçekten de biz Windows, Linux ve Mac OS X gibi işletim sistemlerinde durum böyledir. Biz bu sistemlerde işletim sisteminin bulunduğu kod ve veri alanlarına okuma ya da yazma amaçlı erişmek istersek "sayfalama hatası (page fault)" kesmesi oluşacaktır. Dizin tablosu girişindeki U biti o dizin tablosu elemanın gösterdiği sayfa tablosu girişleri için ana şalter görevindedir. Yani dizin tablosu girişindeki U biti 1 ise bunun gösterdiği sayfa tablosu girişlerindeki U biti 0 olsa bile sayfa "user mode"dadır.

Her iki giriş türünde yine PWT (Page Write Through) ve PCD (Page Cache Disable) bitleri ortaktır. Bu iki bit işlemcinin içerisindeki cache (L1 cache) mekanizmasında etkili olmaktadır. PWT biti 1 ise yazma işlemi doğrudan işlemcinin eriştiği belleğe yapılır. İşlemcinin kendi cache'i yazmada devreye sokulmaz. PCD = 1 ise ilgili sayfa (ya da ilgili sayfaların hepsi) işlemci tarafından cache'e alınmaz.

Her iki giriş türünde yine 5'inci bit olan A (Access) biti ortaktır. Bu bit ilgili sayfaya ya da sayfa tablosuna her erişildiğinde işlemci tarafından 1 yapılmaktadır. Böylece işletim sistemi bazı durumlarda hangi sayfalara daha fazla erişildiğine yönelik istatistiği bu bitleri dikkate alarak yapabilmektedir.

Sayfa tablosundaki 6 numaralı bitine D (Dirty) biti denilmektedir. İşlemci sayfaya her yazma yapıldığında bu biti 1 yapar. Böylece işletim sistemi sanal bellek mekanizmasında bu fiziksel sayfayı boşaltırken boşuna bu sayfanın içeriğini diske geri yazmaz. D biti dizin tablosu girişlerinde kullanılmamaktadır.

Dizin girişindeki 7 numaralı bit sayfa tablosunun 4 MB'lik büyük sayfalara ilişkin mi yoksa 4 KB'lik normal sayfalara ilişkin mi olduğu bilgisini tutmaktadır. Eğer bu bit 0 ise sayfa 4K uzunlunda, 1 ise 4 MB uzunluğundadır. Bu 7 numaralı bit sayfa tablosu girişlerinde kullanılmamaktadır.

Her iki giriş türünde de [12-31] numaralı bitler (toplam 20 bit) fiziksel sayfa numarası belirtirler. Dizin girişleri için bu bitler sayfa tablosunun fiziksel sayfa numarasını belirtirken, sayfa girişleri için sayfanın fiziksel sayfa numarasını belirtmektedir.

Sayfa düzeyinde "kod çalıştırma" koruması Intel işlemcilerine belirli bir modelden sonra eklenmiştir. Bunun etkin olabilmesi için işlemcinin PAE modunda olması gerekmektedir. 32 bit normal korumalı modda Intel işlemcilerinde bu özellik yoktur. Çalışma düzeyinde koruma bazı virütik kodlara karşı güvenliği artırmayı hedeflemektedir. Şöyle ki: Sayfalar çalıştırma koruması olan ve olmayan biçiminde özelliklendirilebilmektedir. Eğer sayfanın çalışma koruması varsa o sayfadan kod çalıştırılamaz. Çalıştırılmak istenirse "sayfalama hatası (page fault)" oluşur. Artık yeni ve modern pek çok işlemcide bu tarz "çalıştırma koruması" bulunmaktadır.

9.7.4. Çok Prosesli (Multiprocessing) ve Çok Thread'li Çalışmanın Temelleri

Tek bir işlemci varken aynı anda birden fazla kod nasıl birbirlerinden bağımsız olarak çalışabilmektedir? İşte genellikle işletim sistemleri prosesleri zaman paylaşımlı (time sharing) olarak çalıştırmaktadır. Zaman paylaşımlı çalışma tekniği bilgisayarlarla insanların hiç operatör olmadan etkileşime girdiği 50'li yılların sonlarına doğru geliştirilmiştir. Eskiden bilgisayarlar çok pahalıydı ve sayıları çok azdı. Onlardan azami ölçüde fayda sağlayabilmek için çok erken dönemlerde zaman paylaşımlı çalışma modeli ortaya atılmıştır.

Bir prosesin o andaki tüm durumu aslında prosesin bellek alanı ve yazmaç değerlerinden oluşmaktadır. Eğer çalışmakta olan kodun bellek alanı korunup CPU yazmaçları saklanırsa sonra o yazmaçlar yeniden yüklenerek kod kaldığı yerden çalışmasına devam ettirilebilir.

Çok prosesli sistemlerde bir prosesin çalışmaya ara verilip başka bir prosesin çalıştırılmaya devam ettirilmesi sürecine "processler arası geçiş (process switch / task switch)" denilmektedir. Çok thread'li sistemlerde aynı prosesin farklı akışları arasında da geçiş söz konusu olabilir. Bu nedenle bu durumu kapsayacak biçimde çalışmakta olan koda ara verilip sıradaki kodun kalınan yerden çalışmaya devam ettirlmesine daha genel olarak "bağlamsal geçiş (context switch)" de denilmektedir.

Zaman paylaşımlı çok prosesli çalışma uygulayan işletim sistemleri kendi aralarında ikiye ayrılmaktadır:

1) "Preemtive" olmayan (nonpreemptive) çok prosesli sistemler

2) "Preemptive" çok prosesli sistemler

"Preemtive" olmayan sistemlere "cooperative multitask" sistemler de denilmektedir. Bu sistemlerde prosesler arası geçiş o anda çalışmakta olan proses tarafından isteye bağlı olarak yapılmaktadır. Çalışmakta olan program uzun süre beklemeye yol açabilecek bir sistem fonksiyonu çağrıldığında bu fonksiyon çalışmakta olan prosese ara verip onun CPU yazmaç bilgilerini saklayıp bekleyen yeni prosesi CPU yazmaçlarına yükleyerek çalıştırmaktadır. Tabii isterse proses bir sistem fonksiyonuyla çalışmayı kendi isteğiyle başka prosese de bırakabilmektedir. "Preemptive" olmayan sistemlerin çekirdek tasarımları oldukça basittir. "Preemptive" sistemlere göre pek çok sorun tasarımda göz ardı edilebilmektedir. Ancak bu tür sistemler yeteri kadar verimli ve güvenli değildir. Çünkü bu tür sistemlerde bir proses CPU'yu tekeline alabilmektedir. (Örneğin bir kod sonsuz döngüye girdiğinde bundan tüm sistem etkilenebilmektedir.) Geçmişte kullanılan Windows 3.X sistemler, Palm OS gibi sistemler "preemptive" olmayan çok prosesli sistemlere örnek olarak verilebilir.

"Preemptive" sistemlerde prosesler arası geçiş donanım kesmesi yoluyla belli periyotlarda zorla (yani herhangi bir makine komutunda) yapılmaktadır. (Zaten İngilizce "preemption" sözcüğü "zorla ele geçirmek" anlamına gelmektedir.) "Preemtive" sistemlerde prosesler arası geçiş ya da bağlamsal geçiş belli bir çalışma zamanı dolduğunda donanım kesmesi yoluyla yapıldığı için bu sistemler hem daha adil bir CPU paylaşımına olanak sağlarlar hem de genel olarak daha verimli olma eğilimindedir. Bugün kullandığımız Windows, Linux, Mac OS X sistemleri "preemptive" çok prosesli ve çok thread'li işletim sistemleridir.

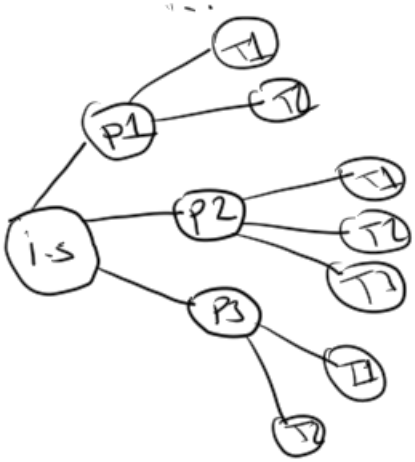
Zaman paylaşımlı çalışmada bir prosesin (ya da thread'in) parçalı çalışma süresine "quanta süresi" ya da "quantum" denilmektedir. Proses ya da thread eğer hiç bloke olmazsa quanta süresi boyunca çalışır. Bu süreyi doldurduğunda donanım kesmesi yoluyla bağamsal geçiş yoluyla çalışmasına ara verilmektedir.

Pekiyi çalışmasına ara verilmiş olan proses ya da thread'lerin CPU yazmaç bilgileri nerede bekletilmektedir? İşte işletim sistemleri ara verilmiş olan kodun o andaki yazmaç bilgilerini proses ya da thread için tahsis ettikleri "proses kontrol bloğu" ya da "thread kontrol bloğu" denilen bir veri yapısının ieçrisnde saklamaktadır. Bu veri yapıları işletim sisteminin diğer kod ve data'ları gibi "supervisor" mod ile korunmuş fiziksel sayfalarda tutulmaktadır.

Prosesin bağımsız olarak çizelgelenen akışlarına "thread" denilmektedir. Thread'lerin ilk denemeleri 80'li yıllarda yapılmıştır. Ancak işletim sistemlerine 90'lı yıllarda girmiştir. Microsoft'un ilk thread'li işletim sistemi Windows NT'dir. Bunu Windows 95 izlemiştir. Linux sistemlerinin ilk versiyonları thread'sizdi.

Daha sonra onlar da thread'lere sahip oldular. Apple thread'leri ilk kullanan firmalardandır. Steve Jobs'un Apple'dan ayrılarak kurduğu NeXT firması tarafından gerçekleştirilen NeXTStep işletim sistemi daha o zamanlarda thread'e çok benzer bir yapı kullanıyordu. Bugün artık thread'ler işletim sistemlerinin vaz geçilmez unsurlarından olmuşlardır.

Thread'li işletim sistemlerinde bağlamsal geçiş (context switch) prosesten prosese değil thread'ten thread'e yapılmaktadır. Bağlamsal geçiş oluştuğunda işletim sistemi o anda çalışmakta olan thread'in çalışmasına ara vererek diğer bir thread'i kaldığı yerden çalışmasına devam ettirmektedir. Bağlamsal geçiş sırasında geçilen thread aynı prosesin diğer bir thread'i olabileceği gibi başka bir prosesin thread'i de olabilir.

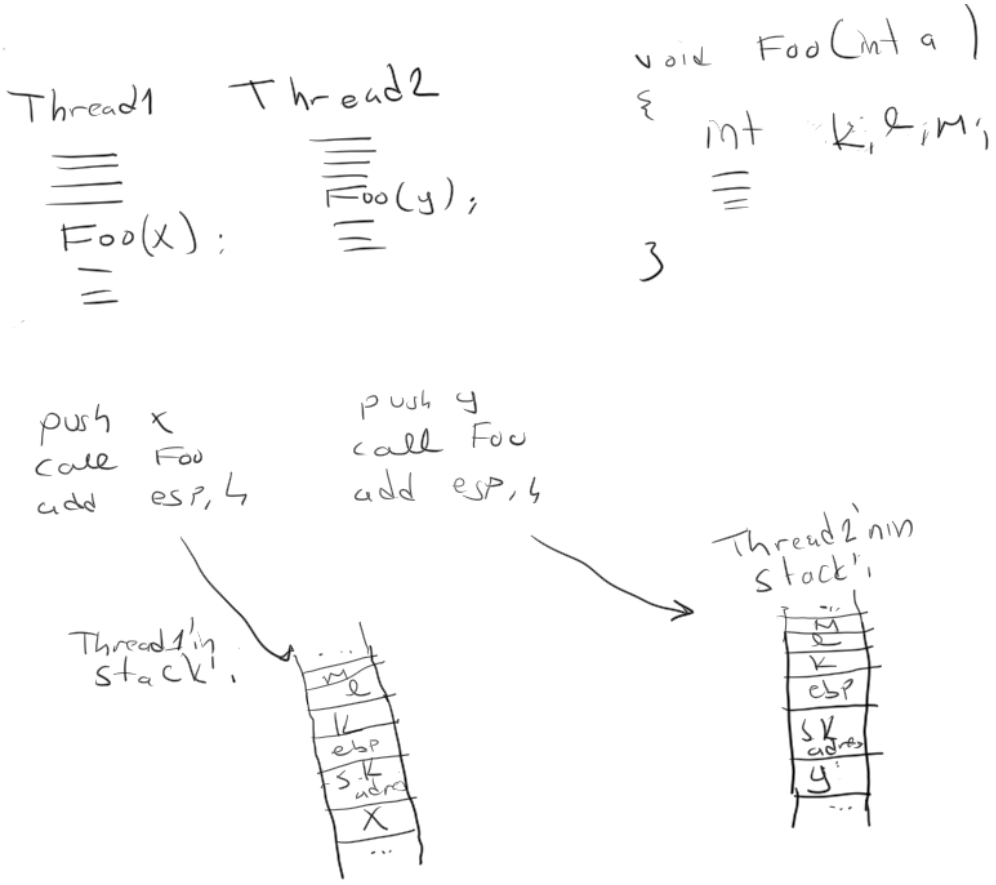


Thread'ler sayesinde bir program birden fazla akışa sahip olabilmektedir. Bu akışların biri beklese bile diğerleri çalışmaya devam eder.

Thread'li sistemlerde bir proses bir thread'le çalışmaya başlar. Bu thread'e prosesin ana thread (main thread) denilmektedir. Diğer thread'ler işletim sisteminin sunduğu sistem fonksiyonlarıyla ya da bu fonksiyonları kullanan kütüphane fonksiyonlarıyla yaratılmaktadır.

Proses ile thread kavramları birbirine karışabilmektedir. Çalışan programın kullandığı tüm kaynaklar ve onun her şeyi proses kavramı ile betimlenir. (Örneğin açılmış dosyalar, prosesin bellek alanı, çalışma dizini, çevre değişkenleri vs.) Thread ise yalnızca bir akış belirtmektedir.

Thread'lerin stack'leri birbirinden ayrılmıştır. Böylece iki farklı thread aynı fonksiyon üzerinde ilerlerken fonksiyonun yerel değişkenleri ve parametre değişkenleri birbirlerine karışmaz. Yani her thread o yerel değişkenin ayrı bir kopyasını kullanıyor durumdadır.



Pekiyi thread'lerin stack'leri nasıl birbirlerinden ayrılabilmektedir? İşte bağlamsal geçiş sırasında işletim sistemi çalışmaya devam edecek thread'in yazmaçlarını CPU'ya yükleyince ESP yazmacını da artık o thread'in stack'ini gösteriyor durumda olur. Böylece artık fonksiyon çağrılırken parametre değişkenleri ve yerel değişkenler o andaki stack'te yaratılacaktır. Her thread'in ayrı bir stack'i olmasına karşın "data" ve "bss" alanları prosese özgüdür ve prosesin tüm thread'leri bu alanları ortak kullanmaktadır. Yani global nesnelerin yaratıldığı "data", "bss" ve "heap" alanlarının thread'e özgü kopyaları yoktur.

Thread'e benzeyen diğer bir kavram da "fiber" kavramıdır. UNIX/Linux sistemlerinde çekirdek tarafından organize edilen bir "fiber" sistemi yoktur. Ancak Microsoft XP'nin sonraki modellerinden itibaren Windows sistemlerine "fiber" özelliğini eklemiştir. "Fiber" işletim sisteminin çizelgelemesine (yani çekirdeğine) dahil olmayan, kütüphane fonksiyonlarıyla sağlanan bir çeşit yapay thread gibidir. İşletim sistemi çekirdeği "fiber"lerin farkında değildir. Çekirdek yalnızca proses ve thread'leri tanımaktadır. "Fiber"ler adeta bir thread'i "cooperative multitask" hale getirmektedir. Örneğin işletim sistemi akışı bizim thread'imize devretmiş olsun. Biz "fiber" sayesinde thread'imiz için ayrılmış olan quanta süresince "Foo" "Bar", "Tar" fonksiyonlarını zaman paylaşımlı olarak çalıştırabiliriz. Akış bir thread'in bir "fiber"inde bloke olursa öbür "fiber"ler çalışamayacaktır. Çünkü "fiber" gerçekleştirimi çekirdek düzeyinde yapılmamaktadır. Halbuki prosesin bir thread'i bloke olduğunda diğer thread'ler çalışmaya devam etmektedir.

10. Sanal Bellek (Virtual Memory) Mekanizması

Sanal bellek (virtual memory) bir programın hepsinin değil, belli kısımlarının fiziksel RAM'e yüklenerek çalıştırılmasını sağlayan bir bellek yönetim tekniğidir. Windows, Linux, Mac OS X gibi sistemler sanal bellek mekanizmasını oldukça etkin bir biçimde kullanmaktadır. Sanal bellek mekanizmasının neye benzediğini ve nasıl işletildiğini şöyle bir örnekle açıklayabiliriz: 10 MB'lik kod ve data/bss alanına sahip çalıştırılabilen bir dosyamız olsun. Biz bu dosyayı çalıştırdığımızda aslında işletim sistemi bu 10 MB dosyanın içerisindeki tüm kod ve data'ları fiziksel RAM'e yüklemez. Onun yalnızca belli bir bölümünü (diyelim 100K'lık bir bölümünü) yükleyerek programı başlatır. Böylece yalnızca bir bölümü fiziksel RAM'e yüklenmiş olan programımız çalışmaya başladıktan bir süre sonra akış kod ya da data bakımından fiziksel bellekte olmayan bir bölgeye gelecektir. İşte tam bu aşamada işlemci programın fiziksel RAM'de olmayan

bir kısma erişmeye çalıştığını anlar ve içsel bir kesme oluşturur. Bu içsel kesme yoluyla işletim sistemi devreye girer, programın erişmek istediği kısmı diskte bularak onu fiziksel RAM'in uygun bir yerine yükler ve çalışmayı devam ettirir. Ta ki yeniden aynı nedenden dolayı bir kesme oluşana kadar...

Intel işlemcilerinde sanal bellek mekanizması segment tabanlı ya da sayfa tabanlı olarak uygulanabilmektedir. Segment tabanlı sanal bellek çok eski bir teknolojidir. Artık hiçbir işletim sistemi tarafından kullanılmamaktadır. Burada biz sayfa tabanlı sanal bellek mekanizması üzerinde duracağız. Diğer işlemci ailelerinde de zaten artık segment tabanlı sayfalama mekanizması mevcut değildir.

İşlemcinin desteği olmadan etkin bir sanal bellek mekanizmasının sağlanması mümkün değildir. Biz burada Intel işlemcilerinde bu sanal bellek mekanizmasının nasıl kurulup işletildiğini ele alacağız.

10.1. Intel İşlemcilerinde Sayfa Tabanlı Sanal Bellek Mekanizmasının İşletilmesi

Intel işlemcilerinde sayfa tabanlı sanal bellek mekanizması işlemcinin "sayfalama (paging)" biriminin desteğiyle yürütülmektedir. Sayfa tabanlı sanal bellek kullanan bir işletim sisteminde bir program çalıştırılmak istendiğinde işletim sistemi programın tamamını değil yalnızca onun bazı sayfalarını fiziksel RAM'e yükleyerek çalışmayı başlatır. İşletim sistemi programa ilişkin fiziksel RAM'e yüklenen sayfaların dizin ve sayfa tablosundaki girişlerinin P bitlerini 1 yapar. Ancak fiziksel RAM'e yüklemediği sayfaların P bitleri 0 olarak set eder. İşte programın çalışması başlatıldığında işlemci doğrusal adresleri fiziksel adreslere dönüştürürken bu P bitlerinin 1 olduğu sayfalarda sorun yaşamaz. Ne zaman program kod ya da data bakımından fiziksel RAM'de olmayan bir sayfaya erişmek istese o sayfanın sayfa tablosundaki P biti 0 olduğundan işlemci tarafından "sayfalama hatası (page fault)" denilen kesme oluşturmaktadır. İşte "sayfalama hatası (page fault)" denilen bu kesme oluştuğunda kontrolü işletim sistemi devralmaktadır. İşletim sisteminin kesmeye yanıt veren kesme kodu da (page fault handler) çalışması kesilen programın hangi sayfaya erişmek istediğine bakar. Erişilmek istenen bu program parçasını diskte çalıştırılabilen dosyanın ya da bir "sayfa dosyasının (page file)" içerisinde arar. Programın o kısmını diskten boş bir fiziksel sayfaya yükler. Sayfa tablosunda kesmeye yol açan sayfa ve/veya dizin girişini düzeltir. (Yani artık o girişin program parçasının yüklendiği fiziksel sayfayı göstermesini sağlar ve o girişin P bitini de 1 yapar.) Sonra akış kesmeye yol açan komuttan devam edecektir. (Intel işlemcilerinde "fault" durumunda stack'e sonraki komutun adresi değil bizzat "fault"a yol açan komutun adresi atılmaktadır. Bu yüzden "sayfalama hatası (page fault)" kesmesinden geri dönen IRET makine komutu çalışmaya sonraki komutla değil kesmeye yol açan komutun kendisiyle devam edecektir. Kesmeler konusu ileride ele alınmaktadır.)

Bu mekanizmayla ilgili gelinen noktaya kadar sorulabilecek sorular ve yanıtları şöyledir:

Soru: Sanal bellek kullanan sistemlerde bir programın ne kadar kısmı fiziksel RAM'e yüklenmektedir?
Yanıt: Küçük bir kısmının yüklendiğini söyleyebiliriz. Ancak başlangıçta tam olarak programın kaç sayfasının fiziksel RAM'e yükleneceği işletim sisteminden işletim sistemine farklılık gösterebilmektedir.

Soru: Bir kısmı yüklenen program çalışmaya başladığında akışın kod ya da data bakımından fiziksel bellekte olmayan kısma erişmesi ne demektir?
Yanıt: Belleğe erişen tüm makine komutları dizin ve sayfa tablosuna başvurmaktadır. Yani sembolik makine dilinde köşeli parantezler içerisinde belirttiğimiz bellek operandlarında işlemci önce bu köşeli parantezlerin içerisindeki değerlerden ona ilişkin doğrusal adresi hesaplar sonra dizin ve sayfa tabloları yoluyla fiziksel RAM'e erişmeye çalışır. İşte bu noktada sayfalama hatası oluşabilmektedir. Örneğin:

```
MOV EAX, [ESI]
```

Burada ESI değeri 0x200080 olsun. Bu doğrusal adrese karşı gelen sayfa tablosu girişinin P biti 0 ise bu doğrusal adresin içinde bulunduğu sayfa fiziksel RAM'de değildir. Bu durumda sayfalama hatası oluşacaktır. Şimdi de kodun bir sayfanın sonuna geldiğini düşünelim. Sonraki komut sonraki sayfada bulunuyor olsun. İşte şimdi yeni komutun bulunduğu (yani EIP yazmacının belirttiği doğrusal adresteki) sayfa fiziksel bellekte değilse kod bakımından yine bellekte olmayan bir bölgeye erişilmiş olmaktadır. Bu durumda da sayfalama hatası oluşacaktır. Şimdi de PUSH işlemi yaptığımızı düşünelim. Fakat ESP yazmacı tam da

fiziksel sayfanın en yukarısını gösteriyor olsun. Artık bu PUSH işleminde ESP yazmacı başka bir sayfayı gösterir durumda olacaktır. İşte o sayfa fiziksel RAM'de yoksa yine sayfalama hatası oluşacaktır.

Soru: Neden işletim sistemi bir programın tamamını değil de onun küçük bir kısmını yükleyip sanal bellek mekanizmasını kullanmaktadır? Başka bir deyişle sanal bellek mekanizmasına neden gereksinim duyulmaktadır?
Yanıt: Şüphesiz fiziksel RAM çok büyük olsaydı bu mekanizmaya çok daha az gereksinim duyulurdu. Ancak sanal bellek sayesinde sistemimizdeki fiziksel RAM kısıtlı olsa bile biz bundan daha büyük çok sayıda programı aynı anda çalıştırabiliriz. Çünkü bu programların hepsi değil yalnızca küçük kısımları yüklenerek bunlar çalıştırılabilecektir.

Soru: Sanal bellek mekanizmasında her sayfalama hatası oluştuğunda bir disk işlemi yapıldığına göre programların çalışması yavaşlamaz mı?
Yanıt: Evet yavaşlar. Bu konuda yapılacak bir şey yoktur. İşletim sisteminin ileride de değineceğimiz gibi akıllıca bir politikası sayfalama hatası miktarını azaltabilmektedir. Disklerin yavaş olması ve fiziksel RAM'in yetersiz olması buradaki yavaşlığı hepten dayanılmaz boyuta getirebilmektedir. Bugün artık yaygınlaşmış olan SSD diskler hızları nedeniyle sanal bellek mekanizmasının da daha hızlı yürütülmesine olanak sağlamaktadır.

Soru: Sanal bellek mekanizmasını kullanan sistemlerdeki optimum fiziksel RAM miktarı ne kadar olmalıdır?
Yanıt: Bu durum devirden devire değişebilmektedir. Şüphesiz fiziksel RAM'i artırdığımızda bu konudaki performansı yükseltiriz. Ancak bu yükseltme doğrusal değildir. Yani belli bir miktardan daha fazla fiziksel RAM'e sahip olmamız bize ciddi bir hız kazancı sağlamayabilir. Fiyat da değerlendirildiğinde kursun verildiği tarih göz önüne alınırsa Windows sistemleri için optimum RAM miktarının 8 GB ya da 16 GB olduğu söylenebilir.

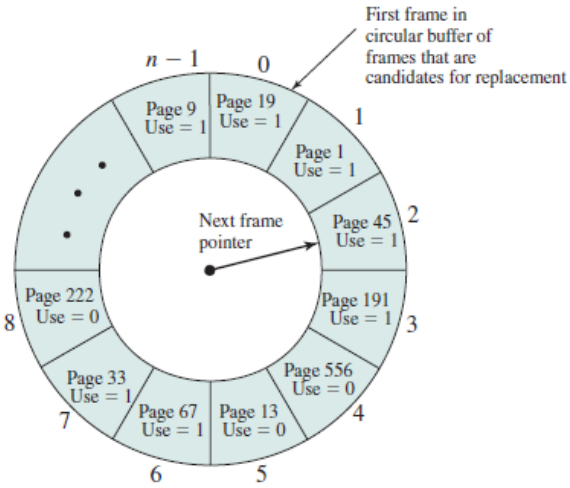
Görüldüğü gibi sanal bellek mekanizması işlemcinin desteğiyle işletim sistemi tarafından kurulup yönetilmektedir. İşletim sistemleri sanal bellek mekanizmasını yönetmek için çekirdek alanı içerisinde çeşitli veri yapıları organize ederler. Bu veri yapıları da çeşitli algoritmalarla işletilmektedir. Örneğin işletim sistemleri genel olarak fiziksel bir sayfayı bir yapıyla temsil ederler. (Linux'ta "page" isimli yapı bir fiziksel sayfayı temsil etmektedir.) Dolu sayfalarla boş sayfaları da birer veri yapısında tutarlar. Tabii dolu (yani kullanılan) bir fiziksel sayfanın hangi proses tarafından kullanıldığı da bu veri yapılarında tutulmaktadır. Boş sayfalarda ardışıllık (reference locality) genel olarak önemlidir. Boş sayfalardan tahsisat işlemlerinde pek çok işletim sistemi "ikiz blok (buddy allocator)" sistemini tercih etmektedir. İşletim sistemi bir sayfa gerektiği zaman önce onu serbest sayfaların tutulduğu veri yapısından elde etmeye çalışır. Eğer bu veri yapısından serbest durumda olan hiçbir sayfa yoksa kullanılan sayfalardan bazıları onu kullanan proseslerden alınarak gereksinim karşılanmaktadır. Pekiyi fiziksel RAM tıka basa doluysa yani işletim sisteminin boş sayfaları tutan veri yapısında hiçbir sayfa yoksa hangi sayfalar prosesler'den kopartılıp ihtiyaç sahibi proses'e verilecektir? İşte bu işleme sayfa yer değiştirmesi (page replacement) denir. Sayfa yer değiştirmesi tipik birkaç algoritma kullanılabilmektedir.

Öncelikle ideal sayfa çıkartma algoritmasının nasıl olabileceğini düşünelim. Eğer biz hangi fiziksel sayfaya en sonra (yani zaman olarak daha geç) erişileceğini bilsek en geç erişilecek sayfayı boşaltmamız uygun olur. Gerçekten her defasında en sonra erişilecek sayfayı boşaltmak optimal yöntemdir. Fakat tabii hangi sayfalara ne zaman erişileceği gibi bir bilginin önceden bilinmesi genel olarak mümkün değildir

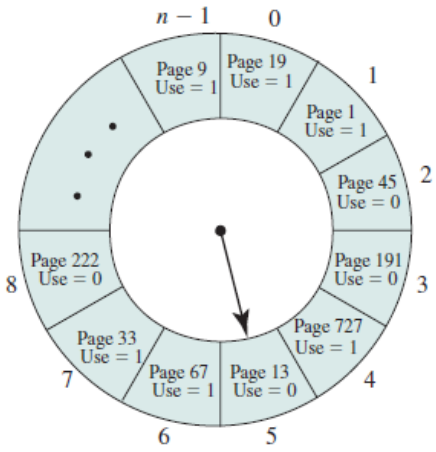
Pek çok araştırma ve simülasyon çalışması son zamanlarda az kullanılan fiziksel sayfaların yakın gelecekte de az kullanılacağını göstermektedir. O halde işletim sistemi eğer fiziksel sayfaların son zamanlardaki kullanımı üzerinde bir istatistik yaparsa son zamanlarda en az kullanılmış olan sayfa boşaltılabilir. Bu yönteme LRU (Least Recently Used) politikası denilmektedir. Pekiyi işletim sistemleri fiziksel sayfalar üzerinde böyle bir istatistiği etkin bir biçimde yapabilir mi? Maalesef böyle bir istatistiğin etkin bir biçimde yapılabilmesi ancak işlemcinin ciddi bir desteği ile mümkün olabilmektedir. (Yani örneğin işlemci her sayfa için ona erişildiğinde bir sayacı artırabilir. İstatistiği bizzat kendisi yapıyor olabilir.) Ancak pek çok işlemci LRU yönteminin etkin uygulanması için bir mekanizmaya sahip değildir.

Pekiyi şimdi de en basit sayfa yer değiştirme (page replacement) algoritmasının nasıl olabileceğini düşünelim. Dolu sayfalardan rastgele bir tanesi seçilerek boşaltılabilir. Bu yönteme "rastgele sayfa yer değiştirmesi (random page replacement)" denilmektedir. Bu yöntemden biraz daha iyisi tüm dolu sayfaların bir FIFO kuyruk sisteminde tutulması olabilir. Sayfalar yeni kullanıma başladığında bu kuyruğun sonuna eklenir. Sayfa boşaltmak istendiğinde de kuyruğun başından alınır. Bu yönteme "FIFO yer değiştirmesi (FIFO page replacement)" denilmektedir.

Anımsayacağınız gibi Intel işlemcileri bir sayfaya erişildiğinde o sayfaya ilişkin sayfa girişinin (page entry) onun A bitini 1 yapmaktadır. Bu davranış yalnız Intel'de değil pek çok işlemci ailesinde de benzer şekilde vardır. İşte bu A bitinden faydalanılarak LRU kadar olmasa da hiç olmazsa FIFO'dan daha iyi bir yer değiştirme politikası izlenebilir. Bu yöntemlere "saat sayfa yer değiştirmesi (clock page replacement)" yöntemleri denilmektedir. Sayfa yer değiştirmesi yöntemlerinin birkaç uyarlaması vardır. Bu yöntemde temel olarak dolu (kullanılan) sayfalar bir FIFO döngüsel kuyruk sisteminde saklanır. Sayfa boşaltılacağı zaman kuyruğun başından itibaren dolu sayfaların sayfa girişlerindeki A bitlerine bakılarak ilerlenir. Eğer kuyrukta sıradaki sayfaya ilişkin sayfa girişinin A biti 1 ise bu bit 0 yapılır, ancak bu sayfa boşaltım için seçilmez; kuyrukta ilerlemeye devam edilir. A biti 1 olduğu sürece hep onların A bitleri 0 yapılarak kuyrukta ilerlenir. A biti 0 olan ilk sayfa boşaltım için seçilir. Yeni bir sayfa dolu hale geldiğinde de bu sayfa kuyruğun sonuna A biti 0 olacak biçimde eklenmektedir. Pekiyi bu yöntemin anlamı nedir? Burada işletim sistemi bir sayfanın A bitini sıfırlayıp ona bir tur sonra baktığında eğer onun A bitini hale 0 görürse bu sayfaya son zamanlarda erişilmediği sonucunu çıkartır. Bu durumda onun yakın gelecekte de kullanılma olasılığı azdır. Aşağıdaki şekil "Wiliam Stallings"in "Operating System Internals and Design Principles" kitabından alınmıştır:



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Figure 8.16 Example of Clock Policy Operation

Burada döngüsel kuyruk sistemi şekilsel olarak gösterilmiştir. İlk durumda kuyruğun başında 19 numaralı sayfa vardır. Burada A biti (şekilde "Use" olarak gösterilmiş) 1 olana kadar onlar sıfırlanarak ilerlenmiştir. A biti 0 olan ilk sayfa 556 numaralı sayfadır. Bu sayfanın boşaltılmasına karar verilmiştir. Bu boşaltma işleminden sonra kuyruğa (kuyruğun sonuna) 727 numaralı sayfa da eklenmiştir.

Şimdi boşaltılacak sayfanın seçildiğini düşünelim. Boşaltılacak fiziksel sayfa sayfa eğer kirlenmişse (yani onun üzerinde bir güncelleme yapılmışsa) işletim sisteminin onu yeniden diske yazması gerekir. (Tabii kirlenmemiş bir sayfanın yazılmasına gerek yoktur. Çünkü o zaten bir biçimde diskte bulunmaktadır.) İşletim sistemleri terminolojisinde diskteki bir sayfanın fiziksel RAM'e çekilmesi sürecine İngilizce "swap in", fiziksel bellekteki bir kirlenmiş bir sayfanın yeniden diske yazılması sürecine de "swap out" denilmektedir.

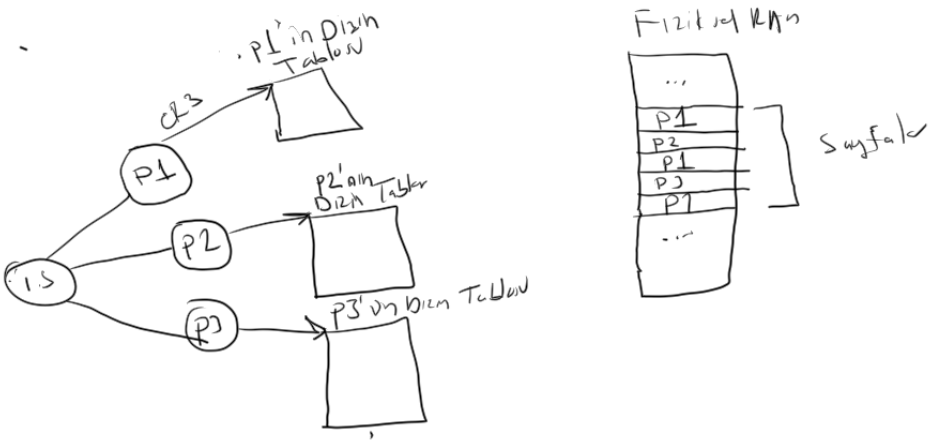
Pekiyi sanal bellek mekanizması altında bir program çalıştırılırken bu mekanizmanın sürdürülmesi için diskte nasıl bir organizasyon yapılmaktadır? Çalıştırılabilen dosyanın mümkün mertebe "swap" amacıyla kullanılması anlamlı olabilir. Ne de olsa programın kod bölümü neredeyse hiç değiştirilmemektedir. İşletim sistemi ne zaman isterse "swap in" yaparken bu sayfaları çalıştırılabilen dosyadan yeniden alabilir. Ancak eğer prosesin fiziksel RAM'deki sayfası kirlenmişse ve bu sayfa "swap out" yapılacaksa artık bu sayfa çalıştırılabilen dosyanın içerisine yazılamayacağından dolayı ayrı bir dosyanın içerisine yazılmak durumundadır. İşletim sisteminin oluşturduğu bu tür dosyalara "sayfa dosyaları (page files)" denilmektedir. Sayfa dosyalarının organizasyonu nerede ve nasıl oluşturulacağı işletim sisteminden işletim sistemine hatta versiyondan versiyona değişebilmektedir. Bazı işletim sistemleri bu sayfa dosyalarını ayrı bir "disk bölümünde (disk partition)" organize ederler. Örneğin Linux sistemleri bu konuda oldukça esnek tasarlanmıştır. Bu sistemlerde birden fazla dosya ya da disk bölümü çekirdek tarafından sayfa dosyası olarak kullanılabilmektedir.

Pekiyi sanal belleğin tüm prosesler için toplam kapasitesi ne kadardır? Aslında bu kapasite çok çok geniştir. Çünkü neticede çok büyük yüzlerce proses küçük bir RAM'de çalıştırılabilmektedir. Ancak yine de bazı işletim sistemlerinde toplam sanal bellek miktarı sistem yöneticisi tarafından ayarlanabilmektedir.

Proseslerin Bellek Alanlarının Sayfalama Mekanizması Yoluyla İzole Edilmesi

Windows, Linux ve Mac OS X gibi sistemler sayfalama mekanizması yoluyla proseslerin fiziksel bellek alanlarını tamamen birbirlerinden ayırmaktadır. Bu sayede bir proses zaten istese de başka bir prosesin kod ya da verilerinin bulunduğu fiziksel bellek alanına erişememektedir. Pekiyi bu durum nasıl sağlanmaktadır?

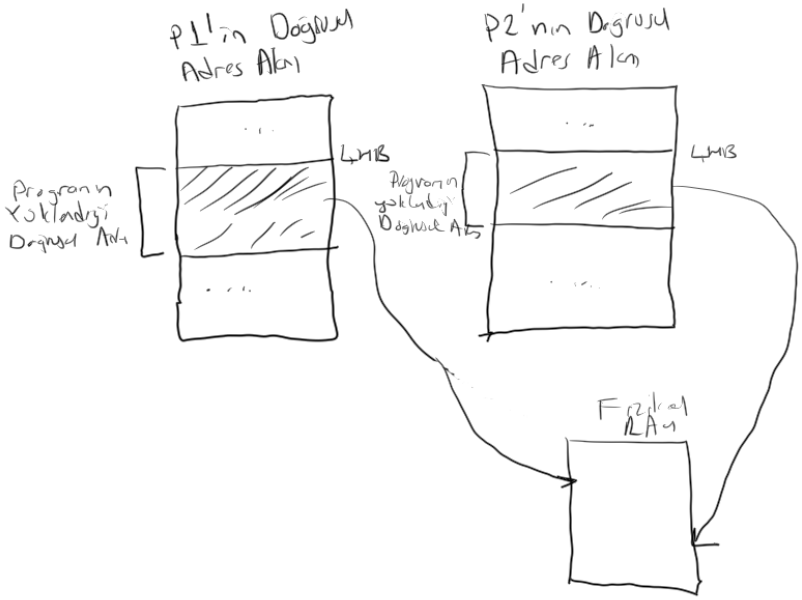
İşletim sistemleri her proses için ayrı bir dizin tablosu ve sayfa tabloları tutmaktadır. Proseslerarası geçiş yapıldığında işlemcinin CR3 yazmacı da geçiş yapılan yeni prosesin dizin tablosunun yerini gösterecek biçimde değiştirilmektedir.



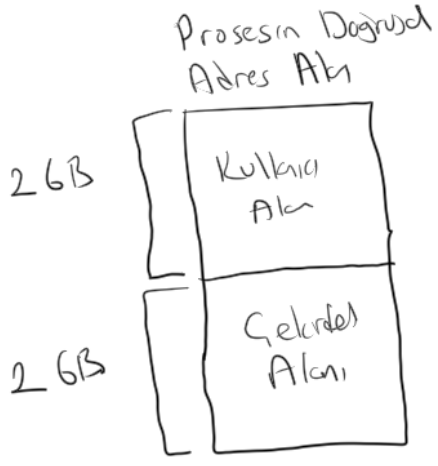
Bir proses çalışırken işlemcinin CR3 yazmacı o prosesin dizin tablosunu göstermektedir. Dolayısıyla

doğrusal adresin fiziksel adrese dönüştürülmesinde bu dizin tablosu ve ona bağlı olan sayfa tabloları kullanılır. İşletim sistemi zaten farklı prosesler için aynı fiziksel sayfayı eşlememektedir. Tabii bu çalışma biçimi sanal bellek mekanizmasıyla da birleştirilmiş durumdadır. Yani proseslerin tüm kod ve verileri fiziksel belleğe yüklenmez onların yalnızca belli kısımları yüklenir.

32 bit Windows, Linux ve Mac OS X gibi sistemlerde sanki prosesler tek başlarına belleğe yüklenmiş ve çalışıyormuş gibi bir durum oluşmaktadır. Örneğin Windows'ta çalışan iki proses de sanki 4 MB'den itibaren 4 GB fiziksel RAM'e tek başına yükleniyormuş gibi çalışır. Fakat aslında sayfalama ve sanal bellek mekanizmaları sayesinde bu iki proses aynı doğrusal adres alanına sahip olsa da farklı fiziksel adresleri kullanıyor durumdadır. Bu nedenle bu sistemlerde biz farklı proseslerde aynı doğrusal adreslere baktığımızda orada farklı şeyler görebiliriz. Çünkü iki farklı programın aynı doğrusal adresleri aslında o anda fiziksel RAM'de farklı yerleri belirtmektedir.



Pekiyi prosesler birbirlerinden sayfalama mekanizması yoluyla izole edilmişlerdir. Fakat işletim sisteminin kodları nerededir? İşte bir proses çalışırken işletim sisteminin de o anda onun doğrusal adres alanı içerisinde bulunması gerekmektedir. Windows, Linux ve Mac OS X gibi sistemlerde işletim sisteminin kodları ve verileri proseslerin doğrusal adres alanlarının aynı yerlerinde bulunmaktadır. Başka bir deyişle her prosesin dizin ve sayfa tablolarının belli kısımları ortaktır. Örneğin 32 bit Windows işletim sistemlerinde prosesin doprusla adres alanının düşük anlamlı 2 GB'si "kullanıcı alanı (user space)", yüksek anlamlı 2 GB'si ise "çekirdek alanı (kernel space)" olarak ayrılmıştır.



Böylece proseslerarası geçiş oluştuğunda dizin ve sayfa tablolarının yalnızca düşük anlamlı 2 GB'lik kısımları bu değişimden etkilenmektedir. Başka bir deyişle tüm proseslerin dizin ve sayfa tablolarının

yüksek anlamlı 2 GB'lik kısımları aynı fiziksel sayfalara yönlendirilmiş durumdadır. Bu durumda 32 bit Windows sistemlerinde bir program en fazla 2 GB uzunlukta olabilmektedir. (Özel bir durum olarak Windows çekirdeğinin son 1 GB'yi kullanması sağlanabilmektedir. Bu da kullanıcı alanının 3 GB'ye çıkması anlamına gelir.) 32 bit Linux ve Mac OS X sistemlerinde de benzer durum söz konusudur. Bu sistemlerde kullanıcı alanı 3GB, çekirdek alanı 1 GB yer kaplamaktadır.

Proseslerin bellek alanlarının izolasyonlarına yönelik tipik soru v yanıtlar şunlardır:

Soru: Proseslerin bellek izolasyonları temel olarak nasıl sapğlanmaktadır?
Yanıt: Her prosesin dizin ve sayfa tabloları biribirinden farklıdır. Proseslerarası geçişte CR3 yazmacı geçilen prosesin dizin tablosunu gösterecek hale getirlir. Proslerin bu farklı dizin ve sayfa tablolarının girişleri farklı fiziksel sayfalara yönlendirilmiştir.

Soru: Farklı proseslerde debugger eşliğinde aynı doğrusal adrese baktığımızda neden aynı değerleri görmüyoruz?
Yanıt: Çünkü iki prosesde aynı doğrusal adres farklı fiziksel adreslere yönlendirilmiştir. Örneğin iki prosesin birisinde 0x450000 adresi aslında başka bir fiziksel sayfada yer belirtirken diğerinde farklı bir fiziksel sayfada yer belirtiyor durumdadır.

Soru: Bu durumda biz Windows, Linux ve Mac OS X gibi sistemlerde başka bir prosesin bellek alanına erişemez miyiz?
Yanıt: evet erişemeyiz. Çünkü o andaki kullanılan dizin ve sayfa tablosu bizim kod ve verilerimizi gösterecek biçimde organize edilmiştir.

Soru: Bu sistemlerde bir proses'te rastgele bir adrese erişmek istediğimde ne olur?
Yanıt: Eğer bu rastgele adres bizim programımızın bir kısmına ilişkinse bir sorun olmadan erişim gerçekleşebilir. Ancak bu adrese ilişkin sayfanın sayfa tablosundaki girişinin P biti 0 ise bu sayfa henüz fiziksel RAM'de yoktur. İşte işletim sistemi bu durumda hemen diskten RAM'e sayfayı aktarmaya çalışmaz. Önce erişilmek istenen yerin gerçekten o prosesin tahsis ettiğinden yani diskte onun bir karşılığın olduğundan emin olmak ister. Eğer böyle değilse prosesi cezalandırarak sonlandırır. Anımsanacağı gibi işletim sistemi kendisini o anda çalışmakta olan prosesten sayfa koruması yoluyla korumaktadır. Yani örneğin erişilen adres eğer işletim sisteminin kod ya da verilerinin bulunduğu bir adres ise bu sayfalar "supervisor" modda olduğu için erişim sırasında kesme oluşacaktır. Dolayısıyla işletim sistemi de prosesi sonlandıracaktır.

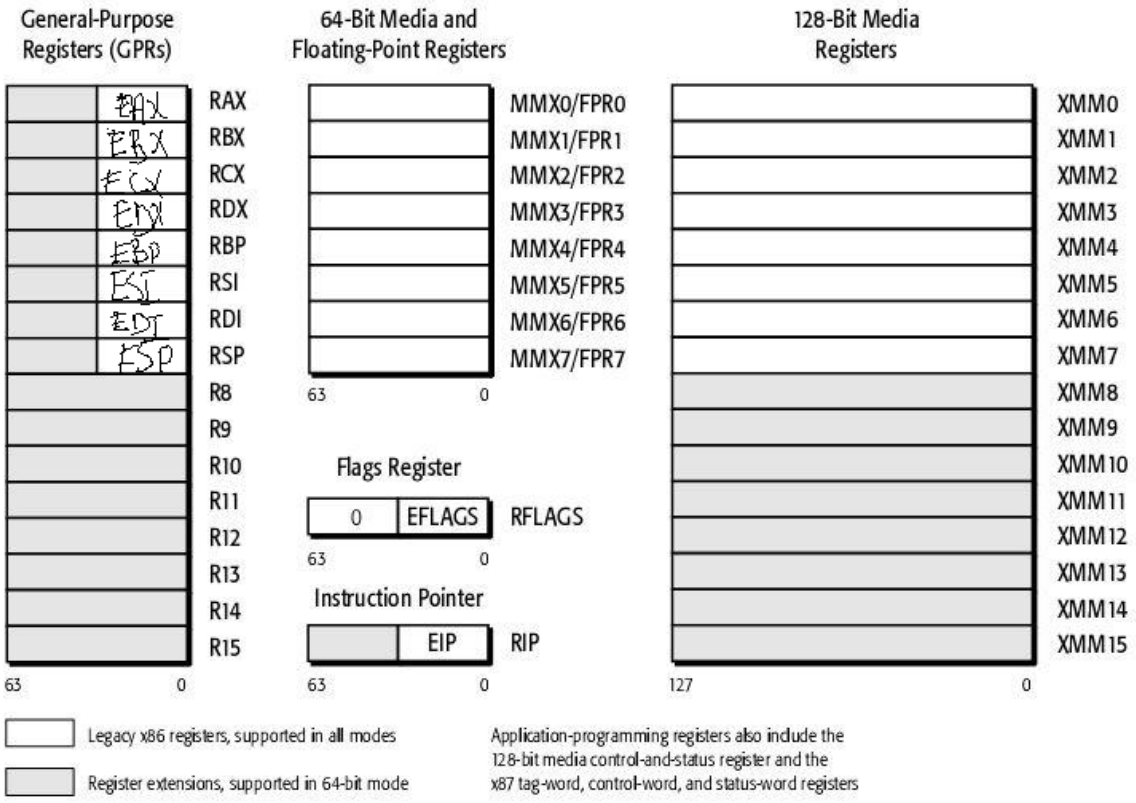
Pekiyi Windows, Linux ve Mac OS X gibi sistemlerde proseslerin bellek alanları tamamen izole edildiğine göre proseslerarası haberleşme (interprocess communication (IPC)) nasıl sağlanmaktadır? Proseslerarası haberleşme bir prosesin diğerine bir grup byte'ı iletmesi sürecine denilmektedir. İşte işletim sistemlerinin proseslerarası haberleşme için sunduğu çeşitli mekanizmalar vardır. (Örneğin mesaj kuyrukları, borular, paylaşılan beşşek alanları gibi). "Paylaşılan bellek alanları (shared memory)" yönteminde işletim sistemi istek üzerine iki prosesin farklı doğrusal sayfalarını aynı fiziksel sayfaya yönlendirmektedir. Bu durumda proseslerden biri bir doğrusal adres kullanarak o adrese birşey yazdığında diğer proses o yazılanları başka bir doğrusal adresten elde edebilmektedir. Çünkü bu iki prosesin sayfa tablosunda o doğrusal adresin yönlendirildiği fiziksel adresler aynıdır.

11. Intel Mimarisinde 64 Bit Çalışma

Anımsanacağı gibi Intel mimarisinin 64 bite yükseltilmesi ilk kez Intel tarafından değil AMD firması tarafından yapılmıştır. Intel AMD'nin tasarımını alarak küçük değişikliklerle kendi tasarımını da yapmıştır. AMD'nin 64 bit tasarımına AMD64, Intel'in 64 bit tasarımına ise X64 denilmektedir. İki tasarım hemen hemen birbirinin aynısıdır. Aralarındaki küçük farklılıklar sistem yazılımı geliştiren firmakar tarafından uyumun bozulmaması için zaten pek kullanılmamaktadır. Biz burada kolaylık olsun diye bu 64 bit tasarımlar için X64 terimini kullanacağız.

X64 tasarımının temel yazmaç yapısı giriş bölümlerinde verilmişti. Burada yeniden bir anımsatma yapmak istiyoruz.

64 bite geçildiğinde Intel yazmaçları da 64 bite yükseltmiştir. Bunun için yazmaç isimlerinin başındaki E öneki 64 bit için R haline (R muhtemelen "Register" sözcüğünden geliyor) getirilmiştir. Örneğn RAX, RBX, RCX, RDX gibi. X64 işlemcilerinin yazmaç yapısı şöyledir:



Şekilden de görüldüğü gibi 64 bit sistemde şu ek farklılıklar söz konusudur:

- 64 bite geçildiğinde Intel genel amaçlı yazmaçlara 8 tane daha eklemiştir. Bunlar R8-R15 yazmaçlarıdır.

- 64 bit modda genel olarak önceki modlardaki 32 bit yazmaçlar, 16 bit yazmaçlar ve 8 bit yazmaçlar kullanılabilmektedir. (Yani örneğin biz 64 bit modda RAX yazmacını da EAX yazmacını da AX yazmacını da AL yazmacını da kullanabiliriz). Ancak 64 bit "long mode"da giriş bölümlerinde de belirtildiği gibi bazı yazmaçların kullanımları için komutu uzatan REX önekleri gelmektedir.

- Intel'in Pentium serisine 64 bitlik MMX komut kümesi eklenmişti. Bu komutlar MMX yazmaçlarını kullanıyordu. 64 bitte özel 128 bitlik XMM komut kümesi ve yazmaçları da mimariye dahil edilmiştir. (Ayrıca 64 bit işlemcilerin sonraki modellerde vektörel işlem yapan yine 16 tane 256 bit YMM yazmaçları da mimariye eklendi. YMM yazmaçlarının düşük anlamlı 128 biti XMM yazmaçları ile çakışmaktadır. Yukarıdaki şekile YMM yazmaçları dahil edilmemiştir.)

255	127	0
	XMM0	YMM0
	XMM1	YMM1
	XMM2	YMM2
	XMM3	YMM3
	XMM4	YMM4
	XMM5	YMM5
	XMM6	YMM6
	XMM7	YMM7
	XMM8	YMM8
	XMM9	YMM9
	XMM10	YMM10
	XMM11	YMM11
	XMM12	YMM12
	XMM13	YMM13
	XMM14	YMM14
	XMM15	YMM15

Yine giriş bölümünde gördüğümüz X64 çalışma modları hakkında da bir anımsatma yapalım. Bu işlemciler yine reset edildiğinde "gerçek moddan (real mode)" çalışmaya başlarlar. Bunlar tamamen 32 bit işlemci gibi de çalışabilmektedir. Bu moda Intel "Legacy Mode" demektedir. Nihayet X64 işlemcilerinin tüm olanaklarının kullanıldığı bir "long mode"u da vardır:

Mod	Anlamı
Gerçek Mod (Real Mode)	Bu mod işlemcinin 16 bit 8086 gibi çalıştığı moddur. 64 bitlik Intel işlemcileri reset edildiğinde çalışma gerçek moddan başlatılmaktadır.
32 Bit Modu (Legacy Mode) Alt Modlar: - V86 modu (V86 Mode) - Korumalı Mod (Protected Mode)	Bu mod 80386 ve Pentium uyumlu moddur. Bu modda işlemci 32 bit bir Pentium işlemcisi gibi kullanılır. AMD64 ve X64 işlemcilerine 32 bit işletim sistemi yüklendiğinde bu işletim sistemleri 64 bit işlemcileri bu modda kullanır.
Long Mod (Long Mode) Alt Modlar: - "Compatibility Long Mode" - "64 Bit Long Mode"	"Compatibility Long mode" 64 bit uygulamalarla 32 bit uygulamaların zaman paylaşımlı biçimde beraber çalıştırılmaları için kullanılan moddur. 64 bit işletim sistemleri 32 bit uygulamaları çalıştırırken işlemciyi "compatibility long mode"da çalıştırmaktadır. "64 bit long mode" ise 64 bit işlemcilerinin 64 bit yeteneğinin kullanılabildiği çalışma modudur. "long mode"da 16 bit programlar çalıştırılamamaktadır. (Böylece örneğimn biz 64 bit Windows sistemlerinde 16 bitlik MS-DOS programlarını çalıştıramayız.)

64 bitlik Windows, Linux ve Mac OS X gibi sistemler X64 işlemcilerini "long mode"da çalıştırmaktadır. Bu "long mode"da çalıştırma sırasında işlemci geçici süre 32 bit moda sokulabilmektedir. "long mode"daki bu alt 32 bitlik moda "compatibility long mode" denilmektedir. Böylece Bu işletim sistemleri yalnızca 64 bit programları değil 32 bit programları da 64 bit programlarla birlikte zaman paylaşımlşı çalıştırabilmektedir.

11.1. NASM ile 64 Bit Programların Derlenmesi ve Bağlanması

64 bit programlar için NASM kaynak dosyasının başında [BITS 64] direktifinin de bulunması gerekmektedir. Windows, Linux ve MAC OS X sistemlerinde 64 bit programlardaki isim dekorasyonları ve çağırma biçimleri de değişmektedir. Fakat öncelikle biz 64 bit sembolik makine dili programlarının Windows'ta ve Linux'ta nasıl derlenip bağlanacağı üzerinde duracağız.

11.1.1 64 Bit Sembolik Makine Dili Programlarının 64 Bit Windows Sistemlerinde Derlenmesi ve Bağlanması

64 bit Windows sistemlerinde NASM programı ile derleme yaparken komut satırında "-f win64" seçeneğinin kullanılması gerekir. Bu seçenek ile NASM Windows'ta çıktı olarak 64 bit COFF formatı üretecektir. İleride görüleceği gibi 64 bit hem Windows hem de Linux ve Mac OS X sistemlerinde isim dekorasyonu ve çağırma biçimleri de farklılaşmaktadır.

64 bit Windows sistemleri için önemli bir uyarı yapmak istiyoruz. 64 bit Windows sistemlerinde Microsoft C derleyicisi olan "cl.exe" programını ve "link.exe" bağlayıcını yine aynı isimlerle fakat farklı dizinlerde bulundurmaktadır. Bu durumda biz komut satırında "cl.exe" ya da "link.exe" programını kullandığımızda bunların 32 bit mi yoksa 64 bit mi olduğunu anlamamız gerekir. eğer bunların bulunduğu dizinler biliniyorsa bu dizinler PATH çevre değişkenine eklenerek ilgili versiyonun çalışması sağlanabilir. Ancak Microsoft bu karışıklığı çözmek için Visual Studio paketi ile birlikte PATh çevre değişkenleri ayarlanmış "komut satırı (cmd.exe)" programları sunmuştur. Böylece biz örneğin "VS2015 X86" ile başlayan komut satırı programı ile komut satırına geçersek burada kullanacağımız "cl.exe" ve "link.exe" 32 bit versiyonlar olacaktır. Eğer biz "VS2015 X64" ile başlayan komut satırını kullanırsak bu durumda bu programların 64 bit versiyonları devreye girecektir. 64 bit derleme ve bağlama için Visual Studio paketindeki "VS2015 X64" ile başlayan komut satırını kullanmayı unutmayınız. Ayrıca Windows'taki bazı DLL'lerin geleneksel isimlerinin sonunda 32 soneki bulunmaktadır. (Örneğin KERNEL32.LIB, USER32.LIB gibi) Windows bu kütüphanelerin 64 bit versiyonlarını da maalesef 32 soneki ile hazırlamıştır. Bu durumda biz 64 bit bir programı link ederken komut satırında KERNEL32.LIB dosyasını görürseniz şaşırmayınız. Bu kütüphane aslında isminin zannettirdiği gibi 32 bit kütüphane değildir. 32 bit kütüphanenin 64 bit versiyonudur. Fakat ismi yine eskisi gibi kalmıştır.

Aşağıda Windows'ta 64 bit "Merhaba Dünya programını görüyorsunuz:

```
; HelloWorld.asm

[BITS 64]

SECTION .data

msg               db   'Merhaba Dunya', 10
msg.written       dd   0

SECTION .text
    global start
    extern GetStdHandle, WriteFile, ExitProcess

start:
    sub      rsp, 32
    mov      rcx, -11
    call     GetStdHandle

    mov      rcx, rax
    mov      rdx, msg
    mov      r8, 14
    mov      r9, msg.written
    push 0
    call     WriteFile

    mov      rcx, 0
    call ExitProcess

    add      rsp, 32
    ret
```

Program aşağıdaki gibi derlenip link edilebilir:

```
nasm -f win64 HelloWorld.asm
link /entry:start /subsystem:console HelloWorld.obj Kernel32.lib
```

11.1.2 64 Bit Sembolik Makine Dili Programlarının 64 Bit Linux Sistemlerinde Derlenmesi ve Bağlanması

Anımsanacağı gibi 64 bit Linux sistemlerinde gcc ve g++ derleyicileri zaten default olarak 64 bit derleme yapmaktadır. Bu sistemlerde 32 bit derleme yapmak için "–m32" seçeneğinin kullanıldığını anımsayınız. NASM ile derleme işlemi Linux sistemlerinde de aynı biçimde yapılır ancak "–f elf64" komut satırı seçeneğinin kullanılması gerekir. Bağlama işlemi için yine ld bağlayıcı benzer biçimde kullanılır. 64 bit Linux sistemlerinde "ld" bağlayıcıları da default olarak 64 bitti. Bunlarda 32 bit bağlama işlemi için "-m elf_i386" seçeneği kullanılıyordu.

Linux sistemlerinde 64 bit NASM sembolik makine dili programı şöyle olabilir:

```
[BITS 64]

SECTION .data

msg         db  "Merhaba Dunya", 10

SECTION .text
    global _start

_start:
    mov     eax, 4
    mov     ebx, 1
    mov     ecx, msg
    mov     edx, 14
    int     80h

    mov     eax, 1
    mov     ebx, 0
    int     80h
```

Görüldüğü gibi Linux'ta 32 bit "Merhaba Dünya" programı ile 64 bit "Merhaba Dünya" programı arasında pek bir fark yoktur. Derleme ve bağlama işlemleri şöyle yapılabilir:

```
nasm -f elf64 helloworld.asm
ld -o helloworld helloworld.o
```

11.2. X64 Mimarisinde 64 Bit embolik Makine Dili Kullanımı

Yukarıda da belirtildiği gibi 64 bit Intel ve AMD işlemcilerinde 64 bitin tüm avantajlarını kullanabilmek için işlemcinin "long mode" denilen moda geçirilmesi gerekmektedir. Bu işlemciler giriş bölümlerinde de belirtildiği gibi hem 16 bit bir 8086 işlemcisi gibi, hem 32 bit bit Pentium işlemcisi gibi hem de tam kapasite bir 64 bit işlemci gibi çalışabilmektedir. 64 bit Windows, Linux ve Mac OS X sistemleri bu işlemcileri 64 bit "long mode"da çalıştırmaktadır.

64 bite geçildiğinde programcı artık tek hamlede 64 bitlik tamsayı işlemlerini yapabileceğini bilmelidir. Örneğin:

```
add     rax, rbx
```

Burada 64 bit iki tamsayı tek hamlede toplanmıştır. Örneğin:

```
mov     rax, [val]
```

Burada da bellekte val adresinden başlayan 64 bit bilgi RAX yazmacına yüklenmiştir.

Genel olarak 64 bit programlamada biz hem 64 bitlik yazmaçları, hem 32 bitlik yazmaçları, hem 16 bitlik yazmaçları hem de 8 bitlik yazmaçları kullanabilmekteyiz. 64 bit yazmaçların kullanımları sırasında bazı REX önekleri komuta dahil olmaktadır. Bu konu komutların ikilik sistemdeki karşılıklarının ele alındığı bölümde açıklanacaktır. Ayrıca 64 bit yazmaçların yalnızca "long mode" da kullanılabildiğini de anımsayınız.

X64 mimarisinde daha önceden de belirtildiği gibi 8 genel amaçlı yazmaç daha eklenmiştir. Bunlar R8, R9, R10, R11, R12, R13, R14, R15 biçiminde isimlendirilmiştir.

X64 mimarisinde yine matematik işlemci komutları aynı biçimde kullanılmaktadır. Ancak 32 Pentium'un belli bir modelinden sonra mimariye eklenmiş olam XMM yazmaçları ve SSE komutları temel toplama, çarpma ve bölme gibi gerçek sayı işlemlerini hiç matematik işlemciye gereksinim duymadan yapabilmektedir.

64 bit mimaride doğal stack elemanları da 64 bittir. Ancak biz stack üzerinde yine 32 bit ve 16 bit PUSH ve POP işlemleri yapabiliriz.

64 bit mimaride artık adresler 64 bit (yani 8 byte'tır). Bu nedenle C'deki göstericiler de 64 bit mimaride artık 4 byte değil 8 byte uzunluktadır. Kullanılan sanal bellek miktarı da teorik olarak çok yükseltilmiştir. Ancak her ne kadar işlemci 64 bitse de Intel ve AMD bunun çok büyük olması nedeniyle fiziksel adres alanını 4 Petta byte (2^{52}byte) sanal adres alanını ise 256 Tera byte (2^{48}) byte'a tutmuşlardır. Ayrıca işletim sistemleri de birtakım çekirdek veri yapılarındaki tabloları büyütmemek için daha küçük limitler kullanmaktadır. (Örneğin 64 bit Windows sistemleri prosesler için sanal bellek alanını 16 TB'de tutmaktadır. Bunun 8 TB'si "user space" diğer 8 TB'si de "kernel space" biçimindedir.)

64 Bit AMD ve Intel İşlemcilerinde yazmaç kullanımı konusunda bazı ayrıntılar vardır. Şimdi bu ayrıntılar üzerinde duralım:

- 64 bir long modda aşağıdaki 8 bitlik yazmaçlar aşağıdaki isimlerle kullanılabilmektedir:

AL, BL, CL, DL, SIL, DIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, R15B, AH, BH, CH, DH

Burada yeni eklenen R8-R1 yazmaçlarının düşük anlamlı byte'larının bağımsız olarak kullanılabilediğine dikkat ediniz. Ayrıca 64 bit modda SI, DI, BP ve SP yazmaçlarının düşük anlamlı byte'ları da bağımsız olarak kullanılabilmektedir. Ancak 64 bit modda AH, BH, CH ve DH yazmaçları kullanılırken bir byte'lık REX öneki gerekmektedir.

- 64 bit modda kullanılabilecek 16 bit yazmaçlar da şunlardır:

AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W

Burada yine R8-R15 yazmaçlarının düşük anlamlı 16 bitlerinin W soneki ile bağımsız olarak kullanılabildiğine dikkat ediniz.

- 64 bit modda kullanılabilecek 32 bit yazmazçlar şunlardır:

EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D,R12D, R13D, R14D, R15D

Burada R8-R15 yazmaçlarının düşük anlamlı 32 bitlerinin D soneki ile bağımsız olarak kullanılabildiğine dikkat ediniz.

- 64 bit modda kullanılabilecek 64 bit yazmaçlar da şunlardır:

RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

64 bit modda MOV komutunda ilginç bir farklılık vardır. Bu modda MOV komutu ile 32 bit yazmaçlara değer aktarıldığında bunların yüksek anlamlı 32 bitleri de komut tarafından sıfırlanmaktadır. Bu özellik maalesef Intel ve AMD dokümanlarında açıkça belirtilmemiştir. Örneğin:

```
mov     rax, -1
mov     ax, 1

RAX = 0000 0000 0000 0000 0000 0000 0000 0001
```

Ancak aynı durum 32 bit işlemcilerde böyle değildir. Yani bu işlemcilerde biz yazmaçların düşük anlamlı kısımlarına MOV komutuyla değer atadığımızda bundan onların yüksek anlamlı kısımları etkilenmez. Örneğin:

```
mov     eax, -1
mov     ax, 1

EAX = FFFF FFFF 0000 0001
```

11.3. X64 Mimarisinde Fonksiyon Mimarisinde Fonksiyon Çağırma Biçimleri

64 bit mimaride C derleyicileri genel olarak tek bir çağırma biçimi (calling convention) kullanmaktadır. Bu çağırma biçimi daha çok 32 bit mimaridedi "fastcall" çağırma biçimine benzemektedir. 64 bit mimaride çok fazla yazmaç olduğu için parametre aktarımının yazmaçlar yoluyla yapılması daha uygun görülmüştür. Ayrıca C derleyicilerinde genel olarak bir isim dekorasyonu uygulanmamaktadır. Yani isimler hiç değiştirilmeden doğrudan kullanılmaktadır. Windows sistemleriyle Linux sistemleri arasında parametre aktarımı ile ilgili bazı farklılıklar vardır. Linux ile Mac OS X arasında bir farklılık yoktur. Burada biz yine her iki sistemdeki özel durumları ayrı başlıklar halinde inceleyeceğiz.

11.3.1. 64 Bit Windows Sistemlerinde C Programalama Dili İçin Kullanılan Çağırma Biçimi

64 bit Windows sistemlerinde Microsoft'un C derleyicileri 32 bitteki çağırma biçimlerini sentaks olarak kabul etse de onları dikkate almamaktadır. (Yani örneğin biz bir fonksiyonun bildiriminde çağırma biçimi olarak __stdcall ya da __cdecl anahtar sözcüklerini kullansak bile bu durum hataya yol açmaz. Ancak bu anahtar sözcükler semantik olarak dikkate alınmamaktadır.

64 Bit Windows sistemlerinde Microsoft C derleyicileri tarafından uygulanan çağırma biçiminin ana hatları şöyledir:

- İlk 4 parametre eğer tamsayı türündense ya da gösterici türündense bunlar sırasıyla RCX, RDX, R8 ve R9 yazmaçlarıyla aktarılmaktadır. Diğer parametreler sağdan sola stack'e 64 bit olarak push edilmektedir. Dörtten fazla parametre söz konusu olduğundan stack çağıran fonksiyon tarafından (caller) dengelenmektedir. 64 bit Microsoft C derleyicilerinde int ve long türleri 4 byte, long long türü ise 8 byte'tır. 8 byte'tan daha kısa olan parametreler RCX, RDX, R8 ve R9 yazmaçlarının düşük anlamlı 8 bit, 16 bit ve 32 bitlik kısımları yoluyla aktarılır. Bu aktarım sırasında bu yazmaçların yüksek anlamlı byte'ları 0 olmaktadır.

- float ve double türünden parametreler XMM0, XMM1, XMM2 ve XMM3 yazmaçlarıyla aktaraılmaktadır. Ancak parametreler karışık türlerdense yazmaçlar pozisyona göre kullanılmaktadır. Yani:

1'inci Parametre: RCX/XMM0
2'inci Parametre: RDX/XMM1
3'üncü Parametre: R8/XMM2
4'üncü Parametre: R9/XMM3

Örneğin:

```
void Foo(int a, int b, int c, int d);
```

Burada aktarımda şu yazmaçlar kullanılacaktır:

```
a -> RCX
b -> RDX
c -> R8
d -> R9
```

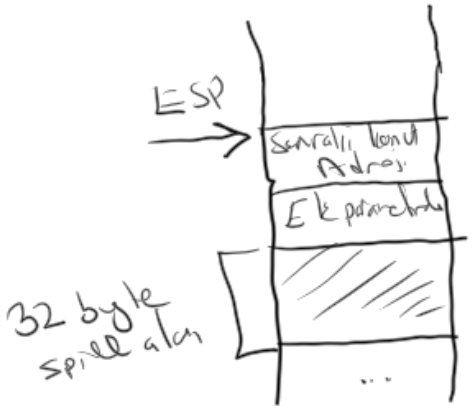
Örneğin:

```
void Foo(int a, double b, int c, double d);
```

a -> RCX
b -> XMM1
c -> R8
d -> XMM3

- Fonksiyonun geri dönüş değeri yine 8 byte ve daha küçük tamsayı türlerine ilişkinse RAX yazmacı ile gerçek sayı türlerine ilişkinse XMM0 yazmacı ile aktarılmaktadır. Daha büyük geri dönüş değerleri stack yoluyla aktarılır. Ancak çağıran fonksiyonun stack'te bu yeri ayırmış olması gerekir.

- Microsoft'un 64 bit çağırma biçiminde çağıran fonksiyon CALL işleminden önce stack'te 4 parametre yazmacı için toplam 32 byte'lık bir tampon alan ("register spill" alan) oluşturmalıdır. Fonksiyonun parametreleri daha az olsa bile yine bu 32 byte'lık alanın oluşturulması gerekmektedir.



64 bit Windows sistemleri için örnek fonksiyonlar şöyle olabilir:

```
; Util.asm

[BITS 64]

SECTION .text
    global Add32, Add64, AddDouble

Add32:
    mov     eax, ecx
    add     eax, edx
    ret

Add64:
    mov     rax, rcx
    add     rax, rdx
    ret
AddDouble:
    addsd   xmm0, xmm1
```

```
    ret
```

Aşağıdaki kod ile test edilebilir:

```
#include <stdio.h>

int Add32(int a, int b);
long long Add64(long long a, long long b);
double AddDouble(double a, double b);

int main(void)
{
    int result32;
    long long result64;
    double resultDouble;

    result32 = Add32(10, 20);
    printf("%d\n", result32);

    result64 = Add64(10000000000, 20000000000);
    printf("%lld\n", result64);

    resultDouble = AddDouble(6.2, 5.4);
    printf("%f\n", resultDouble);

    return 0;
}
```

11.3.2. 64 Bit Linux Sistemlerinde C'de Kullanılan Çağırma Biçimi

Aslında 64 bit çağırma biçimi yalnızca Linux sistemlerinde değil BSD, MAC OS X gibi Unix türevi sistemlerde de tamamen aynıdır. Tıpkı 32 bit gcc derleyicilerinde olduğu gibi 64 bit gcc derleyicilerinde de fonksiyonlar için bir isim dekorasyonu uygulanmamaktadır. Bu çağırma biçimi şöyledir:

- İlk 6 parametre 8 byte ya da daha küçük tamsayı türlerine ilişkinse sırasıyla RDI, RSI, RDX, RCX, R8 ve R9 yazmaçlarıyla aktarılır. İlk 6 gerçek sayı türü ise XMM0, XMM1, XMM2, XMM3, XMM4, XMM5 ve XMM6 yazmaçlarıyla aktarılmaktadır. Eğer fonksiyon daha fazla parametreye sahipse bunlar sağdan sola stack'e push edilmektedir. Stack'in temizlenmesi yine çağıran fonksiyon tarafından yapılmaktadır. Ancak bu aktarım Windows'taki gibi pozisyon temelli değildir.

1'inci Parametre: RDI/XMM0
2'inci Parametre: RSI/XMM1
3'üncü Parametre: RDX/XMM2
4'üncü Parametre: RCX/XMM3
5'inci Parametre: R8/XMM4
6'ıncı Parametre: R9/XMM5

örneğin:

```
void Foo(int a, int b, int c, int d);
```

Burada aktarımda şu yazmaçlar kullanılacaktır:

```
a -> RDI
b -> RSI
c -> RCX
d -> RDX
```

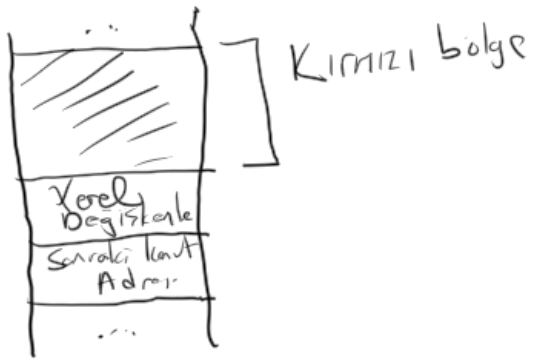
Örneğin:

```
void Foo(int a, double b, int c, double d);
```

a -> RDI
b -> XMM0
c -> RSI
d -> XMM1

- Geri dönüş değeri eğer 8 bte ya da daha küçük tamsayı türlerine ilişkinse RAX yazmacı yoluyla, gerçek sayı türleri ise XMM0 yazmacı yoluyla aktarılmaktadır.

- Çağrılan fonksiyon 128 byte'lık bir "kırmızı bölgeyi (red zone)" kullanma potansiyelindedir. Kırmızı bölge yerel değişkenler için ayrılan potansiyel alandır. Bu bölge çağıran fonksiyon tarafından kullanılmamalıdır.



Çağıran fonksiyon çağrılan fonksiyonunun yerel değişkenlerinin yukarısında (yani daha düşük adreste) 128 byte'lık alana hiç dokunmaması gerekir. Burası çağrılan fonksiyon için ayrılmıştır. Bir çeşit "spill" alanı olarak kullanılmaktadır. Görüldüğü gibi spill alanı bu sistemlerde yerel değişkenlerin ötesinde oluşturulabilmektedir. Özellikle çok thread'li uygulamalarda bu alanın başka bir thread tarafından bozulmaması öngörülmektedir.

64 bit Linux sistemleri için aşağıdaki kod örnekleri verilebilir:

```
; util.asm

[BITS 64]

SECTION .text
    global Add32, Add64, AddDouble

Add32:
    mov     eax, edi
    add     eax, esi
    ret

Add64:
    mov     rax, rdi
    add     rax, rsi
    ret
AddDouble:
    addsd   xmm0, xmm1
    ret
```

Test işlemi u kodla yapılabilir:

```
/* app.c */

#include <stdio.h>
```

```
int Add32(int a, int b);
long long Add64(long long a, long long b);
double AddDouble(double a, double b);

int main(void)
{
    int result32;
    long long result64;
    double resultDouble;

    result32 = Add32(10, 20);
    printf("%d\n", result32);

    result64 = Add64(10000000000, 20000000000);
    printf("%lld\n", result64);

    resultDouble = AddDouble(6.2, 5.4);
    printf("%f\n", resultDouble);

    return 0;
}
```

Derleme ve bağlama işlemleri şöyle yapılabilir:

```
nasm -f elf64 util.asm
gcc -c app.c
gcc -o app app.o util.o
```

NASM'nin Linux versiyonunda 64 bit için -f elf64 seçeneğinin gerektiğine dikkat ediniz.

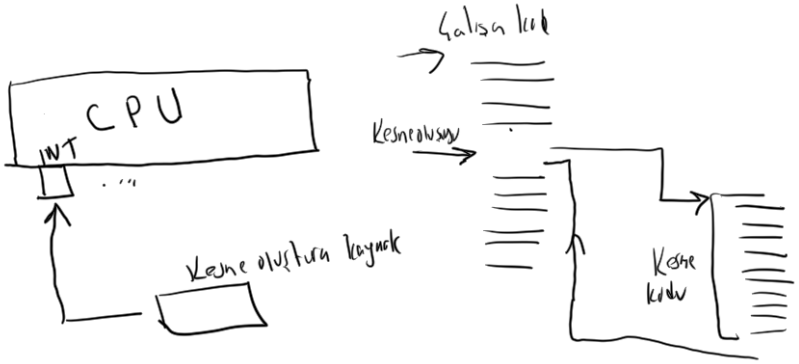
Intel İşlemcilerinde Kesmeler

İşlemcinin çalışmakta olan koda ara vererek başka bir kodu çalıştırması ve bu kodun çalışması bitince önceki kodun kaldığı yerden çalışmaya devam etmesi sürecine "kesme (interrupt)" denilmektedir. ("Interrupt" sözcüğü İngilizce "araya girme (özellikle konuşmalarda vs.) anlamına gelmektedir".)

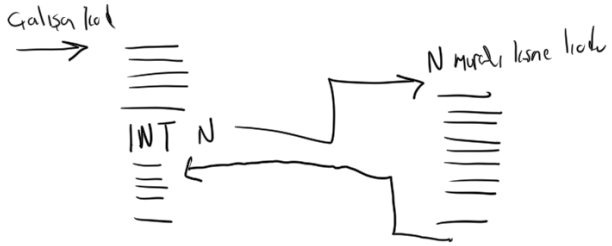
Kesme işlemciye özgü bir kavramdır. İşlemcilerin hemen hepsinde (mikrodenetleyiciler de dahil olmak üzere) çeşitli biçimlerde kesme mekanizması vardır. Kesmeler fonksiyon çağrılarına (yani CALL işlemlerine) benziyor olsa da aslında çeşitli bakımlardan bunlardan ayrılmaktadır.

Kesmeler oluş biçimine göre üçe ayrılmaktadır:

1) Donanım kesmeleri (Hardware Interrupts): Kesme dendiğinde default olarak donanım kesmeleri anlaşılır. Neredeyse hemen her işlemcide ve mikrodenetleyicide donanım kesmeleri vardır. Donanım kesmeleri işlemcinin özel bir pucu (bu uca genellikle INT pini denir) dışarıdan elektriksel olarak uyarılarak oluşturulmaktadır. (Yani donanım kesmeleri asenkron olarak dışarıdan başka bir donanım birimi tarafından işlemcinin kesme ucu uyarılarak oluştururlur):



2) Yazılım Kesmeleri (Software Interrupts): Yazılım kesmeleri programcı tarafından makine koduyla oluşturulan kesmelerdir. Örneğin Intel işlemcilerinde INT makine komutu yazılım kesmesi oluşturmak için kullanılmaktadır. Yazılım kesmelerinde kesmeyi oluşturan kaynak bizzat programcının kendisidir. Zaten yazılım kesmelerinin mekanizma olarak fonksiyon çağrılarından (CALL işleminden) çok büyük farkları yoktur. Pek çok işletim sisteminde sistem fonksiyonları CALL makine komutu yerine yazılım kesmeleriyle (örneğin Intel'de INT makine komutuyla) çağrılmaktadır.



3) İçsel Kesmeler (Internal Interrupts): Bu kesmeler bizzat işlemcinin kendisi tarafından oluşturulmaktadır. Intel terminolojisinde "exception" denilmektedir. Örneğin sayfalama mekznizması ve koruma mekanizması hep bu tür içsel kesmelerle yönetilmektedir. Yani içsel kesmeler işlemcinin bir makine komutunu çalıştırırken bazı uygunsuzluklar yüzünden kendisinin oluşturduğu kesmelerdir. Örneğin DIV makine komutu tamsayı bölmesi yapar. Aşağıdaki gibi bir kodun çalıştığını varsayalım:

```
XOR     EBX, EBX
DIV     EBX
```

Burada EAX / EBX işlemi yapılmaktadır. Ancak EBX'teki değer 0 olduğu için bölmenin bir anlamı yoktur. Artı Sonsuz değeri tamsayı olarak ifade edilebilen bir değer değildir. (Halbuki matematik işlemcide sıfıra bölme geçerli bir işlemdir). İşte bu duurmda işlemci "divide by zero" diye isimlendirilen bir kesme oluşturur. İşletim sistemleri de bu kesme oluştuğunda genellikle prosesi sonlandırırlar.

Kesmenin cinsi ne olursa olsun bir kesme oluştuğunda araya girilip çalıştırılan koda "kesme kodu (interrupt handler)" denilmektedir. Pek çok işlemcide kesmelerin numaraları vardır. Yani numaralar sayesinde çok sayıda kesme varmış gibi bir etki oluşturulabilmektedir. Örneğin Intel işlemcilerinde toplam 256 kesme numarası vardır. Zaten örneğin INT makine komutu da kesme numarasını argüman olarak alır. Örneğin:

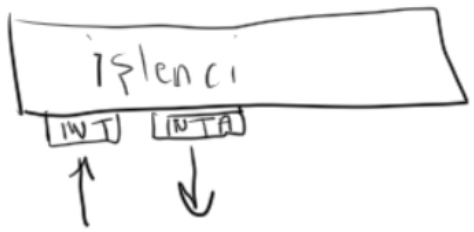
```
....
int         0x10            ; 10h numaralı kesme yazaılımsal olarak çağrılıyor
....
```

Geleneksel olarak kesme numaraları Intel'de 16'lık sistemde belirtilmektedir.

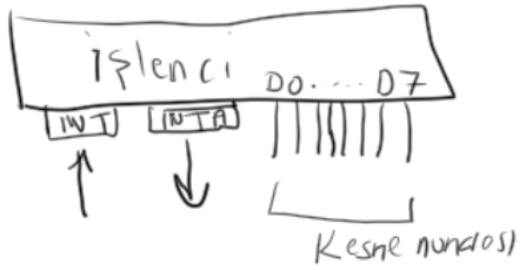
Donanım Kesmeleri (Hardware Interrupts)

Donanım kesmeleri yukarıda da belirtildiği gibi işlemcinin kesme ucunun (INT ucu) elektriksel olarak

uyarılmasıyla oluşturulur. Intel işlemcilerinin INT ucu elektriksel olarak aktive edildiğinde işlemci kesmeyi kabul etmeyebilir. Bu durum EFLAGS (64 bitte RFLAGS) yazmacının 9 numaralı biti olan IF bayrağıyla belirlenmektedir. Eğer IF bayrağı 1 ise işlemci donanım kesmelerini kabul eder, 0 ise kabul etmez. IF byrağını set etmek için STI, reset etmek için CLI makine komutları kullanılabilmektedir. İşlemci donanım kesmesini INT ucunu aktive eden donanım birimine bildirmek zorundadır. Bu da işlemcinin INTA (Ibterrupt Acknowledgement) ucu ile yapılır.



Intel mimarisinde kesme ister yazılımsal olsun, ister donanımsal olsun isterse içsel kesme olsun her kesmenin bir numarası vardır. Teorik olarak her numara farklı bir kesme kodunu belirtme potansiyeline sahiptir. Örneğin 13h kesmesi oluştuğunda dallanılacak kod ile 10h kesmesi oluştuğunda dallanılacak kod ayrı ayrı belirlenebilir. Donanım kesmelerinde kesme numarası kesmeyi oluşturan kaynak tarafından işlemcinin veri yolunun D0-D7 uçlarıyla elektriksel olarak iletilir.

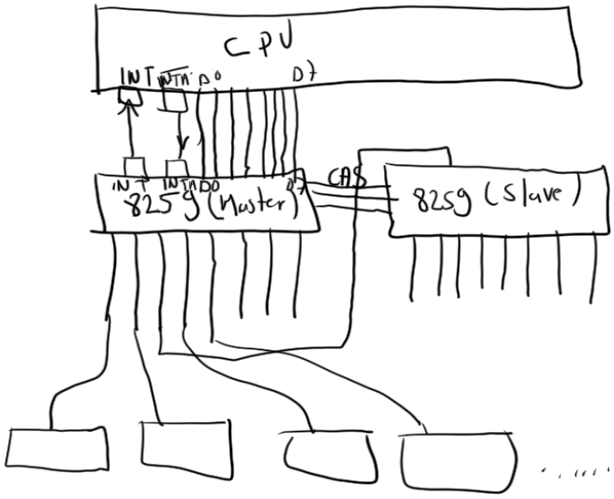


Bu durumda Intel işlemcilerindeki donanım kesmesine ilişkin protokol özet olarak şöyledir:

1) Donanım bitimi işlemcinin INT ucunu uyarır
2) İşlemci bunu kabul ederse kabul ettiğini INTA ucunu aktive ederek belirtir.
3) Kesmenin kabul edildiğini anlayan donanım birimi kesmenin numarasını elektriksel olarak D0-D7 uçlarından işlemciye bildirir.

Pekiyi işlemcinin INT ucunu tek bir donanım birimi mi aktive edebilmektedir? İşte birden fazla donanım birimi aynı uca bağlanamayacağı için birden fazla donanım biriminin kesme oluşturabilmesi için ismine "kesme denetleyicisi (interrupt controller)" denilen bir donanım birimi tasarlanmıştır. Bugün Intel tabanlı PC mimarisinde bu amaçla Intel'in "8259 Kesme Denetleyicisi" kullanılmaktadır. 1980 yılında piyasaya sürülen ilk PC'lerde bir tane kesme deteleyicisi kullanılıyordu AT modelleriyle birlike bu ikiye yükseltilmiştir. Ayrıca Intel 80486 ve Pentium modelleriyle birlikte ismine APIC (Advanced Programmable Interrupt Controller) denilen ayrı bir işlemcinin kendi çipi içerisinde bulunan bir kesme netleyicisini daha kullanmaktadır. Bu modern APIC denetleyicileri yerel (local) ve IO olmak üzere ikiye ayrılmaktadır. Bu konu ileride ele alınacaktır. Ancak klasik donanım kesmeleri hala APIC yoluyla değil 8259 kesme denetleyicisi yoluyla tetiklenir.

APIC sistemini bir yana bırakırsak bugün kullanılan PC'lerimizdeki temel doanım kesmeleri 8259 kesme denetleyicileri ile çoklanmaktadır.



Bu durumda kesme oluşturacak donanım birimleri aslında kesme denetleyicinin uçlarına bağlıdır. Bu birimler kesme denetleyicisini uyarırlar. Kesme denetleyici ise işlemcinin INT ucunu uyarır. İşlemci kesme kabul ederse bunu kesme denetleyicisine INTA ucu ile iletir. Kesme denetleyicisi de kesme numarasını D0-D7 uçlarını kullanarak işlemciye iletir. 70'lerin teknolojisi ile tasarlanan 8259 çok güçlü özelliklere sahip değildir. Ancak bugün hala geriye doğru uyumun korunması için kullanılmaktadır. İki 8259'un kaskat bağlanması için birincinin bir giriş ucunun diğerinin INT ucuna bağlanması gerekmektedir. Bu nedenle iki 8259'un kullanıldığı PC mimarisinde kesme denetleyicilerinde toplamda 16 değil 15 kesme ucu bulunmaktadır. PC mimarisinde birinci kesme denetleyicisinin (buna "master" denilmektedir) 2 numaralı ucu bu amaçla seçilmiştir. PC terminolojisinde kesme denetleyicilerinin giriş uçlarına IRQ (Interrupt Request) denilmektedir. Bu durumda bugün kullandığımız PC'lerde APIC'i yazmasak klasik 15 IRQ hattı vardır. Bu hatlara çeşitli yardımcı işlemciler bağlanmıştır. 8259 kesme denetleyicinin programlanabilen şu özellikleri vardır:

- IRQ giriş hatları uyarıldığında işlemciye gönderilecek kesme numarasının taban değeri belirlenebilmektedir. Örneğin BIOS bilgisayar açıldığında ilk 8259 (master) için taban değeri 8, ikinci 8259 (slave) için 70h olarak programlamaktadır. Bu durumda PC açıldığında birinci 8259'un 0 numaralı ucu uyarıldığında işlemciye 8 numaralı kesme, 1'inci ucu uyarıldığında 9 numaralı kesme ve sonraki uçlar için de artan numaralı kesmeler gönderilecektir. Windows, Linux ve MAC OS X gibi sistemler bu taban değerleri kendilerine göre değiştirmektedir.

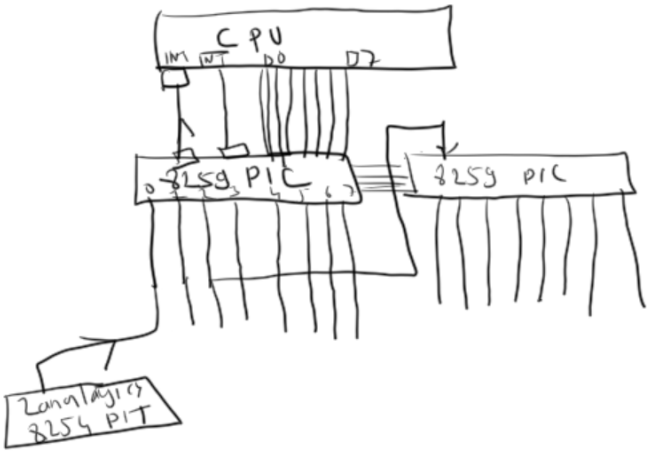
- Aynı anda birden fazla donanım birimi kesme denetleyicisinin farklı uçlarını uyarırsa kesme denetleyicisi bir önceliğe göre (default olarak düşü ucun önceliği vardır) bunları sıraya sokabilmektedir.

- Kesme denetliyicisi programlanarak onun belli uçları pasif hale (disable) getirilebilir. Örneğin bu sayede biz belli IRQ uçlarını pasif hale getirerek o uçtan kesme oluşmasını engelleyebiliriz.

Şimdi biz PC mimarisindeki IRQ hatlarının hangi birimlere ve ne nedenle bağlandıklarını açıklayalım.

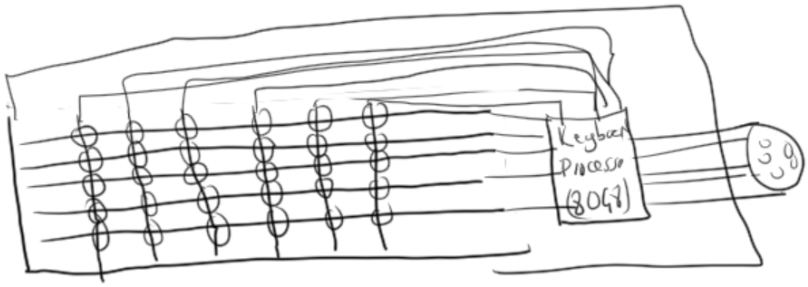
IRQ-0: Bu hatta Intel'in 8254 zamanlayıcı denetleyicisi (Programmable Interval Timer) bağlıdır. 8254 aslında darbe üreten basit bir işlemcidir. Belli bir periyoda ayarlanabilmektedir. Böylece her periyot dolduğunda 8254 kesme denetleyicisinin 0 numaralı ucunu uyarır. O da işlemcide kesme oluşturur. Bu kesmeye gelenksel olarak "Timer Kesmesi (Timer Interrupt)" denilmektedir. İşletim sistemleri uzunca bir süredir prosesler arası ve thread'ler arası geçiş için bu kesmeyi kullanmıştır. 8254 bilgisayar açıldığında BIOS tarafından 18.2 hertz frekansa (yani saniyede 18.2 kere) ayarlanmaktadır. Fakat Windows gibi Linux gibi işletim sistemleri bu frekansı 100 hertz ya da 1000 herts gibi bir değere çeker. 1000 herts periyor olarak 1 mili saniyeye karşılık gelmektedir. Bu durumda her bir milisanyide bir timer kesmesi oluşmaktadır. Çok çekirdekli sistemlerde timer kesmesi yerine artık "Yerel APIC (Local APIC)" içerisindeki zamanlayıcılardan faydalanılmaktadır. Ancak tek işlemcili sistemlerde klasik 8254 çalışması devam etmektedir. 8254 işlemcisinin nasıl programlandığı kurusumuz konu içerisinde değildir. Bugün bu zamanlayıcı artık çok

küçültülerek borad üzerinde çeşitli chiplerin içerisine yerleştirilmiş durumdadır.



IRQ-1: Bu hatta klavye denetleyicisi (keyboard controller) denilen işlemci bağlıdır. Klavye denetleyicisi Intel'in 8042 gibi işlemcileriyle yapılıyordu. Sonra arayüz aynı olacak biçimde başka devreler de kullanılmıştır. Klavye geleneksel olarak bir kabloyla bilgisayara bağlıdır. Bu kablonun bilgisayardaki ucunda klavye denetleyicisi bulunur. Bugün klasik PS2 klavye arayüzü artık kullanım dışı kalmak üzeredir. Klavyeler daha çok USB portları yoluyla bilgisayara bağlanmaktadır. Ancak temel prensip yine çok benzerdir.

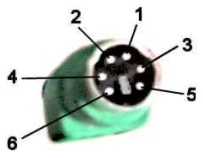
Klavye devresi aslında basit bir biçimde ele alırsak bir matris gibidir. Yatay ve düşey tellerin kesişim noktasında tuşlar vardır. Bir tuşa basıldığında yatay ve düşey teller kısa devre olur ve hangi tuşa basıldığı belirlenir. Sonra basılan tuş dış dünyaya kodlanır. Klavyenin içerisinde basılan tuşu tespit eden ve dış dünyaya kodlayan bir işlemci de vardır. Buna klavye işlemcisi (keyboard processor) denilmektedir. Bugün pek çok firmanın yaptığı aynı arayüze sahip farklı klavye işlemcileri vardır. Klavye üretici firmalar bunları kullanmaktadır.



PC zamanlarında klavye konnektörleri daha büyüktü. Sonra IBM tarafından tasarlanan ve PS/2 denilen küçük konnektörler kullanılmaya başlandı. Aşağıda PS/2 klavye konnektörünü görüyorsunuz:

PS/2 keyboard connector (MINI-DIN6)

Connector Pin #	Purpose
Pin 1	KBDAT (data)
Pin 2	not used
Pin 3	GND
Pin 4	VCC (+5V)
Pin 5	KBDCLK (clock)
Pin 6	not used

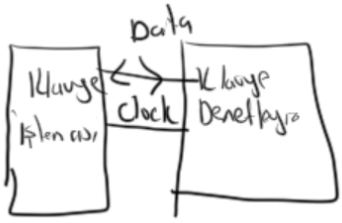


ıouse connector pinout is identical to PS/2 keyboard.

Konnektördeki yalnızca 4 ucun kullanıldığına dikkat ediniz. Aktarım seri bir biçimde Data/clock sistemiyle yapılmaktadır. Klavyedeki devreler PC'den aldıkları 5V ile beslenmektedir. Klavyede bir tuşa basıldığında tuşun numarası seri olarak bit bit dış dünyaya (yani klavye denetleiyicisine) iletilir. Bu tuşun numarasına

klavye tarama kodu (scan code) denilmektedir. Klavye işlemcisi yalnızca tuşa basıldığında değil, el tuştab çekildiğinde de dış dünyaya kod gönderir. Basılırken gönderilen tarama koduna klavye terminolojisinde "make code", çakilirken gönderilen koda da "break code" denilmektedir. Bugünkü PC klavyeleri "make code" olarak 7 bit bir sayı göndermektedir. Break code olarak önce önce bir F0 byte'ı sonra da make code'un aynısını göndermektedir. Bu iki kodu da klavye denetleyici almaktadır.

Klavyede parmak bir tuşa basılı olarak bekletildiğinde bir süre sonra periyodik bir biçimde o tuşa basılıyormuş etkisi yaratılmaktadır. Bu sürece "typematic" denir. İlk basımdan typematic'e kadar geçen süre ve typematic süresi klavye içerisindeki klavye işlemcisi tarafından programlanır. Klavye işlemcisi ile klavye denetleyicisi arasındaki bağlantı çift yönlüdür.



Klavye üzerindeki ışıklı tuşlar klavye devresi tarafından basılma sırasında yakılmaz. "Tuşun ışığı yak" komutu bilgisayar tarafındaki klavye denetleyici tarafından klavye işlemcisine gönderilir. Işık öyle yakılır.

Klavye üzerindeki ALT gibi, Num Lock gibi, Shift gibi tuşların diğer tuşlardan hiçbir farkı yoktur. Bunlar da benzer biçimde make code ve break code göndermektedir. Klavyenin üzerindeki tuşların sembolleri dilden dile değişebiliyorsa da tuşların tarama kodları değişmez. Bu kodları alan işletim sistemi hangi tuşa basıldığına tarama koduna ve o andaki seçilen klavye cinsine bakarak karar verir.

Klavyeden bir tuşa basıldığında ya da el tuştan çekildiğinde bilgisayar içerisindeki klavye denetleyicisici kesme denetleyicisini uyararak 1 numaralı IRQ'nun oluşmasına yol açar. Yani klavyeden her tuşa bastığımızda ve elimizi çektiğimizde bir kesme oluşmaktadır. İşletim sistemleri bu kesme oluştuğunda bnasılan ya da çekilne kodu klavye denetleyicisinden alarak bir kuruk sistemine ekler. Klavyeden tuş bekleyen prosesler de aslında basılan tuşu bu kuyruk sisteminden almaktadır.

IRQ-2: Bu IRQ ucu kaskat bağlantı için kullanıldığından dışsal bir birime bağlı değildir.

IRQ-3-IRQ4: IRQ-3 ucu COM2 biçiminde isimlendirilen ikinci seri port işlemcisine (8250/16550 UART) IRQ-4 ucu da COM1 biçiminde isimlendirilen seri port işlemcisine bağlıdır. Böylece seri porta bir bilgi geldiğinde kesme oluşmakta ve kesme kodu da gelen bilgiyi alıp tamponlayabilmektedir. Bugünkü PC'lerde artık seri port bulunmamaktadır. Ancak endüstriyel pek çok aygıt hala seri port kullanmaktadır. Seri port klasik olarak veri haberleşmesi için, fare ve modem için kullanılıyordu. Ancak seri porttan çalışan yazıcılar da vardı. Bugün madem ki seri portlar artık bilgisayarlarımızda yok seri port gerekn yerlerde USB'yi seri porta dönüştüren konnektörlerden faydalanılmaktadır:

IRQ-5: Bu uç LPT2 biçiminde isimlendirilen ikinci paralel port işlemcisine bağlıdır. Parale port da kesme gönderebilmektedir. Ancak bugün artık paralel portlar da yerine tamamen USB portlarına bırakmış durumdadır.

IRQ-6: Bu uç eskiden 8272 floppy denetleyicine bağlıydı. Artık floppy kullanımı kalkmıştır. Floppy okuma ya da yazma işlemini bitirdiğinde kesme oluşturabiliyordu.

IRQ-7: Bu uç LPT1 biçiminde isimlendirilen birinci paralel port işlemcisine bağlıydı.

IRQ-8: Bu uç ikinci kesme denetleyicisinin (slave PIC) ilk ucudur. Gerçek zaman saatine (RTC) bağlıdır. RTC beli bir zaman geldiğinde kesme oluşturabiliyordu. Aslında RTC'nin kesmesi neredeyse hiç kullanılmamıştır.

IRQ-9: Bu uç eski PC'lerde reserve edilmişti. Ancak yeni PC'lerde ACPI denetleyicine bağlıdır.

IRQ-10: Bu uç serbest bırakılmıştır. Genellikle SCSI sürücüleri bu ucu kullanmaktadır.

IRQ-11: Bu uç serbest bırakılmıştır. Çeşitli kartlar kesmeye gereksinim duydğunda bu hattı kullanabilmektedir.

IRQ-12: Bu uç da serbest bırakılmıştır. Tipik olarak PS2 fareler bu ucu kullanmaktadır.

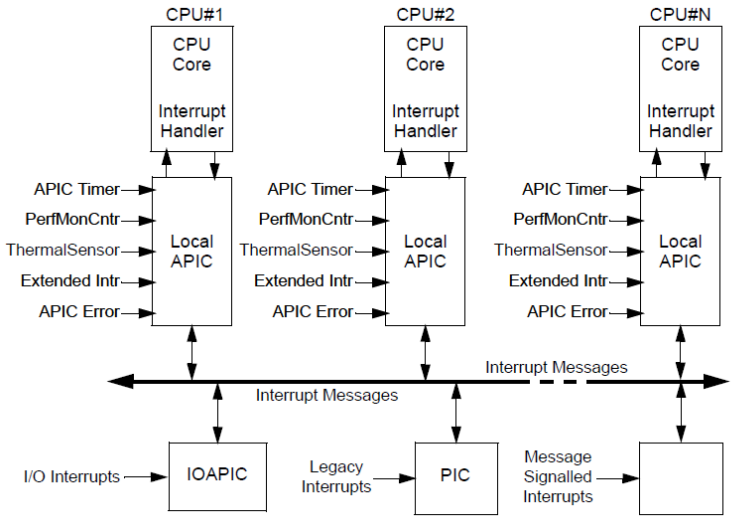
IRQ-13: Bu uç matematik işlemci tarafından işlemlerde hata oluştuğunda ana işlemciyi bilgilendirmek için kullanılmaktadır.

IRQ-14: Bu uç hard disk ve CD'ler için 1 numaralı IDE ve ATA denetleyicilerine bağlıdır. Hard diskten bir okuma ya da yazma yapılması istendiğinde disk devresi bunu yapar, aktarımı gerçekleştrdikten sonra işlemciyi kesme yoluyla haberdar eder. Böylece işletim sistemleri disk işleminin bittiğini anlayarak prosesi yeniden wait kuyruklarından çizelgeye dahile derler.

IRQ-15: Bu uç hard disk ve CD'ler için 2 numaralı IDE ve ATA denetleyicilerine bağlıdır.

APIC ve IOAPIC

Intel'in 8259 Kesme Denetleyicisi uzunca bir süe kullanılmıştır ve hale bazı ana kartlarda kullanılmaya devam etmektedir. 8259 tek bir işlemci ya da çekirdekle birlikte kullanılabiliyordu. Intel Pentium 4 modelleriyle birlikte işlemcinin içerisine entegre edilmiş olan yeni bir kesme denetleyicisi tasarladı. Buna APIC (Advanced Programmable Interrupt Conroller) denilmektedir. Daha sonraları çok çekirdekli sistemler için CPU'nun dışında bir IO APIC devresi de tasarlandı. Bugün CPU içerisindeki APIC devresine daha çok "yerel APIC (local APIC)" CPU'nun dışındaki APIC devresine "IO APIC" denilmektedir.



Yerel APIC yukarıda da belirtildiği gibi her çekirdeğin içerisinde o çekirdeğe dahil edilmiş biçimde

bulunmaktadır. Yerel APIC için kesme oluşturan kaynaklar şunlardır:

- IO APIC (İleride ele alınıyor)
- İlgili çekirdeğin INT ucu
- Diğer bir çekirdek (bir çekirdek diğerine kesme gönderebilmektedir)
- Çekirdeğin zamanlayıcısı (APIC timer) ya da performans sayacı (performance counter)

Görüldüğü gibi her çekrideğin ayrı bir yerel kesme denetleyicisi vardır. Bu kesme denetleyicileri ortak bir yola (bus) bağlanarak dışarıdaki başka bir kesme denetleyicisi ile uyarılabilmektedir. İşte dışarıda bulunan bu global kesme denetleyicisine IO APIC denilmektedir. IO APIC yerel APIC'lere bağlı gibi düşünülebilir. IO programlanarak belli bir çekirdekteki yerel APIC'i uyarma yoluyla onda kesme oluşturabilmektedir. Bugün IO APIC olarak Intel'in 82093AA işlemcisi kullanılmaktadır.

Bugünkü klasik SMP ana kartlarda mimari kesmelere ilişkin donanım mimarisi şu biçimdedir: Çekirdeklerin yerel APIC'leri ortak bir yola (bus) bağlanmıştır. Bu yol da dışarıdaki IO APIC denilen kesme denetleyicisinin kontrolündedir. Böylece IO APIC istenilen bir çekirdekte kesme oluşturabilmektedir. IO APIC'in 24 kesme ucu vardır. Yani bu bakımdan çift 8259'dan daha geniş olanaklara sahiptir. Aynı zamanda IO APIC kesme öncelikleri konusunda daha esnek programlanabilmektedir. Yeni ana kartlarda arık genellikle 8259 bulundurulmamakta ve doğrudan geleneksel IRQ kaynakları IO APIC'e bağlanmaktadır. IO APIC'in programlanmasıyla klasik IRQ numaraları yine aynı biçimde set edilmektedir. Yani bugün biz hala temel IRQ'ları ana kartımızda 8259 varmış gibi düşünebiliriz.

Her yerel APIC'in ayrı bir zamanlayızı devresi kesme üretebilmektedir. Böylece IR0 ile oluşturulan periyodik kesmeler her çekirdek için ayrı ayrı çok daha yüksek frekansta oluşturulabilmektedir. Bazı işlem sistemleri artık thread çizelgelemesinde threadlerarası geçiş için bu çekirdeklerin yerel APIC'lerimdeki zamanlayıcılardan faydalanmaktadır.

Pekiyi bugün örneğin 8 çekirdekli bir ana kartımızda klasik IRQ'lar oluştuğunda IO APIC bunları hangi çekirdeğe iletmektedir? İşte işletim sistemi IO APIC'i programlayarak bu IRQ'ları herhangi bir çekirdeğe yönlendirebilmektedir. Default olarak pek çok sistem bunu 0'ıncı (yani ilk) çekirdeğe yönlendirmektedir. Gerçekten de örneğin Linux'un pek çok modeli klasik IRQ'ları IOAPIC yoluyla 0'ıncı çekirdeğe yönlendirir. Tabii aslında IRQ'ların hangi çekirdek tarafından işlendiğinin çoğu kez bir önemi yoktur. Çünkü IRQ'lar sistem düzeyinde işlemler yapar. Yani yaptığı işlemler global düzeyde tüm sisteme yöneliktir. Bu işlerin hangi çekirdek tarafından yapıldığının bir önemi yoktur. Fakat yine de IRQ'ların hepsinin aynı çekirdeğe atanması o çekirdeğin yükünü göreli olarak artırmaktadır. Her ne kadar bu yük önemli düzeyde değilse de bu yükün paylaştırılması da istenebilir. Bu sürece "irq balancing" denilmektedir. Bir zaman önceye kadar Linux çekirdekleri "irq balancing" içermiyordu. Şimdi artık yeni Linuz çekirdekleri hangi CPu çekirdeği boşsa IRQ'yu ona yönlendirmektedir. Bir IRQ'nun IO APIC yoluyla belli bir çekirdekte çalıştırılmasına "IRQ affinity" denir. Aşağıda 3 çekirdekli bir sistemde /proc/interrupt dosyasından elde edilen çıktı görülmektedir:

```
           CPU0       CPU1       CPU2
  0:         35          0          0   IO-APIC-edge      timer
  1:          1          0        154   IO-APIC-edge      i8042
  8:          0          0          0   IO-APIC-edge      rtc0
  9:          3          0          1   IO-APIC-fasteoi   acpi
 12:          0          0        228   IO-APIC-edge      i8042
 14:          0          0          0   IO-APIC-edge      ata_piix
 15:          0          0        311   IO-APIC-edge      ata_piix
 19:        103         11        106   IO-APIC  19-fasteoi   eth0
 20:          0          0       7773   IO-APIC  20-fasteoi   vboxguest
 21:          0          0      11242   IO-APIC  21-fasteoi   0000:00:0d.0, snd_intel8x0
 22:          0          0         29   IO-APIC  22-fasteoi   ohci_hcd:usb1
NMI:          0          0          0   Non-maskable interrupts
LOC:      10396       9872       9293   Local timer interrupts
SPU:          0          0          0   Spurious interrupts
PMI:          0          0          0   Performance monitoring interrupts
IWI:          0          0          0   IRQ work interrupts
RTR:          0          0          0   APIC ICR read retries
RES:       8372       9367       6006   Rescheduling interrupts
CAL:       3806       4803        489   Function call interrupts
TLB:        520        305        353   TLB shootdowns
TRM:          0          0          0   Thermal event interrupts
THR:          0          0          0   Threshold APIC interrupts
MCE:          0          0          0   Machine check exceptions
MCP:          1          1          1   Machine check polls
HYP:          0          0          0   Hypervisor callback interrupts
ERR:          0
MIS:          0
```

İlk sütun IRQ numarasını diğer sütunlar ise toplam kaç kere bu IRQ'nun oluştuğunu göstermektedir. Son sütunda IRQ'nun hangi amaca hizmet ettiği raporlanmaktadır. Burada görüldüğü gibi bu Linux çekirdeği thread'lerarası geçiş için yerel APIC'lerin zamanlayıcılarını kullanmıştır (bunları 10 ms. kurduğu görülmüştür.)

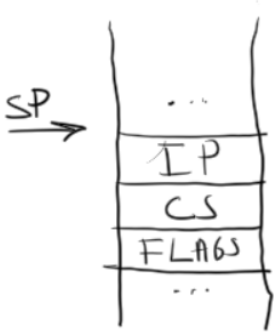
Kesme Sırasında Intel İşlemcilerinin Davranışı

Türü ne olursa olsun (yazılım kesmesi, donanım kesmesi ya da içsel kesme) bir kesme oluştuğunda bazı ortak işlemler işlemci tarafından yapılmaktadır. Intel işlemcileri kesme oluştuğunda sırasıyla EFLAGS/RFLAGS, CS, ve sonraki komuta ilişkin EIP/RIP yazmaçlarını stack'e atar. Geri dönüş adresinin yanı sıra bayrak yazmaçlarının da kesme sırasında stack'e atılması mutlak gereklidir. Bu nedenle bu işlem programcının sorumluluğundan alınıp makine kodunun kendisine verilmiştir. Fakat diğer yazmaçların çıkışta orijinal değerlerini koruması için dallanılan kesme kodunun bu yazmaçları saklayıp geri yüklemesi gerekir. Gerçekten de kesme kodları (inerrupt handler) hemen işin başında tüm yazmaçları stack'e atıp çıkarken de onları geri alırlar. Intel bu işlemleri kolaylaştırmak için PUSHA/PUSHAD ve POPA/POPAD makine komutlarını bulundurmuştur.

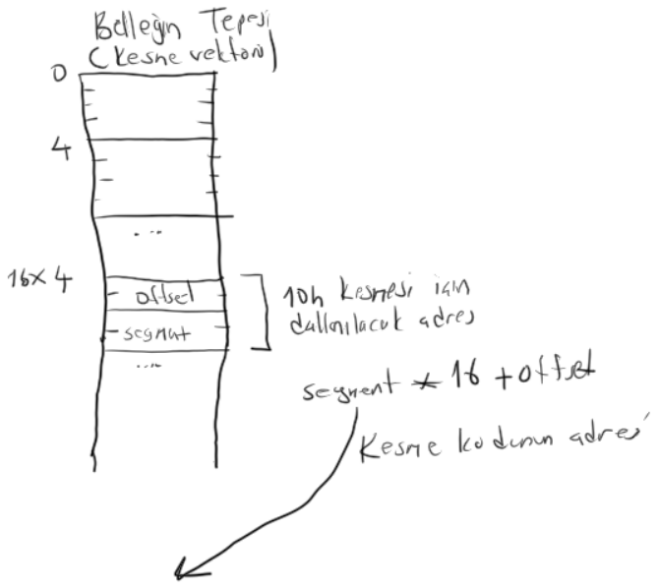
Pekiyi kesme oluştuğunda dallanılacak kod (interrupt handler) nasıl belirlenmektedir? İşte genel olarak işlemcilerde bir kesme oluştuğunda dallanılacak kodun adresinin belirtildiği veri yapısına "kesme vektörü (interrupt vector)" denilmektedir. Kesme vektörünün yapısı gerçek mod / V86 moduyla korumalı modda farklıdır. Gerçek mod ve V86 modunda belleğin tepesindeki İLK 1024 byte kesme vektörü olarak kullanılır. Burada her kesme için 2 byte offset ve 2 byte segment değerleri vardır. Segment 16 ile çarpılıp offset ile toplanarak elde edilen adrese dallanılır. Örneğin gerçek modda yazılım kesmesi oluşturan aşağıdaki gibi bir makine kodu verilmiş olsun:

```
INT 10h
```

Burada işlemci önce FLAGS yazmacını sonra sırasıyla CS ve IP yazmaçlarını stack'e atar:



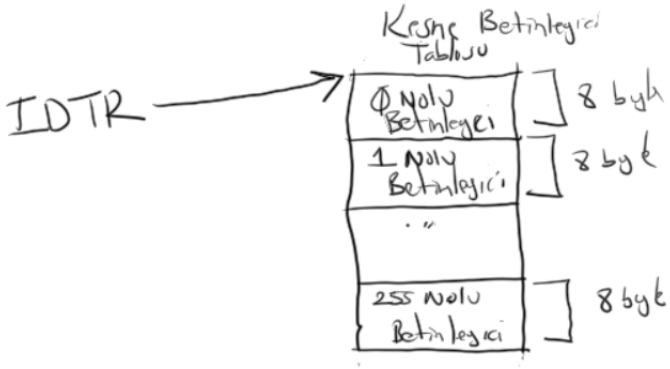
Sonra işlemci 10h (16) değerini 4 ile çarparak oradan sırasıyla dallanılacak adresin offset ve segment değerlerini elde eder:



Nihayet segment 16 ile çarpılıp offset ile toplanarak nihai adres elde edilir. Intel işlemcilerinin Gerçek od ve V86 modundaki 16 bit çalışması ayrı bir konu olarak temel hatlarıyla ele alınacaktır.

Gerçek modda ve V86 modunda kesme kodundan (interrupt handler) IRET makine komutuyla geri dönülür. IRET önce bayrak yazmacını sonra da CSD ve IP yazmaçlarını stack'ten alarak geri dönüşü sağlar.

32 bit ve 64 bit korumalı modda kesme vektörü tamamen değiştirilmiştir. Bu modda kesme vektörü yerine "Kesme Betimleyici Tablosu (Interrupt Descriptor Table)" denilen bir tablo kullanılır. Kesme betimleyici tablosu toplam 256 kesme betimleyicine (interrupt descriptor) sahiptir. Kesme betimleyicileri korumalı mod konusunda belirtildiği gibi 8 byte uzunluğunda içerisinde aban adresin ve offset adresinin bulunduğu betimleyicilerden oluşmaktadır.



Pekiyi "Kesme Betimleyici Tablosu" nerede bulunmalıdır? İşte gerçek modda olduğu gibi bu tablonun belleğin tepesinde bulundurulması zorunlu değildir. İşlemci Kesme Betimleyici Tablosunu IDTR isimli yazmacın gösterdiği yerde arar. Bu durumda işletim sistemini yazanlar önce Kesme Betimleyici Tablosunu RAM'de herhangi bir yerde oluştururlar sonra IDTR yazmacını tablonun başlangıç adresini gösterecek biçimde ayarlarlar.

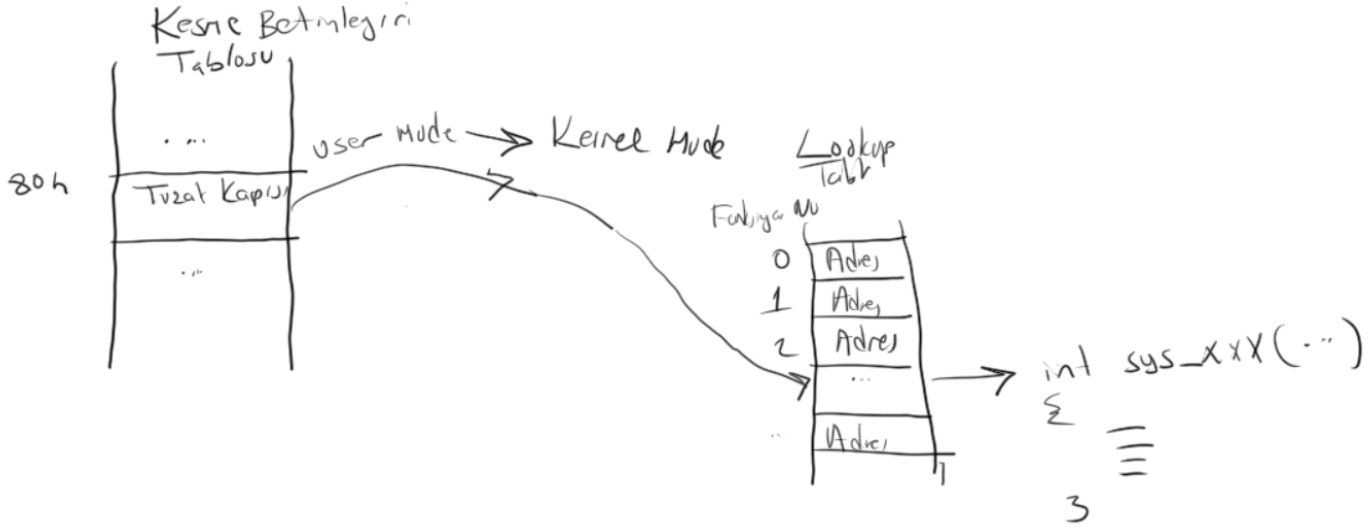
Kesme betimleyici tablosunda 3 tür betimleyici bulunabilmektedir:

1) Kesme Kapısı Betimleyicisi (Interrupt Gate Descriptor)
2) Tuzak Kapısı Betimleyicisi (Trap Gate Descriptor)
3) Görev Kapısı Betimleyicisi (Task Gate Descriptor)

Bu kapı betimleyicilerinin hepsi kesme oluştuğunda akışın istenilen adrese dallanması için kullanılmaktadır. Tabii bu betimleyicilerin kapı (gate) betimleyicileri olması otomatik öncelik yükseltiminin de yapılabileceği

anlamına gelir. Başka bir deyişle bir kesme oluştuğunda kodun önceliği yükseltilerek (kernel moda geçilerek) ilgili adresteki kod çalıştırılabilmektedir. Gerçekten de korumalı modda çalışan işletim sistemlerinde bir kesme oluştuğunda kesme kodu nun çekirdek modunda çalıştırılması için bu kapılar yoluyla öncelik yükseltmesi yapılmaktadır. "Kesme Kapısı Betimleyicisi" ile "Tuzak Kapısı Betimleyicisi" bir fark dışında birbirinin aynısıdır. O fark da şudur: Kesme Kapısı Betimleyicisi yoluyla kesme koduna dallanılırken işlemcinin bayrak yazmacındaki IF bayrağı reset edilir. Böylece kesme kodu çalıştığı sürece işlemci dışsal kesmelere kapatılmış olur. Tabii kesme kodu isterse STI makine komutuyla işlemciyi dışsal kesmelere yine açabilir. Oysa "Tuzak Kapısı Betimleyicisi" ile kesme koduna dallanılırken IF bayrağı otomatik olarak reset edilmemektedir. Görev Kapısı Betimleyicisi pek kullanılmamaktadır. Bu betimleyici kesme oluştuğunda "donanımsal proseslerarası geçiş (hardware task switching)" amacıyla kullanılmaktadır. Yani kesme oluştuğunda proseslerarası geçiş yapılabilir. Ancak daha önceden de belirtildiği gibi bunun zaman maaliyeti yüksek olduğu için donanımsal olarak proseslerarası geçiş işletim sistemi geliştiricileri tarafından tercih edilmemektedir.

Bugün kullandığımız Linux, BSD ve Mac OS X sistemlerinde sistem fonksiyonları 80H yazılım kesmesi yoluyla çağrılmaktadır. Linux'ta sistem fonksiyonlarının numarası ve parametreleri yazmaçlar yoluyla aktarılırken BSD ve Mac OS X sistemlerinde stack yoluyla (stack'e sağdan sola push edilerek) aktarılmaktadır. Bu sistemlerde Kesme Betimleyici Tablosunun 80h girişine bir tuzak kapısı (trap gate) yerleştirilmiştir. Bu tuzak kapısı kodun önceliğini sıfıra yükselterek çekirdek içerisinde bir yere dallanmayı sağlar. Orada da sistem fonksiyon numarasına bakılarak uygun sistem fonksiyonuna call edilmektedir.



NT grubu Windows sistemlerinde kernel moda yine benzer biçimde 2E numaralı kesme yoluyla geçilmektedir. Windows'ta ismine "API Fonksiyonları" denilen kütüphane fonksiyonları vardır. Bu fonksiyonların bazıları "Doğal API (Native API) Fonksiyonları" denilen daha düşük seviyeli, API fonksiyonlarını çağırmaktadır. İşte kernel moda geçiş bu doğal API fonksiyonlarının içeriissnde uygulanan 2E yazılım kesmesi yoluyla yapılmaktadır.

Pekiyi neden Linux, BSD ve Mac OS X sistem fonksiyonlarının kesme yoluyla çağrılmasını öngörmüşlerdir? Bunun en önemli nedeni çağrısının hiçbir kütüphaneye gereksinim duyulmadan tek hamlede yapılabilmesidir. Yani biz nerede olursak olalım o noktada hiçbir kütüphaneye gereksinim duymaan 80h kesmesi ile o noktada sistem fonksiyonlarını çağırabililiriz. Halbuki Windows'ta API fonksiyonlarının büyük bölümü "System32.DLL", "Kernel32.DLL" ve "User32.DLL" isimli dosyalardadır. Bunların çağırılması için ise bu DLL'lere link aşamasında referans edilmesi gerekmektedir. Tabii Windows'ta da 2E kesmesi doğrudan çağrılabilir. Ancak bu kesmeler bir işin belirli kısımlarını yapan oldukça aşağı seviyeli kodları çalıştırmaktadır. Bu nedenle Linux, BSD ve Mac OS X sistemlerinde olduğu gibi bu kesme programcılar tarafından doğrudan kullanılmaz.

Intel İşlemcilerindeki İçsel Kesmeler

32 bit ve 64 bit Intel işlemcilerinde korumalı modda ilk 32 kesme numarası işlemcinin çeşitli makine komutları işletilirken kendisinin oluşturduğu kesmelerdir. Bu içsel kesmeler kendi aralarında üç gruba ayrılmaktadır:

1. **Fault:** Fault durumunda akış kesmeye yol açan makine komutunun kendisiyle devam eder. Aynı zamanda bir hata kodu oluşan fault hakkında bilgi vermek için üretilmektedir.

2. Trap: Burada akış trap'a yol açan makine komutundan sonraki komutla devam eder.

3. Abort: Abort oluştuğunda akışın devamı mümkün olmayabilir. Yani o anda çalışmakta olan program kodu artık güvenli olarak devam ettirlmez.

Aşağıda Intel işlemcilerindeki içsek kesmelerin listesi görülmektedir:

Table 8-1. Interrupt Vector Source and Cause

Vector	Exception/Interrupt	Mnemonic	Cause
0	Divide-by-Zero-Error	#DE	DIV, IDIV, AAM instructions
1	Debug	#DB	Instruction accesses and data accesses
2	Non-Maskable-Interrupt	#NMI	External NMI signal
3	Breakpoint	#BP	INT3 instruction
4	Overflow	#OF	INTO instruction
5	Bound-Range	#BR	BOUND instruction
6	Invalid-Opcode	#UD	Invalid instructions
7	Device-Not-Available	#NM	x87 instructions
8	Double-Fault	#DF	Exception during the handling of another exception or interrupt
9	Coprocessor-Segment-Overrun	—	Unsupported (Reserved)
10	Invalid-TSS	#TS	Task-state segment access and task switch
11	Segment-Not-Present	#NP	Segment register loads
12	Stack	#SS	SS register loads and stack references
13	General-Protection	#GP	Memory accesses and protection checks
14	Page-Fault	#PF	Memory accesses when paging enabled
15	Reserved	—	
16	x87 Floating-Point Exception-Pending	#MF	x87 floating-point instructions
17	Alignment-Check	#AC	Misaligned memory accesses
18	Machine-Check	#MC	Model specific
19	SIMD Floating-Point	#XF	SSE floating-point instructions
20–29	Reserved	—	
30	Security Exception	#SX	Security-sensitive event in host
31	Reserved	—	
0–255	External Interrupts (Maskable)	#INTR	External interrupts
0–255	Software Interrupts	—	INT*n* instruction

Table 8-2. Interrupt Vector Classification

Vector	Interrupt (Exception)	Type	Precise	Class[2]
0	Divide-by-Zero-Error	Fault	yes	Contributory
1	Debug	Fault or Trap		Benign
2	Non-Maskable-Interrupt	—	—	
3	Breakpoint	Trap	yes	
4	Overflow			
5	Bound-Range	Fault		
6	Invalid-Opcode			
7	Device-Not-Available			
8	Double-Fault	Abort	no	
9	Coprocessor-Segment-Overrun			
10	Invalid-TSS	Fault	yes	Contributory
11	Segment-Not-Present			
12	Stack			
13	General-Protection			
14	Page-Fault			Benign or Contributory
15	Reserved			
16	x87 Floating-Point Exception-Pending	Fault	no	Benign
17	Alignment-Check		yes	
18	Machine-Check	Abort	no	
19	SIMD Floating-Point	Fault	yes	
20–29	Reserved			
30	Security Exception	–	yes	Contributory
31	Reserved			
0–255	External Interrupts (Maskable)	—[1]	—[1]	Benign
0–255	Software Interrupts			

Note:
1. *External interrupts are not classified by type or whether or not they are precise.*
2. *See "#DF—Double-Fault Exception (Vector 8)" on page 220 for a definition of benign and contributory classes.*

Şimdi bazı önemli içsel kesmeler hakkında kısa bazı bilgiler vermek istiyoruz:

0 Numaralı İçsel Kesme: Bu kesme DIV ve IDIV makine komutları tarafından sıfıra bölme durumunda oluşturulmaktadır. Genellikle işletim sistemleri abort tarzındaki bu kesme oluştuğunda prosesi sonlandırmaktadır.

1 Numaralı İçsel Kesme: İşlemcinin EFLAGS (FLAGS ya da RFLAGS de dahil olmak üzere) yazmacındaki TF (Trap Flag) bayrağı "single step" kesmesi de denilen 1 numaralı kesmenin oluşmasına neden olur. TF bayrağı 1 ise işlemci her komutu çalıştırdığında 1 numaralı içsel kesmeyi oluşturur. Böylece bu kesmeye ilişkin kesme kodu kontrolü ele alır ve işlemcinin yazmaçlarını vs. görüntüleyebilir. Gerçekten de makine kodu düzeyindeki debugger'lar (machine levele debuggers) bu içsel kesme sayesinde işlemlerini yapmaktadır.

2 Numaralı İçsel Kesme: Bu kesme işlemcinin NMI ucu aktive edildiğinde oluşturulmaktadır. İşlemcinin INT ucunun yanı sıra bir NMI (None Maskable Interrupt) ucu vardır. Bu uç aktive edildiğinde 2 numaralı kesme oluşur. İsmi üzerinde bu kesme IF bayrağı resetlenerek (örneğin CLI komutuyla) kapatılamaz. PC mimarisinde bu uca güç kaynağından gelen bildirim ve RAM'den gelen parity bildirimleri bağlanmıştır. Güç kaynağı kritik bir seviyenin altına düştüğünde ya da RAM'e erişimde parity hatası oluştuğunda oluşturulmaktadır.

3 Numaralı İçsel Kesme (Breakpoint Kesmesi): Bu kesme INT3 isimli makine komutu tarafından oluşturulur. Debugger'lar için düşünülmüştür. INT makine komutu iki byte yer kapladıığı halde INT3 bir byte yer kaplamaktadır. Böylece debugger'lar fazla yer kaplamadan "breakpoint" amacıyla bu nakine komutunu kullanabilmektedir. Şöyle ki: Derleyicinin ürettiği kodlarda kaynak kod temelindeki debug işlemlerinde satır sonlarına bir işlevi olmayan bir byte uzunluğunda NOP (No Operation) komutlarını yerleştirir. Bu NOP komutları normal çalışmada hızı nano ölçüde azaltsa da bir soruna yol açmaz. Ancak

debug işlemi sırasında debugger vreakpoint işleminde bu NOP komutu yerien INT3 komutunu yerleştirir. Aslında bu debugger'lar break point koymak için NOP gibi boş bir komuta da gereksinim duymayabilirler. (Örneğin doğrudan çalıştırılabilir bir program debug edilirken makine kodu temelinde break point uygulanabilir). Debugger bu durumda breakpoint konulan yerin başındaki makine komutunun ilk byte'ını saklayarak onu INT3 komutuyla değiştirir. Böylece akış o noktaya geldiğinde 3 numaralı kesme oluşur ve debugger kodu devreye girer. İşte debugger bu noktada sakladığı byte'ı yeniden INT3 komutunu ezerek oraya yerleştirebilir. Böylece orijinal kod yine çalışabilir. Pek çok debugger INT3 komutunu bu teknikle yerleştirmektedir.

4 Numaralı İçsel Kesme: Bu kesme bazı aritmetik işlem yapan makine komutlarında taşma olduğu zaman tetiklenir. Örneğin ADD makine komutuyla iki işaretli sayıyı toplayacak olalım. Eğer sonuçta bir taşma gerçekleşirse bu kesmenin oluşturulması sağlanabilir. Bu kesmeyi kesmeyi oluşturan aslında INTO isimli bir byte'lık bir makine komutudur. INTO komutu OF = 1 ise dört numaralı kesmeyi oluşturmaktadır. Örneğin:

```
ADD EAX, [EBX]
INTO
```

Bazı dillerde taşma durumu exception oluşmasına ve programın çökmesine yol açabilmektedir.

6 Numaralı İçsel Kesme: İkilik sistemdeki her dizilim bir makine komutuna karşılık gelmemektedir. İşte işlemci bir komuta karşılık gelemeyen bir dizilimle karşılaştığında bu kesmeyi oluşturur.

11 Numaralı İçsel Kesme: Segment tabanlı sanal bellek kullanımında doğrusal adres fiziksel adrese dönüştürülürken betimleyicinin içerisindeki P biti 0 ise segment fiziksel bellekte değildir ve bu kesme oluşturulur.

12 Numaralı İçsel Kesme: Bu kesme stack taşması olduğunda üretilir.

13 Numaralı İçsel Kesme: Bu kesmeye "General Protection Fault" denilmektedir. Bu kesme korumalı modda özel makine komutlarının çalıştırılması sırasında, segment limitlerinin aşıldığı durumlarda ve diğer bazı kontrollerdeki başarısızlıklar nedeniyle oluşur.

14 Numaralı İçsel Kesme: Bu kesmeye "Page Fault" denilmektedir. Bir sayfaya erişim sırasında sayfa fiziksel bellekte değilse ya da read-only bir sayfaya yazma yapılmak istenişse bu kesme oluşturulur.

Fault tarzı bir içsel kesme oluştuğunda bir hata kodu da üretilmektedir. Her fault'a özgü bir hata durumu vardır. Örneğin "General Protection Fault" isimli içsel kesme oluştuğunda işlemci aynı zamanda bu içsel kesmenin neden oluştuğunu anlatan bir hata kodunu yükleme işleminin yapıldığı segment yazmacının "index birlerine" kodlamaktadır. Örneğin:

```
MOV CS, AX
```

Burada AX içerisinde 0 değeri olsun. Bir selektöre 0 değerine ilişkin betimleyici yüklenmek istenirse "General Protection Fault" kesmesi oluşur. İşte burada fault sırasında CS yazmacının yüksek anlamlı 13 biti (index bitleri) hatanın nedenini belirten bir değerle doldurulacaktır. Bu konuda ayrıntılı bilgi Intel dokümanlarından izlenebilir. Örneğin doğrusal adresin fiziksel adrese dönüşütürülmesi sırasında eğer sayfa fiziksel bellekte değilse işlemci "Page Fault" isimli kesmeyi oluşturacaktır. Bu kesme oluştuğunda işlemci kesmenin oluşmasına yol açan doğrusal adresi CR2 yazmacına yükler. Ayrıca bu kesme oluştuğunda kesmenin nedne oluştuğuna ilişkin hata kodu stack'in tepesine push edilmektedir. Böylece kesme kodu stack'in tepesinden bu değeri alabilir.

Intel MMX, SSE ve AVX Komut Kümeleri

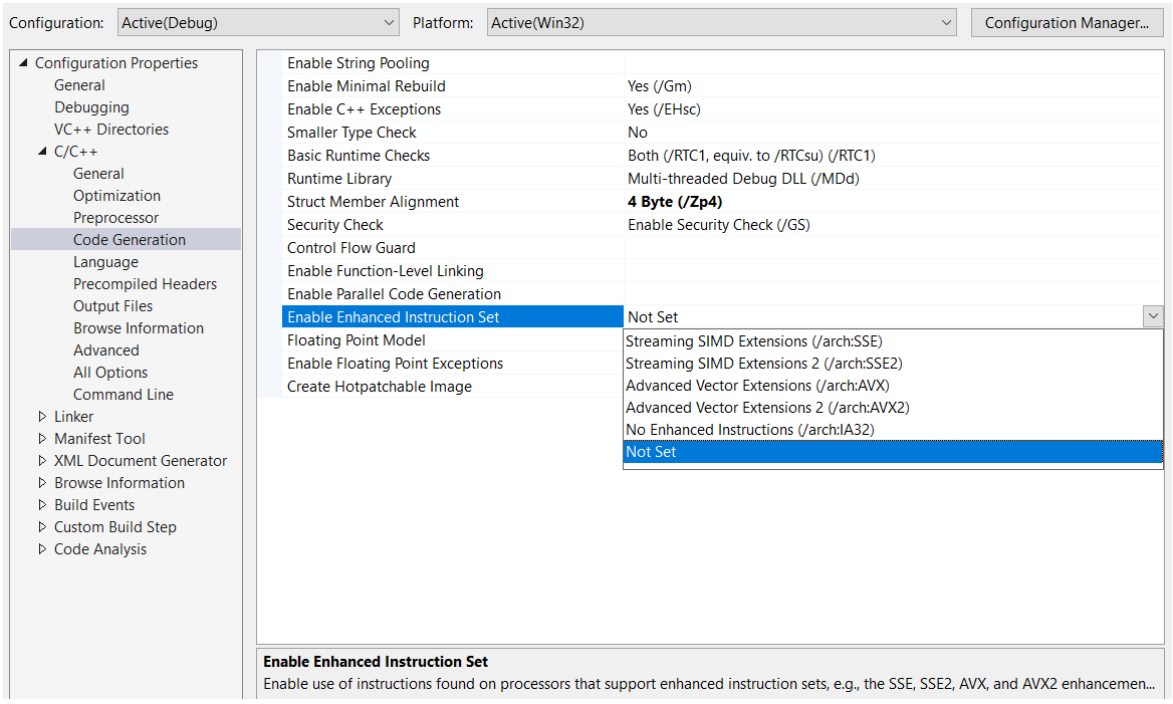
Tıpkı yazılımlarda olduğu gibi donanımlara ve işlemcilere de versiyon ilerledikçe birtakım eklentiler (extentions) eklenmektedir. Intel işlemcilerine de MMX, SSE ve AVX isimleri altında ciddi bazı eklemeler

yapılmıştır.

Tarihsel olarak ilk küme eklentisi MMX'tir. MMX "MultiMedia eXtension" sözcüklerinden kısaltılmıştır. İlk kez 1997'de Pentium işlemcilerine dahil edilmiştir. MMX teknolojisi ile Intel 8 tane 64 bitlik MM yazmacını (MM0-MM7) mimariye dahil etmiştir. Bu yazmaçlar üzerine paralel artimetik ve bitsel işlemler yapan komutlar vardır. Böylece programcı tek bir komutla aynı anda birden fazla 8 bit, 16 bit, 32 bit işlemleri yapabilmektedir. Tek bir makine komutuyla birden fazla işlemin paralel birlikte yapılması modeline SIMD (Single Instruction Multiple Data)" denilmektedir. SIMD tarzı komutlar modern işlemcilerde artık yaygın olarak görülmektedir. Özellikle görüntü işleme, işaret işleme uygulamalarında bu komutlar önemli ölçüde yer ve hız kazancı sağlamaktadır. Ancak MMX teknolojisi gerçek sayı işlemlerine yönelik değildi. MMX komutları yalnızca tamsayılar üzerinde işlemler yapabilmektedir.

MMX teknolojisinden sonra Intel "SSE (Streaming SIMD Extensions)" ismiyle mimariye yeni SIMD yazmaçları ve komutları ekledi. Bu yeni SIMD yazmaçları XMM öneki ile isimlendirilmiştir. Her biri 128 bit (16 byte) olan toplam 8 tane XMM yazmacı vardır (XMM0-XMM7). SSE ile eklenen XMM yazmaçları ve komutları gerçeksayı işlemlerine yöneliktir. Gerçekten Intel SSE'yi mimariye dahil ettikten sonra gerçeksayı işlemcisininm başı F ile başlayan komutlarının büyük bölümü artık derleyiciler tarafından kullanılmaz olmuştur. SSE gerçek işlemciyle bütünleşik tasarlandığı için gerçeksayı işlemcisine göre daha hızlı çalışmaktadır. Intel SSE teknolojisine birkaç kez eklentiler de yapmıştır. SSE2'de XMM yazmaçları tıpkı MMX yazmaçları gibi tamsayı işlemlerinde kullanılabilir hale gelmiştir. SSE3 ve SSE4'te de birçok komut eklemesi yapılmıştır. Nihayet Intel ve AMD anlaşarak 64 bir işlemcilerine "AVX (Advanced Vector Extensions)" ismiyle yeni yazmaçlar ve komutlar eklemiştir. Böylece mimariye YMM isimli her biri 256 bit olan 16 tane yazmaç eklenmiştir. Bu 16 yazmaç daha sonra 32'ye çıkartılmıştır. YMM yazmaçlarının düşük anlamlı 128 biti XMM yazmaçları gibidir. Nihayet AVX'in sonraki sürümlerinde Intel ve AMD bu kez her biri 512 bit (64 byte) olan ZMM yazmaçlarını ve komutlarını mimariye eklemiştir.

Intel'in buarad ele aldığımız eklenti komutları tüm Pentium modellerinde bulunmamaktadır. Bu nedenle genel derleyiciler default durumda bu yeni eklentileri kullanmazlar. Örneğin Visual Studio IDE'sinde (cl.exe komut satırı derleyicisinde) biz derleyicinin bu komutları kullanmasını sağlayabiliriz:



Biz burada önce SSE eklentilerini, daha sonra MMX eklentilerini daha sonra da AVX eklentilerini ele alıp açıklayacağız.

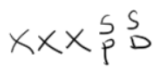
Intel SSE SIMD Komutları

Intel'in SSE komutları için mimariye 8 tane 128 bit (16 byte) XMM yazmaçları (XMM0-XMM7) eklenmiştir. SSE komutları ve bu yazmaçlar gerçek sayı işlemlerine yöneliktir. Ancak SSE2 (yani SSE'nin sonraki versiyonu) ile birlikte bunlar tamsayı işlemleri de yapar hale getirilmiştir. Temel SSE komut kümesinin iki özelliği vardır:

1) Bunlar gerçek sayı işlemleri yapan komutlardan oluşmuştur. Dolayısıyla gerçek sayı işlemleri bu sayede artık matematik işlemci tarafından değil asıl işlemcideki SSE birimi tarafından daha hızlı ve etkin yapılabilmektedir.

2) SSE komutları SIMD (Single Instruction Multiple Data) işlemlerine izin vermektedir. Dolayısıyla aynı anda birden fazla gerçek sayı işlemi birlikte yapılabilmektedir.

SSE komutlarının isimleri (menemonics) genel olarak şöyledir:



Burada XXX komutun ne yaptığına yönelik ismi temsil etmektedir. Komutun sonundaki S ya da D işlemin hangi duyarlılıkta yapıldığını belirtir. Buradaki S "Single" sözcüğünden kısaltılmıştır ve "float" türünü temsil eder. D ise "Double" sözcüğünden kısaltılmıştır ve double türünü temsil eder. En sondaki karakterden bir önceki karakter yine S ya da P olabilir. Buradaki S tek bir değer üzerinde işlem yapılacağını (single), P ise (packet) birden fazla değer üzerinde paralel işlem yapılacağını belirtir.

İlk öğrenilecek SSE komutu MOVXX komutlarıdır. Bunlar bellek ile XMM yazmaçları arasında ya da XMM yazmaçlarının kendisi arasında yükleme yaparlar. Bellek operandı yine daha önceden sözü edilen biçimde oluşturulmaktadır. Örneğin:

```
MOVSS XMM0, [EBX]
```

Burada EBX yazmacının belirttiği adresten 4 byte'lık float değer XMM0 yazmacına (onun düşük anlamlı 4 byte'ına) yerleştirilmektedir. Örneğin:

```
MOVSS [EBX], XMM0
```

Burada da XMM0 yazmacının içerisindeki (onun düşük anlamlı 4 byte'ındaki float değer belleğe EBX yazmacının gösterdiği yere aktarılmaktadır. Örneğin:

```
MOVSD XMM0, XMM1
```

Burada XMM1 içerisindeki double değer XMM0 içerisine atanmıştır. Örnein:

Komutun SIMD biçimi (yani P'li biçimi) için iki ayrı seçenek vardır: Hizalamalı (aligned) ve hizalamasız (unaligned). Hizamalı biçim için bellek elemanının 16'nın katlarında olması gerekir. Hizalamasız biçimde böyle bir zorunluluk yoktur. Hizalamalı biçim MOVAPS ya da MOVAPD ismi ile hizalamasız biçim ise MOVUPS ya da MOVUPD ismi ile temsil edilmektedir. Bu komutlar 4 tane float değeri ya da 2 tane double değeri mov işlemine sokarlar. Örneğin:

```
MOVUPS XMM0, [EBX]
```

Bu komut ile EBX adresinden itibaren toplam 4 float değer XMM0 yazmacına yüklenmektedir.

Toplama işlemi için ADDXX komutları vardır. Bunlar benzer biçimde yazmaç yazmaç ve bellek yazmaç arasında açlışabilirler. Bu komutlar da yine ADDSS, ADDSD, ADDPS, ADDPD biçimlerine sahiptir.

Örneğin biz iki float sayıyı SSE komut kümesini kullanarak toplayan bir C fonksiyonunu 32 bit Windows sistemleri için NASM ile aşağıdaki gibi yazabiliriz:

```
[BITS 32]

SECTION .text
    global _Add

_Add:
    push ebp
    mov     ebp, esp
    movss   xmm0, [ebp + 8]
    addss   xmm0, [ebp + 12]
    sub     esp, 4
    movss   [ebp - 4], xmm0
```

```
    fld     dword [ebp - 4]

    mov     esp, ebp
    pop     ebp
    ret
```

Test işlemi de şöyle bir kodla yapılabilir:

```
#include <stdio.h>

float Add(float a, float b)
{
    return a + b;
}

int main(void)
{
    float a = 10.1, b = 20.2, c;

    c = Add(a, b);
    printf("%f\n", c);

    return 0;
}
```

Sembolik makine dili kodunda Add fonksiyonunun geri dönüş değerinin yine matematik işlemcinin stack yazmacında bulundurulduğunda dikkat ediniz. Biz 32 bit cdecl çağırma biçiminde SSE komutlarını kullanıyor olsak bile fonksiyonun geri dönüş değeri gerçek sayı türündense yine onu matematik işlemcinin stack yazmacında bırakmak zorundayız.

Şimdi iki tane 4 float değeri tek hamlede toplayan AddVect isimli bir fonksiyonu yazalım:

```
[BITS 32]

SECTION .text
    global _AddVect

_AddVect:
    push ebp
    mov     ebp, esp

    mov     eax, [ebp  + 8]
    movups  xmm0, [eax]
    mov     eax, [ebp + 12]
    movups  xmm1, [eax]
    addps   xmm0, xmm1

    mov     eax, [ebp + 16]
    movups  [eax], xmm0

    mov     esp, ebp
    pop     ebp
    ret
```

Kod şöyle test edilebilir:

```
#include <stdio.h>

typedef struct tagFLOATS {
    float a, b, c, d;
} FLOATS;

void AddVect(const FLOATS *f1, const FLOATS *f2, FLOATS *f3);

int main(void)
{
    FLOATS f1 = { 10.1, 10.2, 10.3, 10.4 }, f2 = { 10.1, 10.2, 10.3, 10.4 }, f3;
```

```
    AddVect(&f1, &f2, &f3);

    printf("%f\n%f\n%f\n%f\n", f3.a, f3.b, f3.c, f3.d);

    return 0;
}
```

Aynı işlem iki double değer için de şöyle yapılabilirdi:

```
; Util.

[BITS 32]

SECTION .text
    global _AddVect

_AddVect:
    push ebp
    mov      ebp, esp

    mov      eax, [ebp  + 8]
    movupd   xmm0, [eax]
    mov      eax, [ebp + 12]
    movupd   xmm1, [eax]
    addpd    xmm0, xmm1

    mov      eax, [ebp + 16]
    movupd   [eax], xmm0

    mov      esp, ebp
    pop      ebp
    ret
```

Aşağıdaki kodla test işlemi yapılabilir:

```
#include <stdio.h>

typedef struct tagDOUBLES {
    double a, b;
} DOUBLES;

void AddVect(const DOUBLES *f1, const DOUBLES *f2, DOUBLES *f3);

int main(void)
{
    DOUBLES f1 = { 10.1, 10.2}, f2 = { 10.1, 10.2}, f3;

    AddVect(&f1, &f2, &f3);

    printf("%f\n%f\n\n", f3.a, f3.b);

    return 0;
}
```

Benzer biçimde çıkartma işlemi için SUBSS, SUBSD, SUBPS ve SUBPD komutları çarpma için MULSS, MULSD, MULPS ve MULPD komutları, bölme için de DIVSS, DIVSD, DIVPS ve DIVPD komutları vardır. Örneğin 4 float sayının çarpımı örneği aşağıdaki gibi olabilir:

```
; Util.

[BITS 32]

SECTION .text
    global _MulVect

_MulVect:
```

```
    push ebp
    mov      ebp, esp

    mov      eax, [ebp  + 8]
    movups   xmm0, [eax]
    mov      eax, [ebp + 12]
    movups   xmm1, [eax]
    mulps    xmm0, xmm1

    mov      eax, [ebp + 16]
    movups   [eax], xmm0

    mov      esp, ebp
    pop      ebp
    ret
```

Kod şöyle test edilebilir:

```
#include <stdio.h>

typedef struct tagFLOATS {
    float a, b, c, d;
} FLOATS;

void AddVect(const FLOATS *f1, const FLOATS *f2, FLOATS *f3);

int main(void)
{
    FLOATS f1 = { 10.1, 10.2, 10.3, 10.4 }, f2 = { 10.1, 10.2, 10.3, 10.4 }, f3;

    MulVect(&f1, &f2, &f3);

    printf("%f\n%f\n%f\n%f\n", f3.a, f3.b, f3.c, f3.d);

    return 0;
}
```

SSE komutları XMM yazmaçlarıyla bellek arasında da işlem yapabilmektedir. Örneğin:

```
ADDPS   XMM0, [EBX]
```

komutuyla XMM0 yazmacındaki 4 float değer paralel olarak EBX adresindeki 4 float değerler toplama işlemine sokulmaktadır. Ancak bu duurmda bellek operandlarının 16'ya hizalanması gerekir. Yani bu 4 float (ya da 2 double) değerin 16'nın katlarında bulunması gerekir. Bunun için bildirimde Microsoft derleyicilerinde __declspec(align(n)) belirleyicisi, gcc derleyicilerinde de __attribute__((aligned(n)) belirleyicileri kullanılmalıdır. Örneğin:

```
[BITS 32]

SECTION .text
    global _AddVect

_AddVect:
    push ebp
    mov      ebp, esp

    mov      eax, [ebp  + 8]
    movupd   xmm0, [eax]
    mov      eax, [ebp + 12]

    addps    xmm0, [eax]

    mov      eax, [ebp + 16]
    movupd   [eax], xmm0

    mov      esp, ebp
```

```
    pop     ebp
    ret
```

Test işlemi aşağıdaki kodla yapılabilir:

```
#include <stdio.h>

typedef struct tagFLOATS {
    float a, b, c, d;
} FLOATS;

void AddVect(const FLOATS *f1, const FLOATS *f2, FLOATS *f3);

int main(void)
{
    FLOATS __declspec(align(16)) f1 = { 10.1, 10.2, 10.3, 10.4 }, f2 = { 10.1, 10.2, 10.3, 10.4 }, f3;

    AddVect(&f1, &f2, &f3);

    printf("%f\n%f\n%f\n%f\n", f3.a, f3.b, f3.c, f3.d);

    return 0;
}
```

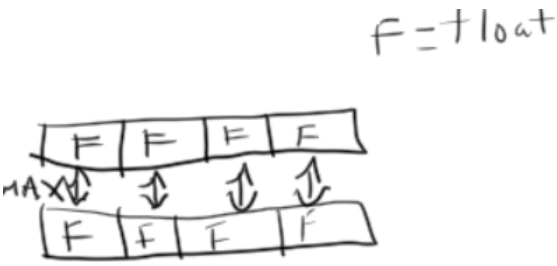
Karekök işlemi için SQRTSS, SQRTSD, SQRDPS ve SQRTPD komutları bulundurulmuştur. Örneğin:

```
; Util.

[BITS 32]

SECTION .text
    global _mysqrt

_mysqrt:
    push ebp
    mov     ebp, esp
    sub     esp, 8

    movsd   xmm0, [ebp + 8]
    sqrtsd  xmm1, xmm0
    movsd   [ebp - 8], xmm1
    fld     qword [ebp -8]


    mov     esp, ebp
    pop     ebp
    ret
```

Aşağıdaki kod ile test işlemi yapılabilir:

```
#include <stdio.h>

double mysqrt(double a);

int main(void)
{
    double a = 10, b;

    b = mysqrt(a);
    printf("%f\n", b);

    return 0;
}
```

MAXPS ve MAXPD komutları 4 float ya da 2 double değerin en büyüklerini verir.



Örneğin:

```
MAXPS   XMM0, XMM1
```

Burada XMM0'ın düşük anlamlı 4 byte'ındaki float değer ile XMM1'in düşük anlamlı 4 byte'ındaki float değer karşılaştırılıp hangisi büyükse XMM0'ın düşük anlamlı 4 byte'ına float olarak atanır. Sonra aynı işlem geri kalan 3 float değer için de aynı biçimde yapılır.

MINPS ve MINPD komutları da aynı işlemi minimum değer için yapar.

XMM komut kümesi daha sonraları tam sayılar üzerinde de işlem yapabilir hale getirildi. Bazı tamsayı işlemleriburada ele alacağız. Tamsayı işlemleri "paket (packet)" yani "vektör" olarak yapılmaktadır. Aslında XMM teknolojisinden önceki MMX teknolojisi zaten aşağıda ele alıncak olan tamsayı işlemlerini vektörel olarak yapabiliyordu. SSE komutlarının temel yeniliği gerçek sayılar üzerinde işlemler yapmasıydı. Fakat ne olursa olsun MMX yazmaçları 64 bitti SSE ile birlikte MMX'teki bu komutlar XMM yazmaçlarıyla yani 128 bit biçimde işlem yapabilecek hale getirilmiştir.

ANDPS ve ANDPD komutları sırasıyla 4 byte'lık 4 değeri ve 8 byte'lık 2 değeri bit düzeyinde and işlemine sokarlar. Örneğin:

```
[BITS 32]

SECTION .text
    global _AndVect

_AndVect:
    push ebp
    mov     ebp, esp

    mov     eax, [ebp  + 8]
    movupd  xmm0, [eax]
    mov     eax, [ebp + 12]

    andps   xmm0, [eax]

    mov     eax, [ebp + 16]
    movupd  [eax], xmm0

    mov     esp, ebp
    pop     ebp
    ret
```

Aşağıdaki kodla test işlemi yapılabilir:

```
#include <stdio.h>
#include <immintrin.h>

typedef struct tagINTS32 {
    int a, b, c, d;
} INTS32;

void AndVect(const INTS32 *i1, const INTS32 *i2, INTS32 *i3);

int main(void)
{
```

```
    INTS32 __declspec(align(16)) i1 = {0x12345678, 0x12345678, 0x12345678, 0x12345678 },
        i2 = {0x87654321, 0x87654321, 0x87654321, 0x87654321 }, i3;

    AndVect(&i1, &i2, &i3);

    printf("%08X\n%08X\n%08X\n%08X\n", i3.a, i3.b, i3.c, i3.d);

    return 0;
}
```

Benzer biçimde ORPS ve ORPD komutları bit düzeyinde vektörel OR işlemini XORPS ve XORPD komutları da bit düzeyinde vektörel EXOR şlemi yapar.

Vektörel toplama için PADDB, PADDW, PADDD ve PADDQ komutları bulundurulmuştur. Bunlar sırasıyla 16 tane 1 byte'lık, 8 tane 2 byte'lık, 4 tane 4 byte'lık ve 2 tane 8 byte'lık değerleri vektörel olarak toplama işlemine sokar. Örneğin:

```
; Util.

[BITS 32]

SECTION .text
    global _AddIntVect

_AddIntVect:
    push ebp
    mov     ebp, esp

    mov     eax, [ebp  + 8]
    movupd  xmm0, [eax]
    mov     eax, [ebp + 12]

    paddd   xmm0, [eax]

    mov     eax, [ebp + 16]
    movupd  [eax], xmm0

    mov     esp, ebp
    pop     ebp
    ret
```

Aşağıdaki kodla test işlemi yapılabilir:

```
#include <stdio.h>
#include <immintrin.h>

typedef struct tagINTS32 {
    int a, b, c, d;
} INTS32;

void AddIntVect(const INTS32 *i1, const INTS32 *i2, INTS32 *i3);

int main(void)
{
    INTS32 __declspec(align(16)) i1 = {10, 11, 12, 13}, i2 = {14, 15, 16, 17}, i3;

    AddIntVect(&i1, &i2, &i3);

    printf("%d\n%d\n%d\n%d\n", i3.a, i3.b, i3.c, i3.d);

    return 0;
}
```

Benzer biçimde PSUBB, PSUBW, PSUBD ve PSUBQ komutları vektörel çıkartma işlemi için kullanılmaktadır. PMULLW, PMULLD, PMULLDQ komutları da vektörel çarpma amacıyla kullanılmaktadır. PMULLW komutu 4 tane 4 byte'lık tamsayıyı vektörel olarak çarpar sonuçların düşük anlamlı byte'larını hedefe yerleştirir. PMULLD

ve PMULLDQ benzer işlemleri yapmaktadır.

Intel AVX SIMD Komutları

AVX (Advanced Vector Extensions) SSE teknolojisinin bir ileri versiyonu olarak çıkmıştır. AVX teknolojisi de kendi içerisinde AVX, AVX-2 ve AVX-512 biçiminde uyarlamalara sahiptir. AVX teknolojisiyle birlikte şu yenilikler Intel ve AMD işlemcilerine eklenmiştir:

- XMM yazmaçları genişletilerek YMM biçimine getirildi ve bunlar 256 bite (32 byte'a) yükseltildi. Dolayısıyla YMM komutlarıyla biz örneğin 8 tane float sayı ya da 4 tane double sayı üzerinde vektörel işlemler yapabilmekteyiz.

- YMM yazmaçlarının düşün anlamlı 128 biti XMM olarak organize edildi. Böylece biz bu teknolojide örneğin XMM0 yazmacına atama yaptığımızda aynı zamanda YMM0 yazmacının düşük anlamlı 128 bitine atama yapmış olmaktayız.

- AVX teknolojisi işlemcinin 32 bit modunda 8 tane YMM yazmacına sahiptir. Ancak 64 bit modunda 16 tane YMM yazmacı kullanılabilmektedir. (Halbuki SSE teknolojisinde hem 16 bitte hem de 32 bitte toplam 8 XMM yazmacı bulunmaktadır.)

- AVX ve AVX-2 eklentilerinde yeni birtakım ekstra komutlar da işlemciye dahil edilmiştir.

- AVX'in son sürümü AVX-3 yerine AVX-512 biçiminde isimlendirilmiştir. Bu sürümde hem YMM yazmaçlarının sayısı 32'ye yükseltilmiş hem de 512 bitlik (64 byte'lık) yeni ZMM yazmaçları mimariye eklenmiştir.

AVX komut kümesi Intel ve AMD işlemcilerine 2011 yılında eklenmiştir. 2011'den sonraki Core-I modelleri genel olarak bu teknolojisye sahiptir. Ancak AVX-512 çok yenidir ve Core-I modellerinde şimfdilik bu teknoloji yoktur. AVX-512 şimdilik Intel'in Xeon işlemcilerinde kullanılmaktadır. O halde AVX komutlarını kullanan programların öncelikle CPU'nun bu komutları destekleyen bir CPU mu olup olmadığı test etmeleri gerekmektedir. Örneğin bir program CPU'u test ederek onun AVX-512 desteği olduğunu anlarsa bir fonksiyon kümesini AVX desteği olduğunu anlarsa diğer bir fonksiyon kümesini, SSE deteği olduğunu anlarsa diğer bir fonksiyon kümesini kullanabilir. Intel ve AMD işlemcilerinde SSE, SSE2, SSE3, SSE4, AVX, AVX-2 ve AVX-512 destekleri sorgulanabilmektedir. Bu işlemin nasıl yapıldığı buarada açıklanmayacaktır. Intel ya da AMD dokümanlarına bakılabilir.

YMM yazmaçlarını kullanan AVX komutları SSE komutları ile büyük ölçüde aynıdır. Yalnızca komutların başına 'V' harfi getirilmektedir. Örneğin: VADDPS, VMULPD, VMOVUPS gibi. Aşağıda 8 tane float sayıyı toplayan bir örnek verilmiştir:

```
[BITS 32]

SECTION .text
    global _AddVect

_AddVect:
    push ebp
    mov      ebp, esp

    mov      eax, [ebp  + 8]
    vmovupd  ymm0, [eax]
    mov      eax, [ebp + 12]

    vaddps   ymm0, [eax]

    mov      eax, [ebp + 16]
    vmovupd  [eax], ymm0

    mov      esp, ebp
    pop      ebp
    ret
```

Aşağıdaki gibi bir kodla test işlemi yapılabilir:

```
#include <stdio.h>
```

```
typedef struct tagFLOATS {
    float a, b, c, d, e, f, g, h;
} FLOATS;

void AddVect(const FLOATS *f1, const FLOATS *f2, FLOATS *f3);

int main(void)
{
    FLOATS __declspec(align(16)) f1 = { 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8 }, f2 =
{ 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8 }, f3;

    AddVect(&f1, &f2, &f3);

    printf("%f\n%f\n%f\n%f\n%f\n%f\n%f\n%f\n", f3.a, f3.b, f3.c, f3.d, f3.e, f3.f, f3.g, f3.h);

    return 0;
}
```

SSE ve AVX Makine Komutlarının C'de Fonksiyon Biçiminde Kullanılması

Bilindiği gibi derleyicinin hiç prorotip görmeden doğrudan tanıdığı ve CALL işlemiyle değil de inline biçimde kod açarak işleme soktuğu derleyiciye özgü özel fonksiyonlara "içsel fonksiyonlar (intrinsic functions)" denilmektedir. Tabii içsel fonksiyon kavramı C standartlarında yoktur. Bunlar birer eklenti olarak derleyicilerde bulunmaktadır. Dolayısıyla her C derleyicisinin içsel fonksiyon listesi diğerlerinden farklı olabilir. İşte Microsoft C ve gcc derleyicilerinin de SSE ve AVX komutlarını kullanmamızı sağlayan içsel fonksiyonları vardır. O halde biz örneğin SSE ve AVX vektörel komutlarını kullanmak için sembolik makine dilinden faydalanmak zorunda değiliz.

SSE ve AVX komutları için Intel çeşitli veri türleri ve intrinsic fonksiyon prototipleri önermiştir. Hem Microssoft C/C++ derleyicileri hem de gcc derleyicileri bu fonksiyonları ve veri türlerini aynı biçimde desteklemektedir. Intel'in bu dokümanlarına "Intel Intrinsic Guide" denilmektedir, https://software.intel.com/sites/landingpage/IntrinsicsGuide/ adresinden erilebilir.

SSE ve AVX makine komutları için aşağıdaki veri türleri tanımlanmıştır:

Tür	Anlamı
__m128	128 bit vektörel float türü. Yani 4 float değeri temsil eder.
__m128i	128 bit vektörel tamsayı türünü temsil eder. Yani 16 tane 1 byte'lık, 8 tane 2 byte'lık 4 tane 4 byte'lık ya da 2 tane 8 byte'lık tamsayı değeri temsil eder.
__m128d	128 bit vektörel double türü. Yani 2 double değeri temsil eder.
__m256	256 bit vektörel float türü. Yani 8 float değeri temsil eder.
__m256i	128 bit vektörel tamsayı türünü temsil eder. Yani 32tane 1 byte'lık, 16 tane 2 byte'lık 8 tane 4 byte'lık ya da 4 tane 8 byte'lık tamsayı değeri temsil eder.
__m256d	256 bit vektörel double türü. Yani 4 double değeri temsil eder.

Bu veri türlerinin nasıl typedef edileceğini Intel belirtmemiştir. Dolayısıyla bu typedef biçimi derleyiciden derleyiciys değişebilmektedir. Örneğin __m128 türü Microsoft derleyicilerinde şöyle type edilmiştir:

```
typedef union __declspec(intrin_type) __declspec(align(16)) __m128 {
     float              m128_f32[4];
     unsigned __int64   m128_u64[2];
     __int8             m128_i8[16];
     __int16            m128_i16[8];
     __int32            m128_i32[4];
     __int64            m128_i64[2];
     unsigned __int8    m128_u8[16];
     unsigned __int16   m128_u16[8];
     unsigned __int32   m128_u32[4];
 } __m128;
```

__128i türü ise şöyle typedef edilmiştir:

```
typedef union __declspec(intrin_type) __declspec(align(16)) __m128i {
    __int8              m128i_i8[16];
    __int16             m128i_i16[8];
    __int32             m128i_i32[4];
    __int64             m128i_i64[2];
    unsigned __int8     m128i_u8[16];
    unsigned __int16    m128i_u16[8];
    unsigned __int32    m128i_u32[4];
    unsigned __int64    m128i_u64[2];
} __m128i;
```

__m256 türü şöyle typedef edilmiştir:

```
typedef union __declspec(intrin_type) __declspec(align(32)) __m256 {
    float m256_f32[8];
} __m256
```

__m256i türü ise şöyle typedef edilmiştir:

```
typedef union  __declspec(intrin_type) __declspec(align(32)) __m256i {
    __int8              m256i_i8[32];
    __int16             m256i_i16[16];
    __int32             m256i_i32[8];
    __int64             m256i_i64[4];
    unsigned __int8     m256i_u8[32];
    unsigned __int16    m256i_u16[16];
    unsigned __int32    m256i_u32[8];
    unsigned __int64    m256i_u64[4];
} __m256i
```

Yukarıdaki typedef'lere bakıldığında biz bu türlere ilişkin nesnelere küme parantezi içerisinde ilkdeğer verebileceğimiz anlamı çıkmaktadır. Örneğin:

```
__m128 a = {10.1, 10.2, 10.3, 10.4};
```

Fakat bu durum taşınabilirlik bakımından tavsiye edilmez. Çünkü yukarıda da belirtildiği gibi bu türlerin nasıl typedef edileceği standart olarak belirlenmemiştir. Bu nedenle bu nesnelere atam işlemlerinin yine intrinsiz fonksiyonlarla yapılması tavsiye edilir:

Fonksiyon	İşlevi
`__m128 _mm_set_ss(float a);`	Tek bir float değeri __m128 türüne yerleştirmek için kullanılır. Hedef türün yüksek anlamlı diğer byte'ları sıfırlanmaktadır.
`__m128d _mm_set_sd double a);`	Tek bir double değeri __m128d türüne yerleştirmek için kullanılır. Hedef türün yüksek anlamlı diğer byte'ları sıfırlanmaktadır.
`__m128 _mm_set_ps (float e3, float e2, float e1, float e0);`	Dört float değeri __m128 türüne atamak için kullanılır.
`__m128d _mm_set_pd (double e1, double e0);`	İki double değeri __m128d türüne yerleştirir.
`void _mm_store_ss (float* mem_addr, __m128 a);`	__m128 içerisindeki tek bir float değeri (düşük anlamlı float değeri) adresiyle aldığı float nesnenin içerisine yerleştirir.
`void _mm_store_sd (double* mem_addr, __m128d a);`	__m128d içerisindeki değeri adresiyle aldığı double nesneye yerleştirir.
`void _mm_storeu_ps (float* mem_addr, __m128 a);`	4 float değeri __m128 türünden alarak bir float diziye yerleştirir.
`void _mm_storeu_pd (double* mem_addr, __m128d a);`	2 double değeri __m128d türünden alarak bir double diziye yerleştirir.
`__m128i _mm_set_epi8 (char e15, char e14, char e13, char e12, char e11, char e10, char e9, char e8, char e7, char e6, char e5, char e4, char e3, char e2, char e1, char e0)`	16 tane 1 byte'lık tamsayı değeri __m128i türünden nesneye yerleştirmek için kullanılır.
`__m128i _mm_set_epi16 (short e7, short e6,`	8 tane 2 byte'lık tamsayı değeri __m128i türünden nesneye

`short e5, short e4, short e3, short e2, short e1, short e0);`	yerleştirmek için kullanılır.
`__m128i _mm_set_epi32 (int e3, int e2, int e1, int e0);`	4 tane 4 byte'lık tamsayı değeri __m128i türünden nesneye yerleştirmek için kullanılır.
`__m128i _mm_set_epi64(__m64 e1, __m64 e0);`	4 tane 4 byte'lık tamsayı değeri __m128i türünden nesneye yerleştirmek için kullanılır.

Örneğin:

```
__m128 a = _mm_set_ss(12.3F);
__m128d b = _mm_set_sd(12.3);
__m128 c = _mm_set_ps(10.1F, 10.2F, 10.3F, 10.4F);
__m128d d = _mm_set_pd(10.1, 10.2);
float e;
_mm_store_ss(&e, a);
double f;
_mm_store_sd(&f, b);
float g[4];
_mm_storeu_ps(g, a);
double h[2];
_mm_storeu_pd(h, d);
__m128i i;
i = _mm_set_epi8(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16);
i = _mm_set_epi16(1, 2, 3, 4, 5, 6, 7, 8);
i = _mm_set_epi32(1, 2, 3, 4);
i = _mm_set_epi64(1, 2);
```

İşlem yapan intrinsic fonksiyonların isimlendirme biçimleri şöyledir: Fonksiyonların başı her zaman _mm ile başlamaktadır. Sonra bunu yapılacak işleme ilişkin bir sözcük izler (örneğin _mm_add_xxx). Bunu da daha sonra "ss", "sd", "ps", "pd", "pixx" sözcükleri izlemektedir. "ss" tek bir float değer için işlem yapılacağını, "sd" tek bir double değer için işlem yapılacağını, "ps" vektörel float için işlem yapılacağını, "pd" ise vektöreş double için işlem yapılacağını belirtmektedir.

Örneğin:

```
#include <stdio.h>
#include <xmmintrin.h>

int main(void)
{
    __m128 a, b, c;
    float __declspec(align(16)) result[4];

    a = _mm_set_ps(10.1F, 10.2F, 10.3F, 10.4F);
    b = _mm_set_ps(10.5F, 10.6F, 10.7F, 10.8F);
    c = _mm_add_ps(a, b);
    _mm_store_ps(result, c);

    printf("%f, %f, %f, %f\n", result[3], result[2], result[1], result[0]);

    return 0;
}
```

XMM komutlarının performansı aşağıdaki gibi bir kodla test edilebilir:

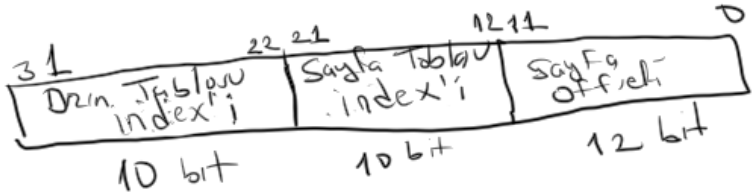
```
#include <stdio.h>
#include <Windows.h>
#include <xmmintrin.h>

#define COUNT    10000

int main(void)
```

```
{
    __m128 a, b, c;
    float __declspec(align(16)) f1[COUNT], f2[COUNT], f3[COUNT];
    int i;
    LARGE_INTEGER li1, li2, freq;
    __int64 result1, result2;

    QueryPerformanceFrequency(&freq);

    for (int i = 0; i < COUNT; ++i) {
        f1[i] = i;
        f2[i] = 2;
    }

    QueryPerformanceCounter(&li1);
    for (i = 0; i < COUNT; i += 4) {
        a = _mm_load_ps(f1 + i);
        b = _mm_load_ps(f2 + i);
        c = _mm_mul_ps(a, b);
        _mm_store_ps(f3 + i, c);
    }
    QueryPerformanceCounter(&li2);

    result1 = li2.QuadPart - li1.QuadPart;


    QueryPerformanceCounter(&li1);
    for (i = 0; i < COUNT; ++i)
        f3[i] = f1[i] * f2[i];
    QueryPerformanceCounter(&li2);

    result2 = li2.QuadPart - li1.QuadPart;
    printf("Ratio=%f\n\n", (double)result1 / result2);

    return 0;
}
```

```
Ratio=0.338028
```

Intel ve AMD İşlemcilerinin Sayfalama Modları

Biz daha önce 32 bit Intel işlemcilerinde doğrusal adreslerin nasıl fiziksel adreslere dönüştürüldüğünü görmüştük. Klasik 32 bit modda (legacy mode) doğrusal adres üç parçaya ayrılıyordu:



Pekiyi 64 bit long modda sayfalama nasıl yapılmaktadır? İşte Intel ve AMD aslında işlemcilerine pek çok sayfalama modu eklemiştir. Bugün 64 bit Windows, Linux ve Mac OS X sistemleri long modda 4K uzunluğunda sayfa modunu kullanmaktadır. Bu modda aşağıda da görüleceği gibi 48 bit (64 bit değil) doğrusal adres 52 bit fiziksel adrese dönüştürülmektedir. Yani başka bir deyişle bu modda bir programın sanal bellek alanı en fazla 2^{48} = 256 TB ve işlemcinin adresleyebileceği maksimum fiziksel RAM de 2^{52} = 4 PT kadardır. Tabii şu andaki ana kartlar genellikle maksimum 64 GB fiziksel RAM takılmasına izin vermektedir. Görüldüğü gibi Intel ve AMD "long mode"da her ne kadar yazmaçları 64 bite çıkarttıysa da ve adres bilgileri 64 bit olsa da tam olarak 64 bit bellek desteği vermemektedir. Tabii ileride bu iki firma bu değerleri yükseltecek biçimde tasarımlarını ilerletebilirler.

Aslında Intel ve AMD işlemcilerinde değişik configurasyonlarda bir çok sayfalama modu bulunmaktadır. Bu modlar arasında geçiş bazı kontrol yazmaçlarının bazı bitleriyle oynanarak yapılmaktadır. Aşağıda AMD dokğmanlarından alınmış olan bir tabloyu görüyorsunuz:

Table 5-1. Supported Paging Alternatives (CR0.PG=1)

Mode		Physical-Address Extensions (CR4.PAE)	Page-Size Extensions (CR4.PSE)	Page-Directory Pointer Offset	Page-Directory Page Size	Resulting Physical-Page Size	Maximum Virtual Address	Maximum Physical Address
Long Mode	64-Bit Mode	Enabled	–	PDPE.PS=0	PDE.PS=0	4 Kbyte	64-bit	52-bit
					PDE.PS=1	2 Mbyte		
	Compatibility Mode			PDPE.PS=1	–	1 Gbyte		
Legacy Mode		Enabled	–	PDPE.PS=0	PDE.PS=0	4 Kbyte	32-bit	52-bit
					PDE.PS=1	2 Mbyte		52-bit
		Disabled	Disabled		–	4 Kbyte		32-bit
			Enabled		PDE.PS=0	4 Kbyte		32-bit
					PDE.PS=1	4 Mbyte		40-bit

Buradaki tabloda “Legacy Mode” 32 bit çalışma modunu “Long Mode” ise 64 bit çalışma modunu belirtmektedir. 32 bit modda toplamda dört farklı sayfalama seçeneği vardır:

Sayfa Uzunluğu	Prosesin Doğrusal Adres Alanı	Maksimum Toplam Fiziksel RAM
4 KB (12 bit)	4 GB (32 bit)	4 GB (32 bit)
4 MB (22 bit)	4 GB (32 bit)	1 TB (40 bit)
4 KB (12 bit)	4 GB (32 bit)	4 PB (52 bit)
2 MB (21 bit)	4 GB (32 bit)	4 PB (52 bit)

32 bit işletim sistemleri daha önce de gördüğümüz 4 KB sayfa uzunluklu 4 GB doğrusal adresli ve en fazla 4 GB fiziksel adresli sayfalama modunu kullanmaktadır. Zaten en eskiden yalnızca bu mod vardı. Diğer modlar işlemci versiyonları ilerledikçe ortaya çıktı.

İşlemciyi 64 bit “long mode”a geçirdiğimizde sayfalama mod seçenekleri üçe inmektedir:

Sayfa Uzunluğu	Prosesin Doğrusal Adres Alanı	Maksimum Toplam Fiziksel RAM
4 KB (12 bit)	16 EB (64 bit) (Aslında 256 TB yani 48 bit)	4 PB (52 bit)
2 MB (21 bit)	16 EB (64 bit) (Aslında 256 TB yani 48 bit)	4 PB (52 bit)
1 GB (30 bit)	16 EB (64 bit) (Aslında 256 TB yani 48 bit)	4 PB (52 bit)

64 bit işletim sistemleri yukarıda da belirtildiği gibi genel olarak 4 KB sayfalı 256 TB doğrusal adresli, 4 PB fiziksel adresli modu kullanmaktadır.

Şimdi de yukarıdaki modlarda kullanılan sayfa dönüştürme mekanizmasını ele alalım.

32 Bit Çalışma Modunda 4K’lık Sayfalarla 4GB Fiziksel Belleğin Kullanımı

Bu konu sayfalama bölümünde ayrıntılarıyla ele alınmıştı. Anımsanacağı gibi bu modda CR3 yazmacı dizin tablosunun fiziksel adresini gösteriyordu. Doğrusal adresler de üç bileşene ayrılıyordu (Intel ve AMD dokümanlarında düşük adresin daha aşağıda gösterildiğini anımsayınız):

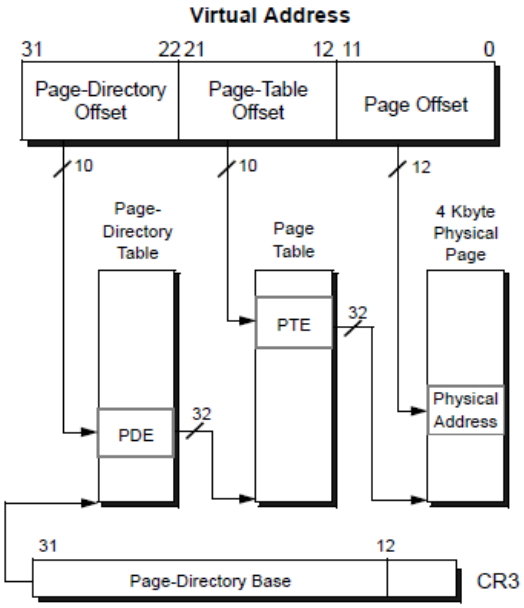


Figure 5-4. 4-Kbyte Non-PAE Page Translation—Legacy Mode

32 Bit Çalışma Modunda 4MB'lik Sayfalarla 1 TB Fiziksel Belleğin Kullanımı

İşlemci bu moda geçirildiğinde yine CR3 yazmacı dizin tablosunun fiziksel adresini gösterir. Ancak doğrusal adres (ya da sanal adres) iki parçadan oluşmaktadır: 10 bit Dizin Tablosu İndeksi ve 22 bit Sayfa Offseti

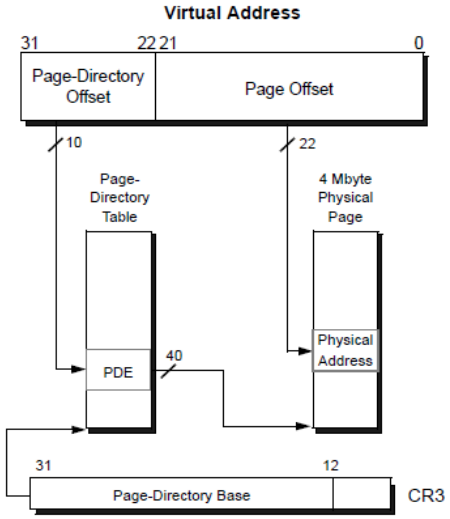


Figure 5-7. 4-Mbyte Page Translation—Non-PAE Paging Legacy-Mode

Görüldüğü gibi doğrusal adreslerin yüksek anlamlı 10 biti yine dizin tablosunda indeks belirtmektedir. Geri kalan 22 bit 4MB'lik sayfada offset belirtir. Bu modda dizin girişlerinin formatı da şöyledir:

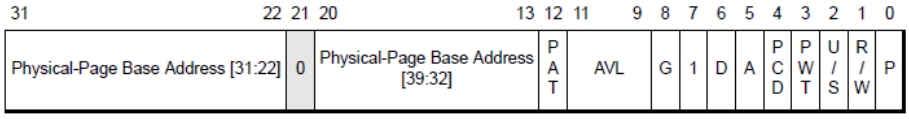


Figure 5-8. 4-Mbyte PDE—Non-PAE Paging Legacy-Mode

Bu modda doğrusal adres alanı 32 bit olmasına karşın fiziksel adres alanı 40 bittir. Dolayısıyla bu modda 1 TB fiziksel bellek kullanılabilir durumdadır. Dizin girişlerinde sayfanın yeri 4 MB'ye hizalanmıştır. (Yani

örneğin dizin girişinde 10 değeri yazıyorsa bu değer 10 * 4 MB anlamına gelir.) Dizin girişlerinde sayfalar için 18 bit ayrıldığına dikkat ediniz. O halde $2^{18}*2^{22} = 2^{40}$ = 1 TB fiziksel belleğe erişilebilmektedir.

32 Bit Çalışma Modunda 4K'lık Sayfalarla 4 PB Fiziksel Belleğin Kullanımı

Bu modda doğrusal adres alanı yine 32 bit yani 4 GB'dir. Ancak fiziksel adres alanı 52 bite (4 PB) yükseltilmiştir. Bu modda doğrusal adres 4 bileşene ayrılmaktadır: Dizin Tablosu Gösterici Offseti, Dizin Tablosu İndeksi, Sayfa Tablosu İndeksi ve Sayfa Offset'i.

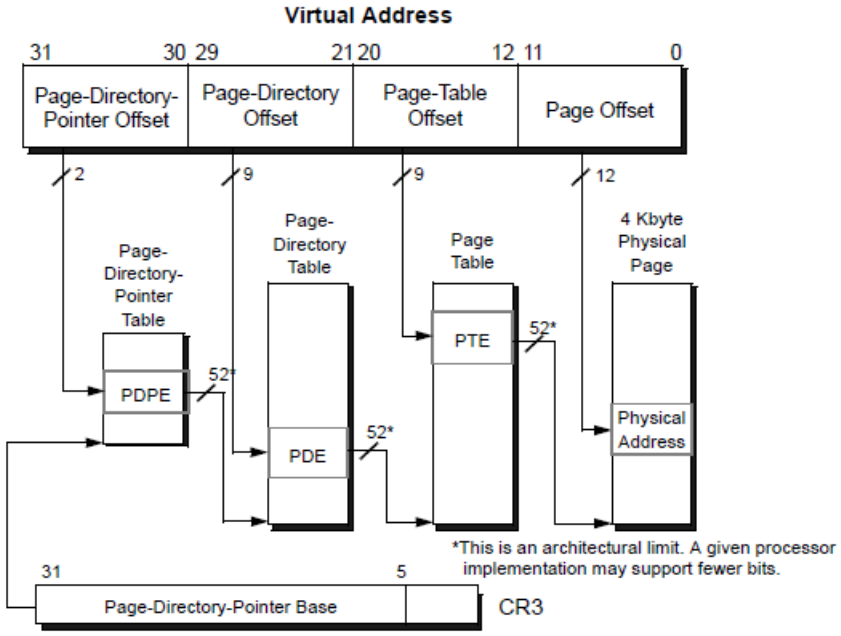


Figure 5-9. 4-Kbyte PAE Page Translation—Legacy Mode

Sayfa tablosu offset'i 2 bittir ve 4 girişten oluşmaktadır. Her giriş Dizin Tablosu Gösterici Tablosunda bir indeks belirtir. Böylece 4 tane dizin tablosu vardır. Her dizin tablosu girişi yine bir sayfa tablosunun yerini göstermektedir. Sayfa tabloları da yine sayfaların yerlerini gösterir. Dizin Tasblosu Gösterici Tablolarına ilişkin girişlerin formatı şöyledir:

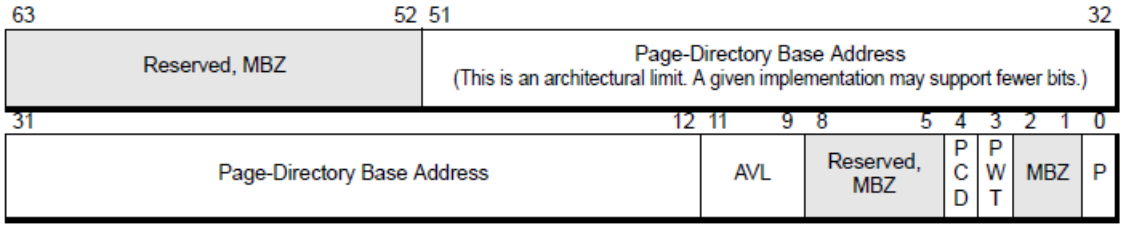


Figure 5-10. 4-Kbyte PDPE—PAE Paging Legacy-Mode

Burada girişlerin 64 bit olduğunu görüyorsunuz. İşte bu 64 bitlik girişlerin içerisinde 4K'ya hizalanmış biçimde 40 bitlik dizin tablolarının adresleri tutulmaktadır. Dizin tablosu girişlerinin formatı da şöyledir:

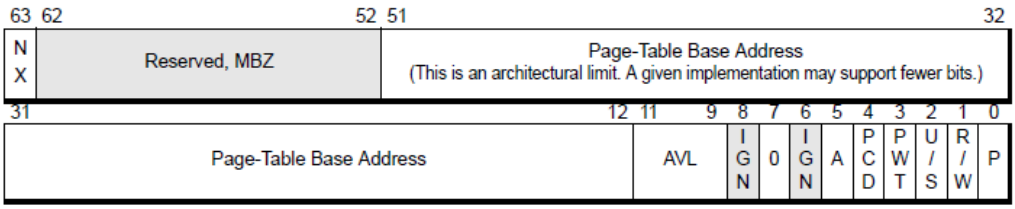


Figure 5-11. 4-Kbyte PDE—PAE Paging Legacy-Mode

Burada da yine sayfa tablolarının yerleri için 40 bit ayrıldığını görüyorsunuz. Sayfa tobloları da 4K'ya hizalanmış olmak zorundadır. Tabii bu girişler de 64 bit olduğuna göre dizin tablolarında toplam 512 giriş bulunacaktır. Sayfa tablosu girişlerinin formatı da şöyledir:

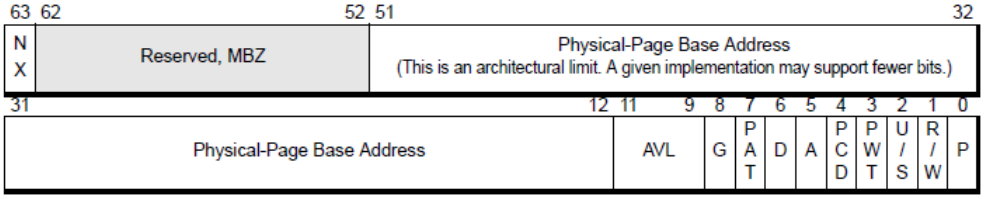
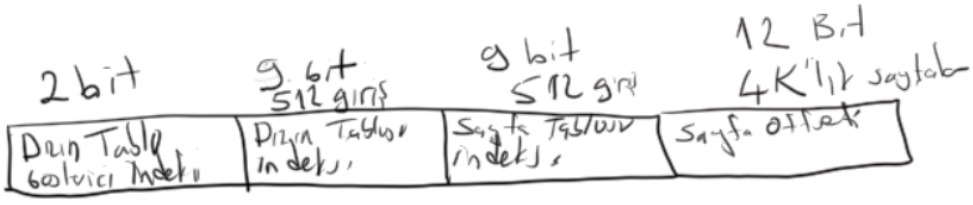


Figure 5-12. 4-Kbyte PTE—PAE Paging Legacy-Mode

Burada da sayfa offseti olarak yine 40 bitin bulunduğuna dikkat ediniz. Sayfalar yine 4K'ya hizalanmış olarak bulunmak zorundadır.

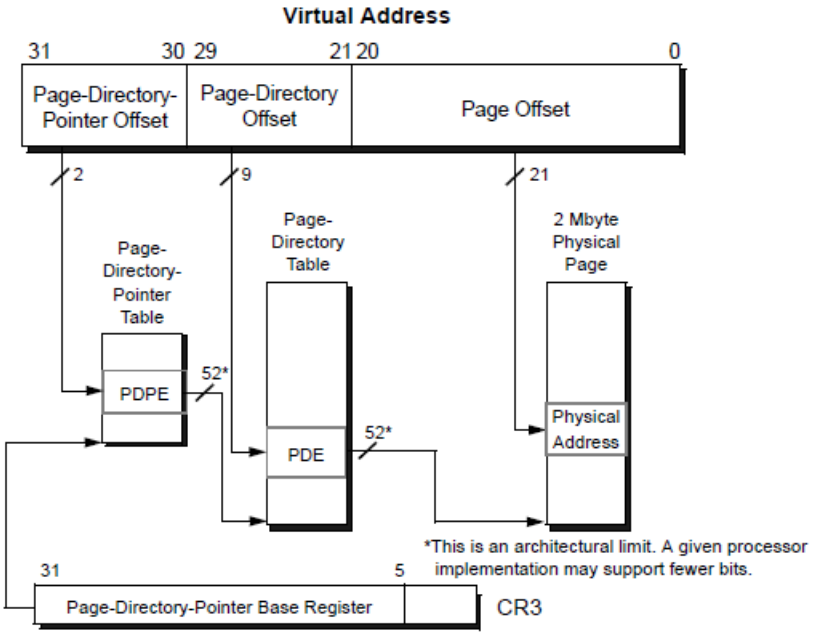
Bu modda doğrusal adresi oluşturan bitlerin sayıları şöyledir:



Pekiyi 52 bit fiziksel adres nasıl oluşturulmaktadır. Sayfa tablolarının girişlerinde sayfaların yerleri içinm 4K'ya hizalanmış 40 bitlik girişler bulunmaktadır. Bu durumda $2^{40}*2^{12} = 2^{52} = 4$ PB'lik bir fiziksel adres kullanılabilmektedir.

32 Bit Çalışma Modunda 2 MB'lik Sayfalarla 4 PB Fiziksel Belleğin Kullanımı

Bu modda sayfalar 2 MB uzunluğundadır. Toplam 52 bit fiziksel RAM adreslenebilmektedir. Yine doğrusal adresler 32 bit uzunluğundadır. Bu modda doğrusal adres üç kısma ayrılır: Dizin Tablosu Gösterici Offseti, Dizin Tablosu İndeksi ve Sayfa Offseti



Dizin Tablosu Gösterisi Offset'i yine 2 bitten oluşmaktadır. Bu iki bit toplam 4 tane dizin tablosunu gösterir.

Dizin tabloları toplam 512 girişe sahiptir. Yani doğrusal adresteki diizn indeksi 9 bit uzunluğundadır. Bu modda sayfa tabloları yoktur. Dizin tablosundaki her giriş 2 MB'ye hizalanmıştır. Dizin Tablosu Gösterici Tablosu girişi şu formattadır:

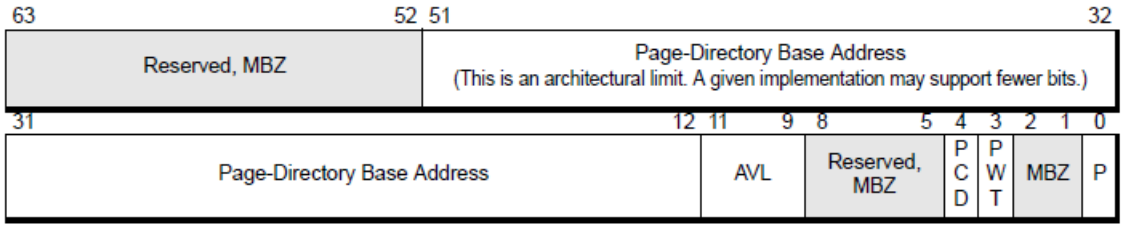


Figure 5-14. 2-Mbyte PDPE—PAE Paging Legacy-Mode

Dizin tablosu girişinin formatı ise şöyledir:

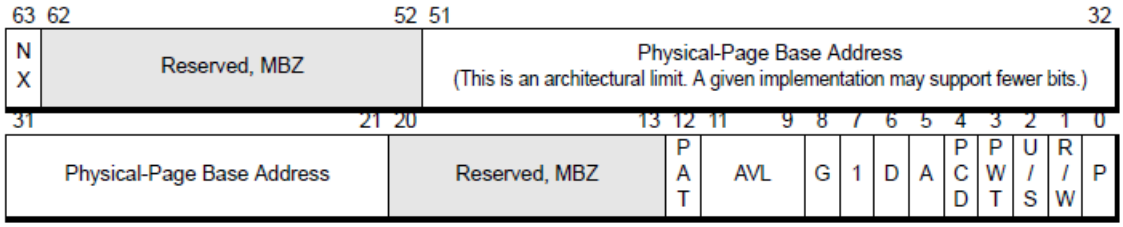


Figure 5-15. 2-Mbyte PDE—PAE Paging Legacy-Mode

Bu modda yine toplam erişilebilecek fiziksel RAM 4 PB yani 52 bittir. Doğrusal adres ise 32 bitten oluşur. Sayfalar 2 MB uzunluğundadır. Dizin girişlerinde sayfaların fiziksel adresleri için 31 bit kullanıldığına dikkat ediniz. Sayfaların 2 MB'ye hizalı olması gerekmektedir.

64 Bit Çalışma Modunda 4 K'lık Sayfalarla 4 PB Fiziksel Belleğin Kullanımı

Bugün kullandığımız 64 bit Windows, Linux ve MAC OS X sistemlerinde sayfalama mekanizmasında bu mod kullanılmakradır. Bu modda doğrusal adres 64 bit değildir, 48 bittir. Yani bir prosesin sanal bellek alanı en fazla 256 TB olabilir. Tabii Intel ve AMD bu limiti ileride gerekirse geçmişe doğru uyumu bozmadan yükseltmeyi hedeflemektedir. Burada 48 bitlik doğrusal adresin kullanılmayan 16 biti kononik biçimde olmak zorundadır. Aksi halde işlemci #GP (General Protection Falt) kesmesi oluşturur. Kanonik biçim nedir? 48'inci bitin en yüksek anlamlı biti (47'inci biti) 1 ise geri kalan yüksek anlamlı 16 bitin 1 olması, 0 ise geri kalan yüksek anlamlı 16 bitin 0 olması durumuna kanonik biçim denilmektedir. (Zaten 128'inci TB'ye kadar tüm yüksek anlamlı bitler 0 olmaktadır.)

Bu modda 48 bitlik doğrusal adres 5 kısma ayrılmaktadır:

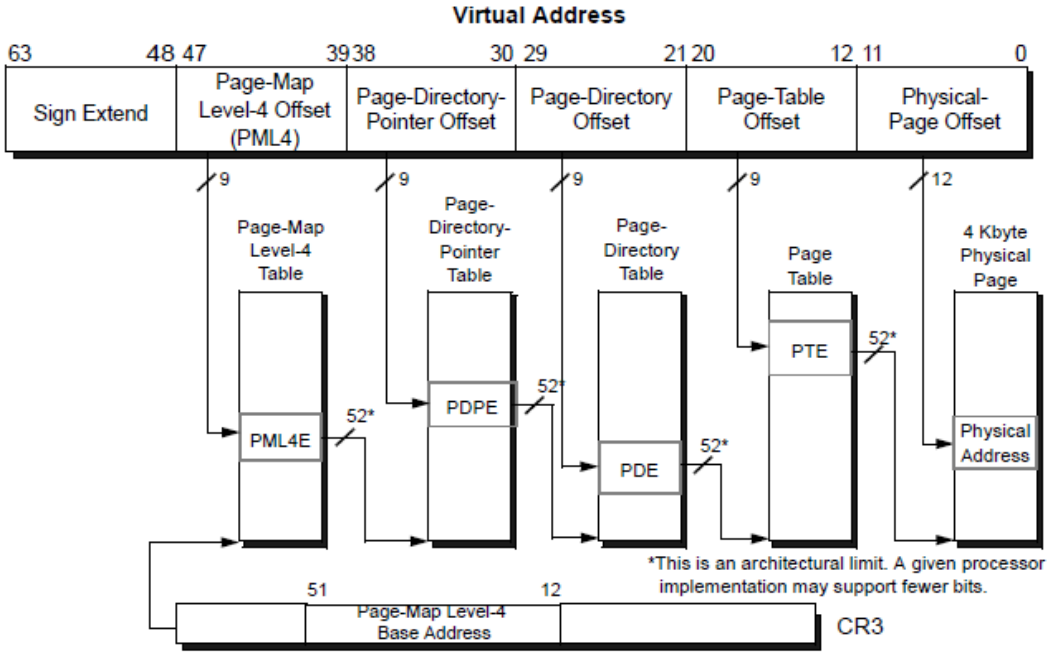


Figure 5-17. 4-Kbyte Page Translation—Long Mode

Doğrusal adresin yüksek anlamlı 9 biti (Tabii 48 bit üzerinden) "Sayfa Haritası Düzey 4 Tablosu (Page-Map Level-4 Offset)" denilen bir tabloda indeks belirtir. Bu tablodaki girişlerin formatı şöyledir:

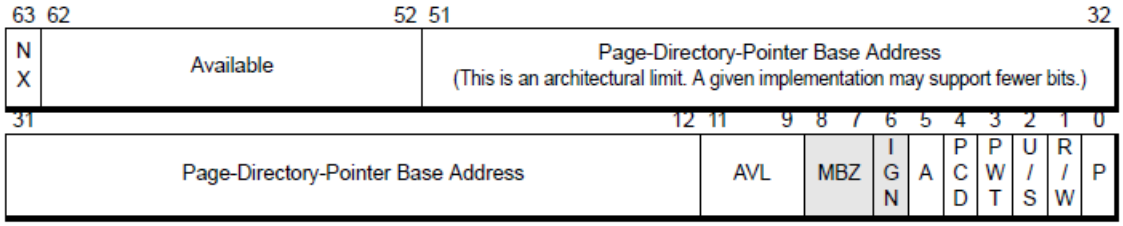


Figure 5-18. 4-Kbyte PML4E—Long Mode

Sayfa Haritası Düzey 4 Tablosu her biri 8 byte olan 512 girişten oluşmaktadır. Her girişte toplam 40 bitlik bir alanda "Sayfa Dizini Gösterici Tablo"larının fiziksel adresleri vardır. Sayfa Dizini Gösterici Tabloları 4K'ya hizalanmış olmak zorundadır ($2^{40}*2^{12} = 2^{52}$). Doğrusal adresin yine 9 bitlik ikinci bölümü "Sayfa Dizini Gösterici Tabloları"nda bir indeks belirtir. Sayfa Dizini Gösterici Tablolarının girişleri aşağıdaki gibidir:

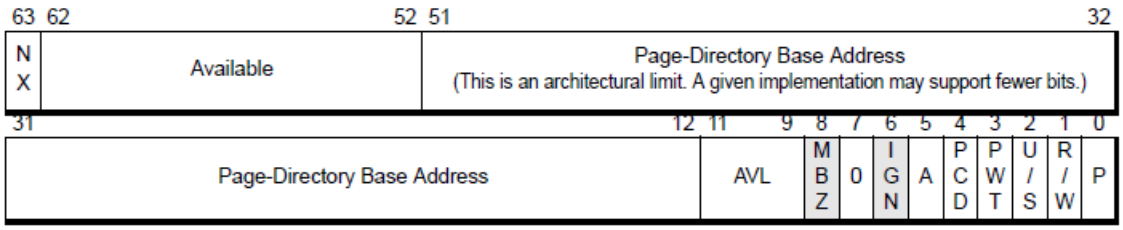


Figure 5-19. 4-Kbyte PDPE—Long Mode

Burada da benzer biçimde 40 bitlik alan içerisinde dizin tablolarının fiziksel adresleri tutulmaktadır. Dizin tabloları da 4K'ya hizalanmış olmak zorundadır. İşte doğrusal adresin 9 bitlik üçüncü bölümü dizin tablolarında bir indeks belirtmektedir. Dizin tabloları da her biri 8 byte olan toplam 512 girişten oluşmaktadır. Bu girişlerin formatları şöyledir:

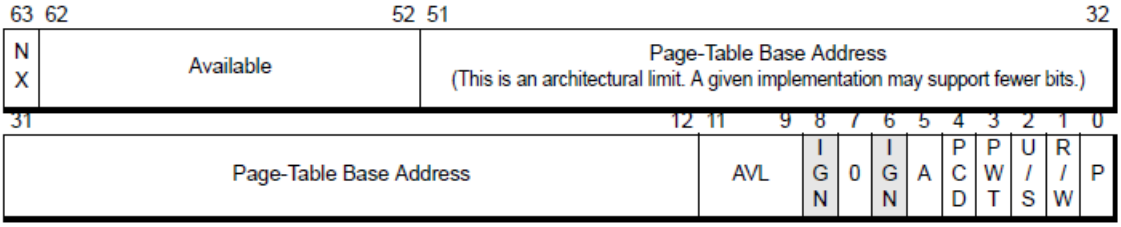


Figure 5-20. 4-Kbyte PDE—Long Mode

Burada da görüldüğü gibi yine 40 bit alan içerisinde sayfa tablolarının fiziksel adresi belirtilmektedir. Sayfa tabloları da yine 4K'ya hizalanmıştır. Doğrusal adresin 9 bitlik dördüncü bölümü yine sayfa tablolarında bir indeks belirtir. Sayfa tablolarının girişleri de aşağıdaki biçimdedir:

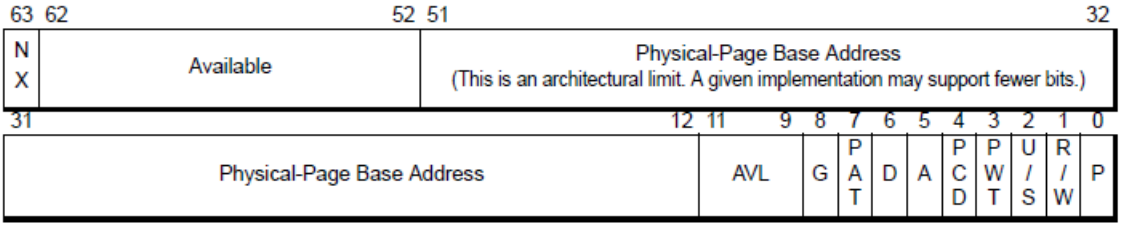


Figure 5-21. 4-Kbyte PTE—Long Mode

Burada da toplam 40 bitlik alan içerisinde sayfa tablolarının fiziksel adresleri tutulmaktadır. Sayfa tabloları da 4K'ya hizalanmış olmak zorundadır. Nihayet doğrusal adresin düşük anlamlı 12 biti sayfa içerisinde offset belirtir.

64 bit "long mode"da CR3 yazmacı da 64 bite yükseltilmiştir. CR3 yazmacının 40 biti Sayfa Haritası Düzey 4 Tablosunun fiziksel RAM'deki adresini göstermektedir.

64 Bit Çalışma Modunda 1 GB'lik Sayfalarla 4 PB Fiziksel Belleğin Kullanımı

Bu modda sayfalar 1 GB'ye hizalanmış biçimde bulunurlar. 48 bitlik doğrusal adresler 3 bölümden oluşmaktadır:

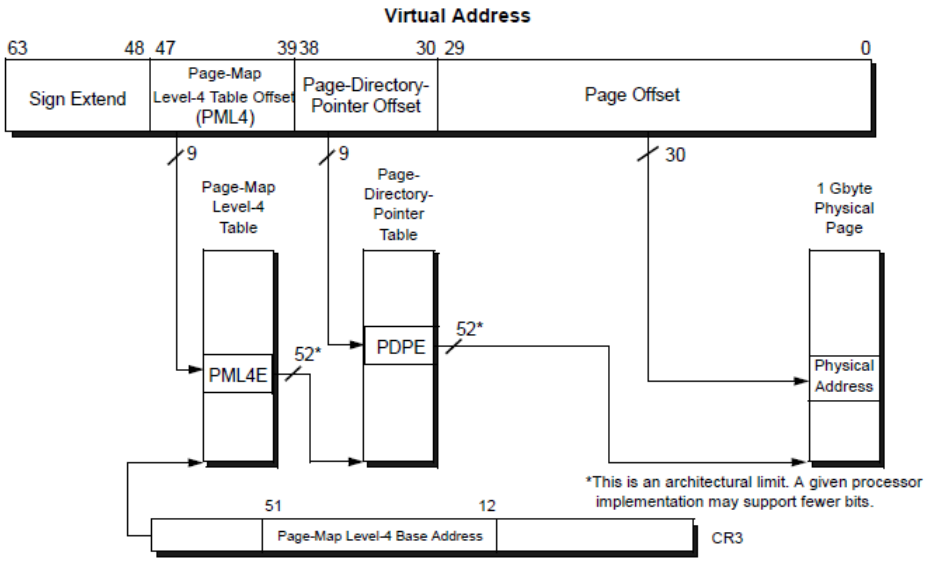


Figure 5-26. 1-Gbyte Page Translation—Long Mode

Doğrusal adresin yüksek anlamlı 9 biti "Sayfa Haritası Düzey 4 Tablosunda" bir offset belirtmektedir. Sayfa Haritası Düzey 4 Tablosu her biri 8 byte olan toplam 512 girişe sahiptir. Girişlerin formatı şöyledir:

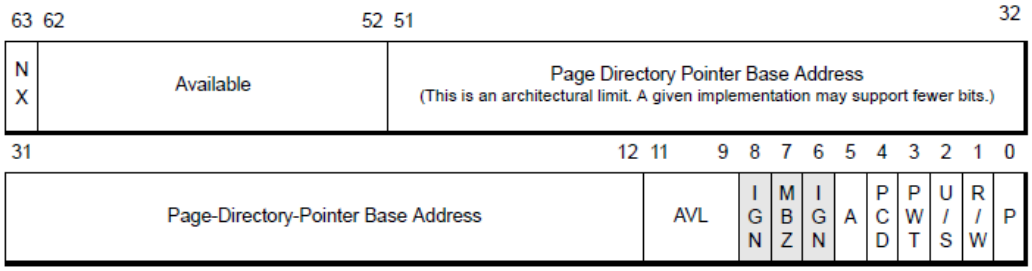


Figure 5-27. 1-Gbyte PML4E—Long Mode

Şekilden de görüldüğü gibi Sayfa Haritası Düzey 4 Tablosunun girişleri içerisindeki 40 bitlik alanlar Sayfa Dizini Gösterici Tablolarının RAM'deki başlangıç adreslerini göstermektedir. Sayfa Dizini Gösterici Tablolarının girişleri de şöyledir:

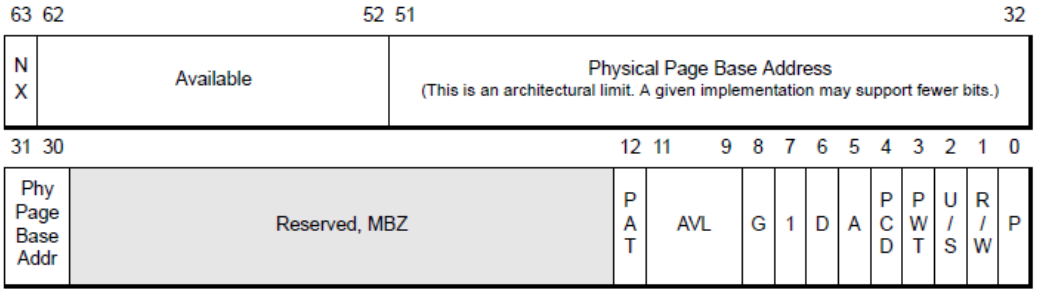


Figure 5-28. 1-Gbyte PDPE—Long Mode

Doğrusal adresin yüksek anlamlı ikinci bölümü de Sayfa Dizini Gösterici Tablosunda bir indeks belirtir. İşte Sayfa Dizini Gösteri Tablolarındaki girişler de doğrudan 1 GB'lik sayfaların fiziksel RAM'deki adreslerini belirtmektedir. Sayfaların 1 GB'ye hizalanmış olması gerekmektedir. Bu nedenle bu girişlerde sayfaların yerleri için 22 bit alan kullanılmıştır.

64 Bit Çalışma Modunda 2 MB'lik Sayfalarla 4 PB Fiziksel Belleğin Kullanımı

Bu modda doğrusal adres 4 bölümden oluşmakttadır:

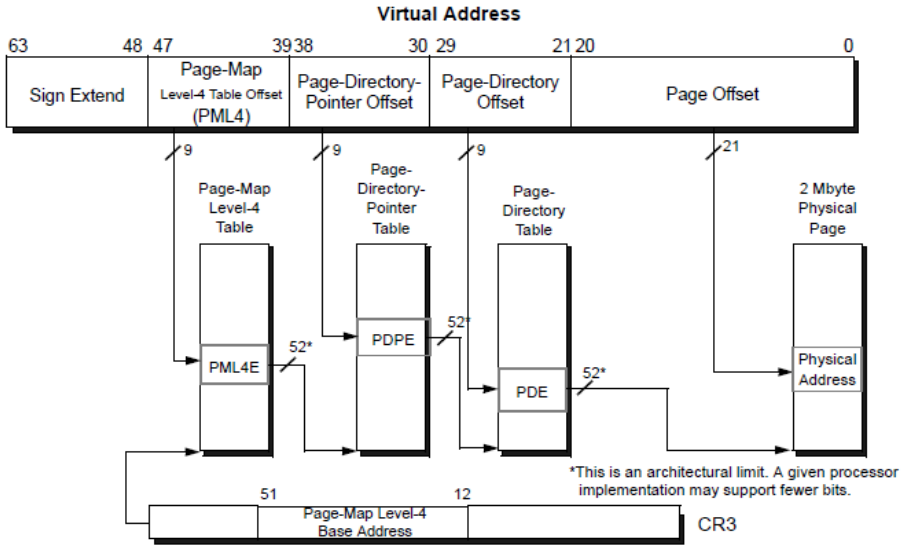


Figure 5-22. 2-Mbyte Page Translation—Long Mode

Yine doğrusal adresin yüksek anlamlı 9 biti Sayfa Haritası Düzey 4 Tablosunda bir indeks belirtir. Yüksek anlamlı ikinci 9 bit ise Sayfa Dizini Gösterici tablolarında bir indeks belirtir. Sonraki 9 biti ise Dizin

Tablosunda bir indez belirtmektedir. Nihayet düşük 21 bit sayfa içerisinde offset belirtir. Sayfa Haritası Düzey 4 Tablosuna ilişkin 8 byte'lık girişlerin formatı şöyledir:

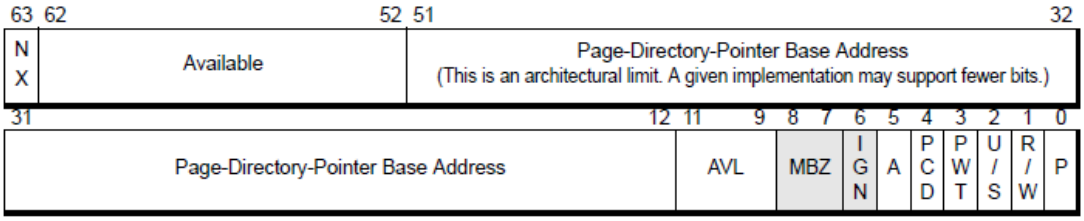


Figure 5-23. 2-Mbyte PML4E—Long Mode

Burada yine 40 bitlik bir fiiziksel adres alanı olduğunu görüyorsunuz. Sayfa Dizini Gösterici Tablolarının 8 byte'lık girişleri de aşağıdaki formattadır:

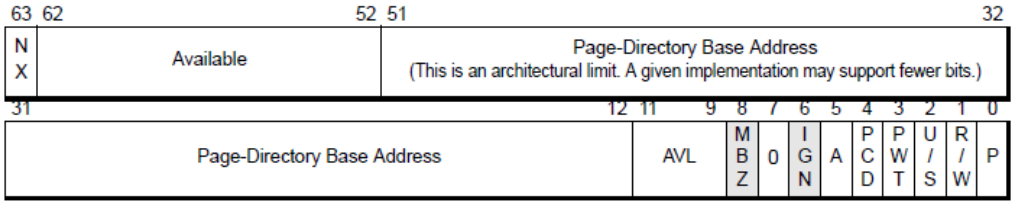


Figure 5-24. 2-Mbyte PDPE—Long Mode

Nihayet Dizin Tablosu girişleri de aşağıdaki gibidir:

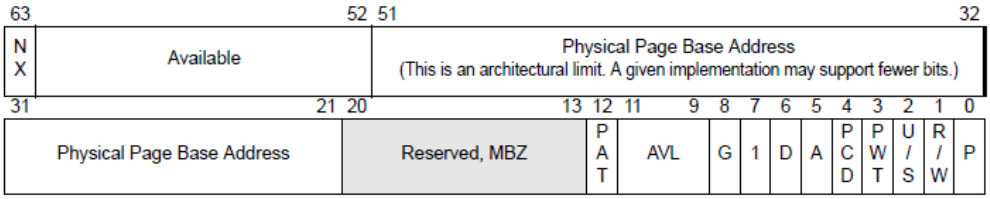
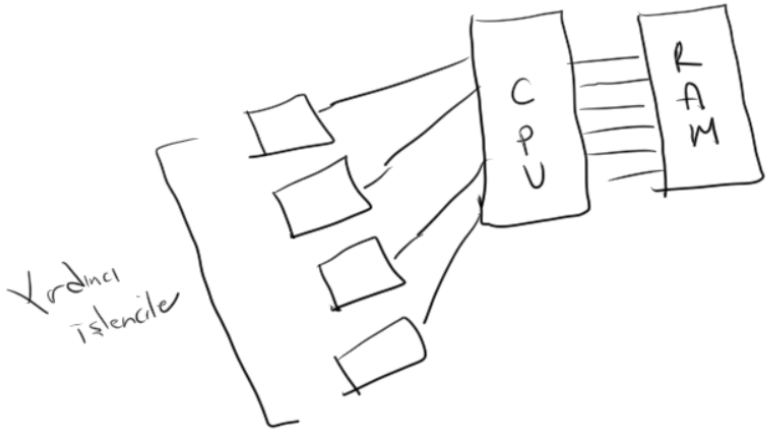


Figure 5-25. 2-Mbyte PDE—Long Mode

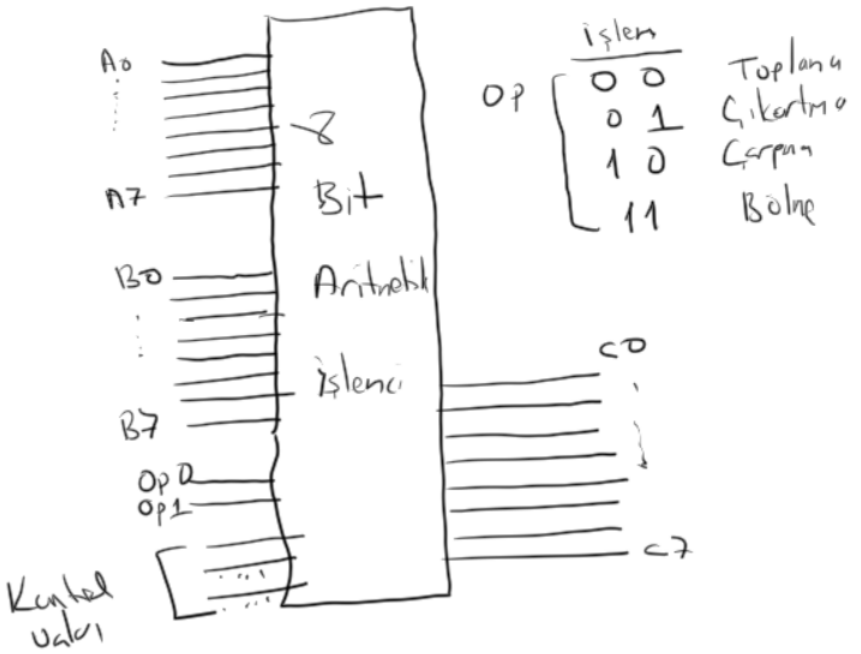
80X86 Mimarisinde Donanımsal IO İşlemleri

Programlanabilen elektronik birimlere işlemci denilmektedir. Bir bilgisayar mimarisinde ana işlemleri yapan merkezi bir işlemcinin (Central Processing Unit) yanı sıra yerel işlemleri yapan pek çok yardımcı işlemci (processor) de bulunabilmektedir. Merkezi işlemciyi başbakan olarak düşünürsek yardımcı işlemciler bakanlar gibi düşünülebilir. Yardımcı işlemciler merkezi işlemciyle elektriksel olarak bağlantı halindedir ve bunlar merkezi işlemci tarafından elektriksel düzeyde ikilik sistemde programlanmaktadır. O halde merkezi işlemci yalnızca ana bellekle (RAM ile) değil aynı zamanda yardımcı işlemcilerle de elektriksel olarak bağlantı halindedir.



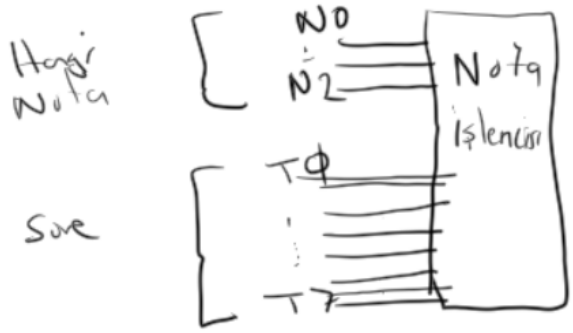
Tabii buradaki şekil gerçek bir bağlantı biçimine göre değil kavramsal olarak çizilmiştir. Merkezi işlemci RAM'deki herhangi bir byte onun adresini bilerek yani adresini elektriksel düzeyde ikilik sistemde RAM'e ileterek erişir. Benzer biçimde yardımcı işlemcilerin de birer adresi vardır. Geeleneksel olarak kişisel bilgisayarlarda bu adreslere "port numaraları" denilmektedir. Bu durumda merkezi işlemci belli bir yardımcı işlemciye komut gönderebilmek için onun port numarasını bilmek zorundadır.

Pekiyi bir yardımcı işlemcinin elektriksel olarak programlanması ne anlam ifade etmektedir? Yardımcı işlemciye ne yapması gerektiği +5V, 0V (tipik olarak) biçiminde ikilik sistemde elektriksel olarak iletilmektedir. Yardımcı işlemci işlemini yaptıktan sonra sonucu da merkezi işlemciye yine bu biçimde iletir. Örneğin 8 bit iki sayıyı dört işleme sokan bir yardımcı işlemci olduğunu düşünelim. Bu işlemcinin pek çok ucu vardır. Bu uçların çoğu merkezi işlemciye bağlanmaktadır.



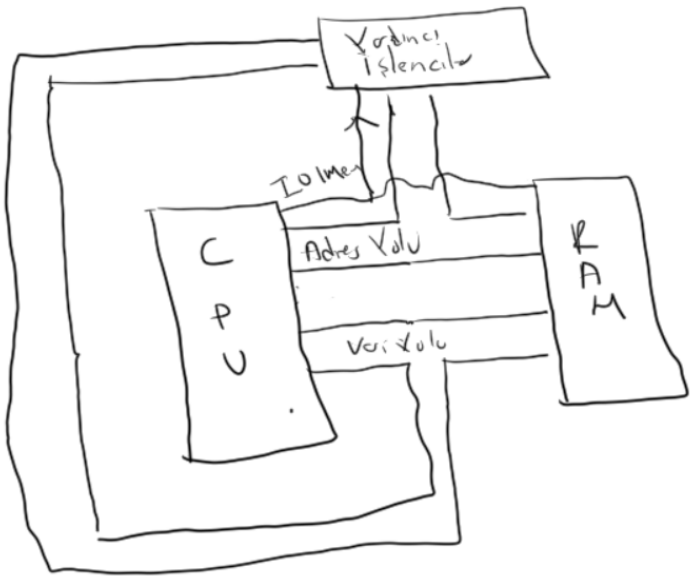
Burada ana işlemci iki operandı yardımcı işlemcinin sırasıyla A0-A7 ve B0-B7 uçlarına elektriksel işaret olarak iletebilir. Sonra ana işlemci hangi işlemin yapılacağını da yine Op0-Op1 uçlarından yardımcı işlemciye iletebilir. Buradaki yardımcı işlemci işlemini yaptıktan sonra sonucu elektriksel olarak C0-C7 uçlarına verecek biçimde tasarlanmıştır. O halde bu işlemden sonra ana işlemci bu uçları okuyarak sonucu yardımcı işlemciden alabilir. Burada gösterilmek istenen şey şudur: Merkezi işlemci teorik olarak yardımcı işlemciye ne yapacağını yani komutu, onun operadlarını ikilik sistemde elektriksel işaret olarak gönderebilmekte ve sonucu da yine elektiriksel işaret olarak alabilmektedir. İşte merkezi işlemciye bu biçimde çok sayıda yardımcı işlemci bağlıdır.

Şimdi de basit bir ses çıkartan işlemci düşünelim. Bu işlemci DO-Si ana notalarını belli bir süre çıkartsın. Notalar (Do-Re-Mi-Fa-Sol-La-Si) 3 uçla yardımcı işlemciye iletilebilirler. Onun ne kadar süre bu noratı çıkartacağı da 8 uçtan iletilecek olsun. Birim de 1/10 saniye olsun. O halde bu işlemcinin bizim için önemli olan uçları şöyle olabilir:



Tabii merkezi işlemci yardımcı işlemcileri kafasına göre progralamaz. Merkezi işlemci programcının kodunu çalıştırır. Programcı bu yardımcı işlemcilere komut gönderecek makine komutlarını oluşturur. Merkezi işlemci de komutları böyle gönderir. Yani aslında yardımcı işlemcileri de programcı programlamış olur.

Pekiyi merkezi işlemci ile yardımcı işlemciler arasındaki bağlantı biraz daha gerçekçi bir biçimde nasıl ifade edilebilir? İşte giriş konularından da anımsanacağı gibi merkezi işlemcinin üç önemli uç grubu vardır: Adres Yolu (Adres Uçları), Veri Yolu (Veri Uçları) ve Kontrol Yolu (Kontrol Uçları). Gerçekten de mekezi işlemcinin adres uçları yardımcı işlemciyi seçmek için, veri uçları da onlara komut ve bilgi gönderip onlardan bilgi almak için kullanılmaktadır. Anımsanacağı gibi bu uçlar aynı zamanda RAM'e bilgi yazmak ve RAM'den bilgi almak amaçlı da kullanılmaktadır.

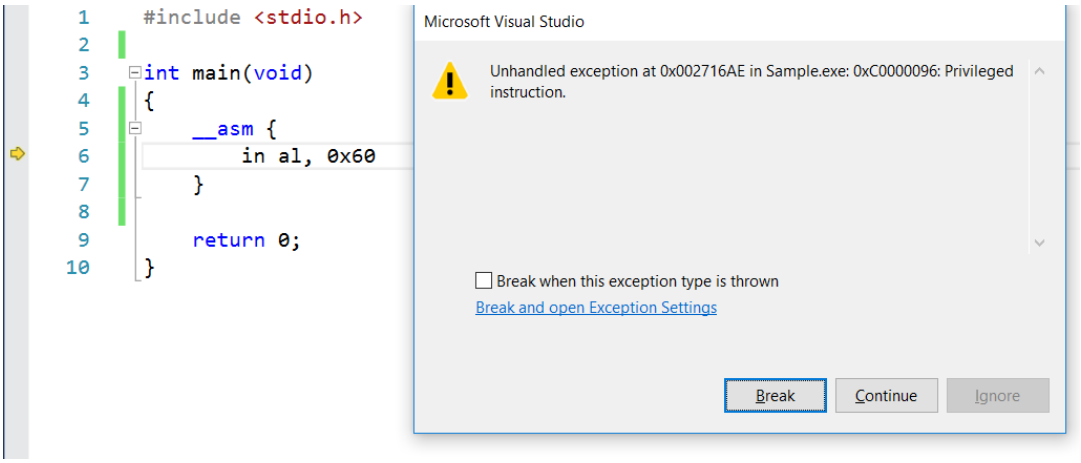


Pekiyi merkezi işlemci ile RAM arasında ve yardımcı işlemciler arasında adres uçları ve veri uçları ile bağlantı sağlandığına göre gönderilen bilginin yardımcı işlemciye mi yoksa RAM'e mi gönderildiği nasıl belirlenmektedir? İşte bunun için merkezi işlemcinin bir ya da iki kontrol ucu bulunur. Bu kontrol IO/Mem gibi bir isimle etiketlenmektedir. Örneğin bu uç 1 ise bu elektriksel işaretler yardımcı işlemcilere 0 ise RAM'e gönderilir. Ya da başka bir deyişle RAM bu uç 1 ise bu işaretlerle ilgilenmemektedir. Bu durumda bu işaretlerle yardımcı işlemci ilgilenmektedir. İşte yardımcı işlemcilere bilgi gönderip onlardan bilgi almak için bu IO/Mem kontrol uçlarını uygun biçimde konumlandıran özel makine komutları vardır. Genellikle işlemcilerde bu komutlar IN ve OUT biçiminde isimlendrilirler.

Şimdi komut gönderilirken ve bilgi alınırken yardımcı işlemcilerin nasıl adreslendiği üzerinde duralım.

Nasıl belleğin her byte'ının bir fiziksel adresi varsa yardımcı işlemcilerin de onları diğerlerinden ayırmak için kullanılan port numaraları vardır. Bazı yardımcı işlemcilerin tek bir port numarası varken bazılarının birden fazla port numarası bulunabilmektedir. Hangi yardımcı işlemcilerin hangi port numaralarını ne amaçla kullandıkları PC mimarisine ilişkin dokümanlara başvurularak öğrenilebilir.

Intel mimarisinde IN ve OUT komutları öncelikli (privileged) komutlardır. Dolayısıyla herhangi bir prosesin bu komutları kullanıcı mdundan (CPL = 3) kullanması Genel Koruma Hatasına (General Protection Fault) yol açar. Örneğin:



OUT makine komutunun genel biçimleri şöyledir:

```
OUT imm8, AL
OUT imm8, AX
OUT imm8, EAX
OUT DX, AL
OUT DX, AX
OUT DX, EAX
```

Porta 1 byte göndermek için AL yazmacı, 2 byte göndermek için AX yazmacı ve 4 byte göndermek için EAX yazmacı kullanılmaktadır. Port numarası eğer bir byte sınır içerisindeyse (0-255) bu durumda port numarası doğrudan sayı olarak komutta belirtilebilir. Ancak bir byte'tan daha büyük port numaraları için DX yazmacı kullanılmalıdır. Örneğin:

```
MOV     AL, 0xFF
OUT     0x21, AL
```

ile 0x21 numaralı porta 0xFF değeri gönderilmektedir. Örneğin:

```
MOV     DX, 0x3F2
MOV     AX, 0x1234
OUTDX, AX
```

Burada 0x3F2 numaralı porta 16 bit olarak 0x1234 değeri gönderilmiştir.

Ayrıca DS:ESI ya da DS:RSI yazmaçlarının gösterdiği yerden porta aktarım yapan ve ESI ya da RSI yazmaçlarını komut genişliği kadar artıran OUTSB, OUTSW ve OUTSD komutları vardır.

Porttan bilgi almak için kullanılan IN komutunun da genel biçimi şöyledir:

```
IN AL, imm8
IN AX, imm8
IN EAX, imm8
```

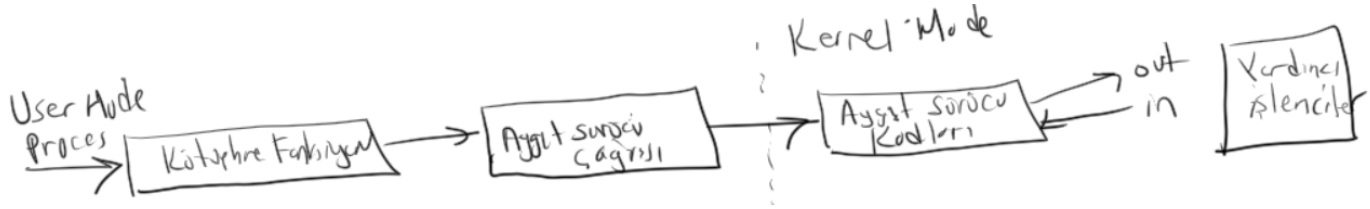
```
IN AL,DX
IN AX,DX
IN EAX,DX
```

Burada da benzer biçimde eğer port numarası 1 byte sınırlarında ise (0-255) biz port adresini sabit bir sayı ile ifade edebiliriz. Ancak iki byte sınırları içerisindeyse biz port numarasını DX yazmacında bulundurmak zorundayız.

Korumalı moddaki sistemlerde IN ve OUT komutları bunların yazılmış aygıt sürücüler yoluyla gerçekleştirilebilir. Örneğin Windows için "inpout32" isimli açık kaynak kodlu aygıt sürücüsü bu amaçla kullanılabilir. Linux sistemlerinde benzer biçimde /dev/port isimli aygıt sürücüsü zaten hazır olarak bulunmaktadır. Bu aygıt sürücüsü open fonksiyonuyla açılıp dosya göstericisi port numarasına konumlandırıldıktan sonra read ve write fonksiyonlarıyla okuma yazma yapılabilir. Ancak bu aygıt sürücünün sahibi root kullanıcısıdır ve diğer kullanıcılara herhangi bir hak verilmemiştir. Tabii dosyanın erişim hakları istenildiği gibi root kullanıcısı tarafından (sudo chmod komutu ile) değiştirilebilir.

Biz bilgisayarımızın genişleme yuvasına bir kart taktığımızda o kartın üzerinde de çeşitli yardımcı işlemciler bulunabilmektedir. Artık o kart üzerinde yardımcı işlemciler bizim bilgisayar donanımımızın bir parçası haline gelmiştir. İşte biz de o işlemcilere OUT komutuyla komutlar gönderip IN komutuyla onlardan komutlar alabiliriz.

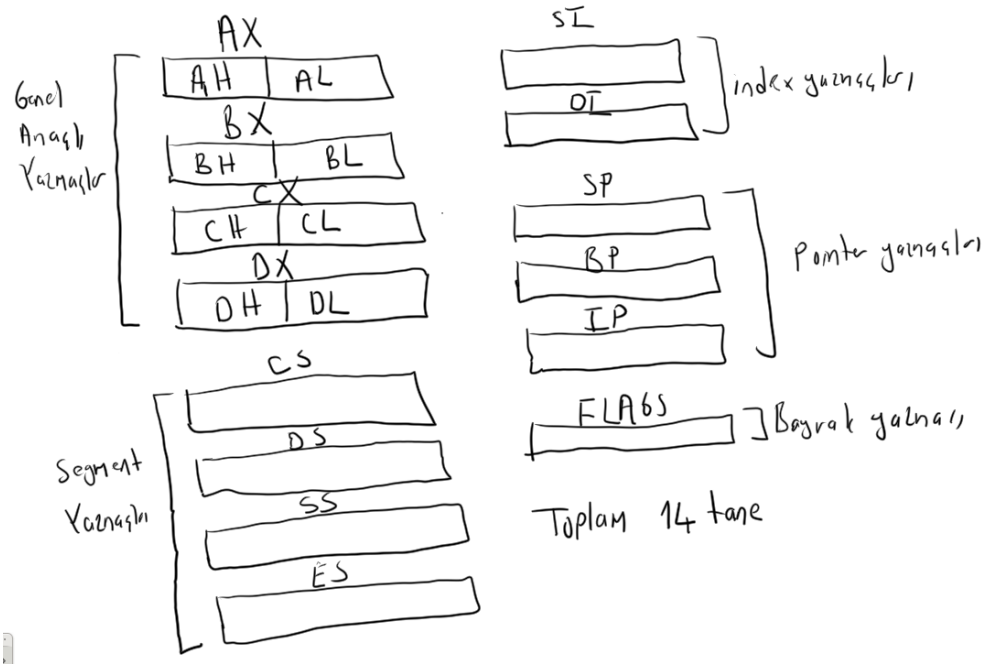
Genişleme yuvasına taktığımız kartlar nasıl işlevsellik kazanmaktadır. İşte kartı üreten firma burada işlemcileri programlayan kodları (yani IN ve OUT içeren kodları) sembolik makine dili ve C kullanarak yazıp bunu bir aygıt sürücü olarak sunmaktadır. Artık bu aygıt sürücünün fonksiyonları kullanıcı modundan (user mode) çağrılıp oradaki yardımcı işlemcilere çeşitli işler yaptırılabilmektedir. Tabii üretici firma bu fonksiyonları çağıran daha yüksek seviyeli kütüphaneeleri de kullanıcılara veriyor olabilir.



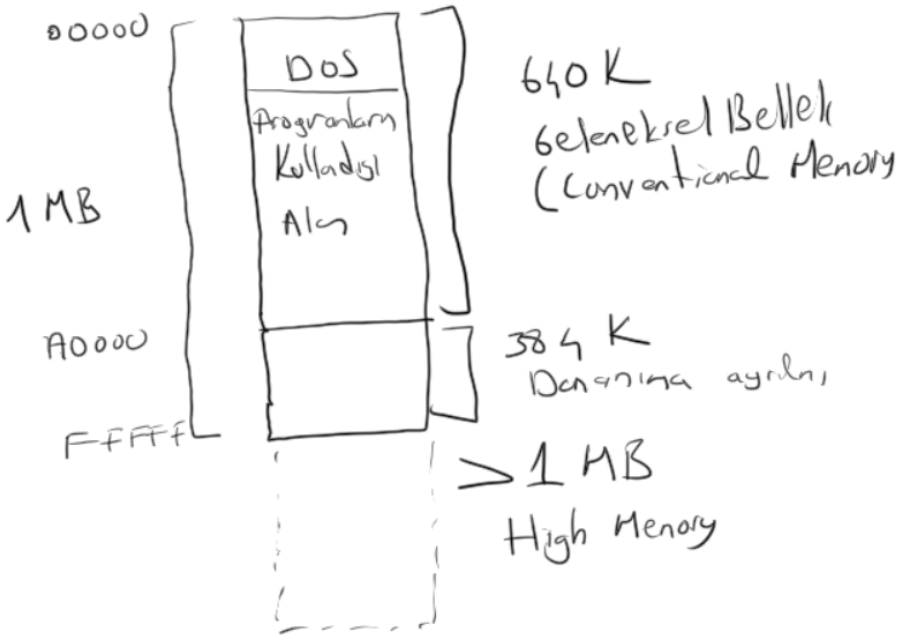
80X86 İşlemcilerinde 16 Bit Çalışma

Daha önceden de belirtildiği gibi Inetl x86 ailesi işlemciler reset edildiğinde çalışma "gerçek mod (real mode)" denilen 16 bit modunda başlamaktadır. Bu işlemci ailesi için ilk yazılmış olan DOS işletim sistemi de zaten gerçek mod çalışmaya göre tasarlanmıştır. X86 ailesine 32 bit mod eklendiğinde 32 bit mod programlarla 16 bit DOS programlarının zaman paylaşımlı çalışmasını mümkün hale getirmek için V86 modu (V86 mode) denilen bir mod da mimariye eklenmişti. 64 bit long modda 16 bit programlar çalıştırılamamaktadır. Ancak 32 bit programlar çalıştırılabilmektedir.

16 bit modda yazmaçlar 16 bittir:



16 bit modda segment yazmaçları selektör görevinde değildir. Bu modda işlemcinin adresleyebildiği bellek miktarı 1 MB'dir. Zaten ilk PC'lerde bu 1 MB'nin yüksek anlamlı 384K'sı bellek tabanlı IO işlemleri için donanım birimlerine ayrılmıştı. (Örneğin 1 MB'nin son 64K'sında EPROM alanı bulunmaktadır). Böylece geri kalan 640K belleğe daha sonraları "geleneksel bellek (conventional memory)" denilmiştir. Bu nedenden dolayı DOS sistemleri en fazla 640K kullanabiliyordu:



16 bit modda bellek operandlarının 16 bitlik kısmına offset denilmekteydi. Köşeli parantezler içerisinde ancak 16 bit sabit değer yazılabiliyordu. Zaten indeks yazmaçları da 16 bitti. Örneğin:

```
MOV SI, 0x1234
MOV AX, [SI]
```

Offset 16 bit olduğu için maksimum 64K2lık yer değiştirme sağlayabilmektedir. 1 MB adres alanındaki

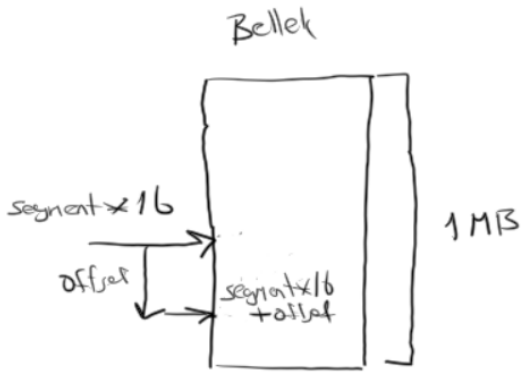
hedef adres segment değerinin 16 ile çarpılıp offset ile toplanmasıyla elde edilmektedir. (16'lık sistemde 16 ile çarpma sayının sonuna bir tane sıfır ekleme ile eşdeğerdir.) Bu modda köşeli parantezler içerisine yazılabilecek yazmaç kombinasyonları da şöyleydi:

```
[disp16]
[BX]
[BX + disp8]
[BX + disp16]
[SI]
[SI + disp8]
[SI + disp16]
[DI]
[DI + disp8]
[DI + disp16]
[BP]
[BP + disp8]
[BP + disp16]
[BX + SI]
[BX + SI + disp8]
[BX + DI + disp8]
[BX + SI + disp16]
[BX + DI + disp16]
[BP + SI]
[BP + SI + disp8]
[BP + DI + disp16]
```

Her bellek operandının ilişkin olduğu bir segment yazmacı vardır. Eğer köşeli parantez içerisinde BP yazmacı varsa o bellek operandının default segment yazmacı SS'dir. Köşeli parantez içerisinde BP yazmacı yoksa o bellek operandının default segment yazmacı ise DS'dir. Aşağıdaki tabloda bunu daha açık gösterebiliriz:

Bellek Operandı	**Default Segment Yazmacı**
[disp16]	DS
[BX]	DS
[BX + disp8]	DS
[BX + disp16]	DS
[SI]	DS
[SI + disp8]	DS
[SI + disp16]	DS
[DI]	DS
[DI + disp8]	DS
[DI + disp16]	DS
[BP]	SS
[BP + disp8]	SS
[BP + disp16]	SS
[BX + SI]	DS
[BX + SI + disp8]	DS
[BX + DI + disp8]	DS
[BX + SI + disp16]	DS
[BX + DI + disp16]	DS
[BP + SI]	SS
[BP + SI + disp8]	SS
[BP + DI + disp16]	SS

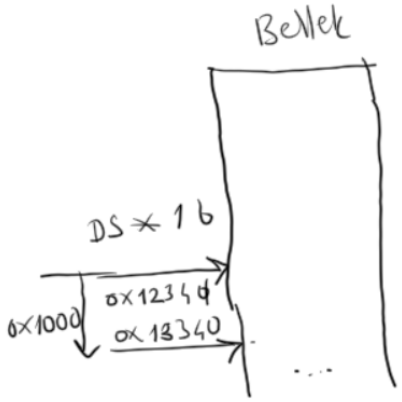
Segment 16 ile çarpıldığında hex sistemde sonu 0 olan bir değer elde edilir. İşte bu değer offset ile toplandığında gerçek etkin adres bulunur.



Örneğin DS'nin değeri 0x1234 olsun:

```
MOV SI, 0x1000
MOV AX, [SI]
```

Burada segmenti 16 ile çarptığımızda 0x12340 değeri elde edilir. Buna da 0x1000 toplanırsa sonuç 0x13340 olur.



16 bit sistemde segment sabit tutulduğunda offset değişimi ile en fazla 64K'lık bir alana erişilebilmektedir.

Gerçek modda hiçbir koruma mekanizması yoktur. Dolayısıyla bu modda her türlü işlem yapılabilmektedir. Örneğin bu modda IO portlarına erişilebilir. Kesme vektörü kancalanabilir. Bir gösterici ile başka proseslerin bellek alanlarına erişilerek orası bozulabilir.

Gerçek modda ve V86 modunda default çalışma biçimi 16 bittir. Zaten bu mod 1978 yılında tasarlanmış olan 8086 işlemcisini taklit etmektedir. Ancak ne olursa olsun bu modda biz istersek 32 bit yazmaçları kullanarak 32 bit işlemler yapabiliriz. Ancak bu durumda komutların başına 0x66 ve/veya 0x67 önekleri gelmektedir. Bu durum geçmişe doğru uyumu bozmamaktadır. Çünkü bu önekler 8086 işlemcisinde tanımlı değildir. Ancak gerçek modda ve V86 modunda (hatta 32 bit modda) 64 bit yazmaçlar kullanılamamaktadır. 64 bit yazmaçlar yalnızca 64 bit mod olan "long mode"da kullanılabilmektedir.

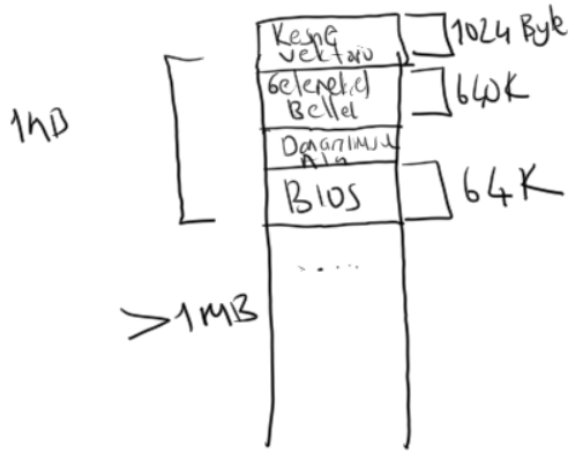
Intel Tabanlı Kişisel Bilgisayar Mimarisinde Reset İşlemi

Daha önceden de belirtildiği gibi her işlemci reset edildiğinde çalışma belli bir adresten başlamaktadır. İşte Intel'in (tabii AMD'nin de) X86 işlemcileri reset edildiğinde çalışma gerçek moddan ve CS: FFFF, IP: 0000 adresinden başlamaktadır. Bu da 1 MB bellek alanının son 16 byte'ına karşılık gelir (FFFF0). İşte burada bir kodun bulunması gerekmektedir. Reset vektöründeki kod RAM gibi ucucu bir bellekte olamaz. Çünkü reset edildiğinde tüm bellek sıfırlanmaktadır. PC'lerde 1 MB'lik adres alanının son 64K'sı fiziksel RAM'e değil

silinmeyen yani kalıcı bir ROM belleğe yönlendirilmiştir. Eskiden bu ROM bellek EPROM tekniği ile gerçekleştiriliyordu. Daha sonraları EEPROM bellekler bu amaçla kullanmaya başlandı. Bugün artık bu sayede buradaki programları doğrudan güncelleyebiliyoruz.

Bir aygıtın kalıcı belleğine üretici firma tarafından yerleştirilmiş olan kodlara ve verilere "firmware" denilmektedir. İşte PC'lerin de ilk 1 MB'nin son 64 K'sında bir "firmware" bulunmaktadır. PC terminolojisinde bu "firmware"e "ROM BIOS" denilmektedir. BIOS terimi "Basic Input Output System" sözcüklerindne kısaltmadır. BIOS'un içerisinde şu tür kodlar ve veriler vardır:

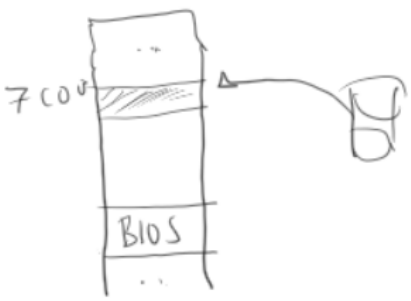
- Kesme vektörünün düzenlenmesi
- POST (Power On Self Test) kodları
- ROM Bootstrap kod
- Gerçek moddaki kesme kodları
- Temel donanımsal işlemleri yapan kodlar (bu kodlar zaten kesme kodları biçimindedir.)



Reset vektöründeki ilk kodlar genellikle önce kesme vektörünü düzenlemektedir. Kesme kodları BIOS içerisindedir. Ancak kesme oluştuğunda o kodlara yönlendirme yapan vektör henüz reset sırasında ilkdeğerlerini almamıştır. Bu durumda gelen kesmeler makinenin çökmesine yol açabilirler. O halde reset işlemi sırasında tüm donanım kesmelerinin de işin başında engellenmiş olması gerekir. İşte reset kodları önce kesme vektöründeki adresleri BIOS içerisindeki kodları kesme kodlarını gösterecek biçimde ayarlamaktadır.

POST işlemi sırasında bilgisayar sistemine hangi birimlerin bağlı olduğu belirlenir ve bunlara ilişkin bilgiler ilk 1024 byte'ın (kesme vektörünün) biraz ilerisine yerleştirilir. Yukarıdaki şekilde gösterilmeyen bu alana "BIOS Haberleşme Alanı (BIOS Communication Area)" denilmektedir.

POST işleminden sonra sıra işletim sisteminin yüklenmesi gelmiştir. İşte BIOS kodlarının bu kısmına "bootstrap" program denilmektedir. Klasik BIOS kodları tipik olarak ilk bootable aygıtın ilk sektörünü RAM'de 7C00 adresine okur ve oraya jump eder. Örneğin ilk bootable aygıt hard disk ise BIOS'taki "bootstrap" kod onun ilk sektörünü bellekte 7C00 adresine yükleyip kontrolü oraaya devredecektir.



Böylece reset sonrasında akış otomatik olarak ikincil bellekteki koda aktarılmış olur. Buradaki kod dolaylı olarak işletim sistemini yüklemektedir. Genellikle işletim sistemleri tek aşamada yüklenmez. Bootable aygıttaki ilk sektöre "boot sektör" denilmektedir. Kontrol boot sektördeki programa aktarıldığında buradaki program da işletim sisteminin asıl yükleyicisini yükler ve kendini oraya aktarır. Nihayet o program da asıl işletim sisteminin kendisini yükleyip oraya atlamaktadır. Örneğin Linux'un kaynak kodlarında bu bölüm "arch/i386/boot" dizininde bulunmaktadır:

Name	Size	Last modified (GMT)	Description
Parent directory		2009-05-27 19:05:25	
compressed/		2009-05-27 19:05:25	
tools/		2009-05-27 19:05:25	
Makefile	2833 bytes	2001-08-05 20:13:19	
bootsect.S	10626 bytes	2003-08-25 11:44:39	
install.sh	974 bytes	2001-09-14 21:04:06	
setup.S	26229 bytes	2002-08-03 00:39:42	
video.S	38962 bytes	2001-07-05 18:28:16	

Burada bootsect.S dosyası "GASM"de yazılmış olan ve ikincil belleğin boot sektörüne yerleştirilecek programdır. Buradaki program bazı ayarlamaları yaptıktan sonra setup.S dosyasını diskten belleğe yükleyerek oraya jump eder. Böylece artık setup.S dosyasında yazılmış olan sembolik makine dili programı çalışmaktadır. İşletim sisteminin aslıl büyük çekirdek kodları (kernel image) bu setup.S programı tarafından yüklenmektedir. İşte setup.S programı işletim sisteminin çekirdek kodlarını RAM'e yükledikten sonra artık oraya jump eder. Linux kaynak kodlarında bu jump edilen nokta "init/main.c" içerisindkei start_kernel fonksiyonudur.

```
352 asmlinkage void __init start_kernel(void)
353 {
354         char * command_line;
355         extern char saved_command_line[];
356 /*
357  * Interrupts are still disabled. Do necessary setups, then
358  * enable them
359  */
360         lock_kernel();
361         printk(linux_banner);
362         setup_arch(&command_line);
363         printk("Kernel command line: %s\n", saved_command_line);
364         parse_options(command_line);
365         trap_init();
366         init_IRQ();
367         sched_init();
368         softirq_init();
369         time_init();
370
371         /*
372          * HACK ALERT! This is early. We're enabling the console before
373          * we've done PCI setups etc, and console_init() must be aware of
374          * this. But we do want output early, in case something goes wrong.
375          */
376         console_init();
377 #ifdef CONFIG_MODULES
378         init_modules();
379 #endif
```

İşletim Sistemi Geliştirmesinde Derleme, Bağlama ve Boot Süreci

İşletim sistemi geliştirirken derleyicinin ve bağlayıcının ürettiği kodların meta-data bilgilerini içermemesi gerekir. Başka bir deyişle bu kodlar düz makine komutları ve düz veriler dışında hiçbir başlık kısmına sahip olmamalıdır. GCC ve NASM derleyicileri bu biçimde kod oluşturmayı mümkün hale getirmektedir. Ancak

Microsoft'un derleyicisinde bunu yapmaya yönelik seçenekler (command line switches) yoktur. Tabii aslında önce işletim sistemine bağımlı "object module" ve "executable" dosyalar elde edilip bunların içerisinden saf kod ve veriler de çekilebilir. Örneğin GNU'nun "binutils" temel paketi içerisindeki "objcopy" programı bunu yapabilmektedir. Ne olursa olsun işletim sistemi yazımı için genel olarak UNIX/Linux ortamları daha uygundur.

NASM'de başlık kısmı olmayan düz kod dosyası (flat binary) komut satırında -f bin seçeneği ile yaratılabilir. Örneğin:

```
nasm -f bin -o boot.bin boot.asm
```

Burada boot.asm dosyası derlenerek "boot.bin" düz binary kod dosyasına dönüştürülmüştür. Kod dosyaları içerisindeki ".data" ve ".bss" bölümleri sanki dosya belleğin tepesinden itibaren yüklenecekmiş gibi offset değeri almaktadır. Tabii programcı isterse hiç ".data" ya da ".bss" bölümlerini kullanmayabilir. Veriler için yer ayırma işlemlerini de ".text" bölümü içerisinde yapabilir.

Tabii işletim sistemi kodlamasında pek çok C dosyası ve sembolik makine dili dosyası ayrı ayrı derlenip birarada link işlemine sokulmaktadır. GNU ld bağlayıcısı ile link işlemi aşağıdaki komut satırı seçenekleriyle yapılabilir:

```
ld -o kernel.o -m elf_i386 -Ttext 0x100000 -N -e _KernelMain -Map kernel.map <object dosya listesi>
```

Burada "-o" çalıştırılabilir dosyanın ismini belirtmektedir. "-m elf_i386" 32 bit bir ELF formatı üretileceğini belirtir. "-T text 0x100000" seçeneği ise bağlayıcının programın nereye yükleneceği fikriyle kod üretmesini sağlamaktadır. Burada kodun ilk 1 MB'nin hemen başına yüklenmesi öngörülmüştür. -N seçeneği "bss" ve "data" alanının sayfaya hizalanmamasını sağlamaktadır (Bu seçenğin kullanılması zorunlu değildir.) "-Map kernel.map" seçeneği ise bağlayıcının bir "map" dosyası oluşturmasını sağlamak için kullanılmıştır. "map" dosyasında sembollerin hangi offsetlerde olduğu ve ne kadar uzunlukta olduğu gibi debug amaçlı faydalı bilgiler vermektedir. "-e _KernelMain" seçeneği "KernelMain" isimli fonksiyonun "entry point" olarak ele alınmasını sağlamaktadır. Aksi halde bağlayıcı _start isimli sembolü "entry point" olarak ele almaktadır. Bu sembol bulunamayınca da uyarı vermektedir.

Yukarıdaki link işlemi sonucunda üretilen dosya "flat binary" değildir. Bizim bu dosyasnın içerisindeki meta data bilgileerini atıp kod ve verileri alarak "flat binary" dosya elde etmemiz gerekir. Bu işlem klasik olarak "objcopy" isimli araç ile yapılmaktadır. "objcopy" ile "flat binary" dosya oluşturma işlemi aşağıdaki komut satırı seçenekleriyle gerçekleştirilebilir:

```
objcopy -R .note -R .comment -S -O kernel.o kernel.bin
```

Bu komutta ELF dosyası içerisindeki çeşitli meta-data alanlar atılmış ve "flat binary" bir dosya elde edilmiştir.

İşletim sistemini oluşturan boot kodu, setup kodu ve kernel dosyası uç uca getirilip tek bir "dosya haline dönüştürülebilir. Bu dosya da boot medyasının başından itibaren ona yazılabilir. Dosyaları uç uca eklemek "dd" shell komutuyla yapılabileceği gibi aşağıdaki gibi basit bir C programıyla da yapılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

#define BLOCK_SIZE      8192
#define FLOPPY_SIZE     1474560

int main(int argc, char *argv[])
{
    int i;
```

```
int oflag = 0;
int ch;
char *outfile = "kernelboot.bin";
FILE *fdest;
FILE *fsource;
char buf[BLOCK_SIZE];
size_t n;
long size, leftSize;
char *zeroBlock;

while ((ch = getopt(argc, argv, "o:")) != -1) {
    switch (ch) {
        case 'o':
            oflag = 1;
            outfile = optarg;
            break;
        default:
            fprintf(stderr, "invalid switch:%c\n", ch);
            exit(EXIT_FAILURE);
    }
}

if ((fdest = fopen(outfile, "wb")) == NULL) {
    fprintf(stderr, "cannot open file: %s\n", outfile);
    exit(EXIT_FAILURE);
}

for (i = optind; i < argc; ++i) {
    if ((fsource = fopen(argv[i], "rb")) == NULL) {
        fprintf(stderr, "cannot open file: %s\n", argv[i]);
        exit(EXIT_FAILURE);
    }

    while ((n = fread(buf, 1, BLOCK_SIZE, fsource)) > 0)
        if (!fwrite(buf, 1, n, fdest)) {
            fprintf(stderr, "cannot write file!\n");
            exit(EXIT_FAILURE);
        }
    if (ferror(fsource)) {
        fprintf(stderr, "cannot read file!\n");
        exit(EXIT_FAILURE);
    }
    fclose(fsource);
}

size = ftell(fdest);
leftSize = FLOPPY_SIZE - size;

if ((zeroBlock = (char *) calloc(BLOCK_SIZE, 1)) == NULL) {
    fprintf(stderr, "cannot allocate mmeory!\n");
    exit(EXIT_FAILURE);
}

while (leftSize > 0) {
    if (!fwrite(zeroBlock, 1, leftSize >= BLOCK_SIZE ? BLOCK_SIZE : leftSize, fdest)) {
        fprintf(stderr, "cannot write file!\n");
        exit(EXIT_FAILURE);
    }
    leftSize -= BLOCK_SIZE;
}

fclose(fdest);

return 0;
```

```
}
```

Test için gerçek makine kullanılabilir. Elde edilen dosya boot aygıtının sıfırıncı sektöründen itibaren oraya kopyalanabilir. (Örneğin bir flash belleğe bunu yazabiliriz. Sonra da flash belleği "bootable" aygıt haline getirebiliriz.) Test için en bait yollardan biri elde edilen bu imajı bir CDROM ISO dosyasına dönüştürme, sonra onu CDROM (ya da DVD ROM) içerisine yazarak makineyi bu sürücüden boot etmektir. Tabii bu işlemler oldukça zahmetlidir. Bu tür durumlarda sanalmakinelerden faydalanmak çok pratiktir. Ne de olsa sanal makine CD ROM ya da DVD ROM imajından kolaylıkla boot edilebilmektedir. ISO dosyası oluşturma işlemi için birtakım araçlardan faydalanılabilir. Fakat Linux sistemlerinde mkisofs komutu ile bu kolaylıkla yapılabilir. Komutun kullanımı aşağıdaki gibidir:

```
mkisofs -o kernelboot.iso -b kernelboot.bin .
```

Burada "-o kernelboot.iso" seçeneği oluşturulacak CDROM imajını belirtmektedir. "-b kernelboot.bin" dosyası ise ISO olarak yazılacak dosyayı belirtir.

Intel 80X86 Mimarisinde Makine Komutlarının Kodlanması

Bu bölümde Intel x86 serisi işlemcilerin makine komtlarının 2'lik sistemde nasıl ifade edildiği konusu ele alınacaktır. Buradaki bilgi aynı zamanda "disassembler" tarı programların yazımında da kullanılabilir. Maalesef Intel'in komut kodlaması (instruction encoding) oldukça karmaşık bir yapıdadır. Halbuki daha önceden de söz edildiği gibi RISC işlemcilerinde komutlar genellikle aynı uzunlukta olduğu için onlarda komut kodlaması da oldukça basittir. Aslında komut kodlaması bir bakımdan işlemcinin performansı üzerinde de etkili olabilmektedir. Anımsanacağı gibi Intel mimarisinde komutla hep aynı uzunlukta değildir. 1 byte olan makine komutları olduğu gibi 10 byte'tan uzun makine komutları bile vardır. Intel mimarisinde komut kodlamasını karışık hale getiren bir unsur da mimarinin geriye doğru uyumluluğudur. 16 bit, 32 bit ve 6 bit komutların birarada kullanılabiliyor olması birtakım öneklerin (prefix) kullanılmasını zorunlu hale getirmiştir. Burada biz önce 32 bit çalışmayı temel alacağız.

Komutların genel formatı Intel'in dokümanlarında aşağıdaki şekille temsil edilmiştir:

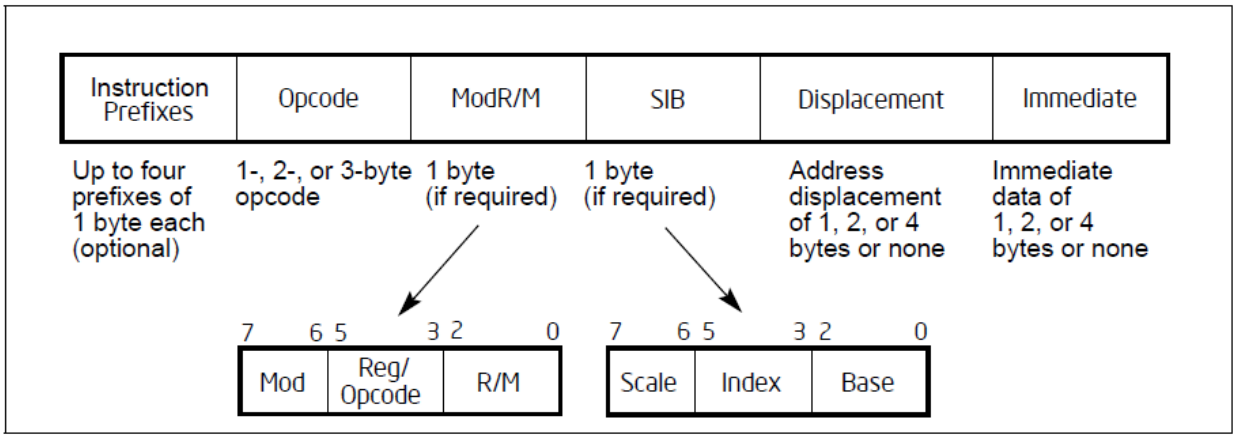


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Burada "Instruction Prefixes" komuttaki önekleri temsil etmektedir. Komutta hiç önek olmayabilir ya da dört kadar farklı önek bulunabilir. "Opcode" komutun ne komutu olduğunu anlatan teşhis byte'larıdır. Bunlar 1 ile 3 byte uzunlukta olabilirler. "Disaasmbler" programlar önce bu "Opcode" alanına bakarlar. "Mod R/M" eski dokümanlarda "Mod Reg R/M" biçiminde belirtiliyordu. Biz burada "Mod Reg R/M" kısaltmasını kullanacağız. Bu 1 byte komuttaki yazmacın hangi yazmaç olduğunu ve köşeli parantez içerisindeki ifadenin nasıl oluşturulduğunu belirtmektedir. "SIB (Scale Index Base)" 32 bit mimariyle gelmiş olan köşeli parantez içerisindeki çarpansal faktörü temsil etmektedir. "Displacement" köşeli parantez içerisindeki sabit değerlerdir. Nihayet "Immediate" de köşeli parantez içerisinde olmayan sabitleri temsil

etmektedir. Örneğin:

```
MOV EAX, [EBX + 100]
```

Burada kodlamanın "Opco" kısmı bize komutun MOV komutu olduğunu verir. Komutun ikinci operandının [EBX + displacement] biçiminde olduğu "Mod Reg R/M" ile kodlanmaktadır. 100 değeri de "Displaceement" ile kodlanır. Bu komut 32 bit modda bir "Prefix", "SIB" ve "Immediate" kısmı içermemektedir. Örneğin:

```
ADD dword [EAX + ECX * 2 + 100], 200
```

Burada kodlamanın "Opcode" kısmı bie komutun ADD komutu olduğunu verir. "Mod Reg R/M" köşeli parantez içerisinde [EAX + displacement + SIB] kalıbının olduğunu belirtir. ECX * 2 SIB'de kodlanmıştır. 200 ise "Immediate" olarak kodlanır.

Komutun "Opcode" Kısmı

Her ne kadar Intel resmi dokümanlarında açıklamayı bu biçimde yapmamış olsa da aslında komutun "Opcode" kısmında komutun teşhis edilmesini sağlayan ilk 6 bittir. Yani "Opcode" kısmı aşağıdaki gibi bir yapıya sahiptir. Ancak istisnaları da vardır:

```
xxxxxxdw
```

Buradaki "d" "direction" sözcüğünden gelmektedir. Bu "d" biti komutun "Mod Reg R/M" kısmındaki Reg'in (yani yazmacın) hedef operand olup olmadığını belirtmektedir. d = 0 ise "Mod Reg R/M"deki "Reg" hedef operand, d = 1 ise kaynak operand'tır. (Intel sisteminde hedef operandın sol komutun sol tarafında, AT&T sentaksında sağ tarafında olduğunu anımsayınız.) "w" biti "width" sözcüğünden kısaltmadır. Bu bit işlem genişliğini anlatır. 32 bit modda işlemler ya 32 bit genişliğinde ya da 8 bit genişliğinde yapılmaktadır. 32 bit modda 16 bit işlem yapılabilir. Ancak bunun için 0x66 öneki gerekmektedir. Tabii komutta bir yazmaç varsa (komutta yazmaç olmayabilir) buradaki "w" yanı zamanda yazmacın uzunluğunu da belirtir. Yani buradaki "w" "Mod Reg R/M" kısmındaki "Reg" ile belirtilen yazmacın 8 bit mi yoksa 32 bit mi olduğunu da belirtmektedir. Eğer d = 0 ise işlem 8 bit uzunluğundadır, d = 1 ise 32 bit uzunluğundadır.

Komutun "Mod Reg R/M" Kısmı

Komutun "Mod Reg R/M" kısmı komuttaki yazmaç ve bellek operandını betimlemektedir. 16 bit modda ve 32 bit modda "Mod Reg R/M" tabvloları farklıdır. (Çünkü 16 bit ve 32 bit adres operandlarında farklılık vardır.) "Mod Reg R/M" kısmındaki "Mod" 2 bit, "Reg" 3 bit ve "R/M" de 3 bittir:

Buradaki "Reg" kısmı 3 bitten oluştuğuna göre 8 seçeneğe sahiptir. İşte bu 8 seçeneğin her biri bir yazmacı belirtir. Komutun "Opcode" kısmındaki "w" bitine göre bu yazmaç 32 bitte ya 32 bitlik bir ya da 8 bitlik bir yazmaçtır. Bu kısmın "Mod" ve "R/M" si ya bellek operandını ya da ikinci yazmacı oluşturmak için kullanılmaktadır.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

"Mod Reg R/M"tablolarında R/M 100 olan satırlar komutta bir SIB kısmının olduğunu göstermektedir. Eğer komutta bir SIB kısmı varsa artık "Mod Reg R/M" byte'ının Mod ve R/M'si bir bellek ya da yazmaç belirtmez. Bellek operandı artık SIB byte'ından alınır. Komutun SIB (Scale Index Base) byte'ı köşeli parantez içerisinde iki yazmaç toplamını belirtir. Ancak bu toplamların bir tanesi çarpansal biçimdedir.

Komutun SIB Kısmı

Komutun SIB (Scale Index Base) kısmı Intel dokğmanlarında aşağıdaki şekille açıklanmıştır:

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

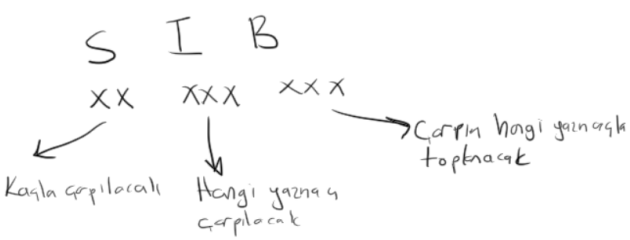
r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

SIB byte'ında S (Scale) yukarıdaki şekilde SS ile temsil edilmiştir. Bu yazmacın kaç ile çarpılacağını belirtir, 2 bit uzunluğundadır. (I) yukarıdaki şekilde Index ile temsil edilmiştir. Index 3 bitten oluşur, çarpım işlemine sokulacak yazmacı belirtir. B (Base) ise tablonun yukarısındaki 3 bitle temsil edilmiştir. Toplama işlemindeki çarpım biçiminde olmayan diğer yazmacı belirtir.



Komutun "Displacement" Kısmı

Komutun "Displacement" kısmı olmayabilir. eğer varsa her zaman SIB byte'ından sonra bulunmaktadır.

Tabii SIB byte'ı da olmayabilir. Bu durumda "Displacement" hemen "Mod Reg R/M" kısmından sonra bulunur. Komutun "Displacement" kısmı 1 byte, 2 byte ya da 4 byte olabilir. 32 bit modda bu kısım ya 1 byte ya da 4 byte olabilmektgedir.

Komutun "Immediate" Kısmı

Komutun "Immediate" kısmı komuttaki sabit değerdir. Örneğin:

```
MOV AL, 100
MOV EBX, 12345678
```

Tabii komutun "Immediate" kısmı işlem genişliği kadar olmalıdır. 32 bit modda öneksiz komuttaki sabit ya 8 bit ya da 32 bit olabilir. "Immediate" kısım komutun sonunda bulunur. Immediate içeren komutların "opcode" kısımları farklıdır. Yani örneğin:

```
XOR EAX, 0x12345678
```

komutu ile

```
XOR     EAX, [ECX]
```

komutundaki opcode kısımları farklıdır. Her komutun "mod reg r/m" biçiminde olmadığına dikkat ediniz.

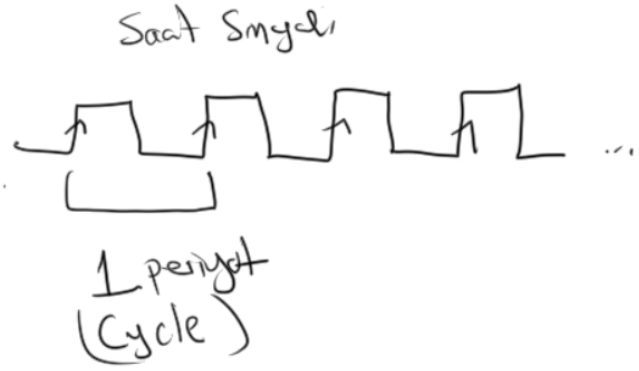
Komuttaki Önekler

Intel makine komutlarında "opcode" kısmının başında (yani komutun en başında) bri byte'tan 4 byte'a kadar birtakım her biri bir byte olan önekler gelebilmektedir. Örneğin 32 bit modda 16 bit yazmaçlar için 0x66 öneki, 16 bir bellek operandı için ise 0x67 öneki kullanılır. Yukarıdada belirtildiği gibi bu öneklerin ikisi birden komutun başında bulunabilmektedir. Bunun dışında XMM yazmaçları için ayrı bir önek, "long mode"da 64 bit yazmaçlar için ayrı önekler kullanılmaktadır. Biz burada bu öneklerin ayrıntılarını görmeyeceğiz. Ancak Intel'in dokümanlarında bu konu ayrıntılarıyla ele alınmaktadır. Kursumuzda 32 bir bir disassembler kodunun temeli "Src/x86/19-DisassemblerCodeBase" projesinden verilmiştir. Burada bazı eksikler söz konusudur ancak "decode" işleminin algoritmik temeli bu örnekte ele alınmıştır.

Intel İşlemcilerinde Komutların Çalışma Süreleri (Instruction Timing)

Mantık devrelerinde (logic circuits) girişler değişmedikten sonra çıkışların değişmesi için bir neden yoktur. Bir giriş değişir değişmez hemen bu durum çıkışa yansımaz. Giriş konumlandırıldıktan sonra belli periyotlarda değişim işlemi başlatılır. Bunun için kullanılan düzeneğe "saat sinyali (clock signal)" denilmektedir. Saat sinyali genellikle kare dalga biçiminde 5V ve 0V arasında gidip gelen periyodik bir sinyaldir.

Mantık devrelerinin girişleri değiştirildiğinde saat sinyali ile yeni işlemler başlatılmaktadır. İşlemler saat sinyalinin çıkan kenarıyla başlatılabileceği gibi düşen kenarıyla da başlatılabilir. İşte bu nedenle bir mantık devresinin hızı saat sinyalinin frekansına bağlıdır. CPU'lar da mantık deverelerinden oluşmuştur. Onlarda da komutların çalıştırılması için saat sinyali kullanılır. Örneğin 1 GZ hızında bir saat sinyali ile çalışan işlemcide bir sinyalin periyodu 1 / 1000000000 saniyedir (1 nano sn).



Mikroişlemcilerdeki saat sinyalinin bir periyoduna genellikle "cycle" ("saykıl" biçiminde okunuyor) denilmektedir. Komutların hepsi 1 cycle'da yapılmak zorunda değildir. Bir komut genel olarak tasarıma bağlı biçimde n cycle süre alabilmektedir. Örneğin bir mikroişlemcide çarpma işlemi 30 cycle sürüyorsa biz o mikroişlemciye 1 GZ hızında saat frekansı uyguluyorsak çarpma işlemi bu durumda 30 / 1000000000 sn hızında yapılmış olacaktır. Madem ki mikroişlemcilere değişik saat frekansları uygulanabilmektedir, o halde komutların ne kadar zaman aldığının saniye cinsinden belirtilmesi uygun olmaz. İşte bunun yerine komutların hızları "cycle sayısı" ile ifade edilmektedir. Tabii farklı firmalar (örneğin Intel ve AMD) aynı mikroişlemciyi farklı komutlar farklı cycle sayılarıyla yapılacak biçimde üretebilmektedir. Hatta aynı firmanın değişik modellerinde bu farklı olabilmektedir. O halde bir komutun cycle sayısı yalnızca işlemcinin türü ile değil model numarasıyla da ilgili olabilmektedir. Bu durumda iki aynı frekansta çalışan iki işlemcinin performanslarını kıyaslamak da o kadar kolay değildir. İşte performan kıyaslamasında en gerçekçi yollarından biri "benchmark" testleridir. Bu testlerde bazı algoritmalar çalıştırılarak ne kadar süre aldığına bakılır. Bu benchmark gerçek hayat hızı hakkında iyi bir fikir verebilmektedir.

İşlemci üreten firmalar işlemcilerine en fazla hangi frekansta saat sinyali uygulanabileceğini belirtmektedirler. Genellikle kullanıcıların ürünü satın alırken gördükleri hızlar bunlardır. Tabii bir işlemciye önggörülenden daha hızlı bir saat sinyali uygulamak firmanın garanti ettiği bir durum değildir. Buna "over clock" denilmektdir. "Over clock" altında bazı mikroişlemcilerin işini yapamaz hale geldiği bazılarının kalıcı olarak hasar gördüğü bazılarına ise hiçbir şey olmadığı bilinmektedir.

Eskiden işlemcilerin komutlarının kaç cycle olduğu kesin bir biçimde belirlenebiliyordu. Fakat artık modern işlemcilerde bunu belirlemek de o kadar kolay değildir. Bunun nedenleri şöyle açıklanabilir:

- Modern işlemcilerin cache mekanizması oldukça gelişmiştir. Dolayısıyla komutun ve operandın o anca cache'te olup olmaması komutun çalışma hızını değiştirebilmektgedir.

- Modern işlemciler komutlar birbirinden bağımsızsa daha hızlı işlem yapılabilmesi için onların sırasını değiştirebilmektedir (instruction reordering). Dolayısıyla aslında komutun kaç cycle süreceği bazen o komutun önündeki ve arkasındaki komutlara da bağlı ollabilmektedir.

- Modern işlemcilerde "jump prediction" özellikleri vardır. Bu da jump komutlarının cycle sayısı üzerinde etkili olabilmektedir.

- İntel mimarisindeki bazı önekler bazı durumlarda komutların cycle süreleri üzerinde etkili olabilmektedir.

Bugün kullandığımız Intel'in ya da AMD'nin x86 türevi yeni işlemcileri dışarıdan bir saat frekansı almakla birlikte bunu belli değerlerle çarparak (örneğin 4 ya da 8) asıl frekansı elde etmektedir. İşlemcinin çalışma frekansı bu çarpılmış olan değerdir. Buna taban frekans (base frequency) denilmektedir. İşletim sistemleri bu dışsal saat frekansını kimi zaman değiştirerek işlemcinin hızını duruma göre düşürüp yükseltebilmektedir. Örneğin dizüstü ve mobil aygıtlarda batarya azalmışsa işlemcinin çalışma frekansı düşürülmekte, eğer o anda aygıt fişe takılmışsa yükseltilmektedir. Frekansın düşürülü yükseltilmesi başka etmenlere de bağlı olabilmektedir. Yani özetle artık özellikle dizüstü ve mobil aygıtlarda işlemciye verilen saat frekanssı dinamik olarak sürekli değişebilmektedir.

Tüm bu anlatılanlar göz önüne alındığında Intel'in Pentium PRO serisinden sonra genel oalrka komutların cycle sürelerinin önceden tam olarak belirlenmesinin çok zor olduğu söylenebilir. Ancak yine de işlemci modleine göre öngörülere ilişkin çeşitli analizler üretici firma tarafından ve çeşitli şahıslar tarafından yapılmıştır. Biz de kırs dokğmanlarına "Agner Fog" tarafından yazılmış olan bir dokümanı dahil ettik (Doc\Others\x86-InstructionTiming.pdf).

80X86 İşlemcilerinde Bağlamsal Geçiş (Context Switch)

Bit thread'in çalışmasına ara verilip diğer bir thread'in kalınan yerden çalışmaya devam ettirilmesi sürecine "bağlamsal geçiş (contect switch)" denilmektedir. Bağlamsal geçiş "preemptive" işletim sistemlerinde donanım kesmeleriyle yapılmaktadır. PC mimarisinde tipik olarak "IRQ1 Timer" kesmesi bu amaçla kullanılmaktadır. Kesme oluştuğunda akış kernel moda geçerek işletim sisteminin kesme koduna aktarılır. Bağlamsal geçiş burada yapılmaktadır. Bağlamsal geçiş sırasında geçiş oluştuğu noktadaki tüm CPU yazmaçlarının (matematik işlemci de dahil olmak üzere) bir veri yapısına aktarılması ve yeni geçilecek thread'in yazmaçlarının CPU yazmaçlarına geri yüklenmesi gerekir. Genellikle thread'li işletim sistemleri her thread için bir veri yapısı (thread kontrol bloğu) oluşturmaktadır. Yazmaçlar bu veri yapısının içerinde saklanırlar. Linux sistemlerinde proses kontrol bloğu (yani task_struct yapısı) thread'ler için de aynı biçimde kullanılmaktadır. Yazmaç değerleri burada geçici olarak saklanır.

Preemptive olmayan ("cooperative" de denilmektedir) sistemlerde bağlamsal geçiş (ya da prosesler arası) geçiş kesme yoluyla değil açıkça bir fonksiyonun çağrılması yoluyla yani yazılımsal olarak yapılmaktadır. Gerçekten de işletim sistemi bir proseste IO olayı gerçekleştiğinde bağlamsal geçiş yaparak prosesin gereksiz beklemesini engellemektedir.

Çok threadli preemptive sistemlerde bile bir çeşit "cooperative" thread oluşturma mekanizması bulunabilmektedir. Bu mekanizmaya "fiber" denilmektedir. Fiber işlemleri işletim sisteminin çekirdeği tarafından bilinmez. Tamamen "user" modda gerçekleştirilebilmektedir. Windows işletimi de belli bir versiyonundan sonra Fiber kavramını API fonksiyonlarıyla destekler hale gelmiştir.

SON...