80X86 SEMBOLİK MAKİNA DİLİ

Bu döküman *Kaan Aslan*'ın 1998 yılında *C ve Sistem Programcıları Derneği*'nde vermiş olduğu *80X86 Sembolik Makina Dili* kursunda sınıfta tutulmuş notlardan oluşmaktadır. Notlar üzerinde hiçbir düzeltme yapılmamıştır. Notlardaki hataların pek çok nedenden kaynaklanabileceğini göz önünde bulundurunuz.

Notları tutan arkadaşa teşekkürler...

80X86 Sembolik Makina Dili (80X86 Assembly Language)

Neden Sembolik Makina Dili Öğrenilmelidir?....9
Mikroişlemcilerin Çalışma Biçimleri....9
Tipik Bir RAM'in Yapısı....10
CPU ile RAM Arasındaki Bağlantı....10
Makina Komutu Kavramı....11
Makina Komutlarının Genel Biçimi....11
80x86 Mikroişlemcisinin Çalışma Modları....12
8086 İşlemcisinin Yazmaç Yapısı....12
80386 ve Yukarı Modellerin Register Yapısı....14
Makine Komutlarındaki Bellek Operandları....15
16 Bit Çalışmada Bellek Operandının Oluşturulması....16
16 Bit Adresleme İşlemi....18
32 Bit Adresleme İşlemi....19
32 Bit Bellek Operandının Oluşturulması....20
16 Bit Çalışmada Flag Register'ı....20
CF(Carry Flag)....20
PF(Parity Flag)....21
AF(Auxilary Carry Flag)....21
ZF(Zero Flag)....21

SF(Sign Flag) 21
TF(Trap Flag)..21
IF(Interrupt Flag)...21
DF(Direction Flag) 22
OF(Overflow Flag) 22
Turbo Debugger 22
CS ve IP Register'larının Önemi..22
Makine Komutlarının İncelenmesi...23
MOV Komutu 23
ADD Komutu...23
ADC Komutu 24
Komutların Operand Uyumu 24
SUB Komutu..26
CMP Komutu 27
SBB Komutu (Subtract with Barrow) 27
Kalıp: 28
MUL Komutu..28
DIV Komutu..29
Bit Düzeyinde İşlem Yapan Komutlar 30
AND Komutu..30
TEST Komutu...30
Kalıp: 30
OR Komutu...31
XOR Komutu 31
SHL ve SHR Komutları...32
SAR ve SAL Komutları...32
Döndürme Komutları...32
Öteleme Ve Döndürme Komutlarının Biçimleri...33
Stack Kullanımı...33
POP Komutu 34
Stack Kullanımının Amacı..35
PUSHA ve POPA Komutları...36
INC ve DEC Komutları 36
XCHG Komutu 36
CBW(convert byte to word) ve CWD(convert word to double word) Komutları...............36
Dallanma Komutları...37
Yer Değiştirme(displacement) Kavramı..37
Koşulsuz JMP Komutu..38
Koşullu Dallanma Komutları...39
Eşitlik Karşılaştırması 40
Eşitsizlik Karşılaştırması 40
İşaretsiz Sayıların Karşılaştırılması...41
İşaretli Sayıların Karşılaştırılması...41
Diğer Koşullu JMP Komutları 42
Alt Programların Çağırılması..42
Alt Programdan Geriye Dönüş..43
Bayraklar Üzerinde Özel İşlem Yapan Komutlar 44
Sembolik Makine Dili Nedir?...45
Exe Dosyanın Yapısı ve Yüklenmesi 45
Tipik Bir Sembolik Makine Dili Programı 46

Code, Data ve Stack Bölümlerinin Belirlenmesi 47
Semboller 47
Data Sembollerinin Tanımlanması 48
Program Yüklendiğinde Register'ların Durumları 49
Programın Sonlandırılması 50
Code Sembolleri 50
proc Bildirimi 50
Gerçek ve Sahte Kodlar(real/pseudo) 51
Sabitler 51
Sabitlerin Çeşitli Tabanlarda Gösterimleri 51
Alfabetik Sabitler(string'ler) 52
Gerçek Sayı Sabitleri 52
BCD Türden Sabitler 52
Yer Sayacı(location counter) 52
ASM Listing Dosyası 53
Diziler Üzerinde İşlemler 53
LEA(load effective address) Komutu 53
Sembolik Makine Dilinde For Döngülerinin Oluşturulması 54
Kalıp: 54
Kalıp: 54
Kalıp: 54
Kod Sembolleri 55
Alt Programlarla Çalışma 55
Alt Programlara Parametre Geçirilmesi ve Alt Programların Geri Dönüş Değerlerini Almak 56
Fonksiyonların Geri Dönüş Değerlerinin Oluşturulması 61
Fonksiyon Çağırmalarına İlişkin Çeşitli Örnekler 61
Sembolik Makine Dilinde Yazılan Fonksiyonları C'den Çağırılması 62
Birleştirme sırasında çıkacak problemler 63
İç İçe Döngüler 64
Kalıp: 64
Kalıp: 65
Kalıp: 65
Sembolik Makine Dilinde cdecl Çağrımına Uygun Fonksiyon Yazımına Örnekler 67
C Derleyicilerin Sembolik Makine Dili Çıktıları 70
OFFSET ve SEG Operatörleri 71
EXE Dosya Formatı 71
Exe Dosyası Başlık Kısmı 72
PS P(Program Segment prefix) 73
Yükleme Sonrasında Register'ların İlk Konumları 74
Derleyicilerin Başlangıç Kodları(Startup Module) 74
C'de char Parametrelerin ve Geri Dönüş Değerlerinin Seyrek Kullanılması 76
KESMELER(interrupts) 76
INT Makine Komutu ve Kesmelere Dallanılması 77
IRET makine komutu 78
Kesmenin Hook Edilmesi 78
Kesme Kodunun Yazılması 79
Sembolik Makine Dilinde Dolaylı JMP ve CALL İşlemleri 80
Bellek Erişimlerinde Segment Yükleme Durumları 80
Kesmelerin Fonksiyonları ve Alt Fonksiyonları 81

Kesmenin Parametreleri ve Geri Dönüş Değerleri ... 81
DOS ve BIOS Kesmeleri ... 81
10h Video Kesmesi ... 82
INT 10h, F:0Ah(write char) ... 82
INT 10h, F:0Eh(write char) ... 83
21h Kesmesi ... 85
INT 21h F:01h ... 85
INT 21h F:2 ... 86
INT 21h F:7 ve F:8 ... 86
INT 21h F:0Ah (buffered keyboard input) ... 86
INT 21h F:25h(set interrupt vector) ... 87
INT 21h F:35h(get interrupt vector) ... 87
INT 21h F:39h(create sub directory) ... 87
INT 21h F:9h ... 88
Makro Kullanımı ... 89
Include İşlemi ... 90
LOCAL Komutu ... 92
Matematik İşlemcinin Kullanılması ... 93
Matematik İşlemci Nedir? ... 93
Normal İşlemci ile Matematik İşlemcinin Birlikte Çalışması ... 94
Matematik İşlemcinin Register Yapısı ... 94
Noktalı Sayıların Bellekte Tutulma Biçimleri ... 94
Noktalı Sayı Formatları ... 95
Yuvarlama Hatası(rounding error) ... 95
Float(short real) Format ... 96
BIAS Değerinin Anlamı ... 97
Noktalı Sayı Formatında Özel Sayıları ... 97
Double(long real) Formatı ... 98
Long Double(extended real) Formatı ... 98
Matematik İşlemcide Noktalı Sayılarla İşlemler ... 98
FLD Komutu ... 99
FST ve FSTP Komutları ... 99
Tam Sayılara İlişkin PUSH ve POP Komutları ... 99
FADD, FADDP Komutları ... 100
C'de Gerçek Sayı Türlerine Geri Dönen Fonksiyonları ... 101
FMUL ve FMULP Komutları ... 102
FDIV ve FDIVP Komutları ... 102
FSIN, FCOS, FTAN, FSQRT Komutları ... 102
BORLAND derleyicilerin Matematik İşlemci Seçenekleri ... 103
Gerçek Sayı Emülasyonu ... 103
Sembolik makine dilinde gerçek sayıları tutan sembollerin tanımlanması: ... 103
PipeLine İşlemi ... 104
Normal İşlemciyle Matematik İşlemcinin Senkronizasyonu ... 104
Wait komutunun çalışma biçimi: ... 105
Gerçek Sayıların Karşılaştırılması : ... 106
Status Register(status word) ... 106
Normal İşlemcilerde Bayrak Register'ı Üzerinde İşlemler ... 107
C0, C2 ve C3 Status Register Bit'lerinin Karşılaştırmadaki Anlamları ... 108
Programlama Dillerindeki Yerel Değişkenlerin Kullanılması ... 109
Hizalama(Alignment) ... 116

Word Hizalaması(word alignment) ... 116
Dword Hizalaması(dword alignment) ... 117
Hizalama Problemleri ... 117
C++'ta Bir Sınıfın Üye Fonksiyonlarının Sembolik Makine Dilinde Çağırılması ... 119
C'de Değişken Sayıda Parametre Alan Fonksiyonların Sembolik Makine Dilinde Yazımı .. 121
C'de değişken sayıda parametre alan fonksiyonların yazımı: ... 123
C'de Değişken Sayıda Parametre Alan Fonksiyonlara Örnekler ... 125
Uzak Göstericilerin Yüklenmesi ... 126
lds, les, lfs, lys Makine Komutları: ... 126
C Derleyicilerinin Uzak Gösterici İşlemlerini Ele Alış Biçimi ... 127
CPU'nun Durumunun Saklanması ... 127
CPU Konumunun Saklanıp Geri Yazılması Sırasında Dikkat Edilecek Durumlar ... 128
1) CS ve IP değerlerinin saklanıp yüklenmesi: ... 128
2) SS:SP register'larının saklanması ve yüklenmesi: ... 129
3) Diğer Register'ların saklanması ve yüklenmesi: ... 130
C'de Yerel Olmayan Dallanmalar ... 130
setjmp Fonksiyonu ... 131
longjmp Fonksiyonu ... 131
Neden longjmp Kullanılır? ... 132
setjmp ve longjmp Fonksiyonlarının Sembolik Makine Dilinde Yazımı ... 133
DOS'ta Uzak Modellerde İşlemler ... 134
Tiny Model ... 134
Small Model ... 134
Medium Model ... 134
Compact Model ... 134
Large Model ... 135
Huge Model ... 135
Uzak Modellerde Data Göstericileriyle İşlemler ... 135
Win32/UNIX Flat Model Sistemi ... 136
Ayrıntılı Segment Tanımları ... 136
Segment Tanımlama İşleminin Genel Biçimi ... 136
Segment'leri Exe Kod İçerisindeki Dizilim Sırası ... 139
Segment Tanımlamasında Kullanılan Sınıf İsminin Segment Sıralamasına Etkisi ... 140
C Derleyicileri ve Segment'ler ... 141
Tiny model ... 142
Small model ... 142
Medium model ... 142
Compact model ... 142
Large model ... 142
Huge model ... 142
Segment Kavramının Önemi ... 142
Ayrıntılı Segment Tanımlamaları ve Bellek Modeli ... 144
Birden Fazla Data Segment'i ile Çalışmak ... 144
Birden Fazla Kod Segment'i ile Çalışmak ... 144
Win32/UNIX Flat Modellerde Segment İşlemleri ... 145
Grup Kavramı ... 145
Grup Kullanımı ... 146
Assume Bildirimi ... 147
OMF Formatı ... 148
OMF Formatını Genel Yapısı ... 148

16 VE 32 BIT OMF KAYITLARI ... 148
Data ve Kod Sembollerinin obj Dosyaya Yazılması ... 150
Makine Kodlarının ve Statik Dataların obj Modüle Yazılması ... 152
Basitleştirilmiş Segment Kullanımında Segment İşlemleri ... 153
Relocatable Adresler ... 153
Relocatable Adresler ve Relocation Tablosu ... 154
COM Dosyalar ... 154
ORG Komutu ... 155
COM Programların Yazımı ... 155
Tek Segment'li COM Programlarının Yazılması ... 155
Çok Segment'li COM Programının Yazılması ... 156
Tiny Model Programlar ve COM Dosyaları ... 157
COM Programlarının Önemi ... 157
COM Programının Kullanılmasına Tipik Bir Örnek: Boot programının yazılması ... 157
İşletim Sistemi Yazımında İzlenecek Yöntem ... 159
İşletim Sisteminin Sistem Fonksiyonlarının Çağırılması için Kullanılan Yöntemler ... **Error! Bookmark not defined.**
Modüllerle Çalışma ... 160
Communal Tanımlama ... 162
String Komutları ... 163
Komutlarda Önek(prefix) ... 163
66 ve 67 Önekleri ... 163
Segment Önekleri ... 164
REP Öneki ... 164
LODS Komutu ... 164
LODSB Komutu ... 164
LODSW Komutu ... 165
STOS Komutu ... 165
REP Önekinin İşlevi ... 165
STOSB ... 165
Small model memset örneği: ... 166
MOVS Komutu ... 166
MOVSB ... 166
MOVSW ... 168
Heap Algoritması ... 168
Boş Bağlı Liste Algoritması ... 169
TINUX Sisteminde Heap Organizasyonu ... 170
Kernel Heap Fonksiyonlarının Tasarımı için Yapılacak İşlemler ... 171
SCAS Komutu: ... 172
Örnek Thread Kütüphanesi: ... 174
Thread Kütüphanesinin kullanımı: ... 174
InitThreadLib() fonksiyonu: ... 174
CreateThread() Fonksiyonu: ... 174
ExitThread() Fonksiyonu: ... 175
GetThreadExitCode() Fonksiyonu: ... 175
CloseHandle() Fonksiyonu: ... 175
EndThreadLib() Fonksiyonu: ... 175
Thread Kütüphanesinin Kullanılmasında Dikkat Edilmesi Gereken Noktalar: ... 175
Thread Kütüphanesinin İçsel Tasarımı: ... 176
C'de Inline Sembolik Makine Dili ... 177

DOS Sisteminde Heap Yönetimi 178
Tiny Model........ 178
Small Model........ 178
Medium Model........ 178
Compact Model........ 179
Large Model........ 179
Huge Model 179

Kaynaklar:

Peter Abel 80x86 Assembly Language
The Art of Assembly Language
Microsoft Macro Assembler 5.0 ve 6.0
Diğerleri

Assembler'da program yazabilmek için en azından bir Assembler derleyicisi ve bir de linker programı gereklidir. Borland firmasının assembler derleyicisi TASM.EXE dosyası, linker programı ise TLINK.EXE programıdır. Microsoft assembler derleyicisi ise MASM.EXE, linker programı ise LINK.EXE programıdır. Ayrıca bir assembler programının incelenmesi amacıyla "debugger" programları kullanılır. Borland firmasının TD.EXE Microsoft firmasının CV.EXE programları vardır.

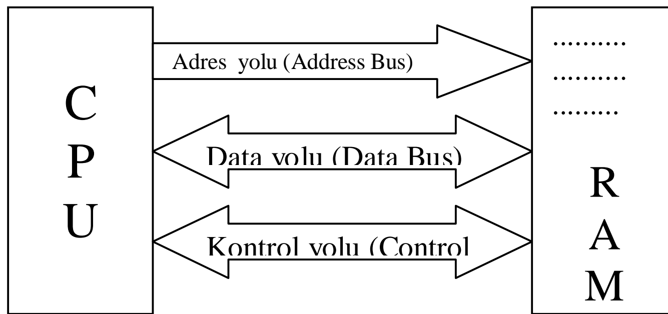
Assembler (Sembolik Makine Dili) taşınabilir genel bir dil değildir. Tamamen Mikroişlemciye bağlı olarak değişir.

Neden Sembolik Makina Dili Öğrenilmelidir?

1) Zaman ve kapasite problemi olan kritik kodların makine dili seviyesinde yazılması gerekebilir.C bu tür kodlar için yüksek seviyeli kalmaktadır.
2) Aşağı seviyeli programlama bilgisi bilinç düzeyini artırır. Böylece problemlerin nedenleri daha iyi anlaşılır.

Mikroişlemcilerin Çalışma Biçimleri

Mikroişlemci, bellekle 3 grup uç yoluyla bağlıdır.



Mikroişlemci döngüsel bir biçimde işlemlerini gerçekleştirir. Yani;
1) Bir grup byte topluluğunu RAM'den okur (instruction fetch).
2) Bu byte topluluğunun hangi komut olduğunu yorumlar.
3) Bu komutu çalıştırabilmek için elektronik devreleri çalıştırır.
4) Tekrar birinci adıma dönülerek işlemler yinelenir.

Bazı mikroişlemcilerde makine komutlarının uzunlukları hep aynıdır (Özellikle RISC grup işlemciler). Oysa Intel gibi pek çok CISC ailesi işlemcilerde komutlar farklı uzunluklarda olabilir. Mikroişlemci komutu yorumladıktan sonra uzunluluğunu da elde etmiş olur. Böylece bir sonraki komutun yerini de saptayabilir.

Mikroişlemcinin önemli bir bölümü, işlemlerin yapılmasını sağlayan mantık devreleriyle kaplıdır. Makine komutlarının operantları belleğe ilişkin olabilir. Örneğin bir makine komutu "500 numaralı bellek bölgesindeki sayıyı 1 artır" biçiminde olabilir. Bu durumda işlemci komutu çalıştırırken bellekten yine okuma ve yazma yapmak zorundadır. Yani mikroişlemci yalnızca komutu elde etmek için değil, komut içerisndeki operantları elde etmek için de belleğe erişir.

Tipik Bir RAM'in Yapısı

RAM'ler entegre devre biçiminde üretilir. Dış dünya ile bağlantıyı sağlayan çeşitli uçları vardır. Bir RAM genellikle byte biçiminde organize edilmiş gözeneklerden oluşur. Her gözenek içerisinde tipik olarak 8 bit vardır. Ama 8 bit olması zorunlu değildir. RAM'ler genellikle kapasite olarak ;

gözenek sayısı * gözenekteki bit sayısı'dır.

Örnek: 1024 * 8

1024 byte'lık bir RAM'de herhangi bir gözeneği seçebilmek için ikilik sistemde 10 tane elektriksel hat gerekir. Dışarıdaki bir devre bu uçlara **5 volt** ya da **0 volt** gerilim uygulayarak bir sayı oluşturur. RAM devresi de bu bilgiyi alarak hangi gözeneğin seçilmiş olduğunu anlar. Bir gözeneği seçmekte kullanılan bu uçlara RAM'in adres uçları denir. A0, A1, A2, ... biçiminde gösterilir. Bunun dışında gözenek seçildikten sonra okuma mı, yoksa yazma mı yapılacağını anlatmaya sıra gelir. Bu işlem ayrı bir uçtan yapılır. BU uca **R / W** ucu denir. Örneğin bu uca **5 volt** uygulanırsa okuma yapılacağı, **0 volt** uygulanırsa yazma yapılacağı anlamına geliyor olabilir. RAM'in gözeneklerdeki bilgiyi aktarmak için kullanılan bir veri ucu vardır.Bu uçlar genellikle D0, D1, D2, ... olarak isimlendirilir. (www.dalsemi.com adresinden Akbil'in mekanizmasını öğrenebilirsin (1991 işlemci).) Bu durumda 1024 * 8'lik bir RAM de 10 tane adres ucu, 8 tane data ucu ve bir tane de R/W ucu olmalıdır. ve ek olarak başka uçlar da vardır.

RAM'den okuma şöyle yapılır:
1.Adım : Adres uçlarına gözenek numarası girilir.
2.Adım : R/W okuma konumuna getirilir.
3.Adım : Bir süre beklenir ve data uçları örneklenir.

Yazma işlemi ise şöyle yapılır:
1.Adım : Gözenek numarası adres uçlarına bırakılır.
2.Adım : R/W ucu yazma konumuna getirilir.
3.Adım : Yazılacak bilgi ikilik sistemde data uçlarına bırakılır.

CPU ile RAM Arasındaki Bağlantı

Genel olarak CPU'nun adres uçları RAM'in adres uçlarıyla, data uçları da RAM'in data uçlarıyla bağlantılıdır. Benzer biçimde CPU'nun kontrol uçları RAM'in çeşitli kontrol uçlarıyla bağlantılıdır.

Bir CPU'nun adresleyebildiği maksimum fiziksel RAM kapasitesi vardır. Bu kapasite CPU'nun adres uçlarının sayısıyla doğrudan ilgilidir. Örneğin Z-80 ve 8080 işlemcilerinin 16 adres ucu vardır. Bu işlemciler ancak 64 KB bellek kullanabilirler. Intel 286 işlemcisinin 24 adres ucu, 386 ve 486 işlemcilerinin 32 adres ucu vardır. İşlemcinin RAM'den bir seferde transfer edeceği bilgi veri data uçlarının sayısına bağlıdır. Z80 ve 8080 işlemcileri 8, 8086 ve 80286 işlemcileri 16, 386 ve 486 işlemcileri 32 data ucuna sahiptir.

CPU içerisindeki elektronik devrelerle bağlantılı olan RAM'den çekilen bilginin geçici süre saklanmasında kullanılan CPU içerisindeki küçük bellek bölgelerine *register* denir. Bir mikro işlemcinin kaç bitlik mikro işlemci olduğu register uzunluğu ile belirlenir. Örneğin 80386 işlemcisi 32 bitlik bir işlemcidir, çünkü 32 bit register'lara sahiptir. Bir mikro işlemcinin register uzunluğu işlemcinin bir hamlede kaç bitlik bilgi üzerinde işlem yapabildiğini anlatır. Örneğin 8086 16 bitlik bir mikro işlemcidir. 32 bit iki sayı toplanacak olsa bu toplama işlemi tek işlemde değil ancak iki işlemde yapılabilir(C'de int türü derleyiciyi yazanlar tarafından genellikle işlemcinin register uzunluğu kadar alınır).

Makina Komutu Kavramı

Mikroişlemci her işlemi bir makina komutuyla yapar. Makina komutu işlemciye hangi işlemin yapılacağını anlatan byte topluluğudur. Intel işlemcilerinde makina komutlarının byte uzunlukları farklı olabilmektedir. Her mikroişlemcinin bir komut kümesi vardır. Bütün program bu komutlarla ifade edilmek zorundadır. Makina komutlarının sayısı CISC ailersi mikroişlemcilerde, RISC ailesi mikroişlemcilerine göre daha fazla ve çeşitlidir.

Makina Komutlarının Genel Biçimi

Her makina komutu gerçekte ikilik sistemde bir byte topluluğudur. Ancak sembolik makina dilinde sayılar yerine sembolik ifadeler kullanılarak gösterilirler. Zaten sembolik makina dili derleyicilerinin yaptığı şey genelde sembolik olarak yazılmış olan bu komutları sayılara dönüştürmektir. Bir makina komutu hangi işlemin yapılacağını anlatan bir işlem bilgisi ve operandlardan oluşur. Makina komutları tek operandlı ya da iki operandlı olabilirler. Makina komutlarının genel biçimi şöyledir:

Komut operand
Komut operand1, operand2

Örneğin:

INC AX
ADD AX, BX

Intel işlemcilerinde tek operandlı komutlarda operand register'a ya da belleğe ilişkin olabilir. İki operandlı komutlarda her iki operand da belleğe ilişkin olamaz. Operandlardan herhangi birisi belleğe diğeri register'a ilişkin olabilir. Operandlardan her ikisi de register'a ilişkin olabilir. Bazı makina komutlarının operandı yoktur. Bu komutlardan bazıları default bir takım register'ları operand olarak kullanırlar. Özetle makina komutları :

- Ya operandsız olur,
- Ya tek operandlı olur,
- Ya da iki operandlı olur.
- İki operandlı komutlarda her iki operand da belleğe ilişkin olamaz.

Genel olarak bir operand register'a, belleğe ya da sabite ilişkin olabilir. İki operandlı komutlarda bir operand belleğe ilişkinken, diğer operand bir sabite ilişkin olabilir. Sonuç olarak Intel işlemcilerinde komutların rastlanabilen biçimleri şunlardır:

Komut
Komut sabit
Komut reg
Komut mem
Komut reg, mem
Komut reg sabit
Komut reg, reg
Komut mem, reg
Komut mem, sabit

80x86 Mikroişlemcisinin Çalışma Modları

80x86 mikroişlemcisinin üç çalışma modu vardır.

1. Gerçek Mod (Real Mode)
2. Sanal86 Mod (Virtual 86 Mode)
3. Korumalı Mod (Protected Mode)

80x86 işlemcileri reset edildiğinde çaçlışma gerçek modda başlar. Korumalı moda gerçek moddan yazılım yolu ile geçilmektedir. 8086, 8088, 80186 işlemcileri sadece gerçek moda çalışabiliyordu. 80286 işlemcisii gerçek mod ve korumalı modlarda çalışabilmektedir. 80386 ve sonrası bu üç modu destekelemektedir. 80X86 işlemciler gerçek modda çok küçük farklılıklar dışına hızlı bir 8086 gibi çalışmakatadır. DOS işletim sistemi gerçek modda çalışabilecek biçimde tasalanmıştır. 8086 işlemcisi 1 MB bellek kullanabilen 16 bit bir mikroişlecidir. Bu nedenle gerçek modda ancak 1 MB bellek kullanılabilir. Korumalı mod koruma mekanizmasının, sanal bellek kullanımının, çok işlemli çalışmanın, mümkün olduğu en ileri çalışma modudur. UNIX ve Windows sistemleri korumalı modda çalışmaktadır. Sanal 86 Modu 8086 gibi çalışmanın sağlandığı ancak korumalı modun çeşitli özelliklerinin kullanılabildiği bir ara moddur. Windows işletim sisteminde komut satırı Sanal 86 Modunda çalışmaktadır çünki Windows işletim sisteminde kullanılan taskswitch mekanizmasında Gerçek Mod kullanılamamaktadır. Windows işletim sisteminde DOS penceresi açıldığında yada herhangi bir DOS programı çalıştırıldığında işlemci Sanal 86 moduna geçmektedir. Ancak işletim sisteminin açılışında F8 tuşuna basılarak Sadece Komut İstemi seçeneği seçildiğinde Gerçek Modda çalışma sözkonusu olur.

8086 İşlemcisinin Yazmaç Yapısı

8086 mikroişlemcisi toplam 14 yazmaca sahiptir.

4 adet genel amaçlı yazmaç vardır

AX(Accumlator Register),
BX(Base Register)
CX(Count Register)
DX(Data Register)

Bu yazmaçlar bütün olarak 16 bit biçiminde kullanılabilir yada düşük ve yüksek anlamlı kısımları bağımsız 8 bitlik yazmaçlar gibi de kullanılabilir. Yani 12 adet yazmaç ifadesi yazılabilir. 8 bitlik parçalar bütünü oluşturur. Yani örneğin AH ve AL yazmaçlarına yükleme yapıldığında AX yazmacı oluşturulmuştur.

2 adet index yazmacı vardır.

SI(Source Index Register)
DI(Destination Index Register)

Bu iki yazmaç 8 bitlik parçalara bölünmemiştir. Data bölgesini indekslemek amacıyla kullanılır.

3 Adet Gösterici Yazmacı Vardır (Pointer Register)
IP(Instruction Pointer Register)
SP(Stack Pointer Register)
BP(Base Pointer Register)

4 Adet Segment Yazmacı Vardır
CS(Code Segment Register)
DS(Data Segment Register)
SS(Stack Segment Register)
ES(Extra Segment Register)

1 Adet Bayrak Yazmacı Vardır
F

Bütün yazmaçlar 16 bit uzunluğundadır ancak sadece genel amaçlı yazmaçlar ayrıca parçalara ayrılmışlardır.

Her komut her yazmaç ile çalıştırılamayabilir. Aritmetik işlemler, karşılaştırma işlemleri yada bit işlemleri için Genel Amaçlı Yazmaçların hepsi kullanılabilir. SI ve DI yazmaçları indeksleme amacıyla tasarlanmış olmalarına karşın Genel Amaçlı Yazmaçlarla aynı işlemlere kullanılabilirler.

Aritmetik, karşılaştırma ve bit işlemleri 16 bit ise AX, BX, CX, DX, SI, DI yazmaçlarıyla yapılabilir. Aynı işlemler 8 bit yapılacak ise AH, AL, BH, BL, CH, CL, DH, DL yazmaçları kullanılabilir.
Kural 2 operandlı bir makina komutunun sonuçları her zaman soldaki operand bozularak onun içerisine yazılır. Tek operandlı makina komutunun sonuçları operand içerisindeki değer bozularak yazılmaktadır. Örneğin:

ADD AX, BX

işleminde sonuç AX yazmacına yazılacaktır. Yada örneğin:

ADD MEM, AX

işleminde MEM ile belirtilen bellek bölgesindeki bilgi ile AX yazmacı içerisindeki bilgi toplanır sonuç MEM ile belirtilen bellek bölgesine yazılır. Yada örneğin

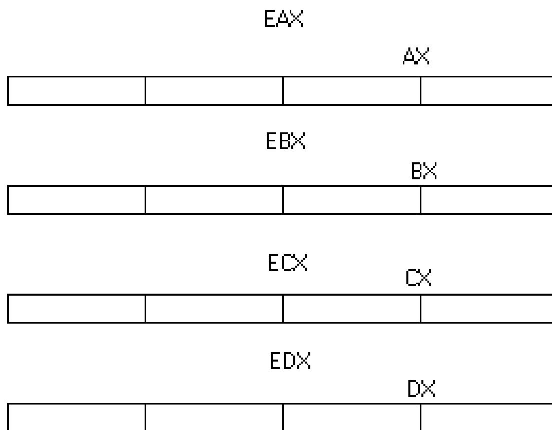
INC AX

Sonuç operand üzerine yazılır.

80386 ve Yukarı Modellerin Register Yapısı

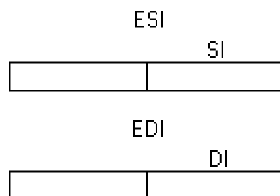
80386'dan itibaren geçmişe uyum korunarak register'lar 32 bite yükseltilmiştir. 80386 ve sonraki modellerin pek çok register'ı vardır. Ancak bu register'ların çoğu korumalı mod ile ilgilidir. Bu modellerde asıl işlevsel olan register'lar 8086 işlemcisindekilerin genişletilmiş biçimleridir.

- Genel amaçlı register'lar uyum korunarak 4 byte'a yükseltilmiştir.

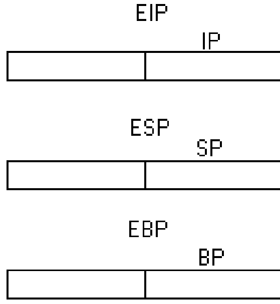


Bu durumda örneğin EAX register'ı bütün olarak EAX biçiminde, 16 bit AX biçiminde ya da 8'er bitlik AL ve AH biçiminde kullanılabilir. Bu register'ların yüksek anlamlı 16 bitleri bağımsız olarak kullanılamamaktadır.

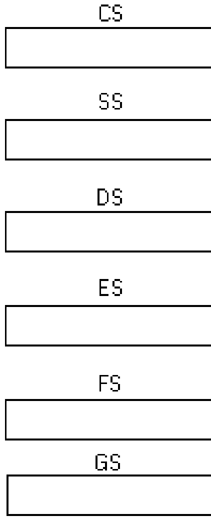
- Index register'lar da 32 bite yükseltilmiştir.



- Pointer register'lar da 32 bite yükseltilmiştir.



- Segment register'lar 16 bit olarak kalmıştır, FS ve GS isimli iki segment register daha eklenmiştir.



- Flag register 32 bite yükseltilerek EFLAGS ismini almıştır.

Makine Komutlarındaki Bellek Operandları

Bir makine komutunda sabit sayılar doğrudan bellek operandları ise köşeli parantez içerisinde gösterilirler. Örneğin:

MOV AX, 100

100 bir sabittir. Bu komut 100 sayısının AX register'ına atanacağını belirtir. Oysa

MOV AX, [100]

100 numaralı bellek bölgesindeki bilginin AX register'ına atanacağını belirtir. Mikroişlemci register bellek işlemlerinde bellekten ne kadar bilginin transfer edileceğini register operandına bakarak anlar. Örneğin:

MOV AX, [100]

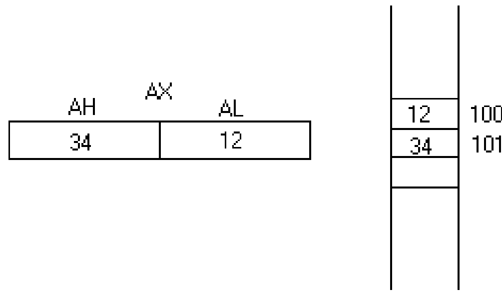
Burada register operandı 2 byte olduğu için 100 ve 101 numaralı byte değerleri AX register'ına atanacaktır. Eğer komut

MOV AL, [100]

biçiminde verilseydi, register operandı 1 byte olduğu için yalnızca 100 numaralı bellek bölgesindeki bilginin AL register'ına atanacağı anlamına gelir.

Intel işlemcileri "Little Endian" notasyonunu kullanır. Yani sayısının düşük anlamlı byte değeri düşük anlamlı adreste bulunur. Örneğin:

MOV AX, [100] komutu aşağıdaki gibi işlenecektir.



16 Bit Çalışmada Bellek Operandının Oluşturulması

16 bit çalışmada köşeli parantezler içerisine getirilecek bellek operandları sabit ya da register içerecek biçimde olabilir. Köşeli parantez içerisinde bellek operandı biçiminde bulunabilecek register'lar BX, BP, DI ve SI register'larıdır. BX ve BP register'larına base register'lar SI ve DI register'larına ise index register'lar denir. Köşeli parantez içerisinde bu register'lar tek başlarına bulunabilir ya da base ve index register toplamı biçiminde bulunabilir. İki base register toplamı ve iki index register toplamı yasaklanmıştır. 16 bit çalışmada bütün bellek operandları aşağıdaki gibi oluşturulabilir.

disp8	8 bitlik bir sabit sayı
disp16	16 bitlik bir sabit sayı

1. [disp16]
2. [BX], [BP], [SI], [DI]
3. [BX + disp8], [BP +disp8], [SI+ disp8], [DI +disp8]
4. [BX + disp16], [BP +disp16], [SI+ disp16], [DI +disp16]
5. [BX + SI], [BX + DI], [BP + SI], [BP + DI]
6. [BX + SI + disp8], [BX + DI + disp8], [BP + SI + disp8], [BP + DI + disp8]
7. [BX + SI + disp16], [BX + DI + disp16], [BP + SI + disp16], [BP + DI + disp16]

Buradaki olasılıklar sözel olarak şöyle açıklanabilir:
1. Köşeli parantez içerisinde 16 bit sabit bir sayı
2. BX, BP, SI, DI register'ları tek başlarına köşeli parantez içerisinde
3. BX, BP, SI, DI register'ları disp8 toplamıyla köşeli parantez içerisinde

4. BX, BP, SI, DI register'ları disp16 toplamıyla köşeli parantez içerisinde
5. Bir base ve bir index register'ı toplamı köşeli parantez içerisinde
6. Bir base, bir index ve disp8 toplamı köşeli parantez içerisinde
7. Bir base, bir index ve disp16 toplamı köşeli parantez içerisinde

[disp8]	Geçersiz
[disp16]	Geçerli
[AX + BX]	Geçersiz
[BP + SI]	Geçerli
[SI + DI + disp16]	Geçersiz
[BX]	Geçerli
[BX + DX]	Geçersiz
[BX + SI + disp16]	Geçerli

C'de	Assembler karşılığı
int a; *int *p;* *p = &a;* **p = 100;*	*MOV p, &a* *MOV BX, p* *MOV [BX], 100*

Her bellek operandının default bir segment register'ı vardır. BX, DI, ve SI register'larının default segment register'ı DS, BP register'ının SS'dir. İki register toplamında eğer toplamda BP register'ı bulunuyorsa BP'nin default register'ı olan SS toplam bellek operandının segment register'ı olur. Özetle:

1. BX, SI, DI tek başlarına ya da disp8, disp16 toplamlarıyla bulunduğunda default segment register DS'dir.
2. BP tek başına ya da disp8, disp16 toplamlarıyla bulunduğunda default segment register SS'dir.
3. Base, index ve disp8, disp16 toplamlarında eğer toplamlardan bir BP register'ı ise default segment register SS'dir. Yoksa DS'dir.
4. [disp16] operandının segment register'ı DS'dir.

Örneğin:

İşlem	Default Segment Register
[BX]	DS
[BP + SI]	SS
[SI + disp16]	DS
[disp16]	DS
[SI + BX]	DS

Bellek operandının default segment register'ı 1 byte uzunluğunda makine komutu eklenerek değiştirilebilir. Sembolik makine dilinde bu değiştirme işlemi *segreg:[operand]* biçiminde yapılır. Örneğin:

ss:[BX]
cs:[SI]
es:[BX]
ds:[BP]

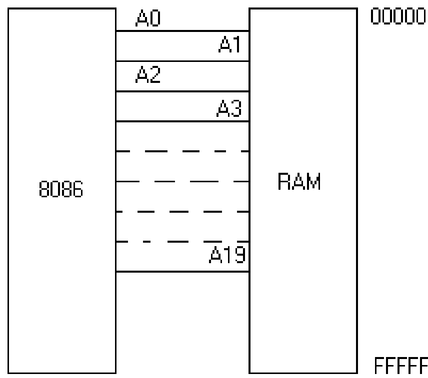
Örneğin bir komut içerisinde aşağıdaki gibi kullanılabilir:

MOV AX, ES:[BX]

Bellek operandının sembolik makine dilinde belirtilmesinde iki eşdeğer biçim kullanılır. bellek operandı toplam içeriyorsa + sembolü kaldırılıp gösterim ayrık köşeli parantezlerle yapılabilir. Örneğin [BX + SI] ile [BX] [SI] eşdeğerdir. Ya da örneğin [BX + SI + disp8] ile [BX] [SI] [disp8] eşdeğerdir. Toplamanın değişme özelliği olduğu için toplam ifadesindeki elemanlar yer değiştirebilir.

16 Bit Adresleme İşlemi

8086 mikroişlemcisi 1 MB bellek kullanabilecek biçimde tasarlanmıştır. 8086 işlemcisinin 20 tane adres ucu vardır(A0-A19). 1 MB bellek 5 hex digit ile gösterilebilir.



16 bit çalışmada köşeli parantez içerisindeki bellek operandı en fazla 16 bit uzunluğunda olabilir. Toplam bu uzunluğu aşsa bile yüksek anlamlı bitler atılarak düşük anlamlı 16 bit elde edilir. 8086 işlemcisi bir bellek operandıyla karşılaştığı zaman 20 bitlik fiziksel adresi şöyle bulur:

1. Fiziksel adres 2 byte segment, 2 byte offset bilgisinden elde edilir. Segment değeri segment register'larının içerisindedir. Offset değeri bellek operandı olarak köşeli parantez içerisindedir. Mikroişlemci bellek operandıyla ilişkili segment register'ı tespit eder ve o register'ın değerini 16 ile çarpar(sağına 0 ekler).
2. Bunu köşeli parantez içerisinde bulunan offset değeriyle toplar. Sonuç 5 hex digit uzunluğunda bir bilgidir. Bu bilgi adres yoluna verilir.

Örneğin:

DS = 1234
BX = 1000 ise

MOV AX, [BX]

işlemi sonucu AX register'ına 1234 * 16 + 1000 = 13340 fiziksel adresindeki veri yerleştirilir. Görüldüğü gibi istenilen bir fiziksel adrese erişebilmek için yalnızca köşeli parantez

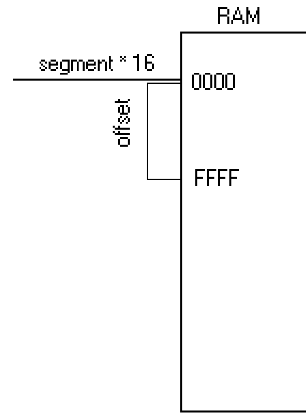
içerisindeki offset'i ayarlamak yetmez. İlgili segment register'ın içerisinde de uygun değerin olması gerekir. Örneğin:

SS = 138F
BP = 1000 ise

MOV AX, [BP + 10]

işlemi sonucunda AX register'ına 138F * 16 + 1000 + 10 = 14900 fiziksel adresindeki veri yerleştirilir.

Segment register içerisindeki değeri hiç değiştirmeden yalnızca offset değerini değiştirerek segment * 16 adresinden ancak 64 KB uzaklaşılabilir.



İstenilen fiziksel adrese erişmek için ters bir işlem yapmak gerekir. Yani:

1. Fiziksel adres segment-offset çiftine ayrıştırılır. Bir fiziksel adres için pek çok segment-offset çifti yazılabilir.
2. Segment değeri bir segment register'ına yazılır. Offset değeri köşeli parantez içerisinde bellek operandı biçiminde oluşturulur.

32 Bit Adresleme İşlemi

80386 ve sonrası 32 bit işlemcilerdir. Bu işlemcilerde 32 bit register'lar vardır ve bellek operandları yani köşeli parantez içerisindeki değer 32 bit olabilir. 32 bit register'lar bu işlemcilerde yalnızca korumalı modda değil, gerçek modda ve sanal 86 modda da kullanılabilmektedir. Tabii 32 bit register'ların kullanıldığı bir DOS programı 8086 işlemcisinin bulunduğu bir makinede çalışmaz. Yani bir DOS programında 32 bit hesaplamalar yapabilmek için EAX, EBX gibi register'lar kullanılabilir. Ancak program yine bir DOS programıdır. Örneğin yine 1 MB bellek kısıtlaması vardır. 80386 ve sonraki modellerde ve korumalı modu kullanan bir işletim sistemiyle çalışılıyorsa tam ve rahat bir 32 bit çalışmayı gerçekleştirebiliriz.

32 bit bellek adreslemesinde köşeli parantez içerisindeki değer 32 bit olabilir. Tabii DOS sistemi için yani gerçek mod ya da sanal 86 modu için bu işlemin bir faydası yoktur. Tabii korumalı modda bu çeşit bir adresleme kullanılmaktadır.

32 Bit Bellek Operandının Oluşturulması

Korumalı modda 32 bitlik bellek operandı 16 bit'ten daha geniş ve farklı biçimde oluşturulmaktadır. 32 bit'lik bellek operandları şöyle oluşturulur:

1. [disp32]
2. [EAX], [EBX], [ECX], [EDX], [EBP], [ESI], [EDI]
3. [EAX + disp8], [EBX + disp8], [ECX + disp8], [EDX + disp8], [EBP + disp8], [ESI + disp8], [EDI + disp8]
4. [EAX + disp32], [EBX + disp32], [ECX + disp32], [EDX + disp32], [EBP + disp32], [ESI + disp32], [EDI + disp32]
5. EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP'nin ikili bütün toplamları
6. EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP'nin bütün ikili toplamları + disp8
7. EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP'nin bütün ikili toplamları + disp32
8. 5., 6. ve 7. maddelerdeki oluşumlarda ikinci toplam operandı olan register 1, 2, 4, 8 çarpanlarını alabilir.

[EAX]	Geçerli
[EBX * 4]	Geçersiz
[ECX + EAX * 4]	Geçerli
[EAX + EBX + disp16]	Geçersiz
[EAX + EBX + 32]	Geçerli
[EAX * 2 + disp8]	Geçersiz
[ESI + EAX * 4 + disp32]	Geçerli
[EAX + EAX]	Geçerli

16 Bit Çalışmada Flag Register'ı

8086 işlemcisinde F ya da FLAG biçiminde belirtilen ve ismine bayrak register'ı denilen bir register vardır. Bu register 386 ve sonraki modellerde korumalı modda kullanılmak üzere EFLAGS ismi verilerek 32 bit'e yükseltilmiştir. Flag register'ı bit bit anlamlı bir register'dır. Her bit'in anlamlı diğerinden farklıdır. Mikroişlemci çeşitli makine komutlarını çalıştırdıktan sonra komutların sonuçları hakkında ilave bir takım bilgiler verir. Örneğin "Bir toplama işlemi yapıldığında taşma olmuş mudur?" ya da "Oluşan sayının işaret bit'i nedir?" gibi. Flag register'ının bit'leri komut çalıştırıldıktan sonra işlemci tarafından 1 ya da 0 yapılır. Bu bit'lerin 1 yapılmasına "set edilmesi", 0 yapılmasına "reset edilmesi" denir. Her makine komutu bayrak register'ının bit'leri üzerinde etkili olmaz. Bir komutun bütün bit'leri etkileyeceği anlamı da çıkmamalıdır. Bir komutun flag register'ının hangi bit'lerini etkilediği, bu etkiden çıkan sonuçlar ilgili komut öğrenilirken ayrıca öğrenilmek zorundadır.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

CF(Carry Flag)

Bu flag aritmetik, bit ve mantıksal işlemlerde sayının bütününe ilişkin bir elde oluştuğunda 1 yapılır. Örneğin:

AL: 13

BL: F2
ADD AL, BL

Burada bir toplama yapılmıştır. Ancak sonuç 1 byte'ı aşmıştır. Bu durumda işlemci CF bit'ini set eder.

PF(Parity Flag)

Bir işlem sonrasında düşük anlamlı byte içerisindeki 1'lerin sayısı çift ise bu bit set edilir, tek ise reset edilir(Bu yönteme odd parity yöntemi denir).

AF(Auxilary Carry Flag)

Dördüncü bit'ten beşinci bit'e doğru oluşan elde durumunda set edilir. Elde yoksa reset edilir. Özellikle BCD(binary coded decimal) işlemleri için düşünülmüştür. HEX sistemde düşünüldüğünden düşük anlamlı hex digit'teki eldeyi belirtir. Örneğin:

AL:18
BL: 1C
ADD AL, BL

işleminde CF 0, AF 1 olur.

ZF(Zero Flag)

İşlemin toplam sonucu 0 ise bu flag set edilir, değilse reset edilir. Örneğin:

SUB AX, BX

işleminde iki register'ın değerleri eşitse sonuç 0 olacağı için ZF set edilir.

SF(Sign Flag)

İşlem sonucunda elde edilen sayının en solundaki bit'i bu flag'e yansıtılır. Örneğin:

ADD AX, BX

işlemi sonucunda SF 1 ise s AX içerisinde negatif bir sayı vardır. SF 0 ise pozitif bir sayı vardır.

TF(Trap Flag)

İşlemci her komutu çalıştırdıktan sonra bu flag'in durumuna bakar. Eğer bu flag set edilmiş ise 1 numaralı içsel kesmeyi çağırır. Bu kesmeye tek adım kesmesi(single step interrupt) denilmektedir.

IF(Interrupt Flag)

İşlemcinin INT ucu uyarıldığında işlemci kesme durumuna geçmeden önce bu flag'e bakar. Eğer bu flag 1 ise(normal olarak 1'dir) kesme kabul edilir. Böylece işlemci ACK ucunu aktive ederek bunu bildirir. Eğer bu flag 0 ise işlemci kesmeyi görmezlikten gelir,

kesme koduna dallanmaz, işlemine devam eder. Çok işlemli bir işletim sisteminde bu flag'in 0 yapılması tüm sistemi çökertebilir. Çünkü donanım kesmeleri devre dışı bırakıldığında process'ler arası geçiş işlemi de durur. Adeta sistem tek işlemli olarak devam eder. Korumalı modda sıradan bir programcının bu flag'i 0 yapması çeşitli biçimlerde engellenmiştir.

DF(Direction Flag)

Bu flag string komutları için işlemci tarafından kullanılmaktadır. Bu komutlarda transferin yönünü anlamak için bu flag'e bakar.

OF(Overflow Flag)

İşaretli sayı üzerinde meydana gelen taşma durumunu tespit etmek amacıyla kullanılır. Yani işlem sonucunda işaret biti değişmişse bu flag set edilir. Örneğin:

AX: 7FFF
INC AX

OF biti pozitif bölgeden negatif bölgeye geçişte ya da tersinde set edilebilir.

Turbo Debugger

Turbo debugger programı ile şunlar yapılabilir:

- Manuel olarak makine komutları girilebilir ve çalıştırılabilir.
- Komutlar çalıştırılırken CPU register'larının durumları incelenebilir.
- Bellekteki herhangi bir bölge incelenebilir.
- Bir exe dosya program yüklenerek makine kodları incelenebilir.

CS ve IP Register'larının Önemi

IP register'ı offset belirtir. Segment register'ı CS'dir ve değiştirilemez. Mikroişlemci komutları şöyle çalıştırmaktadır: CS:IP register'ının gösterdiği yerden bir byte topluluğunu okur, bunu komut olarak yorumlar, işlemini yapar ve IP register'ını komutun uzunluğu kadar arttırır. Yani CS ve IP birlikte işlemcinin o anda çalıştırdığı makine komutunun adresini belirtmektedir. Aslında her mikroişlemcide ve mikrodenetleyicide bu görevi yapan bir register vardır. Genellikle bu register PC(program counter) biçiminde isimlendirilir. Turbo debugger'a ilk girildiğinde bütün segment register'lar aynı değeri gösterir. Bu segment adresi ilk boş bölgenin adresidir. Turbo debugger'da ana pencerede şöyle bir görüntü vardır:

CS:IP	Makine komutunun hex karşılığı	Komutun sembolik karşılığı
CS:0100	56	PUSA SI

Sağ taraftaki pencerede register'ların değerleri vardır. Komut çalıştırılınca bu değerler değişebilir. En sağdaki pencerede flag register'ındaki bit'ler görülmektedir. Aşağı pencerede DS segment register'ı kullanılarak bellekten bir kesit verilmiştir. Bir makine komutu yazılırsa makine komutu bandın bulunduğu adrese girilmiş olur. Bir register'ın durumunu dışarıdan değiştirmek için o register'ın üzerinde sayı yazılır. F7 tuşu bir komutu çalıştırmaya yarar.

Makine Komutlarının İncelenmesi

Her makine komutunun firma tarafından belirlenmiş biçimleri vardır. Bir komut öğrenilirken bütün bu biçimlerin öğrenilmesi gerekir. Komutların açıklanmasında kullanılacak kısaltmalar şunlardır:

Sreg	Segment register
Reg	Register
reg/mem	Register veya memory
Sabit	Sabit bir sayı

MOV Komutu

Bu komut register ve memory arasında transfer işleminde kullanılır. Biçimleri şunlardır:

1. reg ßà reg/mem
2. reg/mem ß sabit
3. sreg ßà reg/mem

Not: Bir işlem yapılacaksa ilk tercih edilecek register AL ya da AX olmalıdır. Çünkü işlemci eğer operand AL ya da AX ise bazı komutları daha etkin çalıştırır.
Görüldüğü gibi doğrudan segment register'a sabit atanamamaktadır. Örneğin:

MOV SS, 1234

yerine

MOV AX, 1234
MOV SS, AX

yapılabilir. MOV makin komutu hiçbir flag register'ını etkilemez. Bellekte istenilen bir bölgeyi görütülemek için bellek penceresine gelinir, mouse'un sağ tuşuna basılır, goto seçilir. Orada şunlar geçerlidir:

- Yalnızca offset
- sreg:offset
- segment:offset

Not: Hex sitemde alfabetik karakterlerle başlayan sayılar değişken isimleriyle karışabileceği için başına 0 eklenerek girilmelidir. Örneğin: FC10 biçiminde değil 0FC10 biçiminde.

ADD Komutu

Biçimleri:

1. reg ßà reg/mem
2. reg/mem à sabit

Örnekler:

MOV AX, BX

MOV AX, [SI]

Bu komut AF, CF, OF, PF, SF ve ZF flag'leri üzerinde etkili olmaktadır.

ADC Komutu

Biçimleri:
1. reg ßà reg/mem
2. reg/mem à sabit

Mikroişlemcilerin hemen hepsinde bu işlemi yapan bir toplama komutu vardır. ADD komutundan farklı olarak iki operandın toplamından başka bir de CF bayrağını da toplar. Bu komut yardımıyla işlemcinin register uzunluğundan büyük iki tamsayı toplanabilir. Böylelikle 16 bit sistemde 32 bit'lik iki sayı önce düşük anlamlı WORD değerleri ADD komutuyla sonra yüksek anlamlı byte'ları ADC komutuyla toplanabilir.

Örneğin: DOS altında C'de iki long sayıyı topladığımızda işlemler aşağıdaki gibi yapılacaktır.
long a=0x12345678, b = 0x87654321;
c = a + b;

DS:200 12345678
DS:204 87654321
MOV AX, [200]
ADD AX, [204]
MOV [208], AX
MOV AX, [202]
ADC AX, [206]
MOV [20A], AX

78	200
56	201
34	202
12	203
21	204
43	205
65	206
87	207
	208
	209
	20A
	20B

Belleğin istediğimiz bir bölgesine bir değer girmek için mouse bellek bölgesine getirilip, istenilen bölgede click yapılır ve istenilen sayılar klavyeyle girilir.

Komutların Operand Uyumu

İstisnaları olmasına karşın genel olarak iki operand alan makine komutlarında sol taraftaki hedef operand ile sağ taraftaki kaynak operandın uzunluklarının aynı olması gerekmektedir. Örneğin aşağıdakiler geçersizdir.

MOV AX, BL
MOV EBX, CX

Eğer operandlardan biri register diğeri bellek ise bellek operandının uzunluğu register'a bağlıdır. Örneğin:

MOV AL, [SI]	1 byte
MOV [SI], AX	2 byte

Ancak sol taraftaki operand bellek, sağ taraftaki operand bir sabit olduğunda debugger programı ya da sembolik makine dili derleyicileri sabit olan operandın 1 byte mı yoksa 2 byte mı olduğunu anlayamazlar. Örneğin:

MOV [SI], 1C

burada 001C sayısı mı söz konusudur, yoksa yalnızca 1C sayısı mı söz konusudur? Bu işlemde belirsizlik *byte ptr*, *word ptr*, *dword ptr* belirleyicileriyle sağlanır. Bu belirleyiciler bellek operandlarının önüne yerleştirilir. Örneğin:

MOV word ptr [SI], 1C

burada iki byte'lık işlem söz konudur. Yani aslında 001C sayısı belirtilmiştir. Bu belirleyiciler register bellek işlemlerinde de kullanılabilirler ama bir işlevi yoktur. Örneğin:

MOV AX, word ptr [SI]

Register bellek işlemlerinde 32 bit'lik register'lar kullanılabilir. Bu durumda bellek operandı 16 bit ya da 32 bit offset içerebilir. Örneğin aşağıdaki iki komut da geçerlidir:

MOV EAX, [SI]
MOV EAX, [ESI]

Yani özetle 32 bit register kullanıldığında bellek operandına ilişkin offset(yani köşeli parantez içerisindeki değer) 2 byte ya da 4 byte olabilir. DOS altında çalışıyorsak 4 byte register 2 byte offset yapısını serbest bir biçimde kullanabiliriz. Ancak 4 byte register 4 byte offset yapısını DOS'ta değil, korumalı modda kullanmalıyız. Örneğin 2 long sayıyı 32 bit register'lar kullanarak aşağıdaki gibi toplayabiliriz:

MOV EAX, [200]
ADD EAX, [204]
MOV [208], EAX

Bir komutta işlemcinin değerlendirdiği iki parametre söz konusudur:

- İşlem genişliği
- Offset genişliği

İşlem genişliği kaç byte bilgi üzerinde işlem yapıldığı anlamına gelir. Offset genişliği ise bellek operandının kaç byte offset içerdiğini anlatır.

Özetle register bellek ve bellek sabit işlemlerinde işlem genişliği ve offset genişliğine ilişkin bütün kombinasyonlar geçerlidir. Ancak DOS altında 4 byte offset genişliğini kullanmamalıyız. Oysa korumalı modda bütün kombinasyonları rahatlıkla kullanabiliriz.

İşlem genişliği	Offset genişliği	Örnek komut	Gerçek mod	Korumalı mod
4	4	*MOV EAX, [ESI]*	✘	✔
4	2	*ADD EAX, [SI]*	✔	✔
2	4	*MOV AX, [ESI]*	✘	✔
2	2	*MOV AX, [SI]*	✔	✔
1	2	*MOV AL, [SI]*	✔	✔
1	4	*MOV AL, [ESI]*	✘	✔

Tabii offset genişliğinden bahsedebilmek için operandlardan birinin bellek olması gerekir. Örneğin:

- *MOV AX, [ECX]* komutun işlem genişliği 2 byte, offset genişliği 4 byte'tır. İşlem geçerlidir. DOS'ta tavsiye edilmez.
- *MOV EAX, [CX]* komutun işlem genişliği 4 byte'tır. Ancak bellek operandı yanlış düzenlenmiştir.
- *MOV EAX, [EBX]* komutun işlem 4 byte, offset genişliği 4 byte'tır. İşlem geçerlidir. DOS'ta tavsiye edilmez.
- *MOV dword ptr [ECX], 10* işlem genişliği 4 byte, offset genişliği de 4 byte'tır. Komut geçerlidir, DOS'ta tavsiye edilmez.
- *MOV dword ptr [SI], 10* işlem genişliği 4 byte, offset genişliği 4 byte'tır. Geçerlidir ama DOS'ta tavsiye edilmez.

Not: İşlem genişliği ve offset genişliği sırasıyla 66 ve 67 ön ekleriyle makine kodunda belirtilir.

SUB Komutu

Biçimleri:

1. reg ßà reg/mem
2. reg/mem à sabit

Etkilediği bayraklar: AF, CF, OF, PF, SF, ZF

Aslında pek çok mikroişlemcide ayrı bir çıkarma devresi yoktur. İkinci operandın ikiye tümleyeni alınır. Toplama devresine sokulur. Örneğin

MOV AL, 1C
MOV BL, 2F
SUB AL, BL

işlemi şöyle yapılır:

1C	0001 1100
2F	0010 1111
-2F(2F'nin 2'ye tümleyeni)	1101 0001

```
      1C        0001 1100
     -2F        1101 0001
      ED        1110 1101
```

Çıkartma işleminde elde edilen sonuç pozitif ya da negatif olabilir. Yani bir çıkartma işlemi sonucunda CF bayrağına bakarak soldaki operandın sağdaki operanddan işaretsiz sistemde büyük olup olmadığını anlayabiliriz. Sayıların işaretsiz sistemde olduğu varsayımıyla birinci operand ikinci operanddan büyükse CF 0, küçükse CF 1 olur.

Not: İkinci operandın 2'ye tümleyeni alınıp toplandığında aslında birinci operand ikinci operanddan büyükse CF 1, küçükse CF 0 olmaktadır. Ancak işlemci bu toplama işleminden sonra CF bayrağının tersini almaktadır. Sonuç olarak çıkartma işlemi sonucunda CF bayrağına bakarak operandların işaretsiz sistemde büyüklük-küçüklük ilişkisini kurabiliriz.

Tabii 2'ye tümleyen aritmetiğinde işaretli ve işaretsiz toplama ve çıkartma kavramları yoktur. Zaten normal bir toplama ve çıkartma işlemi hem işaretli hem işaretsiz sistemde anlamlıdır. Yani register'lar içerisine yerleştirilmiş olan sayılar işaretli kabul edilirse sonuçta işaretli yorumlanmalıdır, işaretsiz kabul edilmişse sonuçta işaretsiz yorumlanmalıdır. Örneğin:

```
MOV  AL, FE
MOV BL, 01
ADD AL, BL
```

Alà FE	İşaretsizà 254	İşaretlià -2
BLà 01	İşaretsizà 1	İşaretlià 1
ALà FF	İşaretlià 255	İşaretsizà -1

_aha_bura_

CMP Komutu

CMP komutu kullanım bakımından tamamen SUB komutu gibidir. Ancak operandlar çıkartma işleminden etkilenmezler. Yalnızca işlemden baraklar etkilenir. Örneğin:

```
CMP AX, BX
```

AX – BX işlemi yapılır ama sonuç AX register'ına atanmaz. CMP komutu yalnızca bayrakları etkiler. Genellikle CMP komutunu dallanma komutları izler. SUB ve CMP komutları bayrakları öyle etkiler ki bayraklara bakarak her türlü karşılaştırma sonucu çıkarılabilir.

SBB Komutu (Subtract with Barrow)

Bu komut SUB komutunun Carry'li versiyonudur. *SBB x, y* işleminde x - y - c işlemi yapılır. Bu komut 16 bit çalışmada iki 32 bit sayının çıkartılması işleminde kullanılmaktadır.

```
MOV  AX,  [1FC0]
SUB AX, [1FC4]
MOV [1FC8], AX
MOV AX, [1FC2]
SBB AX, [1FC0]
MOV [1FCA], AX

MOV EAX, [1FC0]
```

SUB EAX, [1FC4]
MOV [1FC8], EAX

Kalıp:

16 bit işlem genişliği ile 32 bitlik iki tamsayının toplanması önce sayıların düşük anlamlı word değerlerinin ADD ile yüksek anlamlı word değerlerinin ADC ile toplanması ile sağlanır. Çıkartma işleminde ise düşük anlamlı word değerleri için SUB yüksek anlamlı word değerleri için SBB kullanılır. Tabii 32 bit toplama ve çıkartma işlemleri 32 bit yazmaçlar ile tek hamlede yapılabilir.

MUL Komutu

Komutun biçimleri
- MUL reg/mem (8 bit)
-MUL reg/mem (16 bit)
-MUL reg/mem (32 bit)

Çarpma işleminin tek operandı vardır. Diğer operand işlemci tarafından default olarak eğer 8 bit çarpma yapılıyorsa AL, 16 bit çarpma yapılıyorsa AX biçiminde alınır. 32 bit içinse EAX default olarak alınır. Etkilediği bayraklar CF ve OF(AF, PF, SF ve ZF belirsiz)

Örnek
MOV AL, 3
MOV BL, 3
MUL BL

Çarpma sonucu eğer 8 bit çarpma söz konusu ise AX yazmacında 16 bit çarpma söz konusu ise DX:AX biçiminde yani yüksek anlamlı word DX düşük anlamlı word AX'te olacak biçimde, eğer 32 bit çarpma söz konusu ise EDX:EAX biçiminde olacaktır.

Komutun işlem genişliği operanda bakılarak anlaşılır. Operand yazmaç ise problem yoktur. Operand bellek ise operand ise genişliği byte ptr, word ptr yada dword ptr ile belirtilir. Örneğin:

MUL BX	16 bit	Geçerli	Sonuç DX:AX
MUL CL	8 bit	Geçerli	Sonuç AX
MUL EBX	32 bit	Geçerli	Sonuç EDX:EBX
MUL [offset]	8 bit	Hata	
MUL word ptr[offset]	16 bit	Geçerli	Sonuç DX:AX

Örneğin DOS altında bir çarpma işlemi için şöyle bir makine komutu üretilir.

MOV AX, a
MUL word ptr b
MOV c, AX

int türünün 32 bit olduğu sistemlerde böyle bir işlem doğrudan 32 bit yazmaçlarla yapılabilir.

16 bit sistemlerde bir grup çarpma işlemi şöyle yapılır.

```
unsigned int a,b,c,d;
d = a*b*c;

MOV AX, a
MUL word ptr b
MUL word ptr v
MOV d, AX

unsigned int a,b,c;
d = (a+b) * c;
```

işlemi şöyle yapılır.

```
MOV AX, a
ADD AX, b
MUL word ptr c
MOV d, AX
```

DIV Komutu

Komutun biçimleri
1. DIV reg/mem 8 bit
2. DIV reg/mem 16 bit
3. DIV reg/mem 32 bit

DIV komutu da tek operand alır. Bölünen operand 8 bitlik bölmede AX, 16 bit bölmede DX:AX ve 32 bitlik bölmede EDX:EAX içerisinde olmalıdır. Bölme işlemi sonucunda tamsayı bir bölüm sonucu ve yine bir tamsayı bir kalan sonucu elde edilir.

8 bitlik bölmede AL bölüm AH kalan
16 bitlik bölmede AX bölüm DX kalan
32 bitlik bölmede EAX bölüm EDX kalan biçimindedir.

Örneğin C dilinde iki tamsayıyı böldüğümüzde işlem bu makine komutu ile yapılır. Programlama dillerindeki mod operatörleri yine bu komutu kullanırlar.

16 bit bölmelerde DX yazmacının içerisinde uygun sayının bulunduğu garanti altına alınmalıdır. Örneğin C dilindeki

```
unsigned int x,y,z
z = x / y;
```

işlemi şöyle yapılabilir.

```
MOV DX, 0
MOV AX, x
DIV word ptr y
MOV z, AX
```

MUL ve DIV komutları işaretsiz çarpma ve bölme işlemleri yaparlar. Yani örneğin:

MUL BX

gibi bir işlemde işlemci AX ve BX içerisindeki sayıların işaretsiz sayıları olduğunu düşünecektir. İşaretli çarpma ve bölme işlemleri için IMUL ve IDIV komutları kullanılır. Bu komutların bütün kullanılışı biçimleri işaretsiz versiyonlarda olduğu gibidir. Yalnızca operandlar ve sonuç işaretli sistemde ele alınır.

Bit Düzeyinde İşlem Yapan Komutlar

Bu komutlar iki operandlıdır. Sayıların karşılıklı bit'leri üzerinde işlemler yaparlar.

AND Komutu

Biçimleri:
1. AND regßà reg/mem
2. AND reg/memßsabit

Örneğin:

AND AX, BX
AND AL, [SI]
AND [SI], BL
AND word ptr [SI], 7F

Komutun etkilediği bayraklar PF, SF, ZF'dir. CF ve OF her zaman sıfırlanır. AF belirsizdir.

TEST Komutu

TEST komutu tamamen AND komutuyla aynı işlemi yapar. Ancak operandlar etkilenmez, yalnızca bayraklar etkilenir.

Kalıp:

Bir sayının sıfır olup olmadığını anlamak birkaç biçimde yapılabilir. Ancak derleyicilerin tercih ettiği en etkin yöntem sayının kendisiyle AND ya da OR işlemine sokulmasıdır. Bu işlemden sonra ZF bayrağına bakılır ve karar verilir. Örneğin:

AND AX, AX

Kalıp:

Bir tamsayının tek ya da çift olup olmadığı en düşük anlamlı bit'inin 0 ya da 1 olmasıyla belirlenebilir. Bunu anlamanın en iyi yöntemi sayıyı 1 ile AND işlemine sokup ZF bayrağına bakmaktır. Örneğin:

TEST AX, 1

Aynı teknik etkin bir kod üretimi için C'de de uygulanabilir. Örneğin aşağıdaki kod

if (x % 2 == 0) {

```
}
```

yerine bu kod tercih edilebilir:

```
if (x & 1) {

}
```

Kalıp:

Bir sayının negatif ya da pozitif olduğunu anlayabilmek için yine kendisiyle AND çekip SF bayrağına bakmak gerekir. Örneğin:

```
MOV AX, mem
TEST AX, AX
```

OR Komutu

Biçimleri:
1. OR reg ßà reg/mem
2. OR reg/mem ß sabit

Karşılıklı bit'leri OR işlemine sokar.

XOR Komutu

XOR işlemi iki operand aynı ise 0, farklıysa 1 değerini veren bir işlemdir.

a	b	a XOR b
0	0	0
1	0	1
0	1	1
1	1	0

Özellikle şifreleme işlemlerinde tercih edilir.

Kalıp:

Bir sayıyı kendisiyle XOR işlemine sokarsak 0 elde ederiz. Bir register sıfırlanmak istendiğinde aşağıdaki yöntemlerden birisi kullanılabilir:

```
MOV AX, 0

SUB AX, AX

AND AX, 0

XOR AX, AX
```

Sabit içeren ifadeler makine komutunu uzattığı için elenmelidir. O halde SUB AX, AX ya da XOR AX, AX tercih edilmelidir. Geleneksel olarak XOR komutu tercih edilmelidir

XOR ve OR komutlarının etkilediği bayraklar PF, SF, ZF'dir. CF ve OF her zaman sıfırlanır. AF belirsizdir.

SHL ve SHR Komutları

Biçimleri:
1. SHL; SHR reg/mem, 1
2. SHL; SHR reg/mem, CL
3. 80186'dan sonrası için: SHL; SHR reg/mem, sabit

Komutun etkilediği bayraklar AF ve CF'dir. OF, PF, SF ve ZF belirsizdir. Öteleme sonrasında kaybedilen bit CF bayrağında saklanır. Birden fazla öteleme yapıldığında son ötelemede kaybedilen bit CF'de saklanacaktır.

80186'ya kadar öteleme işlemleri için iki makine komutu vardı: Bir kez ötelemekte kullanılan makine komutu ve CL register'ı içerisindeki değer kadar ötelemekte kullanılan makine komutu. Bu yüzden birden fazla öteleme yapılacaksa öteleme sayısı CL register'ına yerleştirilmek zorundaydı. 1 ya da CL değerleri komut yazılırken belirtilmek zorundadır. Ancak bu bilgiler makine koduna yansımaz(Yani aslında makine komutları "1 kez ötele" ve "CL kadar ötele" biçimindedir). Ancak 80186'dan sonra hiç CL register'ına yerleştirme yapmadan sabit bir sayı kadar öteleme yapmaya yarayan bir komut eklenmiştir.

SAR ve SAL Komutları

Aritmetik öteleme komutlarıdır. Aslında sola aritmetik öteleme biçiminde bir komut yoktur. SAL ile SHL komutları aslında aynı komutlardır.(Debugger'lar ve derleyiciler sanki SAL gibi bir komut varmış gibi bu komutu kabul ederler.)

Sağa aritmetik ötelemede butun bitler bir sağa kaydırılır ancak en soldan işaret biti 0 ise 0 ile , 1 ise 1 ile besleme yapılır.

Döndürme Komutları

C de döndürme işlemi yapan bir bit operatörü yoktur. Ancak makine dilinde genellikle döndürme komutları vardır. Sola ve sağa döndürme işlemleri öteleme işlemleri gibidir ancak kaybolan bit besleme işlemi için kullanılır.
örnek:

1000 0101
1100 0010 (döndürmeden sonra)

Komutların etkilediği bayraklar :CF ve OF
Döndürme işleminde dönen bit aynı zamanda CF bayrağında saklanmaktadır.

Kalıp:
Bir byte bilginin nible'larını (4 bit) yer değiştirmek için bilgi sağa yada sola 4 kez döndürülür. Örneğin AH içinde yer değiştirme yapacak olalım:

MOV CL,4
ROR AH,CL

Normal döndürme işlemine ilişkin ROR ve ROL isimli iki komut vardır. Bunlara ek olarak bu iki komutun bir de eldeli versiyonları vardır. Bu komutlar: RCL ve RCR

CF=1
AH: 1110 1010
RCR AH,1
AH:1111 0101 CF:0(sondaki bit sağa carry li döndürmede)

Eldeli versiyonlarda CF bayrağı sanki sayının en düşük anlamlı bitiymiş gibi işlem görür. Yani döndürme işlemine aktif olarak katılır.

Kalıp:
16 bit işlem genişliği kullanarak 32 bitlik bir sayıyı sola ve sağa ötelemek için düşük anlamlı word değeri ötelemek yüksek anlamlı word değeri döndürmek gerekir.

DX:AX gibi bir 32 bitlik bilgi olsun bunu sola bir otele.
SHL AX,1
RCL DX,1

Öteleme Ve Döndürme Komutlarının Biçimleri

Bütün öteleme ve döndürme komutları 3 biçimden oluşur.

1. Komut reg/mem ,1
2. Komut reg/mem,CL
3. Komut reg/mem,sabit (80186 ve sonrası için)

ROL AX,5	80186 ve sonrası için doğru
ROR word ptr [SI][BX], CL	Doğru
SAR byte ptr [SI+10],1	Doğru

Stack Kullanımı

Stack işlemcinin bilgileri geçici sure için depolamakta kullandığı RAM bölgesidir. Stack LIFO kuyruk yapısı biçiminde kullanılır.

Stack bir bölge olmasına karşın stack'in tepesi biçiminde isimlendirilen aktif bir yeri vardır. Stack' in tepesi SS:SP reg. çifti ile belirlenir. SP segment register'ı içsel olarak SS biçiminde belirlenmiştir ve değiştirilemez. Stack ile ilgili 2 işlem tanımlıdır.

1. STACK e bilgi konulması (PUSH)
2. STACK ten bilgi alınması (POP)

PUSH ve POP komutlarının biçimleri şunlardır:
1. PUSH (POP) reg/mem
2. PUSH(POP) sreg

PUSH ve POP işlemleri gerçek modda ve sanal 86 modda (yani DOS modunda) 16 bit, korumalı modda 32 bit bilgi üzerinde işlem yapar. Gerçi DOS modunda 32 bitlik stack işlemi yapılabilir ancak tavsiye etmiyoruz. Genel olarak mikroişlemcilerde PUSH ve POP

komutları işlemcinin bir register uzunluğu kadar bilgi üzerinde işlem yapacak şekilde tasarlanmaktadır. Örneğin DOS altında aşağıdaki işlemler geçerlidir.

PUSH AX
PUSH BX
PUSH word ptr [SI]
PUSH CS
PUSH BP

aşağıdakiler geçersizdir.

PUSH AL
PUSH byte ptr [SI]

Korumalı modda stack bölgesinin tepesi SS:ESP register'ı ile belirtilen bölgedir.

PUSH EAX
POP EBP

PUSH işleminde önce SP DOS modunda 2 byte korumalı modda 4 byte azaltılır. Bilgi azaltılmış değerden itibaren yerleştirilir.

Tipik bir programın çalışmasında stack için n byte yer ayrılmıştır ve SP reg' i bu bölgenin en altına çekilir. Tipik bir ".exe" program aşağıdaki gibi 3 bölgeden oluşmaktadır. Program belleğe yüklendiğinde SP reg. stack bölgesinin en altına konumlandırılır.

Kod
Data
Stack

Programın stack bölgesi başlangıçta belirlenir daha sonra büyültülemez ve küçültülemez. Aşırı derecede PUSH işlemi yapılırsa SP stack için ayrılan bölgeyi geçer buna stack overflow denir(stack'in yukarıdan taşması). Stack taşmaları işlemci tarafından otomatik olarak tespit edilemez. Stack bölgesini büyüklüğüne programcı karar verir ancak program başladığında SP reg. konumlandırılmasını yükleyici yapar.

POP Komutu

Bu işlemde SP register'ı ile belirtilen bölgeden 2 byte alınır (korumalı modda ESP'nin gösterdiği yerden 4 byte) ve SP 2 byte artırılır.

Bir kere PUSH bir kere POP yapıldığında SP eski değerine çekilmiş olur. Örneğin aşağıdaki işlemler sonrasında AX ve BX register'larının içlerindeki değerler yer değiştirilebilir.

PUSH AX
PUSH BX
POP AX
POP BX

Stack Kullanımının Amacı

Stack kullanımını 2 amacı vardır:
1. Bilgilerin geçici olarak saklanması
2. Programlama dillerinde yerel değişkenlerin saklanması ve parametre aktarılması

Biz burada yalnızca birinci kullanım amacı üzerinde duracağız.

Örneğin CX register'ı içindeki bilgiyi tutmak isteyelim. Ve elimizde hiçbir boş register kalmamış olsun. bu durumda örneğin CX register'ını zorunlu olarak bozmak durumunda kalabiliriz. (örneğin CL register'ının yüklenmesini gerektiren bir öteleme işlemi olabilir.) İşte CX içindeki bilgi geçici süre stack'te saklanabilir.

```
PUSH CX
MOV CL, 3
ROL AX,CL
POP CX
```

Bazen birden fazla register geçici süre saklanacak olabilir. Bu durumda onları ters sırada POP ile almak gerekir. Örneğin:

```
PUSH CX
PUSH AX
....
POP AX
POP CX
```

Bir bellek bölgesi içerisindeki bilginin stack'e atılması da söz konusu alabilir. Örneğin

```
PUSH word ptr [SI]
```

Böyle bir işlem özel durumlarda kullanılır.

PUSH ve POP komutları herhangi bir biçimde bayrakları etkilemez.

Stack ile ilgili diğer iki önemli komut PUSHF ve POPF komutlarıdır. Normal olarak flag register'ı birkaç özel komut dışında hiçbir biçimde kullanılamaz. (genel gösterimlerdeki reg flag ve segment register'ları haricindeki tüm register'ları, sreg ise yalnızca segment register'larını göstermektedir.)

Flag register'ı içindeki bilgiyi almak ve flag register'ına yeni bir değer yüklemek için PUSHF ve POPF komutları kullanılır.

Örneğin bayrak register'ı içine bilgi yerleştirmek için:

```
PUSH AX
POPF
```

komutları; bilgi almak için:

```
PUSHF
POP AX
```

komutları kullanılabilir. POPF komutları tüm bayrakları etkiler.

Bazen bütün register'ların stack'te saklanması gerekebilir. Örneğin bir donanım kesmesi oluştuğunda çağrılacak bir kod yazmak istesek kesme çıkışında bütün register'ların ilk konumuna getirilmesi gerekir. bunun için kesme koduna girişte bütün register'lar stack'te saklanmalı, çıkışta da hepsi geri alınmalıdır.

Bu işlemi kolaylaştırmak için 2 özel komut vardır.

PUSHA ve POPA Komutları

PUSHA komutu sırasıyla 16 bit sistemde AX, CX, DX, BX, SP, BP, SI, DI register'larını stack'e atar. POPA ters sırada geri çeker.

INC ve DEC Komutları

Bu komutlar tek operandır ve operandını tek arttırır ya da azaltır.

Biçimleri:
INC/DEC reg/mem

Komutun etkilediği bayraklar: AF, OF, PF, SF ve ZF'dir(Bu komutlar CF bayrağı üzerinde etkili olmaz, çünkü zaten elde oluştuğunda ZF bayrağıyla bu tespit edilebilmektedir).

XCHG Komutu

Biçimleri:
XCHG regßà reg/mem

İki bellek bölgesini yer değiştirmek için aşağıdaki gibi bir işlem yapabiliriz. Bu komut herhangi bir bayrağı etkilememektedir.

Kalıp:
A ve B ile ifade edilen iki bellek bölgesindeki verilerin yerlerinin değiştirilmesi aşağıdaki makine komutları dizisiyle yapılabilir:
MOV AX, A
XCHG AX, B
MOV A, AX

CBW(convert byte to word) ve CWD(convert word to double word) Komutları

Bu komutlar byte'tan word'e ya da word'den dword'e işaretli dönüşüm yapan komutlardır. Bu komutlar operandsızdır. Ancak gizlice AX ve DX register'ları üzerinde etkili olurlar. Örneğin C'de küçük tamsayı türünün büyük tamsayı türüne dönüştürülmesi işleminde bu makine komutlarından yararlanılır.

CBW komutunda dönüştürülecek sayı AL register'ının içerisine yerleştirilir. Komut uygulandıktan sonra dönüştürülmüş olan sayı AX'ten alınır. Örneğin C'de aşağıdaki gibi bir dönüşüm olsun:

```
char x = -1;
int y;

y = x;
```

derleyici aşağıdaki gibi bir kod üretecektir:

```
MOV AL, FF
CBW
```

CWD komutunda dönüştürülecek bilgi AX'e yerleştirilir. Komut uygulanır. Sonuç DX, AX'ten alınır.

Bu komutların dönüştürme dışında DX register'ını ayarlaması dışında işaretli bölme komutlarından önce kullanılmasına sık rastlanır. Örneğin C'de 16 bit iki sayıyı aşağıdaki gibi bölecek olalım:

```
int a, b, c;

a = b/ c;
```

bu işlem için DX:AX register'larının hazırlanması gerekir. DX register'ı sayının işaretine göre 00 ya da FF'lerle doldurulacaktır. İşlem aşağıdaki gibi yapılabilir:

```
MOV AX, b
CWD
IDIV word ptr c
MOV a, AX
```

Dallanma Komutları

Intel işlemcilerinde programlama dillerinde karşılaştığımız if komutlarını karşılayabilmek için bir grup dallanma komutları vardır. Intel sisteminde dallanma komutları iki gruba ayrılır:

1. Koşulsuz dallanma komutları
2. Koşullu dallanma komutları

Koşulsuz dallanma komutları tıpkı goto deyimi gibidir. Oysa koşullu dallanma komutlarında önce CMP komutuyla bir karşılaştırma yapılır, sonra bu karşılaştırmaya göre dallanma sağlanır. Aslında dallanma işleminde yapılan tek şey IP ya da EIP register'larına değer atamaktır. JMP komutları genellikle operand olarak bir sayı alır. Bu sayı akışın aktarılacağı yani IP ya da EIP register'ının alacağı değeri belirtmektedir. JMP komutları ikilik sistemde önce JMP komutunun varlığını belirten 1 ya da 2 byte sonra bir yer değiştirme(displacement) değeri alır.

Yer Değiştirme(displacement) Kavramı

Sembolik makine dilinde JMP komutlarının operandları bir sayı değil, bir etikettir. Bu etiket değerlerinin sayıya dönüştürülmesi derleyicinin görevidir. Debugger'larda durum böyle değildir. Örneğin:

JMP EXIT
....
....
EXIT:

Debugger'larda JMP işlemi için hedef adres kullanılır. örneğin:

JMP 1000

Sembolik makine dilinde ve debugger'larda böyle olmasına karşın makine dilinde JMP komutunun operandı yer değiştirme miktarı biçiminde bulunmaktadır. Yani sembolik makine dili derleyicileri ve debugger'lar işlem kodunu belirlerken yer değiştirme miktarını hesaplarlar.

İşlem kodu: Komutun makine dilindeki ikilik sistemdeki karşılığıdır. Yer değiştirme miktarının origin noktası JMP komutundan bir sonraki komutun yeridir. Yani sonraki komutun yeri 0 olmak üzere ileriye doğru yapılan dallanmalar pozitif, geriye doğru yapılan dallanmalar negatif yer değiştirme miktarı biçiminde verilir.

Koşulsuz JMP Komutu

Koşulsuz JMP komutunun 5 ayrı biçimi vardır:

1. <u>Segment içi kısa JMP(direct within segment short JMP)</u>
   Bu komut 2 byte uzunluğundadır. İlk byte EB biçimindedir. İkinci byte yer değiştirme miktarını belirtir. Bu komut ile en fazla [-128, +127] uzaklıklara dallanılabilir. Bu dallanma biçimi sembolik makine dilinde short anahtar sözcüğü kullanılarak belirtilir. Örneğin:

   JMP short EXIT
   ...
   ...
   ...
   EXIT:

   Sembolik makine dili derleyicileri ileriye doğru dallanmalarda(forward jump) short anahtar sözcüğü kullanılmamışsa dallanmanın kısa olduğunu anlayamaz. Ancak geriye doğru dallanmalarda bunu anlar.
2. <u>Segment içi doğrudan yakın JMP(direct within segment near JMP)</u>
   Bu komut 3 byte uzunluğundadır. Komutu anlatan byte E9 biçimindedir. Bu byte'ı 2 byte'lık yer değiştirme miktarı takip eder. Yani yer değiştirme miktarı [-32768, +32767] arasındadır. Bu komutla segment'in her yerine dallanılabilir. Pozitif ya da negatif yer değiştirmede segment dışına çıkılırsa sarma işlemi(wrapping) uygulanır.
3. <u>Segment içi dolaylı yakın JMP(indirect within segment JMP)</u>

Bu komut 2 byte uzunluğundadır. FF komutu anlatan byte'tır. Diğer byte register ya da bellek operandını belirtir. Örneğin:

JMP near [SI]

burada işlemci DS:SI bölgesinden 2 byte çeker ve bunu doğrudan IP register'ına yerleştirerek dallanır. Bu komutta yer değiştirme miktarı değil, offset söz konusudur. C'de fonksiyon

göstericileri bu çeşit bir dallanma işlemini akla getirir. Tabii burada JMP komutu yerine CALL komut bulunacaktır. Örneğin:

```
void (*p) (void);
p = func;
p();
```

C kodu, aşağıdaki şekilde sembolik makine dilinde ifade edilebilir:

```
MOV p, func
CALL near p
```

Dolaylı JMP işleminde register'lar da kullanılabilir. Örneğin:

```
JMP AX
```

4. <u>Segment'ler arası doğrudan JMP(direct intersegment JMP)</u>

Bu komut 5 byte uzunluğundadır. Komut EA ile başlar. Bundan sonra iki byte offset ve iki byte segment bilgisi alır. Offset IP register'ına segment CS register'ına yerleştirilir ve başka bir segment'e dallanılır. Komut sembolik makine dilinde ve debugger'larda *JMP segment:offset* biçiminde bildirilir. Örneğin:

```
JMP 0FFFF:0000
```

Makine reset edildiğinde CS:FFFF, IP:0000 adresini alır. Yani makine'yı reset etmek içinb yeterlidir.

5. <u>Segmentler arası dolaylı JMP(indirect intersegment JMP)</u>

   Bu işlem segment içi dolaylı JMP işleminin segment'ler arası version'ıdır. Yani operand olarak bir bellek bölgesini alır. Atlanacak segment ve offset değerlerini oradan çeker. Komutu anlatan byte FF'tir. Komut sembolik makine dilinde ve debugger'larda aşağıdaki gibi kullanılır:

JMP far [SI]

   Burada *JMP* anahtar sözcüğünden sonra *far* anahtar sözcüğünün getirilmesi gerekir. Yoksa işlemin segment içi mi yoksa segment'ler arası mı olduğu anlaşılamaz. Bellek operandıyla belirtilen bölgeden çekilen ilk word değer IP register'ına, sonraki word değer ise CS register'ına yerleştirilir.

Koşullu Dallanma Komutları

Bu komutlar tamamen programlama dillerindeki if komutlarının karşılığıdır. Koşullu JMP işlemi sırasında işlemci yalnızca bayrakların durumuna bakar. Yani koşullu JMP komutları bayrakların durumuna bakılarak yapılan JMP komutlarıdır. Teorik olarak SUB ya da CMP işleminden sonra bayraklar bütün işaretli ve işaretsiz karşılaştırmaları yapacak biçimde etkilenir. Koşullu JMP komutlarının çoğu SUB ya da CMP komutlarından sonra anlamlı olacak biçimde tasarlanmıştır. Bu komutlar SUB ya da CMP komutu olmadan da kullanılabilir. Ancak çoğu kez anlamsız olur.

Intel'in 16 bir işlemcilerinde koşullu JMP komutlarının işlem kodu 2 byte uzunluğundadır. Birinci byte'ı komutun kendisini, diğer byte'ı yer değiştirme miktarını (displacement) anlatır. Yani 16 bit mimaride koşullu JMP komutlarıyla en fazla bulunulan yerden 128 byte uzaklığa dallanma yapılabilir. 80386 ve sonrasında koşulu JMP komutları 2 byte ve 4 byte yer değiştirme yapılabilecek biçimde genişletilmiştir.

Eşitlik Karşılaştırması

İki değerin eşitliği CMP komutundan sonra ZF bayrağına bakılarak tespit edilebilir. Bu testi yapan makine komutu JZ/JE'dir(Bu iki komut birbirinin aynısıdır). Koşullu JMP komutları koşul sağlanmışsa belirtilen bölgeye dallanmayı sağlarlar. Koşul sağlanmamışsa bir sonraki komuttan devam ederler. Aşağıdaki C karşılaştırmasının sembolik makine dili karşılığı:

```
if (x == y)         MOV AX, x
        ifade1;     CMP AX, y
else                JZ @1
        ifade2;     İfade2
                    JMP NEXT
                    @1:
                    ifade1
                    @2:
```

Burada tipik bir if komutunun sembolik makine dilindeki karşılığı görülmektedir.

Eşitsizlik Karşılaştırması

CMP komutundan sonra JNZ/JNE komutu ile ZF bayrağı 0 ise dallanılabilir.

Kalıp:
Bellekte bulunan bir değerin 0 olup olmadığını anlamak için iki yöntem kullanılabilir.

1. Değişkeni doğrudan 0 ile karşılaştırıp ZF bayrağına bakmak,
2. Değişkeni register'a çekip kendisiyle AND ya da OR işlemine sokmak. Bu yöntem daha etkindir. Aşağıdaki if ifadesinin sembolik makine dili karşılığı:

```
if (result)         MOV AX, result
        ifade1;     AND AX, AX
else                JNZ @1
        ifade2;     İfade2
                    JMP @2
                    @1:
                    ifade1
                    @2:
```

Bazen karşılaştırma işlemlerinde ek bir takım makine komutları söz konusu olabilir. Örneğin:

```
if (x + y == a * b) MOV BX, x
        ifade1;     ADD BX, y
else                MOV AX, a
        ifade2;     MUL word ptr b
                    CMP AX, BX
                    JZ @1
                    İfade2
                    JMP @2
```

```
@1:
ifade1
@2:
```

İşaretsiz Sayıların Karşılaştırılması

İşaretsiz sayıların karşılaştırılması için SUB ya da CMP komutundan sonra CF ve ZF bayraklarına bakmak yeterlidir. Aşağıda açıklanan komutlar yalnızca bu bayraklara bakmaktadır. İşaretsiz karşılaştırma yapan komutlar ve eş değerleri aşağıda verilmiştir:

Komutlarda büyüktür için above sözcüğü, küçüktür için below sözcüğü kullanılmaktadır. Karşılaştırma işleminin anlatımı SUB ya da CMP komutunun birinci operandı dikkate alınarak kurulmuştur.

JA(jump if above)/JNBE(jump if not below or equal)
JB(jump if below)/JNAE(jump if not above or equal)
JAE(jump if above or equal)/JNB(jump if not below)
JBE(jump if below or equal)/JNA(jump if not above)

```
if (x >= y)             MOV AX, x
        ifade1;         CMP AX, y
else                    JAE @1
        ifade2;         ifade2
                        JMP @2
                        @1:
                        ifade1
                        @2:
```

İşaretli Sayıların Karşılaştırılması

İşaretli sayıların karşılaştırılmasında SUB ya da CMP komutundan sonra OF, ZF ve SF bayraklarına bakılır. Karşılaştırmanın anlatımı yine birinci operanda göre kurulmuştur. Büyüktür için greater, küçüktür için less sözcükleri seçilmiştir. İşaretli karşılaştırma komutları şunlardır:

JG(jump if greater)/JNLE(jump if not less or equal)
JL(jump if less)/JNGE(jump if not greater or equal)
JGE(jump if greater or equal)/JNL(jump if not less)
JLE(jump if less or equal)/JNG(jump if not greater)

```
if (x >= y)             MOV AX, x
        ifade1;         CMP AX,y
else                    JGE @1
        ifade2;         ifade2
                        JMP @2
                        @1:
                        ifade1
                        @2:
```

Diğer Koşullu JMP Komutları

İşaretli ve işaretsiz karşılaştırma komutlarının dışında CF, OF, PF ve SF bayraklarının durumuna göre dallanmayı sağlayan 8 koşullu JMP komutu da vardır:
JC(jump if carry)
JNC(jump if not carry)
JO(jump if overflow)
JNO(jump if not overflow)
JP(jump if parity)
JNP(jump if not parity)
JS(jump if sign flag set)
JNS(jump if not sign flag set)

Alt Programların Çağırılması

C'de bir fonksiyon çağırıldığında fonksiyonun ana bloğu bittiğinde ya da fonksiyon içerisinde return anahtar sözcüğü kullanıldığında akış çağırılma işleminden sonraki koddan devam eder. Fonksiyonların çağırılması için Intel işlemcilerinde CALL makine komutu, fonksiyondan geri dönmek için ise RET komutu kullanılır. CALL makine komutunun JMP'den tek farkı dönüş adresinin stack'te saklanmasıdır. CALL makine komutu dallanmadan önce CALL komutundan sonraki komutun adresini kendi içerisinde otomatik olarak stack'e PUSH eder. Böylece dallanma işlemi yapıldığında dönüş adresi stack'tedir. İstediğimiz zaman RET makine komutuyla o adresi stack'ten alarak geri dönebiliriz. Bu durumda

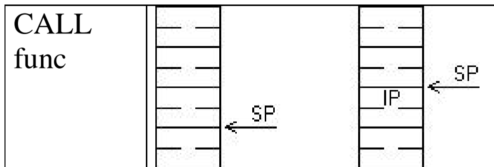
CALL adr

eşdeğerindeki bir komut

PUSH sonraki_adr
JMP adr

biçiminde yapılır. CALL makine komutunun 4 biçimi vardır:

1. <u>Segment içi doğrudan CALL(direct within segment CALL)</u>
   Bu makine komutunu E8 byte'ıyla teşhis edilir. Komutun kendisi 3 byte uzunluğundadır. 2 byte'lık bir yer değiştirme miktarı vardır. Yani segment'in her tarafına dallanılabilir. Dallanmadan önce yalnızca IP register'ı stack'e PUSH edilir. Örneğin:

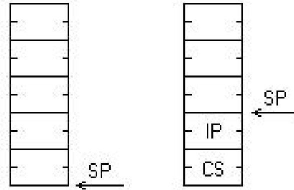
   Fonksiyondan geriye dönmek için sembolik olarak *POP IP* işleminin yapılması gerekir. Tabii POP IP biçiminde bir makine komutu yoktur. Bu makine komutunun ismi RET komutudur. Yani aslında C'de fonksiyonun ana bloğu bittiğinde derleyici bizim görmediğimiz bir RET komutu yerleştirmektedir.

2. Segment içi dolaylı CALL(indirect within segment CALL)
   Burada komutun operandı bellek ya da register'dır. Programın akışı operand ile belirtilen bellekteki adrese gider. Bu çağırma işleminin segment'ler arası dolaylı çağırma işleminden ayrılması için CALL komutundan sonra *near* anahtar sözcüğünün getirilmesi gerekir. Örneğin:

   CALL near [SI]
   CALL near [BP – 2]

   Komutu anlatan byte FF biçimindedir.

3. Segment'ler arası doğrudan CALL(direct intersegment CALL)
   Segment'ler arası CALL işleminden geri dönülebilmesi için hem CS hem de IP register'larının stack'e atılması gerekir. Komut önce PUSH CS, sonra PUSH IP işlemleriyle sonraki komutun segment ve offset bilgilerini stack'e atar ve operand biçiminde verilmiş olan CS ve IP değerlerini yükleyerek dallanmayı gerçekleştirir.



   Komut 5 byte uzunluğundadır. Sembolik makine dilinde *CALL segment:offset* biçiminde kullanılır. Örneğin:

CALL 1FC0:2C15

   Segment'ler arası CALL işleminden RETF makine komutuyla geri dönülür.

4. Segmentler arası dolaylı CALL(indirect intersegment CALL)
   Bu komutun operandı bellek ya da register'dır. Belirtilen bellek bölgesindeki düşük anlamlı word IP register'ına, yüksek anlamlı word CS register'ına çekilerek dallanma gerçekleştirir. Komutta *far* anahtar sözcüğü kullanılmalıdır. Örneğin:

   CALL far [SI]
   CALL far [BP – 2]

Alt Programdan Geriye Dönüş

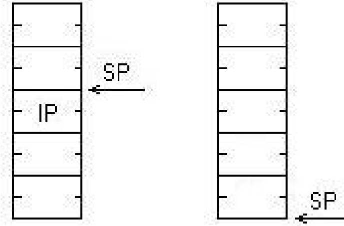
RET makine komutu CALL komutuyla stack'e atılmış olan dönüş adresini geri yükleyerek dönüşü gerçekleştirir. RET komutları segment içiyse RET ile segment'ler arası ise RETF ile ifade edilir. 4 çeşit RET komutu vardır:

1. Segment içi RET
   Bir byte uzunluğunda bir makine komutudur. Komutun işlem kodu C3'tür. Komut POP IP gibi bir işlemi gerçekleştirir.
2. Segment içi artırımlı RET

Bu makine komutu 3 byte uzunluğundadır. Komut anlatan byte C2'dir. Komut iki byte'lık bir sayı operandı almaktadır. Sembolik makine dilinde *RET n* biçiminde gösterilir. Örneğin:

RET 4

Komut önce POP IP işlemini yapar, daha sonra SP register'ını parametresinde belirtilen miktarda byte artırır. Örneğin:

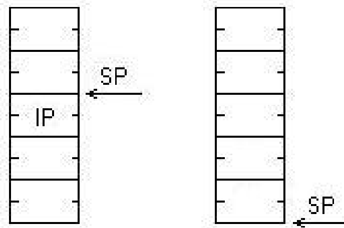


Bu komut özellikle pascal çağırma biçimini ve Windows programlamasında __stdcall çağırma biçimini gerçekleştirmek amacıyla kullanılmaktadır.

3. Segment'ler arası RET
   Sembolik makine dilinde RETF komutuyla belirtilir. Komut sırasıyla *POP IP* ve *POP CS* işlemlerini yapar ve başka bir segment'e geri döner. Komut CB değeriyle anlatılır. Komut 1 byte uzunluğundadır.

4. Segment'ler arası artırımlı RET
   Komut 3 byte uzunluğundadır ve *RETF n* biçiminde gösterilir. Komut önce RETF işlemini gerçekleştirir, daha sonra SP register'ını n kadar artırır. Örneğin:

   RETF 4 gibi bir komut çalıştırılırsa;



Bayraklar Üzerinde Özel İşlem Yapan Komutlar

Normal olarak bayrak konumlarını değiştirmek için PUSHF ve POPF makine komutlarından faydalanılabilir. Ancak bazı kritik bayraklar için onları set eden ve reset eden özel makine komutları vardır. Bu komutlar şunlardır:

CLC clear CF
STC set CF
CLI clear IF
STI set IF
CLD clear DF

STD set DF
CMC complement CF

Sembolik Makine Dili Nedir?

Debugger programlarında komut o anda girilir ve çalıştırılır. Girilen komut kümesinin kaynak kod olarak bir düzyazı dosyası içerisinde saklanması mümkün değildir. Sembolik makine dili bir derleyiciye sahip olan, makine komutlarında yazılmış bir programı exe hale getirebilen bir çalışma biçimini oluşturur. Böylelikle istenilen değişiklikle kaynak kod içerisinde yapılıp saklanabilmektedir. Sembolik makine dosyalarının uzantısı asm olarak belirlenir. Asm programı tıpkı C programı gibi derlenip link işlemine sokulmak zorundadır. Sembolik makine dili derleyicileri genellikle tek bir exe dosya biçimindedir. Borland'ın TASM, Microsoft'un MASM programları kullanılan tipik derleyicilerdir. TASM programları Borland derleyici paketlerinin içerisinde zaten bulunmaktadır. Sembolik makine dili derleyicileri geleneksel olarak bir editöre sahip değildir. Programlar herhangi bir editörde yazılıp komut satırından derlenir. Tipik bir programın derlenmesi işlemi:

TASM <dosya ismi>;

Obj modül tllink.exe ya da link.exe programlarıyla komut satırından link edilerek exe dosya elde edilir. TLINK programının tipik kullanımı şöyledir:

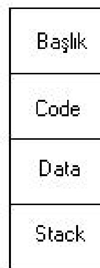
TLINK <obj dosya ismi>;

Exe Dosyanın Yapısı ve Yüklenmesi

Exe dosyaları içerisindeki bilginin organizasyonuna göre şöyle evrim geçirmiştir:

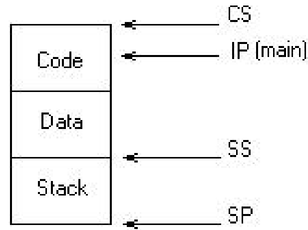
DOS à MZ formatı
Win3.x à NE formatı
Win32 à PE formatı

Bir exe dosyanın yapısı tipik olarak şöyledir:



Exe dosyanın başlık kısmında programın yüklenmesine ilişkin çeşitli bilgiler bulunur. Bir exe dosya çalıştırılacağı zaman blok olarak RAM'e yüklenir. Yükleme sırasında MZ kısmında başlık kısmı atılmaktadır. PE formatında başlık kısmıyla birlikte yükleme yapılır. Program işletim sisteminin yükleyici(loader) programı tarafından yüklenir. Program belleğe yüklendiğinde artık çalışmakta olan bir program haline gelir. Çalışmakta olan programlara process denir. Programın bütün makine komutları exe dosyanın code bölümündedir.

Program belleğe yüklendikten sonra nasıl çalışır hale getirilecektir? Programın çalışır hale getirilmesi aslında CS:IP register'larına program koduna ilişkin ilk değerlerinin verilip kontrolün bırakılmasından başka bir şey değildir. Exe dosya içerisinde her şey ikilik sistemde mikroişlemcinin anlayacağı biçimdedir. Örneğin bir C programında yüzlerce fonksiyon olabilir. Ancak C programları main fonksiyonundan çalışmaya başlar. Peki yükleyici main fonksiyonun başlangıç segment ve offset değerlerini nereden bilecektir? İşte programın başlangıç kod adresi yani CS:IP register'larına verilecek ilk değer exe dosyanın başlık kısmına linker tarafından yazılır. Yükleyici stack oluşumu için SS ve SP register'larına da ilk değer verir. SP program yüklendiğinde stack bölgesinin en sonunu gösterecek biçimdedir. Ancak yükleyici DS register'ına programın data bölgesinin başlangıç adresini otomatik olarak vermez. DS register'ı programcı tarafından uygun bir biçimde yüklenmek zorundadır. Bu durumda tipik bir exe dosya yüklendiğinde register durumları aşağıdaki gibi olacaktır.



Tipik Bir Sembolik Makine Dili Programı

Sembolik makine dili programları yalınlaştırılmış segment tanımlamalarıyla ya da ayrıntılı segment tanımlamalarıyla yazılabilir. Genel yapı DOS ve UNIX, Win32 programları arasında küçük farklılıklar göstermektedir. Biz bu bölümde tipik bir DOS programının organizasyonunu inceleyeceğiz.

Sembolik makine dilinde büyük-küçük harf duyarlılığı yoktur. Dilin karakter kümesi C'den fazla olarak @ ve ? işaretlerini kapsayacak biçimdedir. Tipik bir DOS programı önce bir bellek modelinin belirtilmesiyle başlar. Toplam 6 bellek modeli vardır:

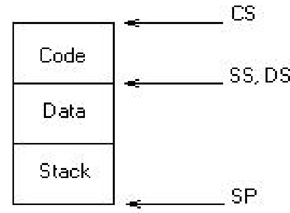
Tiny
Small
Medium
Compact
Large
Huge

Bellek modeli ileride ayrıntılarıyla ele alınacaktır. Yalnızca DOS için geçerli olan bir kavramdır. Bellek modeli şu belirlemeler konusunda etkili olur:

- Programın toplam kod data ve stack büyüklüğü
- Program yüklendiğinde segment register'ların birbirlerine göre olan durumu
- JMP ve CALL komutlarının segment içinde kalıp kalmayacağı

C'de bellek yönetimi small modeldir. Small model programlarda code bölümü 64 Kb'ı geçemez. CS register'ı code bölümünün başına çekilmiştir. Bu nedenle JMP ve CALL komutları segment içinde kalmak zorundadır. Small modelde programın data ve stack toplamı da 64 Kb'ı geçemez. Small model programlarda sembolik makine dili derleyicisi DS ve SS

register'larının aynı değerde olduğunu varsayarak kod üretirler. Bu durumda tipik bir small model program yüklendiğinde önemli register'ların durumları aşağıdaki gibi olacaktır:



Program yüklendiğinde DS hariç CS, SS ve SP register'larının değeri yükleyici tarafından verilir. Programın hemen başında DS register'ına şekilde gösterilen değeri bizim vermemiz gerekir. Bu değer aşağıdaki gibi verilebilir:

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

Bu değerin sembolik ismi @data biçimindedir. Bu durumda tipik bir sembolik makine dili programı bu iki satırla başlamalıdır. Tabii program data bölgesini kullanmak zorunda değildir. Bu durumda DS register'ının da o bölgeyi göstermesine gerek kalmaz. Bu komutla birlikte artık şekildeki register durumu elde edilmiştir.

Code, Data ve Stack Bölümlerinin Belirlenmesi

Yalınlaştırılmış segment sistemiyle programın code, data ve stack kısımları sırasıyla .CODE .DATA ve .STACK bildirimleri ile oluşturulur. Bu bölümler herhangi bir sırada yazılabilir ancak exe dosya içerisindeki yerleşim code, data ve stack biçiminde olacaktır. Normal olarak stack bölümünün yalnızca uzunluğu belirtilir. Eğer uzunluk belirtilmezse 1024 byte alınacaktır. Örneğin:

```
.STACK 512
.STACK 100h
```

Semboller

Bir code ya da data'ya ilişkin offset adresi belirten alfabetik isimlere sembol denir. Yani semboller C'deki değişkenler gibidir. Sembollere sembolik makine dilinde variable ya da label da denilmektedir. Semboller code ve data sembolleri olmak üzere ikiye ayrılır. Bir data sembolü kullanıldığında aslında [] içerisinde o sembolün offset değeri bulunan bir ifade kullanılıyormuş gibi işlem görür. Örneğin: sample sembolünün offset değeri 100 olsun:

```
MOV AX, sample
```

gibi bir kod aslında

```
MOV AX, [100]
```

anlamına gelir. Bir data sembolü + operatörüyle birlikte kullanılabilir. Örneğin:

```
MOV AX, sample + 2
```

Bu durumda sembolik makine dili derleyicisi sembolün belirttiği offset ile ilgili bir toplama yapıldığını düşünür. Yukarıdaki kod

MOV AX, [102]

anlamına gelir. + operatörüyle [] tamamen aynı anlama gelir. Örneğin:

MOV AX, sample + SI	*MOV AX, [SI + 100]*
MOV AX, sample[SI]	*MOV AX, [SI + 100]*
MOV AX, sample + [SI][BX]	*MOV AX, [SI + BX + 100]*

Özetle bir sembol aslında [] içerisinde bir sayı anlamına gelir.

Data Sembollerinin Tanımlanması

Bir programın statik ömürlü değişkenleri data segment içerisinde yer alır. Bir data sembolü .DATA bölümünde tanımlanmak zorunda değildir, ancak tipik olarak burada tanımlanmalıdır. Data sembolü tanımlamanın genel biçimi şöyledir:

Sembol	*DB(define byte)*	Değer
	DW(define word)	
	DD(define double word)	
	DQ(define quad word)	
	DT(define ten byte)	

Örneğin:

X1 db 100
X2 dw 250
X3 dd 500

Sembol tanımlama tamamen C'deki değişken tanımlaması gibidir. Sembol isminden sonra getirilen anahtar sözcükler nesnenin uzunluğunu belirtir. Değer tahsis edilen alanın içerisine yerleştirilecek sayıdır. Sembol değeri yerine ? kullanılırsa ilk değer verilmediği yani rasgele bir değer alınacağı anlatılır. Bir sembol birden fazla nesneden oluşabilir. Yani tıpkı C'deki diziler gibi olabilir. Örneğin:

X4 db 100 dup (?)
X5 db 10 dup (5)

Dup ifadesinin genel biçimi şöyledir:

n dup (val)

bu ifade ile belirtilen uzunlukta n tane tahsisat yapılır. Her elemana () içerisindeki değer verilir. Bir sembolden başlayarak birden fazla nesne oluşturulabilir. Bu durumda tanımlama işleminde her bir değer arasına , konur. Örneğin:

X6 dw 100, 200, 300, 400, 500

Özetle sembol yalnızca başlangıç yeri belirtmektedir. İster dup ifadesiyle olsun, isterse aralarına , yerleştirilmiş olsun bir sembole ilişkin değerler ardışık bir biçimde yerleştirilir.

Bir sembol derleyici tarafından obj modüle linker tarafından da exe dosyanın data bölümüne yerleştirilir. Derleyici data bölümünün ilk sembolünün offset değeri 0 olacak biçimde bütün sembollerin offset değerlerini belirler. Örneğin

```
.DATA
X1 db 100, 200, 300, 138, 163
X2 dw 200
```

Burada X1 sembolünün offset değeri 0, X2 sembolünün offset değeri 4'tür. Örneğin:

```
MOV AL, X1 + 2
```

komutu aslında derleyici tarafından aslında

```
MOV AL, [0002]
```

değerine dönüştürülür. Yani AL register'ına 138 yüklenir. Tabii derleyici yalnızca sembole uygun offset değerlerini üretir. Normal olarak DS register'ının data bölgesinin başlangıcını göstermesi gerekir. Bunun için daha sembolü kullanmadan önce

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

İşlemlerini yapmamız gerekir.

Program Yüklendiğinde Register'ların Durumları

Bir exe programı yüklendiğinde register durumları şöyle olacaktır:

- Yükleyici CS ve IP register'larının başlangıç değerleri exe dosyanın başlık kısmından alarak kendisi yükler. Linker başlık kısmına CS için code bölümünün başlangıç segment adresini yerleştirmektedir. Bir sembolik makine dili programı *end* komutuyla bitirilir. end komutunun genel biçimi şöyledir:

  ```
  end [sembol]
  ```

  end komutunun aşağısına yazılan ifadelerle derleyici ilgilenmez. end komutunun yanına bir code sembolü yazılır. Bu sembol programın başlangıç offset değerini belirtir. Bu durumda programın başlangıç IP değeri end komutuyla programcı tarafından tespit edilir. Derleyici tarafından obj modüle, linker tarafından da exe dosyanın başlık kısmına yazılır.
- SS ve SP register'larının değerleri yükleyici tarafından SS stack bölgesinin segment adresi, SP stack bölgesinin en altının offset değeri olacak biçimde yüklenir.
- DS register'ı yükleyici tarafından otomatik olarak yüklenmez, programcının yüklemesi gerekir.
- ES register'ı DOS uygulamalarında yükleyici tarafından PSP(program segment prefix) bölgesini gösterecek biçimde yüklenir.
- AX, BX, CX, DX register'ları yükleyici tarafından 0'lanır.

Programın Sonlandırılması

DOS gibi tek işlemli bir sistemde yükleyici programı yükler ve register'lara ilk değerlerini verir. Sonra kontrolü programa bırakır. Programda bir bozukluk olursa işletim sistemine geri dönüş mümkün olmaz. İşletim sistemine geri dönmek kabaca CS ve IP değerlerinin işletim sisteminin kodlarını gösterecek duruma getirilmesidir. DOS'ta kontrolü sisteme bırakmak için C'de exit fonksiyonunu çağırırız. Aslında exit fonksiyonu 21H kesmesinin 4CH numaralı fonksiyonu çağırmaktadır. Bu fonksiyon çağırılmadan önce AL register'ına exit fonksiyonunun içerisine yazdığımız değer olan programı exit code'u girilebilir. Özetle DOS'ta yazdığımız bir sembolik makine dili programının sonlandırılması şöyle yapılabilir:

```
MOV AX, 4C00h
INT 21h
```

Code Sembolleri

Bir sembol programın code bölgesinde de bir offset belirtebilir. Böyle sembollere code sembolleri denir. data ve code sembolleri ileride ayrıntılarıyla ele alınacaktır. Bir code sembolü temel olarak 3 biçimde oluşturulur:

1. JMP etiketleri biçiminde
   Örneğin:
   ```
   ....
   ....
   EXIT:
   ....
   ....
   ```
2. proc bildirimi biçiminde
3. Segment bildirimi biçiminde

proc Bildirimi

Genel biçimi:

```
sembol proc [near/far]
.....
sembol endp
```

Örneğin:

```
main proc near
.....
.....
main endp
```

Bir proc bildirimi endp bildirimiyle sonlandırılır. Sembolik makine dili programına güzel bir görünüm vermek amacıyla kullanılır. proc bildirimiyle belirtilen sembolün offset değeri bildirimin yapıldığı yerdir. proc anahtar sözcüğünden sonra near ya da far anahtar sözcüğü getirilmemişse default durum bellek modeline göre değişmektedir.

Gerçek ve Sahte Kodlar(real/pseudo)

Sembolik makine dilinde makine komutları doğrudan makine kodu olarak exe dosyaya yansır. Ancak yalnızca derleyicinin bilgi edindiği, program derlenip link edildikten sonra exe kodu içerisinde yer almayan, programı yazarken kullandığımız çeşitli yardımcı ifadeler de vardır. Makine koduna yansıyan ifadelere gerçek kod, yansımayan ifadelere sahte kod denir. Örneğin proc bildirimi bir sahte koddur.

```
/*----------ilkprog.asm----------*/
   .model small
   .CODE
   main proc near
      MOV AX, @data
      MOV DS, AX

      MOV AL, X1
      MOV X2, AL

      MOV AX, 4c00h
      INT 21h
   main endp

   .DATA
   X1 db 10
   X2 db 20

   .STACK 100h

   END main
/*----------ilkprog.asm----------*/
```

Sabitler

Sabitler makine komutlarında ya da data sembollerine ilk değer vermekte kullanılabilirler.

Sabitlerin Çeşitli Tabanlarda Gösterimleri

Sabitler doğrudan yazıldığında 10'luk sistemle yazıldığı kabul edilir. Örneğin:

name db 100

Sayının sonuna 'b' getirerek sabit ikilik sistemde yazılabilir. Örneğin:

name db 10101101b

Sekizlik sistemde yazılmış sabitler sayının sonuna 'o' ya da 'q' getirilerek belirtilir. Örneğin:

name db 135q

Sabitler default olarak 10'luk sistemde anlaşılırlar, ancak sayının sonuna 'd' getirerek de bu belirlemeyi yapabiliriz. Örneğin:

name db 100d

Sabitler 16'lık sistemde sayının sonuna 'h' getirilerek ayırt edilirler. 16'lık sistemdeki sabitler sayısal karakterlerle başlıyorsa başına '0' getirmek gerekir.

Alfabetik Sabitler(string'ler)

db uzunluklu data sembollerine ilk değer string ifadesiyle verilebilir. " ile ' arasında bir fark yoktur. Bir data sembolüne string ifadesiyle ilk değer verildiğinde derleyici belleğe string içerisindeki karakterlerin ASCII karşılıklarını yerleştirir. Örneğin:

name db "Kaan Aslan"

String'lerin sonuna C'de olduğu gibi NULL karakter eklenmez. NULL karakter eklenecekse aşağıdaki gibi yapılabilir:

name db "Kaan Aslan", 0

Not: Komut operandı oluşturulurken *+ n* gibi bir ifade *[n]* biçiminde de belirtilebilir. Örneğin:

name + 2 yerine *name[2]*, *name + SI* yerine *name[SI]* kullanılabilir.

Gerçek Sayı Sabitleri

dd, dq ya da dt uzunluklu data sembollerine noktalı sayılarla ilk değer verebiliriz. Örneğin:

n dd 3.4
n dq 3.4
n dt 3.4

Bu durumda derleyici noktalı sayıyı IEEE 754 standardına göre dönüştürerek belleğe yazar. db ve dw uzunluklu data sembollerine noktalı sayılarla ilk değer verilemez.

BCD Türden Sabitler

dt uzunluklu bir data sembolüne noktalı sayılarla değil de tam sayılarla ilk değer verilirse sayılar derleyici tarafından BCD olarak belleğe yerleştirilir. Örneğin:

n dt 1012345

Yer Sayacı(location counter)

Sembolik makine dili derleyicisi kod üzerindeki her satıra ismine yer sayacı denilen bir offset numarası karşılık getirir. Yer sayacı bir satırdaki bilginin exe dosya içerisindeki kendi segment'ine göre offset'ini belirtmektedir. Derleyici makine komutlarının uzunluklarını bildiğine göre kod bölgesindeki tüm satırların yer sayaçlarını hesaplayabilir. Sembolik makine dilinde her komut bir satıra yazılmak zorundadır. Yani ifadeleri birbirinden ayırmak için ; değil CR/LF kullanılmaktadır. Ancak sarılar tümden boş bırakılabilirler. Örneğin aşağıdaki program parçasında görünen 4 satırın da yer sayacı da aynıdır:

EXIT:

MOV AX, BX

Yani komutun şöyle yazılması aynı anlama gelir:

EXIT: MOV AX, BX

ASM Listing Dosyası

Sembolik makine dili derleyicileri derleme işlemi ister başarılı, ister başarısız olsun isteğe bağlı olarak bir listing dosyası verebilmektedir. Listing dosyası programcının analiz yapması için üretilmektedir. Listing dosyasının içerisinde şunlar bulunur:

1. Komutların işlem kodları,
2. Her satırın yer sayaç değeri,
3. Programa ilişkin diğer teknik bilgiler.

Listing dosyasını oluşturabilmek için derleme işlemi TASM ile yapılırken komut satırı ; ile kapatılmaz, /L argümanı verilir. Listing dosyasının ismi program ismiyle aynı, uzantısı lst'dir.

Diziler Üzerinde İşlemler

Bellekte ardışık bir yapı içerisinde bir döngü içerisinde dolaşabilmek için index'leme işlemi yapmak zorundayız. Bu durumda [] içerisinde SI, DI ya da BX gibi bir index register'ı olmalı. Bu index register döngüye girmeden önce dizinin başlangıç adresiyle yüklü olmalıdır. Ya da dizi elemanlarına erişmek için *[sabit + index_reg]* yapısı kullanılabilir. İşte bu biçimde döngüde her yinelenme ile index register'ın değeri gerektiği kadar arttırılır ve dizi elemanlarına erişilir. Bu bölümde çeşitli döngülerin oluşturulması, dizilerin taranması gibi temel işlemler üzerinde durulacaktır.

LEA(load effective address) Komutu

En çok kullanılan makine komutlarından birisidir. Komutun biçimi:

LEA reg ß mem

Komutun sol tarafındaki operand bir register sağ tarafındaki bir operand bir bellek olmak zorundadır. Örneğin:

LEA AX, [BX – 4]
LEA BX, [SI + BX – 10]

Bu makine komutu [] içerisindeki offset değerinin register'a yüklenmesini sağlar. Örneğin:

LEA SI, [BX + 4]

Burada BX register'ının içerisinde 1000h değeri olsun. Komut bir MOV komutu olsaydı 1004h adresinden başlayan 2 byte'lık veri SI register'ına atanırdı. Bu komut 1004 değerinin

SI register'ına atanmasına yol açar. [] içerisindeki ifadeler bellekte bir yer belirttiğine göre bu komut bir nesnenin adresinin alınmasında kullanılabilir.

Sembolik Makine Dilinde For Döngülerinin Oluşturulması

Programlama dillerinde for döngüleri n kez yinelenmeyi sağlayan deyimlerdir. For döngüleri sembolik makine dilinde iki biçimde kurulabilir. Bu iki biçim arasında ciddi bir farklılık yoktur. Bu biçimler kalıp öğrenilebilir. Döngünün yenilenme miktarı bellekte tutulabileceği gibi, bir register'da da tutulabilir. Bu durumlarda herhangi bir register kullanılabilir. Geleneksel olarak CX register'ı tercih edilir.

Kalıp:

MOV CX, 0 *JMP @1* *@2:* .... *INC CX* *@1:* *CMP CX, n* *JB @2*	0'dan n'e kadar çalışacak bir döngü

Kalıp:

MOV CX, 0 *@2:* *CMP CX, n* *JAE @1* .... *INC CX* *JMP @2* *@1:*	0'dan n'e kadar çalışacak başka bir döngü

Bu döngü kalıpları tipik bir C for döngüsünün açılımlarıdır. Burada n defa dönmek için azaltımlı yöntemler de kullanılabilir.

Kalıp:

MOV CX, n *@1:* .... *DEC CX* *JNZ @1*	*do {* .... *--n;* *} while(n)*

Bir dizinin en büyük elemanını bulan sembolik makine dili programı:

```
/*********dongu.asm**********/
  .model small

  .DATA
  ARRAY DB 1,2,3,4,5,6,7,8,9,0
  MAX DB ?

  .CODE
  MAIN:
     MOV AX, @DATA
     MOV DS, AX
     MOV SI, 1
     MOV DL, ARRAY
     JMP @1

  @2:
     CMP DL, ARRAY + SI
     JGE @3
     MOV DL, ARRAY + SI

  @3:
     INC SI

  @1:
     CMP SI, 10
     JB @2
     MOV MAX, DL
     MOV AX, 4C00H
     INT 21H

  .STACK 100H

  END MAIN
/*********dongu.asm**********/
```

Kod Sembolleri

Programın kod bölümünde offset belirten sembollere denir. Kod sembolleri iki biçimde oluşturulur:

1. Etiket bildirimiyle
2. proc bildirimiyle

Etiket bildirimiyle oluşturulan kod sembollerine ancak aynı segment içerisinde JMP ve CALL komutları uygulanabilir. Bunlar tamamen *near proc* bildirimiyle eşdeğerdir. Eğer kod sembolü proc bildirimiyle oluşturulmuşsa bu sembol JMP ve CALL komutlarıyla kullanıldığında default olarak proc bildirimi near ise segment içi, far ise segment'ler arası CALL komutu anlaşılır.

Alt Programlarla Çalışma

near proc bildiriminin etiket bildirimiyle oluşturulmuş kod sembolünden hiçbir farkı yoktur. proc bildirimini bitiren syntax ifadesi yalnızca okunabilirliği arttırmak için adeta bir alt program görüntüsü vermek için kullanılır. Yani örneğin bir proc bildirimi CALL yapılmış

olsun; programın akışı endp bildirimine geldiğinde eğer RET komutu kullanılmadıysa bir geri dönüş oluşmaz. Yani proc bildirimi RET komutunu içermemektedir.

Bu durumda bir sembolik makine dili programı tıpkı bir C programı gibi tasarlanabilir. Yani çeşitli alt programlar proc bildirimi ile yazılabilir, program da bu alt programlardan bir tanesiyle başlayabilir. proc bildirimlerinin sonuna RET makine komutu konulmalıdır. RET makine komutu proc bildirimi için değil, CALL makine komutu için kullanılmaktadır. Tipik bir sembolik makine dili programı tek parça halinde değil, C'de olduğu gibi alt programlar ve makrolar biçiminde yazılmalıdır. Özellikle makro kullanımı çok yaygındır. makro konusu ileride ele alınacaktır. Şüphesiz bir proc bildirimine JMP komutuyla da dallanılabilir. Bu durumda RET makine komutuyla rasgele bir yere geri dönülebilir. Kod sembollerine JMP ya da CALL makine komutuyla dallanırken işlemin segment içi mi yoksa segment'ler arası mı olduğu şöyle tespit edilir:

1. Kod sembolü etiket bildirimiyle oluşturulmuşsa her zaman segment içidir.
2. Kod sembolü proc bildirimiyle oluşturulmuşsa proc bildiriminin near ya da far olmasına bağlı olarak durum değişir. Ancak okunabilirliği arttırmak için kod sembolünden önce durumu vurgulamak amacıyla *near ptr* ve *far ptr* anahtar sözcükleri kullanılabilir. Örneğin:
   JMP near ptr func
   CALL near ptr func
   CALL far ptr sample

Alt Programlara Parametre Geçirilmesi ve Alt Programların Geri Dönüş Değerlerini Almak

Bir alt programa parametre aktarımının uygulamada kullanılan 3 yöntemi vardır:

1. Register'lar kullanılarak parametre aktarımı:
   Bu yöntemde alt program çağırılmadan önce parametreler, önceden belirlenmiş register'lara yazılırlar. Örneğin: 3 parametreli bir alt program söz konusu olsun; alt program birinci parametreyi AX, ikinci parametreyi BX ve üçüncü parametreyi CX register'ından alacak biçimde yazılabilir. Çağıracak kişi de çağırmadan önce bu parametreleri bu register'lara yazmak zorundadır. Bu tür parametre aktarım biçimine fastcall denilmektedir. C derleyicilerinde fonksiyonun geri dönüş değerinin sağına *fastcall*, *_fastcall* ya da *__fastcall* yazılarak belirtilir. Böyle parametre aktarımı olabilecek en hızlı yöntemdir. Bu nedenle sembolik makine dili programcıları arasında yaygın olarak kullanılır. Bu biçimdeki parametre aktarımının standart bir dokümantasyonu yoktur. Programcılar aktarımı farklı register'larla yapabilirler. Derleyicilerin fastcall çağırma biçimiyle kullandıkları register'lar da dokümante edilmemiştir. Ancak denemelerle bulunabilir. Yöntemin en kötü tarafı register yetersizliğinden dolayı az sayıda parametreye sahip olan fonksiyonlarda uygulanabilmesidir.
2. Data segment kullanılarak parametre aktarımı:
   Bu yöntemde alt program çağırılmadan önce parametreler .data bölümü içerisindeki çeşitli alanlara yazılır. Alt programlar da bu alanlardan parametreleri alacak biçimde tasarlanır. Bu yöntem C derleyicileri tarafından pek tercih edilmemektedir.
3. Stack kullanılarak parametre aktarımı:
   Bu yöntem C derleyicileri tarafından en sık kullanılan yöntemdir. pascal, cdecl, stdcall çağırma biçimleri bu yöntemi kullanmaktadır. Bu yöntemin birkaç biçimi vardır. Biz bu biçimleri C'deki karşılıklarına göre inceleyeceğiz.
   1. cdecl Yöntemi
      Bu durum DOS, Windows ve UNIX sistemlerinde derleyicilerin uyguladığı default yöntemdir. C'de fonksiyonun geri dönüş değerinin sağına *cdecl*, *_cdecl* ya da *__cdecl*

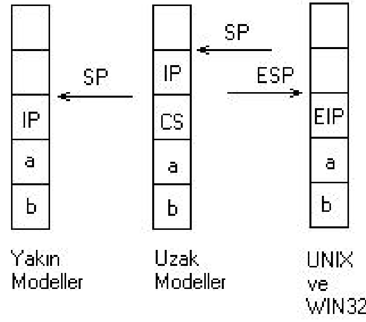
yazılarak belirtilir. C tarzı fonksiyon çağırmada parametreler sağdan sola fonksiyonu çağıran tarafından stack'e PUSH edilir. Sonra fonksiyon CALL edilir. Örneğin:

void func(int a, int b);

fonksiyonu şöyle çağırılmalıdır:

```
PUSH b
PUSH a
CALL func
```

Akışın fonksiyona geldiği noktadaki stack durumu şöyle olacaktır:



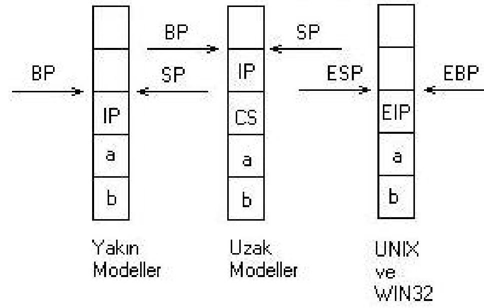
Bütün fonksiyon çağırma biçimlerinde yalnızca AX, BX, CX, DX register'ları fonksiyonu tasarlayan tarafından bozulabilir. Ancak bunun dışındaki bütün register'lar fonksiyon içerisinde bozulsa bile fonksiyondan çıkılmadan önce orijinal değerleriyle yeniden yüklenmelidir. Çünkü C derleyicileri fonksiyonu çağırmadan önce diğer register'lara belirli bilgileri yazmış olabilir ve fonksiyonu çağırdıktan sonra o bilgileri kullanacak olabilir.
Şimdi akış çağırılan fonksiyondadır ve çağırılan fonksiyon stack'ten parametreleri çekmek durumundadır. Stack bölgesini index'lemek için BP register'ı kullanılmak zorundadır. Bunun için BP register'ının da ayrıca saklanması gerekir. İşte fonksiyondaki ilk iki makine komutu DOS sisteminde

```
PUSH BP
MOV BP, SP
```

Win32 ve UNIX sistemlerindeyse

```
PUSH EBP
MOV EBP, ESP
```

biçiminde olmalıdır. Bu iki makine komutundan sonraki stack durumu aşağıdaki gibi olacaktır:



Bu durumda bu iki makine komutundan sonra artık BP register'ı çivilenmiş olur. Fonksiyonun içerisinde artık PUSH komutlarını kullanabiliriz. SP yukarı doğru hareketine devam eder. Ancak BP çivilenmiştir. İlk parametrenin yeri şöyledir:

DOS yakın modeller için	[BP + 4]
DOS uzak modeller için	[BP + 6]
UNIX ve Win32 sistemleri için	[BP + 8]

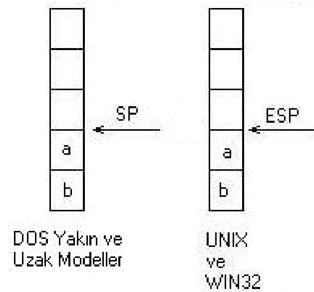
Fonksiyondan çıkarken BP register'ını geri alarak çıkmalıyız. Bunun için DOS'ta

POP BP
RET/RETF

komutları, Win32 ve UNIX sistemlerinde

POP EBP
RET

komutları kullanılır. Fonksiyondan çıkıldıktan sonra stack'in durumu aşağıdaki gibi olacaktır:



Görüldüğü gibi henüz stack eski haline gelmemiştir. Fonksiyonu çağıranın SP register'ını toplam parametre miktarı kadar artırarak stack'i dengelemesi gerekir. Stack'in dengelenmesi için DOS'ta:

ADD SP, n

UNIX ve Win32 sistemlerinde:

ADD ESP, n

makine komutları kullanılabilir. Bu durumda *void func(int a, int b);* gibi bir fonksiyonu bir C derleyicisi DOS'ta:

```
PUSH b
PUSH a
CALL func
ADD SP, 4
```

şeklinde, UNIX ve Win32 sistemlerinde

```
PUSH b
PSUH a
CALL func
ADD ESP, 8
```

çağıracaktır. Böylece bu biçimdeki parametre aktarımına uygun parametre iskeletleri de DOS için:

```
func proc near
PUSH BP
MOV BP, SP
....
....
POP BP
RET
func endp
```

UNIX ve Win32 sistemleri için:

```
func proc near
PUSH EBP
MOV EBP, ESP
....
....
POP EBP
RET
func endp
```

şeklinde olacaktır.

Biz burada çağıran kişinin C derleyicisi olduğu fikriyle standart kuralları açıkladık. Eğer programın tamamını sembolik makine dilinde yazıyorsak bir takım register'larda bilgi saklamayacaksak, örneğin BP register'ının PUSH edilmesine gerek kalmayabilir. Ancak aktarım ana hatlarıyla yine böyle olacaktır.

2. pascal Yöntemi
   pascal yöntemi kullanılarak parametre aktarımına pascal tarzı fonksiyon çağırımı denilir. Pascal derleyicileri fonksiyonları bu biçimde çağırmaktadır. C'de pascal tarzı fonksiyon çağırabilmek için fonksiyonun geri dönüş değerinin sağına *pascal*, *_pascal* veya *__pascal* yazılır. Bu aktarım biçiminde cdecl aktarım biçimine göre iki farklılık vardır:
   1. Parametreler sağdan sola değil, soldan sağa stack'e aktarılır.
   2. Stack'in dengelenmesi çağıran fonksiyon tarafından değil, çağırılan fonksiyon tarafından *RET n* makine komutuyla yapılmaktadır.

Örneğin:

void func(int a, int b);

fonksiyonunun çağırılmasında yalnızca şu işlemler yapılır:

```
PUSH a
PUSH b
CALL func
```

Fonksiyonun parametreleri kullanma biçimi yine aynıdır. Stack frame'in düzenlenmesi için

```
PUSH BP
MOV BP, SP
```

komutları gerekebilir. Ancak [BP + 4]'te ilk parametre değil, son parametre bulunacaktır. Fonksiyondan çıkış

```
POP BP
RET n
```

makine komutlarıyla yapılır. Görüldüğü gibi stack'in dengelenmesi çağırılan fonksiyon tarafından yapılmaktadır. Bu durumda

void pascal func(int a, int b);

gibi bir fonksiyonun iskelet kodu

```
func proc near
PUSH BP
MOV BP, SP
...
...
POP BP
RET 4
func endp
```

şeklinde olacaktır.

3. <u>stdcall Yöntemi</u>
   Bu yöntem Win32 API fonksiyonlarının çağırılmasında kullanılan yöntemdir. Ayrıca C++'ta sınıfın üye fonksiyonları da default olarak bu biçimde çağırılmaktadır. Bu çağırma biçimi DOS ve UNIX sistemlerinde desteklenmemektedir. Win32 sistemlerinde bu çağırma biçiminin kullanılabilmesi için fonksiyonun geri dönüş değerinin sağına *stdcall*, *_stdcall* ya da *__stdcall* yazılır. Ayrıca Win32 sistemlerinde *WINAPI* ve *CALLBACK* makroları da *__stdcall* anlamına gelmektedir.

   Bu çağırma biçimi cdecl ile pascal çağırma biçimlerinin bir karışımıdır. Yani parametreler sağdan sola stack'e atılır ama stack'in dengelenmesi çağırılan fonksiyon tarafından *RET n* komutuyla yapılır.

Parametrelerin sağdan sola aktarılması *printf*, *scanf* gibi değişken sayıda parametre alan fonksiyonların tasarımını mümkün hale getirmektedir. Stack'in çağırılan fonksiyon tarafından *RET n* komutuyla dengelenmesi diğer yönteme göre daha hızlıdır. Bu yüzden stdcall çağırma biçimi en iyi biçimdir.

Fonksiyonların Geri Dönüş Değerlerinin Oluşturulması

Çağırma biçimi ne olursa olsun fonksiyonların geri dönüş değerleri fonksiyon çağırıldıktan sonra register'lardan alınır. Fonksiyonu yazanın geri dönüş değerini uygun register'larda bırakarak çıkması gerekir. Kurallar şöyledir:

1. DOS altında çalışmada
   a. Geri dönüş değerleri 2 byte olan tüm fonksiyonların geri dönüş değerleri AX register'ında bırakılmalıdır.
   b. Geri dönüş değerleri 4 byte olan fonksiyonların geri dönüş değerleri DX:AX register'larında bırakılmalıdır.
2. Win32/UNIX altında çalışmada
   a. Geri dönüş değerleri 2 byte olan fonksiyonların geri dönüş değerleri EAX register'ının AX kısmında bulunmalıdır.
   b. Geri dönüş değerleri 4 byte olan fonksiyonların geri dönüş değerleri A;EAX register'ının içerisinde bırakılmalıdır.

Her iki sistemde de geri dönüş değeri 1 byte olan fonksiyonlar AL register'ını kullanmaktadır. Ayrıca geri dönüş değeri float, double ya da long double olan fonksiyonların da geri dönüş değerleri matematik işlemcinin stack register'larında tutulmaktadır.

Fonksiyon Çağırmalarına İlişkin Çeşitli Örnekler

1. Aşağıdaki C fonksiyonun sembolik makine dili karşılığını yazınız.

int Add(int a, int b);

DOS yakın modeller için

```
_add proc near
        PUSH BP
        MOV BP, SP
        MOV AX, [BP + 4]
        ADD AX, [BP + 6]
        POP BP
        RET
_add endp
```

Win32/UNIX için

```
_add proc near
        PUSH EBP
        MOV EBP, ESP
        MOV EAX, [EBP + 8]
        ADD EAX, [EBP + 12]
        POP EBP
        RET
_add endp
```

2. Gösterici parametresi alan fonksiyonlar için aşağıdaki örnek verilebilir.

*int total(int *pArray, int nSize);*

DOS yakın modeller için

```
total proc near
        PUSH BP
        MOV BP, SP
        MOV BX, [BP + 4]
        XOR AX, AX
        XOR CX, CX
        JMP @1
@2:     ADD AX, [BX]
        ADD BX, 2
        INC CX
@1:     CMP CX, [BP + 6]
        JL @2
        POP BP
        RET
total endp
```

Win32/UNIX için

```
total proc near
        PUSH EBP
        MOV EBP, ESP
        MOV EBX, [EBP + 8]
        XOR EAX, EAX
        XOR ECX, ECX
        JMP @1
@2:     ADD EAX, [EBX]
        ADD EBX, 2
        INC ECX
@1:     CMP ECX, [EBP + 12]
        JL @2
        POP EBP
        RET
total endp
```

Sembolik Makine Dilinde Yazılan Fonksiyonları C'den Çağırılması

Linker programı birden fazla obj modülü bir exe halinde birleştirecek biçimde yazılmıştır. Bu yüzden C ve ASM dosyaları ayrı ayrı derlenir, her ikisi birden standart C kütüphaneleri kullanılarak link edilir. Modüller halinde çalışma daha sonra ileride ele alınacaktır. Birden fazla modülle link işlemi için bilindiği gibi make ya da proje dosyası oluşturulur. Proje dosyası içerisinde C dosyası ile sembolik makine dilinde derlediğimiz obj dosyası bulunmalıdır. Örneğin util.asm dosyası içerisinde fonksiyonlarımız olsun; biz bu fonksiyonları den.c içerisinden çağıracak olalım. Util.asm derlenmeli ve proje dosyası içerisinde den.c ile util.obj olmalıdır. Program exe'ye dönüştürüldüğünde C'de yazdıklarımızla sembolik makine dilinde yazdıklarımız aynı exe dosya içerisinde birleştirilmiş olur.

Birleştirme sırasında çıkacak problemler

1. Derleme işleminde büyük/küçük harf duyarlılığını sağlamak için /mx seçeneği kullanılmalıdır. Örneğin:
   TASM /mx util.asm;
2. Bir fonksiyonun diğer modüllerden çağırılabilmesi için public bildiriminin yapılması gerekir. Public bildirimi herhangi bir yerde yapılabilir ancak en iyi yer programın sonunda yapılmasıdır. Public bildirimi şöyle yapılır:
   public <fonksiyon ismi>
3. Programın stack tanımlaması C derleyicisi tarafından C modülünde zaten yapılmıştır. Bu yüzden ASM modülünde stack tanımlamasına gerek yoktur. ASM modülü statik data kullanmayacaksa data modülüne de gerek yoktur.
4. C derleyicileri C'de yazıldığının anlaşılması için fonksiyon isimlerinin başına "_" getirerek obj modülüne yazmaktadır. Yani örneğin biz C'de printf fonksiyonunu çağırdığımızda derleyici fonksiyonu _printf ismiyle obj modülüne yazar. Biz de sembolik makine dilinde yazacağımız fonksiyonları C'den çağırabilmek için onların başına "_" getirmeliyiz.
5. Fonksiyonlar C'den çağırılırken prototip bildiriminin bulundurulması gerekir. C derleyicisi çağırılan fonksiyonun hangi register'dan alınacağını fonksiyon prototipine bakarak anlamaktadır. Fonksiyon prototipi yazılmazsa geri dönüş değerinin int olduğu varsayılır ve geri dönüş değeri AX register'ından alınacak biçimde kod üretilir.

Sınıf çalışması:

```
/*****dosutil.asm*****/
  .MODEL small

  .CODE

  _add proc near
     push   bp
     mov    bp, sp

     mov    ax, [bp + 4]
     add    ax, [bp + 6]

     pop    bp
     ret
  _add endp

  _multiply proc near
     push   bp
     mov    bp, sp

     mov    ax, [bp + 4]
     mul    word ptr [bp + 6]

     pop bp
     ret
  _multiply endp

  public _add
  public _multiply
```

```
    end
/*****dosutil.asm*****/
```

```
/*****dostest.c*****/
    #include <stdio.h>

    int multiply(int a, int b);
    int add(int a, int b);

    void main(void)
    {
        printf("%d\n", add(10, 20));
        printf("%d\n", multiply(10, 20));
    }
/*****dostest.c*****/
```

İç İçe Döngüler

İç içe döngüler için .data bölgesinde semboller tanımlayarak döngü değişkeni problemi çözülebilir. Ya da stack kullanılarak döngü değişkenlerinin karışması engellenebilir. Örneğin n defa dönen iç içe iki döngü stack yöntemiyle şöyle tasarlanabilir: Burada her iki döngü değişkeni de CX olarak düşünülmüştür.

Assembly	C
XOR CX, CX *JMP @1* *@2:* *PUSH CX* *XOR CX, CX* *JMP @3* *@4:* .... *INC CX* *@3:* *CMP CX, n* *JL @4* *POP CX* *INC CX* *CMP CX, n* *JL @2*	*for (i = 0; i < 10; ++i)* *for (j = 0; j < 10 ; ++j) {* *}*

Kalıp:

Bellekteki iki elemanın yerlerinin değiştirilmesi:

```
MOV AX, M1
MOV BX, M2
MOV M2, AX
MOV M1, BX
```

Kalıp:
Bellekteki iki elemanın yerlerinin değiştirilmesi:

```
MOV AX, M1
XCHG AX, M2
MOV M1, AX
```

Kalıp:
Bellekteki iki elemanın yerlerinin değiştirilmesi:

```
PUSH M1
PUSH M2
POP M1
POP M2
```

Win32 ve UNIX Sistemlerinde Sembolik Makine Dilinde program Yazarken Dikkat Edilmesi Gereken Hususlar

Win32 ve UNIX sistemlerinin doğal çalışma modu 32 bit data ve 32 offset kullanılan moddur. Hem DOS'ta hem de Win32/UNIX sistemlerinde 8 bit'lik, 16 bit'lik, 32 bit'lik register işlemleri yapılabilir. Ancak 66H ve 67H prefix'lerinden dolayı DOS sisteminde 32 bit register'ların UNIX ve Win32 sistemlerinde 16 bit register'ların kullanılmasında makine komutlarının uzamasından kaynaklanan bir maliyet söz konusudur. Yani Win32 ve UNIX sistemlerinde 16 bit register'ların kullanılmasını alışkanlık haline getirmemeliyiz. DOS'ta 32 bit register'ların kullanılmasının zarardan çok faydası vardır.

Win32 sistemlerinde programın çalışması sırasında segment değeri 0'dır. Bütün belleğe 32 bit offset bilgisiyle erişilebilir. Bu sistemlerde segment register'ların kullanılmasının normal kullanım bakımından bir önemi yoktur.

32 bit programlamada offset oluşturmak için bütün register'ların [] içerisinde kullanılabildiği unutmamak gerekir.

Ayrıca 32 bit programlamada [] içerisinde bir register 1, 2, 4 ya da 8 ile çarpılabilir. Bu özellik sayesinde dizi elemanlarına erişim oldukça kolaylaştırılmıştır.

Win32 sistemlerinde kullanılan obj modül formatına COFF(common object file format) denir. Dolayısıyla obj dosyaların bu formatta olması gerekir. Ancak Win32 linker programları eski OMF(object module format) formatından COFF formatına dönüşüm yapmaktadır. Örneğin Win32 sistemleri için

```
int sum(const int *pArray, int size);
```

prototipine uygun assembler kodu Win32 için

/*****sum.asm*****/

```
.386
.model flat
.code
```

```
_sum proc near
    push    ebp
    mov     ebp, esp
    xor     eax, eax
    xor     ecx, ecx
    mov     ebx, [ebp + 8]
    jmp     @1
@2:
    add     eax, [ebx + ecx * 4]
    inc     ecx
@1:
    cmp     ecx, [ebp + 12]
    jl      @2
    pop     ebp
    ret
_sum endp
public _sum

end
```

/*****sum.asm*****/

olabilir.

*void square(int x, int *pResult);*

fonksiyonu Win32 için

```
_square proc near
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp + 8]
    imul    eax, [ebp + 8]
    mov     ebx, [ebp +12]
    mov     [ebx], eax
    pop     ebp
    ret
_square end
```

şeklinde yazılabilir.

Sembolik Makine Dilinde cdecl Çağrımına Uygun Fonksiyon Yazımına Örnekler

1. *size_t asm_strlen(const char *str);* fonksiyonunun yazımı.

```
/*****util16.asm*****/
   .model small
   .code

   _asm_strlen proc near
      push     bp
      move     bp, sp
      xor      ax, ax
      mov      bx, [bp + 4]
      jmp      @1
   @2:
      inc      ax
      inc      bx
   @1:
      mov      dl, [bx]
      and      dl, dl
      jnz      @2:
      pop      bp
      ret
   _asm_strlen endp

   public _asm_strlen

   end
/*****util16.asm*****/
```

```
/*****test16.c*****/
   #include <stdio.h>

   size_t asm_strlen(const char *str);

   void main(void)
   {
      char s[80];

      gets(s);
      printf("%d\n", asm_strlen(s));
   }
/*****test16.c*****/
```

2. İki int türünden değişkenin adresini alarak onları yer değiştiren swap fonksiyonunun yazımı(*void swap(int *, int *);*).

```
/*****util16.asm*****/
  .model small
  .code

  _swap proc near
    push    bp
    mov     bp, sp
    push    si
    mov     bx, [bp + 4]
    mov     si, [bp + 6]
    mov     ax, [bx]
    xchg    ax, [si]
    mov     [bx], ax
    pop     si
    pop     bp
    ret
  _swap end

  public _swap

  end
/*****util16.asm*****/
```

```
/*****test16.c*****/
  #include <stdio.h>

  /*Prototypes*/
  void swap(int *p1, int *p2);

  /*Main*/
  void main(void)
  {
    int a = 10, b = 20;

    swap(&a, &b);
    printf("%d %d\n", a, b);
  }
/*****test16.c*****/
```

Bilindiği gibi AX, BX, CX, DX register'larının dışındaki register'lar kullanılmak istendiğinde fonksiyon çıkışında orijinal değerleriyle yüklenmelidir. Saklanacak register'lar stack frame düzenlendikten sonra, yani

PUSH BP
MOV BP, SP

komutlarından sonra stack'e PUSH edilebilir. Stack frame'in düzenlenmesinden sonra PUSH işlemi parametreleri çekme konusunda bir değişikliğe yol açmaz. Örneğin SI register'ını kullanacak olalım. Tipik bir fonksiyonun giriş kodu

PUSH BP

MOV BP, SP
PUSH SI

şeklinde olmalıdır ve çıkış kodu da

POP SI
POP BP
RET

şeklinde olacaktır.

3. n faktoriyeli alan fonksiyonu yazınız(*long factorial(int n);*).

```
/*****util16.asm*****/
  .model small
  .code

  _factorial proc near
     push      bp
     mov       bp, sp
     xor       dx, dx
     mov       cx, 1
     mov       ax, 1
     jmp       @3@1
  @3@2:
     mul       cx
     inc       cx
  @3@1:
     cmp       cx, [bp + 4]
     jle       @3@2
     pop       bp
     ret
  _factorial endp

  public _factorial

  end
/*****util16.asm*****/
```

```
/*****test16.c*****/
  #include <stdio.h>

  long factorial(int n);

  void main(void)
  {
     printf("%ld\n",
  factorial(12));
  }
/*****test16.c*****/
```

4. Aşağıdaki C kodunun sembolik makine dili karşılığını yazınız.

C	Sembolik makine dili
void disp(void);	*.data*
int add(int a, int b);	*x dw ?*
	y dw ?
int x, y;	*str1 db "%d\n", 0*
	str2 db "%d %d\n", 0
void main(void)	
{	*.code*
printf("%d\n", add(10, 20));	
x = 50;	*_main proc near*
y = 100;	*push push bp*
disp();	*mov bp, sp*
}	*mov ax, 20*
	push ax
void disp(void)	*mov ax, 10*
{	*push ax*
printf("%d %d\n", x, y);	*call _add*
}	*add sp, 4*
	push ax
int add(int a, int b)	*lea ax, str1*
{	*push ax*
return a + b;	*call _printf*
}	*add sp, 4*
	mov word ptr x, 50
	mov word ptr y, 100
	call _disp
	pop bp
	ret
	_main endp
	_disp proc near
	push word ptr y
	push word ptr x
	lea ax, str2
	push ax
	call _printf
	add sp, 6
	ret
	_disp endp
	_add proc near
	push bp
	mov bp, sp
	mov ax, [bp + 4]
	add ax, [bp + 6]
	pop bp
	ret
	_add endp
	end

C Derleyicilerin Sembolik Makine Dili Çıktıları

Neredeyse tüm C derleyicileri istendiğinde sembolik makine dilinde çıktı verebilir. Borland C++ 3.1'de "Options\Compiler\Code Generation" mönüsü altında "Generate Assembler Source" seçeneği ile derleyici sembolik makine dili çıktısı üretebilir. Derleyici obj yerine asm dosyası üretecektir. TC 2'de sembolik makine dili çıktısı için komut satırı

versiyonu kullanılır. bunun için –s seçeneği kullanılmalıdır(tcc –S dosya.asm). aynı işlem Borland C++ 3.1 ile komut satırında da yapılabilir(bcc –S dosya.asm). Borland C++ 4.0'dan sonra 32 bit sembolik makine dili çıktısı elde etmek için bcc32.exe programı kullanılır. Visual C++ derleyici ortamında sembolik makine dili çıktısı elde etmek için "Project\Settings\C/C++\Listing Files\Listing File Type" seçilebilir. Visual C++ derleyici ortamı 2.0 sürümünden itibaren 16 bit kod üretmemektedir.

OFFSET ve SEG Operatörleri

Bu iki operatör operand olarak bir sembol ismi alır. Bir sembolün offset bilgisini programın çalışma zamanı sırasında LEA komutu ile çekebiliriz. Örneğin:

LEA AX, name

Ancak bir sembolün offset adresini daha derleme aşamasında derleyiciden öğrenebiliriz. Çünkü bütün sembollerin offset adresleri derleme sırasında hesaplanabilir. Offset operatörü şöyle kullanılır:

OFFSET sembol

Bu operatör ilgili sembolün offset adresinin elde edilmesi için kullanılır. Bu operatörün ürettiği değer sabit bir sayıdır. Bu durumda aşağıdaki iki komut tamamen eşdeğer işlevlere sahiptirler:

LEA AX, name	*MOV AX, OFFSET name*

Benzer biçimde SEG operatörü de sembolün içinde bulunduğu segment'in değerini sabit olarak elde eder. Örneğin:

MOV AX, SEG name

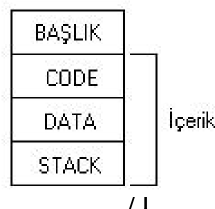
NOT: OFFSET ve SEG gibi operatörler okunabilirlik açısından büyük harfle yazılmalıdır.

EXE Dosya Formatı

Exe dosya içerik bakımından çeşitli formatlarla belirtilir. DOS'ta kullanılan exe dosya formatına MZ formatı denir. Windows 3.1'in exe formatına NE, Win32 exe formatına PE dosya formatları denir. Formatı ne olursa olsun tipik bir exe dosya iki kısımdan oluşur:

1. Başlık
2. İçerik

Başlık exe dosyanın hemen başından başlar ve değişken uzunlukta olabilir. İçerik kısmında programımızın kod, data ve stack kısmını oluşturan gerçek makine komutları bulunmaktadır. DOS exe dosyası başlık kısmı atılarak belleğe yüklenir. Halbuki NE ve PE formatları başlık kısmıyla belleğe yüklenmektedir.



Exe Dosyası Başlık Kısmı

Başlık bölümü linker tarafından obj modüllerden elde edilen bilgilerden oluşturulmaktadır. DOS exe başlık kısmında programın yüklenmesine ilişkin çeşitli bilgiler vardır. Yükleyici ancak bu bilgilerden faydalanarak programı yükleyebilir. DOS exe başlık formatı aşağıdaki gibidir.

Offset	**Uzunluk**	**Açıklama**
0(0)	word	Magic number(4D5A)
2(2)	word	Son sektörde kalan byte uzunluğu
4(4)	word	Başlık dahil dosya sektör uzunluğu
6(6)	word	Relocation elemanlarının sayısı
8(8)	word	Başlık kısmının paragraf uzunluğu
A(10)	word	Yükleme için gerekli minimum paragraf sayısı
C(12)	word	Yükleme için gerekli maksimum paragraf sayısı
E(14)	word	SS değeri
10(16)	word	SP değeri
12(18)	word	Checksum
14(20)	word	Programın başlangıç IP değeri
16(22)	word	Programın başlangıç CS değeri
18(24)	word	Relocation tablosunun yeri
1A(26)	word	Overlay sayısı
1C(28)	varies	
varies	varies	
varies	varies	
varies	varies	
varies	varies	

Magic number	Burada MZ karakterlerinin ASCII numaraları olan 4D5A bulunmalıdır. DOS işletim sistemi programı yüklerken bu değeri kontrol etmez ama başka işlemlerde etmektedir.

2 numaralı offset'te bulunan bilgi başlık + içerik toplamının 512'ye bölümünden elde edilen kalanı verir. 4 numaralı offset'te ise başlık + içerik toplamının kaç tane 512'lik sayfaya sığacağı bilgisi vardır. Özetle yükleyici başlık + içerik uzunluğunu aşağıdaki gibi hesaplar:

*Başlık + içerik = 4 numaralı offset * 512 + (512 - 2 numaralı offset değeri) % 512;*

Buradan hesaplanan uzunluk exe dosyanın gerçek uzunluğu değil, dosya içerisindeki makine kodlarının uzunluğudur(başlık + içerik). Yükleyici burada belirtilen uzunluktan başlık kısmının uzunluğunu çıkartır, yüklenecek kısmı tespit eder. Yani biz exe dosyanın sonuna bir şeyler eklesek yükleyici eklenen bilgileri yüklemez. Çünkü yükleyici orada yazılan değer üzerinden hareket eder.

Debugger gibi programlar debug altında okunabilirliği arttırmak için sembol tablosu denilen bir tablo kullanırlar. Sembol tablosu linker tarafından exe dosyanın sonuna eklenir. Yükleyici eklenen bu bilgiyi yüklemez. Yani exe dosyanın sonuna bir şeyler eklesek programın yüklenmesi sırasında problem çıkmayacaktır.

Relocation elemanlarının sayısı	Bu konu ileride ele alınacaktır.
Başlık kısmının paragraf uzunluğu	Bir paragraf 16 byte'tır. Burada başlık kısmının uzunluğu verilmektedir. Başlık kısmının uzunluğu burada verilen sayının 16 ile çarpımıdır. Artık yükleyici başlık uzunluğunu da bildiğine göre başlık kısmını atarak programı yükleyebilir hale gelmiştir.
Yükleme için gereken minimum ve maksimum paragraf sayıları	Buradaki bilgiler gerçekte fazlaca kullanılmamaktadır. Genellikle minimum 0 ve maksimum 0xFFFF biçimindedir. Yüklemek için gereken minimum bellek dosyanın yüklenmesinden sonra kesinlikle tahsis edilmesi gereken ekstra bellek miktarını belirtir. Örneğin burada 100 sayısı yazıyor olsun. Program belleğe yüklendikten sonra sanki programın alanıymış gibi 1600 byte daha bellekten tahsis edileceğini belirtir. Buradaki istek karşılanmazsa yükleyici başarısız olur, mesaj vererek programı sonlandırır. Yükleme için gerekli maksimum paragraf sayısı yükleyicinin ne kadarı mümkünse o kadarını karşılaması gerektiği değerdir. Burada genellikle 0xFFFF değeri bulunur. Bu değer 16 ile çarpılırsa 1 MB eder. Durumdan çıkan sonuç: minimum 0 ve maksimum 0xFFFF ise(hemen her zaman böyledir) yükleyici programı belleğe yükledikten sonra geri kalan bütün alanı tahsis etmektedir.
SS değeri	Burada program yüklendiğinde SS segment register'ının alacağı değer yazmaktadır. Aslında burada yazılan değer gerçek segment değeri değildir. İçerik kısmını 0 olmak üzere göreli bir değerdir. Linker obj modülden .CODE ve .DATA bölümlerinin uzunluklarını elde eder. SS register'ının göreli değerini başlık kısmına yazar. Yükleyici de programı nereye yüklediyse o değeri başlıkta belirtilen SS değeriyle toplar ve SS register'ına atar.
SP değeri	Burada program yüklendikten sonra yükleyici tarafından SP register'ına atanacak değer bulunmaktadır. Bu bilginin yazılması şöyle olmaktadır: Sembolik makine dili programcısı .STACK bildirimiyle stack'in uzunluğunu verir. Bu uzunluk obj modüle yazılır. Linker bu bilgiyi alarak exe dosyanın başlık kısmına yazar.
Checksum	Exe dosyanın bozulmuş olup olmadığını tespit etmek amacıyla linker bu bölgeye bir checksum değeri yazar. Ancak yükleyici yüklerken bu checksum değerini kontrol etmemektedir.
CS ve IP değerleri	Burada programın başlangıç CS ve IP değerleri bulunmaktadır. Tabii CS değeri exe dosyanın içerik kısmına göre göreli bir değerdir. DOS altında çalışan pek çok virüs kodu burada belirtilen CS:IP değerleriyle ilgilenir.
Relocation tablosunun yeri	Burada başlık kısmında bulunan relocation tablosunun yeri belirtilmektedir. Relocation kavramı ileride ele alınacaktır.
Overlay sayısı	Overlay çalışma ile ilgilidir. Burada genellikle 0 sayısı bulunur.

PS P(Program Segment prefix)

DOS'un yükleyicisi bir exe ya da com dosyayı yüklerken önce 256 byte uzunluğunda ismine PSP denilen bir alan oluşturur. Dosyayı hemen bu alanın altına yükler. Yani RAM'de yüklenmiş olan bir programın hemen yukarısında 256 byte'lık bir PSP bloğu vardır. PSP'nin

içerisinde programın sonlanmasına ilişkin ve çeşitli olaylara ilişkin bilgiler vardır. PSP iki tane 128 byte'lık bölgeden oluşmaktadır. Birinci 128 byte'lık bölge yükleme bilgilerini içerir. Buradaki bilgilerin bazıları ileride ele alınacaktır(Zaten bu bölümdeki bazı bilgiler eskiden undocumented biçimindeydi). PSP'nin ikinci 128 byte'lık bölümünde programın komut satırı argümanları bulunmaktadır. DOS'un yükleyicisi bir program çalıştırıldığında önce PSP'yi yaratır. Sonra programı PSP'nin altına yükler. Programın komut satırı argümanlarını da PSP'nin ikinci 128 byte'lık bölümüne yerleştirir. Bu bölümde komut satırı argümanlarından sonra bulunan kısımlarda rasgele değerler bulunur. Yani biz saf bir sembolik makine dili programı yazacaksak komut satırı argümanlarını PSP'den almak zorundayız. C'de derleyicinin başlangıç kodu(startup code) komut satırı argümanlarını PSP'den alır, bunu .DATA bölümüne çeker ve ayrık string'lerin başlangıç adreslerini yine .DATA bölümündeki bir alana yerleştirir. Sonra argv ve argc değerlerini PUSH ederek main fonksiyonunu çağırır. main içerisinde kullandığımız argv dizisi aslında .DATA bölümündedir. Komut satırı argümanlarının PSP'deki görüntüsü aşağıdaki gibidir:

Yani önce bir byte uzunluk, sonra komut satırı argümanlarına ilişkin yazının tamamı, sonra 0x0D numaralı ENTER karakteri, en son da rasgele karakterler.

Peki biz PSP'nin başlangıç adresini nasıl öğrenebiliriz? İşte yükleyici exe dosyayı yükledikten sonra DS ve ES register'ları yükleyici tarafından PSP'nin segment adresini içerecek biçime getirilir. Yani bir program böylece bellekte 16'nın katlarına yüklenmek zorundadır. PSP'nin adresine 256 toplarsak programın yükleme adresini buluruz. Bu adres exe dosyanın başlık kısmı atıldığında yüklenecek içerik kısmının başlangıç adresidir. Saf sembolik makine dili programı yazarken içeride PSP adresine gereksinim duyabiliriz. Bunun için kodun hemen başında PSP segment adresini DS ya da ES segment register'ından alarak .DATA bölgesinde bir yerde saklamak uygun olabilir. DOS altında çalışan C derleyicilerinde dos.h header dosyasında bulunan *_psp* isimli değişken PSP'nin segment adresini vermektedir. Bu durumda biz bir C programı içerisinde de *_psp* değişkeninden faydalanarak programın yükleme adresini elde edebiliriz.

Yükleme Sonrasında Register'ların İlk Konumları

- CS, IP, SS ve SP değerleri exe başlık kısmından alınarak yüklenir.
- DS ve ES PSP'nin segment adresiyle yüklenir.
- AX, BX, DX, SI, DI ve BP 0 olarak yüklenir.
- CX register'ının içerisindeki bilgi herhangi bir biçimde olabilir. Undocumented bir bilgidir.

Derleyicilerin Başlangıç Kodları(Startup Module)

Her programlama dilinde programın çalışmaya başladığı bir nokta vardır. Örneğin C'de programın başlangıç noktası main fonksiyonudur. Ancak gerçekte process'in çalışmaya başladığı nokta yani exe dosyanın içerisinde belirtilen nokta main fonksiyonu değildir. Gerçekte program derleyicinin başlangıç kodu denilen bir giriş kodundan başlar, main fonksiyonu oradan çağırılır. Derleyicinin başlangıç kodunda işletim sisteminden işletim

sistemine değişebilecek çeşitli kritik işlemler yapılmaktadır. Örneğin DOS'ta komut satırı argümanları PSP içerisinden alınarak main fonksiyonuna bu kod tarafından geçirilmektedir. main fonksiyonu başlangıç kodundan çağırıldığına göre main bittikten sonra akış tekrar başlangıç koduna döner bittikten sonra DOS altında klasik olarak 21h kesmesinin 4Ch fonksiyonu çağırılmaktadır. DOS'ta zaten exit fonksiyonunun yaptığı şey bu kesmenin çağırılmasından ibarettir. DOS'ta tipik bir başlangıç kodu:

```
başlangıç kodu
....
....
....
CALL _main
MOV AH, 4Ch
INT 21h
```

Benzer biçimde main fonksiyonu içerisinde return anahtar sözcüğü kullanıldığında aslında main fonksiyonunun geri dönüş değeri programın exit code'u olmaktadır. Bu durum sembolik makine dilinde şöyle açıklanabilir: main de sıradan bir fonksiyon olduğuna göre geri dönüş değeri AX register'ında bırakılacaktır. Doğrudan AL içerisindeki değer exit code'u olarak 21h kesmesi içinde kullanılabilir.

```
CALL _main
MOV AH, 4Ch
INT 21h
```

Görüldüğü gibi C'nin standardizasyonunda main fonksiyonunun geri dönüş değeri ya void olur ya int olur ama DOS'ta biz int desek bile process'in exit kodu 1 byte'ı geçemeyeceği için yüksek anlamlı byte'ın bir önemi kalmamaktadır. main fonksiyonunda return kullanmakla exit fonksiyonunu çağırmak arasında işlevsel bir farklılık yoktur.

Bir C kodu içerisinde

```
exit(n);
```

biçiminde exit fonksiyonunun çağırılması sırasında derleyici şu kodu üretir:

```
MOV AX, n
PUSH AX
CALL _exit
ADD SP, 2
```

Bu durumda exit fonksiyonu da şöyle yazılmış olmalıdır:

```
_exit proc near
        PUSH BP
        MOV BP, SP
        MOV AL, [BP + 4]
        MOV AH, 4Ch
        INT 21h
        POP BP
        RET
_exit endp
```

Derleyicilerin başlangıç kodları işletim sistemine özgü işler yapmaktadır. Bu yüzden gerektiğinde devre dışı bırakılabilir. Bu yüzden derleyici firmaları başlangıç kodlarını sembolik makine dilinde kaynak kod olarak vermektedir. Geleneksel olarak başlangıç kodlarının isimleri C0.ASM biçimindedir. Bunun derlenmiş hali ise C0.OBJ biçimindedir. DOS'ta başlangıç kodları bellek modeline göre değişiklik göstereceğinden C0.ASM dosyasının çeşitli bellek modelleri için yeniden derlenmesi gerekir. Örneğin C0.ASM'nin small model için derlenmiş halinin ismi C0S.ASM'dir. Large model için C0L.OBJ biçimindedir. Aslında biz C2de tek bir modülle çalışsak bile linker otomatik olarak başlangıç kodunun bulunduğu modülü link işlemine sokmaktadır. Örneğin den.c isimli dosyanın çalıştırılması şöyle gerçekleştirilir:

1. Den.c derlenir, den.obj oluşturulur.
2. C0X.OBJ ile den.obj birlikte link edilerek den.exe yapılır.

Modüllerle çalışma ileride ele alınacaktır.

Bir C programı içerisinde main fonksiyonunu tanımlamadan programı derleyebiliriz. Ancak link işlemini yapamayız. Bu durumda hata link aşamasında linker'ın başlangıç kodundan belirtilmiş olan main isimli fonksiyonu hiçbir modülde bulamaması biçiminde ortaya çıkacaktır.

C'de char Parametrelerin ve Geri Dönüş Değerlerinin Seyrek Kullanılması

C'de char türüne ilişkin parametreler ve geri dönüş değerleri char yerine genellikle int türden verilirler. C'de char parametresine sahip parametre değişkenleri PUSH makine komutuyla stack yoluyla fonksiyona aktarıldığı için aslında derleyici tarafından sanki int türündenmiş gibi işlem görmektedir. C'de int türü tipik olarak çalışılan işlemcide bir register uzunluğu kadar alınmaktadır. Genellikle bir register uzunluğundaki bilgiler stack'e atılır. Özetle parametre char türünden olsa bile aslında pek çok sistemde sanki int türdenmiş gibi aktarılmaktadır. Bu yüzden programcıların bir bölümü char türden parametre kullanmak yerine bu doğallığı vurgulamak amacıyla int türden parametre kullanmaktadır. Aynı durum geri dönüş türü için de kullanılır.

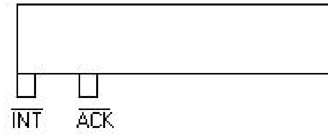
KESMELER(interrupts)

Kesme mikroişlemcinin o anda çalıştırdığı koda ara verip, başka bir kodu çalıştırması işlemidir. Kesme oluştuğunda mikroişlemcinin çalıştırdığı koda kesme kodu(interrupt handler) denilmektedir. Kesmeler çağırılma biçimlerine göre 3'e ayrılır:

1. Yazılım kesmeleri(software interrupts)
2. Donanım kesmeleri(hardware interrupts)
3. İçsel kesmeler(internal interrupts)

Yazılım kesmeleri INT makine komutuyla oluşturulan kesmelerdir. Programcı kesme oluşturmak istediği zaman INT makine komutunu kullanabilir. INT makine komutu tıpkı CALL makine komutu gibi program akışını kesme koduna geçirmektedir. INT makine komutuyla oluşturulan kesmeden IRET makine komutuyla geri dönülür.

Donanım kesmeleri mikroişlemcinin dışındaki bir birim tarafından mikroişlemcinin INT ucunun aktive edilmesiyle oluşturulur. Kesme denilince ilk akla gelen donanım kesmeleridir. Mikroişlemci bir makine komutunu çalıştırdıktan sonra, bir sonraki makine



komutuna geçmeden INT ucunun elektriksel olarak aktive edilip edilmediğine bakar. Intel işlemcileri INT ucu aktive edildiğinde IF bayrağının durumuna göre kesme koduna dallanır ya da kesmeyi görmezlikten gelir. IF bayrağının set ve reset eden STI ve CLI makine komutları vardır. Mikroişlemci IF bayrağı 1 ise kesmeyi kabul eder, 0 ise görmezlikten gelir. Özetle programcı donanım kesmelerini devre dışı bırakmak için CLI makine komutunu, tekrar devreye sokmak için STI komutunu kullanabilir. Özellikle kritik işlemler öncesinde donanım kesmelerinin devre dışı bırakılması işlem bittikten sonra tekrar açılmasına sıklıkla rastlanır. Mikroişlemcinin bir de ACK ucu vardır. kesmeyi kabul ederse işlemci bu ucu kendisi aktive etmektedir.

Yani INT ucu bir giriş ucu, ACK ucu ise çıkış ucudur. Bu durumda tipik bir Intel işlemcisinin donanım kesme protokolü şöyledir:

1. İşlemci her komutun sonunda INT ucuna bakar.
2. INT ucu aktive edilmişse IF bayrağının durumuna bakar. IF 0 ise kesmeyi görmezlikten gelir. IF 1 ise kesmeyi kabul eder ve ACK ucunu aktive eder.

İçsel kesmeler mikroişlemcinin bir problemle karşılaştığında kendisinin çağırdığı kesmelerdir. Örneğin DIV makine komutunda bölen değer 0 ise işlemci bölme işleminin geçersizliğini anlatabilmek için içsel kesme oluşturmaktadır.

INT Makine Komutu ve Kesmelere Dallanılması

Intel işlemcilerinde toplam 256 kesme vardır. Her kesme bir numarayla belirtilir. Genellikle numaralar hex sistemde belirtilir. INT makine komutunun işlem kodu CCh'tır. Bu komuttan sonra kesme numarasını anlatan 1 byte'lık bir sayı gelmektedir. Sembolik makine dilinde komut şöyle yazılır:

INT <kesme numarası>

Intel işlemcileri için RAM'in tepesindeki ilk 1024 byte kesme vektörü olarak kullanılmaktadır. Yani kesme vektörünün yeri 0x00000-0x003FF bölgesidir. Kesme vektörü her kesme numarası için 4 byte'lık kayıtla biçimindedir. bu 4 byte'ın ilk 2 byte'ı kesme oluştuğunda IP register'ının alacağı değeri, diğer 2 byte'ı CS register'ının alacağı değeri belirlemekte kullanılır. Kesme vektörü tipik olarak aşağıdaki gibidir.:

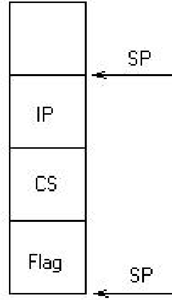
Adres	Kesme vektörü		*Kesme No*
0-3	IP0	CS0	0
4-7	IP1	CS1	1
8-B	IP2	CS2	2
C-F	IP3	CS3	3

...	...	...	...

Bu durumda n numaralı bir kesmeye ulaşmak için işlemci n sayısını 4 ile çarpar. Oradaki ilk iki byte'ı IP register'ına, sonraki 2 byte'ı CS register'ına yerleştirir ve dallanmayı gerçekleştirir. Bu durumda biz n numaralı kesme oluştuğunda istediğimiz bir kodun çalıştırılmasını sağlayabiliriz. Kesme vektörünün yeri 80386 işlemcisiyle birlikte değiştirilebilecek biçime getirilmiştir. Ancak önceki işlemcilerde bu yer sabittir ve belleğin tepesindedir. Ayrıca kesme vektörünün bu yapısı yalnızca mikroişlemcinin gerçek modda ve sanal 86 modda çalışması durumunda geçerlidir. Windows ve UNIX gibi korumalı modda çalışan işletim sistemlerinde kesme vektörü yerine kesme betimleyici tablosu(interrupt descriptor table) denilen bir tablo kullanılmaktadır. Bu tablonun yapısı biraz daha karmaşıktır.

INT makine komutunda sırasıyla şunlar yapılmaktadır:

1. İşlemci geri dönüş için sırasıyla bayrak register'ını, CS register'ını ve IP register'ını stack'e atar(CALL makine komutundan farklı olarak bayrak register'ı da stack'e atılmaktadır).
2. İşlemci IF ve TF bayraklarını sıfırlar.
3. Kesme numarasını 4 ile çarpar, kesme vektöründeki segment ve offset değerlerini CS ve IP register'larına yükler.



Kesmeden geri dönmek için kullanılan IRET makine komutu da sırasıyla IP, CS ve bayrak register'larını stack'ten çeker. Böylece orijinal koda geri döner.

IRET makine komutu

RETF
POPF

komutlarına(mantıksal olarak) eşdeğerdir.

Kesmenin Hook Edilmesi

Bir keme oluştuğunda dallanılacak adres belleğin tepesindeki kesme vektöründe yazmaktadır. Kesme vektöründe n numaralı kesmenin CS ve IP değerleri yerine kendi koduna ilişkin CS ve IP değerlerini yerleştirirsek kesme oluştuğunda bizim belirlediğimiz kod çalıştırılır. Bizim bu kodu IRET makine komutuyla sonlandırmamız gerekir. Eğer biz bu kod içerisinde orijinal kesme kodunu çağıracaksak tıpkı INT makine komutu uygulanmış gibi çağırmalıyız. Bu işlem

```
PUSHF
CLI
CALL old_cs:old_ip
```

komutlarıyla yapılabilir. Burada IF bayrağının reset edilmesi genel olarak önemli değildir. Örneğin vektördeki eski değer cs_old ve ip_old biçiminde olsun. Biz de cs_new, ip_new biçiminde değiştirmiş olalım.

```
....  <------------ cs_ new: ip_new        ....  <------------ cs_old:ip_old
....                                       ....
....                                       ....
....                                       ....
PUSHF                                      IRET
CLI
CALL cs_old:ip_old
....
....
....
IRET
```

Hook etme işlemi bittikten sonra istenirse vektör eski haline getirilebilir. Orijinal kesme kodundan döndükten sonra bir şey yapmayacaksak hook eden kodu bitirebiliriz. Bu durumda kesmenin oluşması nedeniyle gerçekleşen stack durumu orijinal kodda uygulanmış IRET tarafından geri alınabilir:

```
....
....
....
....
PUSH cs_old:ip_old
```

Kesme Kodunun Yazılması

İster yazılım kesmesi için kesme kodu yazacak olalım, isterse donanım kesmesi için kesme kodu yazacak olalım, kesme kodundan IRET komutuyla çıkmadan önce bozduğumuz bütün register'ları eski durumuyla geri yüklememiz gerekir. Yani kesmenin çağırılmasıyla herhangi bir durum değişikliği oluşmamalıdır. Register'ların saklanmasında stack bölgesi kullanılabilir. Bu durumda kesme kodunun başında kullanılacak register'lar stack'e PUSH edilir. Kesme kodunun sonunda bu register'lar POP edilerek IRET uygulanır. Örneğin yazacağımız kesme kodu içerisinde AX, BX, CX, DX register'larını bozacak olalım, kod aşağıdaki gibi organize edilmelidir:

```
PUSH AX
PUSH BX
PUSH CX
PUSH DX
....  ┐
....  │ Kesme
....  │ kodu
....  ┘
POP DX
POP CX
POP BX
POP AX
```

Sembolik Makine Dilinde Dolaylı JMP ve CALL İşlemleri

Sembolik makine dilinde dolaylı JMP ve CALL işlemleri için önce bir data sembolü oluşturulur, sonra JMP ya da CALL komutları *word ptr* ya da *dword ptr* operatörleriyle data sembolü belirtilerek kullanılır. Örneğin:

```
.data
adr1    dw      1234h
adr2    dd      12345678h

.code
JMP     word ptr        adr1
CALL    dword ptr       adr2
```

Burada adr1 ve adr2 sembolleri sırasıyla dallanılacak segment içi ve segment'ler arası adresleri tutmaktadır.

Bellek Erişimlerinde Segment Yükleme Durumları

Normal olarak bellek operandının default bir segment register'ı vardır. Ancak bu default durumu segment yüklemesi ile değiştirebiliriz. Örneğin:

```
MOV AX, [SI]
```

burada bellek operandının default segment register'ı DS'dir. Ancak biz

```
MOV AX, ES:[SI]
```

gibi bir komutla bunu değiştirebiliriz. Buradaki değiştirme işlemi bir komutluk gerçekleşmektedir. Özellikle ES üzerinden yükleme yapılmasına çok rastlanır. Çünkü ES register'ı bu tür amaçlar için düşünülmüştür. Örneğin kesme vektörüne erişecek olsak bir segment register'dan faydalanmamız gerekir. Bu tür durumlarda ilk akla gelecek register ES register'ı olmalıdır.

C için sembolik makine dilinde fonksiyon yazarken AX, BX, CX, DX register'larının yanı sıra ES register'ının da bozulmasında bir sakınca yoktur. Örneğin intrno numaralı kesme için vektörü 1234:5678 biçiminde değiştirmek isteyelim. Bu işlem aşağıdaki gibi yapılabilir:

```
MOV BX, intrno
MOV CL, 2
SHL BX, CL
XOR AX, AX
MOV ES, AX
MOV word ptr ES:[BX], 5678h
MOV word ptr ES:[BX + 2], 1234h
```

C'de uzak göstericileri kullanarak bu işlemi yapmak çok kolaydır:

```
WORD far *pIntr = (WORD far *) (intrno * 4);
*pIntr++ = 0x5678;
*pIntr = 0x1234;
```

Ancak bu işleme de gerek yoktur. Zaten kesme vektörüne değer yerleştiren ve o değeri alan setvect ve getvect isimli fonksiyonlar vardır. Bir kesme kodu C'de *interrupt* anahtar sözcüğünden faydalanılarak da yazılabilir. Bu anahtar sözcük fonksiyonun geri dönüş değeriyle ismi arasına yerleştirilir. Bu tür fonksiyonlara parametre aktarımı amaç bakımından uygun değildir, yani tür fonksiyonlar parametresiz olmalıdır. Bu tür fonksiyonları derlerken derleyici fonksiyon girişinde bütün register'ları stack'e PUSH eder, çıkışta stack'ten POP eder ve fonksiyonu IRET makine komutuyla sonlandırır. Bu tür fonksiyonların adresleri(isimleri) segment ve offset içeren adres belirtir ve adresleri doğrudan kesme vektörüne yerleştirilebilir.

Kesmelerin Fonksiyonları ve Alt Fonksiyonları

Kesmelerin ana girişi 256 tane olmasına rağmen kesme koduna dallanıldığı zaman çeşitli register'ların durumlarına bakılarak içeride switch-case işlemiyle farklı yollara sapılabilir. Kesme kodundan sapılan yollara kesmenin fonksiyonları, kesmenin fonksiyonlarından sapılan yollara kesmenin alt fonksiyonları denir. Genellikle kesme fonksiyonlarına sapmak için Ah, alt fonksiyonlara sapmak içinse AL register'ı kullanılır. bir kesmenin fonksiyon ve alt fonksiyon numaraları kesme çağırılmadan önce AH ve Al register'larına yazılmalıdır.

Kesmenin Parametreleri ve Geri Dönüş Değerleri

Kesme kodları da parametre isteyebilirler. Parametreler kesme çağırılmadan önce belirlenmiş bazı register'lara yerleştirilirler. Kesme kodu çalıştıktan sonra bir değer iletecek olabilir. Kesme kodu IRET ile sonlanmadan önce ileteceği değerleri belirlenmiş bazı register'lara yazar. kesmeyi çağıran kişi o register'lardan geri dönüş değerini alır.

DOS ve BIOS Kesmeleri

Bazı kesmelere ilişkin kodlar EPROM'a yerleştirilmiştir. Makine açılır açılmaz bile kodlar EPROM'da hazır olarak bulunmaktadır. 1 Mb bellek alanının son 64 Kb'ı EPROM bölgesidir. EPROM içerisindeki kesme kodlarının bulunduğu bölüme BIOS(basic input-output system) denir. adres alanının son 64 Kb'lık kısmı(F0000-FFFFF) kısmı ana kart üzerinde normal SIM'lerden farklı bir yerdedir. Aslında SIM içerisinde bu bölgede vardır ama donanımsal olarak buraya yapılan erişimler EPROM'a yönlendirilmektedir. Yani biz son 64 Kb'a erişmek istesek SIM'e değil, EPROM'a erişmiş oluruz. Ancak CMOS seçenekleriyle bilgisayar açıldığında EPROM'daki açılış kodu EPROM içerisindeki BIOS kodlarını SIM'e taşıyabilir. Böylece SIM'e erişim EPROM'a erişimden daha hızlı olduğu için hız kazanılmış olur. BIOS içerisindeki kesme kodları işletim sistemi bile olmadan yapılacak en temel işlemleri gerçekleştirebilecek kodlardır. Önemli BIOS kesmeleri şunlardır:

INT 13h Aşağı seviye disk işlemleri
INT 10h Video işlemleri
INT 16h Klavye işlemleri
INT 14h Seri port işlemleri

Örneğin işletim sisteminin yüklenmesi için 13h kesmesi kullanılmak zorundadır. Ya da bir boot yönetici program klavyeden bilgiyi 16h kesmesiyle almak zorundadır.

DOS kesmeleri DOS işletim sistemi yüklendikten sonra hayat kazanan kesmeleridir. En önemli DOS kesmesi 21h kesmesidir. Denilebilir ki 21h kesmesinin fonksiyonları zaten DOS'un kendisidir. DOS işletim sisteminin bütün sistem fonksiyonları 21h kesme kodu biçiminde organize edilmiştir. Halbuki Windows ve UNIX sistemlerinde durum böyle değildir. Örneğin dosya açma-kapama, dosyadan okuma-yazma gibi işlemlerin hepsi 21h kesmesinin fonksiyonlarıyla yapılır. DOS'un sistem fonksiyonları yani 21h kesmesinin fonksiyonları eninde sonunda BIOS kesmelerini çağırarak aşağı seviye işlemlerini yapar. Örneğin absread fonksiyonu 25h DOS kesmesinden başka bir şey değildir. Bu kesme işlemini 13h'ı çağırarak yapmaktadır. Win32 ve UNIX sistemlerinde aşağı seviyeli işlemler genel olarak BIOS kesmeleriyle yapılmaz. Çünkü BIOS kesmeleri korumalı modda çalışacak biçimde tasarlanmamıştır. Bu işletim sistemleri BIOS kesmeleri ne yapıyorsa onları korumalı modda yapan kodlara sahiptir. Fakat yine de DOS ile ilişkinin tam olarak kesilemediği Win9x ve ME sistemlerinde vwin32.vxd device driver'ından çağırmalar yaparak işlemcinin modunun değiştirilmesi suretiyle BIOS ve DOS kesmeleri kullanılabilmektedir.

10h Video Kesmesi

10h BIOS kesmesi video kartıyla ilgili bütün önemli işlemleri yapmaktadır. Örneğin video modun değiştirilmesi, cursor'ın konumlandırılması, ekrana bütün video modlarına uygun yazıların yazılması vs.

INT 10h, F:0Ah(write char)

Bu fonksiyon tüm video modlarda cursor'ın bulunduğu yere bir karakteri yazdırmak için kullanılmaktadır. Parametreler:

AH	0Ah
AL	Karakter
BH	Sayfa no
BL	Renk bilgisi
CX	Yazılacak karakter sayısı 7 6 5 4 3 2 1 0 B Z Z Z Ş Ş Ş Ş

Geri dönüş değeri yoktur.

Örnek:

```
/*****10hf0ah.asm*****/
   .MODEL SMALL
   .DATA
   mname db 'a'

   .CODE
   main proc near
      mov    ax, @data
      mov    ds, ax

      mov    ah, 0ah
      mov    al, mname
      mov    bl, 0
      mov    cx, 10
      int    10h

      mov    ax, 4c00h
```

```
    int     21h
main endp
end main
.STACK 100h
```

/*****10hf0ah.asm*****/

INT 10h, F:0Eh(write char)

Bu fonksiyon da karakter yazmakta kullanılır. Karakter yazan DOS kesmesi doğrudan bu fonksiyonu çağırmaktadır. Aslında programlama dillerinde ekrana yazı yazan tüm fonksiyonlar sonunda bu fonksiyonu çağırırlar. Parametreler:

AH	0Eh
AL	Karakter
BH	sayfa no
BL	Şekil rengi(grafik modda geçerli)

Geri dönüş değeri yoktur. Bu fonksiyon ekrana karakter yazdırmakta kullanılan en genel fonksiyondur. Ekrana her çeşit yazı yazılmak istendiğinde en aşağı seviyeli fonksiyon olarak tercih edilmelidir. Fonksiyon her zaman renk bilgisini ihmal ederek karakter basar ve cursor'ı ilerletir.

Sınıf çalışması: Data bölümünde xname isimli bir sembol alarak sonu NULL karakterle biten bir string giriniz. Ekrana INT 10h F:0Eh kesmesini kullanarak karakter yazan putchar isimli alt programı yazınız(yazdırılacak karakter stack yöntemiyle fonksiyona aktarılacaktır). Bu alt programı kullanarak xname isimli sembolü NULL karakter görene kadar ekrana yazdırınız.

/*****10hf0eh.asm*****/

```
.model small
.data
xname db "eren, mehmet", 0

.code
putchar proc near
    push    bp
    mov     bp, sp
    mov     al, [bp + 4]
    mov     ah, 0Eh
    mov     bh, 0
    mov     bl, 7
    int     10h
    pop     bp
    ret
putchar endp

main proc near
    mov     ax, @data
    mov     ds, ax
    xor     si, si
    jmp     @2
    @1:
    mov     al,xname + si
    xor     ah, ah
```

```
    push      ax
    call      putchar
    add       sp, 2
    inc       si
    @2:
    cmp       byte ptr xname + si, 0
    jnz       @1
    mov       ax, 4c00h
    int       21h
main endp
end main
.stack
```
/*****10hf0eh.asm*****/

0Eh fonksiyonunun 0Ah fonksiyonundan birkaç farkı vardır. bu farklar 0Eh fonksiyonunun tercih edilmesine sebep olur. Bunlar:

- 0Ah cursor'ı ilerletmez, fakat 0Eh ilerletir.
- 0Eh fonksiyonu satır sonuna gelindiğinde scroll yapabilmektedir.
- Özel kontrol karakterleri için(7, 9, 10, 13 ASCII numaralı karakterler gibi) 0Ah fonksiyonu ASCII tablosunda belirtilen görüntüleri çıkartır. Halbuki 0Eh fonksiyonu bunlar için bilinen işlevleri yerine getirir.

Bir data sembolünü index'lemek için 3 yöntem kullanılabilir. Data sembolünün ismi xname olsun:

1. *MOV SI, offset xname*
   Artık [SI] ifadeleriyle sembol index'lenir.
2. *LEA SI, xname*
   Burada da [SI] ile index'leme yapılabilir.
3. *xname+ SI ya da xname[SI]*
   Burada önce SI 0'lanmalı, daha sonra ifadeyle index'leme yapılmalı.

Sınıf çalışması: xname isimli bir data sembolünü, sonu NULL karakterle biten ilk değer vererek tanımlayınız. Bu semboldeki bütün küçük harfleri büyük harfe dönüştürünüz.

Not: Büyük-küçük harf dönüştürmesini sembolik makine dilinde etkin olarak yapmak için karakterin ASCII koduna 32 eklemek ya da çıkartmak yeterlidir. Bunun için sayının 5 numaralı biti ile işlem yapılabilir.

/*****upper.asm*****/
```
.model small
.data
xname db "eren, mehmet", 0
.code
main proc near
    mov    ax, @data
    mov    ds, ax
    mov    si, offset xname
    jmp    @1
    @2:
    mov    al, [si]
    cmp    al, 97
    jb     @3
    cmp    al, 122
    ja     @3
    and    al, 11011111b
```

```
    mov    [si], al
    @3:
    inc si
    @1:
    cmp    byte ptr [si], 0
    jnz    @2
    mov    ax, 4c00h
    int    21h
  main endp
  .stack 100h
  end main
/*****upper.asm*****/
```

21h Kesmesi

21h kesmesi neredeyse DOS işletim sisteminin kendisidir. DOS'un bütün sistem fonksiyonları bu kesmenin fonksiyonları biçimindedir. 21h kesmesine DOS sürümleriyle birlikte yeni fonksiyonlar eklenmiştir. 21h fonksiyonları eninde sonunda 10h BIOS kesmesini çağırmaktadır. Yani seviye olarak BIOS fonksiyonlarının yukarısındadır.

INT 21h F:01h

Parametreleri:

AH 01

Geri dönüş değeri:

AL Okunan karakter

Bu fonksiyon C'de kullandığımız getche fonksiyonunun ta kendisidir. Klavyeden bir tuşa basılana kadar bekler. Bu fonksiyon kendi içerisinde 16h BIOS kesmesiyle karakteri okur, 10h BIOS kesmesinin 0Eh fonksiyonuyla ekrana yazar.

```
/*****getchee.asm*****/
  .model small
  .code
  main proc near
    mov    ah, 1
    int    21h
    mov    ah, 0Eh
    mov    bh, 0
    int    10h
    mov    ax, 4c00h
    int 21h
  main endp
  .stack 100h
  end main
/*****getchee.asm*****/
```

Bu fonksiyon Ctrl-C işlemi ile 23h DOS kesmesini çağırmaktadır. 23h DOS kesmesine "control break handler" denir. DOS işletim sistemi 21h kesmesi ile 4Ch fonksiyonunu çağırarak programı sonlandırır. 23h kesmesi hook edilerek Ctrl-C işlemine karşı özel şeyler yaptırılabilir. Ayrıca bu fonksiyonun önemli bir özelliği de şudur: eğer özel bir tuşa basılırsa

fonksiyon 0 değeriyle geri döner. Bu durumda aynı fonksiyon bir daha çağırılırsa karakterin extended scan code'u alınır. Bu durumda C'de getch ya da getche fonksiyonları kullanılarak(bu fonksiyonlar zaten bu kesmeyi çağırmaktadır) aşağıdaki gibi scan code elde edilebilir.

```
ch = getch();
if (ch == 0) {
        ch = getch();
        /*özel tuş işlemleri*/
}
else {
        /*normal tuş işlemleri*/
}
```

INT 21h F:2

Bu fonksiyon bir karakteri display etmek amacıyla kullanılır. Parametreler:

AH	2
DL	Görüntülenecek karakter

Bu fonksiyon da doğrudan ekrana karakter yazmakta kullanılan BIOS kesmesini çağırır, ancak Ctrl-C işlemine duyarlıdır. Ctrl-C'ye basılırsa "control break handler" olarak INT 23h kesmesi çağırılır.

INT 21h F:7 ve F:8

7 numaralı fonksiyon C'de kullandığımız getch fonksiyonuna karşılık gelir. Ancak Ctrl-C işlemini dikkate almaz. 8 numaralı fonksiyonun 7 numaralı fonksiyondan farkı Ctrl-C işlemini dikkate almasındadır. Her iki fonksiyonda da kesme sonlandığında okunan karakter AL register'ının içerisine yerleştirilir. Eğer AL 0 ise okunan özel bir tuştur. Bu fonksiyonlardan biri yeniden çağırılırsa extended scan code alınabilir.

INT 21h F:0Ah (buffered keyboard input)

Bu fonksiyon C'de kullandığımız gets fonksiyonuna benzer. DOS kendi prompt'unda bu kesmeyi çağırarak yazı bekler. Benzer biçimde programlama dillerindeki çeşitli fonksiyonlar bu kesmeyi çağırabilmektedir. Parametreler:

AH	0Ah
DS:DX	Buffer

Bu fonksiyon çağırılmadan önce buffer ile tahsis edilen bölgenin ilk byte'ında en fazla ne kadar karakter girileceğine ilişkin bir sayı bulunmalıdır. Giriş enter tuşuna basıldığında sonlandırılır. Kesme girilen karakter sayısını buffer alanının ikinci byte'ına yazar. Girilen karakterler üçüncü byte'tan itibaren yerleştirilmektedir. Girilen yazının sonunda enter karakteri bulunmaktadır. Ancak bu karakter buffer'ın ikinci byte'ına yerleştirilen sayıya dahil değildir(Buffer'ın birinci elemanına yazdığımız sayı enter dahil olmak üzere girişlerin sayısıdır. Yani buradaki sayı 10 ise en fazla 9 karakter girebiliriz).

```
/*****21hfah.asm*****/
.model small
.data
xname   db  10, 11 dup(?)

.code
main proc near
    mov    ax, @data
    mov    ds, ax
    mov    ah, 0ah
    mov    dx, offset xname
    int    21h
    mov    ax, 4c00h
    int    21h
main endp

.stack 100h

end main
/*****21hfah.asm*****/
```

INT 21h F:25h(set interrupt vector)

Bu fonksiyon kesme vektörüne belirli bir kesme için segment:offset adresi yerleştirir. Gerçi bu işlem manuel olarak küçük bir kodla gerçekleştirilebilir. Ancak bu kesme pratik bir çözümdür. Parametreler:

AH	25h
AL	Kesme numarası
DS:DX	Yerleştirilecek vektör adresi

INT 21h F:35h(get interrupt vector)

Bu fonksiyon kesme vektöründen istenen bir kesmenin segment ve offset adreslerinin elde edilmesinde kullanılır. Parametreler:

AH	35h
AL	Kesme numarası

Geri dönüş değeri:

ES	Segment adresi
BX	Offset adresi

INT 21h F:39h(create sub directory)

Bu kesme bir alt dizin yaratmak için kullanılır. Parametreler:

AH	39h
DS:DX	Yaratılacak dizinin isminin bulunduğu yazının başlangıç adresi

Geri dönüş değeri:

Fonksiyonun geri dönüş değeri işlemin başarısını anlatır. CF bayrağına bakılmalıdır. CF bayrağı set edilmiş ise hata vardır, değilse işlem başarılıdır.

INT 21h F:9h

Bu kesme belirtilen bir adresten $ karakteri görülene kadar cursor'ın bulunduğu yerden itibaren yazar. 10h ya da 21h kesmelerinin NULL karakter görene kadar ekrana yazı yazan bir fonksiyonu yoktur. Ekrana bir yazı basmak için sembolik makine dilinde kullanılacak en pratik yöntem bu kesmedir. Parametreler:

AH 9h
DS:DX Yazının adresi

```
/*****21hf39h.asm*****/
.model small

.data
pathname    db  80, 81 dup(?)
text1       db  "Lütfen bir path giriniz:$"
text2       db  "Dizin oluşturulamadı...$"
text3       db  "Dizin oluşturuldu...$"

.code
main proc near
    mov    ax, @data
    mov    ds, ax
    mov    ah, 9
    mov    dx, offset text1
    int    21h
    mov    ah, 0ah
    mov    dx, offset pathname
    int    21h
    xor    bx, bx
    mov    bl, pathname + 1
    mov    byte ptr pathname[bx + 2], 0
    mov    ah, 39h
    mov    dx, offset pathname + 2
    int    21h
    jnc    @1
    mov    ah, 9
    mov    dx, offset text2
    int    21h
    jmp @2
@1:
    mov    ah, 9
    mov    dx, offset text3
    int    21h
@2:
    mov    ax, 4c00h
    int    21h
main endp

.stack 100h
```

```
end main
```

/*****21hf39h.asm*****/

Makro Kullanımı

Makro bir kod yerine genellikle daha geniş olan bir kodun yerleştirilmesi işlemine denir. makro sembolik makine dilinde önemli bir kullanıma sahiptir. Sembolik makine dilinde birkaç komut uzunluğunda olan ve birçok kez tekrarlanan kodlar söz konusudur. Örneğin ne zaman ekrana bir yazı basacak olsak aşağıdaki kalıbı kullanmamız gerekir:

```
MOV AH, 9
MOV DX, OFFSET sembol
INT 21h
```

Bu kadar küçük bir kodun bir proc bildirimiyle yazılması etkin bir yaklaşım değildir. Çünkü zaten bir alt programa parametrelerin aktarılması ve onun çağırılması bile üç komuttan daha uzun sürmektedir. İşte bu durumda bu kod kalıbının defalarca yinelenmesi kaçınılmaz olur. Ancak bu durum kodun görünümünü karmaşık hale getirmektedir. Bu karmaşık görüntü makro çağırmaları biçimine dönüştürülerek algılama kolaylaştırılır. Bir makro çağırıldığında sembolik makine dili derleyicisi CALL makine komutu değil, makroyu oluşturan kodun tamamını koda dahil etmektedir. Makro tanımlamalarının kendisi koda dönüştürülmez. Makro tanımlamaları ya programın başında düzenli olarak yazılır, ya da C'de olduğu gibi ayrı bir dosya biçiminde yazılarak include edilir.

Makro Tanımlamanın Genel Biçimi:

<makro ismi> MACRO [parametreler]

ENDM

Parametreler aralarına boşluk, TAB ya da virgül konularak yazılabilirler. Parametreler tıpkı C'nin makroları gibi makro kodlarında yer değiştirilmektedir. Bir makro şöyle çağırılır:

<makro ismi> [parametreler]

Parametreler arasına virgül, TAB ya da boşluk yerleştirilebilir. Çağırma sırasında makro parametreleri bir sembol, bir register ya da sabit olabilir.

/*****makro00.asm*****/

```
putstr MACRO str
   mov    ah, 9
   mov    dx, offset str
   int    21h
ENDM

.model small
.data
city   db  "Eskişehir$"

.code
main proc near
   mov    ax, @data
   mov    ds, ax
```

```
    putstr  city
    mov     ax, 4c00h
    int     21h
main endp

.stack 100h
end main
```

/*****makro00.asm*****/

Sınıf çalışması: Klavyeden INT 21h F:0Ah ile bir string alan makroyu yazınız. Bu makroyu çağırarak klavyeden string'i alınız. Yazının sonundaki CR(0xD) karakterini $ yaparak putstr makrosuyla ekrana yazdırınız. Yazılacak makronun ismi getstr olup, buffer adresini parametre olarak almalıdır.

/*****makro00.asm*****/

```
;--------------------------------------------
putstr MACRO str
    mov     ah, 9
    mov     dx, offset str
    int     21h
ENDM

getstr MACRO str
    mov     ah, 0ah
    mov     dx, offset str
    int 21h
    xor     bx, bx
    mov     bl, str + 1
    mov     byte ptr str[bx + 2], '$'
ENDM
;--------------------------------------------
.model small
.data
city    db  "Eskişehir$"
strBuf  db  80, 81 dup(?)

.code
main proc near
    mov     ax, @data
    mov     ds, ax
    putstr  city
    getstr  strBuf
    putstr  strBuf[2]
    mov     ax, 4c00h
    int     21h
main endp

.stack 100h
end main
```

/*****makro00.asm*****/

Include İşlemi

Tıpkı C'de olduğu gibi sembolik makine dilinde de bir başka dosya include edilebilir. Sembolik makine dilinin bir ön işlemcisi yoktur. Dosyanın açılması işlemi de derleyicinin kendisi tarafından yapılmaktadır. Dosya include etmenin genel biçimi:

INCLUDE <dosya adı>

Genellikle include dosyalarının içerisine makrolar yazılır. Dosya da asıl kaynak kod içerisinden include edilir.

```
/*****util.inc*****/
;util.inc

getche  macro
    mov     ah, 1
    int     21h
endm

putch   macro   chr
    mov     ah, 2
    mov     dl, chr
    int     21h
endm

getch   macro
    mov     ah, 8
    int     21h
endm

putstr  macro   str
    mov     ah, 9
    mov     dx, offset str
    int     21h
endm

pos     macro   row, col
    mov     ah, 2
    mov     bh, 0
    mov     dh, row
    mov     dl, col
    int     10h
endm

writec  macro   chr
    mov     ah, 0eh
    mov     bh, 0
    mov     al, chr
    int     10h
endm

initprg macro
    mov     ax, @data
    mov     ds, ax
endm

exitprg macro   ecode
    mov     ah, 4ch
    mov     al, ecode
    int     21h
endm
/*****util.inc*****/
```

```
/*****test.asm*****/
  include util.inc

  .model small
  .data

  .code
  main proc near
     initprg

     getche
     putch   al

     exitprg 0
  main endp

  .stack 100h
  end main
/*****test.asm*****/
```

LOCAL Komutu

Sembolik makine dilinde tüm kod ve data sembollerinden program boyunca tek bir tane olmak zorundadır. Bu durum makro yazımlarında probleme yol açabilir. Çünkü içerisinde kod sembolü kullanılan bir makro iki kez çağırıldığında aynı sembol kod içerisinde iki kez görünecektir ve bu da error oluşturacaktır. Örneğin:

```
putstr  MACRO   str
        LEA     BX, str
        JMP     @1
@2:
        MOV     AH, 2
        MOV     DL, [BX]
        INT     21h
        INC     BX
@1:
        CMP     byte ptr [BX] 0
        JNZ     @2
endm
```

makrosu

```
.DATA
xname           db      "ali serçe", 0
yname           db      "necati", 0

.CODE
        ....
        ....
        putstr  xname
        putstr  yname
        ....
```

kodunda iki defa kullanıldığı için derleyici hata verir.

LOCAL komutu kod sembolleri için kullanılır. Kullanım biçimi:

LOCAL sembol1, sembol2, sembol3, ...

Bu komut makro her açıldığında sembolü değişik bir isimle oluşturur. Örneğin Microsoft derleyicileri LOCAL komutu ile belirlenmiş olan sembolleri *??nnnn* biçiminde açmaktadır. Burada nnnn hex sistemde sürekli arttırılan sayıları ifade etmektedir. LOCAL komutu makro içerisinde kullanılmalıdır. Makro kodu;

```
putstr  MACRO str
        LOCAL  @1, @2
        LEA    BX, str
        JMP    @1
@2:
        MOV    AH, 2
        MOV    DL, [BX]
        INT    21h
        INC    BX
@1:
        CMP    byte ptr [BX] 0
        JNZ    @2
endm
```

şeklinde değiştirilirse problem ortadan kalkar.

Matematik İşlemcinin Kullanılması

Matematik İşlemci Nedir?

Mikroişlemciler klasik olarak yalnızca tam sayılı işlemleri yapacak biçimde tasarlanmıştır. Entegre devre teknolojisinin geri olduğu dönemlerde noktalı sayılar üzerinde de işlem yapabilecek büyük bir işlemci tasarlamak mümkün değildi. Buna rağmen noktalı sayı işlemleri yavaş olmasına karşın tam sayılı işlemlerle gerçekleştirilebilmiştir. Intel'in 8086, 8088, 80286, 80386 SX ve DX, 80486 SX modelleri yalnızca tam sayılı işlemler yapabilmektedir. Noktalı sayı işlemlerinin tam sayı işlem yapan makine komutlarıyla gerçekleştirilmesine gerçek sayı emülasyonu denir. Emülasyon yönteminde örneğin iki noktalı sayının çarpımı yüze yakın makine komutu gerektirmektedir. Matematik işlemci gerçek işlemciye bağlanarak çalıştırılan noktalı sayı işlemlerini elektronik devrelerle yapan yardımcı bir işlemcidir. Bir noktalı sayı işlemi emülasyon yöntemi yerine matematik işlemci tarafından çok kısa bir süre içerisinde yapılmaktadır. Örneğin 8086 işlemcisinde emülasyon yoluyla işlemcinin 5 megaherzt hızında olduğu varsayımıyla C'de kullandığımız iki double sayının çarpımı 2100 mikro saniye tutmaktadır. Oysa 8087 matematik işlemcisi aynı işlemi 27 mikro saniyede yapmaktadır. Yani emülasyon yöntemine göre 80 kat daha hızlıdır. 8086, 8088 mikroişlemcileri için üretilen matematik işlemci 8087 işlemcisidir. 80286 işlemcisi için 80287 matematik işlemcisi üretilmiştir. Nihayet 80386, 80486 ve Pentium işlemcileri için halen 80387 matematik işlemcisi kullanılmaktadır. Intel en başta normal işlemciyle matematik işlemciyi elektriksel yolla bağlantı kurularak çalışacak biçimde tasarlamıştır. Geçmişe doğru uyumu korumak zorunluluğu nedeniyle bugünde bu kullanım biçimi devam etmektedir. 80486 DX ve sonraki modellerde matematik işlemci aynı entegre devrenin içersinde ancak ayrık olarak monte edilmiştir. Burada kullanılan matematik işlemci teorik olarak 80387'dir.

Normal İşlemci ile Matematik İşlemcinin Birlikte Çalışması

Bellekten komutlar normal işlemci tarafından alınır. Eğer komut bir matematik işlemci komutu ise normal işlemci tarafından matematik işlemciye verilir ve komut böylece matematik işlemci tarafından işlenir. Bütün matematik işlemci komutları DE ön ekiyle başlar. Normal işlemci komutun DE ön ekiyle başladığını gördüğünde komutu matematik işlemciye verir. Sembolik makine dilinde F ile başlayan komutlar matematik işlemciye ilişkin komutlardır. Örneğin:

FADD
FMUL
FDIV

Bu komutların ilk byte'ı DE ön ekiyle başlamaktadır.

Matematik İşlemcinin Register Yapısı

Matematik işlemcinin 3 önemli register'ı vardır:

1. Data Register'lar:
   Bu register'lar 8 tanedir. Her biri 10 byte'tır ve bir çeşit stack sistemi gibi çalışır.
2. Control Register:
   Bu register matematik işlemcinin durumunu değiştirmek amacıyla kullanılır. 16 bit uzunluğundadır.
3. Status Register:
   Bu register da matematik işlemcinin çalışma durumu hakkında bilgi verir. Yani normal işlemcinin bayrak register'ı gibidir. 16 bit uzunluğundadır.

Noktalı Sayıların Bellekte Tutulma Biçimleri

Noktalı sayıların bellekte tutulma biçimleri konusunda ilk önceleri belli bir standart yoktu. Örneğin DEC firmasının, IBM firmasının tanımladıkları farklı formatlar vardır. Microsoft firması da "Microsoft Binary Format" diye isimlendirilen ayrı bir format tanımlamıştır. Ancak daha sonra kullanmaktan vazgeçmiştir. Daha sonra IEEE noktalı sayıların tutulmasına ilişkin 754 numaralı standardı belirlemiştir. Bu standart donanım firmalarının çoğu tarafından benimsenmiştir. Intel matematik işlemcileri de noktalı sayı formatı olarak bu belirlemeleri kullanmaktadır.

İlk noktalı sayı formatları sabit noktalı formatlardı. Bu formatlarda noktanın belirli bir yerde olduğu varsayılmaktadır. Bu yerin sol tarafına sayının tam kısmı, sağ tarafına ise kesir kısmı yerleştirilmektedir. Ancak daha sonraları kayan noktalı formatlara geçilmiştir. Bu formatlarda sanki nokta yokmuş gibi sayı bir bütün halinde ifade edilir, noktanın yeri sayı içerisine dahil olan ilave bitlerle ifade edilir. IEEE formatı bir kayan nokta formatıdır. IEEE formatında farklı uzunlukta 3 noktalı sayı türü tanımlanmıştır. 4 byte uzunluğundaki formata "short real format"(C'de float), 8 byte uzunluğundaki formata "long real format"(C'de double), 10 byte uzunluğundaki formata "extended real format"(C'de long double) denir.

Noktalı Sayı Formatları

Noktalı sayı formatına dönüşüm şu adımlardan geçilerek yapılabilir:

1. Noktalı sayı ikilik sistemde nokta kullanılarak noktanın sol tarafının ikinin pozitif kuvvetleriyle, sağ tarafı negatif kuvvetleriyle çarpılacak biçimde ifade edilir. Örneğin 12.25 sayısını dönüştürecek olalım. Bu adımda sayıyı şöyle yazarız: 1100.01
2. Format üç kısımdan oluşmaktadır:
   a. İşaret biti
   b. Mantis kısmı
   c. Üstel kısım

   Bu adımda sayının 1.xx durumuna getirilebilmesi için ne kadar sağa ya da sola kaydırılacağı hesaplanır(Bu sayı pozitif ya da negatif olabilir). Sola kaydırmalar pozitif, sağa kaydırmalar negatiftir. Bu sayı BIAS değeri denilen bir değerle toplanır ve formatın üstel kısmına yerleştirilir.
3. Sayıdan nokta atılır. Bir bütün olarak en soldaki 1 çıkartılarak formatın mantis kısmına yerleştirilir. Bu bölümde boş kalan bit'ler 0 ile doldurulur.
4. Sayının işaret biti 0'sa pozitif, 1'se negatif olacak biçimde formatın işaret bit'i kısmına yerleştirilir.

Formattaki bu 3 bölümün kaç bit uzunlukta olduğu ve bu bit'lerin hangi bit'ler olduğu formatta belirlenmiştir.

Yuvarlama Hatası(rounding error)

Kayan noktalı formatlarda noktanın sağ tarafı 2'nin negatif kuvvetleriyle çarpıldığı için onluk sistemde açıkça yazılabilen bazı sayılar bu formatta kesin olarak belirtilemeyebilirler. Örneğin .1 ve .9 ile biten sayıların çoğu bu formatta tam olarak ifade edilememektedirler. Ancak kendisi kesin olarak ifade edilemese de o sayıya yaklaşık bir sayı bu formatta yine de ifade edilebilmektedir. Bir sayının kendisinin tam olarak ifade edilememesinden dolayı onun yerine ona yaklaşık bir sayının ifade edilmesine yuvarlama hatası(rounding error) denir. Formatta sayının mantis kısmı ne kadar büyük tutulursa yuvarlama hatasının olumsuz etkisi o kadar azaltılır. Yuvarlama hatası tam olarak ifade edilen iki sayının işleme sokulup sonuç olarak alınan sayının üzerinde de oluşabilir. Yuvarlama hataları büyük sayılarla çarpma işlemlerinde iyice büyüyebilir. Programlama dillerinin çoğunda iki gerçek sayının eşitliğinin karşılaştırılmasında tam eşit olma durumuna bakılır. Böyle bir karşılaştırma işleminden kaçınmak gerekir. Örneğin aşağıdaki eşitlik matematiksel olarak doğru olduğu halde C'de if deyimi içerisine alındığında doğrulanmayabilir:

1/3+3/4+7/8 == 3/4+7/8+1/3

belki bu iki toplam noktadan sonra 70 basamak aynıdır, fakat tam olarak aynı değildir. Örneğin C'de iki gerçek sayının eşitlik karşılaştırması belli bir duyarlılık karşılığında yapılmalıdır. Bunun için iki yöntem önerilebilir. Birinci yöntem oldukça hızlı çalışır. Sayıların farkının mutlak değeri belli bir sınırdan "küçük mü?" diye bakılır(fabs fonksiyonu). İkinci yöntem popüler olarak pek çok yüksek seviyeli programlama dilinde kullanılmaktadır. Burada sayılar sprintf fonksiyonu ile n basamaklı yazılara dönüştürülür. Yazılar da strcmp fonksiyonu ile blok olarak karşılaştırılır(noktanın ASCII karşılığı sayısal karakterlerden daha küçüktür).

```
int Cmp(double x, double y, int prec)
{
        char s[SIZE], d[SIZE];
        char temp[10];

        sprintf(temp, “%%.%dlf”, prec);
        sprintf(s, temp, x);
        sprintf(d, temp, y);
        return strcpm(s, d);
}
```

Float(short real) Format

31	30 — 23	22 — 0
İşaret	Üstel kısım(8bit)	Kesir kısım(23 bit)

BIAS değeri 127’dir.

Örnek uygulama: 100.25 sayısını float sayı formatına dönüştürünüz.

İkilik düzende	1100100.01
Üstel kısım	6(0110)
Kesir kısmı	1.10010001à 10010001
İşaret biti	0

Kesir kısmı	1001 0001 0000 0000 0000 000
Üstel kısım	6+127=133(10000101)
Sayı	0100 0010 1100 1000 1000 0000 0000 0000(4C88000h)

Kontrol etmek için

```
#include <stdio.h>

void main(void)
{
   float a = 100.25;
   int i;
   unsigned char *p = (unsigned char *)&a;

   for (i = 3; i >= 0; --i)
      printf("%02x ", p[i]);
      putchar('\n');
}
```

C kodu kullanılabilir.

Örnek uygulama: 0.125 sayısını float sayı formatına dönüştürünüz.

İkilik düzende	0.001
Üstel kısım	-3
Kesir kısmı	1.0à 0
İşaret biti	0

Kesir kısmı	0000 0000 0000 0000 0000 000
Üstel kısım	-3+127=124(01111100)
Sayı	0011 1110 0000 0000 0000 0000 0000 0000(3E000000)

Örnek uygulama: 10000000 sayısını ifade edin.

İkilik düzende 100110001001011010000000

İkilik düzende   1111010000100100000

Üstel kısım   19

Kesir kısmı   1.1110100001001000000à1110100001001000000

İşaret biti   0

Kesir kısmı	1110 1000 0100 1000 0000 000
Üstel kısım	19+127=146(10010010)
Sayı	0100 1001 0111 0100 0010 0100 0000 0000 (49742400)

BIAS Değerinin Anlamı

Üstel kısmı BIAS değeriyle topladığımızda iki noktalı sayıyı yüksek anlamlı byte'larından başlayarak byte-byte karşılaştırma olanağını elde ederiz. Örneğin iki pozitif float sayının büyüklük küçüklük ilişkisini araştıracak olalım. Bu iki sayının en yüksek anlamlı byte'larını karşılaştırırız. İşaretsiz olarak hangisi hangisinden büyükse o sayı diğerinden büyüktür.

Float Sınırları İçerisinde En Büyük Pozitif ve En Küçük Pozitif Sayılar Nelerdir?

En büyük sayı: Üstel kısım 1111 1110(254) kesir kısım 0000000000000000000000.
En küçük sayı: Üstel kısım 0000 0001(1) kesir kısmı 0000000000000000000000.

En büyük pozitif sayı	0111 1111 1000 0000 0000 0000 0000 0000(1.701412e+38)
En küçük pozitif sayı	0000 0000 1000 0000 0000 0000 0000 0000(1.175494e-38)

Noktalı Sayı Formatında Özel Sayıları

Noktalı sayı formatı yalnızca belirli sayıları değil, +sonsuz, -sonsuz gibi teorik sayıları da desteklemektedir. Ayrıca bu sayı formatında normal bir sayı olarak yorumlanamayacak kombinasyonlar da söz konusudur. Özel değerler şunlardır:

1. Sıfır sayısı. İki tane sıfır sayısı vardır. Kesir kısmı 0, üstel kısmı 0, işaret biti 1 ya da 0 ise bu sayı sıfır olarak yorumlanır.
2. Sayının işaret biti ne olursa olsun üstel kısım 0, ancak kesir kısmı 0'dan farklıysa bu tür sayılara "denormal" sayılar denir. Denormal sayılar üstel kısmı sayıyı ifade edemeyecek kadar küçük sayılardır ve geçerli sayı olarak ele alınmazlar.
3. Sayının üstel kısmı 0000 0001 ile 1111 1110 arasında ise kesir kısmı ne olursa olsun bu geçerli bir sayıdır. Normal sayılar bu biçimde olmalıdır.
4. Üstel kısmın bütün bitleri 1 ise sayı kesir kısmına bakılarak ya +sonsuz ya -sonsuz ya da tanımsız bir sayı olur. Üstel kısmın hepsi 1, kesir kısmı 0 ve işaret biti 0 ise + sonsuz(01111111100000000000000000000000); üstel kısmın hepsi 1, kesir kısmı 0, işaret biti 1 ise –sonsuzdur(11111111100000000000000000000000). Üstel kısmın hepsi 1, kesir kısmı 0 dışı ise, işaret biti ne olursa olsun sayı tanımsızdır.

Double(long real) Formatı

Sekiz byte'lık noktalı sayı formatıdır. Float formatına göre sayının üstel kısmı ve kesir kısmı büyütülmüştür.

63	62 52	51
İşaret biti	Üstel kısım(11 bit)	Kesir kısmı(52 bit)

BIAS 1023

Long Double(extended real) Formatı

79	78 64	63
İşaret biti	Üstel kısım(15 bit)	Kesir biti(64 bit)

BIAS 16384

Matematik İşlemcide Noktalı Sayılarla İşlemler

Bütün matematik işlemci komutları DE biçiminde olan bir ön ek ile başlar. Normal işlemci bu ön eki gördüğü zaman makine komutunu matematik işlemciye verir. Sembolik makine dilinde noktalı saylar üzerinde işlem yapan komutlar F harfi ile başlar. Matematik işlemci komutları operandsız olabilir, tek operandlı olabilir, ya da çift operanda sahip olabilir. Matematik işlemcinin stack sistemiyle çalışan 8 stack register'ı vardır.

Bu register'ların hepsi 10 byte uzunluğundadır. Bütün noktalı sayılar matematik işlemci içerisinde long double formatına dönüştürülerek bu register'lar içerisinde tutulur. Bu stack sistemine ilişkin bir stack göstericisi(stack pointer) vardır. buradaki stack sistemi döngüseldir. Yani stack göstericisi D0'a geldiğinde bir PUSH işlemi yapılırsa D7 register'ına dönmektedir. Yani stack taşması diye bir durum söz konusu olmaz. Matematik işlemci komutları birkaç grupta incelenebilir:

1. Bellekten matematik işlemcinin stack register'ına PUSH eden komutlar. Bu işlem sırasında stack göstericisi azaltılır ve değer stack göstericisiyle belirtilen register'a yerleştirilir.

2. Bazı komutlar stack göstericisinin gösterdiği bilgiyi almakta kullanılır. Bu komutların bir bölümü bilgiyi stack göstericisinin bulunduğu yerden alır, fakat stack göstericisini arttırmaz. Bir bölümü ise bilgiyi alır ve arttırır.
3. Transfer komutlarının dışında işlem yapan komutlar vardır. Bu komutlar operandsız, tek operandlı ya da çift operandlı olabilir. Operandsız komutlar stack göstericisinin gösterdiği ilk iki register üzerinde işlem yapar. Tek operandlı komutlar stack göstericisinin gösterdiği yerdeki bilgi ile bellek arasında işlem yaparlar. İki operandlı komutlar herhangi iki data register'ı arasında işlem yaparlar.

FLD Komutu

Bu komut PUSH işlemini gerçekleştirir. Yani stack göstericisini 1 azaltır. Bilgiyi stack göstericisinin gösterdiği yere yazar. Başlıca biçimleri şunlardır:

1. *FLD dword ptr mem* Bellekteki float bilgiyi PUSH eder.
2. *FLD qword ptr mem* Bellekteki double bilgiyi PUSH eder.
3. *FLD tbyte ptr mem* Bellekteki long double bilgiyi PUSH eder.
4. *FLD1* Bu komut operandsızdır. 1 sayısını PUSH eder.
5. *FLDZ* Sıfır sayısını PUSH eder.
6. *FLDPI* Pi sayısını PUSH eder.
7. *FLD2E* Logaritma 2 tabanına göre e sayısını($\log_2 e$) PUSH eder.
8. *FLD2T* Logaritma 2 tabanına göre 10 sayısını($\log_2 10$) PUSH eder.
9. *FLDG2* Logaritma 10 tabanına göre 2 sayısını($\log_{10} 2$) PUSH eder.
10. *FLDLN2* Logaritma e tabanına göre 2 sayısını($\log_e 2$) PUSH eder.

FST ve FSTP Komutları

FST stack göstericisinin gösterdiği yerdeki bilgiyi almakta kullanılır. Ancak stack göstericisini arttırmaz. FSTP gerçek bir POP işlemidir. Stack göstericisinin bulunduğu yerden bilgiyi alır ve stack göstericisini arttırır. Şu biçimleri vardır:

1. *FSTP/FSTP dword ptr mem* Register içerisindeki float bilgiyi belleğe almak için kullanılır.
2. *FST/FSTP qword ptr mem*
3. *FST/FSTP tbyte ptr mem*
4. *FST/FSTP ST(n)* ST(0) stack göstericisinin gösterdiği yerdir. ST(1) stack göstericisinin gösterdiği yerden bir ileridir. Bu komut ST (0)'daki bilgiyi ST(n)'e atar.

Tam Sayılara İlişkin PUSH ve POP Komutları

FILD komutu tam sayı formatındaki bir sayıyı matematik işlemcinin stack register'ına PUSH eder. İki temel biçimi vardır:

1. *FILD word ptr mem*
2. *FILD dword ptr mem*

Burada belirtilen sayı her zaman işaretli olarak yorumlanır. Bu durumda C'deki şu dönüştürme

```
double a;
int b;
a = b;
```

için derleyiciler şöyle bir kod üretirler:

FILD word ptr b
FSTP qword ptr a

FIST ve FISTP komutlarıysa bu komutlar matematik işlemcinin stack register'ındaki bilgileri tam sayı formatına çevirerek bellekte istenilen yere yüklerler. Temel olarak iki biçimleri vardır:

1. *FIST/FISTP word ptr mem*
2. *FIST/FISTP dword ptr mem*

Örneğin C'de

long x;
double y;

x = y;

gibi bir işlem için derleyici

FLD qword ptr y
FISTP dword ptr x

gibi bir makine kodu üretir. Görüldüğü gibi C'de tam sayılarla gerçek sayılar arasındaki dönüştürmeler matematik işlemci komutlarıyla yapılmaktadır.

FADD, FADDP Komutları

Bu komutlar iki gerçek sayıyı toplamakta kullanılır. FADD toplama işleminden sonra sonucu stack'e PUSH eder. FADDP toplama işlemi sonrasında POP işlemi yapar.

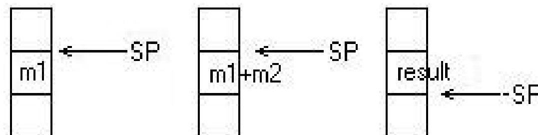
1. *FADD/FADDP dword ptr mem*
2. *FADD/FADDP qword ptr mem*

<u>Toplama örnekleri:</u>

FLD m1
FADD m2
FSTP result

Bu işlem tipik olarak result = m1 + m2 işlemidir. Burada önce m1 stack'e PUSH edilir. Sonra stack göstericisinin gösterdiği yerdeki bilgiyle m2 toplanır ve sonuç POP edilerek result'a alınır.

NOT: Sembolik makine dilinde eğer bellek ya da bellek sabit işlemlerinde bellek olarak bir data sembolü kullanılıyorsa *word ptr*, *dword ptr* gibi belirleme yapmaya gerek yoktur. Eğer operand data sembolü olarak değil de doğrudan [ ] içerisinde register biçimiyle belirtilirse bu belirlemeler yapılmak zorundadır. Yani data sembolleri zaten uzunluk bilgisini içermektedir.



FADD mem komutu stack göstericisinin gösterdiği yerdeki bilgiyle bellekteki bilgiyi toplar, stack göstericisinin gösterdiği yere yazar.

FADD ve FADDP komutlarının operandsız version'ları da vardır. FADD komut ST(0)ßST(0)+ST(1) işlemini yapar.

C'de Gerçek Sayı Türlerine Geri Dönen Fonksiyonları

C'de float, double ve long double türlerinde geri dönüş değerlerine sahip fonksiyonlar geri dönüş değerlerini matematik işlemcinin stack register'ında PUSH edilmiş bir biçimde tutmak zorundadır. Yani çağıran fonksiyon parametreleri stack'e PUSH ettikten sonra fonksiyonu çağırır. Daha sonra FSTP komutuyla geri dönüş değerini matematik işlemcinin stack'inden POP eder. Çağıran fonksiyon matematik işlemcinin stack register'ını dengeleyerek işlemini bitirmektedir. Örneğin:

```
double add(double a, double b);
```

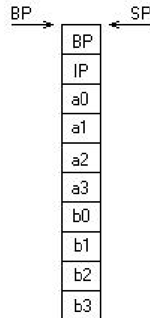
gibi bir fonksiyonun

```
c = add(a, b)
```

şeklinde çağırılması şöyle yapılacaktır:

```
PUSH b3
PUSH b2
PUSH b1
PUSH b0
PUSH a3
PUSH a2
PUSH a1
PUSH a0
CALL _add
FSTP qword ptr c
ADD SP, 16
```

Stack'in görünümü:



Bu fonksiyon şöyle yazılabilir:

```
/*****fadd00.asm*****/
   .model small
   .code
   _add proc near
      PUSH    BP
      MOV     BP, SP
      FLD     QWORD PTR [BP + 4]
      FADD    QWORD PTR [BP + 12]
      POP     BP
      RET
   _add endp

   public _add
   end
/*****fadd00.asm*****/
```

FMUL ve FMULP Komutları

Bu komutlar çarpma işlemi yapar. Üç temel biçimi vardır:

1. *FMUL/FMULP dword ptr mem*
2. *FMUL/FMULP qword ptr mem*
3. *FMUL/FMULP*

FDIV ve FDIVP Komutları

Bu komutlar bölme işlemi yapar. Üç temel biçimi vardır:

1. *FDIV/FDIVP dword ptr mem*
2. *FDIV/FDIVP qword ptr mem*
3. *FDIV/FDIVP*

FSIN, FCOS, FTAN, FSQRT Komutları

Bu komutlar trigonometrik işlemler yapar. Komutların tek biçimi vardır. stack göstericisinin gösterdiği yerdeki bilgiyi işleme sokup yine aynı yere yazarlar.

Sınıf çalışması: Karekök alan mysqrt isimli fonksiyonu matematik işlemci kullanarak yazınız.

```
/*****fsqrt00.asm*****/
   .model small

   .code
   _mysqrt proc near
      PUSH    BP
      MOV     BP, SP
      FLD     QWORD PTR [BP + 4]
      FSQRT
      POP     BP
      RET
   _mysqrt endp

   public _mysqrt
```

end
/*****fsqrt00.asm*****/

BORLAND derleyicilerin Matematik İşlemci Seçenekleri

BORLAND derleyicilerinde Options à Compiler à Advanced Code Generation à Floating Point mönüsünde 4 seçenek vardır:

None	Bu seçenekte kodda matematik işlemci işlemi yapılmayacağı belirtilir. Böylelikle derleyici matematik işlemci için kod oluşturmaz. Eğer noktalı sayı işlemi kullanılırsa error oluşur.
Emulation	Bu seçenek matematik işlemcinin olup olmadığını araştır, varsa onu kullan, yoksa emülasyon kütüphanesiyle fonksiyon çağırarak işlemleri yap anlamına gelir. Bu default olarak installation sırasında belirlenen durumdur.
8087	Bu seçenekle gerçek sayı işlemleri her zaman 8087 matematik işlemci uyumlu komutlar kullanılarak yapılır.
80287/80387	Derleyici her zaman 80287 ya da 80387 uyumlu matematik işlemci uyumlu kod üretir.

Gerçek Sayı Emülasyonu

Emülasyon yöntemi statik ve dinamik biçimlerde kullanılabilir. Statik emülasyonda derleyiciler bir LIB dosyasını link işlemine dahil ederler. Bu LIB dosyası içerisinde noktalı sayı işlemlerini yapan fonksiyonlar bulunur. Derleyiciler gerçek sayı işlemlerini bu kütüphaneden fonksiyon çağırarak yaparlar. Örneğin BORLAND derleyicilerinde EMU.LIB isimli kütüphane bu biçimdeki emülasyon kütüphanesidir. Eğer matematik işlemcimiz yoksa, örneğin iki double sayıyı topluyorsak derleyici bu işlem için FADD makine komutunu kullanmaz, bunun yerine toplama işlemini tam sayılı işlemlerle yapan EMU.LIB içerisindeki bir fonksiyonu çağırır(INT 11h ile matematik işlemci olup olmadığı öğrenilebilir). Dinamik emülasyon yöntemi ilginç bir yöntemdir. Bu yöntemde derleyici normal matematik işlemci kodu üretir. Ancak derleyici bütün DE matematik işlemci ön eki yerine CC komutunu yerleştirir. CC INT 3 isimli bir makine komutudur. INT 3 her zaman 3 numaralı kesmeyi oluşturan tek byte'lık bir makine komutudur. Normal INT makine komutu 2 byte uzunluğundadır. Program çalıştırıldığında bir matematik işlemci komutuna gelindiğinde 3 numaralı kesme çağırılacaktır. Bu kesmenin kodu emülasyon programı tarafından memory resident olarak hook edilmiştir. Hook kodu gerçek koda başvurarak hangi matematik işlemci komutunun kullanıldığını anlar ve uygun fonksiyonu çağırır.

Sembolik makine dilinde gerçek sayıları tutan sembollerin tanımlanması:

Gerçek sayılar "dd", "dq" ya da "dt" semboleriyle belirtilir. Bu belirleyiciler kullnaılarak data sembolleri tanımlandığında sembolik makine dili derleyicileri verilen ilk değerlerde "**.**" olup olmadığına bakarlar. Eğer "." varsa gerçek sayı formatına uygun olarak sayıyı yerleştirirler. "." yoksa ikiye tümleyen aritmetiğine göre tam sayı formatında yerleştirilirler.

Örneğin:

```
x       dd      100
y       dd      100.2
```

İlk değer olarak verilen sayının sonuna "r" getirilebilir. "r" getirme işlemi sayı "**.**" içermediği zaman daha anlamlıdır. Örneğin iki gerçek sayının toplanması şöyle yapılır:

```
.DATA

x       dd      100.2
y       dd      500.3
z       dd      ?
.CODE

        fld     x
        fadd    y
        fstp    z
```

Not: TD programında matematik işlemcimnin register'ları **"View-->Numeric Processor"** seçeneğiyle görüntülenir. Bu görüntüde ST(0), ST(1), ST(2), ... o anda stack gösatericisinin gösterdiği data register 'dan başlayarak register belirtir. ST(0), matemetik işlemcisinin stack göstericisinin gösterdiği register'dır. Bunun gerçekte kaç numaralı data register olduğu debugger'de ST alanında yazmaktadır.

PipeLine İşlemi

PipeLine , mikroişlemci bir makine komutu üzerinde çalışırken saonraki makine komutlarının çeşitlş işlemlerini bu arada yapması anlamına gelir. Böylece çalışma sırası sonraki makine komutuna gelindiğinde o komutun zaten belirli bir parçası yapılmış olur. RISC işlemcilerinde bu mekanizma ideal bir şekilde yapılmaktadır. çünkü RISC işlemcilerinde bütün komutların byte uzunluğu eşittir ve bütün komutlar eşit zamanda çalıştırılırlar. Bu özellikler PipeLine işlemini tasarım bakımından çok kolaylaştırmaktadır. Ancak yine de bir CISC grubu işlemci olan 80x86 ailesinde de ideal olmasa da bir pipe line işlemi yapılmaktadır.

Normal İşlemciyle Matematik İşlemcinin Senkronizasyonu

Normal işlemciyle matematik işlemci arasında elektriksel bir bağlantı söz konusudur. Normal olarak işlemci komutun matematik işlemciyi ilgilendirdiğini anlar ve komutu matematik işlemciye verir. Bundan sonra komutlar artık paralel bir biçimde asenkron olarak işletilmektedir. Bu asenkron çalışmanın iki problemi vardır:

1. Bir matematik işlemci komutundan sonra bir normal işlemci komutu varsa ve bu normal işlemci komutu matematik işlemci komutunun sonucuyla ilgiliyse normal işlemcinin matematik işlemcinin komutu bitirmesine kadar bekletilmesi gerekir. Örneğin,

   ```
   inc  bx
   fstp x
   mov  ax,word ptr x
   ```

   Normal işlemci, matematik işlemci komutunu matematik işlemciye verdikten sonra sonraki komutla çalışmasına devam eder. Oysa sonraki komut matematik işlemci komutndan sonra çalıştırlmak zorundadır. Yukarıdaki örnekte normal işlemcinin matematik işlemci komutundan sonra matematik işlemcinin komutu tamamlanarak bekletilmesi gerekir. Bu

bekletme işlemi "wait / fwait" komutuyla yapılır (wait ve fwait komutları aynı komutlardır.). Bu komut, normal işlemci tarafından işlenen bir komuttur. Bu komutu normal işlemci aldığında matematik işlemci işlemini tamamlayana kadar bekler. Aslında wait komutu her matematik işlemci komutundan sonra yerleştirilmek zorundadır. Wait komutunun gerekip gerekmediği sonraki komuta bağlıdır. Ancak sembolik makine dilinde ne olursa olun matematik işlemci komutlarında sonra "wait"komutunu yerleştirmek pratik bir çözümdür. Peşpeşe matematik işlemci komutları geliyorsa tüm komutlardan sonra bir kez wait yerleştirmek yeterlidir.

2. Diğer bir senkronizasyon problemi yalnızca 80x86 işlemcisinde söz konusu olmaktadır. Çünkü bu senkronizasyon problemi 80x88, 80286, 80386 ve Pentium modellerinde işlemci tarafından otomatik bir biçimde çözülmüştür. Yani bu senkronizasyon problemi bugün tamamen problem olmaktan çıkarılmıştır. Bu durumda bir normal işlemci komutundan sonra bir matematik işlemci komutu geliyorsa 80x86 işlemcisinin pipeline mekanizmasının tasarımından dolayı problemli bir durumm ortaya çıkabilmektedir. Normal işlemci komutu işlerken sonraki komutun matematik işlemci komutu olduğunu gördüğünde daha o komutu bitirmeden sonraki komutu matematik işlemciye iletir. Bu durumda aynı senkronizasyon problemi ortaya çıkmaktadır. Çözüm normal işlemci komutundan sonra matematik işlemci komutundan önce wait komutunu yerleştirmekle çözülebilir.Örnek:

```
     mov     [SI],ax
wait--->
     fld     dword ptr [SI]
```

Bu problem yalnızca 80x86 için söz konusudur. Diğer işlemciler zaten otomatik olarak eğer normal işlemci komutundan sonra matematiki işlemci komutu geliyorsa zaten pipeline işlemini yapmayıp beklemektedir. MASM ve TASM derleyicilerinde default durum 80x86 ve 80x87 işlemcilerine göre kod üretimidir (Bu durum değiştirilebilir, ileride ele alınacaktır). Bu durumda MASM ve TASM default olarak normal işlemci komutlarından sonra matematik işlemci komutu geliyorsa kendisi wait komutunu yerleştirir. Wait komutları genel olarak performans üzerinde ciddi bir etkiye yol açmaz. Sembbolik makine dili derleyicisinin bu durmda wait koumtunu otomatik eklemesinin nedeni bu komutun diğer işlemcilerde gerekmeyebileceği içindir.

Konunun Özeti: Biz matematik işelemci komutlarında sonra normal işlemci komutu geliyorsa wait komutunu yerleştirmeliyiz. Ancak normal işlemci komutunda sonra matematik işlemci komutu geliyorsa wait komutunu yerleştirmemeliyiz.

Wait komutunun çalışma biçimi:

Normal işlemcinin "test", matematik işlemcinin "busy" uçları vardır.

TEST .<--- .BUSY

CPU MATH Proc

Wait komutu aslında işlemcinin "test" ucunu örnekler. Test ucu logic 0'da olduğu sürece işlemciyi durdurur. Matematik işlemci her komutun çalışmasını bitirdiğinde busy ucunu kısa

bir süre logic 1 durumuna çeker. Sonra yeniden bu ucu logic0 durumuna düşürür. Böylece normal işlemciyi wait komutundan aslında matematik işlemci sayesinde çıkmaktadır.

Gerçek Sayıların Karşılaştırılması :

Normal olarak iki noktalı sayının tam eşitliğinin karşılaştırılması anlamlı bir durum değildir. Bemzer biçimde büyüklük-küçüklük karşılaştırılması da anlamlı olmayabilir. Çünkü karşılaştırma işlemleri yuvarlama hatalarını dikkate almadan kesin değerler üzerinde yapılmaktadır. Örneğin ;
2/3 + 4/7 = 4/7 + 2/3 sayısına eşit olmayabilir. Yani iki sayı noktadan sonra aynı olabilir. Ancak matematik işlemci komutuyla karşılaştırırsak mantıksal anlamda çok küçük bir sapma yüzünden aynı çıkmayacaktır.

Status Register(status word)

Bu register, tıpkı normal işlemcinin bayrak register'ı gibi görev yapar. Çeşitli bit'lere sahiptir. 16 bit uzunluğundadır. Şu durumlara yanıt veren bayrakları vardır:

- İşlemde taşma olmuş mudur?
- Sıfıra bölme olmuş mudur?
- Stack göstericisi kaç numaralı register'ı göstermektedir?
- Diğerleri

Noktalı sayıların karşılaştırılmasında da bu register'ın bazı bayrakları kullanılır. Status register bilgisi elde edilebilir. Bu işlem *FSTSW* komutuyla yapılır. Komutun kullanımı tüm matematik işlemcilerde bellek operandı verilerek yapılabilir, ancak 80287 ve sonrasında komut register üzerinden de çalışabilmektedir. Register normal işlemcinin 16 bit register'larından biri olmalıdır. Komutun kullanım biçimi şöyledir:

FSTSW word ptr mem
FSTSW reg (80287 ve sonrası)

Status register içerisindeki bit'ler şöyledir:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	SP			C2	C1	C0	IR	?	PE	UE	OE	ZE	DE	IE

IE(invalid operation)
DE(denormalized operand)
ZE(zero divide)
OE(overflow)
UE(underflow)
PE(precision)
IR(interrupt request)
SP(stack pointer)
C0, C1, C2, C3(condition code)
B(busy)

İki noktalı sayıyı karşılaştırmak için *FCOM* komutu kullanılır. Bu komut status register'ın C0, C2, C3 bayraklarını etkiler. Sonuç bu bayraklara bakılarak belirlenir. *FCOM* komutunun biçimleri şunlardır:

1. *FCOM* (ST ile ST(1) sayılarını karşılaştırır)
2. *FCOM ST(n)* (ST ile ST(n) sayılarını karşılaştırır)
3. *FCOM mem* (ST ile mem sayılarını karşılaştırır)

Bu üç komutun FCOMP isimli POP'lu versiyonları da vardır. Tipik bir karşılaştırma işlemi şöyle yapılır:

Örneğin a ile b karşılaştırılacak olsun

```
FLD a
FCOMP b
```

Bu karşılaştırma işleminden sonra karşılaştırmanın sonucu status register'ın C0, C2, C3 bit'lerine bakılarak tespit edilir. Tabii bu bit'lere bakabilmek için önce status register'ı *FSTSW* komutu ile elde etmek gerekir. Bu elde etme işlemi 80287 ve sonrası için

FSTSW AX

ile, 8087 için iki aşamada

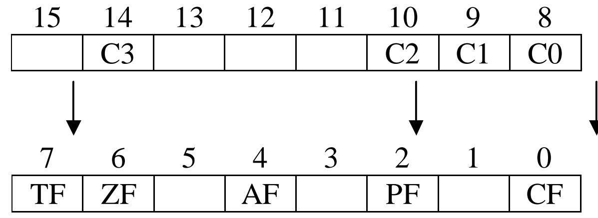
```
FSTSW word ptr mem16
MOV AX, mem16
```

ile yapılabilir.

Normal İşlemcilerde Bayrak Register'ı Üzerinde İşlemler

Bayrak register'ındaki bayraklar üzerinde işlemler yapmanın 3 yolu vardır:

1. CF, DF ve IF bayraklarını set ve reset eden özel komutları kullanmak. Bu komutlar: *CLC*, *STC*, *CMC*, *CLI*, *STI*, *CLD*, *STD*.
2. *PUSHF* komutuyla bayrak register'ını stack'e atıp burada düzeltme yaptıktan sonra *POPF* ile geri çekmek.
   ```
   PUSHF
   MOV BP, SP
   MOV AX, [BP]
   ....                    (flag register'ları üzerinde işlemler).
   MOV [BP], AX
   POPF
   ```
3. *LAHF* ve *SAHF* komutlarını kullanmak.
   Bu komutlar operandsızdır. *LAHF* komutu bayrak register'ının düşük anlamlı byte'ını AH register'ına yükler. *SAHF* AH register'ındaki bilgiyi bayrak register'ının düşük anlamlı byte'ına yükler. Bayrak register'ının düşük anlamlı byte'ında CF, PF, AF, ZF, SF ve TF bayrakları bulunmaktadır. Status register'ın yüksek anlamlı byte'ı ile bayrak register'ının düşük anlamlı byte'ı aşağıdaki gibi birebir eşleştirilebilir:



Şimdi a ile b sayısının karşılaştırılmasının genel kalıbı yazılabilir:

FLD a
FCOMP b
FSTSW AX
WAIT
SAHF

Artık CF = C0, PF = C2 ve ZF = C3 olmuştur.

C0, C2 ve C3 Status Register Bit'lerinin Karşılaştırmadaki Anlamları

Eğer C2 set edilmişse problemli bir durum vardır. Ya operandlardan bir tanesi geçerli bir sayı değildir, ya da +sonsuz, -sonsuz gibi bir değerdedir. Böyle problemli bir durumun araştırılması çoğu kez gerekmez, görmezden gelinebilir. Bu hatanın tespit edilmesi *JP* komutuyla yapılabilir. Eğer C3 set edilmişse iki operand birbirine eşittir. Bu durum *JZ* komutuyla tespit edilebilir. Eğer birinci operand yani örneğimizdeki a(yani ST'de bulunan) ikinci operanddan büyükse C0 reset edilir, küçükse set edilir. Bu durum *JC*, *JB*, *JNC*, *JNB* komutlarıyla tespit edilebilir. Ayrıca küçük eşit ve büyük eşit karşılaştırmaları için *JBE*, *JNE*, *JA*, *JAE* komutları kullanılabilir.

Karşılaştırma örnekleri:

C kodu	Assembler karşılığı

double a, b;
 	
if (a == b)
    ifade1;
else
    ifade2
	FLD a
	FCOMP b
	FSTSW AX
	WAIT
	SAHF
	JZ @1
	İfade2
	JMP @2
	@1:
	ifade1
	@2:

C kodu	Assembler karşılığı

double a, b;
if (a <= b)
    ifade1;
else
    ifade2;
	FLD a
	FCOMP b
	FSTSW AX
	WAIT
	SAHF
	JBE @1
	İfade2
	JMP @2
	@1:
	ifade1
	@2:

Programlama Dillerindeki Yerel Değişkenlerin Kullanılması

Programlama dillerinde yerel değişkenler tıpkı parametre değişkenleri gibi stack bölgesini kullanmaktadır. Intel mimarisinde C derleyicileri fonksiyonun hangi bloğunda olursa olsun tüm yerel değişkenleri fonksiyonun başında stack üzerinde oluşturur. Bunun için

```
PUSH BP
MOV BP, SP
```

komutlarından sonra toplam yerel değişkenlerin miktarı kadar SP register'ı geriye çekilir. n yerel değişkenlerin toplam byte sayısı olmak üzere, bu işlem

```
SUB SP, n
```

ile gerçekleştirilebilir. Yerel değişken kullanan tipik bir fonksiyonun giriş kodu şöyledir:

```
PUSH BP
MOV BP, SP
SUB SP, n
```

Bu durumda yakın modellerde yerel değişkenler için aşağıdaki gibi taralı bir bölge oluşturulmuş olur.

Burada artık SP PUSH işlemlerinde normal olarak kullanılabilir. Yani PUSH ve POP işlemleri taralı bölgeyi bozmaz. Taralı bölgeye BP – k gibi bir ifadeyle erişilebilir. Derleyiciler yerel değişkenleri sırasıyla taralı bölgede oluştururlar. Örneğin func fonksiyonu aşağıdaki gibi olsun:

```
void func(int a, int b)
{
        int x, y, z,m ;

        .....
}
```

Yerel değişkenlerin taralı bölgeye nasıl yerleştirileceği derleyiciden derleyiciye değişiklik gösterebilmektedir. Örneğin Turbo C 2.0 derleyicisinde yerleşim ilk tanımlanan değişken stack bölgesinin düşük anlamlı bölgesinde olacak biçimde yerleştirilmektedir. Borland C ve Microsoft derleyicilerinde ilk tanımlanan değişken yüksek anlamlı byte'ta olacak biçimde

yerleşim yapılmaktadır. Yukarıda belirtilen func fonksiyonu için giriş kodu aşağıdaki gibi olacaktır:

PUSH BP
MOV BP, SP
SUB SP, 8

İçsel organizasyon biçiminin değişmesinin hiçbir taşınabilirlik problemi yoktur. Çünkü yerel değişkenlere dışarıdan zaten erişilemez. Bu durumda fonksiyonu programcı yazmışsa başka fonksiyonlar zaten bu değişkenlere erişmeyecektir. Derleyici yazmışsa sembolik makine dili programcısı erişmeyecektir. Yani derleyicinin uyguladığı yöntem ile sembolik makine dili programcısının uyguladığı yöntem farklı olsa bile herhangi bir problem oluşmaz. Yukarıdaki örnekte TC 2.0 derleyicisinde yerel değişkenlerin yerleri şöyle olacaktır:

xà [BP - 8]
yà [BP - 6]
zà [BP - 4]
mà [BP - 2]

BC 3.0 ve Microsoft derleyicilerinde erişim şöyledir:

mà [BP - 8]
zà [BP - 6]
yà [BP - 4]
xà [BP - 2]

Yerel değişken kullanan fonksiyonları çıkış kodu aşağıdaki gibi olmalıdır:

MOV SP, BP
PUSH BP
RET

MOV SP, BP yerel değişkenler için ayrılan alanın boşaltılması amacıyla kullanılmaktadır.

Çeşitli Örnekler

Aşağıda yerel değişken kullanan fonksiyonların sembolik makine dili karşılıkları verilmiştir. Yerel değişkenlerin yerleşim biçimi olarak Borland C 3.1 + Microsoft C sistemi kullanılmıştır.

C kodu	Assembler karşılığı

int add(int a, int b)
{
    int result;
 	
    result = a + b;
    return result;
}
	_add proc near
	    PUSH BP
	    MOV BP, SP
	    SUB SP, 2
	    MOV AX, [BP + 4]
	    ADD AX, [BP + 6]
	    MOV [BP - 2], AX
	    MOV AX, [BP - 2]
	    MOV SP, BP
	    POP BP
	    RET
	_add endp

/*****yerel001.c*****/	/*****yerel001.asm*****/

C Kodu	Assembler karşılığı

void swap(int *p1, int *p2)
{
    int temp;
 	
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
 	
void main(void)
{
    int x = 10, y = 20;
    swap(&x, &y);
}
	_swap proc near
	    push bp
	    mov bp, sp
	    sub sp, 2
	    push si
	    mov bx, [bp + 4]
	    mov ax, [bx]
	    mov [bp - 2], ax
	    mov si, [bp + 6]
	    mov ax, [si]
	    mov [bx], ax
	    mov ax, [bp – 2]
	    mov [si], ax
	    pop si
	    mov sp, bp
	    pop bp
	    ret
	_swap endp
	
	_main proc near
	    push bp
	    mov bp, sp
	    sub sp, 4
	    mov word ptr [bp – 2], 10
	    mov word ptr [bp – 4], 20
	    lea ax, [bp – 4]
	    push ax
	    lea ax, [bp – 2]
	    push ax
	    call _swap
	    add sp, 4
	    mov sp, bp
	    pop bp
	    ret
	_main endp

/*****yerel002.c*****/	/*****yerel002.asm*****/

Borland derleyicisinin ürettiği asm dosyasından ? ile başlayan debug satırlarını temizleyen program:

```
/*****trimasm.c*****/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define LINELEN         1024
#define MAXPATH                 80

int is_debug_line(const char *buf);

void main(int argc, char *argv[])
{
        char Path[MAXPATH], buf[LINELEN];
        FILE *fs, *fd;

        if (argc != 2) {
                fprintf(stderr, "usage: %s <filename>\n", argv[0]);
                exit(1);
        }

        if (argc > 2) {
                fprintf(stderr, "Too many parameters\n");
                exit(2);
        }
        if ((tmpnam(Path)) == NULL) {
                fprintf(stderr, "Cannot get temp file\n");
                exit(3);
        }
        if ((fs = fopen(argv[1], "r")) == NULL) {
                fprintf(stderr, "cannot open file: %s\n", argv[1]);
                exit(4);
        }
        if ((fd = fopen(Path, "w")) == NULL) {
                fprintf(stderr, "cannot open temp file: %s\n", Path);
                exit(4);
        }
        while (fgets(buf, LINELEN, fs) != NULL) {
                if (!is_debug_line(buf))
                        fprintf(fd, "%s", buf);
        }
        fclose(fs);
        fclose(fd);
        if (remove(argv[1]) == -1) {
                fprintf(stderr, "cannot delete file: %s\n", argv[1]);
                exit(5);
        }
        if (rename(Path, argv[1]) == -1) {
                fprintf(stderr, "cannot rename file %s to file %s\n", Path, argv[1]);
                exit(6);
        }
}
```

int is_debug_line(const char *Buf)
{
while(isspace(*Buf))
++Buf;
return *Buf == '?';
}

/*****trimasm.c*****/

Bir dizi söz konusuysa, C’nin kuralları gereği, hangi derleyici söz konusu olursa olsun, dizinin ilk elemanı düşük adreste olmak zorundadır.

C kodu	Assembler karşılığı
void main(void) *{* *char s[4];* *s[0] = ‘a’;* *s[1] = ‘l’;* *s[2] = ‘i’;* *s[3] = ‘\0’;* *puts(s);* *}*	*_main proc near* *push bp* *mov bp, sp* *sub sp, 4* *mov byte ptr [bp - 4], ‘a’* *mov byte ptr [bp - 3], ‘l’* *mov byte ptr [bp - 2], ‘i’* *mov byte ptr [bp - 1], 0* *lea ax, [bp –4]* *push ax* *call _puts* *pop ax* *mov sp, bp* *pop bp* *ret* *_main endp*
/*****yerel003.c*****/	/*****yerel003.asm*****/

C kodu	Assembler karşılığı
void main(void) *{* *int a = 10;* *int *p;* *p = &a;* **p = a;* *}*	*_main proc near* *push bp* *mov bp, sp* *sub sp, 4* *push si* *mov word ptr [bp - 2], 10* *lea ax, [bp - 2]* *mov [bp - 4], ax* *mov si, [bp - 4]* *mov word ptr [si], 20* *pop si* *mov sp, bp* *pop bp* *ret* *_main endp*
/*****yerel004.c*****/	/*****yerel004.asm*****/

String’ler derleyici tarafından .data alanına yerleştirilir. Sonra başlangıç adresiyle kullanılırlar.

C kodu	Assembler karşılığı
void main(void) *{* *char *p;* *p = “Ankara”;* *puts(p);* *}*	*.data* *s@ db “Ankara”, 0* *_main proc near* *push bp* *mov bp, sp* *sub sp, 2* *mov ax, offset s@* *mov [bp – 2], ax* *push ax* *call _puts* *pop cx* *mov sp, bp* *pop bp* *ret* *_main endp*
/*****yerel005.c*****/	/*****yerel005.asm*****/

Derleyiciler genellikle bir diziye küme parantezleriyle ilk değer verildiğinde verilen ilk değerleri .data alanında statik olarak saklarlar. Daha sonra memcpy gibi bir kopyalama fonksiyonu kullanarak diziye kopyalarlar. Böyle bir işlem derleyici bakımından daha pratiktir. C’de küme parantezleri içerisinde verilen ilk değerlerin sabit ifadesi olma zorunluluğu da buradan gelmektedir. Eğer derleyici ilk değerleri .data bölgesine yazıp kopyalama yapmasaydı dizi çok uzun olduğunda elemanlara tek tek atama yapmak için çok fazla mov komutu gerekirdi. Burada zaman değil, bir kod optimizasyonu düşünülmektedir.

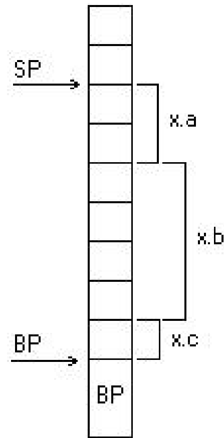
C kodu	Assembler karşılığı
void func(void) *{* *int a[] = {1, 2, 3, 4, 5};* *}*	*.data* *a@ dw 1, 2, 3 ,4, 5* *_func proc near* *push bp* *mov bp, sp* *sub sp, 10* *mov ax, 10* *push ax* *mov ax, offset a@* *push ax* *lea ax, [bp – 10]* *push ax* *call _memcpy* *add sp, 6* *mov sp, bp* *pop bp* *ret* *_func endp*
/*****yerel006.c*****/	/*****yerel006.asm*****/

Nokta(.) operatörüyle bir yapı elemanına erişilmek istendiğinde derleyici doğrudan erişilecek elemanın offset’ini bularak erişimi gerçekleştirebilir. Örneğin:

C Kodu	Assembler karşılığı

struct SAMPLE {
    int a;
    long b;
    char c;
};
 	
void main(void)
{
    struct SAMPLE x;
 	
    x.a = 10;
    x.b = 20;
    x.c = 'a';
    printf("%d %lf %c\n", x.a, x.b, x.c);
}
	.data
	s@ db "%d %lf %c", 10, 0
	.code
	_main proc near
	    push bp
	    mov bp, sp
	    sub sp, 7
	    mov word ptr [bp - 7], 10
	    mov word ptr [bp - 5], 20
	    mov word ptr [bp - 3], 0
	    mov word ptr [bp – 1], 97
	    mov al, [bp – 1]
	    cbw
	    push ax
	    push word ptr [bp – 3]
	    push word ptr [bp – 5]
	    push word ptr [bp – 7]
	    mov ax, offset s@
	    push ax
	    call _printf
	    add sp, 10
	    mov sp, bp
	    pop bp
	    ret
	_main endp

/*****yerel007.c*****/	/*****yerel007.asm*****/



İşaretli tamsayı türünü daha uzun işaretli tamsayı türüne dönüştüren makine komutları:

1. CBW   AL→AX
2. CWD   AX→DX:AX
3. CWDE   AX→EAX
4. CDA   EAX→EDX:EAX

Hizalama(Alignment)

16 bit 8086 ve 8088 işlemcileri 2 byte üzerinde işlem yaparken çift bellek adreslerine tek bellek adreslerinden daha hızlı erişim yapar. Benzer biçimde 80386 ve sonrası işlemciler 4 byte bellek operandı üzerinde çalışırlarken adres 4'ün katlarında ise daha hızlı erişim gerçekleştirirler. Bunun nedeni CPU ile bellek arasındaki bellek biçimindendir.

Tipik bir 32 bit Intel işlemcisi bellek ile 30 adres ucu, 32 veri ucu kullanılarak bağlanmıştır. İşlemci bellekten ancak 4'ün katlarından 4 byte çekebilmektedir. Örneğin işlemci bir hamlede 28'inci byte'tan başlayarak 4 byte çekebilir. Peki 30'uncu byte'tan başlanarak 4 byte çekilmek istense ne yapacaktır? İşlemci her adrese ilişkin komutu kabul eder. Ancak bunu tek hamlede değil, kendi içerisinde iki hamlede gerçekleştirir. İşlemci önce 28'inci byte'tan 4 byte çeker, bunun yüksek anlamlı iki byte'ını saklar. Sonra 32'inci byte'tan 4 byte çeker ve bunun da düşük anlamlı iki byte'ını saklayarak, ancak iki seferde belleğe erişerek işlemini tamamlar. Yani komut çalıştırılmaktadır ama iki aşamada işlem yapıldığı için daha yavaş çalışmaktadır. Eğer bütün nesneler 4'ün katları olsaydı, işlemci bunu kendi içerisinde tek hamlede yapacak ve daha hızlı çalışacaktır. Hizalama derleyicinin 16 bit işlemcilerde nesnelerin ikinin katları olan adreslerde, 32 bit işlemcilerde 4'ün katı olan adreslerde tutmaya gayret etmesidir. Bir byte'lık bir nesnenin herhangi bir adreste tutulmasının bir zararı yoktur. Hizalama bir byte'tan uzun olan nesneler için söz konusudur. Örneğin 16 bit işlemcide hizalama uygulanmak istensin, aşağıdaki fonksiyondaki yerel değişkenlerin tahsisatı nasıl olacaktır?

```
void func(void)
{
        char c;
        int a;
}
```

Derleyici a değişkeni çift adrese gelecek biçimde c ile a arasında boşluk bırakarak yerleşim uygular.

Word Hizalaması(word alignment)

16 bit'lik derleyiciler word hizalaması seçeneğine getirildiklerinde aşağıdaki gibi davranırlar:

1. Fonksiyon blok girişlerinde SP her zaman çift adreste olur.
2. char türü dışındaki tüm türler çift adresten başlayarak yerleştirilir.
3. Yapı değişkenlerinin kendisi çift adresten başlar. Yapının char dışı elemanlarının hepsi çift adreste bulunmak zorundadır. Örneğin:

   ```
   struct SAMPLE {
         char a;
         int b;
   };
   ```

   sizeof(struct SAMPLE) 4'e eşittir. Bu yüzden dinamik tahsisatlarda kesinlikle sizeof operatörü kullanılmalıdır. Byte hizalaması demek derleyicinin böyle bir çabaya girmemesi demektir. Byte hizalaması yapıldığında *sizeof(struct SAMPLE)* 3'e eşit olacaktır. Dinamik bellek fonksiyonları hizalama problemi yüzünden genellikle çift adresten başlayacak biçimde tahsisat yapar. Böylece malloc ile tahsis edilen bir yapı alanına erişimde word hizalaması kullanılmış olsa bile hız kaybı olmayacaktır.

4. Yapının toplam uzunluğu tek ise çifte tamamlanmaktadır. Yani:

```
struct SAMPLE {
    int b;
    char a;
};
```

yapısının da sizeof'u 4'tür.

Borland derleyicilerinde word hizalama için OptionsàCompileràCode generationàWord alignment seçilir. Default durum byte hizalama biçimidir.

Dword Hizalaması(dword alignment)

32 bit işlemcilerde hizalama seçeneği byte ya da dword biçimindedir. Derleyici dword hizalamasına getirildiğinde şunlar olur:

1. Fonksiyon blok girişlerinde SP her zaman dword katlarındadır.
2. char türü dışındaki tüm türler 4'ün katlarındaki adreslere yerleştirilir.
3. Yapı değişkenlerinin kendisi de 4'ün katlarındaki adreslerden başlayacak biçimde yerleştirilir.
4. Yapının toplam uzunluğu 4'ün katları olacak biçimde tamamlanır.

Diziler hangi türden olurlarsa olsunlar aralarında boşluk bulundurulmadan yerleştirilirler. Ancak dizinin başlangıcı word hizalamada çift adreste, dword hizalamada ise 4'ün katlarındaki adrestedir. Visual C++ 6.0 derleyicinde bir proje yaratıldığında hizalama default olarak dword biçimindedir.

Hizalama Problemleri

Hizalama yüzünden tipik birkaç problem söz konusu olmaktadır.

1. Dosyada bulunan bir veri yapısının bir yapı değişkeni içine transfer edilmesi durumunda word ya da dword hizalaması kullanıldığında çakışmama problemi ortaya çıkabilir. Örneğin .x biçiminde bir dosya formatı söz konusu olsun. Dosyanın ilk byte'ları şöyle dizilmiş olsun:
   1 byte ID
   2 byte MAXCOLOR
   4 byte TOTALCOLOR
   Dosyadaki bu durum XFORMAT biçimindeki bir yapı ile ifade edilip, fread ile okuma yapılacak olsun:

```
typedef struct _XFORMAT {
    BYTE id;
    WORD maxcolor;
    DWORD totalcolor;
} XFORMAT;

XFORMAT x;
...
fseek(f, 0, 0);
fread(&x, sizeof(XFORMAT), 1, f);
```

x.maxcolor = ?

Bu kod word hizalama seçeneğinde derlenip çalıştırılırsa yapı elemanlarıyla dosyadaki değerler çakışamayacaktır. Problemi çözmek için derlemenin byte hizalamasında yapılması gerekir. Tabii dosya formatını tasarlayan böyle bir problem çıksın istemiyorsa bütün elemanları çift uzunlukta tanımlayarak hizalama problemini daha tasarım aşamasında ortadan kaldırabilir.

2. Bazen farklı bir hizalama biçimi kullanılarak çeşitli fonksiyonlar derlenerek kütüphaneye yerleştirilmiş olabilir. Bu fonksiyonlar özellikle bir yapı değişkeninin adresini parametre olarak alıyorsa hizalama problemi ortaya çıkabilir. Örneğin kütüphaneye yerleştirilmiş olan fonksiyon word hizalaması kullanılarak derlenmiş olabilir. Biz o fonksiyonu byte hizalaması kullanılan bir koddan çağırmaya çalışırsak problem ortaya çıkar. Bu durumda tasarımcı olarak bir yapı kullanan fonksiyonu kütüphaneye yerleştireceksek yapının elemanlarını char yerine int türünden ifade etmeliyiz. Çünkü int türü mikroişlemcinin bir register uzunluğu kadar seçilir ve hizalama problemi oluşturmaz.

ANSI C standartlarında hizalama konusunda belirleyici bir ifade kullanılmamıştır. Yalnızca yapıların ele alındığı bölümde hizalama yüzünden yapı elemanları arasında boşlukların olabileceğinden bahsedilmiştir. Hizalama kuralları tamamen derleyiciyi yazanlara bırakılmıştır. Bu yüzden farklı derleyicilerin farklı hizalama kuralları olabilir.

Bu durumda bir derleyicide yazılmış ve derlenmiş modülün byte hizalaması dışında bir hizalama kullanılmışsa probleme yol açabileceği unutulmamalıdır.
Yapı göstericisi kullanarak yapı elemanlarına ok operatörüyle ulaşmak aşağıdaki gibi olmaktadır.

C kodu	Assembler karşılığı(16 bit)
struct SAMPLE {	_Set proc near
int a;	push bp
long b;	mov bp, sp
char c;	push si
};	mov si, [bp + 4]
	mov word ptr [si], 10
void Set(struct SAMPLE *p)	mov word ptr [si + 2], 20
{	mov word ptr [si + 4], 0
p->a = 10;	mov word ptr [si + 6], 97
p->b = 20;	pop si
p->c = ‘a’;	pop bp
}	ret
	_Set endp
void main(void)	
{	_main proc near
struct SAMPLE x;	push bp
	mov bp, sp
Set(&x);	sub sp, 7
}	lea ax, [bp – 7]
	push ax
	call _Set
	pop cx
	mov sp, bp
	pop bp
	ret
	_main endp
	/*****yerel008.asm*****/

	Assembler karşılığı(32 bit)

	_Set proc near
	push ebp
	mov ebp, esp
	mov eax, [bp + 8]
	mov dword ptr [eax], 10
	mov dword ptr [eax + 4], 20
	mov byte ptr [eax + 8], 97
	pop ebp
	ret
	_Set endp
	
	_main proc near
	push ebp
	mov ebp, esp
	sub esp, 9
	lea eax, [bp – 9]
	push eax
	call _Set
	pop ecx
	mov esp, ebp
	pop ebp
	ret
	_main endp

/*****yerel008.c*****/	

Fonksiyon göstericileri dolaylı call komutuyla ifade edilir.

C kodu	Assembler karşılığı

void func(void)
{
}
 	
void main(void)
{
void (*p)(void);
 	
p = func;
p();
}
	_func proc near
	ret
	_func endp
	
	_main proc near
	push bp
	mov bp, sp
	sub sp, 2
	mov word ptr [bp – 2], _func
	call word ptr [bp – 2]
	mov bp, sp
	pop bp
	ret
	_main endp

NOT: Kod sembolleri de bir bellek bölgesine mov komutuyla doğrudan atanabilir. Kod sembolleri [] içerisinde değil, doğrudan bir sayı belirtmektedir.

C++'ta Bir Sınıfın Üye Fonksiyonlarının Sembolik Makine Dilinde Çağırılması

Bilindiği gibi C++'ta bir üye fonksiyon sınıf nesnesi yoluyla nokta operatörünü kullanarak ya da sınıf göstericisi yoluyla ok operatörü kullanılarak çağırılır. Üye fonksiyon aslında normal bir fonksiyondur. Yalnızca mantıksal bakımdan sınıfla ilişkilendirilmiştir. Derleyici üye fonksiyon hangi sınıf nesnesiyle çağırılmışsa onun adresini gizlice this göstericisi biçiminde geçirmektedir. This göstericisi fonksiyonun ilk parametresi olarak

gizlice geçirilir. Yani 16 bit yakın modellerde [bp + 4]'te, uzak modellerde ise [bp + 6]'dadır. Derleyiciler genellikle this göstericisini bir indeks register'a çekip, sınıfın veri elemanlarına normal bir yapı gibi erişirler. Bir üye fonksiyon içerisinde başka bir üye fonksiyonun çağırılması durumunda da fonksiyon aldığı this adresini, tekrar stack'e push ederek diğer üye fonksiyonu çağırmaktadır.

<table>
<tr><th>C++ kodu</th><th>Assembler karşılığı</th></tr>
<tr><td>

```
class Date {
public:
   void Disp(void) const;
   void Verify(void) const;
   void Set(int d, int m, int y);
private:
   int day, month, year;
};

void Date::Disp(void) const
{
   printf("%d/%d/%d\n", day, month, year);
}

void Date::Verify(void) const
{
}

void Date::Set(int d, int m, int y)
{
   day = d;
   month = m;
   year = y;
   Verify();
}

void main(void)
{
   Date x;

   x.Set(18, 1, 2001);
   x.Disp();
}
```

</td><td>

```
.data
s@ db "%d/%d/%d", 10, 0

.code
@Date@Set proc near
   push bp
   mov bp,sp
   push si
   mov si,word ptr [bp+4]
   mov ax,word ptr [bp+6]
   mov word ptr [si],ax
   mov ax,word ptr [bp+8]
   mov word ptr [si+2],ax
   mov ax,word ptr [bp+10]
   mov word ptr [si+4],ax
   push si
   call @Date@Verify
   pop cx
   pop si
   pop bp
   ret
@Date@Set endp

@Date@Disp proc near
   push bp
   mov bp,sp
   push si
   mov si,word ptr [bp+4]
   push word ptr [si+4]
   push word ptr [si+2]
   push word ptr [si]
   mov ax,offset s@
   push ax
   call near ptr _printf
   add sp,8
   pop si
   pop bp
   ret
@Date@Disp endp

@Date@Verify proc near
   ret
@Date@Disp endp

_main proc near
   push bp
   mov bp,sp
   sub sp,6
   mov ax,2001
   push ax
   mov ax,1
   push ax
```

</td></tr>
</table>

	mov ax,18 *push ax* *lea ax,word ptr [bp-6]* *push ax* *call @Date@Set* *add sp,8* *lea ax,word ptr [bp-6]* *push ax* *call @Date@Disp* *pop cx* *mov sp,bp* *pop bp* *ret* *_main endp*
/*****yerel010.cpp*****/	/*****yerel010.asm*****/

C'de Değişken Sayıda Parametre Alan Fonksiyonların Sembolik Makine Dilinde Yazımı

C'de printf, scanf gibi bir grup fonksiyon istenildiği kadar çok sayıda parametre alabilmektedir. Değişken sayıda parametre alan fonksiyonların prototipleri "..."(ellipsis) içermektedir. Örneğin:

```
func(int x, ...);
```

Burada fonksiyon en azından bir parametre almak zorundadır. Bu parametreden sonra istenildiği kadar çok parametre alabilir. Bu durumda printf fonksiyonunun prototipi şöyle olmalıdır:

```
int printf(const char *format, ...);,
```

Değişken sayıda parametre alan fonksiyonlar için derleyici yalnızca zorunlu parametreleri kontrol eder. Diğer parametreleri kontrol etmez. Ancak fonksiyon çağırılırken ifade içerisinde yazılmış olan bütün parametreler sağdan sola diğer fonksiyonlarda olduğu stack'e push edilir. Ancak prototipte belirtilmeyen parametreler için aktarım sırasında "int türüne yükseltme" ve "double türüne yükseltme" kuralları uygulanır. Örneğin char int biçiminde, float parametre ise double türü biçiminde stack'e push edilecektir.

Örneğin aşağıdaki gibi çağırılmış olsun:

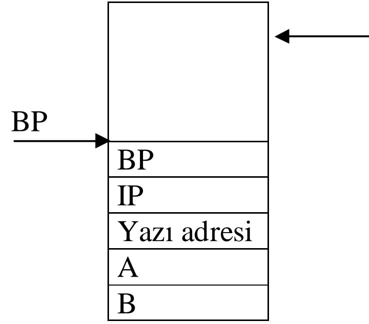
```
int a, b;
...
printf("a = %d, b = %d\n", a, b);
```

Derleyici fonksiyonu şöyle çağıracaktır:

```
push b
push a
push yazı_adresi
call _printf
add sp, 6
```

Programın akışı printf fonksiyonuna girip stack frame düzenlendikten sonra stack bölgesinin görüntüsü şöyle olacaktır:

SP



printf fonksiyonunu tasarlayan kişi yalnızca zorunlu parametre olan yazı adresinin [bp + 4]'te olduğundan emin durumdadır. Ancak fonksiyon çağırıldığında kaç parametrenin stack'te olduğunu bilemez. printf birinci parametredeki yazıyı yorumlayarak fonksiyonun kaç parametreyle çağırıldığını anlar. Yapacağı şey % ile başlayan format karakterlerini saymaktır. Tabii printf fonksiyonu aynı zamanda stack'e push edilmiş olan parametrelerin uzunluklarını da bilmek zorundadır. Bunu format karakterine bakarak anlar. Örneğin ilk parametre %ld ise long bir sayı yazdıracaktır. Bu sayıyı [bp + 6]'dan başlayarak 4 byte çekecektir. Sonraki parametre %d ise [bp + 10]'dan 2 byte çekecektir. Böylece bir format karakteri uygunsuz girildiğinde diğer parametrelerin de yanlış display edilme olasılığı vardır.

Değişken sayıda parametre alan sayılarda fonksiyona yazan kişi fonksiyonun o anda kaç parametreyle çağırlmış olduğunu bilmek zorundadır. Bunun için birinci parametreyi inceler ve duruma göre stack'ten gerekli sayıda parametreleri çeker. Örneğin prototipi aşağıdaki gibi olan bie fonksiyon tasarlayalım:

Fonksiyonun birinci parametresi kaç sayı girildiğini göstersin, fonksiyon diğer parametrelerin toplamına geri dönsün.

Bu fonksiyonun makine dilinde yazılımı şöyle olabilir:

```
.model small
.CODE
_sum proc near
        push    bp
        mov     bp,sp
        push    si
        xor     ax,ax
        mov     cx,[bp+4]
        mov     si,6
        jmp     @1
@2:
        add     ax,[bp+si]
        add     si,2
@1:
        dec     cx
        jnz     @2
        pop     si
        pop     bp
        ret
_sum endp
```

Bazen değişken sayıda parametre alan fonksiyonlarda birinci parametre kaç parametreyle çağırıldığı bilgisini içermez. fonksiyon son parametrenini özel bir değer olmasından hareketle tasarlanır. Örneğin yukarıdaki kodu birinci parametrenin anlamını değiştirip son paramt-etre 0 oluncaya kadar sayıların toplamı biçiminde ifade edebiliriz. Yukarıdaki sum() fomksiyonunu son parametrenin yerini 0 ile tesbit ederek yeniden tasarlayınız. C'de değişken sayıda parametre alan sayıların en az bir paramteresi olmak zorundadır. Bu yüzden fonksiyonun prototipi yine;

```
int sum(int n,...);
```

biçiminde belirtilmelidir.

C'de değişken sayıda parametre alan fonksiyonların yazımı:

C'de değişken sayıda parametre alan foksiyonlar standart makrolarla yazılmaktadır. Bu makrolar kullanılırsa kod sistem bağımsız olarak çalışabilir. Aslında biz sistemi iyi yanıyorsak birinci parametrenin adresini alarak stack zincirini takip edebikiriz. Ancak böyle bir çözüm taşınabilir değildir.

Örnek:

```
int sum(int n,...)
{
        int total = 0;
        int i;
        int *pnum;

        pnum = &n + 1;
        for(i = 0; i < n; ++1)
                total += pnum++;
        return total;
}

void main(void)
{
   printf("%d\n", sum(5,3,2,6,7,4));
}
```

Bu işlemler standart makrolar kullanılarak yapılmalıdır.

1) va_list türünden bir değişken tanımlanır.

```
typedef void *va_list
```

2) va_start makrosu va_list türünden değişken ve birinci parametre ismiyle çağırılır.

```
#define va_start(ap,parmN)        ((ap) = (char *) &parmN + size(parmN))
```

Burada kullanılan "size" başka bir makrodur. Birinci parammetrenin türü char ise, bir sonraki parametrenin türü 1 değil 2 byte sonra olmalıdır. va-start makrosu çağırıldıktan sonra makronun birinci parametresinde stack'teki ilk parametreden sonraki parametrenin adresi vardır.

Bu makro ilk parametreden sonraki parametrenin adresini makronun birinci pararnetresiyle belirtildiği göstericiye atar. Bu makro aşağıdaki gibidir:

```
#define size(x)      ((sizeof(x)+ sizeof(int) - 1) & ~(sizeof(int) -1))
```

3) va_arg makrosu birinci parametresinde va_list türünden bir değişken, ikinci parametresinde tür belirten bir sözcükle çağırılır.
Örneğin:

x = va_arg(ap, int);

Bu makro birinci parametreden sonraki parametrenin belirtilen türde olduğunu varsayarak onu elde eder ve makoronun birinci parametresiyle belirtilmiş olan nesneyi sonraki parametre için günceller.

```
#define va_arg(ap,type)        ((char *)(ap) += size(type), *(type)((char *)(ap)-
size(type)))
```

İşlem bitirildikten sonra **va_end** makrosu **va_list** türünden değişkenle çağırılır. Bu çağırmaya Intel sisteminde gerek yoktur. Dolayısıyla bu makro yerine boşluk atanarak silinir. Ancak başka işlemlerde bu makro gerekebilir.

*#define **va_end(ap)**   ((void)0)*

va_arg için not: ap önce artırılır ve artırılmış adres değeri ap'ye atandı.Şimdi normalde gösterilmesi istediğimiz parametrenin 1 ötesinde (parametrenin kendi boyu kadar) Ama geriye önceki adres değerindeki bilgiyle döner. ama poinet'a atama yapılmaz. (Virgülün sağ tarafı) Böylece güncelleme işlemi yapılmış olur.

```
int sum(int n,....)
{
        int total = 0, i;
        va_list ap;
        va_start(ap, n);
        for(int i = 0; i < n; ++i)
                total += va_arg(ap, int);
        va_end(ap);
        return total;
}
```

Bütün bu makrolar **stdarg.h** içindedir.

C'de Değişken Sayıda Parametre Alan Fonksiyonlara Örnekler

1) void avg(const char *msg, ...);

Bu fonksiyon 0 parametresini görene kadar parametre olarak girilen sayıların aritmetik ortalamasını ekrana yazdırır. Fonksiyonun birinci parametresi aritmetik ortalama yazdırılmadan önce ekrana çıkacak olan yazıyı belirtir.

```
void main(void)
{
        arg("average", 10, 20, 30, 40);
}

void avg(const char *msg, ...)
{
        double  average, total = 0, val;
        int  count = 0;
        va_list   va;
        va_start(va, msg);

        while((val = va_arg(va, int)) != 0) {
                total += val;
                ++count;
        }
        average = total / count;
        printf("%s % lf\n", average);
        va_end(va);
}
```

2) İşletim sisyemi yüklendiğinde hatta makine satın alındığında bir karakteri ekrana basan fonksiyon kesme biçiminde vardır. Bu fonksiyonun putchar() fonksiyonu olduğunu varsayalım. Yalnızca putchar() fonksiyonunu kullanarak %d ve %s formatlarına duyarlı olan printf() fonksiyonunu myprintf() ismiyle yazınız. Aşağıdaki kodla test ediniz.

```
void myprintf(const char *format, ...);

void main(void)
{
        int a = 100;
        char *msg = "deneme";
        myprintf(""a = %d\n%s\n, a, msg);
}
```

Açıklama1; Int bir sayıyı ekrana yazdıran fonksiyon şöyledir.

```
void printd(int n);
{
	if(n < 0) {
		putchar('-');
		n = -n;
	}
	if(n / 10)
		printd(n / 10);
	putchar(n % 10 + '0');
}
```

Açıklama2: Fonksiyonun birinci parametresindeki yazı incelenir. % karakteri görüldüğünde yanındaki karaktere bakılır. Yanındaki karakter "d" ise stack'ten int bir argüman çekilir ve printd() fonksiyonuyla yazılır. "s" ise stack'ten char * türünden argüman çekilir o argüman NULL görülene kadar yazdırılır. %'nin yanındaki karakter bunlardan biri değilse % karakteriyle yanındaki karakteri aynı biçimde yazar.

Uzak Göstericilerin Yüklenmesi

Bilindiği gibi DOS ortamında segment 64 K ile sınırlıdır. Bu durumda fiziksel adreste istenilen adresin görülebilmesi için segment'in de değiştirilmesi gerekir. Uzak gösterici 4 byte uzunluğunda bir bellek alanıdır. Bu bellek alanının düşük anlamlı 2 byte'ında offset, yksek anlamlu 2 byte'ında segment bilgisi bulunur. WIN32 ve UNIX sistemlerinde uzak gösterici kullanılmaz. Bütün segment değeri 0 kabul edilir. Belleğin her yerine yalnızca offset bilgisiyle erişilir. Offset ise 4 byte uzunluktadır. DOS'ta bellek modeli medium, ladge ya da huge ise far anahtar sözcüğü kullanılmasa bile göstericiler uzak göstercilerdir. Bir uzak göstericiyi kullanabilmek için düşük anlamlı 2 byte'ını index register'a, yüksek anlamlı 2 byte'ını ise segment register'a yerleştirmek gerekir. Uzak göstericilerle işlem yaparken hangi segment register'ı seçmeliyiz. CS ve SS segment register'ı kullanmanın anlamı yoktur. DS ise .data bölümünün başlangıç adresini gösterdiğine göre eğer değiştirilecekse bile yeniden bu adrese yüklenmesi gerekir ki bu da zaman kaybı anlamına gelir. İdeal yöntem ES segment register'ını kullanıp segment yüklemesi yapmaktır.

```
char far *p = (char far *) 0xB8000000;
*p = 'a';
```

```
1) mov	si, ...
2) mov	es, ..
3) mov	es:[si],...
```

lds, les, lfs, lys Makine Komutları:

Kullanım Biçimleri:

```
lds	reg,mem
les	reg,mem
lfs	reg,mem
```

lgs	reg,mem

lfs ve lgs komutları 386 ve sonrasında geçerlidir.

Bu komutlar belirtilen bellek adresinden başlayarak ilk iki byte'ı belirtilen register'a sonraki 2 byte'ı ise komtta belirtilen segment register'a yüklerler. Böylece uzak göstericiler tek hamlede yüklenmiş olurlar. Komutun operantı sembolik makine dilinde DD (define double word) türünden bir data sembolü içermelidir. Eğer data sembolü DD türünden değilse **dword ptr** ile tür değiştirilmesi yapılmalıdır. Ancak debugger'larda dword ptr dönüştürülmesi olmadan da komut verilebilmektedir.

C Derleyicilerinin Uzak Gösterici İşlemlerini Ele Alış Biçimi

```
_main proc  near
      push  bp
      mov   bp,sp
      sub   sp,4
      push  si
      mov   word ptr [bp-4],0
      mov   word ptr [bp-2],0xB800h
      les   si,dword ptr [bp-4]
      mov   byte ptr es:[si],61h
      pop   si
      mov   sp,bp
      pop   bp
      ret
_main endp
```

CPU'nun Durumunun Saklanması

Birden fazla kodun aparalel bir biçimde zaman paylaşımlı olarak çalışabilmesi için bir koda ara verildiğinde o kodun tüm register bilgilerinin bellekte bir yerde saklanması gerekir. Ger dönüş işlemi sırasında saklanan yerden bilgiler alınır ve register'lara geri yüklenir. Tabii kodun çalışmasına ara verilip yeniden dönüldüğünde kodun kullandığı bellek alanının bozulmamış olması gerekir. Eğer bozulduysa yeniden düzeltilerek kodun çalışmasına devam edilmesi gereklidir. Çok işlemli bir işletim sistemi tipik bir biçimde şöyle davranır:

1) Bir progran çalıştırılacağı zaman bir sistem fonksiyonu kullanılır. Örneğin bu fonksiyonun WINDOWS sistemlerindeki ismi **CreateProcess()** 'dir.

2) Program belleğe yüklendiğinde ismi artık proses olur. Prosesi yaratan fomksiyon proses bilgilerini tutmak için bir handle alanı tahsis eder. Bu alana **"Process Database"** ya da **"Process Table"** denmektedir. Process Database içinde prosesle ilgili bütün bilgiler tutulmaktadır. Örneğin;

- Prosesin açmış olduğu dosyalar,
- Programın öalıştırılmasında kullanılan komut satırı argümanları ,
- Senkronizasyon nesnelerinin bilgileri,
- Proses kesildiğinde PCU register'larının konumları,

- Prosesin EXIT kodu,
- vs...

3) 8254 yoluyla IRQ oluştuğunda işletin sistemi kontrolü ele alır. O anda çalışmakta olan prosesin register bilgilerini "Process Database" içinde bir bölgeye yazar. Çalışmaya devem edecek olan sıradaki prosesin register bilgilerini o prosesin "Process Databese"inde alarak CPU register'larına yükler. Bu biçimde kodlar arasındaki geçişe **"Context Switch"** ya da **"Task Switch"** denir. Prosesler arsındaki geçiş süresine **"quanta süresi"** denir. WIN32 sistemlerinde bu süre 20 ms kadardır. Quanta süresi çok küçük olduğunda proseslerin çalışması yavaşlar. Çok yüksek olduğunda prosesin dışarıdaki olayları takip edebilme yeteneği zayıflar.

CPU Konumunun Saklanıp Geri Yazılması Sırasında Dikkat Edilecek Durumlar

1) CS ve IP değerlerinin saklanıp yüklenmesi:

CS register'ı doğrudan MOV komutuyla belleğe alınabilir. Ancak IP ile MOV komutu uygulanamaz. Aşağıdaki program parçasıyla CS ve IP'yi saklayabiliriz:

```
        mov     mem1,cs
        call    @1
@1:
        pop     mem2
```

Bazen saklanacak IP değeri ileride başka bir bölgeye ilişkin olması istenebilir. Bunun için aşağıdaki gibi bir yöntem izlenebilir:

```
        mov     mem1,cs
        call    @1
@1:
        add     ax,offset @2-offset @1
        ---
        ---
@2:
```

Barada IP için saklanan değer @2 pozisyonudur. CS ve IP değerlerinin geri yüklenmesi aşağıdaki gibi yapılamaz:

```
mov     cs,mem1
push    mem2
ret
```

Çünkü;
mov   cs,mem1
işlemniden sonra artık CPU başka yerden komut almaya başlayacaktır. CS ve IP register'larının aynı anda yüklenmesi gerekir. Bunun bir kaç yöntemi vardır.

a)

```
push   mem1(CS)
push   mem2(IP)
retf
```

b) IP ve CS değerleri 4 byte'lık bir alana yazılır ve bu alan kullanıllarak **"segmentler arası dolaylı jump"** komutu uygulanır.

```
mov   ax,mem2
mov   dest,ax
mov   ax,mem1
mov   dest+2,ax
jmp   dword ptr dest
```

CS, IP yüklemeleri registe yüklemelerinin en sonunda yapılmak zorundadır. Çünkü CS:IP değiştiği anda kod kendini başka bir yerde bulacaktır.

2) SS:SP register'larının saklanması ve yüklenmesi:

Hem SS hem de SP MOV komutuyla birlikte kullanılabilir. Bu durumda aşağıdaki gibi bir saklana olası gözükmektedir:

```
mov   mem1,sp
mov   mem2,ss
```

Gerçekten saklama işlemi böyle yapılabilir. Ancak geriye yükleme işlemi aşağıdaki gibi yapılabilir mi?

```
mov   ss,mem1
mov   sp,mem2
```

Pek çok IRQ stack olarak programcının yani o anda kesilen programın stack'ini kullanır. Bu yüzden program içinde push işlemi yapmasak bile yine de IRQ'lar ve diğer kesmeler için stack bulundurmalıyız. Yukarıdaki işlemde;

mov ss,mem1

komutundan sonra bir IRQ oluşursa stack olarak kullanılacak bölge güvensiz bir bölge olur. Bunu engellemek amacıyla 8086 ve 8080m işlemcilerinde donanım kesmeleri disable edilmesi gereklidir:

```
        mov   mem1,ss
        mov   mem2,sp
cli
        mov   ss,mem1
        mov   sp,mem2
sti
```

Ancak 80286 ve sonrasında bu işleme gerek yoktur. Çünkü mikroişlemci SS segment register'ına yükleme yapıldığında sonraki komut sonlanana kadar otomatijk olarak kesme kabul etmemektedir. Tabii SS register'ına yükleme komutdan sonra hemen SP register'ına yükleme yapmak gerekir.

Bir kesme oluştuğunda kesmenin SS ve SP değerlerini değiştirerek kendi stack'ini kullanması problemli olabilir. Bu tür durumlarda iç içe kesme çağırmalarında sistem kilitlenebilir. Örneğin DOS'un INT21h

kesmesinin fonksiyonları SS ve SP değerlerini önceden belirlenmiş sabit bir yere çekerler. bu durumda iç içe DOS kesmesi çağırıldığında içteki kesmden çıkıldığında dıştaki kesmenin stack bölgesi bozulmuş olur. DOS'ta bellekte kalan programlar yazılırken bir tuşa basıldığında IRQ1 (INT9) yardımıyla kod ele geçirilir.Bu kod içinde INT21h kesmesi çağırıldığında eğer kesilen kod DOS fonksiyonu içindeyse çıkışta kilitlenme yaşanır. Bellekte kalan programların yazılması için DOS içinde INDOS denilen bir bayrak değişkeni tutulmuştur. Bu bayrak INT21h içindeyken 1, değilken 0 yapılır. Böylece bellekte kalan program aktive edilirken bu bayrağa bakılır. Bu bayrak 1 ise aktive edilmez. Eğer aktive edilecekse DOS fonksiyonları kullanılmaz. Bayrak 1 ise IRQ0 (INT8) hook edilir. Her INT8 oluştuğunda INT8 o bayrağa yeniden bakar. BAyrak 0'a düşünce aktivasyonu INT8 yapar.

3) Diğer Register'ların saklanması ve yüklenmesi:

Dİğer register'ların yüklenmesi sırasında pek çok kombinasyon bozucu etki yaratabilir. Örneğin DS yüklendiği zaman artık SI ve DI ındex register'ları farklı bir yeri göstrecektir.

```
mov   ds,[si+n]
mov   bp,[si+m]
```

Bu tür durumlarda segment yükleme işlemleri tercih edilebilir.

```
mov   ax,ds
mov   es,ax
mov   ds,[si+n]
mov   bop,es:[si+m]
```

C'de Yerel Olmayan Dallanmalar

Bilindiği gibi C'de goto komutunun etiketi fonksiyon faaliyet alanına ilişkindir. Yani goto deyimiyle başka bir fonksiyonun içine dallanma yapılamaz. Bunun C'de yasaklanmış olmasının nedeni şu zorunluluklardan kaynaklanmaktadır:

1) Bir fonksiyondan başka fonksiyona goto yapılabilseydi dallanılan fonksiyonda hiç bir yerel değişkeni ve hiç bir parametre değişkeni kullanamazdık. Çünkü goto basit bir jump komutudur. Dallanılan kodda ise yerel değişkenlere erişmekte kullanılan [bp-n] , parametrelerre erişmekte kullanılan [bp+m] komutları vardır. Şimdi bu komutların hiç bir geçerliliği kalmaz. Çünkü BP register'ı goto komutunun yerleştirildiği fonksiyona ilişkindir. Zaten parametreler stack'te değildir, yerel değişkenler için yer ayrılmamıştır. Oysa başka bir fonksiyona çağırma yöntemiyle dallandığımızda parametreler doğru bir şekilde push edilmiş olur, **"stack frame"** kurulmuş olur.

2) Başka bir fonksiyona dallanma yapılabilseydi stack de dengelenmemiş olarak kalacaktı. Çünkü çıkış sırasında

```
pop    bp
mov    sp,bp
```

gibi komutlar görülemeyecekti. Ancak C'de yine de başka bir fonksiyona dallanma olasıdır. Bunun için prototipleri SETJMP.H içinde bulunan SETJMP ve LONGJMP komutları bulunur.

setjmp Fonksiyonu

Bu fonksiyon çağırıldığı yerdeki CPU konumunu alarak parametresiyle belirtilen jmp_buf yapısı içerisinde saklar. Prototipi:

int setjmp(jmp_buf jmpb);

jmp_buf bir typedef ismidir:

```
typedef struct {
        unsigned j_sp, j_ss;
        unsigned j_flag, j_cs;
        unsigned j_ip, j_bp;
        unsigned j_di, j_es;
        unsigned j_si, j_ds;
}jmp_buf[1];
```

Görüldüğü gibi jmp_buf aslında bir elemanlı bir yapı dizisine ilişkin bir tür ismidir. Bu durumda setjmp fonksiyonunun parametresi bu yapı türünden bir gösterici olur. setjmp fonksiyonundan iki çıkış vardır:

1. İlk çıkış: Bu durumda setjmp 0 ile geri döner ve CPU register'larının konumu kaydedilmiştir.
2. Programın başka bir yerinde longjmp ile yapılan çıkış. longjmp yapıldığında akış sanki setjmp'dan çıkıyormuş gibi geri döner. Bu durumda fonksiyon 0 dışı bir değerle geri döner. O halde çıkışın longjmp dolayısıyla olup olmadığı aşağıdaki gibi anlaşılabilir:

```
if (setjmp(jmpb))
{
    ...
    ...
    ...
    exit(1);
}
```

longjmp Fonksiyonu

Bu fonksiyonla akış, daha önce geçilmiş olan ve setjmp ile kaydedilmiş olan duruma geri döner. Prototipi:

void longjmp(jmp_buf jmpb, int retval);

Fonksiyonun birinci parametresi daha önce setjmp ile saklanılan CPU bilgileridir. İkinci parametre geri dönüldüğünde setjmp'dan hangi değerle çıkılacağını belirlemekte kullanılır. Normal olarak bu değerin 0 dışı bir değer olması gerekir. Zaten 0 verilse bile fonksiyon bunu 0 dışı bir değere çekmektedir. setjmp ve longjmp fonksiyonlarının prototipleri setjmp.h dosyası içerisindedir.

Neden longjmp Kullanılır?

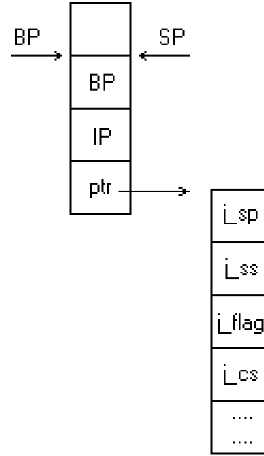
C'de programın çeşitli yerlerinde oluşabilecek problemlerin tek bir noktadan kontrolü için longjmp tercih edilebilir. longjmp işlemi C++'taki exception handling mekanizması gibidir. Ancak C++'ta throw işlemi ile longjmp yapıldığında o zamana kadar yaratılmış olan tüm yerel sınıf nesneleri için bitiş fonksiyonu da çağırılmaktadır.

longjmp Örneği:

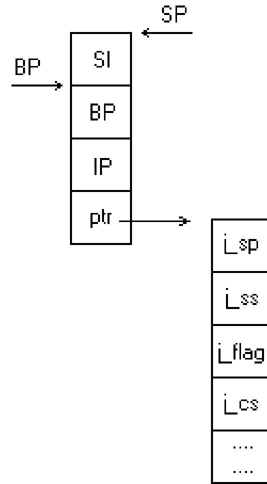
```
/*****longjmp0.c*****/
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf jmpb;
void sample(void);
void func(void);

void main(void)
{
    switch(setjmp(jmpb))
    {
        case 1:
            printf("Cannot open file..\n");
            exit(1);
        case 2:
            printf("Not enough memory..\n");
            exit(1);
    }
    func();
}

void func(void)
{
    FILE *f;

    sample();
    if ((f = fopen("abcdzyx", "r")) == NULL)
        longjmp(jmpb, 1);
    printf("Success..\n");
}

void sample(void)
{
    char *p;

    p = (char *)malloc(0);
    if (p == NULL)
        longjmp(jmpb, 2);
    printf("Success..\n");
}
/*****longjmp0.c*****/
```

setjmp ve longjmp Fonksiyonlarının Sembolik Makine Dilinde Yazımı

setjmp fonksiyonuyla CPU'nun genel amaçlı register'larının konumlarını saklamaya gerek yoktur(Aslında ES ve flag register'larının da saklanmasına da gerek yoktur). setjmp fonksiyonu çağırıldığında stack'in durumu şöyledir:



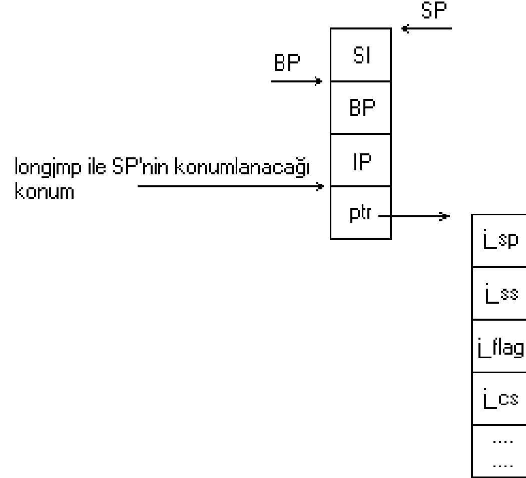
O halde BP + 4'ten elde edilen değer yerleştirme yapılacak yapının adresidir. Daha sonra SI push edilip bu adres SI'ya alınmıştır. Artık yapı elemanlarına [SI + n] ifadesiyle erişebiliriz.



IP olarak setjmp fonksiyonunun çıkışındaki değer saklanmıştır. Derleyici setjmp fonksiyonunu çağırırken komuttan bir sonraki yeri zaten stack'e push etmiştir. Örnek programda:

mov ax, [bp + 2]
mov [si + 8], ax

yapmakla setjmp fonksiyonunun çıkışındaki durumu saklamış oluyoruz. longjmp ile geriye dönüş yapıldığında SP'nin durumu setjmp fonksiyonunun çağırılmadan önceki durumu olmalıdır.



Böylece longjmp ile geri dönüldüğünde artık akış setjmp'dan sonraya düşmüştür. Çalıştırılacak kod derleyicinin stack dengeleme komutudur. Yani setjmp çıkışında derleyici

add sp, 2

gibi bir komut uygulayacak, böylece stack orijinal değerine geri dönecektir. setjmp çıkışında AX 0 olmalıdır. longjmp ile register'lar belirlenen sırada yüklenirler. CS ve IP'yi aynı anda değiştirmek için *FAR JMP* ya da *RETF* komutları uygulanabilir. Örneğimizde RETF tercih edilmiştir.

/****longjmp1.asm*****/
/****longjmp2.prj*****/
/****longjmp3.c*****/

DOS'ta Uzak Modellerde İşlemler

Bilindiği gibi DOS'ta medium, large ve huge modeller uzak modellerdir. Bellek modellerinin teknik açıklaması ayrıntılı segment tanımlamalarından sonra yapılacaktır. C derleyicilerinin başlangıç kodları bellek modellerine göre segment register'ları çeşitli değerlerde tutmaktadır.

Tiny Model

Bu modelde tüm segment register'ları aynı değerdedir. Bu durumda programda CS = DS = SS olduğu için toplam kod, data ve stack 64Kb'ı geçemez.

Small Model

Bu modelde CS bağımsız bir 64Kb'ı gösterir. DS = SS biçimindedir. Program kodu 64Kb'ı geçemez, toplam statik data ve stack 64Kb'ı geçemez.

Medium Model

Bu modelde kod 64Kb'ı geçebilir. Bu durumda bütün fonksiyon çağırmaları *far call* ile yapılır. Fonksiyon göstericileri default olarak uzak göstericidir. Yine bu modelde DS, SS'e eşittir. Bu durumda default göstericiler yakın göstericilerdir. Programın stack ve data bölümleri toplamı 64Kb'ı geçemez.

Compact Model

Bu modelde programın tek bir kod segment'i vardır. Ancak DS ile SS birbirinden ayrılmıştır. Bu durumda toplam statik data ayrı bir 64Kb, toplam stack yine ayrı bir 64Kb

içerisindedir. Fonksiyon çağırmalar default near call biçimindedir. Ancak göstericiler default uzak gösterici olmak zorundadır.

Large Model

Bu modelde birden fazla kod segment'i vardır. Yani program kodu 64Kb'ı geçebilir. DS ile SS birbirlerinden ayrılmıştır. Yani toplam statik data ve stack ayrı ayrı 64Kb uzunluğundadır. Fonksiyon çağırmaları default far call biçimindedir ve default göstericiler yine uzak göstericilerdir.

Huge Model

Bu modelde programın birden çok data segment'i ve stack'i olabilir. Tabii stack 64Kb'ı yine geçemez. Fonksiyon çağırma işlemi default olarak far call biçimindedir. Default göstericiler uzak göstericilerdir.

Uzak Modellerde Data Göstericileriyle İşlemler

Uzak modellerde DS ile SS birbirlerinden ayrıldığı için stack'teki bir göstericiye data segment'teki bir değişkenin adresi, data segment'teki bir göstericiyeyse stack'teki bir değişkenin adresi atanabilir. Bu durumda bütün göstericiler default uzak göstericidir. Örneğin:

```
/*****far0000.c*****/
void func(int *p)
{
        *p = 20;
}

void main(void)
{
        int a = 10;

        func(&a);
}
/*****far0000.c*****/
```

proc bildiriminin near ya da far olmasının etkisi şudur: far proc bildiriminde call makine komutunda far anahtar sözcüğü belirtilmese bile bu komut default olarak far biçiminde ele alınır. Oysa near proc bildiriminde default olarak near call biçiminde ele alınacaktır. Aynı zamanda far proc bildirimi içerisinde yalnızca *ret* makine komutunu yazdığımızda derleyici bunu otomatik olarak *retf* olarak algılamaktadır. Oysa near proc bildirimi içerisinde ret makine komutları yine *ret* olarak ele alınmaktadır.

Derleyiciler başka bir segment'e erişmek istediklerinde CS, DS, SS segment register'larıyla oynamak yerine ES segment register'ını yükleyip segment yükleme komutlarıyla erişim yaparlar. C derleyicileri fonksiyon çağırıldıktan sonra ES register'ının bozulmuş olabileceğini düşünürler. Dolayısıyla fonksiyon içerisinde genel amaçlı register'ların yanı sıra ES segment register'ını da saklamak zorunda olmadan bozabiliriz.

```
/*****far00000.asm*****/
_func   proc   far
   push   bp
   mov bp,sp
   les bx,dword ptr [bp+6]
   mov word ptr es:[bx],20
   pop bp
   ret
```

```
_func   endp

_main   proc    far
    push    bp
    mov bp,sp
    sub sp,2
    mov word ptr [bp-2],10
    push    ss
    lea ax,word ptr [bp-2]
    push    ax
    call    _func
    add sp, 4
    mov sp,bp
    pop bp
    ret
_main   endp
```

/*****far00000.asm*****/

Win32/UNIX Flat Model Sistemi

Win32/UNIX sistemlerinde DOS'ta olduğu gibi bir bellek modeli kavramı yoktur. bu sistemler korumalı modda çalışırlar. Flat model sisteminde segment register2lara gereksinim kalmamıştır. Tüm segment register'lar 0'ı gösterir. Belleğin her tarafına yalnızca 4 byte'lık offset bilgisiyle erişilir. Dolayısıyla flat modelde:

1. Bütün call işlemleri near call biçimindedir. Ancak bilindiği gibi stack işlemleri 4 byte üzerinden yürütülür. Dolayısıyla IP değil EIP register'ı 4 byte olarak stack'e atılacaktır.
2. Bütün göstericiler ve fonksiyon göstericileri yakın göstericilerdir. Yani yalnızca 4 byte'lık offset bilgisi içerirler(32 bit korumalı modda segment bilgisi de kullanılıyor olsaydı göstericiler 6 byte uzunluğunda olurdu).

Ayrıntılı Segment Tanımları

Biz şimdiye kadar basitleştirilmiş segment kavramını kullandık. Ancak yaptığımız işlem karmaşık bir tanımlama biçiminin kolaylaştırılmış biçimiydi. Segment bir programda 64Kb'tan küçük ya da eşit olan ardışık kod ya da data alanıdır. Aslında bir program segment'lerden oluşur. Segment'ler 64Kb olmak zorunda değildir. Daha küçük olabilir. Bir programda birden fazla segment olabilir. Her bir segment ayrı bir yer sayacına(location counter) sahiptir. Yani bir segment'in içerisindeki bütün data ve kod sembolleri 0 numaralı yer sayacından itibaren offset belirtmektedir. Tipik olarak bir segment'in başlangıç paragraf adresi bir segment register'a atanır. Bu durumda segment içerisindeki semboller segment'in başından itibaren bir offset belirtirler.

Segment Tanımlama İşleminin Genel Biçimi

<segment ismi> segment [hizalama biçimi] [birleştirme biçimi] [sınıf biçimi]
....
....
....
<segment ismi> ends

Örneğin:

```
_MYSEG segment
```

.....
.....
_MYSEG ends

Basitleştirilmiş segment tanımlamalarıyla kod yazıldığında derleyici aslında .CODE, .DATA, .STACK denildiğinde ayrı segment'ler açmaktadır.

Segment isimleri ve segment'in bütün özellikleri obj modülü içerisine yazılmaktadır.

Segment ismi isimlendirme kuralına uygun herhangi bir isim olabilir. Microsoft ve Borland derleyicileri standart olarak aşağıdaki isimleri tercih edilir:

_TEXT
_DATA
_BSS
_CONST

Segment'in hizalama biçimi için byte, word, dword, para ve page anahtar sözcükleri kullanılabilir. Hizalama biçimi belirtilmezse default olarak para olduğu kabul edilir. Hizalama biçimi segment'in bir önceki segment bittikten sonra kaçın katlarından başlayacağını belirlemekte kullanılır. Örneğin para 16'nın katları anlamına gelir. Bir önceki segment örneğin 1FC34 adresinde bitmiş olsun. Sonraki segment para hizalama biçimine sahipse araya 12 byte boşluk bırakılarak 1FC40 adresinden itibaren yüklenecektir. Peki bir segment'in hizalanması kimin tarafından gerçekleştirilir?

DOS'un yükleyicisi exe programı blok olarak paragraf başından itibaren belleğe yükler. Bu nedenden dolayı segment hizalama görevi yükleyicinin görevi değil, linker'ın görevidir.

Linker exe kodu segment hizalamasını dikkate alarak iki segment arasında gerekirse boşluk bırakarak oluşturur. Böylece program yüklendiğinde segment'in hizalanması mümkün hale gelmiş olur. Sonuç olarak hizalama biçimi derleyici tarafından obj modüle yazılır, linker tarafından değerlendirilerek exe kod oluşturulur.

Segment'in birleştirme biçimi için *public*, *stack*, *common*, *memory* ya da *at* biçiminde olabilir. birleştirme biçimi belirtilmezse segment birleştirme işlemine sokulmaz. Yani buradaki anahtar sözcüklerden hiçbiri default değildir.

Public Birleştirme Biçimi kod ve data segment'leri için kullanılan bir birleştirme biçimidir. public birleştirme biçimi linker'ın aynı isimli segment'leri birleştirmesi için kullanılır. Örneğin:

	a.asm	b.asm
(1)	*myseg segment para public* .... *myseg ends*	*myseg segment para public* .... *myseg ends* **(4)**
	yourseg segment para public .... **(2)**	

(3)

<table><tr><td>yourseg ends

myseg segment para public
....
....
myseg ends</td><td></td></tr></table>

Burada linker 1, 3 ve 4 numaralı segment’leri birleştirerek exe dosya içerisinde bir araya getirir. Yani exe içerisinde bu segment’ler kesinlikle ardışık bulunacaktır. Bu segment’lerin birleştirilmesinde birbirlerine göre olan durumu modüllerin link ediliş sırasına bağlı olarak değişir. public ile aynı isimli segment’ler birleştirilebilir. Aynı isimli public birleştirme biçimine sahip olan segment’lerin hizalama biçimleri aynı olmalıdır. Çünkü hizalama biçimi segment belleğe yüklendiğinde hangi katlardan başlayacağını belirlemekte kullanılır. Başkası tarafından yazılmış olan bir segment ile birleştirilecek bir kodumuz varsa biz kodumuzu o kodla aynı isimli segment içerisine yazmalıyız ve birleştirme biçimi olarak da public kullanmalıyız. Linker aynı isimli public birleştirme biçimine sahip segment’lerin toplam uzunluğunun 64Kb’ı geçip geçmediğini kontrol eder. Toplam uzunluk 64Kb’ı geçiyorsa işlem link aşamasında error ile sonuçlanır. Örneğin C derleyicileri small modelde çalışırken bütün fonksiyonları _TEXT isimli ve public birleştirme biçimine sahip bir segment’in içerisine yazarlar. Bu durumda biz programı ne kadar çok modülden oluşacak biçimde yazarsak yazalım, yine toplam kod 64Kb’ı geçemeyecektir. Çünkü bütün modüllerde kodlar hep _TEXT isimli segment içerisine yerleştirilir. Linker bunları birleştirirken problemle karşılaşır.

Birleştirme biçimi **stack** anahtar sözcüğü de olabilir. stack birleştirme biçimine sahip olan segment linker tarafından otomatik olarak stack segment olarak yorumlanır. Yani linker exe dosyayı oluştururken SS register’ının başlangıç konumunu stack birleştirme biçimine sahip olan segment’in paragraf adresi olarak belirler. SP register’ını da bu segment’in sonunda olacak biçimde konumlandırır. Stack birleştirme biçimi aynı public birleştirme biçimi gibi işlem görür. Yani aynı isimli stack birleştirme biçimine sahip olan segment’ler tek bir segment olarak birleştirilir. Stack uzunluğu stack birleştirme biçimine ilişkin segment içerisinde tanımlanan toplam data uzunluğu kadardır. Örneğin:

```
_STACK segment para stack
        db      100     dup(0)
        dw      5       dup(0)
_STACK ends
```

Program çalıştığında SS bu segment2in başlangıcını gösterecektir. SP 110(decimal) değerinde bulunacaktır. Bu durumda programın stack uzunluğu da 110 byte uzunluğunda olacaktır. Bir projede normal olarak stack birleştirme biçimine ilişkin aynı isimli tek bir segment bulunmalıdır(Aynı isimli birden fazla stack olabilir. Ancak bunlar zaten linker tarafından birleştirilecektir). Stack birleştirme biçimine sahip birden fazla farklı isimli segment olduğunda linker stack olarak gördüğü son stack birleştirme biçimine sahip olan segment’i alır. Böyle bir durum normal değildir. Stack’e ilişkin segment’in uzunluğunun 16 bit sistemde çift, 32 bit sistemde 4’ün katı olması tercih edilmelidir. 16 bit sistemde SP register’ının tek adreste bırakılması hizalama kavramı yüzünden etkin bir kavram değildir.

Segment **common** birleştirme biçimine sahip olabilir. Ancak bu birleştirme biçimi çok seyrek kullanılır. Common birleştirme biçimiyle linker aynı isimli segment’leri aynı adresten başlayacak biçimde çakıştırır. Örneğin:

```
Seg1 segment para common
        X1      db      ?
Seg1 ends

Seg1 segment para common
        X2      db      ?
Seg1 ends
```

Burada aslında X1 ile X2 program yüklendiğinde aynı yerdir.

Memory birleştirme biçimi tanımlanmış olsa da tamamen public gibi bir anlama gelmektedir.

At birleştirme biçiminden sonra bir bellek paragraf adresi verilir. Böylece linker segment'i o paragraftan itibaren yerleştirir(Aslında linker at birleştirme biçimine sahip segment'leri belleğe yerleştirmez. Programcı yalnızca orada tanımlanan sembolleri kullanarak bu işlemden fayda sağlar). Örneğin:

```
Myseg segment at 0B800h
        X       db      ?
Myseg ends
```

Segment'leri Exe Kod İçerisindeki Dizilim Sırası

Bilindiği gibi exe dosya bir başlık kısmı içermektedir. Bu başlık kısmı yükleyici tarafından atılır. Exe kodunun içerisinde artık makine kodlarından ve statik data'lardan başka hiçbir şey yoktur. Tlink programı ile birden fazla obj modül birleştirilerek tek bir exe yapılabilir. Örneğin a.obj ve b.obj aşağıdaki gibi birlikte link edilebilir:

Tlink a.obj b.obj

Segment'lerin exe dosya içerisindeki yerleşim biçimi 3 faktöre bağlıdır:

1. Derleyicinin obj modüle yazma sırasına
2. Modüllerin linker tarafından ele alınma sırasına
3. Segment'in tanımlanmasında kullanılan sınıf ismine(class type)

Sembolik makine dili derleyicisi normal olarak segment'leri gördüğü sırada obj modül içerisine yazar. Ancak program içerisine yerleştirilen .ALPHA direktifiyle segment'ler yazılış sırasına göre değil, isimsel sıraya göre yazılır. Örneğin a.asm dosyasında segment'ler şöyle belirtilmiş olsun:

```
Y segment
        ....
Y ends

X segment
        ....
X ends
```

Normal olarak segment'ler Y, X sırasıyla obj modüle yazılacaktır. Ancak .ALPHA direktifini kullanırsak X, Y sırasında obj modüle yazılacaklardır. Derleyici aynı isimli public ya da stack

birleştirme biçimine ilişkin birden fazla segment varsa daha derleme aşamasında onları birleştirerek tek bir segment olarak obj modüle yazar. Örneğin:

```
Y segment public
        ....
Y ends

X segment
        ....
X ends

Y segment public
        ....
Y ends
```

Burada her ne kadar 3 segment tanımlanmış olmasına karşın derleyici obj modüle iki segment yazacaktır.

Birden fazla modül birlikte link edildiğinde linker public bir stack birleştirme biçimine sahip segment'leri yine birleştirir. Bunları modüllerin link edilme sırasına göre sınıf isimlerine bakarak obj modüller içerisindeki sıralarına da bakarak exe kodu içerisine yerleştirir. Sınıf isminin sıralamayı nasıl etkilediği sonraki konuda ele alınacaktır. Burada sınıf ismi kullanılmadan segment'lerin exe içerisindeki nihai yerleştirilmelerine ilişkin örnekler vereceğiz:

a.asm	b.asm	c.asm
X1 segment public X1 ends Y1 segment public Y1 ends	Y1 segment public Y1 ends X1 segment public X1 ends Z1 segment public Z1 ends	Z1 segment public Z1 ends Y2 segment public Y2 ends

Burada .ALPHA direktifi kullanılmadığına göre segment'ler obj modüle yerleşim sırasına göre yazılırlar. Link işleminin

tlink a b c

biçiminde yapıldığını düşünelim. Linker bütün modüllerdeki public birleştirme biçimine ilişkin olan aynı isimli segment'leri birleştirir. Segment'lerin nihai yerleşimlerini modüllerin link edilme sırasına bakarak ilk modüldeki segment yukarıda olacak biçimde ayarlar. Sıra: X1, Y1, Z1, Y2 şeklinde olacaktır. Link işlemini tam ters sırada yapmış olsaydık(tlink c b a) exe içerisindeki dizilim Z1, Y2, Y1, X1 şeklinde olacaktı.

Segment Tanımlamasında Kullanılan Sınıf İsminin Segment Sıralamasına Etkisi

Segment tanımlamasında sınıf ismi tek tırnak içerisinde yazılır. Herhangi bir isim olabilir. ismin büyük/küçük harf duyarlılığı yoktur. Örnek:

```
Myseg segment para public ‘CODE’
        ....
Myseg ends
```

Sembolik makine dili derleyicisi ve linker aynı sınıf ismine sahip segment'leri alt alta getirir. Yani birleştirilmeyen herhangi iki segment aynı sınıf ismi verilerek alt alta getirilebilir. Nihai sıralamada sınıf ismi de etkili olmaktadır. Örneğin:

a.asm	b.asm
X2 segment public ‘CODE’ X2 ends Y1 segment public ‘DATA Y1 ends Z1 segment public ‘CODE Z1 ends	Y1 segment public ‘DATA’ Y1 ends Z2 segment public ‘DATA’ Z2 ends Z3 segment public ‘CODE’ Z3 ends

Derleyici a.asm programı derlediğinde obj modüle X2, Z1, Y1 sırasında segment'leri yerleştirecektir. Benzer biçimde b.asm programı derlendiğinde obj modüle Y1, Z2, Z3 sırasında yerleştirir. Bu iki modülün link işleminden sonra(tlink a b) X2, Z1, Z3, Y1, Z2 sırasında exe kodun içerisine yerleştirilecektir. Bu iki modüle ters sırada link edilirlerse(tlink b a) segment'lerin dizilimi Y1, Z2, Z3, X2, Z1 şeklinde olacaktır. Bir segment'in herhangi bir data içermesi zorunlu değildir. Yani içi boş bir segment tanımlaması yapılabilir. Böylece programcılar link edecekleri ilk modülde boş segment tanımlamaları yaparak nihai sıra üzerinde etkili olabilirler. Örneğin kendi yazdığımız a.asm modülü ile başkasını yazdığı b.obj dosyasını birlikte link edecek olalım. Biz b modülünde x1, y1 ve z1 isimli segment'lerin tanımlanmış olduğunu bilelim. Ancak sırası hakkında bir bilgimiz olmasın. Kendi modülümüzde boş segment tanımlamaları yaparak nihai sıra üzerinde etkili olabiliriz. Örneğin:

```
X1 segment para public ‘CODE’
X1 ends

X2 segment para public ‘DATA’
X2 ends

X3 segment para public ‘DATA
X3 ends
```

Böyle bir tanımlamayla kendi modülümüzü link sırasında öne alırsak segment sırasının belirlenmesini sağlayabiliriz.

C Derleyicileri ve Segment'ler

C derleyicileri de Intel sisteminde program kodunu ve data'larını yine çeşitli isimler altında segment'ler içerisine yazarlar. Çünkü obj modül standarttır. Ve bu modüle göre bilgiler segment'ler içerisinde olmak zorundadır. Microsoft ve Borland derleyicileri çeşitli segment isimlerini birleştirme biçimlerini ve sınıf isimlerini ortak bir belirleme ile önceden tespit edilmiş biçimde oluşturmaktadır. Yalnızca Microsoft derleyicileri _CONST isimli ayrı bir segment daha kullanmaktadır. Bu segment Borland tarafından kullanılmaz(obj formatı

yani OMF formatı bazı UNIX sistemleri tarafından da aynı biçimde kullanılmaktadır). C derleyicilerinin çeşitli bellek modellerine ilişkin kullandıkları segment isimleri şunlardır:

Bütün bellek modellerinde _TEXT ile başlayan segment isimleri fonksiyonların makine kodları için, _DATA ile başlayan segment isimleri ilk değer verilmiş statik ömürlü nesneleri saklamak için, _BSS ile başlayan segment isimleri ilk değer verilmemiş statik nesneleri saklamak için kullanılır.

Tiny model	Tiny modeldeki segment tanımlama biçimleri sırasıyla şöyledir: - _TEXT segment byte public 'CODE' - _DATA segment word public 'DATA' - _BSS segment word public 'DATA'
Small model	- _TEXT segment byte public 'CODE' - _DATA segment word public 'DATA' - _BSS segment word public 'DATA' - _STACK segment word stack 'STACK'
Medium model	- filename_TEXT segment byte public 'CODE' - _DATA segment word public 'DATA' - _BSS segment word public 'DATA' - _STACK segment word stack 'STACK'
Compact model	- _TEXT segment byte public 'CODE' - _DATA segment word public 'DATA' - _BSS segment word public 'DATA' - _STACK segment word stack 'STACK'
Large model	- filename_TEXT segment byte public 'CODE' - _DATA segment word public 'DATA' - _BSS segment word public 'DATA' - _STACK segment word stack 'STACK'
Huge model	- filename_TEXT segment byte public 'CODE' - filename_DATA segment word public 'FAR_DATA' - filename_BSS segment word public 'FAR_DATA' - _STACK segment word stack 'STACK'

Segment Kavramının Önemi

Intel sisteminde ister fonksiyon kodu olsun, ister statik data olsun her türlü bilgi bir segment içerisinde bulunmak zorundadır. Obj modülü içerisinde bilgiler yine segment içerisinde bulunurlar. C'de segment'lerle hiç uğraşmasak bile derleyici yine bilgileri obj modüle çeşitli isimlerde segment'ler oluşturarak yazar. Obj modül programlama dili bağımsızdır. Programlama dili ne olursa olsun derleyiciler eninde sonunda bilgileri segment içerisinde obj modüle yazarlar. Bir segment 64Kb'ı geçemez. Segment içerisindeki data 64Kb'ı geçerse böyle bir obj modül oluşturulamayacağı için derleyici error verecektir. Örneğin huge model dışındaki tüm modellerde program ne kadar çok modülden oluşursa oluşsun toplam statik data 64Kb'ı geçemez. Ancak huge modelde modül başına 64Kb'ı geçemez, ancak toplamda 64Kb'ı geçebilir.

Her segment sıfırdan başlayan ayrı bir yer sayacına sahiptir. Yani her segment'in ilk byte'ının offset'i sıfırdır. Segment'lerin başlangıç adresleri(paragraf adresleri) program içerisinden segment ismi geçirilerek kullanılabilir. Örneğin:

```
mov ax, _DATA
mov ds, ax
```

Bir segment'in ismi o segment'in programın çalışma zamanı sırasındaki segment adresini belirten iki byte uzunluğunda sabit bir sayıdır. Örneğin:

```
mov ax, _DATA
```

işleminde gerçekte ax register'ına sabit bir sayı atanmaktadır. O sabit sayı da _DATA segment'inin bellekteki paragraf adresidir. Bu durumda bir data segment'ten bir data sembolünü kullanırken ds register'ının o segment'in başlangıç adresini göstermesi gerekir. Programcı bunu kod içerisinde yapmak zorundadır:

```
mov ax, _DATA
mov ds, ax
```

Ayrıntılı segment tanımlamalarıyla oluşturulmuş örnek iskelet program şöyle olabilir:

```
/*****dtlseg00.asm*****/
_TEXT segment para public 'CODE'
main proc near
        mov     ax, _DATA
        mov     ds, ax
        mov     ah, 9
        mov     dx, offset message
        int     21h
        mov     ax, 4c00h
        int     21h
main endp
_TEXT ends

_DATA segment para public 'DATA'
message db      "Merhaba$"
_DATA ends

_STACK segment para stack 'STACK'
db      100     dup     (?)
_STACK ends

end main
/*****dtlseg00.asm*****/
```

Program yüklendiğinde cs ve ip yükleyici tarafından programı bitiren end direktifinin yanında bulunan kod sembolünün segment ve offset adresleriyle otomatik olarak yüklenmektedir. Yine ss ve sp register'ları otomatik olarak stack birleştirme biçimine sahip olan segment'in segment adresi ve uzunluğu ile yüklenmektedir(Böyle bir segment yoksa ss PSP'yi, sp ise FFFE değeriyle yüklenir. Ancak linker bu tür durumlarda "No stack" biçiminde uyarı vermektedir). Ds ve es register'ları yükleme sırasında psp'yi gösterecek biçimde değer alır. Yani ds register'ının ilgili segment adresiyle yüklenmesi programcının görevidir. Ayrıca programın kodunu içeren segment'lerin bugün kullandığımız linker'larda 'CODE' biçiminde sınıf ismine sahip olması zorunludur. Normal olarak bir kod segment'in içerisinde data sembolü, bir data segment'inin içerisinde de makine kodları yazılabilir. Makine komutlarının bulunduğu segment'in 'CODE' sınıf ismine sahip olma zorunluluğu linker'ın bazı sistemlerde

program kodunun bulunduğu segment'leri teşhis etmesi için düşünülmüştür. Ayrıntılı segment tanımlamalarıyla program yazarken .model bildirimi programın başında kullanılmaz.

Ayrıntılı Segment Tanımlamaları ve Bellek Modeli

Bellek modeli kavramı yüksek seviyeli dillerin derleyicileri için düşünülmüş bir kavramdır. Çünkü bellek modeli kavramı kullanılacak data ve kod segment'lerinin sayısını onların isimlerini vs belirlemekte kullanılır. Halbuki ayrıntılı segment tanımlamalarıyla program yazarken bu belirlemeleri biz istediğimiz gibi yapabiliriz. Yani bellek modeli kavramıyla bir ilgimiz yoktur.

Birden Fazla Data Segment'i ile Çalışmak

Tek bir data segment tanımlayarak çalışırsak programımızın statik dataları 64Kb'ı geçemez. Statik dataların 64Kb'tan fazla olabilmesi için birden fazla data segment tanımlamak gerekir. Birden fazla data segment kullanılırken ds nereyi gösterecektir? İşte ds kullanılan data sembolü hangi data segment'e ilişkinse orayı göstermek zorundadır. Bu durumda bir data sembolü kullanılırken ds'nin doğru segment'i gösterip göstermediğine dikkat edilmelidir. Örneğin message1 isimli sembol _DATA1 segment'inde, message2 isimli sembol _DATA2 segment'inde tanımlanmış olsun. Bu sembollere aşağıdaki gibi erişebiliriz:

```
mov ax, _DATA1
mov ds, ax
mov ax, message1
mov ax, _DATA2
mov ds, ax
mov ax, message2
```

Birden Fazla Kod Segment'i ile Çalışmak

Programın kodu 64Kb'ı geçecekse birden fazla kod segment'i tanımlamak gerekebilir. Bu durumda bazı fonksiyonlar bir kod segmentte, bazıları diğer kod segment'te olabilir. başka bir kod segment'teki fonksiyonu çağırabilmek için far call yapmak gerekir. Aynı segment içerisindeki bir fonksiyonu near call ile çağırabiliriz. Ancak bir fonksiyonun hangi segment'ten çağırılacağını bilemiyorsak ya da bir fonksiyon hem kendi segment'inden hem de dışarıdaki bir segment'ten çağırılacaksa fonksiyonu retf ile bitirmek gerekir. Bu durumda aynı segment'te bile olsak çağırma işlemini far call ile yapmak zorunda kalırız. Sonuç olarak birden fazla kod segment'i ile çalışıldığında en pratik yöntem bütün fonksiyonları retf ile bitirmek fonksiyon hangi segment'ten çağırılırsa çağırılsın far call işlemi yapmaktır. Derleyicilerin bazıları birden fazla kod segment'i ile çalışırken aynı segment içerisinde çağırmaları

```
push cs
call near func
```

biçiminde yaparlar. Bu işlem far call işleminden 1 byte daha az makine koduna sahiptir. Medium, large, huge modellerde farklı modüllerde birden fazla kod segment bulunabilir.

Win32/UNIX Flat Modellerde Segment İşlemleri

Win32 flat modelde bütün segment değerleri 0 adresini göstermektedir. Belleğin her yerine yalnızca offset bilgisiyle erişilebilir. Bu durumda segment kavramının bir önemi kalmamaktadır. Örneğin bir fonksiyon başka bir kod segment'te olsa bile biz o fonksiyonu near call işlemiyle çağırabiliriz(32 bit modda near call 4 byte'lık bir hedef adres almaktadır). Flat modelde birden fazla data segment kullanılmasının da önemli bir anlamı yoktur. Çünkü sembol hangi data segment'te olursa olsun ona 4 byte'lık bir offset ile erişebiliriz. Yine de bu sistemlerde segment tanımlamaları kullanılmaktadır. Bu tanımlamalar ile belirli program bölümlerinin exe dosya içerisinde sıralaması belirlenebilir. Win32'de segment'ler section olarak değerlendirilir. Bir section n sayfadan oluşur(Bir sayfa 4 Kb'tır). Her section bir özellikle belirtilebilir. Örneğin bir section read-only yapılabilir. Böyle bir section içerisindeki bir bölgeye yazma yapılamaz. Win32 ve UNIX sistemlerinde bir segment 64Kb'ı geçebilir. Bir segment'in 64Kb'ı geçip geçemeyeceği use16 ya da use32 bildirimleriyle belirlenir. Örneğin:

```
_TEXT segment para public use32 'CODE'
        ....
_TEXT ends
```

Şimdi bu segment 64Kb'ı geçebilir. Default durum use16'dır. Ancak programın başına .386, .386p, .486 veya .486p bildirimlerinden biri yazılırsa default durum use32 olur.

Grup Kavramı

64Kb içerisinde olan bir ya da birden fazla segment'in oluşturduğu topluluğa grup denir. Grup bildirimi şöyle yapılır:

grup_ismi group segment_ismi, segment_ismi2, ..

Örneğin:

```
DGROUP      group   _DATA, _BSS
```

Grup tanımlamasının segment sıralaması üzerinde hiçbir etkisi yoktur. Segment sıralaması segment'lerin yazılış sırasına ve sınıf isimlerine bağlıdır. Bir grubu oluşturan segment'lerin nihai sıralamada 64Kb alan içerisinde bulunması gerekir. bu segment'lerin aralarında başka segment'ler olabilir. Grup tanımlamasında belirtilen segment isimlerinin sırasının hiçbir önemi yoktur. Bir grubu oluşturan segment'lerin nihai durumda en başta olanıyla en sonda olanı arasında 64Kb'tan daha fazla açıklık olmaması gerekir. Örneğin grup bildirimi şöyle yapılmış olsun:

```
DGROUP      group   _DATA, _BSS
```

Nihai durumda segment dizilimleri aşağıdaki gibi olsun:

_BSS
_PROCESS
_NUMERIC
_DATA

Burada _BSS ve _DATA arasındaki uzaklık 64Kb'ı geçmemek zorundadır. Eğer geçerse link aşamasında "DGROUP exceeds 64Kb" biçiminde bir hata oluşur.

Grup tanımlaması kümülatif bir tanımlamadır. Örneğin bir projede iki modül olsun. Birinci modülde

MYGROUP group _DATA

tanımlaması yapılmış olsun. İkinci modülde ise

MYGROUP group _BSS

tanımlaması yapılmış olsun. Yapılan işlem sonucunda MYGROUP isimli grupta _DATA ve _BSS isimli iki segment vardır. Bir grubu oluşturan segment'lerin nihai durumda 64Kb'ı geçmemesi manual olarak sağlanmak zorundadır. Bunu sağlamak için bu segment'lerin sınıf isimlerini aynı yapmak bir katkı sağlayabilir.

Grup Kullanımı

Bir grubun ismi doğrudan bir segment ismi gibi kullanılabilir. Örneğin:

```
mov     ax, DGROUP
mov     ds, ax
```

Grup ismi grubun başlangıç paragraf adresini belirtir. Bu adres nihai durumda grup içerisinde ilk sırada bulunan segment'in başlangıç adresidir. Grup küçük fakat çok sayıda segment'lerle çalışırken segment register'ların tekrar tekrar kullanılmasını engellemek için tercih edilir. Örneğin:

```
_DATA segment para public 'DATA'
number1 dw      0
number2 dw      1
_DATA ends

_BSS segment para public 'DATA'
number3 dw      ?
number4 dw      ?
_BSS ends
```

Buradaki number1 ve number3 farklı segment'lerde tanımlanmış sembollerdir. Normal olarak bunların kullanılabilmesi için ds register'ının değiştirilmesi gerekir:

```
mov     ax, _DATA
mov     ds, ax
mov     ax, number1
mov     ax, _BSS
mov     ds, ax
mov     ax, number3
```

Bu segment'ler çok küçük olduklarına göre bir grup oluşturabilir. Bu durumda ds register'ını bu grubun başlangıç adresiyle yüklersek bütün sembollere yalnızca offset bilgisiyle erişebiliriz.

```
DGROUP      group   _DATA, _BSS

mov     ax, DGROUP
mov     ds, ax
mov     ax, number1
```

Burada bir problem vardır. Derleyicinin number1 ve number3 sembolleri için kendi segment'lerinin başlangıcından itibaren değil, grubun başlangıcından itibaren adres üretmesi gerekir. Bu işlem assume direktifiyle yapılmaktadır.

Assume Bildirimi

Assume sembolik makine dili derleyicisinin bir data ya da kod sembolünü gördüğünde hangi orijine göre offset üreteceğini belirlemekte kullanılır. Assume işleminin default'u yoktur. Yani assume bildirimini yapmak gerekir(Masm 6.1 derleyicisinde assume yapılmazsa default olarak semboller için kendi segment'lerinin başı orijin alınarak offset üretilir). Basitleştirilmiş segment tanımlamalarıyla program yazarken assume kullanılmaz. Assume bildirimi bellek modeline bakılarak default değerlerle derleyici belirlenir. Assume bildirimi şöyle yapılır:

assume   segreg:orijin, segreg:orijin, segreg:orijin

Örneğin:

```
assume          cs:_TEXT, ds:DGROUP, ss:_STACK
```

Orijin noktası olarak ya bir segment ismi ya da bir grup ismi yazılmak zorundadır. Assume işleminden sonra bir sembol kullanıldığında o sembolün default olarak ilişkin olduğu segment register tespit edilir. O segment register ile belirlenen orijin dikkate alınır. Örneğin:

```
mov     ax, number1
```

number1 sembolünün default segment register'ı ds'dir. Ds ise DGROUP orijinine sahiptir. Ya da örneğin

```
mov     ax, ss:number1
```

Burada number1 sembolünün segment register'ı ss'tir. Ss için orijin _STACK biçimindedir. Bir sembol kullanıldığında derleyici o sembolün ilişkili olduğu segment register hangi segment ya da grup ile assume edilmise o orijinden başlayarak offset hesaplar.Eğer kullanalan sembol assume edilmiş gruo içinde değilse bu durum error oluşturur. Assume işleminin segment register'ının yüklenmesiyle bir ilgisi yoktur. Yalnızca offset belirlenmesiyle ilgisi vardır. Örneğin bir modül içinde grup oluşturmadan birden fazla data segment kullanıldığında hem diğer segment'teki sembolü kullnamadan DS'nin yükenmesi hem de o offset için yeniden assume işleminin yapılması gerekir. Ancak offset operatörü default olarak her zaman sembolün bulunduğu segment'in başından başlayarak bir offset üretmektedir. Örneğin; X1 sembolü _DATA egment'i içinde tanımlanmış olsun. DS ise DGROUP ile assume edilmiş olsun.

```
mov     ax,offset X1
```

DS, DGROUP ile assume edidiği halde derleyici offet için orijin noktasını _DATA kabul eder. Ancak bu işlem aşağıdaki gibi değiştirilebilir:

mov   ax,offset DGROUP:X1

OMF Formatı

16 bit DOS sisteminde kullnaılan .obj dosya formatına **OMF Formatı** denir. OMF Formatı 1978'de Intel tarafından tanımlanmıştır ve Microsoft tarafındn geliştirilmiştir. OMF Formatının 32 bitlik bir versiyonu da vardır. Intel işlemcilerinini kullanıldığı UNIX sistemlerinin çoğunda 32 bit OMF formatı kullanılmaktadır. WIN32 sistemlerinde çalışan Borland derleyicilerinde hala .obj modül formatı olarak OMF formatı kullaınmaktadır. Ancak Microsoft WIN 32 sistemleriyle birlikte yeni bir .obj dosya formatı kullanmaya başlamıştır. Bu formata COFF (Common Object File Format) denir.

OMF Formatını Genel Yapısı

.obj dosya formatı farklı modüllerin birleştirilmesi konusunda önemli bilgiler içeren bir yapıya sahiptir. OMF frmatı sembolik makine dili programının binary bir dosya biçimine dönüştürülmüş hali gibidir. Örneğin içind mkine kodları sement tanımlamaları, sembol tanımlamaları, extern ve public gibi değişken tanımlamaları vardır. OMF formatı Borland firmasının TDUMP.EXE, Microsoft firmasının DUMPBIN.EXE progamlarıyla incelenebilir. Bu programlar dosyanın uzantısına bakarak .exe, .obj, .lib gibi dosyalarıda inleceleyebilir. TDUMP programı şöyle yapılabilir:

```
tdump  a.obj
tdump  a.obj | more
tdump  a.obj > x.txt
```

OMF formatı kayıtlardan oluşur. Her kayıtın ayrı bir ismi vardır. Her kayıtın frmatı birbirlerinden farklıdır.

```
  1.byte       2.byte
-----------|-----------|----------------
rec type   | rec len   | Değişebilir.
-----------|-----------|----------------
```

Her kayıtın ilk iki elemanı sabittir.Bir kaydın ilk byte'ı o kaydın türünü anlatan bir sayı içerir.Daha sonraki 2 byte o kayıtın uzunluğudur. Her kaydın uzunluğu birbirlerinden farklı olabilir. OMF formatında kullanıolan kayıtların isimleri şunlardır:

16 VE 32 BIT OMF KAYITLARI

80	THEADR	8E	TYPDEF	96	LNAMES	9D	MFIX386
88	COMENT	90	PUBDEF	98	SEGDEF	A0	LEDATA
8A	MODEND	91	MPUB86	99	MSEG386	A1	MLED386
8B	H386END	94	LINNUM	9A	GRPDEF	A2	LIDATA
8C	EXTDEF	95	MLIN386	9C	FIXUPP	A3	MLID386

Her kayıdın kayıt uzunluğu kayıt türü ve kayıt uzunluğu dşında geri kalan bilgilerin uzunluğunu belirtir.

OMF formatının kayıtlarını display eden program:

```
/*---------------------- OMF.c----------------*/

typedef unsigned short int WORD;
typedef int BOOL;
typedef unsigned long int DWORD;
typedef struct _OBJREC {
        BYTE recType;
        char *recName;
} OBJREC;
/*    Global Variables  */
OBJREC objRec[] = {
        {0x80, "THEADR"},
        {0x88, "COMENT"},
        {0x8A, "MODEND"},
        {0x8B, "H386END"},
        {0X8C, "EXTDEF"},
        {0X8E, "TYPDEF"},
        {0X90, "PUBDEF"},
        {0X91, "MPUB86"},
        {0X94, "LINNUM"},
        {0X95, "MLIN386"},
        {0X96, "LNAMES"},
        {0X98, "SEGDEF"},
        {0X99, "MSEG386"},
        {0X9A, "GRPDEF"},
        {0X9C, "FIXUPP"},
        {0X9D, "MFIX386"},
        {0XA0, "LEDATA"},
        {0XA1, "MLED386"},
        {0XA2, "LIDATA"},
        {0XA3, "MLID386"},
        {0, NULL}
};
/* FUNCTION PROTOTYPES */
void CheckArg(int argc);
void DispRec(BYTE recType);
void CheckArg(int argc)
{
        if(argc == 1) {
                fprintf(stderr, "usage: disobj <.obj file name>\n");
                exit(1);
        }
        if(argc > 2) {
                fprintf(stderr,"Too many parameter!..\n");
                exit(1);
        }
}



void DispRec(BYTE recType)
{
        int i;
        for(i = 0; objRec[i].recName != NULL; ++i) {
                if(objRec[i].recType == recType) {
```

```
                        printf("%02x\t%s\n", recType, objRec[i].recName);
                        return;
                }
        }
        printf("%02x\t%s\n", recType, "Unknown Record");
}
void main(int argc, char *argv[])
{
        FILE *f;
        BYTE recType;
        WORD length;
        CheckArg(argc);
        if((f = fopen(argv[1], "rb")) == NULL) {
                fprintf(stderr, "Cannot open file!..", argv[1]);
                exit(1);
        }
        while(fread(&recType, 1, 1, f) > 0) {
                DispRec(recType);
                fread(&length, sizeof(WORD), 1, f);
                fseek(f, length, SEEK_CUR);
        }
        fclose(f);
}
/*-------------------------------------------------*/
```

Data ve Kod Sembollerinin obj Dosyaya Yazılması

Bir sembol kullanıldığında derleyici o sembol yerine [ ] içerisinde bir offset değeri geldiğini düşünür. Bu offset değerini de assume bildirimine göre belirler. Ancak bu offset bilgisi o modüle ilişkin göreli bir değerdir. Çünkü linker aynı isimli public birleştirme biçimine sahip segment'leri birleştirdiğinde bu offset değerlerini de yeniden düzenlemek zorunda kalır. Örneğin: iki modüle sahip bir program yazacak olalım. İki modülde de _DATA isimli public birleştirme biçimine sahip segment olsun.

M1.asm

```
_DATA segment para public 'DATA'
        db      100     dup(0)
_DATA ends
```

M2.asm

```
_DATA segment para public 'DATA'
        X1      db      10
        X2      db      20
_DATA ends
```

Burada X1 offset'i m2.asm içerisinde 0 değerine ilişkin olduğu halde

tlink m1 m2

işlemiyle link edilme sonucunda segment'lerin birleştirilmesi sonucunda 100 değerine ilişkin olacaktır. Şimdi bu semboller m2.asm içerisinde şöyle kullanılmış olsun:

```
_TEXT segment para public 'CODE'
        .....
        Mov     al, X1          | A0 0000
        Mov     al, X2          | A0 0100
```

```
        .....
_TEXT ends
```

Buradaki makine komutlarını oluşturan offset bilgileri exe koduna doğrudan yansımamalıdır. Buradaki offset bilgileri o modüle ilişkin göreli değerlerdir. Linker segment'leri birleştirirken bu makine komutundaki offset'ler düzeltilmek zorunda kalınabilir. Bu yüzden derleyici obj modülü oluştururken bütün data ve kod sembollerinin söz konusu edildiği yerleri FIXUP RECORD içerisinde not alır. Fixup record içerisinde ilgili sembolün ismi yoktur, ancak o sembolün hangi segment içerisinde olduğu bilgisi ve o sembolün program kodunun nerelerinde kullanıldığı bilgisi vardır. Bu durumda linker segment'leri birleştirirken o segment'ler içerisindeki sembollerin kullanıldığı makine komutlarını güncellemek zorundadır. Bu işlemi şöyle yapar:

1. Segment'leri birleştirirken aradaki açıklıkları hesaplar.
2. Fixup record'larını inceleyerek bu sembollerin kullanıldığı kod bölgelerini bulur. Buradaki sembol offset'lerini bu açıklık miktarlarını toplayarak nihai offset'leri elde eder.

Bu durumda fixup record sembollerin hangi segment ya da grup içerisine bulunduğunu ve bu sembollerin program kodunun nerelerinde kullanıldığını tutan bir record'dur.

```
/*****seg00001.asm*****/
_DATA segment para public 'DATA'
   X1     db  100
   X2     db  200
_DATA ends

_TEXT segment para public 'CODE'
assume  cs:_TEXT, ds:_DATA
   mov     al, X1
   mov     bl, X2
_TEXT ends

end
/*****seg00001.asm*****/
```

Tasm /l seg00001.asm

```
/*****seg00001.lst*****/
     1 0000              _DATA segment para public 'DATA'
     2 0000  64              X1       db     100
     3 0001  C8              X2       db     200
     4 0002              _DATA ends
     5
     6 0000              _TEXT segment para public 'CODE'
     7                assume  cs:_TEXT, ds:_DATA
     8 0000  A0 0000r            mov       al, X1
     9 0003  8A 1E 0001r         mov       bl, X2
    10 0007              _TEXT ends
    11
    12               end
/*****seg00001.lst*****/
```

Tdump seg00001.obj > seg00001.tdm

/*****seg00001.tdm*****/
000000 THEADR seg00001.asm *000011 COMENT Purge: Yes, List: Yes, Class: 0 (000h)* *Translator: Turbo Assembler Version 3.1* *000034 COMENT Purge: Yes, List: No , Class: 233 (0E9h)* *Dependency File: seg00001.asm 02/22/101 08:13 pm* *00004B COMENT Purge: Yes, List: No , Class: 233 (0E9h)* *End of Dependency List* *000051 LNAMES* *Name 1: ''* *000056 COMENT Purge: Yes, List: No , Class: 161 (0A1h)* *Object Module Extensions Present.* *00005C LNAMES* *Name 2: '_DATA'* *Name 3: 'DATA'* *00006B SEGDEF 1 : _DATA PARA PUBLIC Class 'DATA' Length: 0002* *000075 LNAMES* *Name 4: '_TEXT'* *Name 5: 'CODE'* *000084 SEGDEF 2 : _TEXT PARA PUBLIC Class 'CODE' Length: 0007* *00008E COMENT Purge: Yes, List: No , Class: 162 (0A2h)* *Linker - Pass Two Marker.* *000095 LEDATA Segment: _DATA Offset: 0000 Length: 0002* *0000: 64 C8 d.* *00009E LEDATA Segment: _TEXT Offset: 0000 Length: 0007* *0000: A0 00 00 8A 1E 01 00 ....* *0000AC FIXUPP* *FixUp: 001 Mode: Seg Loc: Offset16 Frame: TARGET Target: SI[1]* *FixUp: 005 Mode: Seg Loc: Offset16 Frame: TARGET Target: SI[1]* *0000B8 MODEND*
/*****seg00001.tdm*****/

Bütün semboller lst dosyasında r sembolüyle belirtilirler.

Makine Kodlarının ve Statik Dataların obj Modüle Yazılması

Segment'ler içerisinde datalar ve makine kodları derleyici tarafından obj modüle linker tarafından da exe dosya içerisine yazılırlar. Derleyici ham data ve kod bilgilerini LEDATA ve LIDATA record'larına yazarlar. Programın makine kodları ve statik dataları LEDATA record'una yazılır. Ancak obj dosyayı şişirmemek için elemanları aynı değeri alan diziler kullanıldığında derleyiciler bu dizi elemanlarını tek tek LEDATA bölümüne yazmazlar. Bu bilgileri "100 tane 0" gibi bir anlatımla LIDATA record'una yazarlar. Örneğin:

X1 db 10, 20, 30, 40, 100 dup (0)

Burada 10, 20, 30, 40 sayıları LEDATA record'una yerleştirilir. "100 tane 0" bilgisi ise LIDATA record'una yerleştirilir. Bu durumda örneğin C'de aşağıdaki gibi bir global bir dizi açmış olalım:

int a[10000];

Şimdi obj modül en azından 20Kb kadar olmayacaktır. Çünkü C derleyicisi bu bilgiyi obj modülün LIDATA record'una yazar. Tabii maalesef mz exe dosya formatında bu bilgi 20000

byte 0 biçiminde gözükecektir. Çünkü mz formatı yükleyici tarafından blok olarak yüklenmektedir.

Basitleştirilmiş Segment Kullanımında Segment İşlemleri

Basitleştirilmiş segment tanımlamalarında aslında yine segment tanımlaması yapılmaktadır. Çünkü omf formatı segment tanımlamalarını içermek zorundadır. Basitleştirilmiş segment tanımlamalarında önce .model direktifiyle bir bellek modeli belirlenir. Bundan sonra .CODE, .DATA, .STACK belirlemeleri yapılığında derleyici belirli segment'lerin tanımlanmış olduğunu varsayar. Ayrıca basitleştirilmiş segment tanımlamalarında DGROUP isimli bir grup da tanımlanmaktadır. Otomatik tanımlanan bu segment ve grupların özellikleri doküman halinde dağıtılmıştır.

Relocatable Adresler

Program içerisinde kullandığımız offset değerleri yani DATA ve CODE sembolleri derleyici tarafından göreli bir biçimde obj dosyasına linker tarafından ise nihai olarak exe dosyaya yazılır. Offset bilgisi programın yüklenme adresiyle değişebilecek bir bilgi değildir. Çünkü segment belleğin neresine yüklenmiş olursa olsun segment register o segment'in başlangıç adresiyle yüklendiğinde offset'ler doğru yerleri gösterirler. Yani offset değerleri fiziksel RAM'de bir adres belirtmezler. Yer aldıkları segment'e ilişkin göreli bir adres belirtirler. Programın içerisinde kullanılan segment ve grup isimleri 2 byte uzunluğunda bir sabit biçiminde koda yansır. Bu segment isimleri ilgili segment'lerin fiziksel RAM'deki paragraf adreslerini belirtmektedir. Segment'lerin paragraf adresleri ancak programın boş olan RAM bölgesine yüklenmesiyle kesinlik kazanır. Bu durumda makine komutunun segment paragraf adresi içeren kısımları exe dosyası içerisinde nasıl olacaktır? İşte linker bu adresleri sanki program belleğin tepesinden itibaren yüklenecekmiş gibi oluşturur. Başka bir deyişle buradaki paragraf adresleri exe dosyanın başından itibaren göreli bir adres belirtirler. Linker exe dosyanın başlık kısmı içerisinde ismine relocation tablosu denilen bir tablo oluşturur. Bu tabloda bütün segment ve grup adresi içeren makine komutlarında bu adreslerin exe dosyanın neresinde olduğu bilgisi vardır. Yükleyici programı bellekte boş bulduğu alana yükler. Sonra relocation tablosuna başvurur. Segment ve grup adreslerinin kullanıldığı yerleri tespit eder. Bu adreslere programın yükleme adresini ekler. Böylece artık bütün makine komutları programın yüklenme yerine bağlı olan fiziksel RAM değerlerini almış olur. Relocatable adres kavramı pek çok işletim sistemi için söz konusu bir kavramdır. Konunun özeti şudur:

1. Çalışabilen kod içerisindeki bazı adresler görelidir. Bazıları fiziksel RAM'de adres belirtir. Fiziksel adreslere relocatable adresler denir.
2. Göreli adresler(Intel sisteminde offset adresleri) program RAM'de nereye yüklenirse yüklensin problem oluşturmazlar. Halbuki relocatable adresler programın yüklenme yerine göre değiştirilmesi gereken adreslerdir(Intel sisteminde segment ve grup adresleri).
3. Linker relocatable adreslerin kesin değerlerini bilemez. Ancak programın RAM'in tepesine yükleneceği varsayımıyla bunların göreli değerlerini bilebilir.
4. Program içerisinde relocatable adreslerin kullanıldığı bölgeler çalışabilen kod üzerinde bir tablo içerisinde belirlenir.
5. Yükleyici programı belleğin boş bölgesine yükler. Relocatable adres tablosuna bakarak gerçek değerlerini hesaplayıp yerleştirir.

Relocatable Adresler ve Relocation Tablosu

Bilindiği gibi .EXE dosya bir başlık kısmıyla başlamaktadır. Başlık kısmının sonunda relocation tablosu bulunur. Yükleyici tüm başlık kısmını atarak programı yüklemektedir. EXE dosya başlık kısmının [06 - 07] offset'lerinde bulunan WORD değeri, relocation tablosu içerisindeki elemanların sayısını belirtir. Relocation tablosunun yeri EXE başlığının [18 – 19] kısmından çekilen WORD değerde saklıdır. Relocation tablosunun başlangıç yeri .EXE dosyasının başından itibaren bir offset belirtir. Relocation tablosunda her eleman için 4 byte yer ayrılır. Bilindiği gibi .EXE dosyanın başlık kısmının uzunluğu başlık kısmının [08 - 09] offset'lerinden çekilen WORD değerle tespit edilebilmektedir. Burada başlık kısmının paragraf uzunluğu verilmiştir. Yani bu değeri 16 ile çarpmak gerekir. Her relocation elemanı 4 byte'tır. bu 4 byte'ın düşük anlamlı WORD'u offset, yüksek anlamlı WORD'u segment olarak değerlendirilir. Yani yüksek anlamlı WORD 16 ile çarpılır, düşük anlamlı WORD ile toplanır. Bu değer .EXE başlığı atılmışken ki durumda bir offset belirtmektedir. Bu offset relocatable bilginin bulunduğu yerdir.

Özetle yükleyici relocatable adresleri şöyle dönüştürür:

1. Yükleyici .EXE dosyanın başlık kısmını atar. Geri kalan kısım bellekte boş olan bölgeye yüklenir.
2. EXE dosyanın başlık kısmından relocation tablosunun yeri ve eleman sayısı tespit edilir.
3. Her relocation elemanı için şu işlemler yapılır: Relocation elemanının yüksek anlamlı WORD'u 16 ile çarpılıp, düşük anlamlı WORD'u ile toplanır. Bellekte yüklü olan EXE dosyanın bu offset'ine erişilir. Bu offset'te bulunan WORD değere programın yüklenme paragraf adresi toplanır.
4. Programın başlangıç register değerleri .EXE başlık dosyasına bakılarak verilir.
5. Kontrol CS : IP 'nin belirtildiği adrese bırakılır.

COM Dosyalar

64K'yı geçemeyen ve bir başlık kısmı olmayan, bu nedenle çok kolay yüklenebilen dosyalara COM dosyalar denir. Bir COM program yükleyici tarafından şöyle yüklenir:

1. Yükleyici COM program için DOS düzeyinde 64K'lık bir alan tahsis eder. 100H uzunluğunda PSP bölgesini oluşturur. COM programı 100H 'tan itibaren belleğe yerleştirir.
2. Bütün segment register'ları PSP paragraf adresine eşitler. IP register'ını 100H değerine çeker. Diğer tüm register'ları sıfırlar.
3. Programın akışını CS : IP 'ye bırakır.

COM program başlık kısmına sahip değildir. Bu nedenle başlık kısmı oluşumuna yol açacak işlemler program içerisinde yapılamaz. Bir COM programda şu özelliklerin bulunması zorunludur:

1. Program içerisinde relocatable bir adres kullanılamaz. Yani segment ve grup isimleri kullanılamaz. Bu durum programın segment'siz yazılacağı anlamına gelmez. Yalnızca segment isimleri kullanılamaz. Örneğin aşağıdaki gibi bir makine komutunu yazamayız:
   ```
   mov     ax, _DATA
   mov     ds, ax
   ```

Bir COM programın code, data ve stack'i aynı 64K'nın içerisinde olmak zorundadır.

2. Program içerisinde stack segment tanımlanamaz (Yani stack birleştirme biçimine sahip olan segment tanımlanamaz). Programın bir stack segment'inin olması bir başlık kısmının olmasını gerektirmektedir. Ancak programın stack'i yükleyici tarafından otomatik oluşturulmaktadır. Yükleyici SS 'i PSP 'ye SP 'yi ise FFFE değerine çeker.
3. Programın başlangıcı IP = 100H olacak biçimde organize edilmelidir. COM programın ilk byte'ı offset 100H 'ta olacak biçimde yüklenir. Bu durumda COM programın ilk byte'larında kesinlikle bir makine komutu olmak zorundadır.

ORG Komutu

Bilindiği gibi derleyici semboller için offset değerlerini yani genel olarak yer sayacını assume edilmiş segment ya da gruba göre orijinleyerek vermektedir. Örneğin

assumecs: _TEXT

ifadesiyle _TEXT segment'i için yer sayacı 0 değerinden başlar. Halbuki pek çok durumda yer sayacının özel bir değerden başlatılması ya da devam ettirilmesi istenebilir. Örneğin COM program yazarken offset'leme işlemini 100H değerinden başlaması gerekir. İşte ORG komutu yer sayacının değerini orijinlemek amacıyla kullanılır. ORG komutunun genel biçimi şöyledir:

org değer

ORG komutu segment'in herhangi bir yerine yerleştirilebilir. Bu komuttan sonra yer sayacı komutta belirtilen değeri alacaktır.

COM Programların Yazımı

COM programları tek segment'li ya da çok segment'li yazılabilir. Ancak yukarıda yazılan genel kurallara uyulmak zorundadır.

Tek Segment'li COM Programlarının Yazılması

Tek segment'li COM programlarında yalnızca code segment tanımlanır. Segment'in yer sayacı 100H değerine çekilir. Programın data'ları segment'in code kısmından sonraya yerleştirilebilir. Ancak en çok uygulanan yöntem segment'in başına bir jmp komutu koymak ve data'ları hemen jmp komutunun altına yerleştirmek biçimidir. Bu biçimdeki tipik bir COM programının kalıbı şöyledir:

```
_TEXT segment para public 'CODE'
        assume  cs:_TEXT, ds:_TEXT, ss:_TEXT, es:_TEXT
        org             100h
main proc near
        jmp             @1
        x1      db      0
        x2      db      0
        @1: .....
                .....
```

```
main endp
_TEXT ends
end main
```

Örneğin:

```
_TEXT segment para public 'CODE'
        assume  cs:_TEXT, ds:_TEXT, ss:_TEXT, es:_TEXT
        org             100h
main proc near
        jmp             @1
        msg     db      "This is a COM program", '$'
@1:     mov             ah, 9
        mov             dx, offset msg
        int             21h
        mov             ax, 4c00h
        int             21h
main endp
_TEXT ends
end main
```

Bir COM programı normal bir biçimde derlenerek .OBJ yapılır. Ancak **/t** seçeneği ile link edilmelidir. Yoksa linker programı .EXE sanır ve bir başlık kısmı oluşturur. Örneğin:

tlink /t den.OBJ

Çok Segment'li COM Programının Yazılması

Bu yöntemde birden fazla segment tanımlanabilir. Ancak bunların bir grup oluşturmaları gerekir. Programın en yukarısındaki segment'in kod içermesi gerekir. Program çok segment'li olmasına karşın program içerisinde ayrıca segment ismi kullanılmaz. Assume ifadesi gruba göre verilir. Dolayısıyla bütün offset'ler zaten gruba göre oluşturulacaktır.

```
/*****com00001.asm*****/
DGROUP group _TEXT, _DATA

_TEXT segment para public 'CODE'
    assume cs:DGROUP, ds:DGROUP, ss:DGROUP
    org     100h
entry:
    mov     ah, 9
    lea     dx, msg
    int     21h
    mov     ax, 4c00h
    int     21h
_TEXT ends

_DATA segment para public 'DATA'
msg db      "Hello$"
_DATA ends

end entry
/*****com00001.asm*****/
```

Tiny Model Programlar ve COM Dosyaları

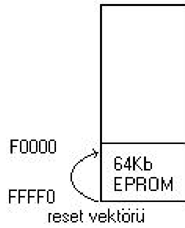
Tiny modelde bir C programı yazıldığında programın stack segment'i yoktur. bütün segment'ler bir grup içerisindedir. Programın başlangıç IP değeri de 100h'tır. Tiny model exe programlar bu genel yapısından dolayı exe2bin yardımcı programıyla com formatına dönüştürülebilmektedir.

COM Programlarının Önemi

Com programları bir başlık kısmına sahip olmadıkları için kolaylıkla belleğe yüklenerek çalıştırılabilirler. Com programları saf makine dilinden oluşmaktadır. Bu nedenle saf makine dili gereksinimi olan pek çok alanda kullanılabilir.

COM Programının Kullanılmasına Tipik Bir Örnek: Boot programının yazılması

Bilindiği gibi bilgisayar açıldığında çalışma CS:FFFF IP:0000 adresinden başlar. Intel işlemcikleri reset edildiğinde gerçek moddan çalışmaya başlar. Yani 1Mb bellekten fazlasını göremez. Bu durumda reset işlemiyle birlikte ilk çalışacak komut FFFF0 adresindeki komuttur. Burası son 64Kb içerisindeki EPROM bölümdür.



FFFF0 adresinde bir far jmp komutu bulunur. Bu far jmp ile akış EPROM'daki POST(power on self test) programına geçirilir. Burada çeşitli self test işlemleri yapıldıktan sonra sıra işletim sisteminin yüklenmesine gelir. Bunun için buradaki program A sürücünün boot sektörünü belleğin 7C00 adresine kopyalar ve CS:0000 IP:7C00 olacak şekilde programı çalıştırır. İşte böylece akış boot sektördeki programa geçirilmiş olur. artık buradaki program işletim sistemini yükleyecektir.

Amacımız boot sektöre bir program yerleştirerek makine açılır açılmaz bizim programımızı çalıştırmak olsun. Bu işlem şu adımlardan geçilerek yapılabilir:

1. Boot sektöre yerleştirilecek com programı yazılır.
2. Bu com programı boot sektöre kopyalanır.

Boot sektöre yerleştirilecek com programı yazılırken bazı noktalara dikkat etmek gerekir. Birincisi org 100h com programı yazabilmek için mutlaka gereken bir işlemdir. Yoksa linker com dosya oluşturamaz. İkinci önemli nokta org 100h'tan dolayı program 7C00'a yüklendiğinde semboller için offset'ler hep 100h fazla çıkacaktır. Bu durumda ds register'ının 07B0'ı göstermesi gerekir. üçüncü önemli nokta ise BPB bloğunu bozup bozmayacağımızdır. Eğer BPB bloğunu bozmak istemiyorsak programımızı BPB bloğunun ötesine yüklemeliyiz. Bütün bu bilgiler doğrultusunda ekrana "Loading CSD Operating System" yazısını basan sembolik makine dili programı şöyle yazılabilir:

```
/*****com00002.asm*****/
_TEXT segment para public 'CODE'
    assume CS:_TEXT, DS:_TEXT, SS:_TEXT
    org     100h
main proc near
    jmp     @1
    msg     db      "Loading CSD Operating System", 0
@1:
    mov     ax, 07b0h
    mov     ds, ax
    call    writestr
INFINITE:
    jmp     INFINITE
main endp

writestr proc near
    mov     si, offset msg
    jmp     @2
@3:
    mov     ah, 0Eh
    mov     bh, 0
    int     10h
    inc     si
@2:
    mov     al, [si]
    cmp     al, 0
    jnz     @3
    ret
writestr endp
_TEXT ends

end main
/*****com00002.asm*****/
```

Com programı hazırlamanın dışında başka bir alternatif yöntem şu olabilir: Yine tek segment'li bir program yazılır. Yani relocatable bilgi kullanılmaz. Bütün segment register'lar aynı segment'e assume edilir. Örneğin:

```
_TEXT segment para public ‘CODE
        assume cs:_TEXT, ds:_TEXT, ss:_TEXT
_TEXT ends
```

Böylece program com değil exe yapılır. Artık org 100h gerekmeyecektir. Böylece üretilen offset'ler de 100h'tan değil 0'dan başlayacaktır. Bundan sonra exe dosyanın başlık kısmı atılarak yükleme gerçekleştirilebilir. Com yerine exe üretilmesiyle 100h karışıklığı engellenmiş olur.

Artık com programını boot sektöre yerleştirmek gerekmektedir. Bu işlem ayrı bir programla yapılabilir.

```
/*****putboot.c*****/
#include <stdio.h>
#include <bios.h>
#include <stdlib.h>

#define TRYNUM        3
#define SECTSIZE         512

void main(int argc, char *argv[])
{
   FILE *f;
   char buf[SECTSIZE];
   int i, result;

   if (argc != 2)
   {
      fprintf(stderr, "Wrong number of arguments..\n");
      exit(1);
   }
   if ((f = fopen(argv[1], "rb")) == NULL)
   {
      fprintf(stderr, "Cannot open file %s\n", argv[1]);
      exit(1);
   }
   fread(buf, 1, SECTSIZE, f);
   for (i = 0; i < TRYNUM; ++i)
   {
      result = biosdisk(3, 0, 0, 0, 1, 1, buf);
      if (!result)
         break;
   }
   if (result)
   {
      fprintf(stderr, "Cannot locate boot file..\n");
      exit(1);
   }
   printf("Boot sector successfuly loaded");
}
/*****putboot.c*****/
```

İşletim Sistemi Yazımında İzlenecek Yöntem

İşletim sistemi en aşağı seviyeli sistem programıdır. Çünkü BIOS kesmelerinin dışında kullanılacak hiçbir yazılmış kod yoktur. Ancak yine de sistemin %95'i C ile yazılabilir. Geri kalan %5'i doğrudan bire bir register kullanan aşağı seviyeli kodlar olup sembolik makine dilinde yazılmalıdır. Peki bu durumda hangi C derleyicisi kullanılacaktır? Herhangi bir derleyici kullanılabilir. Ancak derleyiciler belirli bir işletim sistemi için yazıldığından o işletim sisteminin sistem fonksiyonlarını kullanırlar. Oysa söz konusu işletim sistemi yüklü değildir. Belli başlı problemler şunlardır:

1. Derleyiciler link işleminde startup modül kullanırlar. main fonksiyonu buradan çağırılmaktadır. Örneğin Borland derleyicilerinde small modelde startup modül c0s.obj dosyasıdır. Bu dosya yeniden oluşturulmalıdır. Örneğin startup modül aşağıdaki gibi olabilir:

```
_TEXT segment para public 'CODE'
entry:
      call _main
INFINITE:
      jmp INFINITE
_TEXT ends
end entry
```

2. Standart C fonksiyonlarının pek çoğu da işletim sistemine bağımlı olacak şekilde tasarlanmıştır. Öreğin malloc, printf fonksiyonları DOS derleyicilerinde hep int 21h kesmesini çağırır. Çoğu kez string.h içerisindeki fonksiyonlar rahatlıkla kullanılabilir. Yani standart C fonksiyonlarını kullanırken bu fonksiyonların işletim sistemine bağımlı olup olmadığı test edilir.
3. Derleyicinin nihai çıktısı com ya da exe dosyalarıdır. Com ya da relocation tablosu 0 olan exe programları tercih edilebilir.

Modüllerle Çalışma

Bilindiği gibi modüllerle program yazarken bir global değişken C'de yalnızca bir modülde global olarak tanımlanmalı, kullanılan tüm modüller de extern olarak bildirilmelidir. C'de başka bir modülde tanımlanmış olan bir fonksiyonun çağrılması için extern bildiriminin yapılmasına gerek yoktur. Fonksiyonlar zaten otomatik olarak extern kabul edilir. Tabii derleme zamanı için prototip bulundurmak gerekir. Linker programı birden fazla .OBJ modülü girdi olarak alabilir. Ayrıca .LIB dosyalarını da girdi olarak alabilmektedir. TLINK.EXE programı şöyle kullanılır:

TLINK objfiles,exefile,mapfile,libfiles

1) *TLINK a1 a2*
2) *TLINK a1 a2,myapp*
3) *TLINK a1 a2,myapp,mymap,mylib*
4) *TLINK a1 a2,myapp,,mylib*
5) *TLINK a,,,mylib*
6) *TLINK a,,,mylib yourlib*

Bir global değişkenin başka bir modülde kullanılabilmesi için OMF formatında **pubdef record**'a yazılması gerekir. Sembolik makine dili derleyicilerinin bir data ya da code sembolünü pubdef record'a yazabilmesi için **public** bildiriminin yapılması gerekir. public bildirimi aşağıdaki gibi yapılır:

public sembol_ismi

public bildirimi sembolik makine dilinde herhangi bir yerde yapılabilir. public bildirimi ile derleyici sembolün hangi segment ve hangi offset'te bulunduğunu pubdef record içerisine yazmaktadır. Pubdef record içerisine yazılmayan bir sembol başka bir modülden kullanılamaz. C'de default olarak bütün global değişken ve fonksiyonlar C derleyicileri tarafından public olarak yazılmaktadır. Static global değişkenler ve fonksiyonlar pubdef record'a yazılmazlar. Bu durumda başka bir modülde extern olarak bildirilseler bile kullanılamazlar. Linker extern değişkenlerin makine kodlarını düzeltebilmek için pubdef record'daki bilgilere gereksinim duymaktadır. Örneğin x başka bir modülde tanımlanmış extern bir değişken olsun. Biz program içerisinde

x = 10;

yazmış olalım. Derleyici bu işlemi makine komutuna dönüştürürken x başka bir modülde olduğu için offset değerini bilemez ve offset değerini boş bırakır.

```
mov     word ptr[0000], 10
```

Burada altı çizili olan offset değeri linker tarafından düzeltilecektir. Linker'ın bu düzeltmeyi yapabilmesi için sembolün offset değerini pubdef record'dan elde etmesi gerekir.

Başka modülde public olarak bildirilmiş sembolün kullanılabilmesi için o sembolün kullanılacak modülde **extdef record**'da belirtilmiş olması gerekir. Bir sembolü extdef record'a yazabilmek için **extrn** bildiriminin yapılmış olması gerekir. Extdef record'da sembole ilişkin isim bilgisi bulunmaktadır. Linker bir modülün extdef record'u içerisinde belirtilen sembolü diğer modüllerin pubdef record'larında arar. Bulursa ilişkilendirme yaparak koddaki offset bilgisini düzeltir, bulamazsa link aşamasında error oluşur. Bir sembolün birden fazla modülde pubdef record içerisinde bulunması da error oluşturur. Aynı sembol ne olursa olsun birden fazla modülde pubdef record içerisinde ise yine error oluşur. extrn bildiriminin sembolik makine dilinde bildiriminin genel biçimi şöyledir:

Kod sembolleri için:

```
extrn sembol:   near
                far
```

Data sembolleri için:

```
extrn sembol:   byte
                dword
                qword
                tbyte
```

Örneğin:

```
extrn x1:word
extrn exit:near
extrn x1:byte, x2:word, exit:near
```

extrn bildirimi kaynak kodun herhangi bir yerinde yapılabilir. Ancak Microsoft şu tavsiyede bulunmaktadır:

1) Data sembolleri public karşılığı hangi segment içerisinde yapılmışsa o segment'in içerisinde tanımlanmalıdır. Örneğin x1 sembolü başka bir modülün _DATA isimli segment'inde tanımlanmış olsun. O zaman extrn bildirimi için dummy bir segment tanımlaması açmak gerekir.

```
_DATA segment
    extrn    x1:word
_DATA ends
```

2) near code sembolleri hangi segment içerisinde kullanılacaksa o segment içerisinde extrn bildirimi yapılmalıdır. Örneğin

```
_TEXT segment
    extrn    _func:near
    .....
    .....
    call     _func
    .....
    .....
_TEXT ends
```

3) far code sembolleri kaynak kodun herhangi bir yerinde extrn olarak bildirilebilir.

Microsoft'un bu önerileri bir yana extrn bildirimleri herhangi bir yerde yapılabilir. Aslında OMF formatında sembolün tür bilgisi extdef record içerisine yazılmamaktadır. Tür bilgisi pubdef record içerisinde bulunmaktadır. Extdef record içerisinde yalnızca değişkenin ismi bulunur. extrn bildiriminde tür bilgisinin yazılmasının nedeni derleyicinin syntax bakımından tür kontrolünün uygulanmasını sağlamak içindir.

C derleyicileri geleneksel olarak global ve extern değişkenleri pubdef ve extdef record'larına başına alt tire getirerek yazarlar. C derleyicilerinin sembolün başına alt tire eklemelerinin nedeni sembolik makine dili çıktılarında yanlışlıkla anahtar sözcük oluşturacak isimlerin engellenmesi içindir.

Derleyicilerin tümleşik çevreli versiyonlarında link işlemine otomatik olarak startup modul ve standart C kütüphaneleri dahil edilmektedir. Örneğin Borland derleyicilerinde DEN.C isimli bir C dosyasının link edilmesinin komut satırı karşılığı şöyledir (small modelde derlendiği varsayılmıştır):

```
TLINK c:\tc\lib\c0s den,den,,c:\tc\lib\cs c:\tc\lib\maths c:\tc\lib\emu
c:\tc\lib\graphics
```

Bilindiği gibi main fonksiyonu startup modul tarafından çağrılmaktadır. Eğer C0.ASM dosyası incelenirse çağırma işleminin aşağıdaki gibi yapıldığı görülür:

```
extrn    _main:near
.....
.....
call     _main
```

Görüldüğü gibi main fonksiyonunun aranması startup modul içerisinde çağrılmasından kaynaklanmaktadır. Bu durumda C programında main olmamasından dolayı oluşacak hata link aşamasında linker'ın diğer modüllerde _main sembolünü pubdef record içerisinde bulamamasından dolayı ortaya çıkacaktır.

Communal Tanımlama

Bir sembol hem public hem de extern olarak bildirilirse public bildirim yapıldığı kabul edilir. **comm** bildirimi ile bir sembol belirtildiğinde o sembol için derleme aşamasında yer ayrılmaz. Yer link aşamasında yakın modellerde C_COMMON, uzak modellerde FAR_BSS isimli segment içerisinde ayrılır. Bu segment'i linker oluşturmaktadır ve DGROUP isimli gruba dahil etmektedir. COM bildiriminin genel biçimi şöyledir:

```
comm  sembol:uzunluk:miktar
```

Örneğin:

comm buffer:byte:128
comm num:word

comm kullanımı programcıyı bir sembolü bir modülde public, diğer modüllerde extern yapma zahmetinden kurtarır. Bu durumda comm bildirimi aynı biçimde her modülde tanımlanır. Hiçbir modülün derlenmesinde bir problem ortaya çıkmaz. Yer link aşamasında bir tane ayrılır.

String Komutları

Sembolik makine dilinde ismine string komutları denilen bir grup makine komutu vardır. Bu komutlara string komutları denmesine karşın yazı işlemleriyle bir ilgisi yoktur. Bu komutlar genel olarak blok işlemi yaparlar. Bu komutlarda bir kaynak bölge söz konusu olduğu zaman DS:SI, hedef bölge söz konusu olduğu zaman ES:DI ikilisi kullanılır.

Komutlarda Önek(prefix)

Intel işlemcilerinde tek başına anlamlı olmayan, onu izleyen komut ile anlam taşıyan 1 byte'lık makine komutları vardır. Bu komutlara önek denir. En çok kullanılan önekler 66, 67, F3 (REP), D0-D7(ESC), DS:, ES:, SS:, CS: önekleridir.

66 ve 67 Önekleri

66 öneki gerçek modda ve V86 modunda register operandının yani işlem genişliğinin 16 bit'mi yoksa 32 bit'mi olduğunu anlatır. Bu modlarda 66 öneki komutun önünde ise yapılan işlemde 32 bit register takımı kullanılır. Yani gerçek modda ve V86 modunda 32 bit register'lar kullanabilmek için 66 öneki komuta eklenmektedir. Korumalı modda doğal çalışma 32 bit işlem genişliği içerir. Bu modda tam tersine 32 bit ve 8 bit register'ları kullanmanın bir maliyeti yoktur. Ancak 16 bit register kullanıldığında 66 öneki komuta eklenir.

67 öneki ise gerçek modda ve V86 modunda 32 bit offset genişliği kullanıldığında komuta eklenir. Korumalı modda doğal olan durum zaten 32 bit offset genişliğidir. Orada da tam tersi 16 bit offset genişliği kullanıldığında 67 öneki komuta eklenir. Özetle gerçek modda ve V86 modunda 32 bit register kullanmanın ve 32 bit offset kullanmanın 66 ve 67 öneklerini komuta eklemesi dolayısıyla bir maliyeti vardır. Korumalı modda da tam tersine 16 bit register ve offset genişliği kullanmanın maliyeti vardır.

Bu maliyetler push ve pop işlemleri için de geçerlidir. Sembolik makine dili derleyicisi hangi mod için program yazıldığını varsayacaktır? Örneğin Win32 için kod yazarken 32 bit register'ları kullanacağımız zaman 66 önekinin gelmemesi gerekir. İşte ayrıntılı segment tanımlamalarında gerçek mod / V86 mod ya da korumalı mod için kod yazılıp yazılmadığı **use16** ve **use32** anahtar sözcüklerinin eklenmesiyle belirlenir.

```
                use16
MYSEG segment          para public 'DATA'
                use32
      .....
```

```
        .....
MYSEG ends
```

Default durum use16'dır. Eğer kaynak kodun başına .386, .386p, .486, .486p eklenirse o zaman default durum use32 olur.

Segment Önekleri

Segment yükleme için kullanılan DS:, ES:, SS:, CS: sembolleri aslında birer önektir. Örneğin

```
mov             ax, ss:[bx]
```

işlemi aslında ss: öneki ile onu izleyen

```
mov             ax, [bx]
```

komutlarından oluşmaktadır. Bu durumda ss: öneki aslında "bir sonraki makine komutunda bellek operandı varsa onun segment register'ı ss yap" anlamına gelmektedir. Hatta bazı debugger'lar komutun

```
mov             ax, ss:[bx]
```

biçiminde girilmesine izin vermez.

```
ss:
mov             ax, [bx]
```

biçiminde girilmesine izin verir.

REP Öneki

Bu önek string komutlarıyla birlikte kullanılmaktadır. Bu öneki bir string komutu izlemelidir.

String komutlarının BYTE ve WORD versiyonları vardır.

LODS Komutu

Bu komutun BYTE versiyonu **LODSB**, WORD versiyonu **LODSW**'dur. Komut DS:SI adresindeki bilgiyi BYTE versiyonunda AL register'ına, WORD versiyonunda AX register'ına yükler ve DF bayrağının durumuna göre SI register'ını artırır ya da eksiltir.

LODSB Komutu

```
AL = DS:SI
if (DF == 0)
        SI = SI + 1
else
```

SI = SI – 1

LODSW Komutu

```
AX = DS:SI
if (DF == 0)
        SI = SI + 2
else
        SI = SI - 2
```

Bu komut REP önekiyle kullanılmaz. Bu komut bir blok bilgiyi AX ya da AL'ye alıp işlemek için özellikle tercih edilir. Blok bilgi DS:SI register'larıyla belirlenmelidir. Ondan sonra CLD ya da STD komutlarıyla DF bayrağı ayarlanmalı ve bir döngü içerisinde bir grup bilgi işlenmelidir. Örneğin:

```
        initport db '$CHD0000'
        .....
        cld
        lea             si, initport
        mov             cx, 8
AGAIN:
        lodsb
        out             250, al
        dec             cx
        jnz             AGAIN
```

STOS Komutu

Bu komutun BYTE versiyonu **STOSB**, WORD versiyonu **STOSW** biçimindedir. Komut REP önekiyle kullanılabilir.

REP Önekinin İşlevi

REP öneki yalnızca string komutlarıyla kullanılır. String komutunun bir kez değil, CX register'ının içerisindeki değer kadar yapılmasını sağlar. REP komutu önce CX register'ının değerini 1 eksiltir. Önce CX register'ı kontrol edilir. 0 dışı bir değerse string işlemini yapar, 0 ise string işlemini sonlandırır. Sonra CX register'ının değeri 1 azaltılır. Komutun kullanımı

```
rep     stosb
rep     stosw
```

biçimindedir. Görüldüğü gibi önek komutun bir parçasıymış gibi kullanılır. STOS komutu şu işlemi yapar.

STOSB

```
ES:DI = AL
if (DF == 0)
        DI = DI + 1
else
        DI = DI – 1
```

STOSW

```
ES:DI = AX
if (DF == 0)
        DI = DI + 2
else
        DI = DI – 2
```

STOS komutu REP önekiyle birlikte kullanıldığında aşağıdaki gibi işlem yapar.

```
while(CX > 0) {
        stos
        --cx;
}
```

STOS komutu REP öneksiz de kullanılabilir. REP STOS işlemi özellikle bir bloğun belirli bir değerle doldurulması gibi işlemlerde tercih edilmektedir. Yani tipik memset gibi bir fonksiyon bu komut kullanılarak yazılabilir. Örneğin bir bloğun 100 byte 0'la doldurulması şöyle yapılabilir:

```
mov             ax, seg block
mov             es, ax
les     di, block
cld
mov             al, 0
mov             cx, 100
rep             stosb
```

Small model memset örneği:

void *memset(void *block, char ch, unsigned n);

```
_memset proc near
        push    bp
        mov             bp, sp
        push    di
        mov             ax, ds
        mov             es, ax
        mov             di, [bp + 4]
        mov             cx, [bp + 8]
        mov             al, [bp + 6]
        cld
        rep     stosb
        mov             ax, [bp + 4]
        pop             di
        pop             bp
        ret
_memset endp
```

MOVS Komutu

Bu komutun BYTE biçimi MOVSB, WORD biçimi MOVSW biçimindedir. Komut REP önekiyle kullanılabilir. Uygulamada bir bloğun başka bir bloğa kopyalanması amacıyla kullanılmaktadır. Çalışma biçimi şöyledir:

MOVSB

```
ES:DI = DS:SI
```

```
if (DF == 0) {
        SI = SI + 1
        DI = DI + 1
}
else {
        SI = SI - 1
        DI = DI - 1
}
```

MOVSW

```
ES:DI = DS:SI
if (DF == 0) {
        SI = SI + 2
        DI = DI + 2
}
else {
        SI = SI - 2
        DI = DI - 2
}
```

REP MOVS işlemi de şöyledir:

```
while (CX > 0) {
        movs
        --cx;
}
```

MOVS komutu tipik olarak memcpy gibi bir fonksiyonu yazmakta kullanılabilir.

void *memcpy(void *dest, const void *source, size_t size);

```
_memcpy proc near
        push    bp
        mov             bp, sp
        push    si
        push    di
        mov             ax, ds
        mov             es, ax
        mov             di, [bp + 4]
        mov             si, [bp + 6]
        mov             cx, [bp + 8]
        cld
        rep             movsb
        mov             ax, [bp + 4]
        pop             di
        pop             si
        pop             bp
        ret
_memcpy endp
```

Heap Algoritması

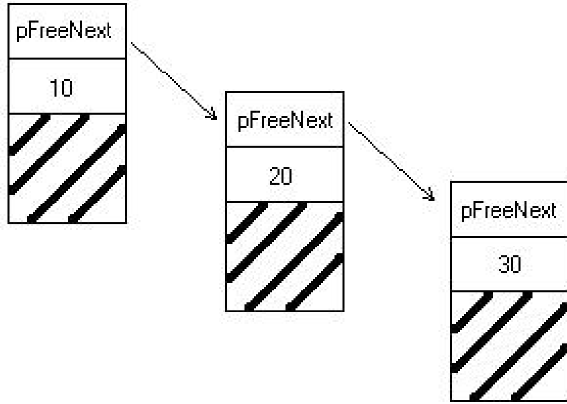
Heap dinamik bellek fonksiyonlarıyla tahsis edilme potansiyelinde olan boş bölgelerdir. Heap organizasyonu için önce bir heap bölgesinin belirlenmesi gerekir. heap bölgesinin neresi olacağı işletim sisteminin tasarımına bağlıdır. DOS'ta heap bölgesinin yeri ve uzunluğu sembolik makine dili kursunda ele alınacaktır. Heap alanı belirlendikten sonra artık malloc, free, realloc gibi tahsisat fonksiyonlarını yazmak gerekir. Bu fonksiyonlar bir tahsisat tablosu kullanmak zorundadır. Tahsisat tablosunun heap içerisinde yer alması gerekir. ancak bu tablonun da dinamik olarak büyütülüp küçültülmesi gerektiğinden tablo sabit bir yerde ve sabit uzunlukta olmamalıdır. DOS'ta, Win32'de ve UNIX sistemlerinde kullanılan malloc algoritması "boş bağlı liste" tekniğini kullanan bir algoritmadır. Bu algoritma Ritchie & Kernighan'ın "The C Programming Language" kitabında açıklanmıştır.

Boş Bağlı Liste Algoritması

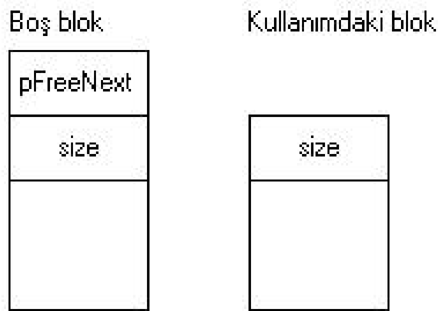
Bu yöntemde bir tahsisat yapıldığında malloc fonksiyonu istenen alan kadar byte'ın yanı sıra o bloğun bilgileri içeren n kadar byte daha tahsis eder. Blok bilgileri en azından tahsis edilen bloğun uzunluğu ve sonraki boş bloğun adresini içermelidir.

Tahsis edilmiş bir bloğun görüntüsü

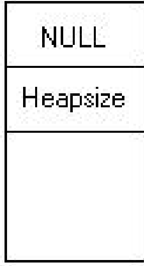
Bu yöntemde bütün boş bloklar bir bağlı liste içerisinde tutulur. Tahsis edilen bloklar ayrıca bağlı listede tutulmaz. Zaten tutulmasına da gerek yoktur. Örneğin heap bölgesinde 10, 20 ve 30 byte'lık boş alanlar olsun. Bu boş alanlar aşağıdaki gibi bir bağlı listede tutulabilir.



Bu algoritmada iki blok söz konusudur. Birincisi boş blok(bu blok bağlı listenin içerisindedir), ikincisi kullanımda olan blok. Kullanımda olan blok için pFreeNext gibi bir göstericiye gerek yoktur.



Peki heap bölgesinin başlangıç durumu nasıldır? Başlangıçta boş bağlı listede bir tane eleman vardır. O da tüm heap bölgesini kapsar.



Bu heap algoritmasını gerçekleştirirken oluşacak problemli durumlar şunlardır:

1. Tahsisat yaparken büyük bir boş blok içerisinde küçük bir blok çekilip alınacaksa (büyük blok – küçük blok) kadar alan boş bağlı listeye yazılmalıdır.
2. Boş bağlı listedeki bloklar ne kadar büyük olursa o kadar iyidir. Bu yüzden free işlemi sırasında boşaltılan alan boş bir bloğun altındaysa büyük ve tek bir blok halinde yeni bir blok oluşturulmalıdır.
3. Minimum bir boş blok uzunluğu seçilebilir. Böylece bağlı listede çok küçük bloklar tutulmaz. Ancak bu tasarım bellek verimini düşürebilir. ,
4. realloc fonksiyonu eski bloğun altında boş alan bulmak için araştırma yapmalıdır.

TINUX Sisteminde Heap Organizasyonu

Birden fazla heap kullanmanın bellek bölünmesi bakımından yararları vardır. Örneğin Win32 sistemlerinde programcı birden fazla heap kullanabilmektedir. TINUX sisteminde de sistemin kendi modüllerinin ayrı birer heap alanı olmalı ve aynı zamanda programcı kendisi birden fazla heap bölgesi yaratabilmelidir. TINUX, UNIX sistemlerini desteklediğine göre klasik olarak process'in tek bir heap'i olmalıdır. Ancak bu sistem hem POSIX standartlarına destek verecek hem de kendine özgü API fonksiyonları olacaktır. Bu durumda bir process içerisinde birden fazla heap yaratılabilecek ancak heap'lerden birisi process çalıştırılmaya başlayınca yaratılacak. Bu heap process'in default heap'i olacak. POSIX standardındaki dinamik bellek fonksiyonları bu process'in default heap'ini kullanacaktır. TINUX sistemi aşağı seviyeli heap API fonksiyonları şunlardır:

```
HHEAP CreateHeap(DWORD dwHeapSize, DWORD dwProtect);
PVOID AllocMem(HHEAP hHeap, DWORD dwSize);
VOID FreeMem(HHEAP hHeap, PVOID pBlock);
PVOID ReallocMem(HHEAP hHeap, PVOID pBlock, DWORD dwNewSize);
BOOL ReleaseHeap(HHEAP hHeap);
HHEAP GetProcessHeap(VOID);
```

POSIX Fonksiyonları

POSIX malloc, free ve realloc fonksiyonları aşağıdaki teknikle TINUX multiheap API'lerini kullanarak process'in default heap'i üzerinde tahsisat yapmaktadır.

```
void *malloc(size_t size)
{
        HHEAP hHeap;

        hHeap = GetProcessheap();
        return AllocMem(hHeap, size);
```

}

Buradaki fonksiyonlar sistem fonksiyonları grubundadır. Halbuki işletim sisteminin kendi modülleri içerisindeki heap yaratılması için küçük bazı ekler yapmak gerekir. CreateHeap fonksiyonuyla verilen handle hangi alanda olmalıdır? Process'in alanında olmamalıdır. Bunun iki nedeni vardır:

1. Sistemin güvenliği
2. Process'in default heap'i içerisinde bu bilgi tutulabilir ki bu da default heap'e önemli bir ayrıcalık kazandırır.

Genel olarak bir sistem fonksiyonunun verdiği handle kernel modülünün heap bölgesi içerisinde olmalıdır. Kernel modülünün heap bölgesi için ayrı fonksiyonlar tasarlanabilir. Bu fonksiyonlara handle geçirilmesine gerek yoktur. Kernel heap bilgileri(heap bölgesinin başlangıcı, uzunluğu, boş bağlı listenin head göstericisi gibi) kernel modülü içerisinde bilinen bir yerde tutulur ve aşağıdaki fonksiyonlarla idare edilebilir:

```
PVOID allocKernelMem(DWORD dwSize);
VOID freeKernelMem(PVOID pBlock);
PVOID reallocKernelMem(PVOID pBlock, DWORD dwNewSize);
```

Kernel Heap Fonksiyonlarının Tasarımı için Yapılacak İşlemler

Kernel heap bilgileri için aşağıdaki global değişkenler alınır:

```
PVOID g_pKernelHeap;
DWORD g_kernelHeapSize;
HEAPLIST *g_pHead;
HEAPLIST *g_pTail;
DWORD g_listSize;

typedef struct _HEAPLIST {
        struct _HEAPLIST *pNext;
        DWORD blockSize;
        BYTE pBlock[MINSIZE];
}HEAPLIST, *pHEAPLIST;
```

Heap sistemini tasarlayacak grup için anahtar notlar:

- Önce kernel heap fonksiyonları tasarlanmalıdır. Daha sonra heap API fonksiyonları tasarlanmalıdır.
- Kernel heap fonksiyonlarının denemesi yapılırken g_pKernelHeap(kernel heap alanının başlangıç adresi) ve g_kernelHeapSize(kernel heap alanının uzunluğu) değişkenlerine belirli değerler atanmalıdır.
- Bütün kernel fonksiyonları çok iyi test edilmelidir.
- Sonra user API heap fonksiyonları tasarlanmalı, CreateHeap fonksiyonu içerisinde hHeap handle alanı allocKernelMem fonksiyonu kullanılarak kernel içerisinde tahsis edilmelidir. ReleaseHeap fonksiyonu ise freeKernelMem fonksiyonunu kullanmalıdır.

SCAS Komutu:

Bu komutun byte biçimi SCASB, word biçimi SCASW biçimindedir. Bu komut aslında

```
CMP     AL,ES:DI
        AX
```

işlemini yapmaktadır. Komutun çalışması şöyledir:

```
SCASB

CMP     AL,ES:DI

if(DF == 0)
        DI = DI + 1;
else
        DI = DI - 1;


SCASW

CMP     AX,ES:DI

if(DF == 0)
        DI = DI + 2;
else
        DI = DI - 2;
```

Görüldüğü gibi bu komut AL ya da AX register'ıyla yalnızca ES:DI adresindeki bilgiyi karşılaştırır. İşlemin yinelemeli olarak devam edebilmesi için başına REP komutunun getirilmesi gerekir. Aslında REP komutu bir değil iki tanedir. Şimdiye kadar kullandığımız REP komutu aynı zamanda REPE ya da REPZ olarak da bilinir. Bunun dışında bir de REPNE / REPNZ komutu vardır. REPNE ve REPNZ aynı komuttur.

1) REP REPE = REPZ
2) REPNE = REPNZ

REPNE = REPNZ diğer string komutlarıyla kullanılmaz. Yalnızca SCAS komutuyla birlikte kullanılır.
Bu durumda ;

REPE SCAS işlemi bir bloktaki ilk eşit olmayan bilgiyi bulmak için,

REPNE SCAS işlemiyse bir bloktaki ilk eşit bilgiyi bulmak için kullanılır.

REPNE SCAS işlemi daha sık kullanılmaktadır.

```
LOST_A DB       100 DUP(*)
....

        MOV     AX,DS
        MOV     ES,AX
        CLD
        MOV     AL,'A'
```

```
        MOV     CX,100
        LEA     DI,LOST_A
        REPNE   SCASB
        JZ      FOUND
NOTFOUND:
        ....
        ....
        ....
FOUND:
        DEC     DI
```

Burada REPNE SCAB işleminden iki nedenle çıkılmış olabilir. Birincisi A karakteri bulunmuştur. Dolayısıyla son CMP işleminden JF=1 oluşmuştur. İkincisi FZ sayacı 0 olmuştur ve karakter bulunamamıştır. Bu durumda FZ = 1 olmuştur. CMP işleminden sonra her durumda DI artırılacak ya da azaltılacaktır. Bunun için karakterin bulunduğu noktada DI 1 eksiltilmiştir. SCAS komutu tipik olarak memchr() standart C fonksiyonunu yazmak amacıyla kullanılabilir.

Sınıf Çalışması: Aşağıdaki C kodunu sembolik makine dilinde yazınız.

/*C Kodu*/	Makine dilindeki karşılığı

void *mymemchr(void *pBlock, char ch, unsigned n)
{
char S[] = "Bu bir denemedir.";
char *p;
 	
p = (char*) mymemchr(s, 'i', strlern(s))
if(p == null) {
printf("Bulunamadı!..\n");
else
printf("Bulundu!..\n");
}
}
	model small
	
	.code
	_mymemchr proc near
	
	push bp
	mov bp, sp
	push di
	
	mov ax, ds
	mov es, ax
	mov di, [bp + 4]
	cld
	mov al, [bp +6]
	mov cx, [bp + 8]
	REPNE SCASB
	jz FOUND
	
	NOTFOUND:
	xor ax, ax
	jmp PROC_END
	
	FOUND:
	dec di
	mov ax, di
	PROC_END:
	pop di
	pop bp
	ret
	
	_mymemchr endp
	
	public _mymemchr
	end

Örnek Thread Kütüphanesi:

Klasik DOS sistemi thread'li çalışmaya olanak vermez. Ancak WIN32 sistemleri thread'lerle çalışmayı desteklemektedir. Klasik UNIX sistemlerinde de thread'li çalışma yoktur. Ancak LINUX sistemlerinde buradaki örnekte olduğu gibi bir thread kütüphanesi eşliğinde thread'lerle çalışma sağlanabilmektedir. Buradaki örnek kütüphane DOS'ta thread'lerle çalışmayı olası duruma getirmek amacıyla tasarlanmıştır. Buradaki kodlar tipik bir preemtif (*İşletin sistemi tarafında kesilen*) işletim sisteminde thread'ler ya da prosesler arasındaki geçiş mekanizmasının nasıl yapıldığını da açıklamaktadır. Bu kütüphanedeki fonksiyonların isimlendirilmesi ve işlevleri WIN32 sistemlerine benzetilmiştir. Kodlama içinde kullanılan isimlendirme kuralları gevşetilmiş Macar notasyonudur. Buna ek olarak aşağıdaki özelliklere dikkat edilmiştir.

1) Bütün global değişkenler 'g_' ile başlatılmıştır.

2) Dışarıya kapalı fonksiyonlar, yani API olmayan fonksiyonların ilk sözcükleri küçük harfle başlatılmıştır. Örneğin; addItem() gibi.

3) Sembolik makine dilinde yazılmış kodların sonuna _ASM eklenmiştir.

4) Başka modülden kullanılacak fonksiyonun başına PUBLIC, kullanılmayacak fonksiyonların önüne PRIVATE sembolik sabitleri getirilmiştir.

Thread Kütüphanesinin kullanımı:

InitThreadLib() fonksiyonu:

Bu fonksiyon thread kütüphanesini initialize eder. Bu yüzden programın başında bir kez çağırılmalıdır.

Prototipi:

BOOL InitThreadLib(void);

CreateThread() Fonksiyonu:

Bu fonksiyon thread akışını yaratır.
Prototipi:

int CreateThread(THRPROC pThrProc, PVOID pParam, WORD stackSize);

Fonksiyonun birinci parametresi thread akışının başlatılacağı fonksiyonun başlangıç adresidir. Thread fonksiyonunun prototipi aşağıdaki gibi olmak zorundadır:

int ThreadProc(PVOID pParam);

Fonksiyonun ikinci parametresi thread fonksiyonu çalışmaya başladığında geçirilecek parametredir.

Fonksiyonun üçüncü parametresi yaratılacak thread'in stack uzunluğudur.

Fonksiyon başarılıysa thread'in handle değerine, başarısızsa -1 değerine geri döner. Bu değer THREAD_ERROR sembolik sabiti biçiminde tanımlanmıştır.

ExitThread() Fonksiyonu:

Bu fonksiyon o anda çalışmakta olan thread'i sonlandırmak amacıyla kullanılır. Eğer ExitThread() çağırılmazsa thread, thread fonksiyonunun sonunda sonlandırılır.

BOOL ExitThread(int exitCode);

Thread'in sonlandırılması thread için ayrılan handle alanının silinmesini sağlamaz. Thread'in exit code'u bu handle alanına yazılır.

GetThreadExitCode() Fonksiyonu:

Bu fonksiyon sonlandırılmış fakat handle alanı silinmemiş bir thread'in exit code'unun alınması için kullanılır.

int GetThreadExitCode(int thrID);

CloseHandle() Fonksiyonu:

Bu fonksiyon thread için ayrılan handle alanını yok eder.

BOOL CloseHandle(int thrID);

Fonksiyonun parametresi thread'in ID değeridir. Geri dönüş değeri işlemin başarısı ya da başarısızlığını belirtir.(BOOL)

EndThreadLib() Fonksiyonu:

Bu fonksiyon thread kütüphanesini kapatır. En sonunda bir kez çağırılmalıdır.

void EndThreadLib(void);

Thread Kütüphanesinin Kullanılmasında Dikkat Edilmesi Gereken Noktalar:

DOS'un tasarımı multithread bir çalışma düşünülmeden yapılmıştır. DOS'un sistem fonksiyonlarının çoğu SS:SP register'larını belirli bir değere çekip stack değiştirirler. Bu durumda thread'lerden birisi DOS fonksiyonunu çağırırken diğeri de çağırırsa öncekinin stack datalarını ezer ve problem ortaya çıkar. Aynı problem DOS'ta memory resident program yazarken de ortaya çıkmaktadır. Ayrıca DOS'ta kullandığımız standart C kütüphanesi de çoklu thread yapısını desteklememektedir. Çünkü bazı fonksiyonlar statik data kullanmaktadır. Özetle şu noktalara dikkat edilmelidir:

1) Özellikle stdio.h içinde olan fonksiyonlar bir thread'te kullanılıyorsa diğer thread'lerde kullanılmamalıdır.

2) Diğer standart C fonksiyonları çoklu thread'e uygun olup olmadığı değerlendirildikten sonra kullanılmalıdır.

Aslında DOS için tamamen çoklu thread'i destekleyen bir C kütüphanesi de yazılabilir.

Thread Kütüphanesinin İçsel Tasarımı:

Çizelgeleme algoritması olarak kullanılan döngüsel çizelgelemede (Round Robin Scheduling) herhangi bir öncelik derecesi kullanılmamıştır. Ancak algoritma öncelik derecelendirmesini destekleyecek biçimde değiştirilebilir.

Her thread'in bilgilerinin saklandığı yapıya thread database denir. Thread database aşağıdaki gibi bir yapıdır:

```
typedef struct _THRDB {
	WORD r_sp, r_ss;
	....
	....
	WORD stackSize;
	void *pStack;
	void *pParam;
	int exitCode;
	int status;
	int next;
	int prev;
}THRDB;
```

Her thread yaratıldıkça bir kuyruk sistemine bu yapı eklenmektedir. Buradaki kuyruk sistemi için bir dizi kullanılmıştır.

```
PRIVATE THRDB g_thrTable[MAX_THREAD];
```

Bu yapı dizisi sanki heap alanı gibi kullanılmıştır. Çift bağlı liste tekniği uygulanarak son yaratılan thread kuyruğun sonuna eklenmiştir. THRDB yapısının status elemanı thread'in ve yapı dizisinin o elemanının o anki durumunu belirtir. Şunlardan bir tanesi olabilir:

THREAD_STATUS_FREE	Slot boş
THREAD_STATUS_EXIT	Thread sonlandırılmış ama slot boşaltılmamış, exit code'u alınabilir. CloseHandle fonksiyonu status elemanını THREAD_STATUS_EXIT'ten THREAD_STATUS_FREE durumuna getirir.
THREAD_STATUS_RUNNING	Thread çalışıyor.

Kuyruk sisteminin başını ve sonunu tutmak için int g_headPos, g_tailPos global değişkenleri kullanılmıştır. Ayrıca o anda çalışmakta olan thread'in ID değeri ve dizideki adresi şu global değişkenlerde tutulmuştur:

```
int g_curThr;
THRDB *g_pCurThr;
```

Kütüphanedeki getFreeItem, addItem ve deleteItem fonksiyonları kuyruk işlemlerini yapmak için tasarlanmıştır. nextThr fonksiyonu yalnızca g_curThr ve g_pCurThr global

değişkenlerinin sonraki thread bilgilerini göstermesini sağlamaktadır. InitThreadLib fonksiyonu ana thread olarak sıfırıncı slot'u kuyruğa ekler. Ayrıca kuyrukta bulunan thread'lerin sayısı g_itemCount global değişkeninde de tutulmaktadır.

Kütüphanede register'lara doğrudan erişmesi gereken kodlar sembolik makine dilinde yazılmıştır. Örneğin cswitch_asm isimli fonksiyon o anda çalışmakta olan thread'in bilgilerini geri yazarak yeni thread'in bilgilerini register'lara yükler. InitThreadLib fonksiyonu içerisinde INT 8 vektörü hook edilmiştir. Artık her INT 8 oluştuğunda cswitch_asm fonksiyonu çalıştırılacaktır. cswitch_asm fonksiyonu çalıştırılacaktır.

C'de Inline Sembolik Makine Dili

C derleyicilerinin çoğu C içerisinde sembolik makine dili yazımını desteklemektedir. Ancak tabii yalnızca makine komutları bu biçimde yazılabilir. Sembolik makine diline özgü anahtar sözcükler yazılamaz. Bunun için Borland derleyicilerinde asm, Microsoft derleyicilerinde _asm anahtar sözcükleri kullanılır. Makine komutu bir satıra gelecek şekilde bu anahtar sözcüklerden sonra yazılır. Örneğin:

```
asm        MOV        ax, bx
```

Birden fazla asm komutu blok içerisine alınabilir.

```
asm {
        MOV         ax, bx
        PUSH        ax
}
```

Bu biçimde yerel değişkenler global değişkenler ve parametre değişkenleri bir data sembolü olarak doğrudan kullanılabilir. Örneğin:

```
int add(int x, int y)
{
        asm {
                Mov         ax, x
                Add         ax, y
        }
}

void main(void)
{
        printf("%d\n", add(10, 20));
}
```

Fonksiyonun stack frame düzenlemesini derleyici yapar. Inline assembler yazımı C standardı değildir. Inline sembolik makine dilinin faydaları şunlardır:

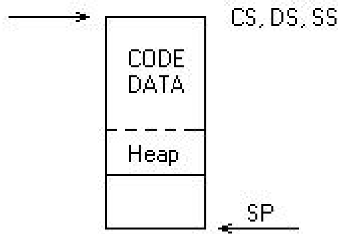
1. Küçük sembolik makine dili kodları için ayrı bir asm modülü yazıp project oluşturmaya gerek kalmaz.
2. Stack frame düzenlenmesi, fonksiyondan çıkış vb gibi ayrıntılarla programcı uğraşmaz.

C derleyicilerinin inline assembler dili yazımını desteklemeleri zorunlu değildir. Örneğin TC 2 derleyicisinin tümleşik çevreli sürümü bunu desteklememektedir.

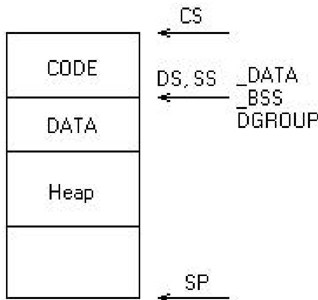
DOS Sisteminde Heap Yönetimi

DOS'ta bir program çalıştırıldığında DOS boştaki tüm belleği process için tahsis eder. Örneğin exe dosya 50Kb olsa, 250Kb boş bellek olsa process için 50Kb değil, 250Kb'ın tamamı tahsis edilecektir. İşte heap organizasyonu derleyicinin başlangıç kodu tarafından process düzeyinde yapılan bir işlemdir. DOS'un heap üretimine ilişkin bir sistem fonksiyonu yoktur. Heap yönetimi tamamen process'in kendisinin yaptığı bir düzenlemedir. Genellikle derleyiciler tarafından yapılır. Derleyicilerin başlangıç kodları içerisinde stack birleştirme biçimine ilişkin 256 byte uzunluğundan bir segment tanımlanmıştır. Ancak bu segment başlangıç modülü tarafından dikkate alınmaz. Derleyicilerin başlangıç kodu main fonksiyonu çağırılmadan önce SS ve SP register'larını, segment register'lar ayarlar ve heap alanını düzenler. Bellek modellerine göre heap durumu şöyledir:

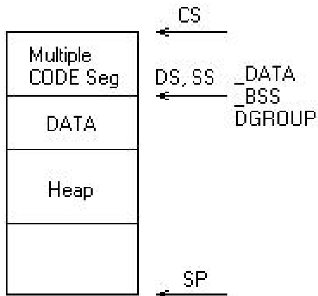
Tiny Model



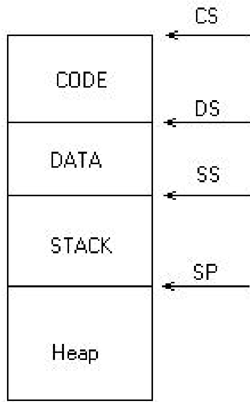
Small Model



Medium Model

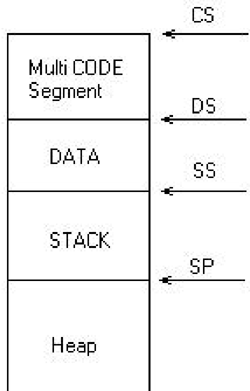


Compact Model



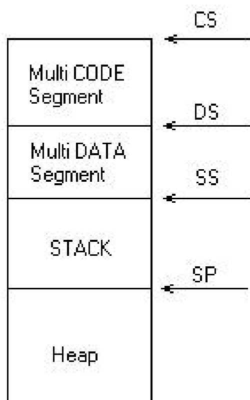
Görüldüğü gibi bu modelde heap tamamen data ve stack bölgesinden yarılmıştır ve exe dosyanın aşağısındaki boş bölgeye alınmıştır. Tabii bütün göstericiler uzak gösterici olmalıdır.

Large Model



Görüldüğü gibi large modelin compact modelden farkı birden fazla code segment olmasıdır.

Huge Model



Huge modelde heap bölgesi yine en aşağıdadır. Ancak birden fazla code ve data segment bulunabilir.