Yrd. Doç. Dr. Ahmet Tevfik İNAN

80x86 ASSEMBLY DİLİ

Gözden Geçirilmiş, Geliştirilmiş, Genişletilmiş
2. Baskı

- Assembly Dili Hakkındaki Doğru ve Yanlışlar
- PC Mimarisi ve 80x86 Ailesi İşlemciler
- Komut Kümesi
- Adresleme Kipleri
- Sözde Komutlar
- EXE ve COM Tipindeki Programlar
- Yordam ve Makro Kullanımı
- Parametre Aktarımı
- Kesme Mekanizması
- Yüksek Seviyeli Diller ile Bağlantı
- Çalışma Ortamının Hazırlanması ve DEBUG Komutları

SEÇKİN | Teknik

80x86 Assembly Dili
Yrd. Doç. Dr. Ahmet Tevfik İNAN

SEÇKİN | **Teknik**
No: 6

ISBN 978-975-02-2496-6

İkinci Baskı: Ağustos 2013 (Ankara)
Birinci Baskı: 2006

Sayfa Tasarımı:
Ömer Candan

Kapak Tasarımı:
İsmail Çam

327 Sayfa, 18,5x23,5 cm.

1. Assembly
2. 80x86
3. Yordam
4. Parametre Aktarımı
5. DEBUG
6. Adresleme
7. Sayı Sistemleri
8. PSEUDO

© Seçkin Yayıncılık San. ve Tic. A.Ş.
Bu kitabın her türlü yayın hakkı Seçkin Yayıncılık San. ve Tic. A.Ş'ye aittir. Yayınevinin yazılı izni olmadan, tanıtım amaçlı toplam bir sayfayı geçmeyecek alıntılar hariç olmak üzere, hiçbir şekilde kitabın tümü veya bir kısmı herhangi bir ortamda yayımlanamaz ve çoğaltılamaz.

Satış ve Dağıtım:

Merkez
Sağlık Sok. No: 21
Sıhhiye / Ankara
Tel: (312) 435 30 30
Faks: (312) 435 24 72
satis@seckin.com.tr

Şube
Strazburg Cad. No: 23/B
Sıhhiye / Ankara
Tel: (312) 230 52 62
ankarasube@seckin.com.tr

Şube
Ankara Adalet Sarayı
K-Blok Zemin Kat
Sıhhiye / ANKARA
Tel: (312) 309 52 48
adliyesube@seckin.com.tr

Şube
Abide-i Hürriyet Cad.
No: 193/A Şişli / İSTANBUL
Tel: (212) 234 34 77
Faks: (212) 231 24 69
sislisube@seckin.com.tr

Şube
Anadolu Adalet Sarayı
C Blok Zemin Kat No: 29
Kartal / İSTANBUL
Tel : (216) 303 11 23
Faks: (216) 303 11 23
kartalsube@seckin.com.tr

Web Sitesi: www.seckin.com.tr

Seçkin Yayıncılık Sertifika No: 12416

Baskı:
Sözkesen Matbaacılık Tic. Ltd. Şti. – **Sertifika No:** 13268
İVOGSAN 1517 Sok. Mat-Sit İşmerkezi No: 2/40 Yenimahalle /ANKARA - Tel: (0–312) 395 21 10

Teşekkür

Kitabın hazırlanma aşamasında; yaptığı düzeltmeler ile desteğini esirgemeyen değerli iş arkadaşım Dr. Banu DİRİ'ye, okuyarak önerileri ile beni yönlendiren Sayın Hocam Prof. Mehmet Yahya KARSLIGİL'e, parçası olmaktan gurur duyduğum Yıldız Teknik Üniversitesi Bilgisayar Mühendisliği Bölümü mensuplarına ve hayatım boyunca her an desteğini hissettiğim Aileme teşekkürü bir borç bilirim.

Dr. Ahmet Tevfik İNAN
İstanbul – 2006

Önsöz

Transistörün icadını takiben elektronik ve (doğal olarak) bilgisayar teknolojisinde pek çok atılım birbirini takip etmiştir. Bilgisayarların düşen fiyatlarına rağmen başarımlarının artmasının arkasında yarı iletken teknolojisindeki ilerlemeler yatmaktadır. 1960'larda hâlâ hantal, özel ortamlarda çalışan, sınırlı sayıda ve bilgili kişilerce yönetilen bir araç olan bilgisayar sistemleri, 1980'lerde, kişisel bilgisayar adı altında günlük hayatımızın vazgeçilmez bir parçası olacağının sinyallerini vermiştir.

Günümüzde boyutları daha da küçülen, başarım ve sahip olduğu özellikler sürekli olarak artan bilgisayarlar artık cepte taşınabilir boyutlara gelmişlerdir. Her ne kadar boyutları küçülmüş olsa da bu tip bilgisayar sistemlerinin kalbini mikro işlemciler oluşturmaktadır.

Mikro işlemciler önceleri 4 bit'lik işlem yapabilme özellikleri ile basit hesaplayıcılarda kullanılmaya başlanmış, zaman içinde gelişen ve değişen ihtiyaçlar doğrultusunda 8, 16, 32 ve günümüzde 64 bit işlem yapabilir hale gelmişlerdir. Farklı üreticiler tarafından değişik özelliklere sahip işlemci aileleri üretilmiş ve üretilmektedir. Günümüz kişisel bilgisayar sistemlerinde tartışmasız olarak pazarı elinde bulunduran Intel'in ürettiği 80x86 işlemci ailesidir. Bu işlemci ailesi ile programlama ara yüzü yönünden uyumlu olan, başka firmaların da ürettiği işlemciler bulunmaktadır.

Bilgisayarımızda kullandığımız sistem ve uygulama yazılımları, hangi programlama dili kullanılarak üretilmiş olurlarsa olsunlar, temelde üzerinde çalıştığı işlemci ailesinin anlayabileceği, 0 ve 1'lerden oluşmuş dizilerdir. İşlemcinin kendine verilen görevleri nasıl yerine getirdiğini anlayabilmek için işlemci içindeki yapının ve bu yapı kullanılarak yapılabilecek işlemlerin neler olduğunun bilinmesi gereklidir. Ancak bu işlemleri 0 ve 1'ler gibi insanoğluna doğrundan hitap etmeyen bir gösterim ile anlamaya çalışmak pek olası değildir. Bu nedenle her işlemci ailesinin yapabileceği işlemlerin tarif edildiği komut kümesinin (Instruction Set) incelenmesi ve bunların işlemcinin hangi özelliklerini kullanarak gerçekleştirildiğinin anlaşılmasında fayda vardır. Böylece üst seviye dillerin nasıl çalıştığını, donanımın buna nasıl karşılık verdiğini anlamak mümkün olduğu gibi, donanımın sahip olduğu özellikleri daha iyi kullanmaya imkân sağlayan programlar yazma becerisini de elde etmek mümkündür.

Intel 80x86 ailesi işlemcilere ait Assembly dili, Bilgisayar Mühendisliği eğitimi içerisinde, Mikro İşlemciler dersine bir ön hazırlık olmanın dışında, Programlama, Derleyici Tasarımı hatta İşletim Sistemleri derslerini bile belirli noktalarda destekler niteliktedir.

Bu kitap, Yıldız Teknik Üniversitesi Bilgisayar Mühendisliği bölümünde verdiğim Assembly Dili dersinde duyulan Türkçe kaynak yetersizliğini bir ölçüde giderebilmek amacıyla oluşturulmuştur. Kitap, Intel 80x86 ailesi işlemcilerde kullanılan assembly dilinin gerçek kipte kullanımını konu almaktadır.

2. Baskının Önsözü

Bu kitap, gerçek kipte 80x86 Assembly Dili ile programlama için bir ders kitabı olarak hazırlanmıştır. Amacımız karmaşık algoritmalar ortaya atıp bunların assembly dili ile programlarını yazmaktan ziyade problemleri mümkün olduğunda basit tutarak okuyucunun assembly dili ile programlamada dikkat etmesi gereken noktalara odaklanmasını sağlamaktır.

Her ne kadar kitap titiz bir çalışma ile hazırlanmış olsa da birinci basımda gözden kaçmış bazı yazım hataları ve cümle düşüklükleri olmuştur. İkinci basımda bu hatalar mümkün olduğunca düzeltilmiş, bazı bölümler okuyucunun daha rahat anlayacağı şekilde yeniden kaleme alınmış ve ek bilgiler ile anlatım zenginleştirilmeye çalışılmıştır. Kitabın ilk basımından günümüze yaşanan teknolojik gelişmelere bağlı olarak 3 ve 4. Bölümdeki bilgiler güncellenmiştir. 5. Bölümde öğrencilerin uygulamada zorlandığı çevrimler konusu yeniden ele alınmış ve yüksek seviyeli programlama dillerinde kullandığımız While ve Repeat-Until gibi koşullu çevrimler akış diyagramları ve farklı kod örnekleri ile daha detaylı şekilde açıklanmıştır. 8. Bölümde ele alınan sözde komutlar, kullanım amaçlarına bağlı olarak yeniden sınıflandırılmıştır. Özellikle ekler bölümünde verilen çalışma ortamının kurulması ile ilgili bilgiler detaylandırılmış, mevcut işletim sistemini, sanallaştırma teknolojilerini ve emülatörleri kullanarak çalışma ortamının tesis edilmesini sağlayacak yazılımlar ve bunların temin edilebileceği internet adresleri verilmiştir. İkinci basımında kitabın genelinde anlatımı kuvvetlendirmek ve ek bilgi vermek üzere dipnotların sayısı ilk basımındakinin iki katına çıkartılmış, kısaltmalar, Türkçe-İngilizce terimler dizini ve kaynaklar zenginleştirilmiştir.

Assembly dilinde programlama yaparken de genel programlama tekniği uyarınca kullanılacak olan algoritmanın akış diyagramının çizilmesi, uç nokta problemlerinin analiz edilmesi gerekir. Akış koda dönüştürülürken kullanılacak değişkenlerin belirlenmesi, koşul ifadeleri ve (koşullu/koşulsuz) çevrimlerde kullanılacak olan dallanma noktalarının iyi belirlenmesi assembly programlamada oldukça önemlidir. Programcıların bu yönteme uygun olarak çalışmasına yardımcı olabilmek amacıyla ekler bölümünde verilen ilk örnek bu şekilde düzenlenmiştir. Ekler bölümünde dizi işlemleri, yığın üzerinden parametre aktarma ve harici yordam kullanımı konularında gerek COM gerek ise EXE tipinde farklı örneklere yer verilmiştir. Ayrıca kullanıcının giriş/çıkış birimlerini kullanarak bir toplama işlemi yapmasını sağlayan örnek ile yüksek seviyeli dillerde sahip olunan okuma/yazma (read/write) tarzı işlemlerin assembly dilinde kesmelerin de yardımı ile ne şekilde yapılabileceği konusunda basit de olsa bir fikir verilmeye çalışılmıştır.

İçindekiler

Teşekkür **5**

Önsöz **7**

2. Baskının Önsözü **9**

Şekil Listesi **23**

Tablo Listesi **25**

Kısaltmalar Dizini **29**

1. ASSEMBLY DİLİNİN YERİ **33**

1.1 Assembly Dili Hakkındaki Yanlışlar ve Doğrular 34

1.1.1 Assembly Dili Hakkındaki Yanlışlar 34

1.1.2 Assembly Dili Hakkındaki Doğrular 36

1.1.3 Neden 80x86 Assembly Dili? 36

2. SAYI SİSTEMLERİ **37**

2.1 İkili (Binary) Sayı Düzeni 37

2.2 Sekizli (Octal) Sayı Düzeni 40

2.3 On Altılı (Hexadecimal) Sayı Düzeni 40

3. SİSTEM ORGANİZASYONU **43**

3.1 FSB (Front Side Bus) 44

3.2 Bellek Bağlantı Yolları 44

3.2.1 SDRAM (Synchronous Dynamic Random Access Memory) 45

3.2.2 DDR-SDRAM (Dual Data Rate SDRAM) 46

3.2.3 DDR2-SDRAM (Dual Data Rate 2 – SDRAM) 47

3.2.1 DDR3-SDRAM (Dual Data Rate 3 – SDRAM) 48

3.2.2 DDR4-SDRAM (Dual Data Rate 4 – SDRAM) 49

3.2.3 RDRAM (Rambus Dynamic Random Access Memory) 50

3.3 Giriş/Çıkış Yolları 50

3.3.1 Yüksek Hızlı Giriş/Çıkış Yolları 51

3.3.1.1 Grafik Birim Bağlantısı 51

3.3.1.2 İkincil Bellek Birim Bağlantısı ... 52
3.3.1.3 Diğer Çevre Birim Bağlantıları ... 54
3.3.1.4 IEEE 1394 ... 54
3.3.1.5 PCMCIA (Personal Computer Memory Card International Association) ... 55
3.3.1.6 USB (Universial Serial Bus) ... 55
3.3.2 Düşük Hızlı Giriş/Çıkış Yolları ... 56
3.3.2.1 Paralel (Eşanlı) İskele ... 56
3.3.2.2 Seri (Dizisel) İskele ... 56

4. GENEL BİLGİLER VE 80X86 AİLESİ ... 57
4.1 Ana İşlemciler ... 57
4.2 Matematik İşlemciler ... 62
4.3 Kişisel Bilgisayarlarda Bellek Kullanımı ve Büyüklüğü ... 67
4.3.1 Genişletilmiş (Expanded) Bellek ... 68
4.3.2 Uzatılmış (Extended) Bellek ... 68
4.4 8086 İşlemcisinin İçyapısı ... 70
4.5 Kesim (Segment) Yapısı ... 70
4.5.1 Kod (Code) Kesimi ... 71
4.5.2 Yığın (Stack) Kesimi ... 72
4.5.3 Veri (Data) Kesimi ... 73
4.5.4 Ek (Extra) Kesim ... 73
4.6 Yazmaç (Register) ... 73
4.6.1 Veri Grubu Yazmaçları ... 73
4.6.1.1 AX: Accumulator ... 74
4.6.1.2 BX: Base Register ... 74
4.6.1.3 CX: Count Register ... 74
4.6.1.4 DX: Data Register ... 75
4.6.2 İşaret Grubu Yazmaçları ... 75
4.6.2.1 SP: Stack Pointer ... 75
4.6.2.2 BP: Base Pointer ... 75
4.6.2.3 IP: Instruction Pointer ... 75
4.6.3 Sıra Grubu Yazmaçları ... 76
4.6.3.1 SI: Source Index ... 76
4.6.3.2 DI: Destination Index ... 76
4.6.4 Kesim Grubu Yazmaçları ... 76
4.7 Bayraklar (Flags) ... 77
4.7.1 CF: Carry Flag ... 77
4.7.2 PF: Parity Flag ... 77
4.7.3 AF: Auxilary Carry Flag ... 77
4.7.4 ZF: Zero Flag ... 77
4.7.5 SF: Sign Flag ... 77

4.7.6 TF: Trap Flag ... 78
4.7.7 IF: Interrupt Flag ... 78
4.7.8 DF: Direction Flag ... 78
4.7.9 OF: Overflow Flag ... 78
4.8 16 / 32 Bitlik İşlemciler Arasındaki Farklılıklar ... 78
4.8.1 32 Bitlik İşlemcideki Diğer Bayraklar ... 80
4.8.1.1 AC: Alignment Check ... 80
4.8.1.2 VM: Virtual 8086 Mode ... 80
4.8.1.3 RF: Resume Flag ... 80
4.8.1.4 NT: Nested Task ... 80
4.8.1.5 IOPL: I/O - Privilege Level ... 80
4.8.2 Kontrol Yazmaçları ... 81
4.8.2.1 CR0: Control Register 0 ... 81
4.8.2.2 CR1: Control Register 1 ... 82
4.8.2.3 CR2: Control Register 2 ... 82
4.8.2.4 CR3: Control Register 3 ... 82
4.8.2.5 Debug ve Test Yazmaçları ... 82
4.9 64 Bit İşlemci Yazmaçları ... 82
4.10 Belleğe Erişim ... 83
4.10.1 Gerçek Kipte Fiziksel Adres Hesabı ... 84
4.10.2 Korumalı Kipte Fiziksel Adres Hesabı ... 85
4.10.2.1 16 Bit Korumalı Kip ... 85
4.10.2.2 32 Bit Korumalı Kip ... 85
4.10.2.3 Sanal (Virtual) 8086 Kipi ... 86
4.11 Çalışma Kipleri Arasındaki Farklar ... 86
4.12 İşletim Sistemleri Açısında Çalışma Kipleri ... 87
4.13 İşlemcinin Komutları Adım Adım Çalıştırması ... 87

5. 80X86 KOMUTLARI (MNEMONIC) VE KOMUT YAPISI ... 89
5.1 Veri Aktarım Komutları ... 90
MOV ... 91
MOVSX ... 92
MOVZX ... 92
LEA ... 93
LDS ... 93
LES ... 94
XCHG ... 94
XLAT / XLATB ... 95
5.2 Aritmetik Komutlar ... 95
ADD ... 95

ADC 96
XADD 96
SUB 96
SBB 97
INC 97
DEC 97
NEG 98
CMP 98
CMPXCHG 99
MUL 99
IMUL 100
DIV 100
IDIV 101
5.3 Dallanma Komutları 101
5.3.1 Koşullu Dallanma Komutları 101
5.3.1.1 Basit Koşullu Dallanma Komutları 102
JZ / JE 102
JNZ / JNE 102
JS 102
JNS 103
JO 103
JNO 103
JP / JPE 103
JNP / JPO 103
5.3.1.2 İşaretsiz Sayı İşlemlerine Ait Koşullu Dallanma Komutları 103
JB / JNAE / JC 104
JA / JNBE 104
JAE / JNB / JNC 104
JBE / JNA 104
5.3.1.3 İşaretli Sayı İşlemlerine Ait Koşullu Dallanma Komutları 105
JL / JNGE 105
JNL / JGE 105
JLE / JNG 105
JG / JNLE 105
5.3.2 Koşulsuz Dallanma Komutları 106
JMP 106
JMP FAR PTR 107
5.3.3 Çağırma Komutları 108
CALL 109
CALL FAR PTR 109
INT 110
INTO 110
5.3.4 Dönüş Komutları 110

RET 111
RETF 111
IRET 112
5.4 Döngü Komutları 112
LOOP 112
LOOPZ / LOOPE 113
LOOPNZ / LOOPNE 113
JCXZ 113
5.5 Bayraklar ile İlgili Komutlar 119
CLC 119
CMC 119
STC 120
CLD 120
STD 120
STI 120
CLI 121
LAHF 121
SAHF 121
5.6 Mantıksal Komutlar 121
NOT 122
OR 122
AND 122
XOR 123
TEST 123
SETxxx 124
BT 124
BTC 125
BTR 125
BTS 126
5.7 Öteleme ve Döndürme Komutları 126
SHL 127
SHLD 127
SAL 128
SHR 128
SHRD 128
SAR 129
RCL 129
RCR 130
ROL 130
ROR 131

5.8 Dizgi (String) Komutları 131
MOVSB 132
MOVSW 132
MOVSD 133
CMPSB 133
CMPSW 133
CMPSD 134
SCASB 134
SCASW 135
SCASD 135
LODSB 135
LODSW 136
LODSD 136
STOSB 136
STOSW 137
STOSD 137
CBW 137
CWD 138
CWDE 138
CDQ 138
BSWAP 138
5.9 Ön Ekler 139
REP 139
REPE / REPZ 139
REPNE / REPNZ 140
LOCK 140
5.10 Yığın Komutları 140
POP 141
POPA 141
POPAD 142
POPF 142
POPFD 142
PUSH 143
PUSHA 143
PUSHAD 144
PUSHF 144
PUSHFD 144
5.11 Giriş/Çıkış Komutları 145
IN 145

INSB 145
INSW 146
INSD 146
OUT 146
OUTSB 147
OUTSW 147
OUTSD 147
5.12 Durdurma ve Bekletme Komutları 147
HLT 148
NOP 148
WAIT 148
5.13 BCD Düzenleme İçin Kullanılan Komutlar 148
AAA 149
AAD 150
AAM 150
AAS 151
DAA 151
DAS 152
5.14 Diğer Komutlar 152

6. 80X86 AİLESİNDE ADRESLEME KİPLERİ 153
6.1 Anlık (Immediate) Adresleme 153
6.2 Yazmaç (Register) Adresleme 153
6.3 Doğrudan (Direct) Adresleme 153
6.4 Yazmaç Dolaylı (Register Indirect) Adresleme 154
6.5 Baz Göreli (Base Relative) Adresleme 154
6.6 Doğrudan İndisli (Direct Index) Adresleme 155
6.7 Dizgi (String) Adresleme 157
6.8 İskele (Port) Adresleme 158
6.8.1 Doğrudan İskele Adresleme 158
6.8.2 Dolaylı İskeleme Adresleme 158

7. ASSEMBLY DİLİNDE PROGRAM YAZMAK İÇİN NELER GEREKLİDİR? 159

8. SÖZDE (PSEUDO) KOMUTLAR/İŞLEMLER 163
8.1 Genel Amaçlı Sözde Komutlar 163
8.1.1 Dosya Düzenlemede Kullanılan Sözde Komutlar 163
PAGE 163

TITLE ... 163
8.1.2 Kesim Düzenlemede Kullanılan Sözde Komutlar ... 164
SEGMENT / ENDS ... 164
ORG (Origine) ... 165
ASSUME ... 165
8.1.3 Veri Tanımlamada Kullanılan Sözde Komutlar ... 166
DB (Define Byte) ... 166
DW (Define Word) ... 166
DD (Define Doubleword) ... 167
DQ (Define Quadword) ... 167
DT (Define Ten Bytes) ... 167
EQU (Equal) ... 167
DUP (Duplication Factor) ... 168
REAL4, REAL8, REAL10 ... 168
TYPEDEF ... 168
PTR (Pointer) ... 169
LABEL ... 169
8.1.4 Yordam Düzenlemede Kullanılan Sözde Komutlar ... 169
PROC (Procedure) / ENDP (End of procedure) ... 170
EXTRN ... 170
PUBLIC ... 171
8.1.5 Makro Düzenlemede Kullanılan Sözde Komutlar ... 171
MACRO (Macro) / ENDM (End of macro) ... 171
INCLUDE ... 171
LOCAL ... 172
EXITM ... 172
8.1.6 Diğer Sözde Komutlar ... 172
LENGTH ... 172
TYPE ... 173
SIZE ... 173
OFFSET ... 173
SEG (Segment) ... 173
REPT (Repetition) ... 174
IRP (Indefinite Repeat) ... 174
IRPC (Indefinite Repeat Character) ... 174
END ... 174
8.2 Koşullu Sözde Komutlar ... 175
IF ... 175
IFE ... 175
IF1 ... 175
IF2 ... 175
IFDEF ... 175
IFNDEF ... 176

IFB 176
IFNB 176
IFIDN 176
IFDIF 176

9. ASSEMBLY PROGRAM TİPLERİ 177
9.1 EXE Tipindeki Assembly Programların Genel Özellikleri 177
9.1.1 Örnek EXE Programlar 180
9.2 COM Yapısındaki Assembly Programların Genel Özellikleri 191
9.2.1 Örnek COM Programlar 194
9.3 COM ve EXE Programlar Arasındaki Farklılıklar 200

10. ASSEMBLY PROGRAMIN TEMEL TAŞLARI: YORDAM VE MAKRO 203
10.1 Yordam Özellikleri 203
10.1.1 Kesim İçi Yordam Kullanımı 203
10.1.2 Kesimler Arası Yordam Kullanımı 209
10.2 Makro Özellikleri 218
10.2.1 Makro Örnekleri 221
10.3 Yordam ve Makro Karşılaştırılması 226

11. ANA VE ALT PROGRAMLARDAKİ KESİMLERİN BİRLEŞTİRİLMESİ 227
11.1 Birleştirilmiş (Combined) Kod Kesimi Kullanımı 227
11.2 Birleştirilmiş (Combined) Veri Kesimi Kullanımı 237

12. YORDAMLAR VE PARAMETRE AKTARMA YÖNTEMLERİ 247
12.1 Yazmaç Üzerinden Parametre Aktarma 247
12.2 Yığın Üzerinden Parametre Aktarma 247
12.3 EXTRN/PUBLIC Kullanımı ile Parametre Aktarımı 253

13. SİSTEMİN ÇALIŞMAYA BAŞLAMASI (START UP) 257
13.1 DOS Bellek Haritası 259

14. KESME (INTERRUPT) NEDİR? NE AMAÇLA KULLANILIR? 261
14.1 Kesme Çeşitleri 264
14.1.1 Dâhili Kesmeler 264
14.1.2 Harici Kesmeler 264
14.1.2.1 Maskelenemez Kesmeler (NMI) 265
14.1.2.2 Maskelenebilir Kesmeler (INTR) 265

14.1.3 Kesme Öncelikleri ... 265
14.1.4 Kesme Oluştuğunda Yapılan İşlemler ... 268
14.2 Vektör Tablosunun Görevi ve Komumu ... 269
14.2.1 Vektör Tablosunda Değişiklik Neden ve Nasıl Yapılır? ... 269
14.2.2 Interrupt Servis Programı Yazarken Nelere Dikkat Edilmelidir? ... 271
14.2.3 Vektöre Tablosunun Eski Haline Getirilmesi ... 272
14.3 DOS ve BIOS Ara Yüzleri ... 272
14.3.1 BIOS (Basic Input Output System) Kesme Fonksiyonları ... 273
14.3.2 DOS Kesme Fonksiyonları ... 274

15. YÜKSEK SEVİYELİ DİLLER VE ASSEMBLY BAĞLANTISI ... 275
15.1 Turbo Pascal'dan Assembly Yordam Çağırma ... 275
15.2 Assembly'den Turbo Pascal Yordam Çağırma ... 278
15.3 Turbo Pascal ve Assembly Kullanımında Dikkat Edilmesi Gerekenler ... 282
15.4 Turbo C'den Assembly Yordam Çağırılması ... 283
15.5 Assembly'den Turbo C Yordam Çağırma ... 284
15.6 Turbo C ve Assembly Kullanımında Dikkat Edilmesi Gerekenler ... 286

16. EKLER ... 287
16.1 Çalışma Ortamının Kurulması ... 287
16.1.1 Mevcut İşletim Sistemini Kullanmadan Çalışma Ortamının Kurulması ... 288
16.1.2 Microsoft İşletim Sistemlerinde Çalışma Ortamının Oluşturulması ... 289
16.1.2.1 Windows 9x ... 289
16.1.2.2 Windows XP ... 290
16.1.2.3 Windows 7 ... 290
16.1.3 Linux İçin Çalışma Ortamını Oluşturma ... 291
16.1.4 Emülatör Kullanımı ... 292
16.2 Debug Komutları ... 292
16.2.1 R (Register-Yazmaç) ... 294
16.2.2 D (Dump / Display – Döküm / Görüntüleme) ... 295
16.2.3 A (Assembly) ... 296
16.2.4 T (Trace / İzleme) ... 296
16.2.5 U (Unassembly) ... 297
16.2.6 E (Edit / Düzenle) ... 298
16.2.7 F (Fill / Doldurma) ... 298
16.2.8 C (Compare / Karşılaştırma) ... 299
16.2.9 G (Go / Git) ... 300
16.2.10 H (Hex Calculator / On Altılı Hesaplayıcı) ... 301
16.2.11 L (Load / Yükle) ... 301

16.2.12 M (Move / Aktar) ... 301
16.2.13 P (Proceed / Devam Etmek) ... 301
16.2.14 Q (Quit / Çıkış) ... 302
16.3 ASCII Karakter Tablosu ... 302
16.4 BIOS Kesme Listesi ... 303
16.5 DOS Kesme Listesi ... 307
16.6 Ek Örnek Kodlar ... 311
16.6.1 Sıralı Dizide Arama ... 311
16.6.2 Dizi İşlemleri ... 313
16.6.3 Dizi Elemanları Üzerinden Bit İşlemleri ... 314
16.6.4 Karekök Hesabı – SQROOT (Harici Yordam) ... 315
16.6.5 Harici Karekök (SQROOT) Yordamını Kullanmak ... 316
16.6.6 Klavyeden Okuma, Hesaplama ve Ekrana Yazdırma İşlemleri ... 317

Kaynaklar ... 321

Türkçe İngilizce Terimler Dizini ... 323

Şekil Listesi

Şekil 3-1: Von Neuman mimari modeli....43
Şekil 3-2: SDRAM a) 168 bacaklı DIMM b) 144 bacaklı SO-DIMM....46
Şekil 3-3: DDR-SDRAM....47
Şekil 3-4: DDR2-SDRAM....48
Şekil 3-5: a) DDR3-SDRAM b) DDR/DDR2/DDR3 Fiziksel karşılaştırma....49
Şekil 3-6: DDR4-SDRAM....50
Şekil 4-1: PC Bellek haritası....68
Şekil 4-2: Genişletilmiş bellek yapısı....69
Şekil 4-3: Uzatılmış bellek haritası....69
Şekil 4-4: 8086 işlemcisinin basitleştirilmiş içyapısı....70
Şekil 4-5: Korumalı kipte fiziksel adres hesabı....86
Şekil 6-1: Örnek 6-3 için bellek haritası....153
Şekil 6-2: Örnek 6-4 için bellek haritası....154
Şekil 6-3: Örnek 6-5 için bellek haritası....154
Şekil 6-4: Örnek 6-6 için bellek haritası....155
Şekil 6-5: Örnek 6-9 için bellek haritası....157
Şekil 7-1: Assembly programın derlenmesi ve çalıştırılma aşamaları....161
Şekil 9-1: COM yapısındaki programının bellekteki yerleşimi....191
Şekil 9-2: COM yapısındaki programın kesim organizasyonu....192
Şekil 10-1: Kesim içindeki yordamın çağırılması (intrasegment call)....204
Şekil 10-2: Yordam kullanılırken yığının durumu....205
Şekil 10-3: Aynı yordamın kod içinde birden fazla yerden çağırılması....206
Şekil 10-4: Makronun program içinde kullanılması....220
Şekil 10-5: Makronun program içinde birden fazla yerde kullanılması....220
Şekil 14-1: 8259A için maske değerinin belirlenmesi....267
Şekil 14-2: Katman Model....273
Şekil 16-1 Sıralı dizide arama akış diyagramı....311

Tablo Listesi

Tablo 1-1: Donanım ve yazılım katmanları 34
Tablo 2-1: İkili sayı düzeninde basamakların ondalık karşılığı 37
Tablo 2-2: İkili sayı düzenindeki ifadelerin onluk sayı düzenindeki karşılığının hesaplanması 37
Tablo 2-3: İkili sayı düzeninde toplama kuralları 38
Tablo 2-4: İkili sayı düzeninde çıkarma kuralları 38
Tablo 2-5: On / iki / sekiz / on altı tabanındaki sayıların dönüşüm tablosu 41
Tablo 2-6: On, iki ve on altı tabanında sayı örnekleri 41
Tablo 3-1: SDRAM tipleri 46
Tablo 3-2: DDR-SDRAM tipleri 47
Tablo 3-3: DDR2-SDRAM tipleri 48
Tablo 3-4: DDR3-SDRAM tipleri 49
Tablo 3-5: DDR4-SDRAM tipleri 50
Tablo 3-6: RDRAM tipleri 51
Tablo 3-7: AGP tipleri 52
Tablo 3-8: PCIe tipleri 52
Tablo 3-9: EIDE tipleri 53
Tablo 3-10: SCSI tipleri 53
Tablo 3-11: Kişisel bilgisayarlarda kullanılan bağlantı yolları 54
Tablo 3-12: IEEE 1394 tipleri 55
Tablo 3-13: PCMCIA tipleri 55
Tablo 3-14: USB tipleri 56
Tablo 4-1: Intel ailesi işlemciler 57
Tablo 4-2: Matematik İşlemciler 62
Tablo 4-3: Matematik işlemcilerin başarımları - birim zamanda yapılan işlem sayısı 63
Tablo 4-4: 80x86 ailesinin karşılaştırmalı tablosu 64
Tablo 4-5: 8086 ailesi yazmaçları 74
Tablo 4-6: Kesim / Göreli konum ikilileri 76
Tablo 4-7: 8086 ailesi bayrakları (PSW) 77
Tablo 4-8: 32 Bit işlemci – Korumalı kip yazmaçları 79
Tablo 4-9: 32 Bit işlemci bayrakları 80
Tablo 4-10: Control Register - 0 81
Tablo 4-11: 64 Bit çalışmada yazmaç kullanımı 83
Tablo 4-12: Farklı çalışma kiplerinde kesimli bellek erişimi için değerler/sınırlar 84
Tablo 4-13: Gerçek ve korumalı kip karşılaştırması 86
Tablo 4-14: İşletim sistemlerine göre işlemcinin çalışma kipleri 87

Tablo 5-1: Komut kümesinde kullanılan kısaltmalar 90
Tablo 5-2: If-then-else yapısının assembly kod eşdeğeri 107
Tablo 5-3: Null else yapısının assembly eşdeğeri 108
Tablo 5-4: Null then yapısının assembly eşdeğeri 108
Tablo 5-5: For-To yapısı 114
Tablo 5-6: For –Downto yapısı 115
Tablo 5-7: While yapısı 116
Tablo 5-8: Repeat-Until yapısı 117
Tablo 5-9: İç içe çevrimli yapısının assembly eşdeğeri (kabarcık sıralama) 118
Tablo 7-1: Assembly programın derlenme ve çalıştırılma aşamaları 162
Tablo 9-1: EXE tipindeki programın iskelet yapısı 178
Tablo 9-2: COM yapısındaki programın iskelet yapısı – Model 1 193
Tablo 9-3: COM yapısındaki programın iskelet yapısı – Model 2 194
Tablo 9-4: COM ve EXE programların karşılaştırılması 201
Tablo 10-1: Yordam tanımı 203
Tablo 10-2: Örnek 10-1'de yazılan programın adım adım çalıştırılması 208
Tablo 10-3: Örnek 10-2. EXE programın adım adım çalıştırılması 217
Tablo 10-4: Makro Tanımı 219
Tablo 10-5: Yordam ve macro karşılaştırması 226
Tablo 11-1: Birleştirilmiş (combined) kod kesimi – ANA.ASM 228
Tablo 11-2: Birleştirilmiş (combined) kod kesimi – ALT.ASM 228
Tablo 11-3: Birleştirilmiş kesimlerin yerleştiği bellek adresleri 235
Tablo 11-4: Örnek programın adım adım çalıştırılması (Örnek 11-1) 236
Tablo 11-5: Birleştirilmiş (combined) veri kesimi – BIR.ASM 238
Tablo 11-6: Birleştirilmiş (combined) veri kesimi – IKI.ASM 238
Tablo 11-7: Örnek 11-2'deki programın adım adım çalıştırılması 243
Tablo 12-1: Yığın üzerinden parametre aktarımı – ILK.ASM 249
Tablo 12-2: Yığının Durumu - Altprg çalışmadan önce 249
Tablo 12-3: Yığın üzerinden parametre aktarımı – IKI.ASM 250
Tablo 12-4: Yığının durumu – PUSH BP yapıldıktan sonra 251
Tablo 12-5: Yığın durumu - Altprg'den dönüldükten sonra 253
Tablo 12-6: EXTRN/PUBLIC kullanımı ile parametre aktarımı - veri içeren program 253
Tablo 12-7: EXTRN/PUBLIC kullanımı ile parametre aktarımı – veri kullanan program 254
Tablo 13-1: DOS bellek haritası 260
Tablo 13-2: DOS Bellek adresleri 260
Tablo 14-1: Kesme öncelikleri 266
Tablo 14-2: XT'ler için IRQ ve kontrol ettikleri dış bağlantılar 266
Tablo 14-3: AT'ler için IRQ ve kontrol ettikleri dış bağlantılar 267

Tablo 15-1: Pascal program içinde assembly yordam çağırma....276
Tablo 15-2: Pascal tarafından çağırılan assembly yordam....276
Tablo 15-3: Pascal-Assembly kullanımında yığının durumu....277
Tablo 15-4: Assembly tarafından çağırılan Pascal program....278
Tablo 15-5: Pascal programı çağıran assembly kodu....280
Tablo 15-6: Turbo Pascal – Assembly program arasında yığın üzerinden parametre aktarımı....281
Tablo 15-7: C'den assembly yordam çağırma....283
Tablo 15-8: Turbo C - Assembly program arasında yığın üzerinden parametre aktarımı....283
Tablo 15-9: C'den çağırılan assembly program....284
Tablo 15-10: Assembly programdan C program çağırma....285
Tablo 15-11: Assembly – C arasında yığın üzerinden aktarılan parametreler....286
Tablo 15-12: C ve Assembly bağlantısında dikkat edilmesi gerekenler....286
Tablo 16-1 Temel yazılım listesi....288
Tablo 16-2: Debug komutları ve basit açıklamaları....293
Tablo 16-3: Debug (-R) yazmaçlar ve bayrakların görüntülenmesi....294
Tablo 16-4: Debug (-R AX) yazmaç değerinin değiştirilmesi....294
Tablo 16-5: Debug (-R F) bayrak değerlerinin değiştirilmesi....294
Tablo 16-6: Debuger'da kullanılan bayrak kısaltmaları....295
Tablo 16-7: Debug (-D DS:0) veri kesiminin içeriği....295
Tablo 16-8: Debug (-D DS:0110, 0115) bellek içeriği....295
Tablo 16-9: Debug (-A 100) assembly program yazmak....296
Tablo 16-10: Debug (-T) komutların adım adım çalıştırılması....297
Tablo 16-11: Debug (-T 5 / -T=0200,5) belirlenen sayıda komutun çalıştırılma....297
Tablo 16-12: Debug (-U CS:0112) nesne kodlarının anımsatıcı karşılıkları....298
Tablo 16-13: Debug (-U CS:0100L8) istenen uzunlukta nesne kodun anımsatıcı karşılığı....298
Tablo 16-14: Debug (-E DS:10) bellek alanında değişiklik....298
Tablo 16-15: Debug (-F DS:0100, 0120, AB) bellek alanının doldurulması....299
Tablo 16-16: Debug (-F DS:0200L8,AB) bellek alanının doldurulması....299
Tablo 16-17: Debug (-C 0100, 0110, 10) bellek içeriklerinin karşılaştırılması....299
Tablo 16-18: Debug (-C 0100, 0100, 100) bellek içerik karşılaştırılması....300
Tablo 16-19: Debug (-C DS:0100, 0110, CS:0300) bellek içerik karşılaştırılması....300
Tablo 16-20: Debug (-G) programın sonlandırılması....300
Tablo 16-21: Debug (-H) on altılı tabanda hesaplayıcı....301
Tablo 16-22: Debug (-M DS:0100, 0110, CS:0500) bellek alanına aktarım....301
Tablo 16-23: Debug (-Q) programın sonlandırılması....302
Tablo 16-24: ASCII karakter tablosu....302
Tablo 16-25: BIOS kesme listesi....303
Tablo 16-26: DOS kesme listesi....307

Kısaltmalar Dizini

AGP	**:** Advanced Graphic Port
ALU	**:** Arithmetic Logic Unit
AMD	**:** Advanced Micro Devices Inc.
ANSI	**:** American National Standard Institute
ASCII	**:** American Standard Code for Information Interchange
ATA	**:** Advanced Technology Attachment
ATAPI	**:** ATA Packet Interface
B	**:** Binary
BCD	**:** Binary Coded Decimal
BIOS	**:** Basic Input Output System
Bit	**:** Binary Digit
CD	**:** Compact Disc
CD-R	**:** CD Recordable
CD-RW	**:** CD – ReWritable
CISC	**:** Complex Instruction Set Computers
CP/M	**:** Control Program/Monitor - Control Program for Microcomputers
CPU	**:** Central Processing Unit
DDR	**:** Dual Data Rate
DIMM	**:** Dual Inline Memory Module
DIP	**:** Dual Inline Package
DMA	**:** Direct Memory Access
DOS	**:** Disk Operating System
DRAM	**:** Dynamic RAM
DVD	**:** Digital Versitile Disc / Digital Video Disc
EA	**:** Effective Address
EBCDIC	**:** Extended Binary Coded Decimal Interchange Code
ECP	**:** Enhanced Capability Port
EIA	: Electronic Industries Alliance
EIDE	**:** Enhanced IDE

EISA : Extended ISA
EMM : Expanded Memory Management
EPP : Enhanced Paralel Port
eSATA : External SATA
FC : Fiber Channel
FIFO : Firts In First Out
FLOPS : FLoating point Operation Per Second
FO : Fiber Optic
FPU : Floating Point Unit
FSB : Front Side Bus
GB : Giga Byte
GDT : Global Descriptor Table
GNU : GNU's Not UNIX
GP : General Purpose
GPL : General Public License
GPU : Graphics Processing Unit
H : Hexadecimal
HT : Hyper Threading
HW : Hardware
I/O : Input/Output
IBM : International Business Machines Corporation
IC : Integrated Circuit
IDE : Integrated Drive Electronics
IDT : Interrupt Descriptor Table
ISA : Industry Standard Architecture
JEDEC : Joint Electron Devices Engineering Council
KB : Kilo Byte
L1 : Level 1 (Cahce)
L2 : Level 2 (Cache)
L3 : Level 3 (Cache)
LDT : Local Descriptor Table

LIFO : Last In First Out

LSB : Least Significant Bit

MB : Mega Byte

MCA : Micro Channel Architecture

MIPS : Million Instruction Per Second

MMF : Multi Mode Fiber

MMX : MultiMedia eXtention

MSB : Most Significant Bit

NAS : Network Attached Storage

O : Octal

PATA : Parallel ATA

PC : Personal Computer

PCB : Printed Circuit Board

PCI : Peripheral Component Interconnect

PCIe : PCI Express

PCIx : PCI X

PCMCIA : PC Memory Card International Association

POST : Power On Self Test

PROM : Programmable ROM

PSP : Program Segment Prefix

PSW : Program Status Word

RAID : Redundant Array of Independent Disk

RAM : Random Access Memory

RDRAM : Rambus DRAM

RIMM : Rambus Inline Memory Module

RISC : Reduced Instruction Set Computers

ROM : Read Only Memory

SAN : Storage Area Network

SAS : Serial Attached SCSI

SATA : Serial ATA

SCSI : Small Computer System Interface

SDRAM : Synchronous Dynamic RAM

SDR : Single Data Rate

SIMD : Single Instruction Multiple Data

SIMM : Single In-Line Memory Module

SMF : Single Mode Fiber

SO-DIMM : Small Outline DIMM

SRAM : Static RAM

SSA : Serial Storage Architecture

SSE : Streaming SIMD Extentions

SW : Software

TI : Texas Instruments Inc.

TP : Twisted Pair

TSR : Terminate and Stay Resident

UDMA : Ultra DMA

USB : Universal Serial Bus

VCD : Video CD

VLB : Vesa Local Bus

XMS : eXtended Memory Specifications

1. ASSEMBLY DİLİNİN YERİ

Assembly programlama dili diğer programlama dillerine göre donanıma en yakın olanıdır. Donanıma yakınlık programcıya sağladığı avantajların yanı sıra ek sorumluluklar da yüklemektedir. Zira yüksek seviyeli programlama dilleri olarak kabul ettiğimiz Pascal, FORTRAN ve C gibi dillerde bir komut ile basit bir şekilde yerine getirilebilen bir işlem, birkaç assembly komutu, hatta bazı durumlarda başlı başına bir program ile yapılabilmektedir. (Klavyeden okuma, ekrana yazma için 16.6.6. bölüme bakınız.)

Bir programın çalıştırılabilmesi, yazılan kodun, kullanılan programlama dili için geliştirilmiş derleyici (compiler) programın süzgecinden hatasız olarak geçmesi ile mümkündür. Derleyici, yazılan programın o programlama dilinin yazım kurallarına (syntax) uygunluğunu, anlamlı (semantic) olup olmadığını kontrol eder. Bu kontrol sonucunda hiçbir hata oluşmaz ise hedef donanım (target platform) üzerinde çalıştırılabilir bir nesne kodunu (object code) üretir. Bazı kaynaklarda makine kodu (machine code) olarak da ifade edilen bu nesne kodu aslında bilgisayarın temellerini oluşturan transistörlerin hangi durumlarda açık, hangi durumlarda kapalı olacağını belirleyen ve 0 ile 1'lerden oluşmuş bir bit dizisinden ibarettir.

Kodlanan programın doğru çalışıp çalışmadığını adım adım denetlemek, var ise işlem ve mantık hatalarını tespit edebilmek, kısaca programı analiz ederek istenenleri yerine getirip getirmediğini anlayabilmek için hata bulma ve analiz yapmayı sağlayan yazılımlardan[1] yararlanılmaktadır. Hata bulucu yazılımlar, kullanıcılara, işlem kodları (operation code) ve bunlarla birlikte kullanılan işlenen(ler)in (operand) oluşturduğu bit dizilerini göstermenin yanı sıra, bu bit dizilişlerinin sembolik (mnemonic) karşılıklarını göstererek programın işleyişinin insan tarafından kolay takip edilebilmesini sağlarlar. Kısaca assembly dili, kullanıcı açısından bakıldığında makine kodundan çok daha basit, rahat okunabilen ve hızla anlaşılabilen sembolik bir dil olarak karşımıza çıkar.

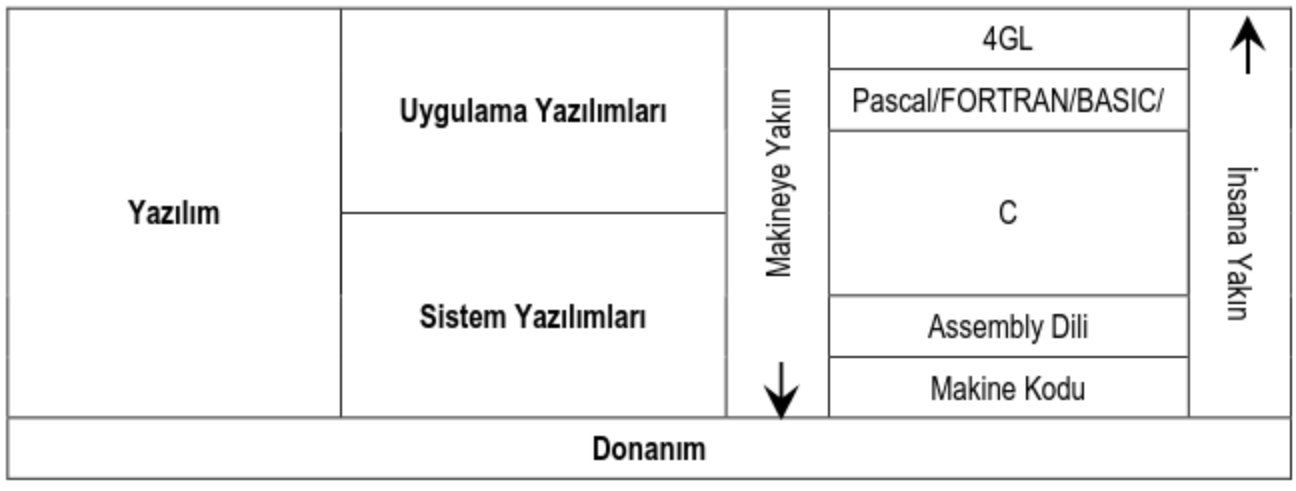
Programlama dilleri, donanıma (hardware) olan yakınlıkları açısından Tablo 1-1'deki gibi seviyelere ayrılabilirler. Üst seviyede yer alan diller insan için rahat okunur, anlaşılır ve takip edilebilir (human readable), alt seviyedekiler (yani donanıma yakın olanlar) ise makine için rahat okunabilir (machine readable) olarak sınıflandırılmaktadır.

Assembly dilinde program yazmak için kullanılacak işlemci ailesi için üretilmiş komut kümesinden (instruction set) yararlanılır. Komut kümesi, hangi işlem kodunun ne amaçla kullanıldığını, hangi durumlarda bu işlem kodunun ne tür işlenen(ler) ile kullanılabileceğini, komutun işletilmesini takiben sonucun nerede oluşacağını veya bu işlemin neleri etkileyeceğini belirleyen en temel kaynaktır.

---

[1] Hata bulma ve analiz yapmaya yarayan yardımcı yazılımlar genel olarak Debugger olarak adlandırılırlar.

Tablo 1-1: Donanım ve yazılım katmanları



Bu özellik göz önüne alındığında assembly dilinde program yazmak isteyenlerin assembly komutlarını ezberlemek yerine, bu komutlar ile ilgili bilgileri içeren bir komut kümesinden yararlanması daha doğru olacaktır.

1.1 Assembly Dili Hakkındaki Yanlışlar ve Doğrular

Bilişim sektöründe assembly dili hakkındaki genel kanı pek olumlu değildir. Pek çok kişi ön yargılı veya yanlış bilgilenmiş olmaları nedeni ile assembly dilinde program yazmanın zor ve zaman alıcı olduğunu söyleyerek kendilerince kötülemeye çalışırlar. Oysa ortaya atılan sebepler aslında değil assembly dili, günlük hayatımızda karşımıza çıkabilecek pek çok problem için de geçerlidir. Bu nedenle bunları sadece assembly diline mal etmeye çalışmak pek doğru bir davranış sayılmaz. Randall Hyde, 1996'da yazdığı "The Art of Assembly Language"[2] eserinde bu dil hakkındaki yanlış ve doğruların neler olduğunu şu şekilde özetlemektedir.

1.1.1 Assembly Dili Hakkındaki Yanlışlar

- "*Assembly dilini öğrenmek zordur*": Bilgi sahibi olmadığımız her şey aslında belirli bir zorluk içermektedir. Bu açıdan assembly dilini öğrenmek, ilk programlama dilini öğrenmekten daha zor sayılmaz.
- "*Assembly dilini okumak ve anlamak güçtür*": Eğer okumayı bilmiyorsanız hangi programla dili olursa olsun okumak ve anlamakta zorluk çekersiniz. Hatta okunmasını ve anlaşılmasını zorlaştıracak şekilde kodlama yapmak da mümkündür.
- "*Assembly dilinde hatayı bulmak zordur*": Nasıl hata aranacağı hakkında deneyim sahibi olmayanlar için ister assembly dili olsun, ister başka bir programlama dili olsun aynı zorluklar geçerlidir.

[2] http://www.plantation-productions.com/Webster/

- "*Assembly dilinde yazılmış programların bakımı zordur*": Aynı durum diğer programla dilleri, mesela C için de geçerlidir. Programcı bu kabiliyeti zaman içinde deneyim sahibi oldukça kazanacaktır.
- "*Assembly dilini kullanarak program yazmak zordur*": Assembly dilinde kod üreten programcılar önceleri ihtiyaç duydukları her parçayı kendileri yazdıkları için program geliştirme süresi uzun olmaktaydı. Ancak günümüzde yüksek seviyeli dillerde görmeye alıştığımız bazı standart program kütüphanelerinden assembly programcıları da yararlanmaktadır. Bu sayede her seferinde tekerleği yeniden icat etmek durumunda kalmayan assembly programcıları da hızlı kod üretebilmektedirler.
- "*Assembly dilinde programlama uzun zaman almaktadır*": Programcıların bir probleme çözüm getirmek için harcadıkları zamanın sadece %30'luk kısmı kodlama için kullanılmaktadır. Yüksek seviyeli diller ile kod üretmenin daha hızı olacağı doğru olmakla birlikte assembly dili ile kodlamanın getirilerini de göz önüne almak gereklidir.
- "*Gelişen derleyici teknolojileri assembly diline olan ihtiyacı ortadan kaldırmıştır*": En iyileştirici derleyicilerin (optimizing compilers) gelişmesi ile daha iyi kod parçaları üretilebilmektedir. Ancak unutulmamalıdır ki bu derleyici teknolojisinin arkasında halen assembly dili ile üretilen kodlar ve assembly dilinden elde edinilen deneyimlerin etkisi vardır.
- "*Günümüzde bilgisayarlar o kadar hızlanmıştır ki, hız için assembly diline olan ihtiyaç ortadan kalkmıştır.*" En hızlı bilgisayarda çalışacak en hızlı programın assembly dilinde olacağı tartışılmaz bir gerçektir. Ayrıca bilgisayar biraz daha hızlı çalışsın diye ek harcama yaparak başarımını arttırmak veya bir üst modele çıkartmak yerine, programları biraz gayret ile düzenleyerek mevcut donanım üzerinde daha hızlı çalışacak şekle getirmek mümkündür.
- "*Uygulamanın hızlanması için assembly dili kullanmak yerine daha iyi bir algoritma kullanılmalıdır*": Neden aynı algoritma assembly dili kullanılarak uygulanmasın?
- "*Günümüzde bilgisayarların bellekleri arttı ve ucuzladı. Bu nedenle yerden (bellekten) kazanmak için assembly dilini kullanmanın gereği kalmadı*": Günümüzde bir daktilo olarak kullanılan bilgisayarlarda dahi yüzlerce MB bellek kullanılmakta hatta bu bile uygulamaya göre yetersiz kalmaktadır. Bu sonu gelmeyecek bir yarış halini alacaktır.
- "*Assembly dili taşınabilir (portable) değildir*": Bu tamamen doğru bir tespittir. 80x86 mimarisi üzerinde çalışan bir assembly programını 68000 mimarisi üzerinde çalıştırmak mümkün değildir. Ancak taşınabilir olduğu iddia edilen C

programlama dilinde yazılmış olan bir programın bile farklı bir türde bilgisayar sistemi üzerinde ilk seferde tam olarak çalışacağının bir garantisi yoktur.

1.1.2 Assembly Dili Hakkındaki Doğrular

- *Hız:* Assembly dilinde yazılmış programlar en hızlı programlardır. Deneyimli bir assembly programcısı, deneyimli bir C programcısının yazdığı programdan daha hızlı çalışacak kodu üretebilir. Bazı uygulamalarda bu oran 5–10 kat gibi ciddi farklara da ulaşabilmektedir.
- *Bellek üzerinde az yer tutma*: Assembly programlar bellek üzerinde en az yer tutan programlardır. Yüksek seviyeli bir dil kullanılarak yazılmış kodun yaptığı işi yapan assembly program en az onun yarısı kadar yer kaplamaktadır. Assembly programlama yerden kazanmak, kaynakları tasarruflu kullanmak demektir. Elinizdeki program kodu 1.5 MB ise 1.44 MB diskete kopyalamazsınız veya bilgisayarınızın belleğine ne kadar büyük program yüklerseniz bellekte aynı anda o kadar az uygulama tutabilirsiniz. Sonuç olarak sisteminizin genel başarımı düşer.
- *Yetenek*: Assembly dili kullanılarak yüksek seviyeli diller ile yapılması çok zor hatta imkânsız olan işleri yapmak mümkündür.
- *Bilgi birikimi*: Assembly dili kullanımı ile elde edilecek deneyimle yüksek seviyeli programlama dilleri kullanarak daha iyi programlar yazma becerisi kazanılacaktır.

1.1.3 Neden 80x86 Assembly Dili?

Her ne kadar assembly dilinin genel mantığı her işlemci ailesinde benzer olsa da, yazmaçlar (register), işlem yönü, adresleme teknikleri ve komut kümelerinde farklılıklar vardır. Günümüzde gerek masaüstü gerek ise taşınabilir bilgisayar sistemleri incelendiğinde donanım olarak Intel firmasınca üretilmiş olan, 80x86 ailesi işlemcilerin veya bu aile ile kod uyumlu olarak çalışan AMD firmasının işlemcilerin yaygın olarak kullanıldığı görülmektedir. Assembly dilinin yaygın olarak bulunan/kullanılan platform üzerinde[3] öğrenilmesinin ve kodlanmasının çok daha kolay olacağı dikkate alınarak, 80x86 ailesi işlemcilerin assembly dilinin bu kitap kapsamında kullanılması uygun görülmüştür.

[3] Kod geliştirmede kullanılan bilgisayar sisteminin işlemcisine bağlı kalmaksızın, çapraz çevirici (cross assembler) veya çapraz derleyici (cross compiler) yazılımları ile tamamen farklı bir işlemci ailesinde çalışacak programlar geliştirmek ve bunları emülatörler üzerinde çalıştırmak, test etmek veya elde edilen makine (nesne) kodunu hedef platform üzerine yükleyerek çalıştırmak mümkündür.

2. SAYI SİSTEMLERİ

Bilgisayar teknolojisinin temellerinin transistöre dayanmakta olduğundan daha önce de bahis etmiştik. Transistör çalışma prensipleri gereği iletimde (kapalı devre) veya kesimde (açık devre) olabilir. Bu nedenle transistör 1 (var) veya 0 (yok) şeklinde ifade edilebilen iki ayrık (discrete) durumu belirleyebilmektedir. Bu nedenle bilgisayarlarda ikili (binary) sayı sistemi yaygın bir kullanıma sahiptir. Ancak ikili sayı sistemi kullanılarak uzun bit dizilerini ifade etmekteki zorluklar sekizli (octal) ve on altılı (hexadecimal) gösterimlerin de kullanılmasını gerekli kılmıştır.

2.1 İkili (Binary) Sayı Düzeni

İkili sayı düzeninde 0 ve 1 rakamları kullanılır[4]. Bit[5] olarak ifade ettiğimiz hanelerin birbirini takip etmesi ile sayı oluşturulmuş olur. Peş peşe dizilen ikili sayı düzenindeki haneler sağdan sola doğru 2 sayısının artan kuvvetleri cinsinden değerleri ifade ederler. Buna göre 8 bit'ten oluşan bir byte'ın on tabanındaki karşılığı, her bit'e karşı düşen 2'nin kuvvetinin o hanede (bit) bulunan değer (0 veya 1) ile çarpılması ile elde edilen değerlerin toplanması ile hesaplanır. On tabanındakine benzer olarak en soldaki hane en anlamlı biti (MSB-Most Significant Bit) en sağdaki hane ise en az anlamlı biti (LSB-Least Signicifant Bit) ifade etmektedir. 8 bit için geçerli değerler Tablo 2-1'de gösterilmiştir.

Tablo 2-1: İkili sayı düzeninde basamakların ondalık karşılığı

MSB							LSB
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
↓	↓	↓	↓	↓	↓	↓	↓
128	64	32	16	8	4	2	1

Buna göre $(1\ 0\ 1\ 1\ 0\ 1\ 0\ 1)_2$ ikili sayısı, Tablo 2-2'den de görüldüğü gibi on tabanında 181 sayısına eşittir.

Tablo 2-2: İkili sayı düzenindeki ifadelerin onluk sayı düzenindeki karşılığının hesaplanması

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	1	0	1	0	1	B
↓	↓	↓	↓	↓	↓	↓	↓	
128	0	32	16	0	4	0	1	= 181

---

4 Sıfır sayısı ile O harfi birbirlerine olan benzerlik nedeniyle karışmakta olup, zaman zaman sıfır sayısı için ∅ sembolü kullanılmaktadır. İkili düzende yazılan sayıları belirlemek üzere **B** harfi sayının sonuna eklenmektedir.

5 Bit, Binary Digit yerine kullanılan bir kısaltmadır.

Onluk dışında bir sayı sisteminde sayıları ifade etmek gerektiğinde sayıyı parantez içine alarak alt indiste sayı sistemini gösteren rakamı kullanmak modern matematikten alışık olduğumuz bir gösterimdir. Buna göre 181 sayısı, $(1\ 0\ 1\ 1\ 0\ 1\ 0\ 1)_2$ olarak da gösterilebilir. Ancak assembly programlamada sayı sistemlerini ifade etmek üzere modern matematikte kullandığımız alt indis ile ifade yöntemini kullanamayız. Bu durumda sayı sistemini sayının sonuna ekleyeceğimiz bir harf ile belirleriz. İkili sayılar için bu harf **B** olup, 181 sayısını **1 0 1 1 0 1 0 1 B** olarak gösteririz.

İkili sayı düzeninde toplama işlemleri yapılırken dikkat edilmesi gereken kurallar Tablo 2-3'te verilmiştir. Toplamaya giren iki sayının da 1 olması durumunda toplam sonucu ikili sayı düzeni kuralları uyarınca 0 olacak ancak bir üst haneye elde aktarımı yapılacaktır.

Tablo 2-3: İkili sayı düzeninde toplama kuralları

0	1	0	1
+ 0	+ 0	+ 1	+ 1
0	1	1	1 0

Örnek 2-1

73		0	1	0	0	1	0	0	1	B
+62	+	0	0	1	1	1	1	1	0	B
135		1	0	0	0	0	1	1	1	B

Benzer şekilde ikili düzende yapılan çıkarma işlemlerinin de bazı kuralları vardır. Bu kurallar Tablo 2-4'te verilmiştir.

Tablo 2-4: İkili sayı düzeninde çıkarma kuralları

0	1	1	0	
- 0	- 0	- 1	- 1	
0	1	0	1	Borç 1

Çıkarma işlemi Tablo 2-4'teki kurallara uyarak yapılabileceği gibi, daha kolay ve tercih edilen diğer bir yöntem ise çıkartılacak sayının ikiye tümleyeni alınıp diğer sayı ile toplanmasıdır.

Örnek 2-2

67		0	1	0	0	0	0	1	1	B
-21	-	0	0	0	1	0	1	0	1	B
46		0	0	1	0	1	1	1	0	B

Aynı işlemi ikiye tümleyen metodu ile yapacak olursak;

21	=	0	0	0	1	0	1	0	1	B	
		1	1	1	0	1	0	1	0	B	Bire tümleyen
	+								1	B	
		1	1	1	0	1	0	1	1	B	İkiye tümleyen

Buna göre 67 sayısı ile 21 sayısının ikiye tümleyenini toplamakla aslında 67 sayısından 21 sayısını çıkarmış oluruz.

67			0	1	0	0	0	0	1	1	B
+(-21)	+		1	1	1	0	1	0	1	1	B
46		**1**	0	0	1	0	1	1	1	0	B

İşlemin sonucunda 9. bit'te elde oluşmuştur. Ancak byte'tın en anlamlı bit'inin 0 olması, sayının pozitif bir sayı olduğunu belirlemektedir. En anlamlı bit'ten taşan eldenin (9. bit) ise bir önemi yoktur. Buna göre sonuç 32+8+4+2 = 46 eder.

Uygulamaya bağlı olarak sayılar işareti veya işaretsiz olarak yorumlanabilir. Bu nedenle programcının sayılar ile yapılan işlemlerde sonucun işaretli olup olmayacağına bağlı olarak gerekli tedbirleri alması gereklidir.

Şimdi başka bir işlem ile sonucun negatif olması durumunu da inceleyelim.

Örnek 2-3

21		0	0	0	1	0	1	0	1	B
- 67	-	0	1	0	0	0	0	1	1	B
- 46		**1**	1	0	1	0	0	1	0	B

Sayının en anlamlı bit'inin 1 olması sayının negatif sayı olduğunu belirtmektedir. Ancak en anlamlı biti dışında kalan bitler doğrudan sayının değerini anlamamıza yetmemektedir. Eğer yukarıda verilen örnekte işaret biti dışındaki haneler değerlendirilecek olursa, 82 sayısı bulunur. Oysa bu sayı elde edilmesi gereken sonuçtan, yani 46 sayısından tamamen farklıdır. Sayının negatif olduğunu anladıktan sonra değerini bulmak için yine ikiye tümleyen mantığından yararlanırız.

	1	1	0	1	0	0	1	0	B	
	0	0	1	0	1	1	0	1	B	Bire tümleyen
+								1	B	
	0	0	1	0	1	1	1	0	B	İkiye tümleyen

Elde edilen sayının on tabanındaki karşılığı 32+8+4+2= 46 olacaktır. Ancak sayının negatif olduğunu bildiğimize göre sonuç **-46** olacaktır.

Aynı işlemleri doğrudan ikiye tümleyen mantığı ile yapacak olursak;

67		0	1	0	0	0	0	1	1	B	
		1	0	1	1	1	1	0	0	B	Bire tümleyen
	+								1	B	
- 67		**1**	0	1	1	1	1	0	1	B	İkiye tümleyen

21		0	0	0	1	0	1	0	1	B
+(-67)	-	1	0	1	1	1	1	0	1	B
- 46		**1**	1	0	1	0	0	1	0	B

2.2 Sekizli (Octal) Sayı Düzeni

Sayılar ikili sayı düzeni dışında, sekizli (octal) olarak da gösterilebilirler. Bu durumda kullanılabilecek taban 8 olduğundan rakamlar 0 – 7 ile sınırlı kalmaktadır. İkili sayı düzeninde ifade edilen sayılar sağdan başlayarak üç haneli gruplara ayrılarak yazılacak olursa, o sayının sekizli sayı düzenindeki karşılığı rahatlıkla elde edilir. Sayının sekizli düzende ifade edildiğini belirlemek üzere **O** harfi sayının sonuna ilave edilmektedir. Byte veya word (2 byte veya 16 bit) düzenindeki sayıların sekizli sayı düzeninde ifadesinde hanelerin 3'ün katları ile ifade edilememesinden kaynaklanan bazı problemler yüzünden sekizli sayı düzeni pek tercih edilmemektedir.

Örnek 2-4

1 1 0 0 0 1 **1 0 1** B sayısı aslında 615 O sayısı olarak da ifade edilebilir.

2.3 On Altılı (Hexadecimal) Sayı Düzeni

Sayıların bilgisayarda kolay ifade şekillerinden bir tanesi de on altılı sayı düzeninde gösterimidir. Bu gösterimde 0 – 9 rakamlarına ek olarak A, B, C, D, E ve F harfleri de kullanılarak sayılar ifade edilir. Eğer iki tabanındaki sayıları sağdan başlayarak dört haneli olarak bölecek olursak, o sayının on altı tabanındaki gösterimini elde ederiz. Sayının on altılı düzende ifade edildiğini belirlemek üzere **H** harfi sayının sonuna ilave edilmektedir. Sayı sistemleri ile ilgili olarak Tablo 2-5'i incelemekte fayda vardır.

Örnek 2-5

1 1 0 0 0 1 0 1 B sayısı aslında C5H sayısı olarak da ifade edilebilir.

Tablo 2-5: On / iki / sekiz / on altı tabanındaki sayıların dönüşüm tablosu

On Tabanı	İki Tabanı	Sekiz Tabanı	On altı Tabanı
0	0000B	00O	00H
1	0001B	01O	01H
2	0010B	02O	02H
3	0011B	03O	03H
4	0100B	04O	04H
5	0101B	05O	05H
6	0110B	06O	06H
7	0111B	07O	07H
8	1000B	**10O**	08H
9	1001B	**11O**	09H
10	1010B	**12O**	0AH
11	1011B	**13O**	**0BH**
12	1100B	**14O**	**0CH**
13	1101B	**15O**	**0DH**
14	1110B	**16O**	**0EH**
15	1111B	**17O**	**0FH**

Sayıların ifadesi sırasında hangi sayı sisteminin kullanıldığını belirlemek üzere sayının sonuna, ikili sayı düzeni için **B**, sekizli sayı düzeni için **O** ve on altılı sayı düzeni için **H** harfleri ilave edilmektedir[6]. Bunların dışında on altılı sayı sistemindeki sayıların harf ile başlayanlarının başına 0 rakamı yerleştirilir. Bu sayede sayı herhangi bir değişken (variable) veya etiket (label) ile karıştırılmadan kullanılabilir. Buna göre ABCDH sayısı, 0ABCDH şeklinde ifade edilmelidir. Oysa 1234H sayısı için buna gerek yoktur. Ancak sayının başına 0 konarak 01234H şeklinde ifade edilmesi bir hata değildir. Sayılar ister ikili, ister on altılı tabanda olsun genel olarak 8, 16 veya 32 bit olacak şekilde ifade edilmelidir. Buna göre Tablo 2-6'daki örnekleri vermek mümkündür.

Tablo 2-6: On, iki ve on altı tabanında sayı örnekleri

On Tabanı	İki Tabanı	On altı Tabanı
5	00000101B	05H
163	10100011B	0A3H
256	0000000100000000B	0100H
41215	1010000100100011B	0A123H

Ayrıca bilgisayarlar arası veri transferini kolaylaştırmak amacıyla harfler, sayılar, noktalama işaretleri ve diğer sembollerin ifadesinde bazı kodlama sistemleri kullanılmaktadır. Bu kodlama sistemleri 2'li, 8'li veya 16'lı sayı sistemleriyle ifade edi-

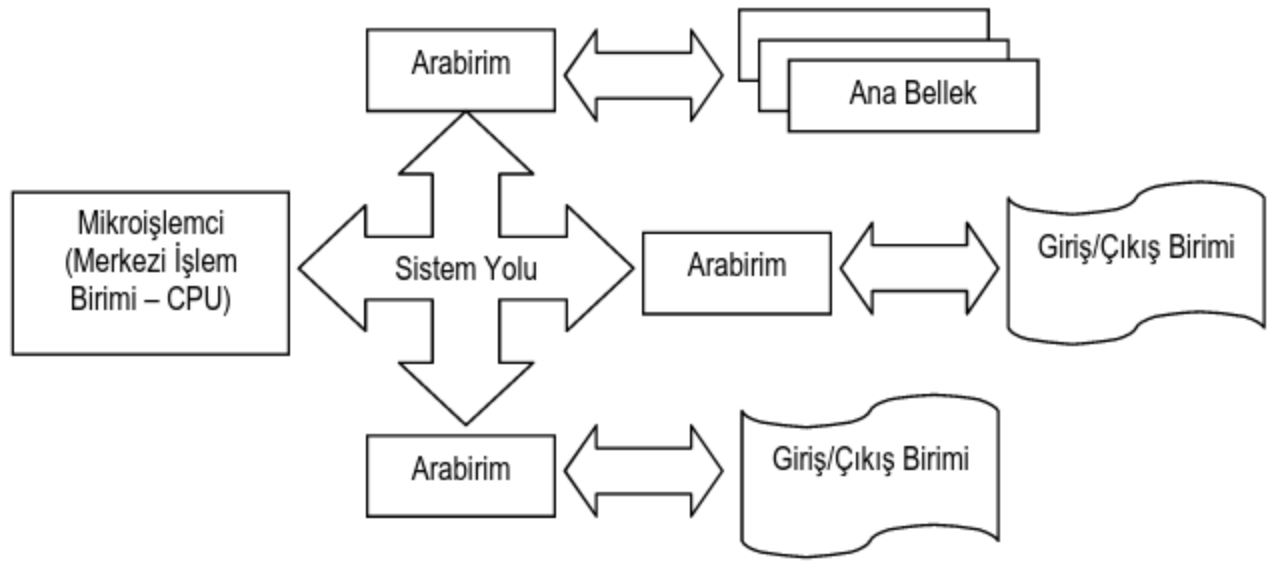
[6] Assembly dili büyük küçük harf duyarlı değildir. Bu nedenle küçük veya büyük harf kullanımı arasında fark yoktur

len değerleri kullanırlar. En yaygın olarak bilineni ASCII (American Standard Code for Information Interchange) ve EBCDIC (Extended Binary Coded Decimal Interchange Code) kodlama sistemleridir. Bu kod sistemleri 8 bit ile 256 adet farklı sembolü ifade edebilmektedirler (detaylar için Tablo 16-24'de bakınız). Mevcut kodlama sistemleri farklı dillerin alfabelerinde kullanılan her türlü sembolün aynı anda ifade etmesine olanak vermez. Bu sıkıntıyı ortadan kaldırmak amacıyla UNICODE adı verilen, aynı anda pek çok farklı sembolü barındıran kodlama sistemlerinden yararlanılmaktadır.

3. SİSTEM ORGANİZASYONU

80x86 assembly dilinde program yazabilmek için öncelikle bilgisayar sistemini oluşturan birimler ve bunlar arasındaki ilişkiler, kısaca donanım hakkında da bilgi sahibi olunması gereklidir. Programın üzerinde çalıştırılacağı mimari ve bu mimarinin mevcut işletim sistemi aracılığıyla nasıl kontrol edildiği hakkında bilgi sahibi olunursa daha rahat ve etkin assembly programlar yazılabilir. Bu nedenle öncelikle 80x86 ailesi işlemcilerin yaygın olarak kullanıldığı kişisel bilgisayar (PC) sistemlerindeki mimari model ve bazı özelliklere değinmekte fayda vardır.

Mimarı yapı, bilgisayar sistemlerinin temel tasarım özelliklerini belirlemektedir. 80x86 ailesi işlemcileri Von Neuman mimari modelini kullanmaktadırlar. Von Neuman modelinde bilgisayar sistemi, merkezi işlem birimi (CPU-Central Processing Unit), Bellek (Memory) ve Giriş/Çıkış (I/O-Input/Output) olarak adlandırılan Şekil 3-1'de ki 3 temel parçadan oluşur.



Şekil 3-1: Von Neuman mimari modeli

Merkezi işlem birimi (kısaca işlemci olarak adlandırılır), tüm aritmetik - mantıksal (arithmetic-logic) işlemlerin yapıldığı yerdir. Komutlar ve üzerinde işlem yapılacak olan veri (data) işlemci tarafından isteninceye kadar bellek üzerinde tutulmaktadır. Giriş/Çıkış birimleri ile işlemci arasındaki ilişki, işlemci ile bellek arasındaki ilişkiye benzerlik göstermektedir. Giriş/Çıkış birimleri işlemci ile dış ortamın bağlantısını sağlamakta, işlemci bu birimler yardımıyla dış dünyaya veri yollayıp alabilmektedir. Von Neuman mimari modelindeki birimlerin birbirleri ile olan bağlantıları yollar (bus) ile sağlanmaktadır. 80x86 ailesinde yollar, adres (address bus), veri (data bus) ve kontrol (control bus) olmak üzere üç alt grup halinde incelenebilir.

Bilgisayar, Von Neuman modelinde tanımlanan temel parçaların yollar ile birbirine bağlanması ile uyum içinde çalışan bir sistem olarak tarif edilir. Örneğin, işlemci bellek erişimi yapmak istediğinde erişmek istediği bellek adresini adres yoluna, bellek erişimi için gerekli bazı kontrol bilgilerini de kontrol yoluna koymalıdır. Bu

değerlerin yola konularak bellek birimine iletilmesini takiben ilgili veri de veri yoluna konulacaktır. Her ne kadar işlem basit gibi görünse de yollar üzerinde yer alan verilerin (adres, kontrol ve veri) yol üzerinde ne süre ile kalacağı (timing) ve diğer birimlerin bu yolu bu sırada nasıl kullanacakları gibi problemler göz önüne alındığında kontrol bilgilerinin ne derece önemli olduğu anlaşılacaktır.

Zaman içinde bilgisayarı oluşturan temel parçaların gelişmesi ile ortaya çıkan ihtiyaçları karşılamak üzere farklı yol tipleri de kullanılmıştır[7]. Bu yol tipleri arasındaki farklar veri genişliği (data width), dönüş oranı veya hızı (cycle rate), aygıt yönetim (device management) özellikleri ve tipleridir. Veri genişliği ve hız, o yol üzerinden aktarılabilecek veri miktarını belirlemektedir. Örneğin 8 bit genişliğinde bir yol üzerinde 1000 MHz'lik bir dönüş oranı kullanılacak olursa saniyede 8 Mbit (1 Mbyte/s) veri aktarımı yapmak mümkün olabilecektir. Bu değer o yol tipinin sağlayacağı başarımı belirlemektedir. Yönetim özellikleri ise bu yol tipinin kullanılması durumunda desteklenebilecek olan cihaz sayısı ve bunların şekillendirilmesi (configuration) ile ilgili özellikleri belirlemektedir. Tip ise yolun ne türde[8] veri aktarımı yapmak üzere tasarlandığını belirler. Buna göre bir kişisel bilgisayar sisteminde olması muhtemel yol yapıları aşağıda belirtilmiştir.

3.1 FSB (Front Side Bus)

İşlemci ile ana kart arasındaki bağlantıyı sağlayan, kullanılan teknolojiye ve tasarıma bağlı olarak, hızları 66–1600[9] MHz arasında değişen yola FSB adı verilir. Günümüzde işlemci hızlarının birkaç GHz olduğu düşünülecek olursa işlemciler ile iletişimi sağlayan bu yolların hızlarının işlemciden mümkün olan en yüksek başarımı almaktan uzak olduğu görülecektir[10]. Ancak bu zayıflık veriyi işlemciye yakın tutarak kısmen çözülebilmektedir. Bu amaçla işlemciler üzerinde ön bellek (cache memory) olarak adlandırılan bellek birimleri bulundurulmakta, işlemcinin işleyeceği komut ve veri ana bellekten komut ve veri boru hatları (pipeline) yardımıyla ön belleğe aktarılmaktadır. Kendi üzerindeki hızlı bellek birimi üzerinden komut ve veriye erişen işlemcinin başarımı da böylece artmaktadır.

3.2 Bellek Bağlantı Yolları

Bellek birimleri ile ana kart arasındaki bağlantı bellek bağlantı yuvaları (socket) ile sağlanmaktadır. Veri genişlikleri, hızları, ebatları ve bacak dizilişlerine (form

---

7 "Understanding your motherboard's bus system", James McPherson, PC Troubleshooter Report, TechProGuild TechRepublic.

8 Seri veya paralel tipte aktarım yapmak mümkündür. Seri aktarımda daha az bağlantı kullanılırken paralel olanda bağlantı sayısı fazladır.

9 Günümüze yaygın olarak 1066, 1333 hatta 1600 MHz hızlarında FSB'a sahip anakartlar kullanılmaktadır. Daha eski nesil teknoloji kullanan anakartlarda bu değerler 66-667MHz arasında değişmekte idi.

10 İşlemci hızı ile FSB hızı arasındaki oran bizim için bir sistem başarım ölçütü olabilir. Oran ne kadar küçük ise sistemin daha yüksek başarımlı olabileceğini öngörebiliriz. Örneğin işlemci hızının 2400 MHz, FSB hızının 800 MHz olduğu durumda oran 3, İşlemci hızının 2000 MHz, FSB'nin 1066 MHz olduğu durumda bu oran yaklaşık 2 olup diğerinden daha başarımlıdır diyebiliriz. Ancak genel olarak bir sistemin başarımı onu oluşturan tüm parçaların başarımı ile ölçülebilir.

factor- biçim katsayısına) bağlı olarak farklı özelliklerde bellek birimleri üretilmiştir. Önceleri DIP (Dual Inline Package) yapısındaki bağımsız bütünleşik devre elemanları teker teker ana kartlar üzerinde bulunan bağlantı yuvalarına takılırken daha sonraları bütünleşik devre elemanları bir baskılı devre (PCB-Printed Circuit Board) üzerine yerleştirilmiş modüller olarak üretilmeye başlanmıştır. Baskılı devre üzerine monte edilmiş olan bütünleşik devre elemanlarından oluşan modüller SIMM (Single Inline Memory Module) ve DIMM (Dual Inline Memory Module) olarak anılır. İlk çıkan SIMM'ler bir seferde 8 bit veri aktarabilirken, işlemcilerin veri yollarının genişlemesi ile ilerleyen dönemlerde bir seferde 32 bit veri aktarabilen SIMM'ler de üretilmiştir. Bunlar arasındaki farkı vurgulayabilmek üzere bellek birimleri, bacak sayılarına göre, 30 ve 72 bacaklı olarak adlandırılmıştır.

SIMM'lerde baskılı devrenin ön ve arka yüzünde yer alan bacaklar birbirleri ile bağlantılı iken DIMM'lerde bu bacaklar[11] birbirlerinden yalıtılmıştır. Ayrıca DIMM'ler bir seferde 64 bit veri aktarımı yapabilmelerinin yanı sıra boyutları, baskılı devre üzerinde yer alan çentiklerin yerleri ve sayıları itibariyle de benzer gibi görünen SIMM'lerden farklıdır.

Özellikle teknolojik geçişlerin yaşandığı dönemlerde üretilen bazı ana kartlarda hangi teknolojinin etkin/standart[12] olacağının tam olarak bilinememesinden ötürü birkaç farklı tipte bellek bağlantı yuvası birden kullanılmıştır. Yaygın olarak kullanılan bellek tipleri şunlardır;

3.2.1 SDRAM (Synchronous Dynamic Random Access Memory)[13]

SDRAM[14] bellekler 64 bit (8 byte) olarak organize edilmiş, DIMM yapısında, 168 bacaklı ve çift çentiklidir (Şekil 3-2). Taşınabilir bilgisayar sistemlerinde

---

[11] Masaüsütü sistemlerde kullanılan DIMM tipindeki bekllek elemanları 168, 184, 240 bacaklı iken genellikle taşınabilir bilgisayarlarda kullanılan SODIMM tipindeki bellek elemanları 72, 144, 200, 204 bacaklı olabilmektedir. Taşınabilir bilgisayarlar için 68, 144 ve 172 bacaklı micro-DIMM türünde bellek elemanları da kullanılmıştır.

[12] JEDEC (The Joint Electon Device Engineering Council), EIA (Electronic Industries Alliance)'ın yarı iletken standartlarından sorumlu oluşumu olup, bellek üreticilerinin ortak bir bellek standardına göre üretim yapmasını hedeflemektedir. SDRAM, DDR, DDR2 ve DDR3 gibi standartları kişisel bilgisayarlarda kullanılan en yaygın bellek standartlarıdır.

[13] SDR-SDRAM (Single Data Rate – SDRAM) olarak da adlandırılır.

[14] Bilgisayar sistemlerinde, yüksek depolama kapasitesini oldukça ucuza sağlaması sebebiyle DRAM (Dynamic RAM) tipinde bellek birimleri kullanılır. Bu bellek birimlerinde her bir bit çok küçük bir kondansatörün şarjlı olması/olmaması (1/0 olması) ile saklanmaktadır. DRAM tipindeki belleklerin üzerinde yer alan, sahip olunan kapasite ile sayısal olarak eşit sayıdaki kapasite elemanının zaman ile boşalmaması için belli aralıklar ile bunların tazelenmesi (şarj edilmesi) gereklidir. Bu tazeleme işlemi yaklaşık olarak 15ms de bir tekrarlanmak zorundadır. Aksi halde veri kaybı yaşanır. DRAM tipindeki bellekler genel olarak işlemciden daha yavaş çalışırlar. SRAM (Static RAM) tipindeki bellek birimleri ise DRAM de olduğu gibi belli aralıklarla tazelenme gerektirmeyen, DRAM'den çok daha hızlı ve işlemcilere yakın hızda çalışabilmektedir. Ancak SRAM'lerde her bit için 6 tane tansistör kullanılmaktadır. Bu durum DRAM ile eşdeğer kapasite sağlayacak olan SRAM'lerin gerek hacmine gerek ise üretim maliyetine etki etmektedir. Bilgisayar sistemlerinde hızlı işlemciler ile yüksek kapasiteli, ucuz ama yavaş DRAM'ler arasında hız farkı, düşük kapasiteli, neredeyse işlemci kadar hızlı, ancak pahalı olan SRAM'lerin farklı seviyelerde (L1, L2, L3 gibi) önbellek olarak kullanılarak azaltılmaya çalışılmıştır. L1 önbellek işlemci çekirdeği ile aynı hızda ve işlemci üzerinde yer almaktadır.

DIMM'lerden fiziksel olarak daha küçük olan 144 bacaklı SO-DIMM[15] (Small Outline DIMM) tipinde bellek elemanları kullanılır. Destekledikleri hıza bağlı olarak saniyede aktarabildikleri veri miktarı da artmaktadır. Bu tipteki bellek birimlerinin özellikleri Tablo 3-1'de verilmiştir.

Tablo 3-1: SDRAM tipleri

Tipi	Veri Genişliği (Byte)	Hızı (MHz)	Kapasitesi (MB/s) (Veri Genişliği x Hız)
PC66	8	66	532
PC100	8	100	800
PC133	8	133	1064

Şekil 3-2: SDRAM a) 168 bacaklı DIMM b) 144 bacaklı SO-DIMM

3.2.2 DDR-SDRAM (Dual Data Rate SDRAM)

DDR-SDRAM bellekler 64 bit (8 byte) olarak organize edilmiş, 2.5 volt ile çalışan, DIMM yapısında, 184 bacaklı ve tek çentiklidir (Şekil 3-3). SDR-SDRAM den farklı olarak çalışma frekansı (G/Ç Hızı) arttırılmadan bir çevrimde iki aktarım[16] gerçekleştirdiği için veri aktarım kapasitesi ikiye katlanmıştır. Standart olarak üretilen DDR-SDRAM'lerden[17] daha üstün başarım değerleri sunan hız aşırtma[18] (over clocking) uygulanmış DDR-SDRAM'lerde piyasada hatırı sayılır bir yer bulmuştur. DDR-SDRAM'ler ile ilgili bilgiler Tablo 3-2'de verilmiştir.

---

[15] SO-DIMM'lerden 72 bacaklı bir seferde 32 bit, 144 bacaklı bir seferde 64 bit ve 200 bacaklı bir seferde 72 bit veri aktarımı yapabilmektedir.

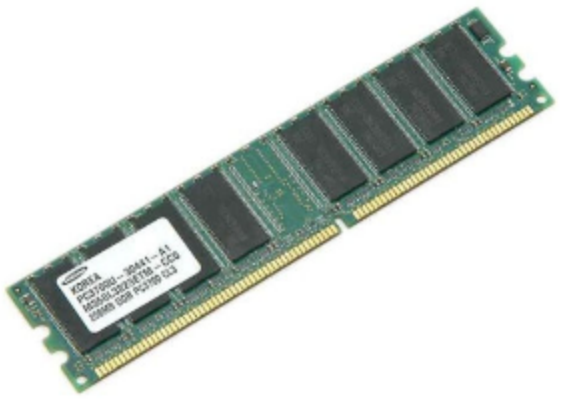
[16] Bir seferde hem inen hem de çıkan kenar ile tetiklenerek aktarım yaptığı için dual olarak adlandırılmıştır.

[17] PC1600, PC2100, PC2700 ve PC3200 JEDEC tarafından DDR-SDRAM standardı olarak belirlenmiştir.

[18] PC3500, PC3700, PC4000, PC4200, PC4400 ve PC4800 hız aşırtma uygulanmış DDR-SDRAM'ler olup, JEDEC tarafından standart kapsamına alınmamıştır.

Tablo 3-2: DDR-SDRAM tipleri[19]

Tipi	Bellek bütünleşik devresi	Veri Genişliği (Byte)	G/Ç Hızı (MHz)	Kapasitesi (GB/s) (2 x Veri Genişliği x Hız)
PC1600	DDR-200	8	100	1.600
PC2100	DDR-266	8	133	2.133
PC2700	DDR-333	8	166	2.667
PC3200	DDR-400	8	200	3.200
PC3500	DDR-433	8	216	3.466
PC3700	DDR-466	8	233	3.733
PC4000	DDR-500	8	250	4.000
PC4200	DDR-533	8	533	4.266
PC4400	DDR-550	8	550	4.400
PC4800	DDR-600	8	600	4.800



Şekil 3-3: DDR-SDRAM

3.2.3 DDR2-SDRAM (Dual Data Rate 2 – SDRAM)

DDR2-SDRAM bellekler 64 bit (8 byte) olarak organize edilmiş, 1.8 volt ile çalışan, dolayısı ile daha az enerji tüketen ve ısınan, DIMM yapısında, 240 bacaklı ve tek çentiklidir (Şekil 3-4). DDR ile benzer özellikleri göstermek ile birlikte gürültü ve girişim problemlerini önleyen sinyalleşme metodu sayesinde DDR-SDRAM'lerin resmi olarak kaldığı 400 MHz G/Ç hızlarından başlayarak 1066 MHz sınırlarına kadar çıkmıştır. Standart olarak üretilen DDR2-SDRAM'lerden[20] daha üstün başarım değerleri sunan hız aşırtma[21] (over clocking) uygulanmış DDR2-SDRAM'ler de piyasada hatırı sayılır bir yer bulmuştur. DDR2-SDRAM'ler ile ilgili bilgiler Tablo 3-3'de verilmiştir.

---

[19] DDR-SDRAM çift kanal (dual channel) çalışma teknolojisini de desteklemektedir. Bu durumda aynı özellikte iki adet DDR-SDRAM kullanılması durumunda kapasite iki katına çıkmaktadır. Yani sistemde bir adet PC1600 özellikli DDR-SDRAM var iken kapasite (bant genişliği) 1.6 GB/s iken aynı özellikli ikinci PC1600 DDR-SDRAM'ın takılması ile kapasite 3.2 GB/s olmaktadır.

[20] PC2-3200, PC2-4200, PC2-5300, PC2-6400 ve PC2-8500 JEDEC tarafından DDR2-SDRAM standardı olarak belirlenmiştir.

[21] PC2-6000, PC2-7200, PC2-8000, PC2-8800, PC2-8888, PC2-9136, PC2-9200, PC2-9600 ve PC2-10000 hız aşırtma uygulanmış DDR2-SDRAM'ler olup, JEDEC tarafından standart kapsamına alınmamıştır.

Şekil 3-4: DDR2-SDRAM

Tablo 3-3: DDR2-SDRAM tipleri[22]

Tipi	Bellek bütünleşik devresi	Veri Genişliği (Byte)	G/Ç Hızı (MHz)	Kapasitesi (GB/s) (2 x Veri Genişliği x Hız)
PC2-3200	DDR2-400	8	200	3.200
PC2-4200	DDR2-533	8	266	4.267
PC2-5300	DDR2-667	8	333	5.333
PC2-6000	DDR2-750	8	375	6.000
PC2-6400	DDR2-800	8	400	6.400
PC2-7200	DDR2-900	8	450	7.200
PC2-8000	DDR2-1000	8	500	8.000
PC2-8500	DDR2-1066	8	533	8.533
PC2-8800	DDR2-1100	8	550	8.800
PC2-8888	DDR2-1111	8	556	8.888
PC2-9136	DDR2-1142	8	571	9.136
PC2-9200	DDR2-1150	8	575	9.200
PC2-9600	DDR2-1200	8	600	9.600
PC2-10000	DDR2-1250	8	625	10.000

3.2.1 DDR3-SDRAM (Dual Data Rate 3 – SDRAM)

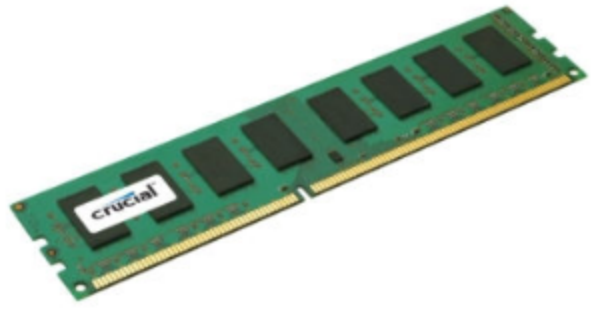
DDR3-SDRAM bellekler 64 bit (8 byte) olarak organize edilmiş, 1.5 volt ile çalışan, dolayısı ile daha az enerji tüketen ve ısınan, DIMM yapısında, 240 bacaklı ve tek çentiklidir (Şekil 3-5). DDR3-SDRAM bellekler, DDR2-SDRAM'in 800 MHz sınırından üretilmeye başlanmıştır. Standart olarak üretilen DDR3-SDRAM'lerden[23] daha üstün başarım değerleri sunan hız aşırtma[24] (over clocking) uygulanmış DDR3-SDRAM'ler de piyasada hatırı sayılır bir yer bulmuştur. DDR3-SDRAM'ler ile ilgili bilgiler Tablo 3-4'de verilmiştir.

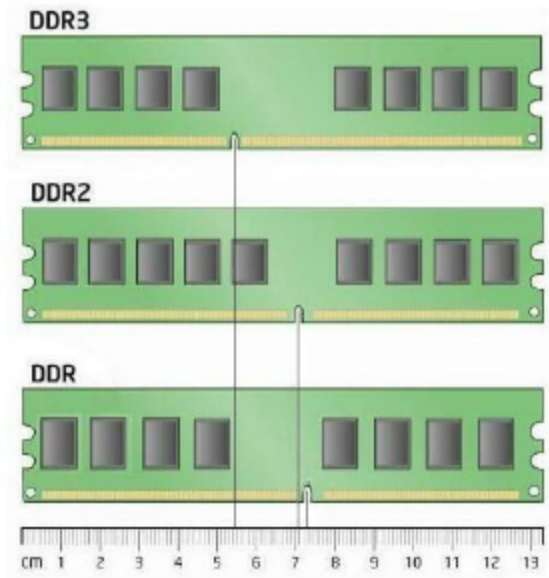
---

[22] DDR2-SDRAM çift kanal (dual channel) çalışma teknolojisini de desteklemektedir. Bu durumda aynı özellikte iki adet DDR2-SDRAM kullanılması durumunda kapasite iki katına çıkmaktadır. Yani sistemde bir adet PC2-3200 özellikli DDR2-SDRAM var iken kapasite (bant genişliği) 3.2 GB/s iken aynı özellikli ikinci PC2-3200 DDR2-SDRAM'ın takılması ile kapasite 6.4 GB/s olmaktadır.

[23] PC3-6400, PC3-8500, PC3-10600, PC3-12800, PC3-14900 ve PC3-17000 JEDEC tarafından DDR3-SDRAM standardı olarak belirlenmiştir.

[24] PC3-11000, PC3-13000, PC3-14400, PC3-14900, PC3-15000 ve PC3-17000 hız aşırtma uygulanmış DDR3-SDRAM'ler olup, JEDEC tarafından standart kapsamına alınmamıştır.





Şekil 3-5: a) DDR3-SDRAM b) DDR/DDR2/DDR3 Fiziksel karşılaştırma

Tablo 3-4: DDR3-SDRAM tipleri[25]

Tipi	Bellek bütünleşik devresi	Veri Genişliği (Byte)	G/Ç Hızı (MHz)	Kapasitesi (GB/s) (2 x Veri Genişliği x Hız)
PC3-6400	DDR3-800	8	400	6.400
PC3-8500	DDR3-1066	8	533	8.533
PC3-10600	DDR3-1333	8	667	10.667
PC3-11000	DDR3-1375	8	688	11.000
PC3-12800	DDR3-1600	8	800	12.800
PC3-13000	DDR3-1625	8	813	13.000
PC3-14400	DDR3-1800	8	900	14.400
PC3-14900	DDR3-1866	8	933	14.993
PC3-15000	DDR3-1866	8	933	15.000
PC3-16000	DDR3-2000	8	1000	16.000
PC3-17000	DDR3-2133	8	1066	17.066

3.2.2 DDR4-SDRAM (Dual Data Rate 4 – SDRAM)

JEDEC tarafından Eylül 2012 de belirlenmiş yeni standarttır. DDR4-SDRAM bellekler 64 bit (8 byte) olarak organize edilmiş, 1.05-1.2 volt ile çalışacak, DIMM yapısında ve 284 bacaklı olacaktır. Taşınabilir bilgisayarlar için 256 bacaklı SO-DIMM kullanılacaktır. DDR4-SDRAM bellekler, DDR3-SDRAM'in kaldığı 2133

[25] DDR3-SDRAM çift kanal (dual channel) ve üç kanal (tri channel) çalışma teknolojisini de desteklemektedir. Bu durumda aynı özellikte iki veya üç adet DDR3-SDRAM kullanılarak kapasite iki-üç katına çıkarılabilmektedir. Yani sistemde bir adet PC3-6400 özellikli DDR3-SDRAM var iken kapasite (bant genişliği) 6.4 GB/s, iki adet var iken 12.8 GB/s ve üç adet var ilen 19.2 GB/s olacaktır.

MHz sınırından başlayarak üretilecek olup 4266 MHz üst sınırına ulaşması beklemektedir. Böylece bellek birimleri de işlemcilere yakın çalışma frekanslarına ulaşmış olacaklardır.

Tablo 3-5: DDR4-SDRAM tipleri[26]

Tipi	Bellek bütünleşik devresi	Veri Genişliği (Byte)	G/Ç Hızı (MHz)	Kapasitesi (GB/s) (2 x Veri Genişliği x Hız)
PC4-17000	DDR4-2133	8	1066	17.066
PC4-25600	DDR4-3200	8	1600	25.600
PC4-34100	DDR4-4266	8	2133	34.128

Şekil 3-6: DDR4-SDRAM

3.2.3 RDRAM (Rambus Dynamic Random Access Memory)

RDRAM, 16 bit (2 byte) veri genişliğinde, 184 bacaklıdır[27]. SDRAM belleklerin yetersiz kalması üzerine geliştirilmiştir. Çalışma frekansları ve tasarım özellikleri nedeniyle bellek birimleri ısınmaktadır. Isının dağıtılabilmesi amacıyla bellek birimlerinin üzerine soğutucu plakalar takılmıştır. Ayrıca tasarımları gereği ek bir sonlandırıcı eleman ile kullanılmaları üretim maliyetlerini de arttırmıştır. Gerek şekil gerekse bağlantılarındaki farklık nedeniyle RDRAM modülleri RIMM (Rambus Inline Memory Module) olarak adlandırılır. RDRAM'ler ile ilgili bilgiler Tablo 3-6'de verilmiştir.

3.3 Giriş/Çıkış Yolları

Bilgisayarların dış dünya ile haberleşmesini sağlayan giriş/çıkış birimleri ile çeşitli özellikte olan giriş/çıkış bağlantı ara yüzleri üzerinden bağlantı kurulmaktadır. Kimi hızlı, kimi yavaş çalışan bu yolların fiziksel bağlantı şekillerinde de farklılıklar vardır. Bu nedenle giriş/çıkış yollarının hızlı ve yavaş ayrımı altında incelenmesi daha doğru olacaktır.

---

[26] DDR4-SDRAM Samsung firması tarafından üretilmiş olmak ile birlikte henüz piyasadan temin edilebilir durumda değildir. 2013 yılı içinde 1.2 volt, 1.1 volt ve 1.05 volt ile çalışan modeller olarak piyasaya çıkması beklenmektedir.

[27] RDRAM 32 bit (4 byte) veri genişliğinde 232 bacaklı, 64 bit (8 byte) veri genişliğinde 326 bacaklı, taşınabilir bilgisayarlar için SO-RIMM yapısında 160 bacaklı olarak da üretilmiştir.

Tablo 3-6: RDRAM tipleri

Tipi	Kanal Sayısı	Veri Genişliği (Byte)	Hızı (MHz)	Kapasitesi (GB/s) (2 x Veri Genişliği x Hız)
PC600-RIMM1200	1	2	2x300	1.200
PC600	2	2	2x300	2x1.200
PC700-RIMM1400	1	2	2x350	1.400
PC700	2	2	2x355	2x1.420
PC800-RIMM1600	1	2	2x400	1.600
PC800	2	2	2x400	2x1.600
PC1066-RIMM2100	1	2	2x533	2.100
PC1066	2	2	2x533	2x2.100
PC1200-RIMM2400	1	2	2x600	2.400
PC1200	2	2	2x600	2x2.400
RIMM3200	2	4	2x400	3.200
RIMM4200	2	4	2x533	4.200
RIMM4800	2	4	2x600	4.800
RIMM6400	2	4	2x600	6.400

3.3.1 Yüksek Hızlı Giriş/Çıkış Yolları

İşlemci teknolojisindeki gelişmeler ile sağlanan yüksek işlem gücünden yararlanabilmek amacıyla sistemlerin çalışması için kaçınılmaz olan çevre birimlerin işlemci ile olan bağlantılarının da hızlandırılması gerekir. Bu amaçla grafik, ikincil bellek ve diğer çevre birimlerinin işlemci ile bağlantısı için farklı özelliklerde yüksek hızlı giriş/çıkış yolları tasarlanmıştır.

3.3.1.1 Grafik Birim Bağlantısı

Kullanılan grafik tabanlı tasarım ve geliştirme yazılımlarının hızlı ve çok boyutlu (3D) işlem yapma ihtiyacı AGP (Advanced Graphic Port) PCIe[28] (PCI Express) gibi yüksek hızlı grafik bağlantılarının geliştirilmesine olanak sağlamıştır. AGP, veri aktarımı için 32 bit (4 byte) genişliğinde bir veri yolu kullanmaktadır. AGP ile ilgili bilgiler Tablo 3-7'de verilmiştir.

PCIe sahip olduğu 16 hat üzerinden yüksek hızlı, iki yönlü veri aktarım metodunu kullanır. PCIe ile ilgili bilgiler Tablo 3-8'de verilmiştir.

---

[28] PCIe, günümüzde yaygın olarak grafik kartlarını takmak için kullanılmaktadır. PCIe ve PCI-X yazılışları ve telaffuzlarının çok benzer olması nedeniyle sıklıkla karıştırılırlar. Oysa bu teknolojiler birbirinden farklı görünmelerinin yanı sıra birbiriyle uyumlu da değildirler. PCIe seri yapıda, noktadan noktaya 1-32 hatlı (her hat; biri alma gideri gönderme amaçlı kullanılan, diferansiyel iletimin yapıldığı kablo çifti olmak üzere toplamda 4 bacağı ifade eder) bağlantı destekler. PCIe x1 36 bacaklı, x4 64 bacaklı, x8 98 bacaklı ve x16 164 bacaklı farklı bağlantı noktalarını tarif etmektedir.

Tablo 3-7: AGP tipleri

Tipi	Veri Genişliği (Byte)	Hızı (MHz)	Kapasitesi (MB/s)
AGP x1	4	66	266
AGP x2	4	133	533
AGP x4	4	266	1066
AGP x8	4	533	2133

Tablo 3-8: PCIe tipleri

Tipi	Kodlama	Hat Sayısı (Lane)	Hat Genişliği (MB/s)	Toplam Kapasite (GB/s) (Hat SayısıxHat Genişliği)
PCIe 1.x	8b/10b	16	250	4
PCIe 2.x	8b/10b	16	500	8
PCIe 3.x	128b/130b	16	1000	16

3.3.1.2 İkincil Bellek Birim Bağlantısı

Bilgisayarların kullandığı ikincil bellek birimleri ile bağlantısını sağlayabilmek amacıyla, Compaq ve Western Digital işbirliği ile IDE/EIDE (Enhanced Integrated Drive Electronics)[29] adı verilen arayüz geliştirilmiştir. Genel olarak her kişisel bilgisayar, üzerinde 2 adet EIDE[30] kanalı ve kanal başına iki cihaz olmak üzere toplam 4 adet cihaz kullanımına imkân sağlamaktadır. Önceleri sadece sabit disk ünitelerinin sürülmesi için kullanılan bu standart arayüz daha sonraları CD-ROM, DVD, teyp ve sökülebilir yüksek kapasiteli disk/disket birimlerini de içine alacak şekilde düzenlenerek ATA/ATAPI (Advanced Technology Attachment Packet Interface) adını almıştır. EIDE birimlerinin yönetimi için işlemci gücü gereklidir. Bu da bu birimlere erişilirken işlemci zamanının bir kısmının bu amaç için ayrılması veya diğer bir deyişle işlemciye bir miktar yük verilmesi anlamına gelir. Ancak geliştirilen DMA (Direct Memory Access) tekniği ile IDE üzerinden bağlı cihazlardan belleğe veri, işlemcinin zamanını çalmadan aktarılabilmiştir. EIDE ile ilgili bilgiler Tablo 3-9'te verilmiştir.

---

[29] İkincil bellek birimleri için yaratılmış standardın adı ATA (Advanced Technology Attachment) olmak ile beraber ticari olarak IDE ve EIDE kısaltmaları kullanılmaktadır. 2003 yılında ATA'nın seri çalışan şekli SATA'nın (Serial Advanced Technology Attachment) piyasaya girmesi sonucu ATA, PATA (Paralel Advanced Technology Attachment) olarak anılmaktadır.

[30] EIDE, IDE'nin geliştirilmiş sürümü olup ikincil bellek birimlerindeki 528 MB'lık sınırı ortadan kaldırmıştır.

Tablo 3-9: EIDE tipleri

Standart	Yaygın Bilinen Adı	Kullanılan Teknik	Hızı (MB/s)	Bağlantı (Bacak)
ATA-1	ATA, IDE	PIO Mode 0	3.3	40 veri + 4 güç
ATA-1	ATA, IDE	PIO Mode 1	5.2	40 veri + 4 güç
ATA-1	ATA, IDE	PIO Mode 2	8.3	40 veri + 4 güç
ATA-2	EIDE, Fast ATA	PIO Mode 3	11.1	40 veri + 4 güç
ATA-2	EIDE, Fast ATA	PIO Mode 4	16.66	40 veri + 4 güç
ATA/ATAPI-4	ATA-4, ATA/33	UDMA/33	33	40 veri + 4 güç
ATA/ATAPI-5	ATA-5, ATA/66	UDMA/66	66	80 veri + 4 güç
ATA/ATAPI-6	ATA-6, ATA/100	UDMA/100	100	80 veri + 4 güç
ATA/ATAPI-7	ATA-7, ATA/133	UDMA/133	133	80 veri + 4 güç
SATA/150	SATA 1.5Gb/s, SATA	8B/10B Encode	150	7 veri + 15 güç
SATA/300	SATA 3.0 Gb/s, SATA II	8B/10B Encode	300	7 veri + 15 güç
SATA/600	SATA 6.0 Gb/s, SATA III	8B/10B Encode	600	7 veri + 15 güç

Tablo 3-10: SCSI tipleri

Tipi	Bilinen diğer adı	Veri genişliği (Bit)	Hız	Kapasite (MB/s)	Birim (Adet)	Bağlantı (Bacak)	Kablo[31] boyu (m)
SCSI–1	Narrow	8	5 MHz	3.5-5	7	50	6
SCSI–2	Fast	8	10 MHz	10	7	50	3
	Fast Wide	16	10 MHz	20	15	68	3
Ultra SCSI	Ultra	8	20 MHz	20	7	50	1.5
	Ultra Wide	16	20 MHz	40	15	68	12
Ultra2 SCSI	Ultra 2	8	40 MHz	40	7	50	12
	Ultra 2 Wide	16	40 MHz	80	15	68	12
Ultra 3 SCSI	Ultra 160	16	80 MHz	160	15	68	12
	Ultra 320	16	160 MHz	320	15	68	12
SSA		Seri	200 Mbit	40	192	2/4 TP, FO	25/10km
SSA-40		Seri	400 Mbit	80	192	2/4 TP, FO	25/10km
FC (Fibre Channel)[32]	FC-AL 1 Gb	Seri	1 Gbit	100	127	FO	50km
	FC-AL 2 Gb	Seri	2 Gbit	200	127	FO	50km
	FC-AL 4 Gb	Seri	4 Gbit	400	127	FO	50km
	FC-AL 8 Gb	Seri	8 Gbit	800	127	FO	50km
	FC-AL 10 Gb	Seri	10 Gbit	1000	127	FO	50km
	FC-AL 16 Gb	Seri	16 Gbit	1600	127	FO	50km
	FC-AL 20 Gb	Seri	20 Gbit	2000	127	FO	50km
SAS 3 Gbit/s		Seri	3 Gbit/s	300	65.535	26-36	10
SAS 6 Gbit/s		Seri	6 Gbit/s	600	65.535	26-36	10
SAS 12 Gbit/s	Geliştiriliyor	Seri	12 Gbit/s	1200	65.535	26-36	10

[31] FO kablonun MMF/SMF olması ve dalga boyuna (850nm, 1310nm, 1490nm ve 1550nm) göre mesafe değişir.

[32] 1G, 2G, 4G ve 8G FC 8b/10b kodlaması kullanır, 10G, 16G ve 20G FC 64b/66b kodlaması kullanır. 1G, 2G, 4G, 8G ve 1G FC birlikte kullanılabildikleri gibi geriye doğru uyumludur. 10GB ve 20GB geriye doğru uyumlu değildir.

3.3.1.3 Diğer Çevre Birim Bağlantıları

Bilgisayarlara değişik özelliklerde çevre birimlerin bağlanabilmesi ve aradaki veri akışının sağlanabilmesi için ek kartlara ihtiyaç duyulmaktadır. Bu kartlar, bilgisayar sistemi üzerinde bulunan genişletme yuvalarına (expention slot) takılır. Kişisel bilgisayarlarda ilk kullanılmaya başlayan ISA (Industry Standard Architecture) / EISA (Extended Industry Standard Architecture) tipindeki yuvalar ve kartlar teknolojideki gelişmeleri ile birlikte yerlerini sırasıyla MCA (Micro Channel Architecture) daha sonra da PCI (Peripheral Component Interconnect) olanlara bırakmıştır. Geliştirilen seri bağlantı teknolojileri ile PCI tipindeki yuvalar ve kartların yanında PCI Express ve PCI-X tipindekiler de görülmeye başlanmıştır. Genişletme yuvaları ile ilgili bazı bilgiler Tablo 3-11'de verilmiştir.

Tablo 3-11: Kişisel bilgisayarlarda kullanılan bağlantı yolları

Tipi	Veri Yolu Genişliği (Bit)	Adres Yolu Genişliği (Bit)	Hızı (MHz)	Kapasitesi (MB/s)
ISA (XT)	8	20	4.77–8	8
ISA (AT)	16	24	8–10	16
EISA	32	32	8–10	32
MCA	32	32	5–20	66
VLB	32	32	40	160
PCI	32	32	33	133
PCI	64	32	33	266
PCI	32	32	66	266
PCI	64	32	66	533
PCI-X 133[33]	64	32	133	1.066
PCI-X 266	64	32	266	2.133
PCI-X 533	64	32	533	4.266

3.3.1.4 IEEE 1394[34]

Seri bir bağlantı şekli olan IEEE 1394 sesli/görüntülü cihazların veri aktarımında kullanılmaya başlayan bir bağlantı şeklidir. SCSI bağlantıların yerini almak üzere tasarlanmış, ucuz ve yüksek bant genişliği ile 63'e kadar cihaz bağlanmasına imkân sağlayan bir teknoloji olup çalışırken üzerine cihaz eklemek ve çıkartabilmek (hot swap) mümkündür. 6 bacaklı bir bağlantı kullanır.

Ancak Sony firması güç taşımak için kullanılan bacakları devre dışı bırakarak 4 bacaklı bağlantı elemanlarını da kullanmıştır. IEEE 1394, USB'nin sağladığı ko-

[33] PCI-X ve PCIe bilgisayarlarda kullanılan hızlı, dahili bağlantı yolları olmaları, yazılışları ve telaffuzlarının çok benzer olması nedeniyle sıklıkla karıştırılırlar. Oysa bu teknolojiler birbirinden farklı görünmelerinin yanı sıra birbiriyle uyumlu da değildirler. PCI-X bağlantı yolu paralel yapıda, geriye yönelik olarak 32 bit PCI birimleri de desteklerken, PCIe seri yapıda, noktadan noktaya bağlantıyı destekleyen, tamamen farklı bir fiziksel yapıdadır.

[34] IEEE 1394 aynı zamanda Apple tarafında Firewire, Sony tarafında i.Link olarak da isimlendirilmiştir.

laylıklar nedeni ile fazla yaygınlaşamamıştır. Firewire ile ilgili bilgiler Tablo 3-12'da verilmiştir.

Tablo 3-12: IEEE 1394 tipleri[35]

Tipi	Diğer adı	Bağlantı bacak sayısı	Cihaz Sayısı	Kablo boyu (m)	Kapasitesi (Mbit/s)
IEEE 1394a	S100	4 veri + 2 güç	63	4.5	100
IEEE 1394a	S200	4 veri + 2 güç	63	4.5	200
IEEE 1394a	Firewire 400/S400	4 veri + 2 güç	63	4.5	400
IEEE 1394b	Firewire 800/S800	9 veya UTP	63	4.5 / 100	800
IEEE 1394b	Firewire 1600/S1600	9 veya UTP	63	4.5 / 100	1600
IEEE 1394b[36]	Firewire 3200/S3200	9 veya UTP	63	4.5 / 100	3.200

3.3.1.5 PCMCIA (Personal Computer Memory Card International Association)

Özellikle taşınabilir türdeki bilgisayarlara bağlanacak olan çevre birimleri için geliştirilmiş bir teknolojidir. Bellek birimleri için TYPE–1 (3.3 mm), ağ bağlantısı, modem için TYPE–2 (5 mm) ve ikincil bellek birimleri için TYPE–3 (10 mm) olmak üzere 3 ayrı tipi mevcuttur. PCMCIA ile ilgili bilgiler Tablo 3-13'de verilmiştir.

Tablo 3-13: PCMCIA tipleri

Veri Genişliği (Bit)	Hızı (MHz)	Kapasitesi (MB/s)
16	2	4
32	33	132

3.3.1.6 USB (Universial Serial Bus)

USB, çevre birimlerinin zahmetsiz bir şekilde bilgisayar sistemlerine bağlanmasını sağlar. 127 tane cihazın birbirine papatya zinciri şeklinde (veya göbek kullanılarak) bağlanmasına imkân veren, birimlerin bağlantı üzerinden beslendiği, 4 tel kullanarak seri iletim yapan bir teknolojidir. Sağladığı esneklik, kapasite ve kullanım kolaylığı sayesinde son derece yaygın olarak kullanılır hale gelmiştir. USB ile ilgili bilgiler Tablo 3-14'de verilmiştir.

---

[35] IEEE 1394 S100'ün gerçek hızı 98.304 Mbit/s dir. Diğer standartlar da bu standart hızın katları olarak oluşturulmuştur. Ancak rakamın hatırlanmasını kolaylaştırmak için değer yuvarlatılarak hız 100 Mbit/s olarak ifade edilmektedir. 4.5m'lik mesafe sınırı, 16 adet kablonun papatya zinciri şeklinde eklenmesiyle 72m'ye çıkarılabilmektedir.

[36] IEEE 1394b, eSATA, USB 3.0 ve Thunderbold gibi teknolojilerin gerisinde kalması nedeniyle kişisel bilgisayarlarda yayğınlaşmamıştır.

Tablo 3-14: USB tipleri

Tipi	Hızı (Mbit/s)	Mesafe (m)	Hız	Açıklama
USB 1.0	1.5	5	Düşük	Klavye, fare kumanda kolu bağlantılarında kullanılır.
USB 1.1	12	5	Tam	Harici depolama birimlerinin bağlantısında kullanılır
USB 2.0	480	5	Yüksek	Harici depolama birimlerinin bağlantısında kullanılır
USB 3.0	5.000	3	Süper	Harici depolama birimlerinin bağlantısında kullanılır

3.3.2 Düşük Hızlı Giriş/Çıkış Yolları

Bilgisayarlara bağlanan bazı çevre birimleri gerek bazı elektromekanik kısıtlamalar gerekse alış-verişte bulunulan veri miktarının az olması nedeniyle daha düşük hızlarda kullanılmaktadır. Bu birimlerin bilgisayar ile bağlantısını sağlamak amacıyla düşük hızlı giriş/çıkış yolları da desteklenmektedir.

3.3.2.1 Paralel (Eşanlı) İskele

25 bacak üzerinden gerçekleştirilen paralel bağlantı ile 115 KB/s hızına ulaşılabilir. ECP/EPP (Enhanced Capability Port/ Enhanced Paralel Port) özellikli paralel iskele 3 MB/s hızında aktarım yapabilir. Genel olarak yazıcı, tarayıcı gibi birimlerin bağlantısı için kullanılan paralel iskele USB'nin sağladığı avantajlar nedeniyle her geçen gün daha az kullanılır hale gelmiştir.

3.3.2.2 Seri (Dizisel) İskele

15 bacak üzerinden gerçekleştirilen seri bağlantı ile 115 Kb/s hızında iletim yapılabilmektedir. Fare (mouse), terminal ve modem gibi yavaş çevre birimlerinin bağlantısı için yaygın olarak kullanılan seri bağlantı USB'nin sağladığı avantajlar nedeniyle her geçen gün daha az kullanılır hale gelmektedir. Günümüzde taşınabilir bilgisayar sistemlerinde seri bağlantılar yerlerini tamamen USB bağlantılara bırakmıştır.

4. GENEL BİLGİLER VE 80X86 AİLESİ

Yarı iletken teknolojisindeki gelişmeler daha hızlı ve büyük değerler ile hesaplama yapabilen işlemcilerin üretilmesine imkân sağlamıştır. Önceleri hesaplama makineleri için üretilen sınırlı kapasiteli işlemciler zaman içinde insanoğlunun vazgeçemediği kişisel bilgisayar sistemlerinin en önemli parçası olmuştur. Intel firması da 80x86 kod adı ile ürettiği mikroişlemcilerinin kapasite, hız ve özelliklerini zaman içerisinde geliştirilerek kişisel bilgisayar pazarındaki en yaygın kullanılan işlemci ailesini oluşturmuştur.

İşlemciler, üzerinde işlem yapabildikleri verinin büyüklüğüne bağlı olarak değerlendirilirler. Bu değerlendirme sonucu 8, 16, 32 ve 64 bitlik işlemciler şeklinde bir sınıflandırma yapmak mümkündür.[37]

4.1 Ana İşlemciler

Intel firmasınca üretilen 80x86 işlemci ailesi ve bunlara ait bazı temel özellikler söyle özetlenebilir;

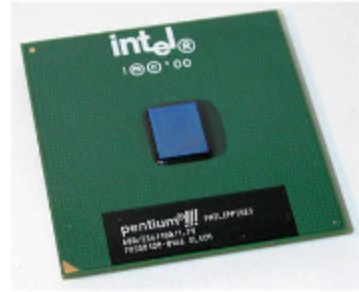
Tablo 4-1: Intel ailesi işlemciler

4004	Intel firması tarafından üretilen ilk mikro işlemci olup 4 bit'lik hesaplama yapar. 4004, Busicom firmasının ürettiği hesap makinelerinde kullanılmak üzere tasarlanmıştır. 640 byte bellek adresler.	
8008	4004'ün iki katı kapasiteli olan bu işlemci Mark–8 olarak bilinen ilk ev bilgisayarlarında ve aptal terminallerde kullanılmıştır.	
8080	8 bit'lik ilk işlemcilerdendir. Bir akümülatör (A) ve altı tane ikincil yazmaç (B, C, D, E, H ve L) bulunan bu modelde 16 bit'lik işlemler yapmak amacıyla B ile C ve H ile L yazmaçları birlikte kullanılabilmektedir. 8080 işlemcisi 2^{16}'lık (64 KB) bellek alanını adresler.	
8085	8085 işlemcisi, 8080 işlemcisinin benzeri olup daha basit bir donanım arabirimi ve daha yüksek başarıma sahiptir.	
8088 **80188**	8 bit'lik veri yoluna sahip olduğu için donanımcılar tarafından 8 bit'lik işlemci, sahip olduğu 16 bit'lik yazmaçlar ile 16 bit'lik işlem yapma kabiliyeti nedeniyle yazılımcılar tarafından 16 bit'lik olarak değerlendirilir. 8086 ailesinin diğer üyeleri ile kod uyumlu olup, 20 adet adres bacağı ile 1 MB (2^{20}) büyüklüğünde bellek adresler. 80188 ise 8088'in bazı ek komutlar ile kontrol amaçlı işlemlerde kullanılmak üzere tasarlanmış halidir.	

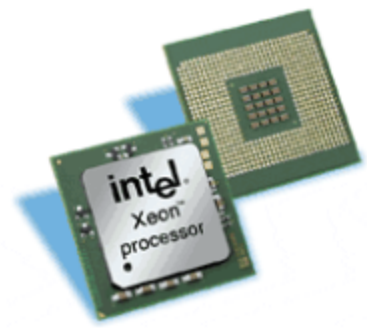
[37] Bazı kontrol amaçlı işlemlerde kullanılmak üzere 7 veya 9 bitlik gibi ikinin kuvveti ile ifade edilmeyen büyüklükler üzerinde işlem yapan işlemciler de vardır.

8086 **80186**	16 bit'lik veri yoluna sahiptir. Yazmaçları 16 bit'liktir. 20 adres bacağı ile 1 MB (2^{20}) bellek adresleyebilir. 80186 ise 8086'nın bazı ek komutlar ile kontrol amaçlı işlemlerde kullanılmak üzere tasarlanmış halidir.	
80286	16 bit'lik mikroişlemcidir. 24 adres bacağı ile 16 MB (2^{24}) bellek adresler. Yazmaçları 16 bit'liktir. Gerçek kip (Real Mode) ve 16 bit korumalı kip (Protected Mode) olmak üzere iki kipte çalışabilir. Ancak kipler arası geçişler problemlidir. Her ne kadar korumalı kip kavramı bu işlemci ile duyurulduysa da tam olarak kullanılamamıştır. Yine bu işlemci ile sanal bellek (Virtual Memory) ve komutları yorumlamak için önceden bir kuyruğa alarak o kuyruk üzerinden işlemler (pre-fetch queue) yapılmaya başlamıştır.	
80386	32 bitlik yazmaçlar kullanmaktadır. 80386 tipi işlemciler de gerçek ve 32 bit korumalı kipte çalışabilmektedirler. Daha önceki modellerde olan tüm özellikler bu işlemcilerde de vardır. Ayrıca 32 bit'lik yazmaçları sayesinde işlem kapasitesi büyük ölçüde arttırılmıştır. **80386DX** işlemcisi 32 bit veri yolu, 32 bit adres yoluna sahiptir. 32 bit'lik yazmaçlara sahiptir. Daha önceki işlemcilerin sahip olduğu özelliklere ek olarak sayfalı bellek kullanımı (paging) desteğine sahiptir. **80386SX** işlemcisi 16 bit veri yolu, 24 bit adres yoluna sahip olup 32 bit'lik yazmaçları vardır. İşlemci 32 bit işlem yapmasına rağmen veri yolunun 16 bit olması nedeniyle donanımcılar tarafından 16 bit'lik bir işlemci olarak değerlendirilmektedir. **80386SL** işlemcisi 80386SX işlemcisinin taşınabilir bilgisayar sistemlerinin ihtiyaçları göz önüne alınarak tasarlanmış, düşük hız ve güç tüketimi olan modeldir. **80376** işlemcisi 80386SX'in korumalı kipte çalışan sürümüdür. **80386EX** işlemcisi 80376 işlemcisinin yerini almıştır.	
80486	32 bit'lik veri yolu ve 32 bit'lik adres yolu vardır. SX ve DX olmak üzere iki ayrı modeli bulunur. Gerçek ve korumalı kiplere ek olarak işlemci üzerinde ön bellek (cache memory) kullanılmaya başlanmış ve DX modellerinde matematik işlemci de ana işlemci ile birleştirilmiştir. Bu seride ayrıca ardışık düzen (pipeline)[38] tekniği de kullanılmaya başlanmıştır. **80486DX** matematik işlemci içinde olan 80486 modelidir. DX2 ve DX4 olarak kodlanan modelleri FSB hızının 2 ve 4 katı hızlarda çalıştığını göstermektedir. **80486SX** matematik işlemcisi olmayan 80486 modelidir. **80486SL** işlemcisi 80486 işlemcisinin taşınabilir bilgisayar sistemlerinin ihtiyaçları göz önüne alınarak tasarlanmış, düşük hız ve güç tüketimi olan modeldir.	

[38] Bir işlemin tamamlanması için komutun alınması, yorumlanması, işlenenlerin alınması, işlemin yapılması ve sonucun üretilmesi aşamalarının bir biri ardına takip edilmesi gereklidir. Ardışık düzende (n) numaralı aşamanın çıkışının, (n+1) numaralı aşamanın girişi olarak düzenlendiği, işini bitiren (n) numaralı aşamanın bir sonraki işlem ile ilgili işlem ile devam ettiği, herhangi bir anda tüm aşamalarda farklı bir işlem ile ilgili işlerin yapıldığı (paralel) çalışma düzenine verilen isimdir.

Pentium	Intel firması, işlemci serileri için kullandığı kodlama sisteminin rakipleri tarafından da kullanılması üzerine bir sonraki seriye 80586 demek yerine Pentium olarak isimlendirmeyi uygun görmüştür. **Pentium** işlemciler 32'bitlik işlem yapabilmelerine rağmen, 64 bit büyüklüğünde veri yoluna sahiptirler. Ayrıca kullanılan ön bellek veri (data cache) ve komut (instruction cache) ön belleği olarak ayrılmıştır. **Pentium MMX**, Pentium işlemcilere MMX (MultiMedia eXtention) olarak adlandırılan bir grup ek komutun eklenmesi ile çoklu ortam uygulamalarındaki başarımın arttırıldığı modeldir.	
P6 Serisi	P6 işlemci serisi, aynı üretim teknolojisini kullanarak Pentium serisinin ötesinde başarım elde edebilmek için tasarlanmış işlemcilerden oluşmaktadır. **Pentium Pro**, temel olarak Pentium işlemcinin sahip olduğu özelliklere sahip olmak ile birlikte komut işleme ile ilgili işlemlerde sağlanan yenilikler ile işlem hızının arttırıldığı bir model olarak karşımıza çıkmaktadır. Bir saat çevriminde ortalama üç komutun çalışmasını tamamlayabilmektedir. Çok kısa bir süre piyasada kalan bu işlemci yerini Pentium II'lere bırakmıştır.	
	Pentium II'nin üzerindeki veri ve komut önbellekleri 16 KB'ta, L2 önbellek 1 MB'ta çıkarılmış, MMX özellikleri eklenmiş ve donanımda çeşitli iyileştirilmeler yapılmıştır.	
	Celeron işlemcisi Internet kullanımı, ofis uygulamaları ve ev ortamında kullanılabilecek her türlü eğitim ve kişisel gelişime yönelik çoklu ortam uygulamalarını kullanmaya imkân sağlayan, işlem gücünü düşük, 128 KB L2 önbellekle ucuz sistemler oluşturmaya yönelik bir işlemcidir. Pentium II (Covington, Mendocino kod isimli) ve Pentium III (Tualatin ve Coppermine kod isimli) taban alınarak geliştirilmiş modelleri vardır. Copermine kod isimli modelde SSE (*Streaming SIMD Extensions)* desteği vardır.	
	Pentium II Xeon, serisinde kendisinden önce üretilen işlemcilerin en üstün özelliklerine sahip olmanın dışında tam hızda çalışan 2 MB kapasiteli L2 önbellek barındıran bir işlemcidir. Çok (4, 8 gibi) işlemcili ölçeklenebilir sistemlerde görev yapabilen bir işlemcidir.	
	Pentium III İlk sürümleri genel olarak Pentium II işlemcisine benzerlik göstermek ile birlikte SSE (*Streaming SIMD Extensions)* adı verilen ve farklı veri kümeleri üzerinde tek bir komut setinin paralel olarak çalıştırılmasını sağlayan ilaveler ile daha yüksek hızlarda erişilmiştir. SSE ile birlikte 8 adet 128 bit kayan noktalı işlem yapma imkânı sağlayan yazmaç kümesi de sisteme eklenmiştir.	

	Pentium III Xeon serisinde kendisinden önce üretilen işlemcilerin en üstün özelliklerine sahip olmanın dışında tam hızda çalışan 2 MB kapasiteli L2 önbellek barındıran bir işlemcidir. Çok (2, 4, 8 gibi) işlemcili ölçeklenebilir sistemlerde görev yapabilen bir işlemcidir.	
Pentium 4	GHz hızlarını zorlayan, daha hızlı giriş/çıkış arabirimleri ile desteklenerek daha üstün başarım sağlamak üzere tasarlanmış işlemci serisidir. **Pentium 4**'de SSE2 (Streaming SIMD Extensions 2) kullanılmaya başlanmıştır. **Pentium 4 HT**'de HyperThreading teknolojisi yanı sıra SSE3 (Streaming SIMD Extensions 3) kullanılmaya başlanmıştır. **Mobil Pentium 4-M** serinin taşınabilir bilgisayar sistemleri için geliştirilmiş modelidir. **Pentium 4EE** ile kullanılmaya başlayan dual-core teknolojisi ile donanım düzeyinde multi-threading desteği sağlanmıştır. SSE, SSE2, SSE3, HyperThreading teknolojilerini barındırmaktadır.	
Itanium	**Itanium** ve **Itanium 2** işlemciler Intel'in 64 bitlik yüksek başarımlı, sunucu seviyesi işlemci ailesi olarak duyurulmuştur. Ancak Itanium serisi 32 bitlik eski uygulamaları emülasyon modunda çalıştırmada beklenen başarımı sağlayamamış, zaman içinde yerini çok çekirdekli, büyük bellek adresleme (2^{64} byte) kabiliyetine sahip Itanium 2 işlemcisine bırakmıştır.	
Core	**Core Duo** işlemciler 32 bitlik çift çekirdekli yapıda olup, taşınabilir bilgisayar sistemlerinin batarya süresini uzatmak adına etkin güç yönetimi özelliklerine sahiptir. 2 MB L2 önbelleğe sahiptir. **Core Solo** işlemciler taşınabilir bilgisayar sistemleri için tek çekirdekli olarak üretilmiştir. **Core 2 Solo** işlemciler düşük güç tüketimi ile taşınabilir bilgisayar sistemlerinde kullanılmak üzere üretilmiştir. **Core 2 Duo** işlemciler 64 bitlik çift çekirdekli yapıda olup, L2 önbellek miktarları 6 MB kadar attırılmıştır. SSE4 (Streaming SIMD Extensions) kullanan hem masaüstü hem de taşınabilir bilgisayar sistemlerinde kullanılan modeldir. **Core2 Quad** işlemciler dört çekirdek barındırmakta olup Core 2 Duo'un iki katı bir başarım sağlamaktadır. **Core 2 Extreme** Core 2 Duo ve Core 2 Quad'in daha yüksek frekansta çalışabilir modelidir. **Core i3** işlemci 2 çekirdekli yapıda olup dâhili GPU barındıran, i serisi işlemciler arasında düşük işlem gücüne sahip olan modeldir. **Core i5** işlemci 2 veya 4 çekirdekli yapıda olup dâhili GPU barındıran, i serisi işlemciler arasında orta seviyede işlem gücüne sahip olan modeldir. **Core i7** işlemci 2, 4 veya 6 çekirdekli yapıda olup, i serisi işlemciler arasında en yüksek işlem gücüne sahip olan modeldir	

Xeon	**Xeon** işlemciler web sunucuları, disk sunucuları (SAN - NAS), arama motorları gibi çok üst derecede başarım gerektiren özel iletişim uygulamalarında kullanılmak üzere tasarlanmıştır. Uygun tümleşik devre takımları (chipset) ile kullanıldıklarında yüksek bellek kapasiteleri, yüksek bant genişlikli bellek ve giriş/çıkış erişimleri sağlarlar. Sahip oldukları 64 bitlik çok çekirdekli yapı, büyük L2 ön bellek, ölçeklenebilir olma (scaleable-çok işlemcili sistemlerde çalışabilirlik) özelliklerinin sağladığı üstün başarımları ile dikkat çekerler.	
Atom	**Atom** işlemciler, özellikle netbook ve taşınabilir sistemlerin gereksinimleri dikkate alınarak 32 ve 64 bit destekli, düşük hacim/güç/maliyet göz önünde bulundurularak geliştirilmiş bir seridir. Günümüzde 2 GHz'i aşan hızlarda çok çekirdekli işlemciler olarak masaüstü sistemlerde de karşımıza çıkmaktadır.	

80486 serisine kadar Intel firmasının yanı sıra AMD (Advanced Micro Devices), Cyrix, IBM (International Business Machines), TI (Texas Instruments) ve NexGen gibi firmalar da aynı bacak dizilişine sahip, kod uyumlu işlemciler üreterek piyasaya sunmuştur. Bu durum doğal olarak kullanıcıların fiyat/başarım oranlarına göre ana kartlarını ve çevre birimlerini değiştirmeden işlemci yükseltmeleri veya değiştirmelerine imkân sağlamıştır[39]. Pazardaki rakiplerinin haksız rekabet ile pazar payının azalttığını gören Intel firması 5. nesil işlemcisi olan Pentium ile başlamak üzere yeni nesil işlemcilerinde farklı bacak dizilişleri, kılıflama ve tasarım özelliklerini kullanmaya başlamıştır. Bu durum karşısında diğer üretici firmalar da kaçınılmaz olarak fiziksel tasarımlarında farklılıklara gitmiş, farklı bacak dizilişlerine sahip ancak yazılım olarak Intel firmasınınkiler ile uyumlu işlemciler üretmişlerdir. Böylece ister Intel olsun, ister olmasın 80x86 tabanlı bilgisayar sistemleri geriye doğru kod uyumlu (backward code compatible[40]) olarak kalabilmiştir. Ancak farklı üreticilerinin işlemcilerinin bacak dizilişlerindeki farklılıklar ana kart üreticileri için önemli bir sorun halini almıştır. Doğal olarak ana kart üreticileri, teknolojiyi sürükleyen işlemcileri destekleyecek olan ana kartların pazarda yer bulabileceğini düşünerek üretimlerini şekillendirmeye başlamışlardır. Bu durum diğer işlemci üreticilerinin zaman içinde güçlerini birleştirerek ayakta kalmaları sonucunu doğurmuştur. Günümüzde yarışta sadece AMD firması kalmıştır. FX, Phenom II, Athlon II ve Semptron serilerini halen Intel işlemciler ile kod uyumlu olarak üretmektedir.

---

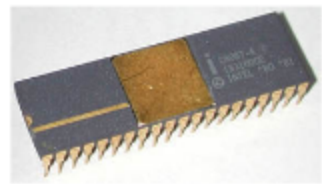
[39] Farklı firmalarca üretilen 8086/8088/80286/80386/80486 işlemciler yazılımın yanı sıra bacak dizilişleri ve kılıflama gibi fiziksel özellikleri ile de uyumlu oldukları için ana kart üzerinde basit ayarlamalar yapılmak kaydı ile birbirlerinin yerine kullanılabilirler.

[40] Eski nesil işlemcilerde çalışan programlar yeni nesil işlemciler üzerinde hiçbir değişiklik yapılmadan kullanılabilirler. Bu kavrama geriye yönelik kod uyumluluğu (backward code compatibility) adı verilir. Ancak yeni nesil işlemciler için geliştirilmiş programların eski nesil işlemciler üzerinde çalışması mümkün olmayabilir.

4.2 Matematik İşlemciler

Her geçen gün artan hesaplama ihtiyacı göz önüne alınarak kayan noktalı işlemlerde (Tablo 4-3) ana işlemcilere yardımcı olmak üzere matematik işlemciler tasarlanmıştır. Ana işlemciler, hesaplamalar sırasında karşılaştıkları kayan noktalı işlemleri, sistemde bir matematik işlemcinin mevcut olduğu durumlarda ona aktararak daha hızlı sonuçlandırabilmektedir. Ana işlemcilerin gelişen özellikleri ve artan hızları ile uyumlu olacak şekilde matematik işlemcilerde de değişiklikler olmuştur. Ancak bir yandan sistem üzerinde iki farklı işlemcinin kullanımının meydana getirdiği tasarım ve üretim zorlukları, diğer yandan işlemciler arasındaki veri alış verişinin neden olduğu zaman kaybı göz önüne alınarak matematik işlemciler, ana işlemcilerin içine üretim aşamasında yerleştirilmiştir. 80486DX işlemcisine kadar olan seride matematik işlemciler ayrı bir tümleşik devre olarak sistemde yer alırken, daha üst model ana işlemciler kendi içlerinde bulunan matematik işlemci desteğinden yararlanmıştır. Buna göre mevcut matematik işlemciler ve bazı özellikler Tablo 4-2'de verilmiştir.

Tablo 4-2: Matematik İşlemciler

8087	8087, 8088, 80186 ve 80188 işlemcilerinin matematiksel işlem güçlerini artırmak üzere tasarlanmış matematik işlemcidir. 8087, 8 adet 80-bit genel amaçlı yazmaç kullanmaktadır. Bütün kayan noktalı işlemler yığından ve harici bellek üzerinden alınan veri üzerinde gerçekleştirilir. Harici bellek üzerindeki veri işlemleri için adres hesaplama işlemleri ana işlemci tarafından yapılır. Çözülen komutun harici bellek erişimi olduğu anlaşıldığında ana işlemci gerekli adres hesaplamalarını yaparak o bellek alanından sahte bir okuma işlemi yapar. Bu sayede 8087 matematik işlemcisi bellek adresini alarak belleğe erişir, bu alandan veriyi okur veya yazar. Bu işlemler devam ederken ana işlemci, 8087'nin kayan noktalı işlemi bitirmesini beklemeksizin diğer işlemlerine devam edebilir. Birden çok kayan noktalı işlemin yürütülmesinin gerekli olduğu durumlarda kullanılan WAIT (FWAIT) komutu ile matematik işlemcinin yeni işlem yapabilir hale gelmesi beklenir. Bu sayede tam sayı ve kesirli sayı işlemleri aynı anda paralel olarak yapılabilir. Intel'in yanı sıra AMD'de benzer kodlu matematik işlemciler üretmiştir.	
80287	80287 matematik işlemcisi, 80286 ailesi işlemcileri ile birlikte kullanılmak üzere üretilmiştir. 80287 mimari olarak Intel 8087'ye benzer, tamamen kod uyumludur. Ancak her iki işlemcinin de bacak dizilişleri farklıdır. Çalıştırdıkları komutlar ve bu komutları gerçekleştirmek için harcadıkları süre hemen hemen aynıdır. Bununla birlikte 80287 ve 8087 arasında şu farklılıklardan söz edilebilir. • 80287, ana işlemci ile an uyumsuz (asynchronous) çalışır. 80286 işlemcisi harici saat işaretini 2'ye bölerken matematik işlemci 3'e böler. Her iki işlemci de aynı saat işaretine göre çalışırlarsa matematik işlemci, ana işlemcinin 2/3 hızında çalışmış olacaktır. • Ana işlemci kayan noktalı işlem yürüteceği zaman matematik işlemcinin müsait olup olmadığını kontrol eder. Eğer müsait değil ise müsait oluncaya kadar bekler. Bunun sonucunda 80287 komutlarının önünde kullanılan WAIT (FWAIT) komutlarının kullanımına gerek kalmaz.	

80387	80387, 80386 işlemcisi ile kullanılan matematik işlemcidir. 80387, 80386 ile aynı hızda çalışır. Gerçek, korumalı ve sanal (virtual) çalışma kiplerini destekler. Kendinden önce kullanılan 8087 ve 80287 ile tamamen kod uyumlu olup aşağıdaki nedenlerden dolayı daha hızlı işlem yapabilir. • Kayan noktalı işlemlerin iyileştirilmiş olması ve daha az saat işareti darbesinde işlemleri sonuçlandırması • Veri yolu genişliğinin 32-bit olması 80387, 80386DX işlemci ile kullanılan **80387DX** ve 80386SX işlemciyle kullanılan **80387SX** olmak üzere iki farklı tiptedir.	
80487	80486SX işlemciler ile birlikte kullanılan matematik işlemcidir. 80486DX ve daha üst modellerin matematik işlemcisi içinde olacak şekilde piyasaya çıkması ile ayrı bir matematik işlemci kullanımına gerek kalmamıştır.	

Tablo 4-3: Matematik işlemcilerin başarımları - birim zamanda yapılan işlem sayısı

	8086/8087	80286/80287	80386/80387	80486DX
Tamsayı işlem	1	2.7	9	20
Kayan noktalı işlem	1	1.7	10	40

80x86 ailesi ve bu aile ile uyumlu olan işlemciler hakkında bazı özet bilgileri bir tablo üzerinden incelemek işlemcilerdeki gelişim/değişim hakkında daha rahat fikir verecektir. Ancak bu tarz bir tablonun en güncel ve tutarlı bilgiler ile hazırlanması da oldukça zordur. Üreticiler rekabetin gereği olarak işlemciler üzerinde çeşitli iyileştirmeler yaparak piyasaya sürmektedir. Bu iyileştirmeler kimi zaman işlem hızlarının, önbellek kapasitelerinin, çekirdek sayılarının arttırılması olabildiği gibi artan transistör sayısına bağlı olarak üretim teknolojisinin değişmesi (daha yoğun bütünleşme tekniklerinin kullanılması), gerilim seviyelerinin düşürülmesine bağlı olarak güç tüketiminin azaltılması ile mobil kullanımda yaşanan batarya süresinin kısalığı probleminin çözümüne yönelik de olabilmektedir. Bunun doğal sonucu olarak işlemcilerin barındırdıkları transistör sayılarında ve başarımlarının ölçütü olan saniye de yapılan milyon işlem sayısı (MIPS) değerlerinde de farklılıklar oluşmaktadır. Buna rağmen Tablo 4-4[41] incelendiğinde Moore yasası uyarınca yaklaşık olarak her 18 ayda işlemcilerde kullanılan transistör sayısının ikiye katlandığı tespitin ne derece doğru olduğu görülecektir.

[41] Tablo 4-4'deki veriler imkânlar ölçüsünde basım tarihi itibari ile piyasada olan işlemcileri de kapsayacak şekilde, ulaşılabilen teknik bilgiler ışığında derlenmiştir.

Tablo 4-4: 80x86 ailesinin karşılaştırmalı tablosu[42]

İşlemci kodu	Üretim tarihi	Yazmaç büyüklüğü (Bit)	Yazmaç adedi ve tipi	Adres yolu (Bit) / Bellek	Veri yolu (Bit)	TRANSİSTÖR SAYISI (ADET)	Üretici firmalar	Hız (MHz)	İşlem miktarı (MIPS)	Matematik işlemci (FPU)	Açıklamalar
4004	1971	4	16 x 4bit	/ (640 B)	8	2.300	Intel	0.740	0.07	Yok	Hesap Makinesinde
8008	1972	8	7 x 8bit	14 / (16 KB)	8	3.500	Intel	0.5–0.8	0.05	Yok	Aptal Terminallerde
8080	1974	8/16	7 x 8bit 2 x 16 bit	16 / (64 KB)	8	6.000	Intel	2-3	0.29	Yok	Trafik lambalarında
8085	1976	8/16	7 x 8bit 2 x 16 bit	16 / (64 KB)	8	6.500	Intel	5	0.37	Yok	Çevre birimlerde
8086	1978	8/16	14 GP	20 / (1 MB)	16	29.000	Intel	4.77–8–10	0.3-0.75	8087	
8088	1979	8/16	14 GP	20 / (1 MB)	8	29.000	Intel	4.77–8	0.33-0.66	Yok	
80286	1982	8/16/32	16 GP	24 / (16 MB)	16	134.000	Intel	6–8–12.5	0.9-2.66	80287	Korumalı kip
80386DX	1985	8/16/32	32 GP	32 / (4 GB)	32	275.000	Intel, AMD, Cyrix,	16–20–25–33–40	5–9.9	80387DX	Paging, Paralel Stage
80386SX	1988	8/16/32	32 GP	24 / (16 MB)	16	275.000	Intel, AMD, Cyrix,	16–20–25–33	2.5–5.1	80387SX	Paging, Paralel Stage
80386SL	1990	8/16/32	32 GP	24 / (16 MB)	16	855.000	Intel, AMD, Cyrix,	20–25	4.1–5.3	80387SX	Paging, Paralel Stage
80486DX	1989	8/16/32	32 GP 80 FPU	32 / (4 GB)	32	1.200.000 1.600.000	Intel, AMD, Cyrix, IBM, TI,	25–33–40–50-DX2/50-DX2/66-DX4/100	20–70	İçinde	8 KB L1 , Ardışık düzen
80486SX	1991	8/16/32		32 / (4 GB)	32	1.185.000	Intel, AMD, Cyrix, IBM, TI	16–20–25–33-SX2/50	20–70	80487SX	
Pentium	1993	8/16/32	32 GP 80 FPU	32 / (4 GB)	64	3.100.000 4.500.000	Intel, AMD, Cyrix, NexGen	60–66–75–90–100–120–133–150–166–200–233–266–300	100–270	İçinde	8+8 KB L1, Ardışık düzen
Pentium Pro	1995	8/16/32	32 GP 80 FPU	36 / (64 GB)	64	5.500.000	Intel	150–166–180–200	250	İçinde	8+8 KB L1 256-1024KB L2

[42] Byte Sep/1991, Byte Jun/1994, Byte Nov/1994 ve Intel, AMD, Cyrix katalog bilgileri, http://www.intel.com ve "General Processor Information"- Tom Burd 1994-2000, http://en.wikipedia.org/wiki/Intel_processor, Upgrading and Repairing PCs, 20th Edition ve Intel® 64 and IA-32 Architectures Software Developer's Manual ve çeşitli internet kaynaklarından derlemiştir.

İşlemci kodu	Üretim tarihi	Yazmaç büyüklüğü (Bit)	Yazmaç adedi ve tipi	Adres yolu (Bit) / Bellek	Veri yolu (Bit)	TRANSİSTÖR SAYISI (ADET)	Üretici firmalar	Hız (MHz)	İşlem miktarı (MIPS)	Matematik işlemci (FPU)	Açıklamalar
PII PII Xeon	1998	8/16/32	32 GP 80 FPU 64 MMX	36 / (64 GB)	64	7.500.000 27.400.000	Intel	233–266–300–333–350–366–400–450	500	İçinde	16+16 KB L1 256–2048 KB L2
PIII PIII Xeon	1999	8/16/32	32 GP 80 FPU 64 MMX 128 XMM	36 / (64 GB)	64	9.500.000 28.100.000 44.000.000 55.000.000	Intel	350–400–450–500–533–550–600–650–667–700–733–750–800–850–866–900–933–1000–1130–1200–1260-1333-1400	1000	İçinde	16+16 KB L1 256–2048 KB L2
Celeron	1999	8/16/32	32 GP 80 FPU 64 MMX 128 XMM	36 / (64 GB)	64	7.500.000 19.000.000	Intel	266–300–333–366–400–433–450–466–500–533–550–566–600–633–650–667–700–733–750–766–800–850–900–950–1000–1060–1100–1130–1200–1260–1330–1400–1500–1600–1700–1800–2000–2100–2200–2300–2400–2500–2530–2660–2700–2800		İçinde	128-512KB L2
Pentium 4 Pentium M Pentium M HT	2000	8/16/32	32 GP 80 FPU 64 MMX 128 XMM	36 / (64 GB)	64	42.000.000 55.000.000 125.000.000 178.000.000	Intel	1400–1500–1600–1700–1800–1900–2000–2200–2260–2400–2500–2530–2600–2660–2800–3000–3060–3200–3400–3600-	1.700	İçinde	256–2048 KB L2
Xeon Xeon MP	2001	8/16/32	32,64 GP 80 FPU 64 MMX 128 XMM	36 / (64 GB) 40 / (1 TB)	64	42.000.000 55.000.000 108.000.000 178.000.000 286.000.000 675.000.000	Intel	1400–1500–1700–1800–1900–2000–2200–2400–2500–2600–2660–2700–2800–3000–3060–3200		İçinde	256–512 KB L2 1–4 MB L3 Çok işlemcili (2, 4, 8 hatta daha fazla) sistemlerde hizmet verme
Itanium Itanium 2	2001	64	128 GP 128 FPU	64	64	221.000.000 410.000.000	Intel	733–800–900–1000–1400–1500–1600-1730-2130-2400-2530	6.67GFLOPS@1.67 GHz,	İçinde	16 KB L1, 256 KB L2, 1,5–32 MB L3 Tek çevrimde 6 komut işleme 1,2,4 ve 8 çekirdek

İşlemci kodu	Üretim tarihi	Yazmaç büyüklüğü (Bit)	Yazmaç adedi ve tipi	Adres yolu (Bit) / Bellek	Veri yolu (Bit)	TRANSİSTÖR SAYISI (ADET)	Üretici firmalar	Hız (MHz)	İşlem miktarı (MIPS)	Matematik işlemci (FPU)	Açıklamalar
Mobile Pentium (Centrino)	2003	8/16/32	32 GP 80 FPU 64 MMX 128 XMM	36 / (64 GB)	64	77.000.000 140.000.000	Intel	900–1000–1100–1200–1300–1400–1500–1600–1700–1800–2000		İçinde	1–2 MB L2
Pentium D	2005	8/16/32/64	32 GP 80 FPU 64 MMX 128 XMM	40 / (1 TB)	64	230.000.000 376.000.000	Intel			İçinde	Çift çekirdek
Core	2006	8/16/32/64	32 GP 80 FPU 64 MMX 128 XMM	36 / (64 GB)	64	151.000.000	Intel	1200–1500–1600–1660–1730–1830–2000–2160–2330		İçinde	1 ve 2 çekirdekli, farklı hız ve önbellek kapasitesi
Core 2	2006	8/16/32/64	32 GP 80 FPU 64 MMX 128 XMM	40 / (1 TB)	64	167.000.000 291.000.000 410.000.000 586.000.000 820.000.000	Intel	1060–1200–1333–1400–1500–1600–1660–1800–1830–1860–2000–2130–2160–2200–2260–2330–2400–2500–2600–2660–2670–2700–2830–2930–3000–3070–3160–3200–3330	61.119	İçinde	2 ve 4 çekirdekli, faklı hız ve önbellek kapasitesi
Atom	2008	8/16/32/64	32 GP 80 FPU 64 MMX 128 XMM	32 / (4 GB)	64	47.000.000 123.000.000 140.000.000 176.000.000	Intel	800-900-1000-1100-1200-1330-1500-1600-1670-1830-1860-1870-1900-2000-2130-2260-	3.300		Tek ve çift çekirdekli Dâhili grafik işlemci ünitesi
Core i	2010	8/16/32/64	32 GP 80 FPU 64 MMX 128 XMM	33 / (8 GB) 35 / (32 GB)	64	624.000.000 995.000.000 2.270.000.000	Intel	1600–1730–1860–2000–2130–2300–2400–2500–2530–2600–2660–2670–2700–2800–2900–2930–3060–3100–3200–3300–3330–3400–3460–3500–3600–3700–3730–3800–3900–4000		İçinde	2, 4 ve 6 çekirdekli, faklı hız ve önbellek kapasitesi

4.3 Kişisel Bilgisayarlarda Bellek Kullanımı ve Büyüklüğü

İşlemcilerin adres yolunun kullanarak erişebileceği (adresleyebileceği) bellek, fiziksel bellek olarak adlandırılır. İşletim sistemleri de işlemcinin bellek yönetimi için sağladığı etkin ve güvenilir bellek erişim yöntemlerini (kesimli veya sayfalı erişim gibi) kullanarak belleğe erişirler. İlk üretilen bilgisayar sistemlerinde kullanılan işlemciler 8 bit veri yolu ve 16 bit adres yolu ile üretilmişlerdir. Adres yolunun 16 bit olması ile 2^{16} (65536) tane farklı, byte boyutlu bellek alanına erişim sağlanabilmiştir. Ancak, o dönemlerde bellek birimlerinin pahalı olması nedeni ile sistemlerde genelde 4-16 KB bellek kullanılmıştır. Zamanla üretim teknolojisinin gelişmesi ve bellek birimlerinin ucuzlaması ile 1980'lerde üretilen kişisel bilgisayarlarda (PC) 64 KB bellek kullanılmaya başlanmış ve bu miktarın çok uzun süre ihtiyacı karşılayacağı düşünülmüştür.[43] Sınırlı bellek kapasitesine sahip olan bilgisayar sistemleri CP/M (Control Program for Microcomputers) adı verilen bir işletim sistemi tarafından kontrol edilmiş, erişilmek istenen bellek adresi,adres yoluna, yapılacak işleme ilişkin kontrol bilgisi, kontrol yoluna konularak o adreste bulunan byte'a istenen şekilde erişim sağlanmıştır. Bu tip adresleme modeli düz bellek modeli (flat memory model[44]) olarak adlandırılır. Ancak zamanla artan ihtiyaçlar, işlemci tasarımlarında daha fazla bellek kullanımına imkân verecek değişikliklerin yapılmasını gerekli kılmıştır. Erişilebilir bellek miktarını arttırmak üzere işlemcilerin adres bacaklarının sayısı da arttırılmıştır. 20 adres bacağı ile adreslenebilen bellek miktarı 2^{20} (1 MB) büyüklüğüne erişmiştir. Bu aşamada düz bir adresleme tekniği ile bellek alanının tamamına erişmemin mümkün olmaması tasarımcıları yeni arayışlar içine itmiş ve kesimli bellek modeli (segmented memory model[45]) olarak adlandırılan yeni bir belek erişimi yöntemi doğmuştur. Gerçek kipte çalışan kesimli bellek modelinde amaç kesim ve göreli konum değerlerini barındıran 16 bitlik yazmaçlardan yararlanarak 20 bit büyüklüğünde bir fiziksel adres değeri oluşturmaktır. Kesimli bellek modelin, düz bellek modele göre en önemli avantajı, yığın alanının kontrol dışında büyümesi durumunda kod veya veri alanlarındaki değerlerin bozulmasının engellenmesidir. Tasarımı itibariyle 8088 işlemcisi 20 adres bacağı ile en fazla 1 MB bellek adresleyebilirken sistemlere 1 MB'dan daha fazla bellek eklemenin de anlamsız olacağı görülmüştür. Zaman içinde CP/M işletim sisteminin yerini DOS (Disk Operating

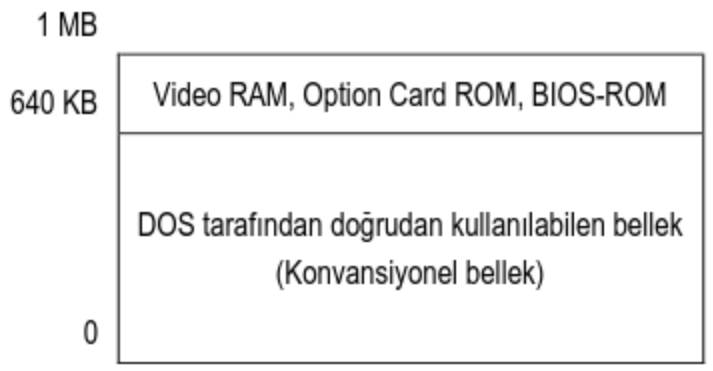
---

43 "*640 KB ought to be enough for anybody*." Microsoft Chairman Bill Gates, 1981

44 Günümüzdeki üretilen Intel işlemciler geriye doğru uyumlu olup, korumalı (protected), gerçek (real-address), sistem yönetim (system management), uyumluluk (compatibility) ve 64 bit olarak ifade edilen farklı kiplerde çalışabilmektedir. Bu kitap gerçek kipte işleyişi anlatmak üzere kaleme alınmıştır. İlerideki başlıklar altında değineceğimiz, gerçek kipte çalışan, komut, veri ve yığının aynı adres alanı içinde olduğu COM uzantılı assembly programlar da genel olarak düz modeli kullanılıyor gibi görünse de aslında bunun "gerçek kip düz bellek modeli" veya "gerçek kip bellek modeli" olarak adlandırılması daha doğrudur. Zira düz bellek model korumalı kipte de kullanılmaktadır.

45 İleriki başlıklar altında değineceğimiz, gerçek kipte çalışan, komut, veri ve yığının ayrı kesimlerde yer aldığı EXE uzantılı assembly programlar kesimli bellek modeli kullanır. 32 bit işlemcilerdeki korumalı kip kullanımında her biri 2^{32} byte büyüklüğünde olmak üzere 16.383 adet farklı kesim kullanılabilmektedir. İşlemci, kesim ve göreli konum (segment:offset) olarak belirlenen bu mantıksal adresi fiziksel adrese dönüştürmektedir.

System) isimli işletim sistemi almış ancak o da belleğin sadece 640 KB'lık kısmını doğrudan kullanabiliş, Video RAM, Option Card ROM, BIOS-ROM ise geri kalan 384 KB'lık kısmını kullanmıştır. (Şekil 4-1).



Şekil 4-1: PC Bellek haritası

4.3.1 Genişletilmiş (Expanded) Bellek

Her ne kadar işlemciler doğrudan adresleyemeseler de uygulamaların bellek ihtiyaçları sürekli artmıştır. Bu açığı kapatmak üzere ilk önceleri 32 MB kapasiteli bellek kartları tasarlanmış ve bu kartlar üzerindeki belleğe erişimi sağlamak amacıyla bazı özel teknikler geliştirilmiştir. İşlemcinin adres alanı içinde olmayan genişletilmiş bellek (expanded memory), 16 KB büyüklüğündeki pencerelere bölünmüş ve bunlar kullanılan sıra anahtarlama (bank switching) yöntemi ile 1 MB'lık bellek aralığına taşınarak erişilmeleri sağlanmıştır. EMM (Expanded Memory Management) yazılımı bu amaçla kullanılmıştır (Şekil 4-2).

4.3.2 Uzatılmış (Extended) Bellek[46]

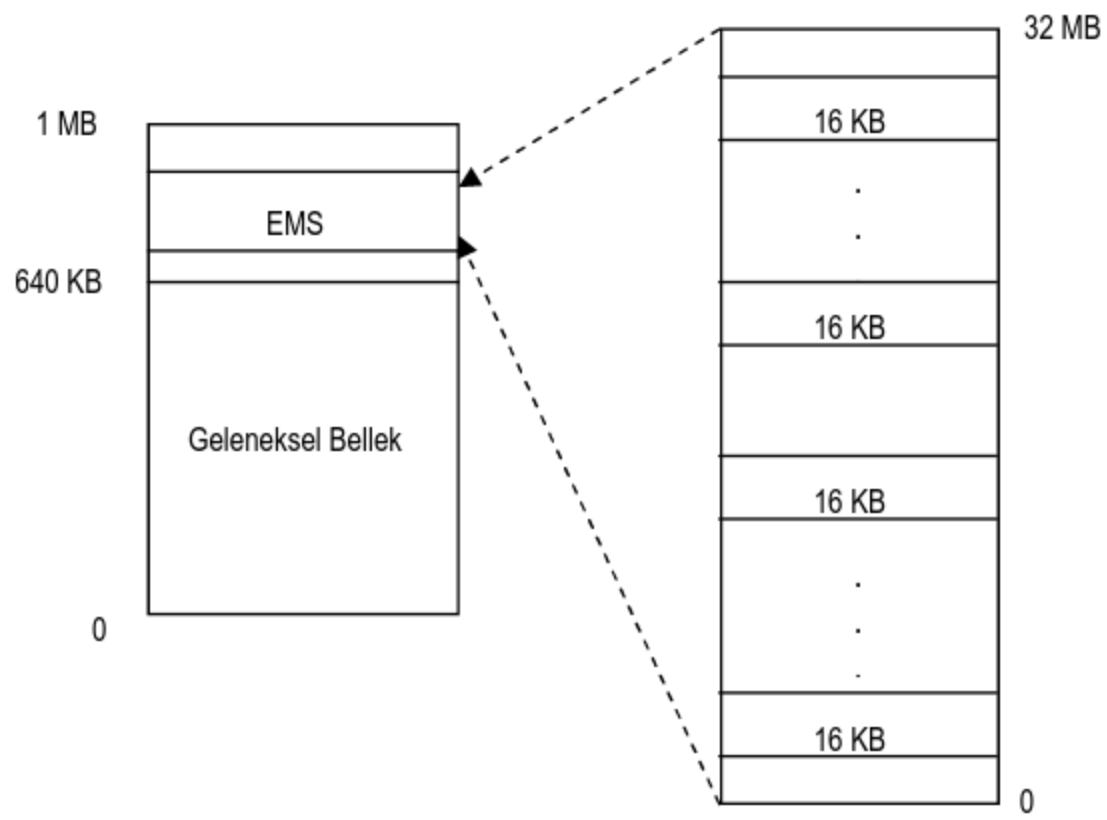
Uzatılmış bellek sınıflaması, belleğin 1 MB üzerinde kalan ve işlemcinin doğrudan adresleyebildiği bölgeyi ifade etmek üzere kullanılır. İşlemcilerin adresleyebilecekleri bellek miktarı sahip olduğu adres bacaklarının sayısı ile orantılı olup uzatılmış bellek kavramı 80286 ve üstü işlemciler için geçerlidir. 8088 veya 8086 işlemcisi, 20 adres bacağı ile 1 MB bellek adresleyebilmektedir. Bu işlemcilerin 1 MB üzerindeki belleği kendi adres bacaklarını kullanarak adresleyemez bu nedenle uzatılmış bellek kavramı bu iki işlemci için kullanılamaz. Buna göre 80286 işlemcisi için 1 MB-16 MB, 80386 işlemcisi için 1 MB-4 GB arasında kalan bölümler uzatılmış bellek olarak ifade edilir. Extended memory, DOS 4.0 ve üstü işletim sistemlerinde RAMDRIVE.SYS, SMARTDRV.SYS ve HIMEM.SYS[47]

---

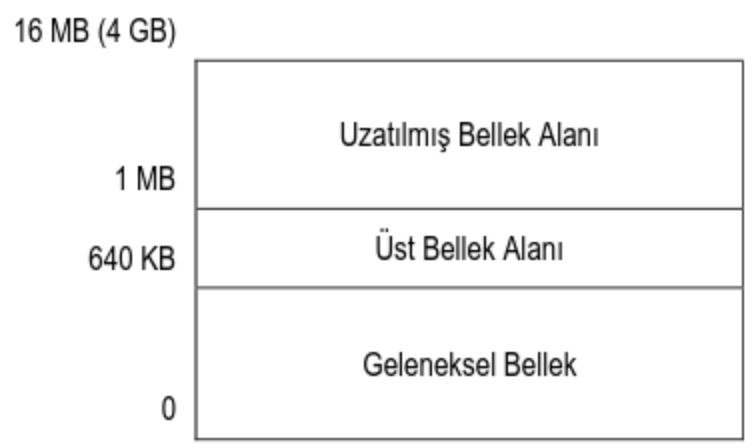
[46] Uzatılmış bellek, DOS tarafından, RAM disk, disk cache ve genişletilmiş bellek emülatörleri dışında yoğun olarak kullanılmamaktadır. Bunun nedeni alt model işlemciler (8088/8086) ile uyumlu kalabilmektir. Ayrıca 80286 işlemcilerin gerçek kipte çalışırken (yani 8088/8086 gibi) 1 MB üzerindeki belleği adresleyebilmeleri için korumalı kipe geçmeleri gereklidir. Oysa bu işlemcide korumalı ve gerçek kip arası geçişler problemlidir.

[47] HIMEM.SYS, Microsoft, Intel, Lotus ve AST tarafından belirlenen XMS (eXtended Memory Specifications) 2.0 uyarlamasına uygun olarak uzatılmış belleğin kullanılmasını sağlamaktadır.

isimli aygıt sürücüleri (device driver) yardımıyla kullanılır ve yönetilir (Şekil 4-3). Diğer yandan yeni nesil işletim sistemleri (Windows NT, W2K, Linux gibi.) korumalı kip düz model (protected mode flat model) adı verilen bir bellek erişim tekniği kullanarak uzatılmış bellek alanlarına erişmektedir. Doğal olarak işlemcilerde de bu özelliği desteklemek için gerekli tasarım değişiklikleri (yazmaçlar ve bayraklara eklemeler) yapılmıştır.



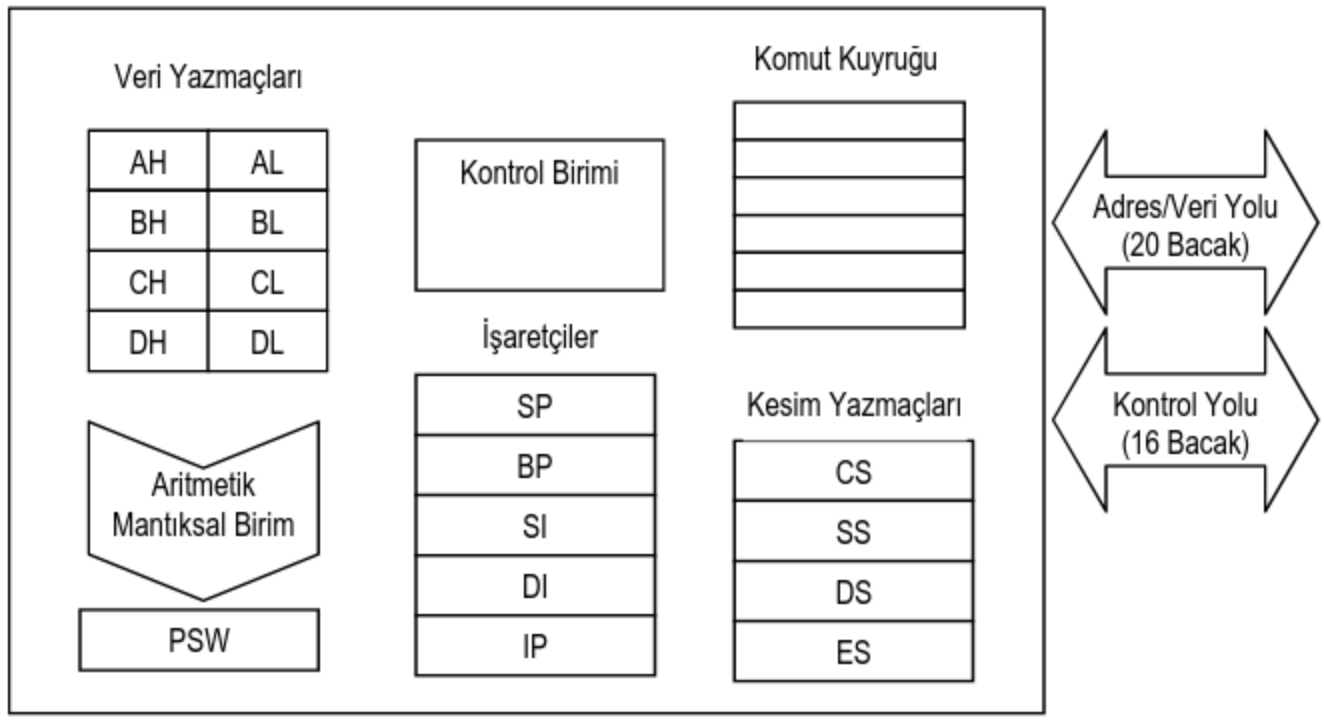
Şekil 4-2: Genişletilmiş bellek yapısı



Şekil 4-3: Uzatılmış bellek haritası

4.4 8086 İşlemcisinin İçyapısı

8086 işlemcisi, işlemleri 3 temel grup altında toplanmış yazmaçları(register)[48] kullanarak gerçekleştirmektedir. Bu yazmaçlar veri (data), kesim (segment) ve işaretçiler (pointer) olarak gruplandırılmıştır. İşlemci içinde 6 byte uzunluğunda komut kuyruğunun (instruction queue) yanı sıra aritmetik ve mantıksal işlemleri yapan bir ünite (ALU-Arithmetic Logic Unit) ve genel işleyişten sorumlu olan kontrol birimi mevcuttur. Yapılan işlemlerin sonuçlarına göre farklı durumları ifade etmek amacıyla kullanılan ve bayrak (flag) olarak adlandırılan değerler, PSW (Program Status Word) isimli yazmaçta saklanmaktadır. 8086 işlemcisinin basitleştirilmiş içyapısı Şekil 4-4'de verilmiştir.



Şekil 4-4: 8086 işlemcisinin basitleştirilmiş içyapısı

4.5 Kesim (Segment) Yapısı

8086 işlemcisinde 20 adres bacağı olmasına rağmen en büyük yazmacı 16 bittir. Bir tek yazmaç üzerindeki değer ile ancak 64 KB'lık bir alanı adresleyebilmektedir. Bu durumda 20 adres bacağı ile adresleyebilmesi mümkün olan 1 MB'lık bellekten yararlanabilmek için kesimli bellek erişim[49] tekniği (segmented memory management) ortaya atılmıştır. 80x86 ailesinde ana bellek her biri çift adresten başlamak üzere[50] 64

---

[48] Yazmaç, küçük kapasiteli (bit ile ifade edilen) ancak çok hızlı bir bellek birimidir. İşlemci, işlemlerini bu özel nitelikli bellek birimlerini kullanmak üzere tasarlanmış olan komutları kullanarak gerçekleştirir.

[49] Intel firması daha önce ürettiği 8080 ve 8085 işlemcileri ile uyumluluğu sağlayabilmek amacıyla kesim yapısını daha üst model işlemcilerinde de kullanmıştır.

[50] Gerçek kipte kesimlerin genel olarak on altıya tam bölünebilir bir adresten başlaması (paragraf boundary) tercih edilir. Bu kesim tanımlamalarında kullanılan PARA seçeneği ile sağlanır.

KB'lık kesimlere (segment) ayrılmıştır. Kesim yazmaçları ve kesim içi işaretçileri sayesinde 80x86 ailesi işlemcileri belleğin tümünü kullanabilmektedir. Temel olarak 4 tane kesim çeşidinden söz etmek mümkündür.

Herhangi bir kesime erişmek ancak o alanın kesim yazmacını kullanmakla mümkündür. Bu kesim yazmaçları **CS**, **SS**, **DS** ve **ES** olarak adlandırılır. Kesim içerisindeki verilere ise kesim içi işaretçilerle ulaşılmaktadır. Kesim içi işaretçiler göreli konum olarak adlandırılan, kesimin başlangıcından olan uzaklığı belirleyen değerleri tutar. Bunu bir cadde üzerindeki binaları belirlemek için kullanılan numaralar (kesim değerleri) ile bina içindeki daireleri belirlemek için kullanılan kapı numaralarına (kesim içi değeri) benzetmek mümkündür.

4.5.1 Kod (Code) Kesimi

Bilgisayarların herhangi bir uygulamayı çalıştırılabilmesi için uygulamayı oluşturan programın (derlenmiş kodun) bellek üzerine yerleştirilmiş olması gereklidir. Buna göre çalıştırılması istenilen komutların bulunduğu bellek alanı kod kesimi olarak değerlendirilir. Kod kesiminin başlangıç adresi **CS** yazmacı ile belirlenmektedir. Bu alan içerisinde hangi komutun işleneceğini gösteren bir işaretçi vardır. Bu işaretçi **IP** (Instruction Pointer) olarak adlandırılır. Çalıştırılması istenilen komutun yerini **CS:IP** çifti belirler.[51] IP yazmacındaki değer, komutun işletilmesini takiben bir sonraki komutun başlangıç adresini gösterecek şekilde otomatik olarak değişir. CISC (Complex Instruction Set Computers) mimari modeli uyarınca tasarlanmış olan 80x86 serisi işlemcilerde her komutun kod kesimi içinde işgal ettiği bellek miktarı farklıdır. IP yazmacının değeri de buna bağlı olarak çalıştırılan her komutun sonunda otomatik olarak değişmektedir. CS ve IP yazmaçlarının değerlerinin program içerisinde gelişi güzel değiştirilmesi sonucu izlenmesi gereken akış değişeceğinden hataya, hatta sistemin kilitlenmesine neden olabilir. Buna rağmen yazılan program içinden başka bir yordamın (procedure) çağırılması (**CALL**) veya kullanılan koşullu (**Jxx**), koşulsuz (**JMP**) dallanma ve çevrim **(LOOP)** komutları ile kullanıcı kendi yazdığı program içerisindeki akışı kontrol edebilir. Bunlara ek olarak, sistemin çalışması sırasında veya çevre birimlerden gelen istekleri işlemciye iletmek için oluşan kesmeler de (**INT**) CS ve IP yazmaçlarının değerlerinin değişmesine neden olacaktır. Ancak bu tip değişiklikler, uygulamanın veya sistemin uyum içinde çalışmasını sürdürebilmesi amacıyla yapılan kontrollü değişiklikler olup doğru parametreler ile kullanılmaları durumunda hiçbir bir probleme sebep olmazlar.

---

[51] Bir adresi belirlemek için bina numarası ve daire numarasından oluşan ikiliden yararlanıldığı gibi bilgisayarın belleğindeki bir alana erişmek için de Kesim:Göreli konum (Segment:Offset) olarak adlandırılan çiftten yararlanılmaktadır. Herhangi bir bellek alanı farklı Kesim ve Göreli konum değerleri ile belirlenebilir. Örneğin 0040:006Ch ile 0000:046Ch aynı bellek alanını gösterir. Detayları için Gerçek Kipte Fiziksel Adres Hesabı konusuna bakınız.

4.5.2 Yığın (Stack) Kesimi

Yığın, son giren ilk çıkar (LIFO-Last In First Out) kuralına göre çalışan bir bellek alanı olarak tanımlanabilir. Yığın kesiminin başlangıç adresi SS yazmacı ile belirlenmektedir. Bu kesim, işletim sistemine dönebilmek için gerekli dönüş parametrelerinin saklanması ve programın çalışması sırasında kullanılan **PUSH** komutlarının kullanımı sonucu oluşan geçici veri saklama ihtiyacını karşılamak üzere ayrılmıştır. Bu kesim içine konulacak verinin nereye yerleştirileceğini veya nereden alınacağını belirleyen işaretçiye **SP** (Stack Pointer) adı verilmektedir. SP yazmacı, 8086 işlemcisinde yığın kesiminin en üstündeki dolu yeri gösterirken, 80286/386/486 işlemcilerde en üstte bulunan boş yeri gösterir. Yığın büyük adresten küçük adrese doğru ilerlediği için **PUSH** komutu ile yığına bir word (2 byte) büyüklüğünde veri yerleştirildiğinde SP yazmacının değeri kendiliğinden 2 byte azalacaktır. Benzer şekilde yığından **POP** komutu ile bir word büyüklüğünde veri alındığında ise SP yazmacının değeri kendiliğinden 2 byte artacaktır[52].

SP yazmacının değerinin PUSH ve POP komutlarının dışındaki bir yol ile değiştirilmesi son giren ilk çıkar şeklinde çalışan bu bellek kesimindeki verinin konulan sırada çekilmesini engelleyeceği için problemlere neden olur. Eğer yığın kesimi içerisinde, yığın içi dizilişi engellemeksizin herhangi bir yerdeki veriye erişilmek isteniyorsa bunun için **BP** (Base Pointer) yazmacı kullanılmalıdır. BP yazmacı yardımıyla yığın kesimi içindeki herhangi bir değere erişerek onun bir kopyasını almak mümkündür. Ancak bu işlem kesinlikle yığın yapısından bir verinin çekilip alınması, boşalan yerin diğer elemanların kaydırılması/sıkıştırılması ile doldurulması anlamını taşımaz.

Kod kesiminde yer alan program içinde kullanılan ve akışı değiştirme özelliği olan bazı komutlar (**CALL** ve **INT** gibi) yığın kesimini bazı özel değerleri saklamak için kullanılır. Yığın kesiminde saklanan değerler genellikle program akışını değiştiren veya kesen işlemin bitirilmesini takiben çalıştırılması gereken bir sonraki komut satırının adresi[53] ve bazen buna ek olarak akış kesilmeden hemen önce bayraklarda bulunan değerlerdir. Adres ve bayrak değerleri kullanılan komutun (**CALL** veya **INT**) bir sonucu olarak sistem tarafından otomatik olarak yığına yerleştirilir. Yığına yerleştirilen bu değerler yine program içerisinde yer alan bazı komutlar yardımıyla (**RET** veya **IRET**) uygun biçimde geri alınırlar. Yığın üzerinden doğru verinin alınamaması sistemin çalışmasını hiç şüphesiz etkileyecektir. Matematiksel formüllerde kullanılan parantezlerin dizilişleri nasıl işlem öncelikleri nedeniyle

---

[52] 8086 işlemcisinde PUSH ve POP komutları ile yığına bir seferde sadece bir word (2 byte) konabilir veya alınabilir. SP yazmacının değeri PUSH veya POP komutları tarafından otomatik olarak ayarlanır. İlk nesil işlemcilerde (8086/8088) SP'nin yığının üstündeki dolu gözü gösterirken daha sonraki dönemde üretilen işlemcilerde (80286/80386/80486 ve ötesi) SP yığının üstündeki boş gözü göstermektedir. Bu durum ilk bakışta bir uyumsuzluk olduğu izlenimini verse de işlemci tasarımı sırasında (mikro kod seviyesinde) yapılan düzenleme ile uyum sağlanmıştır. Ayrıntılar için 5.10. bölümü inceleyiniz.

[53] Adres değeri, kullanılan komutun tipine göre 2 veya 4 byte olacaktır.

formülün vereceği sonucu belirlemekte hayati önem taşıyorsa, yığın üzerinde kullanıcı tarafından gerçekleştirilen işlemlerin sayısı ve sırası da o derece önemlidir.

4.5.3 Veri (Data) Kesimi

Kod kesiminde programların yer aldığı gibi, programların üzerinde işlem yapacağı veri de bellekteki veri kesiminde bulunur. Buna göre veri kesiminde, kullanıcı tarafından tanımlanmış giriş değerleri ile programın çalışması sonucunda üreteceği sonuç değerleri yer alacaktır. Programcı, giriş ve çıkış değerlerinin sayısına ve tipine (byte, word vb.) bağlı olarak veri kesiminde gerekli tanımları yapmaktan sorumludur. Veri kesiminin başlangıç adresi DS yazmacı ile belirlenmekte, kesim içi adreslemelerde ise SI, DI ve gerekmesi durumunda BX yazmaçlarından yararlanılmaktadır.

4.5.4 Ek (Extra) Kesim

Kullanıcının işleyeceği verinin mevcut veri kesiminin (DS) kapasitesinden fazla veya aynı anda iki farklı veri kesimine erişiminin gerekli olduğu durumlarda ek kesimden yararlanılmaktadır. Ek kesim özellikle yükleme (store, move) ve dizgi hareketleri (string move) ile ilgili komutların gerçekleştirilmesi sırasında devreye girmektedir. Ek kesiminin başlangıç adresi ES yazmacı ile belirlenmekte, kesim içi adreslemelerde öncelik sırasıyla DI, SI ve BX yazmaçlarından yararlanılmaktadır.

4.6 Yazmaç (Register)

Yazmaç, işlemci içinde, belirli bir amaç için tasarlanmış, sınırlı kapasiteli, hızlı veri saklama ve/veya işleme devresidir. Her yazmacın farklı türde özellikleri vardır. Yapılan işlemin gereklerine göre byte veya word (işlemcinin özelliğine bağlı olarak double word) olarak kullanılabilirler. Yazmaçlar, bellek hiyerarşisinin en üstünde yer aldığı için diğer bellek birimleri ile karşılaştırıldığında en hızlı bellek birimidir.

Yazmaçlar üstelendikleri görevlere göre gruplandırılırlar (Tablo 4-5). Buna göre 8086 ailesinin sahip olduğu yazmaçlar veri, işaret, sıra, kesim ve bayrak grubu olmak üzere beş alt grup altında incelenebilir.

4.6.1 Veri Grubu Yazmaçları

Veri grubu yazmaçları **byte** (8 bit) veya **word** (2 byte–16 bit) olarak kullanılabilirler. Yazmaçların hangi şekilde kullanılacağı ihtiyaca bağlı olarak değişir. 80386/486 işlemciler 32 bit olup **double word** yazmaçlara sahiptir. Bu yazmaçlar duruma göre 32, 16 veya 8 bit olarak kullanılabilir. 8 bit işlemlerde AL, AH, BL, BH, CL, CH, DL, DH olarak, 16 bit işlemlerde AX, BX, CX, DX olarak (Tablo 4-5), 32 bit işlemlerde EAX, EBX, ECX, EDX olarak isimlendirilerek kullanılırlar. Ayrıca 80386/486 işlemcilerde farklı amaçlar için kullanılan yazmaçlar da mevcuttur (Tablo 4-8).

Tablo 4-5: 8086 ailesi yazmaçları

	15 … 8	7 … 0	
AX	AH	AL	Veri Grubu
BX	BH	BL	
CX	CH	CL	
DX	DH	DL	
	SP		İşaret Grubu
	BP		
	IP		
	SI		Sıra Grubu
	DI		
	CS		Kesim Grubu
	DS		
	SS		
	ES		

4.6.1.1 AX: Accumulator

Giriş/Çıkış (I/O) işlemlerinde, aritmetik, dizgi (string) işlemlerinde kullanılır. 16 bit tek parça AX olarak kullanılabildiği gibi 8 bit'lik iki parça, AH ve AL olarak da kullanılabilir. Bazı komutlarda AX (AH[54] veya AL[55] olarak) gizli işlenen (operand) durumundadır. İşlenen olarak yazmaçları kullanan bazı komutlar, işlenen olarak AX yazmacı kullanılması durumunda daha az yer kaplayacak ve daha hızlı çalışacak şekilde iyileştirilmiştir. Gerek çok çeşitli işlemde işlenen olması gerek ise kullanıldığı işlemlerin daha iyi sonuç vermesi nedeniyle AX yazmacı en çok kullanılan yazmaçtır.

4.6.1.2 BX: Base Register

Genel amaçlı hesaplamalarda ve dizi şeklindeki veri yapılarına erişmek için indis olarak da kullanılmaktadır. 16 bit tek parça, BX olarak kullanılabildiği gibi 8 bit'lik iki parça, BH ve BL olarak da kullanılabilir.

4.6.1.3 CX: Count Register

Çevrim sayılarını belirtmek üzere kullanılır. Kaydırma (shift) ve döndürme (rotate) işlemlerinde tekrar sayısını belirler. 16 bit tek parça, CX olarak kullanılabildiği gibi 8 bit'lik iki parça, CH ve CL olarak da kullanılabilir.

---

[54] H (High), yüksek anlamlı kısmı ifade etmektedir. Buna göre AH, AX yazmacının yüksek anlamlı byte'ını ifade eder.

[55] L (Low), düşük anlamlı kısmı ifade etmektedir. Buna göre AL, AX yazmacının düşük anlamlı byte'ını ifade eder.

4.6.1.4 DX: Data Register

Giriş/Çıkış işlemlerinde, çarpma gibi büyük sayıların oluşabileceği (AX ile birlikte, DX:AX ikilisi olarak), bölme gibi bölüm ve kalan şeklinde iki ayrı sonucun oluştuğu işlemlerde (AX bölüm, DX kalan değerini saklayacak şekilde) kullanılır. 16 bit tek parça, DX olarak kullanılabildiği gibi 8 bit'lik iki parça, DH ve DL olarak da kullanılabilir.

4.6.2 İşaret Grubu Yazmaçları

Tanımlı oldukları kesim içindeki göreli konum değerlerini göstermek amacıyla kullanılırlar. Genel olarak kullanılan komutların sonucunda otomatik olarak değer değiştirirler. Bu nedenle işaret grubuna dâhil olan yazmaçların (BP yazmacı hariç) değerlerinin kullanıcı tarafından değiştirilmesi alışılmış bir durum değildir. IP yazmacının değerindeki değişiklik doğrudan çalıştırılacak komutu, SP yazmacının değerindeki değişiklik ise yığın üzerinde yapılacak işlemi etkileyeceği için kullanıcı tarafından bilinçsiz şekilde değiştirilmeleri beklenenin dışında durumların oluşmasına, programın kontrolden çıkmasına veya sistemin kilitlenmesine neden olabilir.

4.6.2.1 SP: Stack Pointer

SP yazmacı yığın üzerinde yapılan işlemlerde, verinin nereye yerleştirileceğini veya nereden alınacağını belirlemektedir. Yığına erişim PUSH (verinin yığına yerleştirilmesi) ve POP (verinin yığından alınması) komutları yardımı ile yapılmakta ve bu komutların sonucu olarak SP yazmacının değeri sırasıyla azaltılmakta veya artırılmaktadır. 8086 işlemcisinde yığın alanı üzerinde word büyüklüğündeki veri ile işlem yapıldığı için SP yazmacının değeri iki artar veya azalır.

4.6.2.2 BP: Base Pointer

BP yazmacı yığından istenen verilere erişimi kolaylaştırmak amacıyla kullanılmaktadır. SP yazmacı gibi yığın üzerinde kullanılmasına karşın, ondan farklı olarak değeri kullanıcı tarafından belirlenebilmektedir. Bu sayede kullanıcı yığın üzerindeki herhangi bir veriye rahatlıkla BP yazmacını kullanarak ulaşabilir[56]. Bu özelliği nedeniyle farklı program parçaları arasında yığın kullanılarak yapılan parametre aktarımlarında parametrelere erişmek için kullanılmaktadır.

4.6.2.3 IP: Instruction Pointer

IP yazmacı işlemcinin o anda çalıştıracağı komutun kod kesimi içindeki yerini belirlemek üzere kullanılmaktadır. IP yazmacının değeri işletilen komutun uzunluğuna bağlı olarak değişecektir. Ayrıca programın içindeki farklı noktalara ulaşmak için kullanılan koşullu/koşulsuz dallanma (**Jxx/JMP**), alt program çağırma (**CALL**) ve kesme (**INT**) gibi komutlara bağlı olarak da IP yazmacının değeri değişebilir.

---

[56] Yığın üzerindeki veriye erişmek demek onu oradan çekip çıkartmak olarak düşünülmemelidir. BP yazmacı sadece o alana erişmeye ve o verinin ne olduğunu görmeye, kopyasını almaya veya değiştirmeye yardımcı olur. Yığınının boşaltılması veya doldurulması ile ilgili bir işlem gerçekleştirmez.

4.6.3 Sıra Grubu Yazmaçları

İşaret grubu yazmaçlarından SP ve BP yazmaçları, yığın kesimi, IP yazmacı ise kod kesimi içerisinde kullanılmaktadır. Benzer şekilde sıra grubu yazmaçları da veri kesimi içinde[57] kullanılır. İşaretçi grubu yazmaçlarından farklı olarak değerleri kullanıcı tarafından belirlenir.

4.6.3.1 SI: Source Index

SI yazmacı, veri kesiminde tanımlanmış veriye ulaşmak için göreli konum değeri olarak kullanılmaktadır. Dizgi (string) işlemlerinde ise kaynak noktasının göreli konumunu göstermekten sorumludur.

4.6.3.2 DI: Destination Index

DI, SI yazmacına benzer olarak veri kesiminde tanımlanmış veriye ulaşmak için kullanılabildiği gibi eğer var ise ek kesim içindeki veriye de erişmek için kullanılmaktadır. Dizgi işlemlerinde ek kesim içindeki varış noktasının göreli konumunu göstermekten sorumludur.

4.6.4 Kesim Grubu Yazmaçları

Kesim grubu, tanımlı kesimlerin başlangıç adreslerini göstermek üzere kullanılan 16 bit'lik yazmaçlardan oluşmaktadır. Herhangi bir verinin bellek üzerindeki yeri, kesim ve göreli konum değerleri yardımıyla belirlenmektedir. Göreli konum değeri verinin kesim başından kaç byte ileride olduğunu gösteren değerdir. Buna göre göreli konum değeri, kesim içindeki bütün bellek gözlerine erişimi sağlamalı, diğer bir anlatım ile kesim büyüklüğünü belirlemelidir. 8088/8086 işlemcilerinde göreli konum değerlerini tutan IP, SP, BP, SI ve DI yazmaçları 16 bit oldukları için kesim büyüklüğü de 2^{16} yani 64 KB olacaktır. Diğer yandan, 80386 gibi 32 bit'lik yazmaçlar ile göreli konum değerlerinin belirlendiği hallerde kesim büyüklüğü 2^{32} yani 4 GB olacaktır. Ancak bu büyüklükteki kesimler sadece korumalı kipte kullanılabilirler. Gerçek kipteki uygulamalarda 8088/8086 ile daha üst model işlemci (mesela 80386) arasında kesim organizasyonu açısından farklılık yoktur. Kesimler ve her kesim içindeki veriye erişmek için gerekli göreli konum değerini belirleyen işaret ve sıra grubu yazmaçları Tablo 4-6'de gösterilmiştir.

Tablo 4-6: Kesim / Göreli konum ikilileri

	Kesim	Göreli Konum
Komut adresleme	CS	IP
Yığın adresleme	SS	SP
	SS	BP
Veri adresleme	DS	SI, DI ve BX
	ES	DI, SI ve BX

[57] SI ve DI yazmaçlarının yanında BX yazmacı da veri kesimi erişimlerinde göreli konum belirlemek üzere kullanılmaktadır.

4.7 Bayraklar (Flags)

İşlemci üzerinde veri, işaret, sıra ve kesim grubu dışında bir yazmaç daha mevcuttur. Bu yazmaç (PSW), üzerinde bayrak (flag) olarak adlandırılan birer bit'lik özel değerleri tutmaktadır. Bayraklar bazı işlemlerin sonucunda değer değiştirerek sonucu yansıtırlar. 8086 işlemcisinde dokuz adet bayrak vardır. Bayraklar bir bit ile ifade edilmeleri sebebiyle 0 veya 1 değerini alabilirler. Bayrağın değerinin 1 olması durumu **Set** (birleme), 0 olması durumu ise **Clear** (silme) veya **Reset** (ilk durumuna döndürme – sıfırlama) olarak adlandırılır.

Tablo 4-7: 8086 ailesi bayrakları (PSW)

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

4.7.1 CF: Carry Flag

8'li ve 16'lı işlemlerde oluşan elde/ödünç durumunu kullanıcıya belirtmek üzere CF bayrağının değeri 1 yapılır.

4.7.2 PF: Parity Flag

İşlem sonucunda çift eşlik (even parity) var ise (yani 1'lerin sayısı çift ise) durumu kullanıcıya belirtmek için PF bayrağının değeri 1 yapılır. Eğer bit dizilişindeki 1'lerin sayısı tek ise (odd parity) PF bayrağının değeri 0 olur.

4.7.3 AF: Auxilary Carry Flag

8'li işlemlerde düşük anlamlı 4'lüden (nibble) yüksek anlamlı 4'lüye elde/ödünç, 16'lı işlemlerde düşük anlamlı 8'liden yüksek anlamlı 8'liye elde/ödünç aktarılması durumunu kullanıcıya belirtmek üzere AF bayrağının değeri 1 yapılır.

4.7.4 ZF: Zero Flag

İşlem sonucunun sıfır çıkması (eşitlik olması) durumunu kullanıcıya bildirmek üzere ZF bayrağının değeri 1 yapılır.

4.7.5 SF: Sign Flag

İşlem sonucunda sayının negatif olduğunu (en anlamlı bit 1 ise) kullanıcıya belirtmek üzere SF bayrağının değeri 1 yapılır.

4.7.6 TF: Trap Flag

Adım bayrağı olarak adlandırılır. TF bayrağının 1 değeri alması durumunda her komut yürütüldükten sonra bir kesme üretilerek durumun kullanıcı tarafından izlenmesine imkân sağlanır (single-step mode).

4.7.7 IF: Interrupt Flag

IF, maskelenebilir kesme (maskable interrupt) isteklerinin izinli olup olmadığını belirlemek üzere kullanılmaktadır. IF bayrağının değeri 1 ise kesmelere izin verilecektir. IF bayrağının sıfır olması durumunda INT komutu kullanılarak üretilen her türlü maskelenebilir kesme devre dışı kalacaktır.

4.7.8 DF: Direction Flag

Dizgi işlemlerinde işlemin yüksek adresli bellek gözünden başlayarak düşük adresli bellek gözüne doğru yürütülebilmesi için DF bayrağının 1 değerini alması gereklidir. Aksi halde işlem düşük adresten başlayarak yüksek adrese doğru gerçekleştirilecektir.

4.7.9 OF: Overflow Flag

Aritmetik taşma olması halinde durumu kullanıcıya belirtmek üzere OF bayrağının değeri 1 yapılır. Aritmetik taşma işaretli sayılar ile yapılan işlemlerde elde edilen işaretli sonucun tanımlı alana sığmaması (taşması) durumunu belirlemektedir.

4.8 16 / 32 Bitlik İşlemciler Arasındaki Farklılıklar

16 bitten büyük işlemciler gerçek ve korumalı olmak üzere farklı iki çalışma kipini destekler[58]. 32 bitlik işlemcilerde gerçek kip kullanıldığında hızlı bir 8086 işlemcisi gibi çalışmakta ve yazmaçları aksi belirtilmediği sürece 16 bit'lik olarak kullanmaktadır. Ancak özel olarak 32 bitlik kod üretilmesi halinde işlemci üzerinde çalışan programlar 32 bit'lik mimariden faydalanabilmektedir. 32 bit korumalı kipte ek yazmaç ve bayraklar da devreye girmektedir. 32 bitlik işlemcilerin sahip olduğu yazmaçlar Tablo 4-8'da gösterilmiştir.

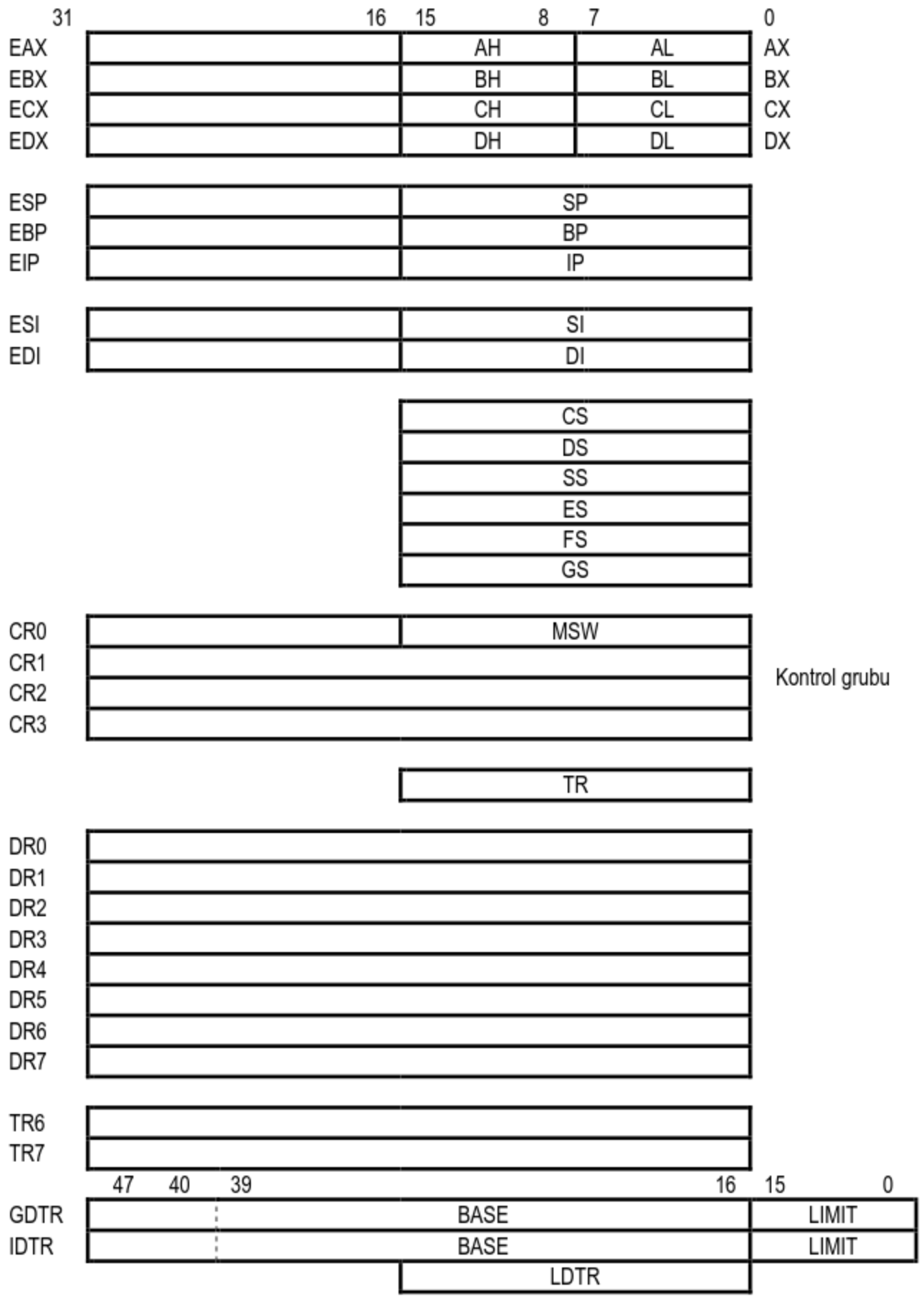
Tablo 4-8'dan da görüleceği gibi 16 bitlik işlemcide kullanılan yazmaçlar 32 bitlik hale gelmişler ve genel olarak isimlerinin başına **E** (Extended) harfi eklenerek yeniden isimlendirilmişlerdir. 8 ve 16 bit işlemlerde daha önceki işlemciler ile tamamen kod uyumlu olarak çalışmaktadırlar. Kesim yazmaçları, GS ve FS yazmaçlarının eklenmesi ile dörtten altıya çıkmıştır[59]. 16 bitlik işlemcideki bayraklara Tablo 4-9'de verilen, korumalı kipte çalışma sırasında ihtiyaç duyulan bayraklar da eklenmiştir.

---

[58] 32 bitlik işlemciler korumalı (protected), gerçek (real-address) ve sistem yönetim (system management) kipleri, 64 bitlik işlemciler ise 32 bitlik işlemcilerin kiplerine ilave olarak, uyumluluk (compatibility) ve 64 bit kiplerinde çalışırlar.

[59] FS ve GS isimlerinin özel bir anlamı bulunmaz. Ek kesim yazmacı (ES) ile benzer görevleri üstlenmelerinden ötürü alfabenin takip eden harfleri kullanılarak isimlendirilmişlerdir.

Tablo 4-8: 32 Bit işlemci – Korumalı kip yazmaçları



Tablo 4-9: 32 Bit işlemci bayrakları

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
													AC	VM	RF		NT	IOPL		OF	DF	IF	TF	SF	ZF		AF		PF		CF

4.8.1 32 Bitlik İşlemcideki Diğer Bayraklar

32 bitlik işlemcilerde kullanılan yazmaçların sayı ve büyüklüklerinde farklılıklar olduğu gibi mevcut bayrak sayılarında da farklılıklar vardır. Buna göre işlemciye eklenen yeni bayraklar ve kısa açıklamaları şöyledir;

4.8.1.1 AC: Alignment Check

AC bayrağı bir değerini almış ise belleğe erişim en hızlı şekilde (en az erişim sayısı ile) yapılır. Bellek 32 bit olarak organize edilmiş olduğu için bellek üzerindeki işlenenlere dörde tam bölünebilir bir adresten erişebilmek başarımın artmasını sağlayacaktır. Aksi durumlarda işlenenin bellekten alınması için birden fazla erişim yapılmak durumunda kalınacaktır.

4.8.1.2 VM: Virtual 8086 Mode

VM bayrağının değeri 1 iken, o anda çalıştırılmakta olan komutun 8086 komut kümesinden olduğu anlaşılır. Uygulama programları bu biti değiştiremez.

4.8.1.3 RF: Resume Flag

RF bayrağı programın çalışması sırasında herhangi bir debug hatası üretilip üretilemeyeceğini kontrol etmektedir. Programın çalışması sırasında beklenmedik bir durum oluşması halinde işlemci CS, EIP ve EFLAG yazmaçlarını yığına atarak özel durumda devreye girmesi gereken programa dallanır. Bu arada RF bayrağının değeri 1 yapılmıştır.

4.8.1.4 NT: Nested Task

Aynı anda yordam çağırma (CALL), kesme (INT) veya beklemedik bir başka durumun oluşması halinde bunların birbirini takiben çalışmasını sağlar.

4.8.1.5 IOPL: I/O - Privilege Level

IOPL bayrağı diğerlerinden farklı olarak 2 bitlik (0-3 arasında) değer tutar. Amacı giriş/çıkış komutlarını çalıştırmak için gerekli yetki seviyesini belirlemektir. IOPL bayrağının değerini sadece 0 numaralı yetki seviyesinde bulunan işlem değiştirme hakkına sahiptir.

4.8.2 Kontrol Yazmaçları

Sayfalama (paging) ve matematik yardımcı işlemci ile ilgili kontrol fonksiyonlarını ve ön bellek (cache) işlemlerini kontrol etmek üzere kontrol yazmaçlarından yararlanılır. Bunlar;

4.8.2.1 CR0: Control Register 0

CR0 yazmacı kendi içindeki bazı bit alanlarının değerlerine göre işlem yapmaktadır.

Tablo 4-10: Control Register - 0

31	30	29									18		16										5	4	3	2	1	0
PG	CD	NW									AM		WP										NE	ET	TS	EM	MP	PE

- **PG** (Paging): PG biti bir yapılarak sayfalama (paging) imkânı sağlanır.
- **CD** (Cache Disable): Bir olduğu zaman ön bellek kullanımı engellenir ve ön bellek dışındaki bir bellek alanından yapılan istekler ön belleğe yazılmazlar.
- **NW** (No Write Through): CD değeri ile aynı değere ayarlanmalıdır. Ön bellek kullanımının engellendiği zaman 1, aksi halde 0 değerini alır.
- **AM** (Alignment Mask): Değeri 1 olduğu zaman EFLAG da bulunan AC bitini aktif hale getirir. Değeri 0 ise AC değerinin bir önemi yoktur.
- **WP** (Write Protect): Sayfalamanın nasıl çalışacağını belirler.
- **NE** (Numeric Exception): Değeri 1 olduğu zaman kayan noktalı işlemlerde meydana gelecek beklenmedik durumların maskelenmesini engeller.
- **ET** (Extention Type): Matematik işlemci var ise değeri 1 olur.
- **TS** (Task Switched): değeri 1 olduğunda takip eden bir sonraki matematik işlemci gerektiren işlem, işletim sisteminden servis alır. Bu sayede ana işlemcinin, matematik işlemcinin durumunu saklamaksızın çok görevli (multitasking) çalışmasına yardımcı olur.
- **EM** (Emulate Math Processor): Değeri 1 olduğu zaman normalde matematik işlemci tarafından yerine getirilmesi gereken işlemler için işletim sisteminden servis alınır. İşletim sistemi ise yapılması gerekli işlemi yazılım olarak gerçekleştirir.
- **MP** (Monitor Co processor): Durumu, WAIT komutunun durumunu etkiler.
- **PE** (Protect Enable): İşlemciyi korumalı kipte çalışmaya geçirmek için kullanılır. Bu işlem genellikle ilk değer atama sırasında yapılır. İşlemci bir kere korumalı kipe geçirildikten sonra gerekli olması durumunda korumalı kip ile gerçek kip arasında anahtarlanabilir.

4.8.2.2 CR1: Control Register 1

80386 ve 80486 da kullanılmamıştır. Daha ileri Intel işlemcileri için kullanılmak üzere tasarlanmıştır.

4.8.2.3 CR2: Control Register 2

Sayfa hatası (Page Fault) meydana geldiği zaman CR2 yazmacı hataya neden olan fiziksel (lineer) adresi tutmaktadır.

4.8.2.4 CR3: Control Register 3

Sayfalama donanımı aynı zamanda CR3 yazmacını da kullanmaktadır. CR3 yazmacı sayfa başlangıç adresinin yüksek anlamlı 20 bitini tutmakla görevlidir.

4.8.2.5 Debug ve Test Yazmaçları

8 tane debug yazmacı (DR0-DR7) ve iki tane de test yazmacı (TR0-TR1) vardır. 80486 işlemcisinde TR3-TR5 ön belleğin test ve kontrolü, TR6-TR7 ise TLB'nin (translation look aside buffer) testi için kullanılmaktadır.

4.9 64 Bit İşlemci Yazmaçları

64 bit işlemcilerde 16 adet genel amaçlı yazmaç bulunmaktadır. 32 bitlik işlenenler ile çalışılması gerektiğinde 32 bitlik işlemcilerde bulunan EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP yazmaçlarına ek olarak R8D-R15D[60] şekilde adlandırılan 8 tane daha ek yazmaç kullanılır. 64 bit işlenenler ile çalışılması gerektiğinde ise yazmaçların basına **R** ön eki eklenerek RAX, RBX, RCX, RDX; RDI, RSI, RBP, RSP ve R8-R15 isimli yazmaçlar kullanılmalıdır. İlk 8 yazmacın başındaki R harfi E harfi ile değiştirildiğinde düşük anlamlı 32 bite erişim (EAX kullanılarak RAX'in düşük anlamı 32 bit'e), R harfi kaldırılarak düşük anlamlı 16 bite (AX ile RAX'in düşük anlamlı 16 bit'e) erişmek mümkündür. Ayrıca ilk dört yazmacın başındaki R harfini kaldırıp sonuna H veya L harflerini koyarak yüksek ve alçak anlamlı 8 bite (AH veya AL ile RAX'in yüksek ve düşük anlamlı bit'lerine) erişim sağlanmaktadır. Yeni eklenen R8-R15 isimli yazmaçlarda ise R8 (qword-64 bit) R8D (R8'in düşük anlamlı dword-32 bit), R8W (R8'in düşük anlamlı word-16 bit) ve R8L (R8'in düşük anlamlı low byte-8 bit) değerine erişmek mümkündür. R8H şeklinde bir kullanım olmadığı gibi R8'in yüksek anlamlı 32 bit'i, R8D'nin yüksek anlamlı 16 bit'ine erişim imkânı da yoktur.

64 bitlik işlemcilerin 64-bit çalışma kipinde byte yazmaçlara erişimlerde bazı sınırlamalar vardır. Hiçbir komut, miras olarak önceki nesil işlemcilerden gelen yazmaçların yüksek byte'ı (AH, BH, CH ve DH) ile yeni byte yazmaçları (RAX'in düşük anlamlı byte'ı R8L) aynı anda kullanamaz. Buna rağmen daha önceki nesilden

---

[60] R8D, R9D, R10D, R11D, R12D, R13D, R14D ve R15D olarak isimlendirilen bu yazmaçlarda başta bulunan R(egiter), sondaki D(ouble-32 bit) manasında kullanılmaktadır. Arada kullanılan sayı ise o yazmacı belirlemektedir.

miras kalan yazmaçların düşük anlamlı byte'ı (AL, BL, CL ve DL) ile yeni byte yazmaçlar (R8L) ile kullanılabilir. Bu sınırlamayı ortadan kaldırmak üzere kullanılan REX ön eki ile AH, BH, CH ve DH sırası ile BPL, SPL, DIL ve SIL'ye dönüştürülür.[61]

Tablo 4-11: 64 Bit çalışmada yazmaç kullanımı

Yazmaç Tipi	REX Kullanılmadan	REX Kullanılarak
Byte (8 bit)	AL, BL, CL, DL AH, BH, CH, DH	AL, BL, CL, DL, BPL, SPL, DIL, SIL, R8L-R15L
Word (16 bit)	AX, BX, CX, DX DI, SI, BP, SP	AX, BX, CX, DX DI, SI, BP, SP R8W-R15W
DWord (32 bit)	EAX, EBX, ECX, EDX EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX EDI, ESI, EBP, ESP R8D-R15D
QWord (64 bit)		RAX,RBX, RCX, RDX RDI, RSI, RBP, RSP R8-R15

4.10 Belleğe Erişim

İşlemcilerinin adresleyebilecekleri bellek miktarı, adres bacaklarının sayısına bağlı olarak değişmektedir. 20 adres bacağı ile 2^{20} yani 1 MB adresleme yeteneğine sahip 8086 işlemcisi çift adresten veriye erişir ve tek seferde 16 bit veri okur/yazar. Eğer tek adresten veri almak gerekirse işlemcinin 16 bit veriyi alması için iki erişim yapması gereklidir. 8088 işlemcisi 8 bit'lik olduğu için 16 bit'lik veriye ancak 2 erişim ile ulaşabilmektedir. 32 bit'lik işlemcilerde de benzer bir durum söz konusudur. Bu nedenle 80386/486 işlemcilerde işlenenler dörde tam bölünebilir adreslerden başlayarak erişildikleri zamanlarda tek erişimde aksi hallerde ise birden fazla erişimde alınabilirler. Derleyicilerin özellikle 32 bit işlemciler için en uygun kodu ürettikleri düşünülerek EFLAG yazmacında AC (Allignment Check) bit'i ve CR0 (Control Register0) yazmacındaki AM (Allignment Mask) bit'i donanım olarak belleğe hızlı erişebilirliği kontrol etmek amacıyla eklenmiştir.

İşlemcinin çalışma kipine bağlı olarak belleğe erişim şeklinde de değişiklikler olmaktadır. Korumalı ve gerçek kipi birbirinden ayıran en önemli özellik korumalı kipte yetkisiz bellek erişimleri için bir koruma olması ve her isteyenin her istediği bellek alanına erişememesidir. Gerçek kipte bunun aksine her isteyen her istediği bellek alanına uygun kesim ve göreli konum ikilisi ile erişebilir. Bu özellik nede-

[61] BPL, RBP'nin, SPL, RSP'nin, DIL, RDI'nin ve SIL, RSI'nin düşük anlamlı 8 biti olacak şekilde bir dönüşüm yapılmaktadır.

niyle gerçek kipte bazı programların diğerlerinin (hatta işletim sisteminin) çalışmasını etkilemesi, sistemi kullanılamaz hale getirmesi mümkündür.

Tablo 4-12: Farklı çalışma kiplerinde kesimli bellek erişimi için değerler/sınırlar

	Gerçek kip	16-bit Korumalı Kip	32-bit Korumalı Kip
Kesim büyüklüğü	16 bit, 64 KB (sabit)	16 bit, 1-64 KB	20 bit, 1 KB-1 MB veya 4 KB-4 GB
Kesim taban adresi	20 bit (1 MB) 16 * kesim yazmacı	24 bit (16 MB) *descriptor*[62] tablosundan	32 bit (4 GB), *descriptor* tablosundan
Kesim koruması	Yok	Var	Var
Kesim yazmacı	Kesim taban adresi / 16	*selectors*[63]	*selectors*

4.10.1 Gerçek Kipte Fiziksel Adres Hesabı

Gerçek kipte 80x86 ailesi işlemciler belleği kesim yazmaçları yardımıyla 64 KB'lık parçacıklar olarak görür ve adreslerler. Kesim içi adreslemeleri ise o kesimle birlikte kullanılan kesim içi işaretçiler yapar. Kesim yapısının bir gereği olarak mevcut 16 bit'lik yazmacın değeri kullanılarak 20 bit'lik[64] bir adres üretilir. CS:IP ve SS:SP de olduğu gibi kesim göreli konum çiftleriyle belirlenmiş olan verilerin adreslenebilen belleğin neresinde olduğunu bulmak için şu işlemin yapılması gerekir;

Fiziksel adres = kesim yazmaç değeri x 16 + işaret yazmaç değeri

Örnek 4-1

CS←2C58H ve IP←0013H olursa CS:IP ikilisi hangi fiziksel adresi belirler?

	2C580H	16 ile çarpma işlemi sayının 4 bit sola kaydırılmasıdır. Bu durumda kesim yazmacının on altı tabanındaki karşılığı bir hane sola kaydırılarak yazılır.
+	0013H	Göreli konum değeri
	2C593H	Toplama sonucunda 20 bit değer elde edilir.

Geçek kipte kesimler bellekte sabit bir adreste bulunmalarına karşın farklı kesim ve işaret yazmaç değerleri kullanılarak da erişilebilirler. Örneğin 2C593H fiziksel ad-

---

62 Korumalı kipte, gerçek kipten farklı olarak kesim taban adresi dışında kesim büyüklüğü (size) ve bu kesimin kullanıldığını gösterir bazı bayraklar da kullanılmaktadır. Bu bilgi 8 byte uzunluğunda descriptor adı verilen bir veri yapısı üzerinde saklanır. Descriptor'lar bellekte Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT) veya Local Descriptor Table (LDT) adı verilen tablolarda saklanır. İşlemci, GDTR, IDTR ve LDTR isimli 3 yazmaç yardımı ile sırası ile GDT, IDT ve LDT isimli tablolara erişebilir. Bu tabloların her biri 8192 taneye kadar descriptor barındırır.

63 Korumalı kipte kesim yazmaçları selectors adı verilen ve descriptor tablolarını indislemek üzere kullanılan değerleri barındırırlar. Kesim yazmaçlarındaki 13 yüksek anlamlı bit bu amaçla kullanılır (bu sayede her tabloda 2^{13} = 8192 adet farklı descriptor bulunabilmektedir). Takip eden bir bit, GDT ile LDT'yi, diğer iki tanesi ise ayrıcalık (privilege) değerini belirlemek için kullanılır.

64 16 bitlik işlemcilerinin 20 tane adres bacağı vardır. 32 bitlik işlemciler gerçek kipte kullanıldıkları zaman alt modeller ile uyumlu olabilmek için 20 tane adres bacağı kullanarak belleğe erişirler.

resine erişmek amacıyla, 2C58:0013H ikilisi kullanılabildiği gibi, 2C50:0093H ikilisi de kullanılabilmektedir.

4.10.2 Korumalı Kipte Fiziksel Adres Hesabı

32 bitlik işlemcilerde ortaya atılan korumalı kip kavramı ile fiziksel adreslemede de bazı değişiklikler oluşmuştur. Gerçek kipten farklı olarak korumalı kipte kesimler bellekte sabit bir yerde bulunmak durumunda olmadığı gibi bellekte dahi olmayabilir. Korumalı kip kendi içinde 16 bit ve 32 bit'lik olmak üzere iki ayrı şekilde incelenebilir.

4.10.2.1 16 Bit Korumalı Kip

Korumalı kip ile birlikte sanal bellek (virtual memory) olarak adlandırılan bir teknik kullanılmaktadır. Bu teknik uyarınca programın kullanmakta olduğu kod ve veri bellekte tutulmalı, diğer kod ve veri blokları kendilerine ihtiyaç duyuluncaya kadar geçici olarak disk üzerinde bulundurulmalıdır. Bu sayede kesimler kendilerine ihtiyaç olduğunda disk üzerinden belleğe alınmakta, kendisine olan ihtiyacın sona ermesi durumunda belleği başka işlemlerde kullanabilmek için tekrar disk birimine geri götürülmektedir[65].

Korumalı kipte her kesim, Descriptor Table adı verilen alanda bulunan bir kayıta referans verecek şekilde değer almaktadır. Bu kayıt sistemin o kesim hakkında sahip olası gereken;

- Bellekte bulunup bulunmadığı
- Bellekte bulunuyor ise adresinin ne olduğu ve erişim için bazı haklara ilişkin bilgileri

içermektedir. Burada fiziksel adres, Descriptor Table'da kesim yazmacının sahip olduğu değer ile belirlenen 32 bit büyüklüğündeki sayı ile göreli konum değerinin toplanması ile elde edilmektedir. Kesim büyüklükleri gerçek kipte olduğu gibi 64 KB ile sınırlıdır.

4.10.2.2 32 Bit Korumalı Kip

Gerçek ve 16 bit korumalı kipten farklı olarak göreli konum değerleri 32 bit'lik yazmaçlar ile ifade edildiğinden ötürü kesim büyüklükleri 4 GB olabilmektedir. Ayrıca kesimler 4 KB büyüklüğünde sayfa (page) adı verilen daha küçük parçalara ayrılabilmekte, bu sayede sanal bellek tekniği kesimler yerine sayfaları bellek ile disk arasında aktarmaktadır. Bu diğer bir değişle bir anda bir kesimin sadece bazı parçalarının bellekte olabileceği anlamına gelmektedir.

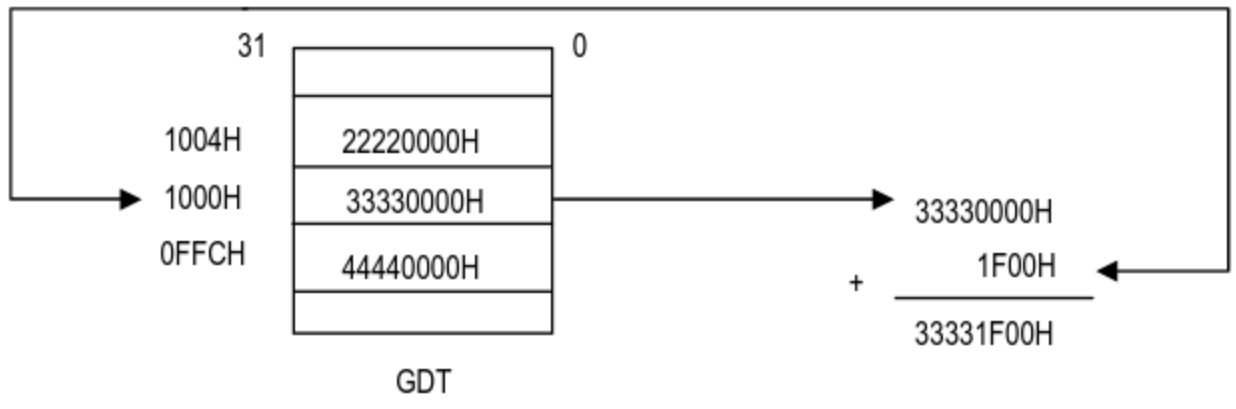
---

[65] Disk birimi üzerinden belleğe alınan kesim işinin bitmesini takiben tekrar disk birimi üzerine aktarılacaktır. Ancak bu aktarım kesimin belleğe alınmadan önce bulunduğu disk alanına yerleştirilmesini garanti edemez. Bu nedenle kesimler her belleğe gidiş / dönüşlerinde farklı bir disk alanına yerleştirilebilir. Bütün bu işlemler işletim sisteminin bellek yönetiminden sorumlu modülü yardımıyla kullanıcıya şeffaf biçimde yapılmaktadır.

Örnek 4-2

Korumalı kipte 1000:1F00 Kesim ve göreli konum ikilisinin belirlediği byte'ın fiziksel adresi nedir?

Örneğimizde kesimi belirleyen adres GDT (Global Descriptor Table) adı verilen alan içindeki 32 bitlik değere ulaşmak için bir işaretçi olarak kullanılmaktadır. Kesim değeri ile erişilen 32 bitlik değerin göreli konum değeri ile toplanması sonucunda korumalı kipte fiziksel adres değeri elde edilecektir.



Şekil 4-5: Korumalı kipte fiziksel adres hesabı

4.10.2.3 Sanal (Virtual) 8086 Kipi

Sanal 8086 kipi, gerçek kip programlarının korumalı kip kontrolü altında çalıştırılması durumunu ifade eder. Genel olarak DOS için yazılmış gerçek kip programlarının Microsoft Windows, OS/2 ve DOSEMU kullanılan LINUX/UNIX işletim sistemlerinde çalıştırılması durumunda kullanılmaktadır.

4.11 Çalışma Kipleri Arasındaki Farklar

Çalışma kipleri arasındaki farklar Tablo 4-13'de kısaca özetlenmiştir.

Tablo 4-13: Gerçek ve korumalı kip karşılaştırması

Gerçek Kip	Korumalı Kip
1. 16 bitlik kesim yazmaçları sayesinde belleğe 64 KB büyüklüğünde bloklar halinde erişmeye izin verir. Adreslenebilen toplam hafıza 1 MB ile sınırlıdır. 2. Aynı bilgisayar üzerinde bir program bir diğerinin kullandığı bellek alanına erişebilir ve bu alan üzerinde değişiklik yaparak onun çalışmasını engelleyebilir. Bu nedenle çoklu görev (multitasking) desteklemez. 3. Yazılım ile BIOS kodlarına veya donanıma doğrudan erişmek mümkündür.	1. Bellek koruması vardır. İzinsiz olarak işlemlerin birbirlerinin bellek alanlarına girmesini engeller ve çoklu görev yapısında çalışır. 2. Donanım olarak, programların çalışma sıralarını değiştirme ve birini durdurup diğerini çalıştırma özelliğine sahiptir (preemptive multitasking). 3. Korumalı kipte çalışan işlemci yazmaçları 32 bit olarak kullanılır. Gerçek kipteki yazmaçlara korumalı kipte yeni yazmaçlar katılır. 4. Günümüz işletim sistemleri açılış sırasında işlemciyi korumalı kipe geçirerek çalışırlar.

4.12 İşletim Sistemleri Açısında Çalışma Kipleri

İşletim sistemlerine göre işlemcinin çalışma kipleri Tablo 4-14'de gösterilmiştir.

Tablo 4-14: İşletim sistemlerine göre işlemcinin çalışma kipleri

İşletim Sistemi	Gerçek Kip	16 Bit Korumalı Kip	32 bit Korumalı Kip
DOS	✓		
Windows 3.x		✓	
Windows 3.2 & Win 32			✓
Windows 9x			✓
OS/2			✓
Win 2K/ XP			✓
Linux			✓

4.13 İşlemcinin Komutları Adım Adım Çalıştırması

80x86 ailesi işlemcileri komutları tek bir saat darbesinde (clock cycle) işleyemezler. Komutun işlenmesi için takip edilmesi gereken ara adımlar şu şekilde özetlenebilir;

1. Komutu belirleyen byte'ın bellek üzerindeki kod alanından alınması (instruction fetch)
2. IP yazmacının bir sonraki byte'ı gösterecek şekilde değiştirilmesi
3. Alınan komutun ne komutu olduğunun ve ne tür parametreler ile çalışacağının tespit edilmesi (instruction decode)
4. Gerekli olması durumunda kullanılacak parametrelerin bellek alanından alınması (operand fetch)
5. IP yazmacının alınan parametre sayısı ve tiplerine de bağlı olarak gerekiyor ise bir sonraki komutu gösterecek biçimde değiştirilmesi
6. Parametreler kullanılarak işlemin gerçekleştirilmesi (execute)
7. Elde edilen sonucun gerekli olan yere yerleştirilmesi (store)

80x86 ailesinin eski ve yeni işlemcileri hemen hemen aynı olan komut kümesini, adresleme tekniğini ve aynı işlem sırasını kullanmalarına rağmen yeni nesil işlemciler daha yüksek hızlarda çalışmaktadır. Bu performans artışının arkasında komutların önceden alınarak bir kuyruğa yerleştirilmesi (pre-fetch queue), ön bellek kullanımı ve ardışık düzende (pipeline) çalışma yatmaktadır. 8086 işlemcisi bu özelliklerin hiçbirinden yararlanmaz iken 32/64 bit işlemcilerde bu özelliklerin hepsi hatta daha fazlası mevcuttur.

5. 80X86 KOMUTLARI (MNEMONIC) VE KOMUT YAPISI

8086 işlemcisi ailesi, karmaşık komut kümeli bilgisayar mimarisine (CISC) uygun olarak tasarlanmış olup her komut birbirinden farklı büyüklüklerde bellek işgal eder. 80x86 Assembly dilinde komut satırı genel olarak şu şekilde yazılır:

{*Etiket :*} **Mnemonic** {{*İşlenen1*}, *İşlenen2*} {**;** Açıklamalar}

Her satırda *Etiket* bulunması gerekmez. Ancak *Etiket* kullanılıyor ise bunu takiben ":" Sembolünün de beraberinde kullanılması gereklidir. Anımsatıcı (Mnemonic), gerçekleştirilecek işlemin ne olduğunu belirleyen sembolik işlem kodudur. Birden fazla sayıda işlenen olması durumunda işlenenlerin "," sembolü ile birbirlerinden ayrılması gereklidir. İlk işlenen **hedef** (destination – *İşlenen1*) ikincisi ile **kaynak** (source – *İşlenen2*) olarak adlandırılır. Gerek duyulması halinde satır sonuna bir açıklama metni eklenebilir. Ancak bu açıklamanın komut olarak yorumlanmasını engellemek üzere başına ";" sembolü konmalıdır.

Assembly dili derleyicisi (assembler) sembolik komutları işleyip, işlemciye ne tür bir işlemi, ne büyüklükteki (byte, word, vs.) ve ne tipteki (yazmaç, bellek alanı, vs.) işlenen(ler) üzerinde gerçekleştireceğini belirleyen makine kodunu üretmekten sorumludur. Assembly dilinde program yazarken makine kodu (machine code) ile kodlama yapmak yerine kullanıcıya daha yakın olan anımsatıcılar kullanılmaktadır. İşlemcinin mimari modelinin doğal bir sonucu olarak, anımsatıcıların yerine getirecekleri işlemleri kaç adet işlenen kullanarak tamamlayacakları da değişmektedir. Hiçbir işleneni olmayan anımsatıcılar olduğu gibi bir veya iki (hatta 80x86 ailesinin üst serilerinde üç) işlenen üzerinde işlem yapan anımsatıcılar da vardır[66]. Genel olarak işlemlerin iki işlenen üzerinde yapıldığını söylemek mümkündür. İşlem yönü sağdan sola doğrudur. Yani sonuç her zaman hedef olarak adlandırılan *İşlenen1* üzerinde oluşacaktır.

İşlenen1←*İşlenen1* **işlem** *İşlenen*2

İşlenenler mutlaka aynı tipte olmalı veya aynı tipe dönüşüm yapılmasını sağlayan bazı **sözde (pseudo)** komutlar kullanılarak (casting) işlem gerçekleştirilmelidir. Komut yanına açıklama satırları eklemek assembly dilinde yazılan programların başkası tarafından incelenmesini veya daha sonra bakıldığında anlaşılmasını kolaylaştıracaktır.

---

[66] 80x86 işlemci ailesinde bazı komutların işleneni yoktur. Genelde bu gibi komutlarda akümülatör gizli işlenen olarak görev yapmaktadır. Benzer bir durum dizgi komutların için de geçerlidir. Dizgi komutları ile ilgili ayrıntılar için 5.8. bölümü inceleyiniz.

Tablo 5-1: Komut kümesinde kullanılan kısaltmalar

acc	Akümülatör (EAX/AX/AL yazmaçlarından herhangi biri)
reg	8/16/32 bit'lik herhangi bir yazmaç
regb	8 bit'lik bir yazmaç
regw	16 bit'lik bir yazmaç
reg32	32 bit'lik bir yazmaç
sreg	Herhangi bir kesim yazmacı
mem	Herhangi bir bellek adresini simgeler
idata	8/16/32 bit'lik herhangi bir değer (3, 0Ah, 012EFh, 'A' gibi)
[..]	İşaret grubu yazmaç veya göreli konum değerini gösteren sayı ile erişilen bellek içeriği [67]
disp8 / disp16	8 bit (-128…0…127) / 16 bit (-32768..0..32767) ile ifade edilebilecek büyüklükteki bir sayı kadar ilerleme
dest / src	Varış işleneni / Kaynak işleneni
opr / opr #	İşlenen / İşlenen 1 / İşlenen 2 / İşlenen 3
italik yazı	Değişken isimleri
0 / 1 / x / ? / --	İşlem sonucunda bayrak değeri clear (0) / set (1) / işleme göre değişmiş (x) / belirsiz (?) / değişmemiş (-) olabilir.
16p	16 bit korumalı kip

İlerleyen bölümlerde, 16 ve 32 bit işlemcilerin gerçek kip çalışması sırasında kullanılan temel komutlar işlevlerine göre gruplanarak, genel yazım kuralları, işlem sonucunda bayrak durumları ve komut işleyişi örnekler ile açıklanacaktır.

5.1 Veri Aktarım Komutları

Bu grupta toplanan komutlar, çeşitli şekillerde veri aktarımları yapmak üzere tasarlanmışlardır. Veri aktarım komutlarının çalışması **bayrakları etkilemez**. Bu, veri aktarım komutlarını diğerlerinden ayıran en önemli özelliklerdendir.

---

[67] Köşeli parantez içinde salt sayı kullanımı sadece debug ortamında mümkündür. Debug ortamında tüm sayılar, H son eki kullanılmaksızın, on altı tabanındadır. Assembly programların derlenerek çalıştırılacağı durumlarda köşeli parantezler içinde erişilmek istenen kesim içinde görev yapan işaret grubu bir yazmacın olması veya veri alanında tanımlı bir diziye erişim yapılacak ise dizinin isminin, köseli parantezler içinde yer alan sayının önünde yazılmış olması gerekir. Kullanılan sayı sistemi B, O, H gibi bir son ek ile belirtilmez ise sayı onluk sistemde kabul edilir. Bu bölümde verilen bazı örnekler her iki tür kullanım için verilmiş olup, debug ortamında kodun derlenerek çalıştırılmasını gerektiren örneğin (ve tersinin) kullanılması hataya neden olur. Örneklerin bu özellikler akılda tutularak incelenmesi işleyişi anlamayı kolaylaştıracaktır.

MOV

8086/286/386/486

move data

8/16/32 Bit

MOV reg, idata
MOV mem, idata
MOV reg, reg
MOV reg, mem
MOV mem, reg
MOV sreg, reg
MOV sreg, mem
MOV reg, sreg
MOV mem, sreg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

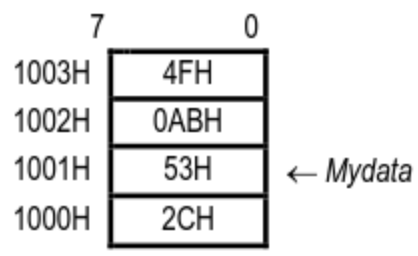
Açıklama:

İki işlenen ile kullanılan **MOV** komutu ikinci işlenendeki veriyi ilk işlenene aktarır. Ancak **MOV** komutunun bazı kısıtlamaları vardır. MOV komutunun her iki işleneni *mem* ve *sreg* olamayacağı gibi ilk işlenen *sreg* ise ikincisi *idata*'da olamaz.

MOV dest, src

şeklinde ifade edilen komut, src'nin değerini dest'e yerleştirir (dest←src). En önemli özelliği bayraklar üzerinde herhangi bir değişiklik yapmamasıdır. **MOV** komutunda her iki işlenenin de aynı tipte olması son derece önemlidir. Aksi halde derleyici tip uyuşmazlığı hatası verecektir. Özellikle işlenenlerden birinin *mem* olması durumunda veri tiplerinin uyumlu olup olmadığını tespit etmekte bazı zorluklar oluşabilir. Bunun için **PTR** komutundan yararlanılır.

Örnek:

MOV *Mydata*, AX	 *Mydata* isimli bellek alanına AX yazmacında bulunan değer yerleştirilmektedir. *Mydata* isimli bellek alanı yazmaç ile aynı tipte tanımlanmış olmalıdır. Aktarım işlemi sırasında AL yazmacındaki değer 1001H adresine, AH yazmacındaki değer ise 1002H adresine yerleştirilecektir.[68]
MOV AL, *Result*	*Result* isimli bellek alanından, AL yazmacına veri aktarılmaktadır. *Result* isimli bellek alanının byte olarak tanımlanmış olması gereklidir.
MOV AH, AL	AL yazmacındaki değer AH yazmacına aktarılmaktadır.
MOV *Mydata*, 12	*Mydata* isimli bellek alanına 12 değeri yerleştirilmektedir. Kaynak işlenen olan 12 sayısı hem byte hem de word bir büyük olarak ifade edilebileceği dikkate alındığında *Mydata* isimli değişkenin byte veya word olarak tanımlanmış olmasının işleyişe bir etkisi olmayacak, her iki durumda da atama işlemi yapılacaktır.
MOV WORD PTR [1001], 120A	Bellek alanına adres ile erişilmesi durumunda tip uyuşmazlığı söz konusu olabilir. Programcı, bellek üzerinde işlemleri hangi boyutta (byte/word) yapmak istediğine bağlı olarak gerekli tip tanımlarını yapmalıdır.. Bunun için uygun tipte adresleme kipi kullanılmalı (bkz. 6. bölüm) veya **PTR** sözde komutu ile tip belirlemesi yapmalıdır. Örneğimize; [1001H]←0AH ve [1002H]←12H değerleri yerleştirilir.

[68] Bellek kullanılarak yapılan word tipindeki veri aktarım işlemlerinde düşük anlamlı (low) byte, düşük adresli bellek gözüne, yüksek anlamlı (high) byte, yüksek adresli bellek gözüne yerleştirilecektir. Bellek alanına bir değişken ismi ile yapılan erişimlerde her iki işlenenin de aynı tipte tanımlanmış olması gereklidir.

MOVSX

80386/486

move with sign extention

8/16p/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
MOVSX reg, reg												
MOVSX reg, mem	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bu komut 8 bit'lik değeri, işareti ile birlikte 16 veya 32 bit'lik alana, 16 bit'lik değeri işareti ile birlikte 32 bit'lik alana yerleştirir. İşaret, en anlamlı bit'in, yüksek anlamlı byte veya word boyunca tekrar edilmesi ile aktarılmaktadır[69]. Genel olarak ilk işlenen, ikinci işlenenden en az bir üst boyutta olmalıdır. Böylece ikinci işlenende bulunan değer daha üst büyüklüğe **işaretli** sayı olarak aktarılmış olur.

Örnek:

MOVSX EAX, AL	AL yazmacındaki 8 bit'lik değeri 32 bit'e uzatarak EAX yazmacına aktarır. Bu işlem sırasında AL yazmacının işaret bit'i (7.bit), 8–31. bit'ler boyunca tekrarlanır. AL←67H ise işlemden sonra EAX←00000067H olacaktır.
MOVSX EDI, WORD PTR [ESI]	ESI yazmacının belirlediği bellek gözündeki 16 bit değeri 32 bit'e uzatarak EDI yazmacına aktarır. 15. bit ile ifade edilen işaret, 16–31. bit'lerde tekrarlanacaktır.
MOVSX CX, DL	DL yazmacındaki 8 bit'lik değeri 16 bit'e uzatarak CX yazmacına yerleştirir. DL←80H ise sonuçta CX←0FF80H değerini alacaktır.

MOVZX

80386/486

move with zero extention

8/16p/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
MOVZX reg, reg												
MOVZX reg, mem	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bu komut 8 bit'lik değeri, 16 veya 32 bit'lik alana, 16 bit'lik değeri 32 bit'lik alana yerleştirir. İşaret uzantısı ise en anlamlı bit'lerin 0 ile doldurulması ile elde edilir. Genel olarak ilk işlenen, ikinci işlenenden en az bir üst boyutta olmalıdır. Böylece ikinci işlenende bulunan değer daha üst büyüklüğe **işaretsiz** sayı olarak aktarılmış olur.

Örnek:

MOVZX EAX, AL	AL yazmacındaki 8 bit'lik değeri 32 bit değere uzatarak EAX yazmacına yerleştirir. 8–31. bit'ler 0 dır. AL←80H ise sonuçta EAX←00000080H olacaktır.
MOVZX EDI, WORD PTR [ESI]	ESI ile belirlenen bellek gözündeki 16 bit değeri 32 bit'e uzatarak EDI yazmacına yerleştirilir. 16–31 bit'ler 0 dır. **WORD PTR** ifadesi ile bellekten bir word alınacağı belirlenmektedir.
MOVZX CX, DL	DL yazmacının 8 bit'lik değeri 16 bit'e uzatılarak CX yazmacına yerleştirilir. 8–15. bit'ler 0 dır.
MOVZX BX, BL	BL yazmacındaki değeri işaret değeri 0 olacak şekilde BX yazmacına yayar. Ancak MOV BH,0 işlemi buna nazaran daha kısa sürede tamamlanacaktır.

[69] Bu komut, 32 bit ve üstü işlemcilerde kullanılmaktadır. Daha alt model işlemcilerde benzer görev yapan **CBW**, **CWD**, **CWDE** komutlarını inceleyiniz.

LEA

8086/286/386/486

load effective address

16/32 Bit

LEA regw, mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
İşlem sonucunda regw de oluşan değer mem ile tanımlı bellek alanının, kesimin başından itibaren kaç byte ileride olduğunu (göreli konum) belirlemektedir. Elde edilecek adres 16 bit uzunluğunda olacaktır. Bu nedenle işlem sonucunun oluşacağı kısım mutlaka word tipinde bir yazmaç olmalıdır. Aynı işlem **OFFSET** sözde komutu ile de gerçekleştirilebilmektedir.

Örnek:

LEA SI, *Mydata*

	7 … 0	
1004H	34H	
1003H	4FH	
1002H	0ABH	← *Mydata*
1001H	53H	
1000H	2CH	

Mydata isimli bellek alanının, tanımlı olduğu kesimin başından kaç byte uzakta olduğunu gösteren değer SI yazmacında oluşacaktır. Yandaki bellek haritasında verilen değerler uyarınca SI←1002H değerini alır.

LDS

8086/286/386/486

load data segment register

16/32 Bit

LDS regw, mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
regw ile belirlenen yazmaç ile DS yazmacına, mem ile belirlenen bellek bölgesinde bulunan değerler yüklenir.
Yapılan işlem: regw←[mem]
DS←[mem + 2]
Bu komut bir seferde DS:SI ikilisinin değerlerinin bellek alanından alınmasını sağlamaktadır. Genel olarak, SI yazmacı ile kullanılmasının arkasında, bellek erişimlerinde DS ve SI yazmaçlarının birlikte kullanılması yatar. Benzer şekilde DI ve BX yazmaçları da bu komut ile birlikte kullanılabilir. mem değeri doğrudan bir bellek adresi olarak verilebildiği gibi bir değişken ile de verilebilir. Bu durumda kullanılacak olan değişkenin 32 bit (double word) olacak şekilde tanımlanmış olması gereklidir.

Örnek:

LDS SI, [0000]

veya

LDS SI, *Mydata*

	7 … 0	
0004H	4DH	
0003H	45H	
0002H	53H	
0001H	53H	
0000H	41H	← *Mydata*

Yandaki bellek haritası uyarınca işlem sonunda SI←5341H[70] ve DS←4553H değerlerini almış olacaktır. *Mydata* değişkeni kullanılarak işlem yapılmak istendiğinde değişkenin double word olarak tanımlanmış olması gereklidir.

[70] Bellek üzerinde yapılan aktarım işlemlerinin genel kuralları uyarınca 0000H adresindeki byte SI yazmacının düşük anlamlı (low) alanına, 0001H adresindeki byte ise yüksek anlamlı (high) alanına yerleştirilmektedir. Bu durumda SI yazmacında 5341H değeri oluşacaktır.

LES

8086/286/386/486

load extra segment register

16/32 Bit

LES regw, mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:

LDS komutu ile aynı özellikleri göstermektedir. Ancak işlem diğerinden farklı olarak ES yazmacı kullanılarak gerçekleştirilir.

Yapılan işlem: regw←[mem]
ES←[mem + 2]

Bu komut bir seferde ES:DI ikilisinin değerlerinin bellek alanından alınmasını sağlamaktadır. Genel olarak, DI yazmacı ile kullanılmasının arkasında, bellek erişimlerinde ES ve DI yazmaçlarının birlikte kullanılması yatmaktadır. Benzer şekilde SI ve BX yazmaçları da bu komut ile birlikte kullanılabilir. mem değeri doğrudan bir bellek adresi olarak verilebildiği gibi bir değişken ile de verilebilir. Bu durumda kullanılacak olan değişkenin 32 bit (double word) olacak şekilde tanımlanmış olması gereklidir.

Örnek:

LES DI, [0001]

veya

LDS SI, *Mydata*

	7 … 0	
0004H	4DH	
0003H	45H	
0002H	53H	
0001H	51H	← *Mydata*
0000H	41H	

Yandaki bellek haritası uyarınca işlem sonunda DI←5351H ve ES← 4D45H değerlerini almış olacaktır. *Mydata* değişkeni kullanılarak işlem yapılmak istendiğinde değişkenin double word tipinde tanımlanmış olması gereklidir.

XCHG

8086/286/386/486

exchange

8/16/32 Bit

XCHG reg, reg
XCHG reg, mem
XCHG mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:

XCHG, işlenenlerin değerlerinin değiş-tokuş edilmesini sağlayan bir komuttur. Aynı sonucu, biraz daha uzun sürede, ek bir yazmaç veya bellek alanı kullanarak ve daha fazla kod üreterek **MOV** komutları ile elde etmek de mümkündür. Ancak assembly dili mevcut kaynakların sınırlı olduğu durumlarda çözümler üretebilen bir dil olup, bu gibi durumlarda **XCHG** komutu tercih edilmelidir.

Yapılan işlem: *temp*←opr1
opr1←opr2
opr2←*temp*

Örnek:

XCHG AX, BX	AX yazmacının sahip olduğu değer ile BX yazmacının sahip olduğu değer değiştirilir. İşlemden önce AX←1234H, BX←1BCDH ise işlemden sonra AX←1BCDH, BX←1234H olacaktır.
XCHG AX, *Mydata*[SI]	Burada ise AX yazmacının değeri ile *Mydata* isimli bellek alanının SI'ıncı adresinden başlayan word yer değiştirmektedir. *Mydata* word olarak tanımlanmalıdır.

XLAT / XLATB 8086/286/386/486

translate byte 8 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
XLAT												
XLATB	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Doğrudan işleneni olmayan bir komuttur. AL yazmacını, başlangıcı DS:BX yazmaçları ile belirlenen adresteki bir tablo içerisinde indis olarak kullanmayı sağlar.
Yapılan işlem: AL←DS:[BX + AL] veya AL←DS:[EBX + AL] dır.

Örnek:

LEA BX, *Ascii2Ebcdic* MOV AL, 'A' XLAT	*Ascii2Ebcdic* isimli bellek alanında ASCII ile EBCDIC kodlama sistemi arasındaki dönüşüm yapmayı sağlayan, 256 byte uzunluğunda bir dizi olduğunu, dizinin indislerinin ASCII kodlar, içeriğinin ise ilgili ASCII koda karşı gelen EBCDIC değer olduğunu düşünelim. AL yazmacına 'A'[71] yerleştirilip, XLAT komutu işlendiğinde diziden 'A' harfinin EBCDIC karşılığı alınarak AL yazmacına konacaktır.

5.2 Aritmetik Komutlar

Bu grupta toplanan komutlar aritmetik işlemleri gerçekleştirmek üzere tasarlanmıştır. Ancak üzerinde işlem yapılacak sayıların işaretli veya işaretsiz olarak değerlendirilmesi ve buna uygun komutların seçilmesi programcının sorumluluğundadır. İşlemlerin sonuçları hem işleneni hem de bayrakları etkilemektedir. Bu gruba dâhil komutların bazılarında tek işlenen, bazılarında çift işlenen olduğu ve bazı komutlarda gizli işlenen olarak **CF** bayrağının işleme dâhil olduğu görülecektir.

ADD 8086/286/386/486

Addition 8/16/32 Bit

ADD reg, idata
ADD mem, idata
ADD reg, reg
ADD reg, mem
ADD mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	x	--	x	--	x

Açıklama:
Verilen iki işleneni toplayarak sonucu ilk işlenene yerleştirir. Toplama işleminde de işlenenlerin aynı tipte olmasına dikkat edilmelidir. İşlem sonucunda bayraklar da etkilenecektir.
Yapılan işlem: dest←dest + src

Örnek:

ADD AX, BX	AX yazmacı ile BX yazmacının değerleri toplanır ve sonuç AX yazmacında oluşur.
ADD *Toplam*, AL	*Toplam* isimli bellek alanının değeri ile AL yazmacında bulunan değer toplanır ve sonuç *Toplam* isimli bellek alanında oluşur. İşleminin gerçekleşebilmesi için *Toplam* isimli bellek alanının da byte olarak tanımlanmış olması gereklidir.
ADD BYTE PTR[1002], 5	Her iki işlenen de byte tipindedir. Bellek adresinin başına **BYTE PTR** yazılmaması veya atanacak değerin 0–255 aralığından büyük verilmesi durumunda hata oluşacaktır.

[71] 'A' ile değer ataması yaptığımızda söz konusu harfin ASCII kodu olan 41H sayısı AL yazmacına atanmış olacaktır.

ADC

8086/286/386/486

add with carry

8/16/32 Bit

ADC reg, idata
ADC mem, idata
ADC reg, reg
ADC reg, mem
ADC mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	x	--	x	--	x

Açıklama:
ADD komutuna benzer özellik göstermekle beraber **CF** bayrağının değeri de toplama işlemine dâhil edilmektedir. Özellikle 16 bit'lik yazmaçlar kullanılarak 32 bit'lik toplama işlemi yapılabilmesi için **CF** bayrağındaki değerin yüksek anlamlı (high) word alanına aktarılması gereklidir. Bu gibi durumlarda **ADC** komutundan yararlanılır.
Yapılan işlem: dest←dest + src + CF

Örnek:

ADC AX, 4	AX←23B8H ve CF←1 ise sonuçta AX←23BDH olur.
ADC WORD PTR [1002], 0002 veya ADC *Sayac*, 257	1002H ve 1003H adreslerinde bulunan word'e 0002 değeri ve CF bayrağının değeri ilave edilerek sonuç aynı adrese word olarak yazılır. *Sayac* isimli değişkene eklenen değer byte sınırının dışında olduğu için değişken word tipinde olmalıdır.

XADD

80486

exchange and add

8/16p/32 Bit

XADD reg, reg
XADD mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	x	--	x	--	x

Açıklama:
İki işlenenin toplamı hedef işlenende oluşurken kaynak işlenen hedef işlenenin toplamdan önceki değerini alır.
Yapılan işlem: *temp*←dest
dest←*temp* + src
src←*temp*

Örnek:

XADD AX, BX	AX←1234H, BX←0002H ise sonuçta BX←1234H, AX←1236H olur.

SUB

8086/286/386/486

subtraction

8/16/32 Bit

SUB reg, idata
SUB mem, idata
SUB reg, reg
SUB reg, mem
SUB mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	x	--	x	--	x

Açıklama:
İlk işlenenden ikinci işlenen çıkarılır ve sonuç ilk işlenende oluşur.
Yapılan işlem: dest←dest - src

Örnek:

SUB AX, BX	AX←AX - BX işlemi yapılmaktadır. Sonuç AX yazmacında oluşur.
SUB WORD PTR[1002], 0009	1002H adresinden alınan word değerinden (1003H adresindeki byte da işleme girer) 0009 sayısı çıkartılır, sonuç 1002H adresli bellek gözüne word olarak yazılır.

SBB

8086/286/386/486

subtraction with borrow

8/16/32 Bit

SBB reg, idata
SBB mem, idata
SBB reg, reg
SBB reg, mem
SBB mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	x	--	x	--	x

Açıklama:
Eldeli çıkarma işlemini yapar. İlk işlenenden ikinci işleneni ve CF bayrağının değerini çıkartır ve sonucu ilk işlenene aktarır.
Yapılan işlem: dest←dest - src - CF

Örnek:

SBB AX, 4	AX←2345H ve CF←1 ise sonuçta AX←2340H olur
SUB EAX, ECX SBB EDX, EBX	EDX:EAX yazmaç çiftinin oluşturduğu 64 bit'lik sayıdan EBX:ECX yazmaç çiftinin oluşturduğu 64 bit'lik sayı çıkarılır. Sonuç EDX:EAX çiftinde oluşacaktır.

INC

8086/286/386/486

increment

8/16/32 Bit

INC reg
INC mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	x	--	x	--	--

Açıklama:
Tek işlenenli bir komuttur ve işlenenin değerini bir arttırmaktadır. Genel olarak indis değerlerini arttırmak için kullanılır. Sadece CF bayrağı üzerinde değişikliğe neden olmaz.
Yapılan işlem: dest←dest + 1

Örnek:

INC AX	AX yazmacının yeni değeri AX←AX+1 olacaktır.
INC WORD PTR [1002]	1002H ve 1003H adresinde yer alan word değeri bir arttırılacaktır.
INC *Mydata*	*Mydata* isimli bellek alanındaki değer bir artırılacaktır.

DEC

8086/286/386/486

decrement

8/16/32 Bit

DEC reg
DEC mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x-	--	x	--	x	--	--

Açıklama:
Tek işlenenli bir komuttur ve işlenenin değerini bir azaltmaktadır. Genel olarak indis değerlerini azaltmak için kullanılır. Sadece CF bayrağı üzerinde değişikliğe neden olmaz.
Yapılan işlem: dest←dest - 1

Örnek:

DEC AL	AL←3AH ise işlem sonucunda AL←39H olacaktır.
DEC *Sonuc*	*Sonuc* isimli değişkenin değeri bir azaltılır.
DEC BYTE PTR [1002]	1002H adresindeki byte'ın değeri bir azaltılır.

NEG

8086/286/386/486

negate / two's complement

8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
NEG reg NEG mem	x	--	--	--	x	x	--	x	--	x	--	x

Açıklama:
Tek işleneni olan bir komuttur. Verilen işlenenin ikiye tümleyenini almak için kullanılır. Diğer bir deyişle işlenenin 0'dan çıkartılması olarak da yorumlanabilir.

Örnek:

NEG AL	AL←05H ise yeni değer AL←0FBH olacaktır.
NEG *Mydata*	*Mydata* isimli bellek alanındaki değerin ikiye tümleyeni aynı alana yerleştirir.
NEG WORD PTR [1002]	1002H ve 1003H de yer alan word'ün ikiye tümleyeni aynı alana yerleştirir.

CMP

8086/286/386/486

compare

8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
CMP reg, idata CMP mem, idata CMP reg, reg CMP reg, mem CMP mem, reg	x	--	--	--	x	x	--	x	--	x	--	x

Açıklama:
İşlenenleri karşılaştırmak için kullanılan bir komuttur. İşlem sonunda -diğerlerinden farklı olarak- işlenenlerin değerleri değişmez. Sadece bayraklar sonucu gösterecek şekilde değişmektedir. **CMP** komutunu takiben karşılaştırmanın sonucuna göre programın akışını değiştirecek olan bir koşullu dallanma komutu kullanılır.
Yapılan işlem: opr1 – opr2

Koşul ifadesi	**İşaretli karşılaştırma**	**İşaretsiz karşılaştırma**
opr1 $>$ opr2	ZF = 0 $\wedge$ SF =OF	CF =0 $\wedge$ ZF=0
opr1 $\geq$ opr2	SF = OF	CF = 0
opr1 = opr2	ZF = 1	ZF = 1
opr1 $\leq$ opr2	ZF = 1 $\wedge$ SF $\neq$ OF	CF = 1 $\vee$ ZF = 1
opr1 $<$ opr2	SF $\neq$ OF	CF = 1

Örnek:

CMP AX, BX	AX–BX işlemi yapılır. Sonuç bayraklar üzerinde görülür.
CMP AL, 00H	AL yazmacından 00H değeri çıkarılır. Sonuç bayraklardadır.
CMP BX, *Result*	BX yazmacından *Result* ismi ile belirlenen bellek alanında bulunan word çıkarılır.
CMP BYTE PTR [1002], 3	1002H adresinde yer alan byte'tan 3 çıkarılır. Sonuç bayraklardadır.

CMPXCHG

80486

compare and exchange

8/16p/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
CMPXCHG reg, reg												
CMPXCHG mem, reg	x	--	--	--	x	x	--	x	--	x	--	x

Açıklama:

dest değeri, akümülatör ile karşılaştırılır Eşitlik söz konusu ise src'deki değer dest'e atanır. Aksi halde akümülatör üzerindeki değer dest ile değiştirilir.

```
Yapılan işlem:   if      acc = dest then
                         ZF←1
                         dest←src
                 else
                         ZF←0
                         acc←dest
                 endif
```

Örnek:

CMPXCHG CX, BX	AX←6BCDH, BX←1234H ve CX←789AH iken işlem yapılacak olursa, AX≠CX olduğu için ZF←0 ve AX←789AH olacaktır.

MUL

8086/286/386/486

unsigned multiplication

8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
MUL reg												
MUL mem	x	--	--	--	?	?	--	?	--	?	--	x

Açıklama:

İşaretsiz çarpma işlemidir. Tek işlenenli olan bir komut olarak görülmek ile birlikte akümülatör bu komutun gizli işleneni durumundadır. Byte bazında işlem yapılmakta ise AL, word bazında işlem yapılmakta ise AX yazmaçları ile çarpma işlemi gerçekleştirilmektedir (32 bit çalışmada EAX ile çarpma işlemi gerçekleşmektedir).Buna göre;

İşlem tipi	Çarpılan	Sonuç
8 bit	AL	AX
16 bit	AX	DX:AX
32 bit	EAX	EDX:EAX

Örnek:

MUL CL	AX←AL * CL işlemi yapılacaktır. AL←80H ve CL←10H ise sonuç AX←0800H olur. Çarpma işlemi **işaretsiz** sayılar ile yapıldığı için sonuç pozitiftir. İşlem 128*16 olup sonuçta elde edilen değer 2048 yani 0800H sayısıdır.
MUL BX	DX:AX←AX * BX olacaktır.
MUL *Mydata*	*Mydata*'nın byte, word veya dword tanımlı olmasına göre sonuç AX, DX:AX veya EDX:EAX yazmaçlarında oluşacaktır.
MUL EBX	EDX:EAX←EAX * EBX olacaktır.
MUL BYTE PTR [1002]	1002H adresindeki byte ile AL çarpma işlemine tabii tutulur. Sonuç AX'de oluşur.

IMUL

8086/286/386/486

integer signed multiplication

8/16/32 Bit

IMUL reg	acc←acc*reg
IMUL mem	acc←acc*mem
IMUL reg, reg	opr1←opr1*opr2
IMUL reg, mem	opr1←opr1*opr2
IMUL reg, idata	opr1←opr1*opr2
IMUL reg, reg, idata	opr1←opr2*opr3
IMUL reg, mem, idata	opr1←opr2*opr3

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	?	?	--	?	--	?	--	x

Açıklama:

MUL komutuna benzer özelliktedir. Ancak burada işlenenler **işaretli sayı** olarak işleme tabii tutulurlar. Programcının kullandığı sayıların özelliklerine göre uygun çarpma işlemini tespit etmesi gereklidir.

Örnek:

IMUL CL	AL←0FBH (-5) ve CL←0FB (-5) ise sonuç AX←0019H (25 olur).
IMUL CL	AL←80H (yani -128) ve CL←05H ise sonuç AX←FD80H olacaktır. Bu işlem işaretli sayılar üzerinde yapıldığına ve elde edilen sonucun en anlamlı biti bir olduğuna göre sonucun negatif olduğu anlaşılmaktadır.
IMUL EBX	EDX:EAX←EAX * EBX

DIV

8086/286/386/486

unsigned divide

8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
DIV reg DIV mem	?	--	--	--	?	?	--	?	--	?	--	?

Açıklama:

İşaretsiz sayılar kullanılarak yapılan bölme işlemidir. Bölme işlemi yüksek seviyeli dillerde olduğu gibi kesirli sonuç üretmez. Bu nedenle hem bölüm hem de kalan değerlerinin sonuç olarak ifade edilmesi gereklidir. Yapılan işlem:

İşlem tipi	İşlem	Bölüm	Kalan
8 bit	AX / src	AL	AH
16 bit	DX:AX / src	AX	DX
32 bit	EDX:EAX / src	EAX	EDX

Örnek:

DIV CH	AX / CH işlemi yapılır. AL←bölüm, AH←kalanı gösterir. AX←082EH ve CH←10H olsun. Sonuç AL←82H AH← 0EH olur.
DIV BX	DX:AX yazmaçlarıyla ifade edilen 32 bit sayı, BX yazmacındaki 16 bit sayıya bölündüğünde sonuç AX←bölüm, DX←kalan olarak gösterilir.
DIV EBX	EAX/EBX işlemi yapılır. EAX←bölüm EDX←kalanı gösterir.
DIV WORD PTR [1002]	DX:AX ikilisinin gösterdiği 32 bit değer 1002H ve 1003H adresini işgal eden word tarafında bölünecektir.

IDIV 8086/286/386/486

integer signed division 8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
IDIV reg												
IDIV mem	?	--	--	--	?	?	--	?	--	?	--	?

Açıklama:
DIV ile benzer olup işlem **işaretli sayılar** üzerinde gerçekleştirilmektedir.
Yapılan işlem: Low (acc)←acc / opr 1
High (acc)←acc modulo opr 1

Örnek:

IDIV CH	AX←0FD80H (-640) ve CL←08H ise, sonuç AL←B0H ve AH←00H olacaktır. Bölme işlemi işaretli bölme ile yapıldığına ve sonucu belirleyen AL yazmacının en anlamlı biti bir olduğuna göre sonucun negatif olduğu anlaşılmaktadır. Bu durumda B0H (176) sayısının ikiye tümleyeni hesaplandığında 50H (80) değerine ulaşır. Sonucun negatif olduğu bilindiğine göre elde edilen değer –50H (-80) olacaktır.
IDIV CH	AX←0FD80H (-640) ve CH←0F8H (-8) ise sonuç AL←50H (80) AL←00H olarak bulunacaktır. Dikkat edilecek olursa AL de oluşan sonucun en anlamlı biti 0 olduğu için elde edilen sayının pozitif olduğu anlaşılmaktadır. Zaten matematik kuralı olarak iki negatif sayının bölümü pozitif sonuç verecektir.

5.3 Dallanma Komutları

Dallanma komutları, programın akışını değiştiren nitelikteki komutlardandır. Diğer deyişle **IP** yazmacının değeri bu komutların çalıştırılması ile değişecektir. Dallanma komutları koşullu ve koşulsuz olmak üzere iki ayrı grup altında incelenebilir.

5.3.1 Koşullu Dallanma Komutları

Koşullu dallanma komutları genellikle bir **CMP** komutunu takiben program akışını başka bir noktaya kaydırmak amacıyla kullanılır. Koşullu dallanma komutları bayrakların durumuna bakarak hangi noktaya (etiket-label) gidileceğini belirlemektedir. Çok gerekmedikçe **CMP** komutu ile koşullu dallanma komutu arasında başka bir komut bulunmaması gereklidir[72].

Koşullu dallanma komutlarında atlama aralığı 8 bit ile (disp8) sınırlıdır. Bu nedenle koşullu dallanma komutları ancak 128 byte geriye (düşük adrese) veya 127 byte ileriye (büyük adrese) ulaşmayı sağlamaktadır. Bu aralığın dışına çıkılmasının gerektiği durumlarda koşulsuz dallanma komutlarından yararlanmak gerekmektedir.

Bazı koşullu dallanma komutları işaretsiz sayılar üzerinde yapılan işlemlerin, diğer bir grubu ise işaretli sayılar ile yapılan işlemlerin sonuçlarına göre programın akışını değiştirmekte kullanılırlar. Her zaman olduğu gibi sonucun işaretli mi? işaret-

[72] **CMP** ile koşullu dallanma komutu arasına bir komutun girmesinin gerektiği çok özel durumların oluşması halinde araya girecek komutun bayraklar üzerinde değişikliğe neden olmayacak türde (örneğin veri aktarım komutları) olmasına dikkat edilmelidir.

siz mi? olarak değerlendirileceğine karar vermek ve buna uygun komutları kullanmak programcının sorumluluğundadır.

İşlemlerin sonuçlarının zaman zaman farklı şekillerde ifade edilmesi mümkündür. Koşullu dallanma komutlarında bu durum kendini göstermektedir. Örneğin CMP AX, BX komutunun çalıştırılması sonucunda ZF bayrağı 1 değerini almış ise; bu AX ve BX yazmaçlarındaki değerlerin farkının sıfır (zero) olduğunu ifade ettiği gibi söz konusu değerlerin birbirlerine eşit (equal) olduğunu da ifade etmektedir. Her durumda da netice aynıdır. JZ veya JE komutlarından hangisinin tercih edildiğinin sonuca bir etkisi olmamak ile birlikte kodu inceleyenin daha rahat anlaması ve uygulanan algoritmayı daha çabuk kavrayabilmesi için AX = BX koşulunun sonucuna bağlı olarak işlem yapılırken JE, (AX-BX) = 0 koşulunun sonucuna bağlı olarak işlem yapılırken JZ komutunun kullanılması daha uygun olacaktır.

5.3.1.1 Basit Koşullu Dallanma Komutları

Basit koşullu dallanma komutları ZF, SF, OF ve PF bayraklarından birinin durumuna göre dallanma yapılmasını sağlarlar.

JZ / JE 8086/286/386/486 jump zero / jump equal
JZ disp8 JE disp8
Açıklama: **CMP** işleminin sonucunun sıfır (eşitlik) olması durumunda, diğer deyişle ZF bayrağının değeri 1 ise dallanma gerçekleşir.

JNZ / JNE 8086/286/386/486 jump not zero / jump not equal
JNZ disp8 JNE disp8
Açıklama: **CMP** işleminin sonucunun sıfır (eşitlik) olmaması durumunda, diğer deyişle ZF bayrağının değeri 0 ise dallanma gerçekleşir.

JS 8086/286/386/486 jump if sign
JS disp8
Açıklama: **CMP** işleminin sonucunda elde edilen değer negatif ise[73] , diğer deyişle SF bayrağının değeri 1 ise dallanma gerçekleşir.

[73] Elde edilen değerin ikili sayı ifadesindeki en anlamlı bit'in 1 olması, sayının işaretli değerlendirilmesi durumunda negatif olduğunu belirler.

JNS	8086/286/386/486
jump no sign	
JNS disp8	
Açıklama: **CMP** işleminin sonucu pozitif ise, diğer deyişle SF bayrağının değeri 0 ise dallanma gerçekleşir.	

JO	8086/286/386/486
jump if overflow	
JO disp8	
Açıklama: **CMP** işleminin sonucunda aritmetik bir taşma söz konusu ise, diğer deyişle OF bayrağının değeri 1 ise dallanma gerçekleşir	

JNO	8086/286/386/486
jump no overflow	
JNO disp8	
Açıklama: **CMP** işleminin sonucunda aritmetik bir taşma oluşmamış ise, diğer deyişle OF bayrağının değeri 0 ise dallanma gerçekleşir.	

JP / JPE	8086/286/386/486
jump on parity / jump parity even	
JP disp8 JPE disp8	
Açıklama: **CMP** işleminin sonucunda çift (even) parity[74] oluşmuş ise, diğer deyişle PF bayrağının değeri 1 ise dallanma gerçekleşir.	

JNP / JPO	8086/286/386/486
jump no parity / jump parity odd	
JNP disp8 JPO disp8	
Açıklama: **CMP** işleminin sonucunda tek (odd) parity[75] oluşmuş ise, diğer deyişle PF bayrağının değeri 0 ise dallanma gerçekleşir.	

5.3.1.2 İşaretsiz Sayı İşlemlerine Ait Koşullu Dallanma Komutları

İşaretsiz sayılar karşılaştırılırken bir değerin diğerinin yukarısında (**A**bove) veya aşağısında (**B**elow) olmasına bakılır. Basit koşullu dallama komutlarında olduğu gibi bir bayrak ile sonuca ulaşanlar olduğu gibi farklı bayrakların birbirlerine göre

[74] İşlem sonucunda elde edilen değerin ikili ifadesinde yer alan 1'lerin sayısı çift ise

[75] İşlem sonucunda elde edilen değerin ikili ifadesinde yer alan 1'lerin sayısı tek ise

durumlarına göre dallanma yapanlar da vardır. Bu komutlarda bayrakların hangi değerleri aldığını öğrenmek yerine işlemin hangi amaca ulaşmak için yapıldığını öğrenmek daha doğru olacaktır.

JB / JNAE / JC	8086/286/386/486
jump below / jump not above equal / jump carry (**unsigned)**	
JB disp8 JNAE disp8 JC disp8	
Açıklama: **İşaretsiz sayılar** ile yapılan **CMP** işleminde opr1'in opr2'nin altında olduğu (üstünde veya eşit olmadığı/elde olduğu) durumda, diğer deyişle CF bayrağının değeri 1 ise dallanma gerçekleşir.	

JA / JNBE	8086/286/386/486
jump above / jump not below equal **(unsigned**)	
JA disp8 JNBE disp8	
Açıklama: **İşaretsiz sayılar** ile yapılan **CMP** işleminde opr1'in opr2'nin üstünde olduğu (altında veya eşit olmadığı) durumda, diğer deyişle CF bayrağının değeri 0 ve ZF bayrağının değeri 0 ise dallanma gerçekleşir.	

JAE / JNB / JNC	8086/286/386/486
jump above equal / jump not below / jump not carry **(unsigned)**	
JAE disp8 JNB disp8 JNC disp8	
Açıklama: **İşaretsiz sayılar** ile yapılan **CMP** işleminde opr1'in opr2'nin üstünde veya eşit olduğu (altında olmadığı/elde olmadığı) durumda, diğer deyişle CF bayrağının değeri 0 ise dallanma gerçekleşir.	

JBE / JNA	8086/286/386/486
jump below equal / jump not above **(unsigned)**	
JBE disp8 JNA disp8	
Açıklama: **İşaretsiz sayılar** ile yapılan **CMP** işleminde opr1'in opr2'nin altında veya eşit olduğu (üstünde olmadığı) durumda, diğer deyişle CF bayrağının değeri 1 veya ZF bayrağının değeri 1 ise dallanma gerçekleşir.	

5.3.1.3 İşaretli Sayı İşlemlerine Ait Koşullu Dallanma Komutları

İşaretli sayılar karşılaştırılırken bir değerin diğerinden büyük (**G**reater) veya küçük (**L**ess) olmasına bakılır. Bu komutların işleyişinde bayrakların birbirlerine göre durumlarına göre dallanma yapılır.

JL / JNGE 8086/286/386/486 jump less / jump not greater and equal **(signed)**
JL disp8 JNGE disp8
Açıklama: **İşaretli sayılar** ile yapılan **CMP** işleminde opr1'in opr2'den küçük olduğu (büyük eşit olmadığı) durumda, diğer deyişle SF ile OF bayraklarının değerleri farklı ve ZF bayrağının değeri 0 ise dallanma gerçekleşir.

JNL / JGE 8086/286/386/486 jump not less / jump greater and equal **(signed)**
JNL disp8 JGE disp8
Açıklama: **İşaretli sayılar** ile yapılan CMP işleminde opr1'in opr2'den küçük olmadığı (büyük eşit olduğu) durumda, diğer deyişle SF ile OF bayraklarının değerleri aynı ise dallanma gerçekleşir.

JLE / JNG 8086/286/386/486 jump less equal / jump not greater **(signed)**
JNL disp8 JGE disp8
Açıklama: **İşaretli sayılar** ile yapılan **CMP** işleminde opr1'in opr2'den küçük eşit olduğu (büyük olmadığı) durumda, diğer deyişle SF ile OF bayraklarının değerleri faklı ve ZF bayrağının değeri 1 ise dallanma gerçekleşir.

JG / JNLE 8086/286/386/486 jump greater / jump not less equal **(signed)**
JNL disp8 JGE disp8
Açıklama: **İşaretli sayılar** ile yapılan **CMP** işleminde opr1'in opr2'den büyük olduğu (küçük eşit olmadığı) durumda, diğer deyişle ZF bayrağının değeri 0 ve SF ile OF bayraklarının değerleri aynı ise dallanma gerçekleşir.

5.3.2 Koşulsuz Dallanma Komutları

Koşullu dallanma komutlarında atlama aralığının 8 bit ile (disp8) sınırlı olduğundan daha önce bahis etmiş, bu nedenle koşullu dallanma komutlarının ancak 128 byte geriye veya 127 byte ileriye ulaşmayı sağlayabildiğini belirtmiştik[76]. Bu sınırlamadan kurtulabilmek için koşulsuz dallanma komutları kullanılabilir. Koşullu dallanma komutları bayrakların değerine göre işlem yaparken koşulsuz dallanma komutları bayrakların değerlerine bağlı olarak işlem yapmaz, kullanıldıkları noktadan itibaren program akışını belirlenen noktaya aktarır.

Yüksek seviyeli dillerde kullanılan karşılaştırma işlemleri (IF-Then-Else), programın akışını iki ayrı kola bölmektedir. Buna göre programın akışını gerektiği gibi değiştirebilmek amacıyla koşullu ve koşulsuz dallanma komutları birbirlerini destekleyecek biçimlerde kullanılırlar.

JMP 8086/286/386/486

near jump

JMP disp
JMP reg
JMP mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:

Programın gereği olarak bulunulan kesim içinde bir noktaya erişilmesi isteniyorsa kesim içi dallanma komutu kullanılır. Buna göre;

JMP disp IP←IP + disp
JMP reg IP←reg
JMP mem IP←[mem]

şeklindeki kullanımlarda üretilen adres 16 bit değerini aşmadığı için aynı kesim içinde dallanma söz konusudur. Bu tür dallanmalar kesim içi (**intrasegment**) olarak değerlendirilir. Diğer deyişle sadece IP yazmacının değeri değişikliğe uğramaktadır.

Örnek:

JMP *new_label*	*new_label* isimli etiketin olduğu yere gidecek şekilde IP yazmacının değeri ayarlanacaktır.
JMP ECX	ECX yazmacının gösterdiği adrese dallanma gerçekleşecektir. Ancak kesim yazmacında bir değişiklik olmadığı için bu dallanma kesim içinde kalacaktır.
JMP 0128	Bulunulan kod kesimi içinde 0128H adresine atlanacaktır.[77]

[76] Koşullu dallanma komutlarının atlama aralıklarının disp8 ile belirlenmiş olması kullanıcıların aklına "*bu sınırı geçtiğimizi nasıl bileceğiz ?*" sorusunu getirmektedir. Aslında sorunun cevabı son derece basittir. Yazılan assembly dilindeki program derlenirken bu tür uygunsuzluklar tespit edildiğinde kullanıcı hata mesajı ile uyarılacaktır. Hatanın derleyici tarafından tespit edilmesini takiben problem önce koşullu dallanma komutu ile atlama aralığı içinde bir adrese atlamak, oradan da JMP komutu ile daha uzakta bulunan gerçek kod parçasına atlamak ile çözülecektir. Diğer yandan 127 byte yer tutacak assembly programın – her komut satırının bellekte yaklaşık olarak 3 byte yer işgal ettiğini kabul ederek- 40 satırdan fazla olacağını da göz önünde bulundurmak gerekir.

[77] Bu kullanım sadece debug ortamı için geçerlidir.

JMP FAR PTR 8086/286/386/486

far jump

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
JMP idata												
JMP mem	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Programın gereği olarak bulunulan kesim dışındaki bir noktaya erişilmesi isteniyorsa dallanma komutu. **FAR PTR** ile kullanılmalı ve akış başka bir kesime aktarılmalıdır. Buna göre;

JMP FAR PTR *label*

şeklinde bir kullanım söz konusu ise IP←OFFSET(*label*) ve CS←SEGMENT(*label*) değerlerini alacaktır. Bu tür dallanmalar kesimler arası (**intersegment**) olarak adlandırılır.

Örnek:

JMP 0AB0:0100	Verilen *idata* değerine göre CS←0AB0H ve IP←0100H değerlerini alacaklar ve kesimler arası dallanma gerçekleşecektir.
JMP FAR PTR *new_label*	*new_label* isimli etiketin olduğu kesime dallanma gerçekleşecektir. Ancak bu tür bir kullanım için etiketi barındıran kodun dallanmayı yapacak kod ile birlikte link edilip tek bir çalıştırılabilir kod üretilmesi gerekir.
JMP FAR PTR [0200]	7 0 0204H 0EEH 0203H 0DDH 0202H 0CCH 0201H 0BBH 0200H 0AAH Bu örnekte verilen 0200H adresinin gösterdiği noktadan önce IP, sonra CS yazmaçları değerlerini alacaktır. Sonuçta; IP←0BBAAH CS←0DDCCH değerlerine sahip olur.

Akış diyagramlarında sıkça karşılaştığımız IF-Then-Else, Null-Else ve Null-Then tipindeki yapılar koşullu ve koşulsuz dallanma komutlarının bir arada kullanılmasını gerektirmektedir. Bu durumları aşağıdaki akış diyagramları, sözde kod ve assembly kod karşılıkları ile incelemeye çalışalım.

Tablo 5-2'deki akışta If ifadesinin then ve else kollarınki bağımsız işlemler tamamlandıktan sonra her iki kol ortak bir noktada birleşmektedir. Sözde kodlamada rahatlıkla yapılan bu işlem assembly dilinde kollardan birinin sonunda kullanılan koşulsuz dallanma (JMP *nextblock*) komutu ile yapılır.

Tablo 5-2: If-then-else yapısının assembly kod eşdeğeri

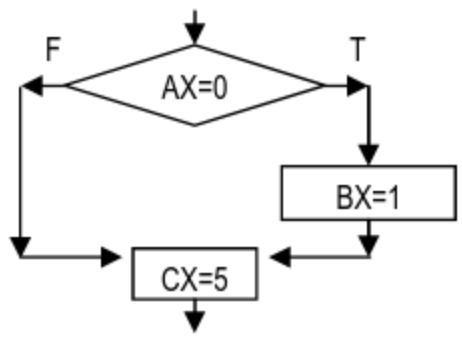
Akış Diyagramı	Sözde Kod	Assembly Kod

F, T, AX=0, BX=1, BX=2, CX=5	If (AX = 0) Then {
	BX = 2
	}
	Else {
	BX = 1
	}
	CX = 5
		CMP AX, 0
		JZ *thenblock*
		MOV BX, 1
		JMP *nextblock*
		thenblock: MOV BX, 2
		nextblock: MOV CX, 5

Tablo 5-3'deki akışta ise else kolu boştur (Null-Else). Tablo 5-2'dekine benzer bir uygulama ile Kod1 ile gösterilen assembly kodu yazmak mümkündür. Ancak daha az mnemonic kullanıp her bakımdan daha üstün bir assembly program yazabilmek için Kod2 ile gösterilen assembly kodda olduğu gibi koşul ifadesinin tersi (JNZ) kullanılarak then kolu içinde yer alan işlem bloğunun üzerinden atlanmış, else kolunda BX=1 işlemi yapılarak koşulun iki kolu *nextblock* isimli etikette buluşturulmuştur.

Tablo 5-3: Null else yapısının assembly eşdeğeri

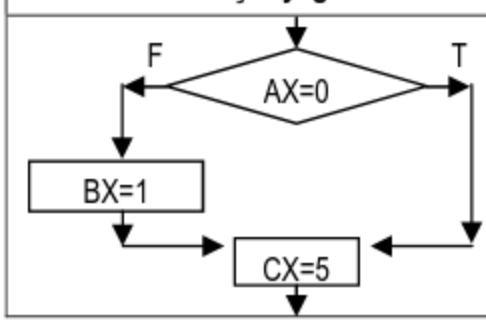
Akış Diyagramı	Sözde Kod	Assembly Kod

	If (AX = 0) Then {
	BX= 1
	}
	CX=5
		Kod:1
		CMP AX, 0
		JZ *thenblock*
		JMP *nextblock*
		thenblock: MOV BX, 1
		nextblock: MOV CX, 5
		
		Kod:2
		CMP AX, 0
		JNZ *nextblock*
		MOV BX, 1
		nextblock: MOV CX, 5

Tablo 5-4'deki akışta ise then kolu boştur. Bu durumda kullanacağımız koşullu dallanma komutu koşul ifadesi ile özdeş (JZ) olacak ve else kolunun üzerinden atlanarak *nextblock* isimli etiketten ortak işlemler yürütülecektir.

Tablo 5-4: Null then yapısının assembly eşdeğeri

Akış Diyagramı	Sözde Kod	Assembly Kod

	If (AX <> 0) Then {
	BX = 1
	}
	CX = 5
		CMP AX, 0
		JZ *nextblock*
		MOV BX, 1
		nextblock: MOV CX, 5

5.3.3 Çağırma Komutları

Dallanma komutlarına benzer olarak yordam çağırmak için kullanılan **CALL** ve kesme çağırmak için kullanılan **INT** komutları da programın akışını değiştirirler. Ancak dallanma komutlarından farklı olarak **CALL** ve **INT** komutları geri dönmeyi amaçlar. Diğer bir deyiş ile dallanma komutları salt gidiş, **CALL** ve **INT** komutlarını ise gidiş-dönüş bileti olarak yorumlanabilir. Gidilen yerden dönülebilmesi için dönüş adresinin bilinmesi gerekir. Dönülmesi gereken adres, gidişi sağlayan komutu takip eden komut

satırının adresi olmalıdır. Ancak bu adresin gidiş işlemi sırasında belirlenmesi ve uygun bir şekilde dönüş anına kadar yığın kesiminde saklanması gerekmektedir.

CALL

8086/286/386/486

near procedure call

CALL disp
CALL mem
CALL reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
CALL komutu, çağırılan yordamın sonlanmasını takiben (**RET** komutunun çalıştırılması ile) kendisini takip eden kod satırına geri döner. Burada "*hangi adrese dönüleceği nasıl biliniyor?*" sorusu akla gelir. İşte bu noktada **CALL** komutunun farklı bir özelliği ortaya çıkmaktadır. **CALL** komutu, önce geri dönüş adresini **yığın** üzerinde saklamakta, sonra gideceği yerin adresini **IP** yazmacına yerleştirip dallanmayı gerçekleştirmektedir. Dallanılan noktadaki **RET** komutunun çalıştırması ile dönüş değeri yığından alınarak **IP** yazmacına yüklenecek ve **CALL** komutunu takip eden komut satırına dönüş sağlanacaktır. Dönüş adresi yığında saklandığı için yığının program içinde doğru kullanımı önem kazanmaktadır. Yığından herhangi bir nedenle farklı bir değerin dönüş adresi olarak alınması programın kontrolden çıkması demektir. **NEAR CALL**, aynı kesim içinde bulunulan yordamları çağırmak için kullanılması nedeniyle kesim içi yordam çağırma (**intrasegment call)** olarak da adlandırılır.

Yapılan işlem: CALL disp IP←IP + disp
CALL reg IP←reg
CALL mem IP←[mem]

Örnek:

CALL *Sort*	*Sort* isimli yordam çağırılmaktadır.
CALL 15AB	15ABH göreli konumunda bulunan yordam çağırılmaktadır.

CALL FAR PTR

8086/286/386/486

far procedure call

CALL idata
CALL mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
CALL komutu hem kesim içi hem de kesimler arası kullanılabilir. **FAR CALL** komutu önce geri dönüş için gerekli olan kesim (CS) sonra göreli konum (IP) değerlerini yığına atar, sonra akışın aktarılacağı noktaya ait yeni kesim ve göreli konum değerlerini uygun yazmaçlara yerleştirerek dallanır. Çağırılan yordamdan dönüşü sağlayan **RETF** komutu ile yığından alınan göreli konum değeri IP, kesim değeri CS yazmaçlarına yerleştirilerek **CALL** komutunu takip eden komut satırına dönüş sağlanır. **FAR CALL**, kesimler arası yordam çağırma (**intersegment call)** olarak adlandırılır.

Örnek:

CALL 0AB0:0100	Verilen idata değerine göre CS←0AB0H ve IP←0100H değerlerini alır ve kesimler arası yordam çağırma gerçekleşir
CALL FAR PTR [0200]	7 0 0204H 05EH 0203H 04DH 0202H 03CH 0201H 02BH 0200H 01AH Bu örnekte verilen 0200H adresinin gösterdiği noktadan önce IP sonra CS yazmaçlarına değerler aktarılacaktır. Sonuçta IP←2B1AH ve CS←4D3CH değerlerini alacaktır. Bu durumda akış 4D3C:2B1A adresinden devam edecek, kodun sonundaki **RETF** komutunun çalıştırılması ile dönüş sağlanacaktır.

INT 8086/286/386/486

software interrupt

INT idata

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	0	0	--	--	--	--	--	--	--	--

Açıklama:
Kesme (Interrupt), yavaş birimler ile hızlı birimler arasındaki uyumluluğu sağlamak amacıyla kullanılan bir mekanizma ve bu amaçla çalıştırılan farklı kesimdeki bir yordamdır. İşlemin gereği olarak mevcut kesme yordamları uygulama içinden çağırılabileceği gibi kesmeler sistemin genel işleyişinin bir gereği olarak herhangi bir anda da kendiliğinden de devreye girebilir. **INT** komutunun çalışması ile önce bayraklar[78] ardından dönüş adresleri (CS, IP) yığında saklanır. Kesme programının göreli konum ve kesim değerinin IP ve CS yazmaçlarına yerleştirilmesi ile dallanma gerçekleşir. **CALL** komutunda olduğu gibi kesme yordamının tamamlanması ile (**IRET** komutu) yığından dönüş adresleri ve bayraklar alınarak kesmenin oluştuğu yere geri dönülür.

Yapılan işlem:

```
PUSHF
PUSH CS
PUSH IP
TF←0
IF←0
IP←0000:[idata * 4]        ; Vektör tablosundan[79] kesme yordamının göreli konum adresini alır.
CS←0000:[idata * 4 +2]     ; Vektör tablosundan kesme yordamının kesim adresini alır.
```

INTO 8086/286/386/486

interrupt on overflow

INTO

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	0	0	--	--	--	--	--	--	--	--

Açıklama:
Bu komut, OF bayrağının değeri 1 ise INT 04H (arithmetic overflow) çağırılmasını sağlar.

5.3.4 Dönüş Komutları

Çağırılan yordamdan dönülmesini sağlamak üzere kullanılan komutlar kısaca dönüş komutları başlığı altında toplanmıştır. Bunlar kullanılan yordam tipi ile uyumlu olarak çalışmakta ve uygun sayıda dönüş değerinin, doğru sırada yığından alınmasına imkân vermektedir. Benzer durum kesme servis programlarından dönüşte de söz konusu olmaktadır. Bu amaçla kullanılan dönüş komutları şunlardır;

---

[78] Kesme öncesinde işletilen komutlar bayraklar üzerinde değişikliğe neden olmuş olabilir. Bayrak değerleri kesme programının çalışması sırasında da değişecektir. Dönüşte işlemlerin kaldığı şekli ile ilerleyebilmesi için bu gereklidir.

[79] Kesmelerin işleyişi ve vektör tablosu yapısı 14. Bölümde ele alınacaktır.

RET

8086/286/386/486

near return from subroutine

8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
RET	--	--	--	--	--	--	--	--	--	--	--	--
RET idata												

Açıklama:
Kesim içinden çağırılan bir yordamdan dönmek için **RET** komutu kullanılır. **PROC** tanımı **NEAR** olduğu için yığından alınan değer IP/EIP yazmacına aktarılarak geri dönüş sağlanır.

Yapılan işlem : IP←POP( )
SP←SP + idata

Örnek:

RET	Yığından aldığı word değeri IP yazmacına yerleştirir. SP yazmacının değeri, yığın üzerinde gerçekleştirilen **POP** işleminin sonucu olarak 2 artacaktır.
RET 6	Yığından alınan değer IP yazmacına atanır. SP yazmacının değeri POP işleminin sonucunda 2 artacaktır. **RET** komutunun yanında verilen değer, SP←SP+6 işleminin yapılmasına, bu sayede çağıranın, çağırılana yığın üzerinden aktardığı parametrelerin otomatik olarak kaldırılmasına imkân verir.

RETF

8086/286/386/486

far return form subroutine

8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
RETF	--	--	--	--	--	--	--	--	--	--	--	--
RETF idata												

Açıklama:
Kesim dışından çağrılan yordamdan dönmek RETF[80] komutu kullanılır. **PROC** tanımı **FAR** olduğu için yığından alınan değerler sırası ile IP ve CS yazmaçlarına aktarılarak geri dönüş sağlanır

Yapılan işlem: IP←POP( )
CS←POP( )
SP←SP + idata

Örnek:

RETF	Yığından sırası ile alınan word değerler IP ve CS yazmacına aktarılır.
RETF 6	Yığından alınan değerler IP ve CS yazmaçlarına alınır. SP yazmacının değeri POP işlemlerinin sonucunda 6 artacaktır. **RET** komutunun yanında verilen idata[81] değeri, SP←SP+6 işleminin yapılmasına, bu sayede çağıranın, çağırılana yığın üzerinden aktardığı parametrelerin otomatik olarak kaldırılmasına imkân verir.

[80] FAR tanımlanmış olan yordam RET komutu ile bitirilmiş dahi olsa derleyici bunu RETF haline dönüştürecektir.

[81] Komutu takip eden sayı SP yazmacının değerine eklenir. Bu sayede yığına parametre olarak saklanmış değerler yığının dışında kalmış (anlamını yitirmiş) olur. Yığına word büyüklüğünde veri sakladığı için *idata* değerinin her zaman parametre sayısının 2 katı olması gereklidir.

IRET 8086/286/386/486

return from interrupt 8/16/32 Bit

IRET

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	x	x	x	x	x	--	x	--	x	--	x

Açıklama:
Oluşan kesme isteklerine işlemcinin hizmet vermesi için yapmakta olduğu işi (geri döndüğünde kaldığı yerden devam edebilecek şekilde) belirli bir süre için bırakması gerekmektedir. Farklı bir kesimde yer alan yordam olarak yorumlayabileceğimiz kesme servis programının sona ermesi **IRET** komutunun işletilmesi ile mümkündür. **IRET** komutu sayesinde yığında saklanmakta olan dönüş adresleri ile bayrak değerleri yığından alınır.

Yapılan işlem : IP←POP( )
CS←POP( )
Flag←POPF( )

5.4 Döngü Komutları

Tekrarlanan cinsten işlemlerde kullanılan döngü komutlarının en büyük özelliği, CX yazmacı ile işlemin kaç defa tekrarlanacağını belirlemeleridir. Ancak yüksek seviyeli dillerden alışıldığının aksine çevrim artarak değil, azalarak ilerler. LOOP komutlarının çalışması bayraklar üzerinde değişikliğe neden olmaz[82].

LOOP 8086/286/386/486

loop until complete

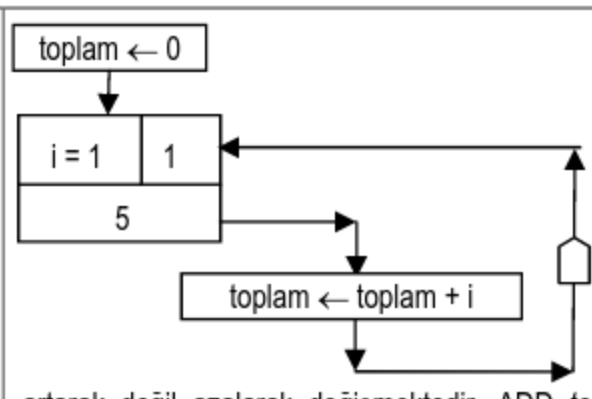
LOOP disp

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
LOOP komutu her çalıştırıldığında CX←CX – 1 işlemi yapılır, CX <> 0 olduğu sürece disp ile belirlenen adresten program çalışmaya devam eder. CX = 0 olduğu anda işlem **LOOP** komutunun altındaki komut satırından çalışmaya devam eder.

Örnek:

```
        MOV toplam, 0
        MOV CX, 5
        MOV SI, 1
L1:     ADD toplam, SI
Next:   INC SI
        LOOP L1
```



Yandaki akışta bir 1-5 arasındaki sayıların toplamı hesap edilmektedir. Algoritmanın doğru çalışabilmesi için önce *toplam* değişkeni sıfırlanır, işlemin tekrar sayısı CX yazmacına konur. Assembly dilinde CX yazmacı çevrim değişkeni olarak kullanılmaz. Zira CX, akış diyagramındaki *i* değişkenin aksine artarak değil azalarak değişmektedir. ADD *toplam*, SI işlemindeki *toplam* isimli değişkenin SI ile birlikte kullanılabilmesi için word tipinde tanımlı olması gerekir.

[82] Benzer olarak 5.9. bölümdeki REP, REPE/REPNE, REPZ/REPNZ ön eklerini de inceleyiniz.

LOOPZ / LOOPE

8086/286/386/486

loop zero / loop equal

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
LOOPZ disp LOOPE disp	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
LOOPZ/LOOPE komutu her çalıştırıldığında CX←CX – 1 işlemi yapılır, CX <> 0 ve ZF bayrağının değeri 1 ise işlem tekrarlanır. Koşullardan herhangi birinin sağlanamaması durumunda çevrim sona ermiş olacaktır. Program **LOOPZ/LOOPE** komutunu takip eden komut satırından çalışmaya devam edecektir.

LOOPNZ / LOOPNE

8086/286/386/486

loop not zero / loop not equal

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
LOOPNZ disp LOOPNE disp	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
LOOPNZ / LOOPNE komutu her çalıştırıldığında CX←CX – 1 işlemi yapılır, CX <> 0 ve ZF bayrağının değeri 0 ise işlem tekrarlanır. Koşullardan herhangi birinin sağlanamaması durumunda çevrim sona ermiş olacaktır. Program **LOOPNZ/LOOPNE** komutunu takip eden komut satırından çalışmaya devam edecektir.

JCXZ

8086/286/386/486

Jump cx zero

8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
JCXZ disp	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Her ne kadar genel yapısı itibariyle bir koşullu dallanma komutu olarak görülse de, çevrim işlemlerinde kullanılan CX yazmacı üzerindeki kontrol özellikleri dolayısıyla bu komutun da döngü amaçlı kullanıldığından bahsedebiliriz. Komut CX=0 ise dallanma yapmayı sağlamaktadır[83]. Ancak **LOOP** komutlarından farklı olarak CX yazmacının değeri otomatik olarak azaltılmaz. Bu özellik, kullanıcının çevrim değişkenin değerini kendisinin belirleyeceği şekilde arttırıp/azaltabilmesi gibi bir seçenek sağlamaktadır.

LOOP komutlarında çevrim sayısı CX yazmacındaki değer ile belirlenirken bunun değişim miktarı her zaman bir azaltma olup kullanıcı tarafından değiştirilemez. Diğer bir değiş ile **LOOP** komutu PASCAL programla dilinde kullanılan For-Downto yapısına benzemektedir. Yüksek seviyeli dillerde farklı artırma/azaltma değerlerine sahip For yapılarının yanı sıra While ve Repeat-Until gibi farklı kontrollü çevrim yapılarından da yararlanılır. Ancak assembly dilinde bu tarz komutlar yoktur.

[83] JCXZ *label1* komut kendi içinde bir karşılaştırma (CMP CX, 0) işlemi yapmakta ve bunun sonucunda eşitlik/sıfır olma koşulu sağlandı ise dallanmakta (JE *label1* / JZ *label1*) aksi halde kendini takip eden satır ile devam etmektedir.

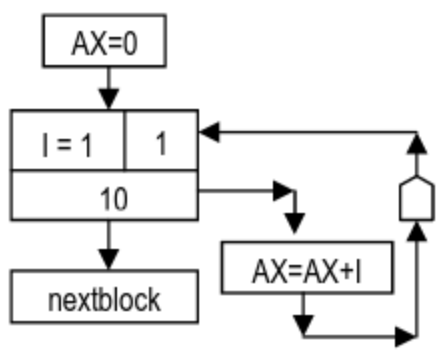
Bu gibi ihtiyaçlara cevap vermek üzere CMP ve koşullu/koşulsuz dallanma komutlarının uygun birleşimleri ile programcının kendi kontrollü çevrimini oluşturması gerekecektir. Genel programlama tekniği açısından For yapısı (LOOP komutunda olduğu gibi) çevrim değişkenin her turda belirlenen sabit artırma/azaltma kadar değişerek belirlenen son değere ulaşıncaya kadar işlemlerin tekrar edilmesi için oluşturulmuş bir yapıdır. **For kullanımında çevrim değişkenine, çevrim içinde programcı tarafından müdahale edilmemelidir**. Çevrim değişkeninin çevrim içinde programcının kontrolünde değişmesi isteniyor, diğer bir deyiş ile çevrimin tekrar sayısı çevrime başlamadan önce bilinmiyor ve/veya çevrim içinde yapılan işlemlerin bir sonucu olarak değişebiliyor ise While veya Repeat-Until tarzı kontrollü çevrim yapılarının kullanılması genel bir programlama kuralıdır. Koşullu çevrim yapılarında -For yapısından farklı olarak- çevrim değişkeninin değeri programcı tarafından değiştirilmelidir.

Bu kısımda assembly programlarımızı yazarken karşılaşabileceğimiz bazı durumları akış diyagramları, sözde kod karşılıkları ve assembly dili kullanılarak kodlanmış şekillerini inceleyerek açıklayalım.

Tablo 5-5'deki akış diyagramı 1-10 arasındaki sayıları toplamak üzere oluşturulmuştur. Çevrimin tamamlanması ile *nextblock* olarak isimlendirilmiş kısım çalışacaktır. Kod analiz edilirse sözde kodda çevrim içinde kalan *AX* değişkeni sırası ile 1, 3, 6, 10…55 değerlerini alırken, Kod1 ile gösterilen assembly kodda tekrarlanan kısmı içinde kalan AX yazmacı ise sırası ile 10, 19, 27, 34…55 değerlerini almaktadır. Bunun nedeni, sözde kodda çevrim değişkeninin 1 den başlayarak 10 doğru ilerlemesi iken, Kod1 ile gösterilen assembly kodda çevrim değişkenini tutan CX yazmaç değerinin 10 dan başlayarak 1'e doğru gerilemesidir.

Tablo 5-5: For-To yapısı

Akış Diyagramı	Sözde Kod	Assembly Kod

	*AX* = 0
	For *I*=1 to 10 Do
	AX= *AX* + *I*
	Next *I*
	nextblock
		Kod:1
		MOV AX, 0
		MOV CX,000AH
		innerfor: ADD AX, CX
		LOOP *innerfor*
		nextblock:
		
		Kod:2
		MOV AX, 0
		MOV SI, 1
		MOV CX, 000AH
		innerfor: ADD AX, SI
		INC SI
		LOOP *innerfor*
		nextblock:

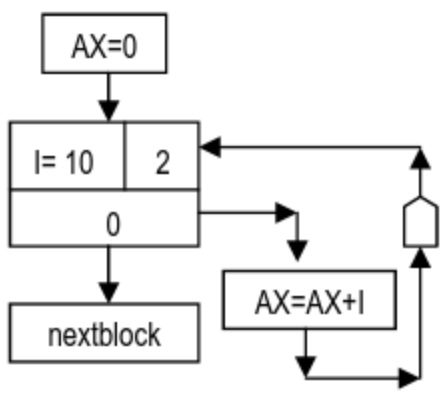
Kod2 ile gösterilen assembly kod parçasında CX yerine SI yazmacı sözde kodda kullanılan *I* isimli çevrim değişkeninin yerine kullanılmış, ilk değeri çevrime başlamadan önce 1 olarak belirlenmiş ve her toplama işlemini takiben bir arttırılarak (INC SI) işlem gerçekleştirilmiştir. Bu düzenlemeden sonra Kod2 ile gösterilen assembly kodun tekrarlanan kısmı içinde kalan AX yazmaç değeri artık sözde kodda olduğu gibi 1, 3, 6, 10…55 şeklinde değişmektedir.

Yüksek seviyeli dillerde (özellikle dizi yapısındaki) değişkenler ile işlem yapılırken çevrim değişkenlerinin indis olarak kullanılması bir alışkanlık olmak ile birlikte bu assembly programlarda doğru değildir. Zaten CX yazmacı hiçbir zaman doğrudan dizi adreslemek[84] amacıyla tasarlanmamıştır.

Assembly dilinde **LOOP** komutu her ne kadar For-Downto yapısına uygun çalışıyor bile olsa çevrim adedini tutan CX yazmacı üzerindeki azaltma birer birer yapılabilir. Bu azaltma değeri programcı tarafından değiştirilemeyeceği için çevrim sayısının tam olarak bilinemediği veya çevrim değişkeninin farklı şekillerde değişmesinin gerektiği problemlerin çözümünde LOOP komutu ihtiyaca cevap veremez.

Tablo 5-6: For –Downto yapısı

Akış Diyagramı	Sözde Kod	Assembly Kod

	*AX* = 0
	For *I*=10 downto 0 Step 2 Do
	AX=AX + *I*
	Next *I*
	nextblock
		Kod:1
		MOV AX, 0
		MOV SI, 000AH
		label1: ADD AX, SI
		SUB SI, 002H
		CMP SI, 0
		JG *label1*
		nextblock:
		
		Kod:2
		MOV AX, 0
		MOV CX, 000AH
		label1: ADD AX, CX
		SUB CX, 002H
		JCXZ *nextblock*
		JMP *label1*
		nextblock:

Tablo 5-6'daki akış diyagramında 10-0 arasındaki çift sayıların toplamı hesaplanmaktadır. Sözde kod analiz edildiğinde *AX* değişkeninin çevrim içindeki ara değerlerinin 10, 18, 24,…30 olacağı görülür. Kod1 ile gösterilen assembly kodda SI yazmacı, sözde koddaki *I* değişkeninin yerini almıştır. Toplam değerinin hesap-

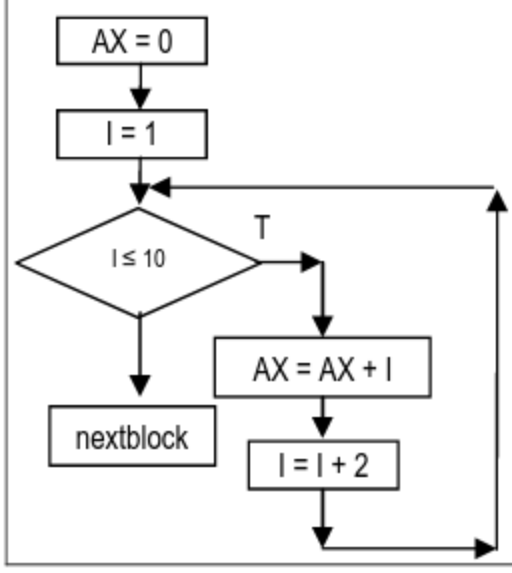
[84] Bellek üzerindeki dizi tipinde bir veri alanına erişmek için SI, DI ve BX yazmaçlarından indis olarak yararlanılmalıdır. Dizi işlemlerinde bir sonraki elemana ulaşılmak üzere indisler arttırılırken/azaltılırken dizinin veri tipine bakılarak indisin artım/azaltım miktarına karar verilir. Örneğin dizi byte tipinde tanımlanmış ise indis değişkeni bir arttırılıp/azaltılırken word tanımlı dizilerde iki artım/azatlım yapılmalıdır.

lanmasını takiben, SI yazmaç değeri iki azaltılıp varılması beklenen son değer olan 0 ile karşılaştırılmakta (CMP SI, 0), SI değeri büyük[85] ise (JG *label1*) çevrimin başına dönülmekte, aksi halde *nextblock* ile devam edilmektedir. Kod2 ile gösterilen assembly kodda, CX yazmacı çevrimin üst sınırını tutmak üzere kullanılmış olmasına karşın **LOOP** komutu yerine çevrim işlemlerinde kullanılan (aslında bir koşullu dallanma komutu olan) **JCXZ** komutundan yararlanılmaktadır. JCXZ, CX yazmacının değeri sıfırdan farklı olduğu sürece yanında yazan *nextblock* isimli etikete dallanacaktır. Koşulun sağlanmaması durumunda kendisini takip eden komut satırı (JMP *label1*) çevrimi başlangıç noktasına döndürecektir. Her iki assembly kod, kontrol sonda yapıldığı için, Repeat-Until tarzı kontrollü bir çevrime benzemektedir.

While yapısı, çevrim koşulun önceden kontrol edilerek çevrime başlandığı ve çevrim değişkeninin çevrim içinde programcının belirlediği şekilde değişebileceği bir yapıyı kurmamızı sağlar. While yapısında, For yapısından farklı olarak çevrim değişkeninin değeri otomatik olarak değişmediği için programcı tarafından, çevrim içinde belirlenir.

Tablo 5-7: While yapısı

Akış Diyagramı	Sözde Kod	Assembly Kod

	*AX* = 0
	I = 1
	While (*I* <=10) Do
	AX= *AX* + *I*
	I= *I*+2
	End
	nextblock
		MOV AX, 0
		MOV SI, 1
		label1: CMP SI, 000AH
		JA *nextblock*
		ADD AX, SI
		ADD SI, 2
		JMP *label1*
		nextblock:

Tablo 5-7'daki akış diyagramında 1-10 arasındaki tek sayıları toplayan bir algoritma görülmektedir. Buna göre çevrim değişkeni olan *I* değişkeni ilk değer olarak 1 ile çevrime başlamış ve çevrim içinde iki artım ile ilerlemiştir. Bu işlem yüksek seviyeli dillerde benzer şekilde For komutu kullanılarak da yapılabilmektedir. Ancak assembly programlamada bu işin LOOP komutu ile yapılmaya

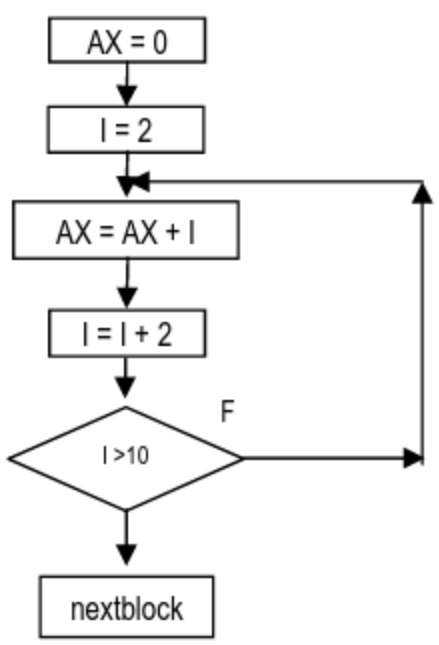
[85] Bu örneğimizde SI yazmacı iki azaltıldığı için bir süre sonra SI yazmacındaki değerin sıfırın altına düşmesi (negatif sayı olması) beklenir. Burada kontrolün işaretsiz sayılar için kullanılan JAE komutu ile yapılması sayının negatif olduğunuzu anlamamıza mani olacağı için uygun değildir. JAE kullanılması mantık hatasına neden olur.

çalışılması son derece anlamsızdır. Her ne kadar bu örneğimizde işlem tekrar sayısı hesaplanabilir olsa da çoğu zaman böyle bir durum söz konusu olmaz. Bu duruma assembly dilinde koşullu/koşulsuz dallanma komutlarını kullanarak bir çözüm üretebiliriz. Assembly kod örneğimizde SI yazmacı *I* değişkeninin yerine kullanılmıştır. Çevrim, SI yazmacındaki değerin 10 ile karşılaştırılması ile başlayıp, 10 değerini aşmasıyla *nextblock* isimli etikette dallanarak (JA *nextblock*[86]) devam edilmektedir. Koşulun sağlanması durumunda toplama işlemini takiben SI değeri iki arttırılıp (ADD SI, 2) çevrimi kontrol eden ifadenin olduğu satıra koşulsuz dallanma (JMP *label1*) yapılacaktır.

Repeat-Until önce çevrime girip çevrim koşulunu sonda kontrol eden bir yapıdır. Bu özelliği ile While'dan farklı olarak tekrarlanan kod kısmı en az bir defa çalıştırılmış olacaktır. Repeat-Until yapısında, For yapısından farklı olarak çevrim değişkeninin değeri otomatik olarak değişmediği için programcı tarafından, çevrim içinde belirlenir.

Tablo 5-8: Repeat-Until yapısı

Akış Diyagramı	Sözde Kod	Assembly Kod

	*AX* = 0
	I = 2
	Repeat
	AX= *AX* + *I*
	I = *I* +2
	Until (*I* >10) Do
	nextblock
		MOV AX, 0
		MOV SI, 2
		labe1: ADD AX, SI
		ADD SI, 2
		CMP SI, 10
		JBE *label1*
		nextblock:

Tablo 5-8'deki akış diyagramında 2-10 arasındaki çift sayıları toplayan bir algoritma görülmektedir. Buna göre çevrim değişkeni olan *I* değişkeni ilk değer olarak 2 ile çevrime başlamış, toplamı tutacak olan *AX* değişkenine eklendikten sonra iki artırılıp kontrol edilmiştir. Repeat-Until yapısında koşul ifadesinin yanlış (False) ol-

86 Örneğimiz toplanacak sayıların pozitif olduğu varsayımı üzerine kuruldur. Hal böyle olunca SI yazmacındaki değer de sürekli artarak ilerleyecektir. Bu nedenle koşul ifadesinde işaretsiz sayıları kullanarak işlem yapma imkânı sağlayan JA komutu kullanılmıştır.

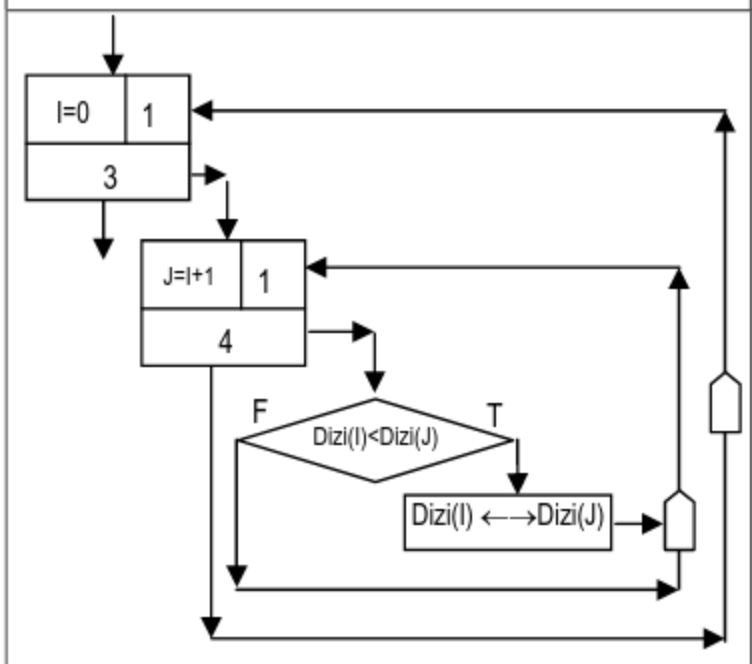
ması durumunda dönüş sağlanmakta (JBE *label1*[87]), aksı halde *nextblock* isimli kod alanı çalıştırılmaktadır.

Programlamada zaman zaman iç içe geçmiş çevrimler kullanmayı gerektiren problemler ile karşılaşırız. Bu gibi durumlarda çevrimlerin farklı olabilecek tekrar sayılarının tek bir CX yazmacı üzerinde tutulamayacağı düşüncesi/endişesi, LOOP komutunun felsefesini kavrayamamış kullanıcıları çoğu zaman daha karmaşık çözümler uygulamaya ve hata yapmaya itmektedir. Daha önce de belirttiğimiz gibi CX yazmacı sadece çevrim sayısını saklamak üzere kullanılmaktadır ve basitçe yığın kullanılarak farklı çevrimlerin çevrim sayıları birbirine karıştırılmadan saklanarak problem çözülebilmektedir.

Tablo 5-9'deki akış diyagramında 5 byte elemanlı bir diziyi büyükten- küçüğe doğru sıralamak için kullanılan kabarcık sıralama (bouble sort) algoritması görülmektedir[88].

Tablo 5-9: İç içe çevrimli yapısının assembly eşdeğeri (kabarcık sıralama)

Akış Diyagramı	Sözde Kod	Assembly Kod

	For *I*=0 to 3 Do
	For *J*=(*I*+1) to 4 Do
	If (*Dizi(I)*<*Dizi(J)*)Then
	Temp=Dizi(I)
	Dizi(I)=Dizi(J)
	Dizi(J)=Temp
	Next *J*
	Next *I*
		MOV CX, 4
		MOV SI, 0
		L2: **PUSH CX**
		MOV DI, SI
		L1: INC DI
		MOV AX; *Dizi*[SI]
		CMP AX, *Dizi*[DI]
		JGE *Next*
		XCHG AX, *Dizi*[DI]
		MOV *Dizi*[SI], AX
		Next: LOOP *L1*
		POP CX
		INC SI
		LOOP *L2*

Akış diyagramından da görüldüğü gibi *J* değişkeni iç kısımdaki çevrimin, *I* değişkeni ise dış çevrimin kontrolünde kullanılmaktadır. Assembly kodda da *I* değişke-

87 Örneğiniz toplanacak sayıların pozitif olduğu varsayımı üzerine kuruludur. Hal böyle olunca SI yazmacındaki değer de sürekli artarak ilerleyecektir. Bu nedenle koşul ifadesinde işaretsiz sayıları kullanarak işlem yapma imkânı sağlayan koşul ifadeleri kullanılması gereklidir. Repeat-Until yapısında koşul yanlış ise dallanma olacağı için akış diyagramında gösterilen > ifadesi yerine (tersini ifade eden) JBE komutu kullanılmıştır.

88 Dizi işlemlerinde ilk indisinin 0 veya 1 olması kullanılan programla diline ait bir özelliktir. Assembly dili dizi işlemlerinde 0. adresi ilk adres olarak kullanır. Örneğimizde akış ile kod arasında birliktelik sağlamak ve anlaşılırlığı sağlamak üzere çevrimler 1-5 arasında değil 0-4 arasında tanımlanmıştır. Her durumda da *Dizi* 5 elemanlıdır.

ninin yerini SI yazmacı, *J* değişkeninin yerini DI yazmacı almıştır[89]. Bu sayede çevrimin tekrar sayısı ile *Dizi* üzerindeki indisler birbirinden ayrılmıştır. Assembly kod incelendiğinde *L1* etiketinin iç, *L2* etiketinin dış çevrimin başladığı noktalar olduğu görülmektedir. Buna göre *L2* etiketindeki komut ile (PUSH CX) dış çevrimin tekrar sayısı yığına saklanmıştır[90]. İç çevrimin bitimi ve dönüşü sağlayacak LOOP *L1* komutundan hemen sonra da yığında saklanmakta olan dış çevrimin tekrar sayısı geri alınmış (POP CX) ve dış çevrim ile ilgili işlemlere devam edilerek LOOP *L2* komutu ile çevrim tamamlanmıştır.

5.5 Bayraklar ile İlgili Komutlar

Daha önceki bölümlerde komutların işleyişini açıklarken, işlemin bayraklar üzerinde ne şekilde etli edeceğine de vurgu yapmıştık. Ancak bazı komutların istenen şekilde çalışabilmesi için ön koşul olarak bayrakların değerlerinin ayarlanması da gerekebilmektedir. Örneğin dizgi işlemlerinde DF bayrağının durumu işlemin yönünü belirlemesi nedeniyle çok önemlidir. IF bayrağının durumu, oluşacak kesmelerin nasıl değerlendirileceğini belirleyecektir. CF bayrağının durumu, aritmetik işlemlerde ve CF bayrağını kullanarak çalışan döndürme (rotate) işlemlerinde son derece önemlidir. O zaman programcının ihtiyaç duyduğunda bayrakların değerlerini değiştirebilmesi için bazı komutların da komut kümesinde bulunması gereklidir. Bayraklar ile ilgili komutlar bu bölümde sıralanmıştır.

CLC 8086/286/386/486

clear carry flag

CLC

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	0

Açıklama:
CF←0 işlemi gerçekleştirilir.

CMC 8086/286/386/486

complement carry flag

CMC

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	x

Açıklama:
CF←CF′ işlemi (tümleme) gerçekleştirilir. Diğer değişle CF bayrağı 1 ise 0, 0 ise 1 haline getirir.

---

89 Dizi tipindeki veri yapılarını kullanan uygulamalarda, CX yazmacı hiçbir zaman -yüksek seviyeli dillerdeki- çevrim değişkenleri gibi dizi indisi olarak kullanılamaz. CX'in görevi çevrimin tekrar sayısını tutmaktır. İndislemede işlemci üzerinde bu amaçla bulunan SI, DI veya BX yazmaçları kullanılır.

90 İç çevrimin tekar sayısı dış çevrimin tekrar sayısına eşit olduğu için iç çevrime başlamadan önce CX yazmacının değerinin değiştirilmesi gerekmez.

STC

8086/286/386/486

set carry flag

STC

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	1

Açıklama:
CF←1 işlemi gerçekleştirilir.

CLD

8086/286/386/486

clear direction flag

CLD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	0	--	--	--	--	--	--	--	--	--	--

Açıklama:
DF←0 işlemi gerçekleştirilir. DF bayrağının durumu dizgi işlemlerinin yönünü belirlemektedir. DF, sıfır ise (SI, DI yazmaçlarıyla belirlenen) adresler artarak ilerler. Diğer deyişle veri aktarımı küçük adresten büyük adrese doğru yapılır.

STD

8086/286/386/486

set direction flag

STD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	1	--	--	--	--	--	--	--	--	--	--

Açıklama:
DF←1 işlemi gerçekleştirilir. DF bayrağının durumu dizgi işlemlerinin yönünü belirlemektedir. DF, bir ise (SI, DI yazmaçlarıyla belirlenen) adresler azalarak ilerler. Diğer değişle veri aktarımı büyük adresten küçük adrese doğru yapılır.

STI

8086/286/386/486

set interrupt flag

STI

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	1	--	--	--	--	--	--	--	--	--

Açıklama:
IF←1 işlemi gerçekleştirilir. Bu donanım kesmelerine izin (enable) verilmesini sağlar.

CLI 8086/286/386/486

clear interrupt flag

CLI

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	0	--	--	--	--	--	--	--	--	--

Açıklama:
IF←0 işlemi gerçekleştirilir. Bu, NMI (Non Maskable Interrupt) dışındaki donanım kesmelerine izin verilmesini engeller.

Bayraklara ilgili ilk grup komutlardan da görüldüğü gibi sık ihtiyaç duyulan bayrakların (CF, DF ve IF) değerleri bağımsız komutlar yardımıyla değiştirilebilmektedir. Ancak daha seyrek ihtiyaç duyulan bayrakların (SF, ZF, AF ve PF) değerlerinin sorgulanması ve/veya değiştirilmesi için ayrı ayrı komutlar üretmek yerine bunu AH yazmacı üzerinden yapmaya imkân sağlayan bazı komutlar da vardır[91].

LAHF 8086/286/386/486

load AH with flag 8 Bit

LAHF

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bayrak değerlerinin AH yazmacına aktarılmasını sağlayan bir komuttur. Bu sayede birkaç bayrağın durumunu bir seferde kontrol etmek mümkün olabilir. Bayraklar AH yazmacında şu şekilde yer alırlar;
Yapılan işlem: AH←flag

SF	ZF	?	AF	?	PF	?	CF

SAHF 8086/286/386/486

store AH in flag 8 Bit

SAHF

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	x	x	--	x	--	x	--	x

Açıklama:
AH yazmacında bulunan değer ile ilgili bayrak değerlerinin tek seferde ayarlanmasını sağlayan bir komuttur.
Yapılan işlem: flag← AH

SF	ZF	0	AF	0	PF	0	CF

5.6 Mantıksal Komutlar

İşlenenler üzerinde istenilen mantıksal işlemin yapılmasını sağlayan komutlar bu grup altında toplanmıştır.

---

91 PUSHF/POPF komutlarını kullanmak kaydı ile bayrak değerleri yığına konup/alınacak bir word ile bir seferde ayarlanabilir.

NOT

8086/286/386/486

not

8/16/32 Bit

NOT reg
NOT mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Tek işleneni olan bir komuttur ve verilen işlenenin bit bazında tümleyeni-ni almayı sağlar.
Yapılan işlem: opr1←opr1′

opr	NOT opr
0	1
1	0

Örnek:

NOT AX	İşlemden önce AX←00FFH ise sonra AX←FF00H olur.

OR

8086/286/386/486

boolean or

8/16/32 Bit

OR reg, idata
OR mem, idata
OR reg, reg
OR reg, mem
OR mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
0	--	--	--	x	x	--	?	--	x	--	0

Açıklama:
Verilen iki işlenene bit bazında **OR** işlemini uygular. Bit bazında olası girişler ve bunların **OR** işlemi sonucunda üretecekleri sonuçlar sağdaki tabloda verilmiştir.
Yapılan işlem: opr1←opr1 OR opr2

opr1	opr2	opr1 OR opr2
0	0	0
0	1	1
1	0	1
1	1	1

Örnek:

OR AL, 80H	AL yazmacını 1000000B ile OR işlemine sokarak en anlamlı bit'ini 1 yapar.

AND

8086/286/386/486

boolean and

8/16/32 Bit

AND reg, idata
AND mem, idata
AND reg, reg
AND reg, mem
AND mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
0	--	--	--	x	x	--	?	--	x	--	0

Açıklama:
Verilen iki işlenene bit bazında **AND** işlemini uygular. Bit bazında olası girişler ve bunların **AND** işlemi sonucunda üretecekleri sonuçlar sağdaki tabloda verilmiştir.
Yapılan işlem: opr1←opr1 AND opr2

opr1	opr2	opr1 AND opr2
0	0	0
0	1	0
1	0	0
1	1	1

Örnek:

AND AL, 0FH	AL yazmacının yüksek anlamlı 4 bit'ini (nibble) maskeler (sıfır yapar).
AND BYTE PTR[0200], 7F	[0200H] adresindeki byte'ı 0111111B ile AND işlemine gireren anlamlı bit sıfırlanır.

XOR

8086/286/386/486

boolean exclusive or

8/16/32 Bit

XOR reg, idata
XOR mem, idata
XOR reg, reg
XOR reg, mem
XOR mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
0	--	--	--	x	x	--	?	--	x	--	0

Açıklama:

Verilen iki işlenene bit bazında **XOR** işlemini uygular. Bit bazında olası girişler ve bunların **XOR** işlemi sonucunda üretecekleri sonuçlar sağdaki tabloda verilmiştir.

Yapılan işlem: opr1←opr1 XOR opr2

opr1	opr2	opr1 XOR opr2
0	0	0
0	1	1
1	0	1
1	1	0

Örnek:

XOR AL, 0FFH	AL yazmacındaki bit'lerden 1 olanlarını 0, 0 olanlarını da 1 yapar (**NOT** gibi)
XOR AX, AX	Tanımı gereği aynı değerde bitler birbiri ile XOR işlemine sokulduğunda sonuç 0 çıkacaktır. İşlem soncu itibari ile MOV AX, 0 ile aynı sonucu üretmiş olmak ile birlikte İşlem süresi ve üretilen kod büyüklüğü dikkate aldığında hem daha kısa zamanda hem de daha az bellek kullanarak bu sonuca ulaşır[92].

TEST

8086/286/386/486

test bits

8/16/32 Bit

TEST reg, idata
TEST mem, idata
TEST reg, reg
TEST reg, mem
TEST mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
0	--	--	--	x	x	--	?	--	x	--	0

Açıklama:

TEST işlemi **AND** işlemi gibi çalışmasına rağmen işlenenler üzerinde diğer mantıksal komutlarda olduğu gibi değişiklik yapmaz, sonuç sadece bayraklarda oluşur. Bu nedenle hemen arkasından koşullu dallanma komutu kullanılarak işlem yapılır.

Örnek:

TEST AL, 0FH JZ *Sifir*	AL yazmacının alçak anlamlı 4 bit'inin (nibble) hepsinin 1 olup olmadığını anlamak için kullanılır. İşlem sonucunda ZF'nin değeri 1 olmuş ise tüm bitler 1 değildir. ZF'nin değeri 0' ise düşük anlamlı 4 bit değeri 1 demektir.
TEST WORD PTR [0200], 8000 JNZ *İsaretli*	[0200H] adresindeki word'ün en anlamlı bit'inin 1 olup olmadığını anlamak için kullanılır. İşlem sonucunda ZF'nin değeri 1 olmuş ise en anlamlı bit 0, ZF'nin değeri 0' ise en anlamlı bit 1 demektir.

[92] Her komutun çalışması için gereken çevrim (cylce) zamanı ve bellek miktarı değişiklik gösterir. Örneğin MOV AX, 0 komutu 4 çevrim zamanı ve 3 byte bellek kullanılırken, XOR AX, AX komutu 3 çevrim zamanı ve 2 byte bellek kullanarak aynı sonucu üretirler. Her ne kadar gerek çevrim zamanın gerek ise kullanılan bellek arasında sadece bir birim fark olsa da XOR komutu çevrim zamanında %25, bellek kullanımında ise %33 iyileştirme sağlamaktadır.

SETxxx

80386/486

set byte on condition

8 Bit

SETxxx dest

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:

Koşula bağlı olarak dest ile belirlenen işlenen 1 veya 0 değerini alır.

Yapılan işlem:

```
if ( xxx ) then
        dest ←1
else
        dest ←0
endif
```

SETA	Set if above / CF = 0 ve ZF = 0	**SETNE**	Set if not equal / ZF = 0
SETAE	Set if above or equal / CF = 0	**SETNG**	Set if not greater (SETLE)
SETB	Set if below / CF =1	**SETNGE**	Set if not greater or equal (SETL)
SETBE	Set if below or equal / CF =1 veya ZF = 1	**SETNL**	Set if not less (SETGE)
SETC	Set if carry / CF =1	**SETNLE**	Set if not less or equal / SF = OF ve ZF =0
SETE	Set if equal / ZF = 1	**SETNO**	Set if not overflow / OF= 0
SETG	Set if greater / SF = OF ve ZF =0	**SETNP**	Set if no parity / PF = 0
SETGE	Set if greater or equal / SF =OF	**SETNS**	Set if no sign / SF = 0
SETL	Set if less / SF <> OF	**SETNZ**	Set if not zero / ZF = 0
SETLE	Set if less or equal / SF <> OF ve ZF = 1	**SETO**	Set if overflow / OF = 1
SETNA	Set if not above (SETBE)	**SETP**	Set if parity / PF = 1
SETNAE	Set if not above or equal (SETB)	**SETPE**	Set if parity even / PF = 1
SETNB	Set if not below (SETAE)	**SETPO**	Set if parity odd / PF = 0
SETNBE	Set if not below equal (SETA)	**SETS**	Set if sign / SF = 1
SETNC	Set if not carry / CF = 0	**SETZ**	Set if zero / ZF = 1

BT

80386/486

bit test

16p/32 Bit

BT reg, idata
BT mem, idata
BT reg, reg
BT mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	?	?	--	?	--	?	--	x

Açıklama:

Bu işlem, işlenenler yardımıyla yeri belirlenen bit'in değerinin CF bayrağına aktarılmasını sağlar. index, dest ile ifade edilen bit dizisi içindeki pozisyonu belirlemek üzere kullanılır. dest ile belirlenen işlenenin 16 bit olması durumunda index'in 0–15, 32 bit olması durumundan ise 0-31 arası değer alacaktır.

Yapılan işlem: CF←BIT (dest, index)

Örnek:

MOV AX, 6 BT *semaphore*, AX JC *semaphore_set*	*semaphore* isimli bellek alanının 6. bit'inin değeri CF'ye aktarılarak kontrol edilmektedir. İşlemin sonunda CF'nin durumuna bağlı olarak ne şekilde devam edileceğini koşullu dallanma komutu belirlemektedir

BTC

80386/486

bit test and complement

16p/32 Bit

BTC reg, idata
BTC mem, idata
BTC reg, reg
BTC mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	?	?	--	?	--	?	--	x

Açıklama:
dest'in index ile ifade edilen bitini CF'ye kopyalandıktan sonra dest üzerindeki ilgili bit'in tümleyenini alır. dest ile belirlenen işlenenin 16 bit olması durumunda index'in 0–15, 32 bit olması durumundan ise 0-31 arası değer alacaktır.

Yapılan işlem: CF←BIT (dest, index)
BIT (dest, index)←NOT (BIT (dest, index))

Örnek:

BTC *semaphore*, 2 JC *bitset*	*semaphore* isimli değişkenin 2.bit'i CF'ye aktarılıp o bitin değeri tümleyeni ile değiştirilmektedir. *semaphore* isimli değişkenin ilk değerinin 1100110011110**1**01B olduğunu düşünürsek 2. bit değeri (en az anlamlı bit en sağdan, sıfırdan başlamak kaydı ile sayıldığı için) 1 olup, bu değer CF'ye aktarılmış ve tümleyeni alındıktan sonra değişkenin yeni değeri 1100110011110**0**01B olmuştur. İşlemin sonunda CF'nin durumuna bağlı olarak ne şekilde devam edileceğini koşullu dallanma komutu belirlemektedir.

BTR

80386/486

bit test and reset

16p/32 Bit

BTR reg, idata
BTR mem, idata
BTR reg, reg
BTR mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	?	?	--	?	--	?	--	x

Açıklama:
dest'in index ile ifade edilen bit'ini CF'ye kopyaladıktan sonra dest üzerindeki ilgili bit'in değerini sıfırlar. dest ile belirlenen işlenenin 16 bit olması durumunda index'in 0–15, 32 bit olması durumundan ise 0-31 arası değer alacaktır.

Yapılan işlem: CF←BIT (dest, index)
BIT (dest, index)←0

Örnek:

BTR AX, 2 JC *bit_prev_stat*	AX yazmacının 2.bit'i CF'ye aktarılır ve o bit sıfırlanır. Örneğimizde AX yazmacında olan değerin 1100110011110**1**01B olduğunu düşünürsek 2. bit değeri (en az anlamlı bit en sağdan sıfır diye başlamak kaydı ile sayıldığı için) 1 olup CF'nin yeni değerini oluştururken kendisi de sıfır olacaktır. Dolayısıyla AX yazmacının yeni değeri 1100110011110**0**01B olur. İşlemin sonunda CF'nin durumuna bağlı olarak ne şekilde devam edileceğini koşullu dallanma komutu belirlemektedir.

BTS 80386/486

bit test and set 16p/32 Bit

BTS reg, idata
BTS mem, idata
BTS reg, reg
BTS mem, reg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	?	?	--	?	--	?	--	x

Açıklama:
dest'in index ile ifade edilen bit'ini CF'ye kopyaladıktan sonra dest üzerindeki ilgili bit'in değerini bir yapar. dest ile belirlenen işlenenin 16 bit olması durumunda index'in 0–15, 32 bit olması durumundan ise 0-31 arası değer alacaktır.

Yapılan işlem: CF←BIT (dest, index)
BIT (dest, index)←1

Örnek:

BTS AX, 2 JC *bit_prev_stat*	AX yazmacının 2.bit'i CF'ye aktarılır ve o bit bir yapılır.. Örneğimizde AX yazmacında olan değerin 1100110011110**0**01B olduğunu düşünürsek 2. bit değeri (en az anlamlı bit en sağdan sıfır diye başlamak kaydı ile sayıldığı için) 0 olup CF'nin yeni değerini oluştururken kendisi de bir olacaktır. Dolayısıyla AX yazmacının yeni değeri 1100110011110**1**01B olur. İşlemin sonunda CF'nin durumuna bağlı olarak ne şekilde devam edileceğini koşullu dallanma komutu belirlemektedir

5.7 Öteleme ve Döndürme Komutları

İşlenenler üzerinde sağa veya sola doğru döndürme (rotate)/öteleme (shift) işlemleri için kullanılan komutlar bu grup altında toplanmıştır. Bu gruba dâhil komutların bazıları CF bayrağını kullanmakta ve işlemlerini bu bayrak değerini de kullanarak gerçekleştirmekte hatta sonuçlarını da bu bayrak üzerinde oluşturmaktadır. 16 bitlik işlemcilerde, işlemin tekrar sayısını belirlemek için kullanılan ikinci işlenen değeri (idata) 1 olarak kullanılmalıdır[93]. Ancak işlemin birden fazla defa tekrarlanmasını gerektiren durumlar için CL yazmacından yararlanılır. CL yazmacındaki değerin ilk işlenen ile uyumlu olarak belirlenmesi gereklidir. İlk işlenen byte tanımlı ise CL yazmacında 1-7 arasında bir değer olması anlamlıdır. Zira 8 bit uzunluğundaki bir işlenenin 8 defa ötelenmesi (işlemin sağa veya sola yapılmasının farkı yoktur) sıfır olmasına, CF bayrağını kullanmadan döndürülmesi (işlemin sağa veya sola yapılması fark etmez) tekrar kendisinin elde edilmesine neden olacaktır. Buna göre ilk işlenen word tanımlı ise 1-15, double word tanımlı ise 1-31 arasında tekrar sayısı verilmesi gerekir. Sola doğru yapılan öteleme işlemleri iki ile çarpma, sağa doğru yapılan öteleme işlemleri ise ikiye bölme ile eşdeğer sonuç üretmektedir.

[93] 32 bit işlemcilerde (80386/486) ikinci işlenen idata da olabilir. Ancak işlem sayısını belirleyen değerin 31 den büyük olmaması için 1FH değeri ile maskelenir. Maskeleme işlemi AND komutu ile yapılır.

SHL

8086/286/386/486

shift left logical

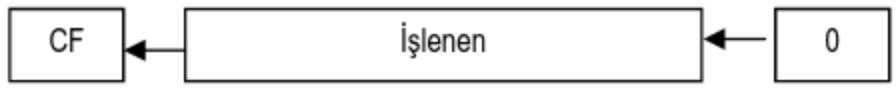
8/16/32 Bit

SHL reg, idata
SHL mem, idata
SHL reg, CL
SHL mem, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	?	--	x	--	x

Açıklama:

İşlenenin en az anlamlı bit'ine sıfır konularak sayının bir basamak sola doğru kaydırılmasını sağlar. İkili aritmetik açısından incelendiğinde sola bir bit kaydırma işlemi sayının iki ile çarpılması ile aynı anlama gelmektedir. Diğer yandan sayının en anlamlı bit'i ise CF bayrağına düşecektir.



Örnek:

SHL AH, 1	AH←10100011B ise sonuç AH←010000110B ve CF←1 olur
SHL *Data*, CL	*Data* olarak tanımlı değişken 1010101000001111B ise ve CL←4 alınırsa sonuç *Data*←1010000011110000B ve CF←0 olur.
SHL EAX, 7	80386/486 işlemcilerde EAX, 7 defa sola kaydırılır.

SHLD

80386/486

shift left double

16p/32 Bit

SHLD reg, reg, idata
SHLD mem, reg, idata
SHLD reg, reg, CL
SHLD mem, reg, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	x	x	--	?	--	x	--	x

Açıklama:

Üç işleneni olan bu komut ilk iki işlenen bir parça olarak değerlendirilir (bu nedenle işlem double olarak tanımlanmıştır) ve verilen sayı kadar sola öteleme işlemi gerçekleştirilir.

Yapılan işlem: SHLD dest, src, count
temp←min (*count*, 31)
value←concatenate (dest, src)
value←*value* << *temp*
dest←*value*

Örnek:

LEA ESI, *Mydata*
MOV EAX, [ESI]
MOV EDX, [ESI+4]
SHLD EDX, EAX, 8

	7 … 0	
0007H	78H	
0006H	00H	
0005H	21H	
0004H	4DH	
0003H	45H	
0002H	53H	
0001H	51H	
0000H	41H	← *Mydata*

Düşük anlamlı 32 bit EAX'e alınınca EAX←45535141H olur
Yüksek anlamlı 32 bit EDX'e alınınca EDX←7800214DH olur.

EDX:EAX ikilisindeki 64 bit'lik değer 8 bit sola kaydırılırsa;
EAX←53514100H
EDX←00214D45H olacaktır.

SAL

8086/286/386/486

shift arithmetic left

8/16/32 Bit

SAL reg, idata
SAL mem, idata
SAL reg, CL
SAL mem, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	?	--	x	--	x

Açıklama:
SAL komutu **SHL** ile aynı özellikte olan bir komuttur. Her ne kadar isminde aritmetik kelimesi geçse de sola kaydırma işlemlerinde işaret bit'inin bir anlamı kalmamaktadır.

SHR

8086/286/386/486

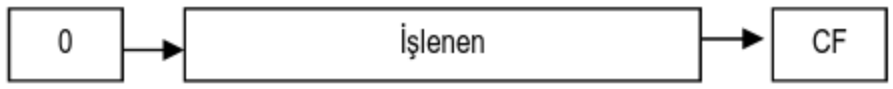
shift right logical

8/16/32 Bit

SHR reg, idata
SHR mem, idata
SHR reg, CL
SHR mem, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	?	--	x	--	x

Açıklama:
İşlenenin bir bit sağa kaydırılmasını sağlayan komuttur. Bir defa sağa ötelemek ile işlenen iki ile bölünmüş olur. Sağdan en anlamlı bit alanına sıfırı sayısı yerleştirilirken an az anlamlı bit ise CF bayrağına düşecektir.



Örnek:

SHR AX, 1	AX yazmacındaki değer ikiye bölünmüş olacaktır. AX←000AH ise işlem sonucunda AX←0005H ve CF←0 olacaktır.

SHRD

80386/486

shift right double

16p/32 Bit

SHLD reg, reg, idata
SHLD mem, reg, idata
SHLD reg, reg, CL
SHLD mem, reg, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	x	x	--	?	--	x	--	x

Açıklama:
Üç işleneni olan bu komut ilk iki işlenen bir parça olarak değerlendirilir (bu nedenle işlem double olarak tanımlanmıştır) ve verilen sayı kadar sağa öteleme işlemi gerçekleştirilir.

Yapılan işlem: SHLD dest, src, *count*
temp←min (*count*,31)
value←concatenate (dest, src)
value←*value* >> *temp*
dest←*value*

Örnek:

MOV EAX, [ESI]
MOV EDX, [ESI+4]
SHRD EDX, EAX, 8
	Düşük anlamlı 32 bit EAX'e yüksek anlamlı 32 bit EDX'e alınır, EDX:EAX ikilisinin oluşturduğu 64 bit'lik değer 8 bit sağa kaydırılır.
	EDX←7800214DH ve EAX←45535141H ise işlem sonucunda
	EDX←00780021H ve EAX←4D455351H olacaktır.

SAR

8086/286/386/486

shift arithmetic right

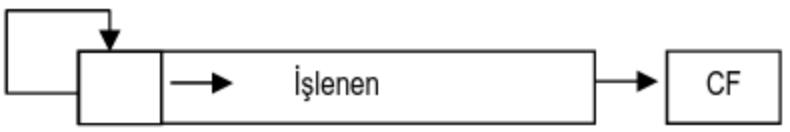
8/16/32 Bit

SAR reg, idata
SAR mem, idata
SAR reg, CL
SAR mem, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	x	x	--	?	--	x	--	x

Açıklama:

İşaretli sayılarda işareti belirleyen en anlamlı bit'in değerini korumak amacıyla kullanılan bir öteleme komutudur. İşlem sırasında işaret bit'i kendi yerinde kalırken işlenenin diğer bit'leri de sağa doğru kaydırılır. Sağa doğru yapılan her kaydırma işlemi ikiye bölme manasına gelmektedir. Ancak bu işlem sonucunda kesirli sayı[94] elde edilemeyeceği için elde edilen sonuç negatif yöne doğru yuvarlatılmış değer olacaktır. Yani –3 sayısı sağa doğru bir defa ötelendiğinde elde edilecek sonuç -1,5 yerine –2 olacaktır.



Örnek:

SAR AH, 1	AH←10000000B yani -128 sayısı olsun. İşlem sonucunda elde edilen değer ise AH←11000000B ya da diğer şekli ile -64 sayısıdır.
SAR AL, 1	AL←11111101B diğer deyişle –3 iken sağa öteleme sonucunda AL←11111110B diğer deyişle –2 değerini alacaktır.

RCL

8086/286/386/486

rotate through carry left

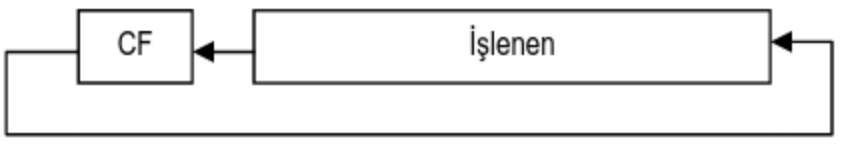
8/16/32 Bit

RCL reg, idata
RCL mem, idata
RCL reg, CL
RCL mem, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	--	--	--	--	--	--	--	x

Açıklama:

Verilen işleneni CF bayrağı üzerinden döndürmek için kullanılan bir komuttur. Bu komutun bir defa çalıştırılması sonucunda CF bayrağında bulunan bit işlenenin en az anlamlı biti haline gelecektir. Bunun gibi CF bayrağı üzerinden yapılan işlemlerde CF bayrağının set (1) veya clear (0) olması büyük önem taşımaktadır. 32 bit işlemcilerde *idata* ile işlem sayısı[95] belirlenebilmektedir.



Örnek:

RCL AH, 1	AH←10101010B ve CF←1 ise sonuç AH←01010101B ve CF←1 olur.

---

[94] Kesirli sayılar ile işlem yaparken bu iş için tasarlanmış komut kümesinin kullanılması gerekmektedir. 80486 öncesi işlemciler kendilerine uygun matematik işlemci olması durumunda kesirli işlem yapmayı sağlayan komut kümesini kullanabilir. 80486 ve üstü işlemcilerde matematik işlemci zaten mevcuttur. Kesirli sayılar ile işlem yapan komutları F harfi (Float) ile başlar.

[95] İşlem sayısının 1-31 arasında olması gereklidir. Bu aralıktan büyük değer verilmemesi için sistem tekrar sayısını 1FH değeri ile maskelemektedir.

RCR

8086/286/386/486

rotate right through carry

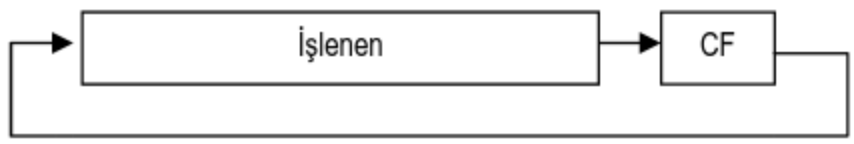
8/16/32 Bit

RCR reg, idata
RCR mem, idata
RCR reg, CL
RCR mem, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	--	--	--	--	--	--	--	x

Açıklama:

RCL ile benzer özellikte bir işlemdir. Ancak bu sefer işlem yönü sağa doğrudur.



Örnek:

STC MOV AH, 10101010B RCR AH, 1	STC komutu işletildiğinde CF←1 olacaktır. AH←10101010B Sonuçta AH←11010101B ve CF←0 olur.
MOV CL, 2 RCR *Sayi*, CL	Word tipindeki *Sayi* isimli değişken 1100110011001100B değerine sahip, CF←0 iken CL yazmacında belirlendiği gibi 2 defa RCR işlemi tekrarlanırsa *Sayi* isimli değişken 0011001100110011B değerini alacak, CF←0 olacaktır.

ROL

8086/286/386/486

rotate left

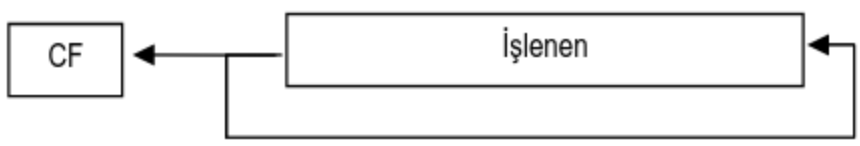
8/16/32 Bit

ROL reg, idata
ROL mem, idata
ROL reg, CL
ROL mem, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	--	--	--	--	--	--	--	x

Açıklama:

Sola doğru yapılan bir döndürme işlemi olmasına rağmen CF bayrağı doğrudan işlemin sonucunu etkilememektedir. Ancak döndürme yapıldıktan sonra işlenenin en anlamlı bit'i CF bayrağına gelmektedir.



Örnek:

ROL AH, 1	AH←11110000B ve CF←0 iken işlem sonucu AH←11100001B ve CF←1 olur.

MOV AX, 00110011001100B
MOV CL, 4
ROL AX, CL
	ROL işlemi CL yazmacında belirlendiği gibi 4 defa tekrarlanırsa
	AX← 1100110011000011B ve CF←1 olacaktır.

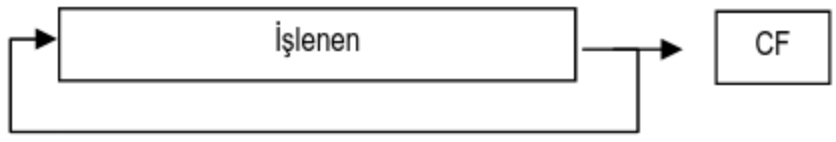
ROR 8086/286/386/486

rotate right 8/16/32 Bit

ROR reg, idata
ROR mem, idata
ROR reg, CL
ROR mem, CL

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	--	--	--	--	--	--	--	--	--	--	x

Açıklama:
İşlem sonucunda işlenen sağa doğru ilerlemektedir. CF bayrağı işlemi doğrudan etkilemez ancak işlem sonunda en az anlamlı bit CF bayrağında olacaktır.



Örnek:

ROR AH, 1	AH←11110000B ve CF←1 iken işlem sonucu AH←01111000B ve CF←0 olur.

5.8 Dizgi (String) Komutları

Dizgi (string) komutları bellek alanı üzerinde byte, word ve double word olmak üzere, farklı büyüklükler ile işlem yapmayı mümkün kılar. Bu komutlardan;

- **MOVS**, bir bellek bölgesinin başka bir bellek bölgesine aktarılması,
- **CMPS**, farklı iki bellek bölgesinin içeriklerinin karşılaştırılması,
- **SCAS**, bellek bölgesinin içeriğinin EAX/AX/AL yazmacı ile karşılaştırılması,
- **LODS**, bellek bölgesindeki değerlerin EAX/AX/AL yazmacına yüklenmesi,
- **STOS**, bellek bölgesinin EAX/AX/AL yazmacının değeri ile doldurulması

gibi genel olarak tekrarlamalı işlemleri gerçekleştirmek için tasarlanmıştır. İşlemlerin tekrar sayısı CX yazmacı ile belirlenirken dizi komutlarının önünde kullanılan **REP** öneki (prefix) çevrimin yapılmasını mümkün kılmaktadır.

Bu gruba dâhil komutlarda DF bayrağının değeri yapılan işlemin yönünü (direction) belirlediği için oldukça önemlidir. DF←1 olması (DN-Decrement-Down) indis yazmaçlarının azalacağını, DF←0 olması (UP-Increment-Up) ise indis yazmaçlarının artacağını belirtmek için kullanılmaktadır. Bu artırma/azaltma değerleri ise kullanılan dizi komutunun sonunda yer alan B(yte), W(ord), D(ouble word) harfleri ile tarif edilen işlem uzunluğu ile belirlenmiş olup sırası ile 1, 2 ve 4 olabilir.

MOVSB

8086/286/386/486

move string byte

8 Bit

MOVSB

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	**0/1**	--	--	--	--	--	--	--	--	--	--

Açıklama:
Byte bazında aktarma işlemi yapan bir komuttur. Komutun görevi DS:SI ikilisinin gösterdiği adresteki byte'ı ES:DI ikilisinin gösterdiği adrese aktarmaktır. **MOV** işlemlerinde olduğu gibi **MOVS** komutları da bayrakları etkilememektedir.

Yapılan işlem: [ES:DI]←[DS:SI]
DI←DI - 1
SI←SI - 1 (DF←1 olması durumu için, aksi halde SI ve DI artırılacaktır)

Örnek:

```
LEA SI, eskiyer
LEA DI, yeniyer
MOV CX, 10
CLD
REP MOVSB
```

	7 … 0	
0209H	66H	
	…	
	….	
0203H	05H	
0202H	~~27H~~/53H	
0201H	~~21H~~/51H	
0200H	~~14H~~/41H	← *yeniyer*
	…	
0009H	77H	
	…	
	…	
0003H	45H	
0002H	53H	
0001H	51H	
0000H	41H	← *eskiyer*

nın sonucu gösterilmiştir.

eskiyer isimli değişkenin bulunduğu adresten başlamak üzere 10 byte değer *yeniyer* (DS ve ES yazmaçlarının aynı olduğunu kabul ederek) isimli değişkenin bulunduğu adrese aktarılmak istenmektedir. DF←0 yapılarak adreslerin artarak ilerleyeceği belirtilmiştir. REP ön eki ile işlem CX de belirlenen sayı kadar tekrar edilecektir. REP her seferinde CX'i bir azaltarak CX=0 oluncaya kadar işlemin tekrarlar. Her turda SI ve DI yazmaçlarının değerleri (işlem byte türünde olduğu için) 1 arttırılacaktır. Buna göre ilk adında [0200] adresine [0000] adresindeki byte yazılacaktır. *yeniyer* isimli değişkenin gösterdiği adreste yer alan eski değerler yok olacaktır. Örnekte işlemin ilk 3 adımı-

MOVSW

8086/286/386/486

move string word

16 Bit

MOVSW

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	**0/1**	--	--	--	--	--	--	--	--	--	--

Açıklama:
Word bazında aktarma işlemi yapan bir komuttur. Komutun görevi DS:SI ikilisinin gösterdiği adresteki word'ü ES:DI ikilisinin gösterdiği adrese aktarmaktır. Word olması dolayısıyla DF bayrağının durumuna bağlı olarak artım ve azaltmalar ikişer olarak yapılmaktadır.

Yapılan işlem: [ES:DI]←[DS:SI]
DI←DI - 2
SI←SI - 2 (DF←1 olması durumu için, aksi halde SI ve DI artırılacaktır)

MOVSD

80386/486
move string doubleword
32 Bit

MOVSD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	**0/1**	--	--	--	--	--	--	--	--	--	--

Açıklama:
Double word bazında aktarma işlemi yapan bir komuttur. Komutun görevi DS:SI ikilisinin gösterdiği adresteki double word'ü ES:DI ikilisinin gösterdiği adrese aktarmaktır. Double word olması dolayısıyla DF bayrağının durumuna bağlı olarak artım ve azaltmalar dörder olarak yapılmaktadır.

Yapılan işlem: [ES:DI]←[DS:SI]
EDI←EDI - 4
ESI←ESI - 4 (DF←1 olması durumu için, aksi halde SI ve DI artırılacaktır)

CMPSB

8086/286/386/486
compare string byte
8 Bit

CMPSB

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:
Belleğin farklı adreslerindeki byte büyüklüğündeki değerlerin karşılaştırılmasında kullanılmaktadır. **CMP** komutunda olduğu gibi iki işlenenin birbirinden çıkarılması sonucu bayraklar etkilenmektedir.

Yapılan işlem: [DS:SI]-[ES:DI] işlemden bayraklar etkilenir.
DI←DI -1
SI←SI - 1 (DF←1 olması durumu için, aksi halde SI ve DI artırılacaktır)

Örnek:

LEA SI, *dizi1* LEA DI, *dizi2* MOV CX, 10 CLD REPE CMPSB JNE *uymaz*	*dizi1* ve *dizi2* (DS ve ES yazmaçlarının aynı olduğunu kabul ederek) isimli bellek alanlarında bulunan 10 byte'lık verinin aynı olup olmadığı kontrol edilmek istenmektedir. CLD komutu ile DF← 0 yapıldığı için SI ve DI değerleri her seferinde bir arttırılacaktır. REPE ön eki işlemin eşitlik söz konusu olduğu sürece devamını sağlar ve her turda CX sıfır oluncaya kadar SI ve DI değerleri otomatik olarak arttırır. Herhangi bir noktada eşitsizlik söz konusu olursa REPE koşulu bozulur ve işlem JNE komutu ile devam eder.

CMPSW

8086/286/386/486
compare string word
16 Bit

CMPSW

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:
Belleğin farklı adreslerindeki word büyüklüğündeki değerlerin karşılaştırılmasında kullanılmaktadır. **CMP** komutunda olduğu gibi iki işlenenin birbirinden çıkarılması sonucu bayraklar etkilenmektedir.

Yapılan işlem: [DS:SI]-[ES:DI] işlemden bayraklar etkilenir.
DI←DI - 2
SI←SI - 2 (DF←1 olması durumu için, aksi halde SI ve DI artırılacaktır)

CMPSD

80386/486
compare string doubleword
32 Bit

CMPSD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:
Belleğin farklı adreslerindeki double word büyüklüğündeki değerlerin karşılaştırılmasında kullanılmaktadır. **CMP** komutunda olduğu gibi iki işlenenin birbirinden çıkarılması sonucu bayraklar etkilenmektedir.
Yapılan işlem: [DS:SI]-[ES:DI] işlemden bayraklar etkilenir.
EDI←EDI - 4
ESI←ESI - 4 (DF←1 olması durumu için, aksi halde SI ve DI artırılacaktır)

SCASB

8086/286/386/486
scan string byte
8 Bit

SCASB

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:
SCASB komutu AL yazmacı ile ES:DI ikilisinin gösterdiği adreste bulunan byte'ı karşılaştırır. Bu karşılaştırmadan sadece bayraklar etkilenir. Bu komutta diğer benzerleri gibi **REP** ön ekleri ile bağlantılı olarak kullanılmaktadır.
Yapılan işlem: AL - [ES:DI] Bayraklar etkilenir.
DI←DI - 1 (DF←1 olması durumu için, aksi halde DI artırılacaktır)

Örnek:

```
LEA DI, mesaj
MOV CX, 000CH
CLD
MOV AL, '*'
REPNE SCASB
JE buldu
```

	7 … 0	
020CH	**2AH**	
020BH	20H	
020AH	20H	
0209H	**2AH**	
0208H	79H	
0207H	6CH	
0206H	62H	
0205H	6DH	
0204H	65H	
0203H	73H	
0202H	73H	
0201H	41H	← *mesaj*

mesaj (DS ve ES yazmaçlarının aynı olduğunu kabul ederek) isimli bellek alanının içinde '*' (ASCII kodu 2AH'dir) karakteri aranmaktadır. Bunun için, DI yazmacına *mesaj* değişkeninin başlangıç adresi, AL'ye aranacak karakter, CX'e tekrar sayısı (12 veya 000CH) verilmiş, CLD komutu ile DF←0 yapılarak DI yazmacının artan yönde değişeceği belirtilmiştir.
Buna göre işlem DI←0209H değerini alana kadar REPNE çevrimi içinde dönecek, bu aşamada [0209] adresinin içeriği AL ile aynı ve ZF←1 olacağı için REPNE koşulu sağlanamayıp çevrimden çıkılacaktır. Çevrimin normal yolla mı bittiği yoksa ZF←1 olduğu için mi bittiğini anlamak için koşullu dallanma komutu ile ZF bayrağının durumu kontrol edilerek arananın bulunup bulunamadığına karar verilmektedir. İşlem ilk karşılaşılan eşitlikte sonlanacaktır.

SCASW

8086/286/386/486
scan string word
16 Bit

SCASW

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:
SCASW komutu AX yazmacı ile ES:DI ikilisinin gösterdiği adreste bulunan word'ü karşılaştırır. Bu karşılaştırmadan sadece bayraklar etkilenir.
Yapılan işlem: AX - [ES:DI] Bayraklar etkilenir.
DI←DI - 2 (DF←1 olması durumu için, aksi halde DI artırılacaktır.)

SCASD

80386/486
scan string doubleword
32 Bit

SCASD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:
SCASD komutu EAX yazmacı ile ES:DI ikilisinin gösterdiği adreste bulunan double word'ü karşılaştırır. Bu karşılaştırmadan sadece bayraklar etkilenir.
Yapılan işlem: AX - [ES:EDI] Bayraklar etkilenir.
DI←DI - 4 (DF←1 olması durumu için, aksi halde DI artırılacaktır)

LODSB

8086/286/386/486
load string byte
8 Bit

LODSB

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	**0/1**	--	--	--	--	--	--	--	--	--	--

Açıklama:
DS:SI ikilisin belirttiği adresten bir byte'ı AL yazmacına yerleştirir. SI yazmacının değeri ise DF bayrağının durumuna bağlı olarak değiştirilir. **LOADSB** komutu bayraklar üzerinde bir değişikliğe neden olmaz.
Yapılan işlem: AL←[DS:SI]
SI←SI - 1 (DF←1 olması durumu için, aksi halde SI artırılacaktır)

Örnek:

```
        LEA SI, input
        LEA DI, output
        LEA BX, ascii2ebcdic
        CLD
L1:     LODSB
        OR AL, AL
        JZ L2
        XLAT
        STOSB
        JMP L1
L2:     .......
```

Örneğimizde, *input* isimli bellek alanındaki ASCII karakterlerin EBCDIC karşılıkları *output* isimli bellek alanına yazılmaktadır. İşlem *input* alanında 00H görünceye kadar devam edecektir. Bu amaçla SI, DI ve BX yazmaçları sırası ile *input*, *output* ve *ascii2ebcdic* isimli bellek alanlarının başını gösterecek biçimde ayarlanmıştır. DF←0 yapıldığı için her işlemden sonra SI ve DI yazmacı arttırılacaktır. *L1* etiketi ile başlayan çevrimde LODSB ile AL yazmacına alınan değer OR işleminden geçirilerek (bu işlem yazmaç üzerinde bir değişikliğe neden olmaz sadece) bayraklar düzenlenir. ZF←1 olmuş ise işlem *L2* etiketine giderek akış sonlandırmaktadır. Aksi halde XLAT komutu ile AL←[BX+AL] işlemi yapılmış ve daha önce ASCII değerini barındıran AL yazmacına bunun karşılığı olan EBCDIC değeri *ascii2ebcdic* tablosundan alınmıştır. STOSB komutu ise [DI]←AL işlemini yaparak hedef alanda EBCDIC kodlardan oluşan diziyi oluşturmaktadır.

LODSW

8086/286/386/486

load string word — 16 Bit

LODSW

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	**0/1**	--	--	--	--	--	--	--	--	--	--

Açıklama:
DS:SI ikilisin belirttiği adresten bir word'ü AX yazmacına yerleştirir. SI yazmacının değeri ise DF bayrağının durumuna bağlı olarak değiştirilir. **LOADSW** komutu bayraklar üzerinde bir değişikliğe neden olmaz.

Yapılan işlem: AX←[DS:SI]
SI←SI - 2 (DF←1 olması durumu için, aksi halde SI iki artırılacaktır)

LODSD

80386/486

load string double word — 32 Bit

LODSD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	**0/1**	--	--	--	--	--	--	--	--	--	--

Açıklama:
DS:SI ikilisin belirttiği adresten bir double word'ü EAX yazmacına yerleştirir. ESI yazmacının değeri ise DF bayrağının durumuna bağlı olarak değiştirilir. **LOADSD** komutu bayraklar üzerinde bir değişikliğe neden olmaz.

Yapılan işlem: EAX←[DS:ESI]
ESI←ESI - 4 (DF←1 olması durumu için, aksi halde ESI dört artırılacaktır)

STOSB

8086/286/386/486

store string byte — 8 Bit

STOSB

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:
LODSB işleminin yaptığının tam tersini yapan bir işlemdir. AL yazmacındaki değeri ES:DI ikilisinin gösterdiği bellek alanına yerleştirir. DI yazmacının değeri DF bayrağının durumuna bağlı olarak değiştirilir.

Yapılan işlem: [ES:DI]←AL
DI←DI - 1 (DF←1 olması durumu için, aksi halde DI artırılacaktır)

Örnek:

MOV DI, 0	Başlangıç adresi olarak 0 seçiliyor.
MOV CX, 100	İşlem tekrar sayısı CX yazmacındaki değer ile belirleniyor.
XOR AL, AL	AL yazmacı sıfırlanıyor.
CLD	DF←0 yapılarak DI yazmacının artarak ilerleyeceği belirleniyor.
REP STOSB	Böylece 0 adresinden itibaren 100 byte'lık bellek alanı sıfır ile dolduruluyor.

STOSW

8086/286/386/486

store string word

16 Bit

STOSW

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:

LODSW işleminin yaptığının tam tersini yapan bir işlemdir. AX yazmacındaki değeri ES:DI ikilisinin gösterdiği bellek alanına yerleştirir. DI yazmacının değeri DF bayrağının durumuna bağlı olarak değiştirilir.

Yapılan işlem: [ES:DI]←AX

DI←DI - 2 (DF←1 olması durumu için, aksi halde DI artırılacaktır)

STOSD

80386/486

store string doubleword

32 Bit

STOSD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	**0/1**	--	--	x	x	--	x	--	x	--	x

Açıklama:

LODSD işleminin yaptığının tam tersini yapan bir işlemdir. EAX yazmacındaki değeri ES:EDI ikilisinin gösterdiği bellek alanına yerleştirir. DI yazmacının değeri DF bayrağının durumuna bağlı olarak değiştirilir.

Yapılan işlem: [ES:EDI]←EAX

DI←DI - 4 (DF←1 olması durumu için, aksi halde DI artırılacaktır)

CBW

8086/286/386/486

convert byte to word

8 Bit

CBW

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:

İşleneni olmayan bu komut AL yazmacında bulunan verinin AX yazmacına yerleştirilmesini sağlar. Bu işlem yapılırken işaret bit'i yüksek anlamlı byte boyunca tekrarlanır.

Örnek:

CBW	AL←10001111B ise sonuç AX←1111111110001111B olur. AL←01111000B ise sonuç AX←0000000001111000B olur.

CWD

8086/286/386/486

convert word to doubleword

16 Bit

CWD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
İşleneni olmayan bu komut AX yazmacında bulunan bilginin DX ve AX yazmaç ikilisine sağlar. Bu işlem yapılırken işaret bit'i yüksek anlamlı word boyunca tekrarlanır. Yani DX yazmacı işaret bit'inin sahip olduğu değer ile doldurulur.

Örnek:

CWD	AX←1000000011111111B ise sonuç DX←1111111111111111B ve AX←1000000011111111B olacaktır.

CWDE

80386/486

convert word to doubleword extended

16 Bit

CWDE

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bu komut ile AX yazmacındaki değerin işaret bit'i EAX yazmacının yüksek anlamlı word'ü boyunca tekrarlanır.

CDQ

80386/486

convert doubleword to quadword

32 Bit

CDQ

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
İşaret biti saklı kalmak kaydıyla double word bir sayıyı quad word sınırına getirmektedir. Bu komut yanına işlenen almaz, giriş verisi olarak EAX yazmacını kabul eder ve sonucu EDX:EAX ikilisinde oluşturur.

BSWAP

80486

byte swap

32 Bit

BSWAP reg32

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Intel işlemciler Little Indian (düşük anlamlı byte'ın düşük bellek adresinde olması) notasyonunu desteklemektedir. 32 bit Little Indian gösterimini Big Indian gösterimine dönüştürmek için 80486 işlemcisinden itibaren kullanılmaya başlamış bir komuttur.

Örnek:

BSWAP EAX	EAX←1234ABCDH ise işlem sonucunda EAX←0DCBA4321H olacaktır.

5.9 Ön Ekler

Ön ekler özelikle **CMPS**, **LODS**, **MOVS**, **SCAS**, **STOS** türü dizgi komutların önünde, işlemlerin istenen sayıda veya gerek koşul sağlanıncaya kadar tekrarlanmasını sağlamak üzere kullanılırlar. İşlemin tekrar sayısı CX yazmacındaki değer yardımıyla belirlenir. Her işlemden sonra CX yazmacı otomatik olarak bir azaltılacaktır. Özellikle tek komutun tekrarlandığı türde işlemlerde LOOP komutu kullanarak işlemleri yapmak yerine ön eklerden yararlanmak bir alışkanlıktır.

REP 8086/286/386/486

Repeat string prefix

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
REP	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
CX <> 0 olduğu sürece **REP** komutunu takip eden işlem tekrarlanır. Diğer değişle CX=0 olunca durur.

Örnek:

LEA DI, *dizi* CLD XOR AL, AL MOV CX, 1024 REP STOSB	*dizi* isimli bellek alanının içeriğinin sıfırlanması için bellek adresi DI yazmacında tutulmaktadır. İşlemin artan adrese doğru yapılması için DF←0 yapılır. İşlem sayısı CX yazmacına aktarılır. STOSB işlemi yapıldıktan sonra REP ön eki gereği CX azaltılır ve sıfırdan farklı olduğu sürece STOSB komutu tekrar işletilir.
LEA SI, *dizi1* LEA DI, *dizi2* CLD MOV CX, 8 REP MOVSB	*dizi1* ile belirlenen bellek alanından 8 byte veri *dizi2* ile belirlenen alana aktarılmaktadır. Bu işlem MOVSB yerine MOVSW ile yapılacak olursa CX yazmacında 8 değeri kullanmak yerine 4 değeri kullanılmalıdır.

REPE / REPZ 8086/286/386/486

repeat equal / repeat zero

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
REPE REPZ	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
ZF bayrağının değeri 1 olduğu sürece işleme devam edilecektir. Diğer deyişle CX = 0 veya ZF bayrağının değeri 0 olduğunda işlem sona erer.

Örnek:

XOR BL, BL CLD MOV CX, 8 LEA SI, *dizi1* LEA DI, dizi2 REPE CMPSB JNE *Son* INC BL *Son*: ...	*dizi1* ve *dizi2* isimli bellek alanlarının içerikleri karşılaştırılmaktadır. CX=0 veya ZF bayrağının değeri 0 ise işlem biter İşlemin hangi koşula bağlı olarak bittiği koşullu dallanma komutu yardımıyla belirlenir. İşlem bir eşitlik sonucu sona ermiş ise BL yazmacının değeri artırılmaktadır.

REPNE / REPNZ 8086/286/386/486

repeat not equal / repeat not zero

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
REPNE REPNZ	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --

Açıklama:
ZF bayrağının değeri 0 olduğu sürece işleme devam edilecektir. Başka bir deyişle ZF bayrağının değeri 1 veya CX=0 olduğunda işlem sona erer.

LOCK 8086/286/386/486

assert hardware lock / signal prefix

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
LOCK	--	---	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bu komut çok işlemcili (multiprocessor) sistemler tarafından desteklenmekte olup, istenilen bellek alanına aynı anda sadece bir komut tarafından erişilmesini sağlamak üzere kullanılan bir ön ektir. Sadece aşağıda tablosu verilen komutların önünde kullanılabilir.

Komut(lar)	Kullandığı işlenen(ler)
BT / BTR / BTS / BTC	mem, opr
XCHG	mem, reg
	reg, mem
ADD / ADC / AND / OR / SUB / SBB	mem, opr
XOR / DEC / NEG / IN / NOT	mem

XCHG komutu doğası gereği çalışırken **LOCK** ön ekini kullanır. Bu nedenle program içerisinde **XCHG** komutunun önüne özellikle **LOCK** ön ekinin kullanılması gerekmemektedir.

5.10 Yığın Komutları

Yığın, çağırılan yordamların dönüş adreslerini, yazmaçların değerlerini ve yordamlar arasında aktarılan parametreleri saklamak amacıyla kullanılmaktadır. 32 bit'lik mimari kullanılmadığı sürece yığın yapısı üzerinde çalışan komutlar her seferinde yığının üstünden bir word alır veya koyarlar. Oysa 80386'dan itibaren 32 bit'lik işlemcilerin devreye girmesi ile yığın işlemleri de 32 bit olarak yapılabilmektedir.

8086 işlemcisinde SP yazmacı yığın üzerinde bulunan dolu gözü göstermektedir. Bu konu başlığında açıklanan komutlarda 8086 işlemcisinin kullandığı yapıya bağlı olarak açıklamalar yapılmıştır. O zaman 8086 işlemcisinde gerçekleştirilen **POP/PUSH** komutlarının nasıl çalıştığını ara adımlarıyla inceleyelim;

POP *src*	*src*←SS:[SP] SP←SP + 2	PUSH *src*	SP←SP - 2 SS:[SP]←*src*

Oysa **80286 işlemcisinden itibaren SP yazmacı yığın üzerinde bulunan boş gözü göstermektedir**. Buna göre **POP/PUSH** komutlarının 80x86 işlemcilerde yerine getirdiği ara adımlar şu şekilde gerçekleşir;

POP *src*	SP←SP + 2 *src*←SS:[SP]	PUSH *src*	SS:[SP]←*src* SP←SP – 2

Bu ara işlemler **POP/PUSH** komutlarının kullanımı ile otomatik olarak gerçekleştirildiğinden dolayı kullanıcının müdahalesi gerekmez. Ancak yığının yordamlar arasında parametre aktarmak için kullanıldığı durumlarda kodun hangi işlemci için üretildiği, yığın üzerindeki parametrelerin yerini belirlemekte önem kazanacaktır. **POP** ve **PUSH** komutları bayraklar üzerinde değişiklik yapmazlar. Ancak bayrak değerlerini yığından çeken **POPF** komutu doğası gereği bayrakları değiştirecektir.

POP 8086/286/386/486

pop word off stack 16/32 Bit

POP regw
POP mem

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Yığının üzerinden alınan veriyi *dest* ile belirlenen işlenene aktarıp yığın işaretçisi (SP yazmacının değeri) arttırılacaktır.

Yapılan işlem:
```
dest ←SS:[ESP]
if (sizeof (dest) = 16 ) then
        ESP←ESP + 2
else
        ESP←ESP + 4
endif
```

Örnek:

POP ECX	Yığından 32 bit değer ECX yazmacına alınır. ESP←ESP + 4 (32 bit işlemciler)
POP AX	Yığından 16 bit değer AX yazmacına alınır. SP←SP + 2 olur.

POPA 80286/386/486

pop all general registers 16 Bit

POPA

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
SP yazmacı haricindeki tüm genel amaçlı yazmaçların yığından alınmasını sağlar. 80286 ve üstü işlemcilerde kullanılabilir.

Örnek:

POPA
```
POP DI
POP SI
POP BP
ADD SP, 2     ; SP yazmacının değerinin değişmesini engellemek için
POP BX
POP DX
POP CX
POP AX        ; işlemlerini gerçekleştirir.
```

POPAD

80386/486

pop all general registers – double

32 Bit

POPAD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
ESP yazmacı hariç 32 bit'lik tüm genel amaçlı yazmaçların yığından alınmasını sağlar.

Örnek:

POPAD

```
POP EDI
POP ESI
POP EBP
ADD ESP, 4          ; ESP yazmacının değerinin bozulmaması için
POP EBX
POP EDX
POP ECX
POP EAX             ; işlemlerini gerçekleştirir.
```

POPF

8086/286/386/486

pop flag from stack

16 Bit

POPF

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	x	x	x	x	x	--	x	--	x	--	x

Açıklama:
Bu komut ile bayrak (PSW) yazmacına yığından alınan 16 bit'lik değer yüklenir. SP yazmacının değeri 2 arttırılır.
Yapılan işlem: FLAG←SS:[SP]
SP←SP + 2

POPFD

80386/486

pop eflag from stack - double

32 Bit

POPF

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
x	x	x	x	x	x	--	x	x	x	--	x

Açıklama:
EFLAG yazmacına yığından alınan 32 bit'lik değer yüklenir. ESP yazmacının değeri 4 arttırılır.
Yapılan işlem: EFLAG←SS:[ESP]
ESP←ESP + 4

PUSH

8086/286386/486

push word to the stack

16/32 Bit

PUSH idata
PUSH reg
PUSH mem
PUSH sreg

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Yığının üzerine konulması istenen veriyi yerleştirir. Daha sonra yığın işaretçisi uygun şekilde azaltılır.

Yapılan işlem:
```
if (sizeof (src) = 16 ) then
        ESP←ESP - 2
else
        ESP←ESP - 4
endif
        SS:[ESP]←src
```

Örnek:

PUSH EAX	Yığına EAX yazmacındaki 32 bit değer konulur.
PUSH AX	Yığına AX yazmacındaki 16 bit değer konulur.

PUSHA

80286/386/486

push all general registers

16 Bit

PUSHA

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
PUSHA 16 bit'lik tüm yazmaç değerlerinin belli bir sıraya uygun olarak yığına saklanmasını sağlar. Ancak yazmaçlar arasında yığın üzerinde işaretçi olarak kullanılan SP yazmacının durumu farklılık göstermektedir. Gerçekleştirilen her PUSH işleminin sonucu olarak SP yazmacının değeri de 2 azalmaktadır. Bu durumda SP yazmacının yığına saklanması sırasında[96] farklı bir yöntemin kullanılması gerekmektedir. Bunun için PUSHA işlemine başlarken SP'nin sahip olduğu değer geçici bir değişkende saklanır ve SP yazmacının yığında saklanma sırası geldiğinde bu geçici değer yığına SP'nin yerine yerleştirilir. Yazmaçların yığına yerleştirilme sırası şöyledir;

Yapılan işlem:
```
temp←SP
PUSH AX
PUSH CX
PUSH DX
PUSH BX
PUSH temp
PUSH BP
PUSH SI
PUSH DI
```

[96] POPA komutunda da SP yazmacının yığından alınma sırası geldiğinde POP SP yerine ADD SP, 2 komutunun kullanıldığına dikkat ediniz. Yığın, yordamlar arasında parametre aktarmak ve dönüş adreslerini saklamak için paylaşılan/kullanılan bir alan olduğu için yığında kesim içi adresi belirleyen SP yazmacının değerinin değişmesi yığın kullanılarak yapılacak olan tüm işlemlerin karışmasına neden olur.

PUSHAD

80386/486

push all general registers – double

32 Bit

PUSHAD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
32 bit'lik tüm yazmaç değerleri aşağıda verilen sıra dâhilinde yığına yerleştirilir. Ancak ESP yazmacının değeri bu aşamada yığına yerleştirilmez.

Yapılan işlem: *temp*←ESP
PUSH EAX
PUSH ECX
PUSH EDX
PUSH EBX
PUSH *temp*
PUSH EBP
PUSH ESI
PUSH EDI

PUSHF

8086/286/386/486

push flag onto stack

16 Bit

PUSHF

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bu komut ile 16 bit ile ifade edilen bayrak değeri yığına yerleştirilir.

Yapılan işlem: SP←SP - 2
SS:[SP]←FLAG

PUSHFD

80386/486

push eflag onto stack

32 Bit

PUSHFD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bu komut ile 32 bit ile ifade edilen bayrak değeri yığına yerleştirir.

Yapılan işlem: ESP←ESP - 4
SS:[ESP]←EFLAG

5.11 Giriş/Çıkış Komutları

İşlemcinin dış dünya (çevre birimler) ile bağlantısını sağlamak üzere giriş/çıkış komutlarından yararlanılmaktadır. Belleğe erişmek için kullanılan adresleme tekniğine benzer olarak giriş/çıkış iskelelerine de (port) erişmek üzere bir adresleme tekniği kullanılmaktadır. Bu komutlar yardımıyla 0–255 sayıları ile adreslenen iskelelere doğrudan erişmek (doğrudan iskele adresleme) mümkün iken numarası 256-65535 arasındakilere erişmek (dolaylı iskele adresleme) gerektiğinde erişilecek olan iskelenin numarasının DX yazmacı yardımıyla komuta işlenen olarak aktarılması gereklidir.

IN 8086/286/386/486

Input 8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
IN acc, idata	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
IN acc, DX	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bu komut dış dünyadan alınan byte, word, doubleword veriyi uygun tipteki akümülatöre (AL/AX/EAX) aktarır. Eğer idata kullanılıyor ise numarası 0-255 arasındaki iskelelere erişilebilir. Eğer DX yazmacı kullanılır ise numarası 256-65535 arasındaki iskelelerden herhangi birine erişilebilir.

Örnek:

IN AX, 41H	41H adresindeki iskeleden AX yazmacına bir word veri alınmaktadır. (Bu işlem 42H adresindeki iskeleyi de kullanır)

INSB 80286/386/486

Input string byte from I/O port 8 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
INSB	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
DX yazmacı üzerinde numarası bulunan iskeleden okunan byte'ın ES:[EDI] adresine yerleştirilmesini sağlar. İşlem sonucunda EDI bir artırılır (DF←0 ise arttırılır, DF←1 ise azaltılır). Bu komut **REP** ön eki ile kullanılmakta olup bu durumda CX yazmacı işlem sayısını tutmak için kullanılır (Korumalı kipte **CPL** ve **IOPL** bayrak değerlerine dikkat edilmelidir).

Yapılan işlem:

```
ES: [EDI] ←port (DX)
if ( DF = 1 ) then
        EDI←EDI + 1
else
        EDI←EDI - 1
endif
```

INSW

80286/386/486

input string word from I/O port

16 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
INSW	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:

INSB komutu ile benzer özellikler göstermektedir. DX yazmacı üzerinde numarası bulunan iskeleden okunan word'ün ES:[EDI] adresine yerleştirilmesini sağlar. İşlem sonucunda EDI iki artırılır (DF←0 ise arttırılır, DF←1 ise azaltılır). Bu komut **REP** ön eki ile kullanılmakta olup bu durumda CX yazmacı işlem sayısını tutmak için kullanılır.

INSD

80286/386/486

input string double word from I/O port

32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
INSD	--	--	-	--	--	--	--	--	--	--	--	--

Açıklama:

INSB ve **INSW** komutları ile benzer özellikler göstermektedir. DX yazmacı üzerinde numarası bulunan iskeleden okunan dword'ün ES:[EDI] adresine yerleştirilmesini sağlar. İşlem sonucunda EDI dört artırılır (DF←0 ise arttırılır, DF←1 ise azaltılır). Bu komut **REP** ön eki ile kullanılmakta olup bu durumda CX yazmacı işlem sayısını tutmak için kullanılır.

OUT

8086/286/386/486

Output to port

8/16/32 Bit

	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
OUT idata, acc OUT DX, acc	--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:

Bu komut akümülatörün (AL/AX/EAX) tutuğu değeri, idata veya DX yazmacının içeriği ile belirlenen numaralı iskeleye aktarır. Eğer idata kullanılıyor ise numarası 0-255 arasındaki iskelelere erişilebilir. Eğer DX yazmacı kullanılır ise numarası 256-65535 arasındaki iskelelerden herhangi birine erişilebilir.

Örnek

MOV DX, 278H OUT DX, AX	İskele numarası 255 den büyük olduğu için bu adres DX yazmacı üzerinden aktarılmakta ve AX yazmacındaki 16 bit'lik değer 278H adresindeki iskeleye yazılmaktadır.(Dışarıya word büyüklüğünde bir değer aktarılacağı için bu işlem sırasında 279H numaralı iskele de kullanılacaktır.)
OUT 21H, AL	AL yazmacındaki byte değeri 21H numaralı iskeleye yazılır.

OUTSB 80286/386/486

Output string byte 8 Bit

OUTSB

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
DS:[ESI] adresinden bir byte'ı DX yazmacında numarası bulunan iskeleye yazar. İşlem sonucunda ESI bir arttırılır (DF←0 ise arttırılır, DF←1 ise azaltılır). Bu komut **REP** öneki ile kullanılmaktadır. Bu durumda CX yazmacı işlem sayısını tutmak için kullanılır.

```
Yapılan işlem:   port (DX) ←DS: [ESI]
                 if      ( DF = 1 ) then
                         ESI←ESI + 1
                 else
                         ESI←ESI - 1
                 endif
```

OUTSW 80286/386/486

Output string word 16 Bit

OUTSW

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
DS:[ESI] adresinden bir word'ü DX yazmacında numarası bulunan iskeleye yazar. İşlem sonunda ESI iki artırılır (DF←0 ise arttırılır, DF←1 ise azaltılır). Bu komut **REP** öneki ile kullanılmaktadır. Bu durumda CX yazmacı işlem sayısını tutmak için kullanılır.

OUTSD 80286/386/486

Output string double word 32 Bit

OUTSD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
DS:[ESI] adresinden bir double word'ü DX yazmacında numarası bulunan iskeleye yazar. İşlem sonunda ESI dört artırılır (DF←0 ise arttırılır, DF←1 ise azaltılır). Bu komut **REP** öneki ile kullanılmaktadır. Bu durumda CX yazmacı işlem sayısını tutmak için kullanılır.

5.12 Durdurma ve Bekletme Komutları

İşlemciler, bazı özel durumların oluşmasına kadar durdurabilir veya ayrı bir matematik işlemci ile birlikte yapılandırıldıklarında uyumlu çalışmayı sağlamak üzere bekletilebilirler. Benzer şekilde işlemcilerin gerek zamanlama gerek ise bellek erişimlerini daha etkin yapabilmek için herhangi bir işlem/hesaplamanın yapılmadığı durumlarda olabilir. Bu amaçlarla kullanılan özel komutlar bu grup altında toplanmıştır.

HLT 8086/286/386/486

halt

HLT

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Bu komut işlendikten sonra, işlemci sıfırlanmadığı (reset) veya bir kesme oluşmadığı sürece durdurulmuş olur. Maskelenemeyen bir kesme (NMI) işlemcinin bu bekleme durumdan kurtulmasını sağlar ve işlemci **HLT** komutunu takip eden komut ile işlemlerine devam eder.

NOP 8086/286/386/486

no operation

NOP

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
İşlemcinin özelliklerinden daha fazla faydalanabilmek amacıyla iyileştirilmiş (optimize) programlarda[97] **NOP** komutundan yararlanılır. Uygulamanın gereği olarak kod içinde yapay gecikmeler de NOP komutu ile sağlanır. Programların kaynak kodunun elde olmadığı zamanlarda çalıştırılabilir nesne kodu (executable) üzerinde değişiklik yapmanın gerektiği durumlar da **NOP** komutu faydalı olmaktadır. Örneğin nesne kodu üzerindeki 3 byte'lık bir komut satırının 2 byte'lık bir komut satırı ile değiştirilmesi durumunda nesne kodu içinde 1 byte'lık alanda eski komuttan arta kalan değer kalacak ve işlemcinin o noktadan itibaren komutları farklı şekilde değerlendirmesine, doğal olarak işleyişin bozulmasına neden olacaktır. Programın içindeki değişiklikten kaynaklanan 1 byte'lık alan NOP komutu ile doldurulacak olursa programın devamı eskiden olduğu gibi doğru çalışacaktır.

WAIT 8086/286/386/486

wait until not busy

WAIT

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
--	--	--	--	--	--	--	--	--	--	--	--

Açıklama:
Matematik işlemci (Math co-processor) kullanılması durumunda işlemcinin matematik işlemciden gelecek sonucu beklemesini sağlamak üzere kullanılan bir komuttur.

5.13 BCD Düzenleme İçin Kullanılan Komutlar

Bilgisayarda oluşan sonuçların herhangi bir çıkış ünitesine (ekran, yazıcı vs.) aktarılması için ASCII veya EBCDIC gibi kodlama sistemlerinden yararlanılmaktadır. Örneğin "*selam*" mesajının bir çıkış ünitesinde görünebilmesi için mesajı oluşturan

[97] Örneğin işlemcinin çift adresten bilgi alması her zaman tercih edilir. Zira tek adresten bir word okunması gereken durumlarda iki okuma işlemi yapılacaktır.

her harfin ASCII karşılığından oluşan dizgenin[98] ilgili üniteye aktarılması gereklidir. Hesaplamalar sonucunda elde edilen değerlerin de kullanıcının rahat yorumlayabileceği şekillere dönüştürülerek çıkış birimlerine aktarılması gereklidir. Bunun için değerlerin öncelikle **BCD** (Binary Coded Decimal / ikili olarak kodlanmış on tabanında sayı) notasyonuna dönüştürülmesi gerekir. Bir değerin ikili olarak kodlanmış on tabanında sayıya (BCD) dönüştürülmesi demek, sayının on tabanındaki gösterime benzer biçimde ve her hane bir byte olacak şekilde düzenlenmesi demektir. Örneğin; 000DH (yani 13) sayısı on tabanındakine benzer ve her hanesi bir byte büyüklükte alana yazılması ile oluşan 0103H değeri sayının BCD ifadesidir. BCD ifadenin bir çıkış ünitesine aktarılması için her byte'ın ASCII karşılığından oluşan dizgenin[99] oluşturulması gereklidir. Yapılan toplama, çıkarma, bölme ve çarpma işlemlerinden önce/sonra gerekli dönüşümleri yapmak için kullanılabilecek olan komutlara bu başlık altında yer verilmiştir.

AAA 8086/286/386/486

ASCII adjust after addition 8 Bit

AAA

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	?	?	--	x	--	?	--	x

Açıklama:

ASCII olarak ifade edilen değerlerin toplanması sonucunda elde edilen değerin uygun BCD kodu şeklinde ifade edilmesini sağlamak üzere kullanılır.

Yapılan işlem:

```
if ( AF  OR ((AL AND 0FH) > 9) then
        AL←( AL + 6) AND 0FH
        AH←AH + 1
        CF←1, AF←1
else
        CF←0, AF←0
endif
```

Örnek:

XOR AX, AX
MOV AL, '6'
ADD AL, '7'
AAA
OR AX, 3030H
	Örneğimizde ASCII kodu ile ifade edilmiş iki sayının toplandıktan sonra elde edilen sonucun BCD olarak ifade edilmesi amaçlanmıştır. AL, 6 sayısının ASCII karşılığı olan 36H değerini alır. Toplama işlemi sonucunda AL←36H+37H=6DH olacaktır. Elde edilen sonucun ekrana yazdırıldığında 13 olarak görünebilmesi için AAA komutu ile dönüşümü yapılacaktır. Bu komuttan sonra AL←03H, AH←AH+1 ve CF←1 değerini alır. Kısaca AX←0103 olmuş olur. AX yazmacında bulunan hanelerin ASCII karşılıklarını elde edebilmek için 3030H ile OR işlemi uygulanır. Böylece elde edilen 3133H değeri ekrana yazdırılacak olursa 13 sayısı görünecektir.

[98] Sırası ile 73H, 65H, 6CH, 61H, 6DH değerleri "selam" mesajını oluşturan harflerin ASCII kodlarıdır.

[99] BCD formundaki sayının 3030H ile OR işlemine sokulması ile 13 sayısının ekranda gösterilmesini sağlayacak ASCII karşılığı olan 3133H dizgisi elde edilecektir.

AAD

8086/286/386/486

ASCII adjust before division

16 Bit

AAD

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	x	x	--	?	--	x	--	?

Açıklama:

Bölme işlemi **öncesinde BCD** olarak ifade edilmiş sayının karşılığı olan değerin AX yazmacında oluşmasını sağlar.

Yapılan işlem: AL←AH*10 + AL
AH←0

Örnek:

MOV AH, '4'	
MOV AL, '5'	AX←3435H
AND AX, 0F0FH	AX←0405H
AAD	İşlem sonrasında AX←002DH (45 sayısı)
MOV BL, 6	
DIV BL	AL←07H (bölen), AH←03H (kalan)
OR AL, 30H	AL←37H olur. Bu ise 7 sayısının ASCII karşılığıdır.

AAM

8086/286/386/486

ASCII adjust after multiplication

8 Bit

AAM

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	x	x	--	?	--	x	--	?

Açıklama:

Çarpma işlemi **sonrasında AL** yazmacında oluşan değerin BCD notasyonundaki karşılığını hesaplar.

Yapılan işlem: AH←AL div 10
AL←AL modulo 10

Örnek:

MOV AL, 4	
MOV AH, 8	
MUL AH	AX←0020H (32 sayısı) Sadece AL yazmacında kalan sonuç değerlendirilir.
AAM	AH←03H, AL←02H
OR AX, 3030H	AX←3332H değerin ekrana yazdırılması durumundaki dizgeyi belirler.

AAS

8086/286/386/486

ASCII adjust after subtraction

8 Bit

AAS

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	?	?	--	x	--	?	--	x

Açıklama:

Yapılan çıkarma işleminden **sonra** elde edilen sonucu BCD notasyonunda verir. CF←0 ise sayı pozitif CF←1 ise negatif kabul edilir. CF←1 ise AL yazmacında oluşan BCD değer on sayısından çıkarılarak aradaki fark bulunur.

Yapılan işlem:

```
if ( AF OR (AL AND 0FH) > 9 ) then
        AL←(AL - 6 ) AND 0FH
        AH←AH – 1
        CF←1, AF←1
else
        CF←0, AF←0
endif
```

Örnek:

MOV AL, '5'
SUB AL, '7'
AAS
	AL←35H
	AL←35H-37H (işlem sonucu 0FEH olup sayı negatiftir.)
	AL←08H, CF←1 (sayı negatif) Bu durumda sonuç –(10-8)=-2'dir.

DAA

8086/286/386/486

Decimal adjust AL after addition

8 Bit

DAA

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	x	x	--	x	--	x	--	x

Açıklama:

Sıkıştırılmış BCD (packet BCD[100]) değerlerin toplanması sonrasında AL yazmacındaki değerin sıkıştırılmış BCD olarak ifade edilmesini sağlar.

Yapılan işlem:

```
if (AF OR ( AL AND 0FH) > 9) then
        AL←AL + 6
        AF←1
else
        AF←0
endif
if ( CF OR ( AL > 9FH)) then
        AL←AL + 60H
        CF←1
else
        CF←0
Endif
```

Örnek:

MOV AL, 75H
ADD AL, 19H
DAA
	Toplama işleminin sonucunda AL←8EH olacaktır. BCD gösterimde haneler 10 sayısından büyük olamayacağı için düşük anlamalı 4 bitte bulunan EH değeri içinden on çıkartılarak üst haneye elde aktarılır. Böylece sonuç AL←94H olur.

[100] Onluk sistemdeki bir sayının hanelerinin 4 bit olarak ifade edilip ikili sistemde kodlanmasıdır.

DAS 8086/286/386/486

decimal adjust AL after subtraction 8 Bit

DAS

OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF
?	--	--	--	x	x	--	x	--	x	--	x

Açıklama:

Sıkıştırılmış (packet) BCD iki sayının çıkarma işleminden **sonra** kalan sayının geçerli bir sıkıştırılmış (packet) BCD sayı olmasını sağlar.

Yapılan işlem:

```
if ( AF OR ((AL AND  0FH) )  > 9 ) then
        AL←AL - 6
        AF←1
else
        AF←0
endif
if ( CF OR ( AL > 9FH )) then
        AL←AL - 60H
        CF←1
else
        CF←0
Endif
```

Örnek:

MOV AL, 42H
SUB AL, 13H
DAS
	AL←42H-13H işleminin sonucu 2FH'dir. Ancak bu sayılar BCD notasyonunda olduğu için çıkarma işlemi sayılar sanki onluk sistemdeymiş gibi yapılacaktır. Bu durumda 42-13 işlemi yapılmış ve sonuç 29 olarak bulunmuş olur. Ancak değer doğal olarak on altı tabanında ifade edileceği için AL←29H olacaktır.

5.14 Diğer Komutlar

80x86 ailesinin işlemcileri geriye doğru kod uyumludur. Yani her yeni nesil işlemci eski nesil işlemcilerin sahip olduğu komutları desteklemekle birlikte gelişen teknoloji ve değişen ihtiyaçları karşılamak üzere yeni komutlar ile donatılmışlardır. İşlemcilerin çalışma kiplerine (gerçek kip, korumalı kip vb.) bağlı olarak da komut kümesi zenginleşmiştir. Bu kitap gerçek kipte programlamayı hedef aldığı için sadece korumalı kipte en yaygın kullanılan komutları tanıtmayı amaçlamaktadır. Ancak bu kitapta tanıtılmayan başka komutların da olduğu göz ardı edilmemelidir. Benzer şekilde kayan noktalı işlemler için geliştirilmiş (gerek ayrı gerek ise mevcut işlemci ile bütünleştirilmiş matematik işlemci üzerinde kullanılabilen) komutlar da bu kitap kapsamında ele alınmamıştır.

6. 80X86 AİLESİNDE ADRESLEME KİPLERİ

Anımsatıcı (mnemonic) olarak adlandırdığımız sembolik komutların işlenenlere erişimine adresleme adı verilir. 80x86 ailesi işlemcilerde kullanılan sekiz ayrı adresleme kipi vardır. Bunlar;

6.1 Anlık (Immediate) Adresleme

İkinci işlenenin idata (sayı veya ASCII değer) olduğu tipteki adresleme anlık adreslemedir. İlk işlenen ile ikinci işlenenin tipleri aynı olmalıdır.

Örnek 6-1

```
MOV AH, 06H        ; AH←06H
MOV CX, 1FFCH      ; CX←1FFCH
MOV AL, 'A'        ; AL←41H
```

6.2 Yazmaç (Register) Adresleme

Yazmaç adresleme bir yazmaçtan bir diğerine değer aktarımı için kullanılır. Komutun iki işleneni de aynı tipte yazmaç olmalıdır.

Örnek 6-2

```
MOV AH, AL   ; AH←AL
MOV CX, AX   ; CX←AX
```

6.3 Doğrudan (Direct) Adresleme

Doğrudan adreslemede işlenenin bulunduğu yerin adresi bir etiket (label) olarak verilir. Bu etiket byte veya word tanımlanmış bir değişken olabilir. Ancak kullanılan yazmacın tipi etiket ile uyuşmalıdır.

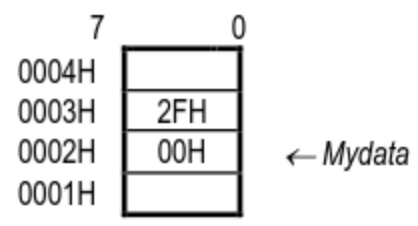
Örnek 6-3

Mydata word olarak tanımlanmış ise;

MOV AX, *Mydata* ; AX←2F00H

Mydata byte olarak tanımlanmış ise;

MOV AL, *Mydata* ; AL←00H



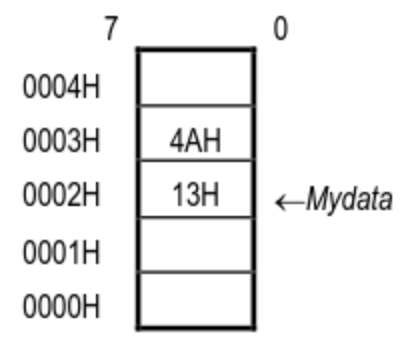
Şekil 6-1: Örnek 6-3 için bellek haritası

6.4 Yazmaç Dolaylı (Register Indirect) Adresleme

Yazmaç dolaylı adreslemede işlenenin bellekteki adresi bir indis yazmacı (SI, DI veya BX) ile gösterilmektedir.

Örnek 6-4

```
MOV SI, OFFSET Mydata   ; SI←0002H
MOV AX, [SI]            ; AX←4A13H
        veya
LEA DI, Mydata          ; DI←0002H
MOV AX, [DI]            ; AX←4A13H
```



Şekil 6-2: Örnek 6-4 için bellek haritası

Örneğimizdeki işlemlerde sırası ile SI ve DI yazmaçları veriye erişmek için gerekli adres bilgisini tutmaktadır. **OFFSET** sözde komutu ile **LEA** anımsatıcısı *Mydata* isimli değişkene erişmek için gerekli olan göreli konum değerinin ilgili yazmaçlara yüklenmesini sağlarlar.

6.5 Baz Göreli (Base Relative) Adresleme

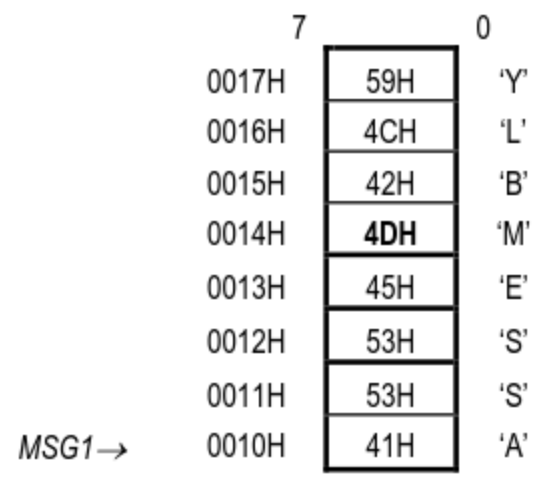
Baz göreli adreslemede işlenenin bellekteki adresi BX yazmacında bulunur. BX yazmacındaki değere ilave edilecek değerler ile göreli olarak bellek üzerindeki farklı adres alanlarına erişerek veri okumak/yazmak mümkündür. Özellikle dizi işlemlerinde sıklıkla kullanılan bir adresleme tekniğidir.

Örnek 6-5

Kodumuz içerisinde aşağıdaki biçimde bir tanımla yapılmış olsun. Bu tanımlamaya göre değerler bellekte Şekil 6-3'de gösterildiği gibi yerleşmiş olacaktır.

```
MSG1 DB 'ASSEMBLY'

LEA BX, MSG1        ; BX←0010H
MOV AL, [BX+4]      ; AL←4DH
```



Şekil 6-3: Örnek 6-5 için bellek haritası

BX yazmacı veri kesimi içinde tanımlanmış olan *MSG1* isimli bellek bölgesinin başlangıç adresini göstermektedir. Böylece [BX+4] ifadesi ile BX yazmacında belirlenen başlangıç adresi baz alınarak 4 byte uzaktaki (göreli) bellek gözünde bulunan byte AL yazmacına aktarılmaktadır.

6.6 Doğrudan İndisli (Direct Index) Adresleme

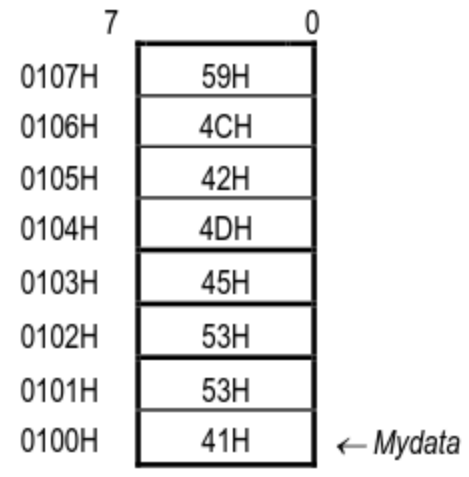
Doğrudan indisli adreslemede sıra grubu yazmaçlarından biri kullanılarak bellekte bulunan veriye erişilmeye çalışılır. Yüksek seviyeli dillerde kullanılan dizi adreslemeye çok benzemekle birlikte dikkat edilmesi gereken bazı farklılıklar vardır. Bu özelliklerin dikkate alınmaması durumunda yazılacak programlarda ciddi hatalar oluşabilir.

Örnek 6-6

Bu yapının özelliklerini incelebilmek için *Mydata* isimli değişkenin aşağıdaki biçimde tanımlandığını varsayalım;

Mydata DB 41H, 53H, 53H, 45H, 4DH, 42H, 4CH, 59H

Bunun bellekteki yerleşimi Şekil 6-4'da verildiği gibi olur;



Şekil 6-4: Örnek 6-6 için bellek haritası

Mydata ismiyle tanımlanmış yapıya bir dizi şeklinde erişmek için SI ve/veya DI (hatta BX) yazmaçlarını birer indis olarak kullanabiliriz. Bu durumda yüksek seviyeli dillerde olduğu gibi o yazmaçların değerlerini dizinin ilk elemanından başlayarak son elemanını gösterebilecek biçimde ayarlamamız gerekecektir. C programlama dilindeki kullanımına benzer olarak assembly dilinde de dizinin ilk elemanı 0 indisli olanıdır. Buna göre dizinin ilk elemanına (0. indisli olanı) erişmek istediğimizde;

```
XOR SI, SI            ; SI←0000H
MOV AL, Mydata[SI]    ; AL←41H
```

komutlarını yazarız. **MOV** komutu bilgisayar tarafından değerlendirilirken önce *Mydata* değişkeninin adresi (effective address) bulunur ve buna SI yazmacındaki değer eklenir. Bulunan yeni değerin bellek üzerinde gösterdiği byte AL yazmacına yerleştirilir. Sembolik olarak göstermek gerekirse;

$AL \leftarrow [EA_{Mydata} + SI]$

işlemi yapılmış olacaktır. Dizinin ikici elemanı (1. indisli olanı) ise (DB ile tanımlanmasından dolayı) bir sonraki adreste olacaktır. *Mydata* isimli tanımın yeri değişmediğine göre, ikinci elemana erişmek için SI değerinin bir defa arttırılması[101] ile bu sağlanır. Kısaca dizinin ikinci elemanına erişmek için bu kodun devamına şu komutları yazmak yeterli olacaktır.

```
INC SI              ; SI←0001H
MOV AH, Mydata[SI]  ; AH←53H
```

Bu örnekte, **MOV** komutu AL, AH ve *Mydata*'yı işlenen olarak almaktadır. **MOV** komutunun doğru çalışması için işlenenlerin tiplerinin aynı olması gereklidir. Aksi halde derleyici bu satıra "**Type Missmatch**" (tip uyuşmazlığı) hatası verecektir.

Örnek 6-7

Bir önceki örnekte verilen dizinin byte olarak değil de word olarak tanımlandığını düşünecek olursak aşağıdaki şekilde veri tanımı yapılması gerekecektir.

```
Mydata DW 5341H, 4553H, 424DH, 594CH
```

Bu tanımlamanın bellekteki görüntüsü de Şekil 6-4'teki gibi olacaktır. Dizimizin ilk elemanına (0.indisli olanına) erişmek için;

```
XOR SI, SI          ; SI←0000H
MOV AX, Mydata[SI]  ; AX←5341H
```

komut satırlarını yazmak yeterli olacaktır. Dizinin ikinci elamanı (1.indisli elamanı) dizinin word olarak tanımlanmış olması nedeniyle 2 byte ileride olacaktır. Buna göre ikinci elemana erişebilmek için;

```
ADD SI, 2           ; SI←0002H
MOV AX, Mydata[SI]  ; AX←4553H
```

komutları verilmelidir. Bir önceki örneğimizde olduğu gibi burada da **MOV** komutunun işlenenleri aynı tipte tanımlanmış olmalıdır. Aksi halde derleyici hata verecektir.

Dizi erişimlerinde yazmaç dolayı adresleme tekniğinden de yararlanılabilir. Bu durumu da Şekil 6-4'ü kullanarak Örnek 6-8 üzerinde inceleyelim.

Örnek 6-8

```
LEA SI, Mydata  ; Mydata isimli değişkenin adresi SI yazmacında saklanıyor. SI←0100H
MOV AL, [SI]    ; AL←41H
```

[101] Byte tanımlı değişken için artım 1, word tanımlı değişken için artım 2 ve dword tanımlı değişken için artım 4'tür.

Bu yöntemde indis yazmacının değerini sıfırdan başlatmak yerine dizinin başlangıç adresi (effective address) SI yazmacına yerleştirilmektedir. Diğer teknikten farklı olarak her seferinde **MOV** komutunda *Mydata* kullanılmadığı için başlangıç adresinin hesaplanması gerekmeyeceği gibi derleyici tarafından özel bir tip kontrolü yapılmayacak, doğrudan ilk işlenenin tipinde olduğu kabul edilecektir.

Her ne kadar yukarıdaki komutlar doğru çalışsa da bunun kurallar uyarınca aşağıdaki şekilde yazılması gereklidir.

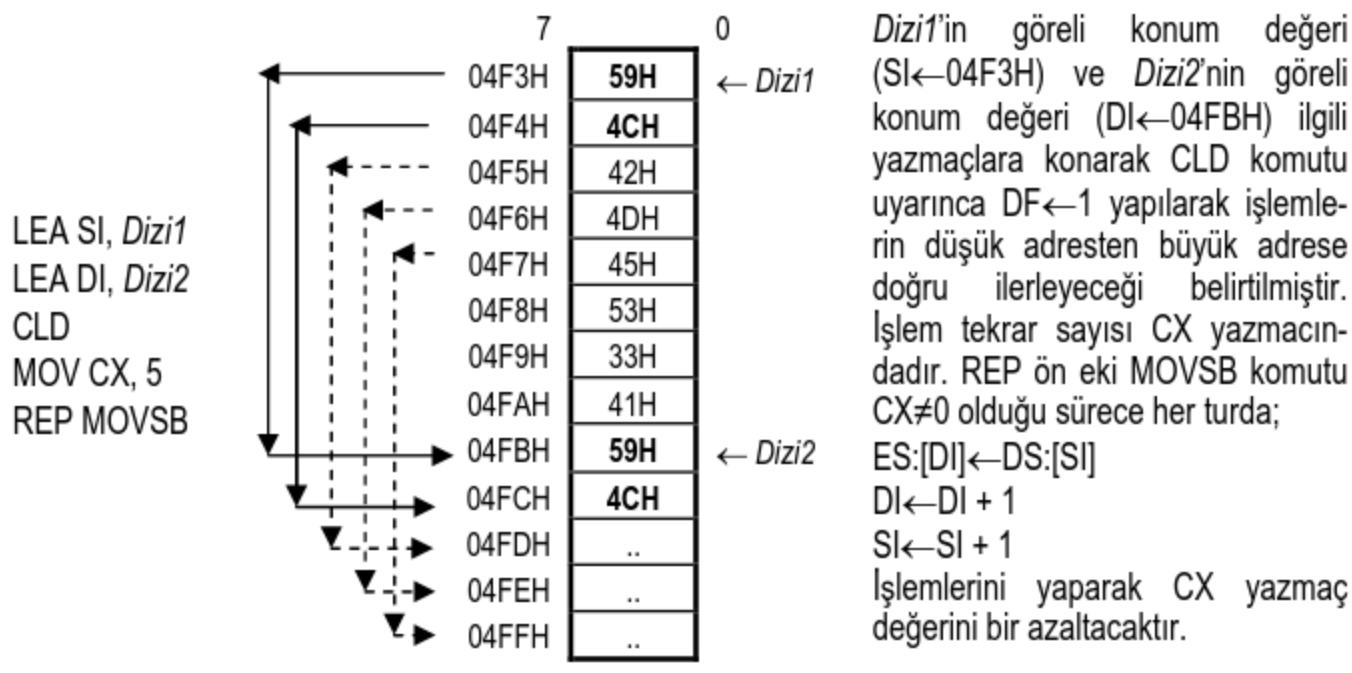
LEA SI, *Mydata*	; Mydata isimli değişkenin adresi SI yazmacında SI←0100H
MOV AL, BYTE PTR [SI]	; SI yazmacının gösterdiği bellek gözünden bir byte alınarak AL yazmacına yerleştirileceği özellikle belirtilmektedir. AL←41H

6.7 Dizgi (String) Adresleme

Bu tür adreslemeler bellek üzerinde bir blok verinin yerinin değiştirilmesi, bir bellek bölgesinin belirli bir sayı ile doldurulması, bir bellek bölgesinde istenen bir sayının aranması gibi işlemlerde yaygın olarak kullanılmaktadır. Tekrarlanan türde işlemler olmaları dolayısıyla **REP, REPZ** ve **REPNZ** ön eklerinden yararlanılmaktadır. **MOVS**, **CMPS**, **SCAS**, **LODS** ve **STOS** komutlarında bu tipte adreslemeden yararlanılır.

Örnek 6-9

Şekil 6-5'te verilen bellek haritası uyarınca *Dizi1* adresinden başlayan 5 byte veriyi *Dizi2* adresinden itibaren yerleştirmeye çalışalım;



Şekil 6-5: Örnek 6-9 için bellek haritası

6.8 İskele (Port) Adresleme

80x86 işlemcisinin dış dünya ile bağlantısı iskele adı verilen iletişim noktaları üzerinden gerçekleştirilir. İskele numaraları 16 bit ile ifade edildikleri için 65.535 adet farklı adres kullanılabilir. İskele adresleri bellek adresi ile karıştırılmamalıdır. İskele adresleme sırasında işlemcinin A0-A15 numaralı adres bacakları aktif iken A16-A19 numaralı adres bacakları 0 durumundadır. Ayrıca kontrol bacaklarında bu adreslemenin bellek için mi? iskele için mi? yapıldığını belirleyen sinyaller olacaktır.

İskele adreslemenin iki temel şekli vardır. Bunlardan birincisi ilk 256 adet adrese (0-255) erişmeyi sağlayan doğrudan iskele adresleme, ikincisi ise daha büyük adreslere (256-65535) erişmeyi sağlayan dolaylı iskele adreslemedir.

6.8.1 Doğrudan İskele Adresleme

Bir byte ile ifade edilebilen bir işlenen olması nedeniyle 0–255 numaralı adreslere doğrudan erişme imkânı sağlar.

Örnek 6-10

```
IN AL, 30H     ; 30H adresindeki iskeleden okunan 8 bit'i AL yazmacına koyar
```

6.8.2 Dolaylı İskeleme Adresleme

Doğrudan iskele adresleme ile erişilemeyen adreslere DX yazmacı yardımıyla erişilmektedir. Bu nedenle erişim dolaylı olarak adlandırılır. Erişilmesi istenen adres numarası DX yazmacına yerleştirilerek komut kullanılacaktır.

Örnek 6-11

```
MOV DX, 03F8H      ; DX←03F8H
OUT DX, AX         ; DX ile belirlenen iskeleye AX yazmacındaki veriyi koyar.[102]
```

[102] Dışarı çıkarılacak olan değer word tipinde olduğu için işlem sırasında 03F9H adresi de kullanılacaktır.

7. ASSEMBLY DİLİNDE PROGRAM YAZMAK İÇİN NELER GEREKLİDİR?

Assembly dilinde program yazmak ve çalıştırabilmek için çok fazla kaynağa ihtiyaç yoktur. Diğer programlama dillerinin ihtiyaç duyduğu onlar hatta yüzlerce MB ile ifade edilen bellek ve disk alanlarına assembly ile programlama yaparken gerek duyulmaz. Son derece basit donanımlar üzerinde assembly programlar yazıp çalıştırmak ve çalışmalarını incelemek mümkündür. Bunun için gerekli programların bulunduğu çalışma ortamının hazırlanması yeterlidir (bkz. 16.1 Çalışma Ortamının Kurulması). Çalışma ortamında bulunması gereken programlar ve ne amaçla kullanıldıkları aşağıda açıklanmıştır;

1. **Metin Düzenleyici:** Assembly program yazmak, ikincil bellek üzerine kayıt etmek, mevcut hataları düzeltmek için metin düzenleyiciden yararlanılır. Assembly için komut satırından (command prompt) çalıştırılabilir herhangi bir metin düzenleyici program yeterli olacaktır.

2. **Assembly Dili Derleyicisi**: Assembler olarak bilinen bu derleyici, kullandığımız işlemci ailesine ait anımsatıcılar ile yazılan ASM uzantılı dosyaların assembly dili yazım kurallarına uygun olarak yazılıp yazılmadığını denetlemek ve OBJ uzantılı dosyaları oluşturmak üzere kullanılacaktır. Yaygın kullanılan derleyiciler arasında Microsoft firmasının Microsoft Macro Assembler (MASM.EXE) ve Borland firmasının Turbo Assembler (TASM.EXE) isimli programlarını sayabiliriz. Bunlar dışında Internet üzerinden bedava temin edilebilen çeşitli assembly dili derleyicileri de kullanılabilir (bkz. dipnot 174).

3. **Linker:** Derleyici tarafından oluşturulan OBJ uzantılı dosyalardan çalıştırılabilir (executable EXE / COM) türde program yaratabilmek üzere Linker isimli programdan yararlanılmaktadır. Microsoft Macro Assembler kullananlar için LINK.EXE, Borland Turbo Assembler kullananlar için TLINK.EXE gerekli olan programlardır.

4. **EXE-COM Dönüştürücü:** Assembly dilinde yazılan programlar EXE veya COM yapısında tasarlanabilir. Bu iki farklı çalıştırılabilir program arasındaki en temel farklılık kesim organizasyonlarındadır. Ancak Assembler/Linker ikilisi her iki tip derlenip link işleminden geçirildikten sonra EXE uzantılı bir dosya üretmektedir. Eğer program COM türünün gereklerine uygun olarak tasarlanmış ise, çalışabilmesi için EXE uzantılı dosyanın bir işlemden daha geçirilerek COM uzantılı hale getirilmesi gereklidir. Bu amaçla işletim sisteminin (DOS) içinde bulunan EXE2BIN.EXE[103] veya EXE2COM.EXE isminde üçüncü parti bir dönüştürücü programa ihtiyaç olacaktır.

---

[103] EXE2BIN.EXE isimli program DOS işletim sisteminin eski sürümlerinde (versiyon 5.0 kadar olanlar) dağıtılmakta iken üst sürümlerde dağıtımı durdurulmuştur. Bunun yerine Chris Dunfond / The Core Softwre Group tarafından yazılmış EXE2COM.EXE isminde bir dönüştürücü program Internet üzerinden temin edilebilir.

5. **Debugger:** Oluşturulan EXE veya COM uzantılı çalıştırılabilir kodların doğru çalışıp çalışmadığını test etmek amacıyla debugger adı verilen yardımcı programlar kullanılır. Bu programlar sayesinde her komut satırı adım adım çalıştırılarak o anki yazmaç değerleri, bellek ve yığın durumu izlenebilir. Debugger yardımcı programı kullanılmakta olan işletim sisteminin bir parçası olarak DEBUG.EXE adı altında bulunabileceği gibi Microsoft Assembler ile birlikte dağıtımı yapılan Codeview (CV.EXE) veya Turbo Assembler ile dağıtımı yapılan Turbo Debugger (TD.EXE) isimli programlar da yaygın olarak kullanılmaktadır.

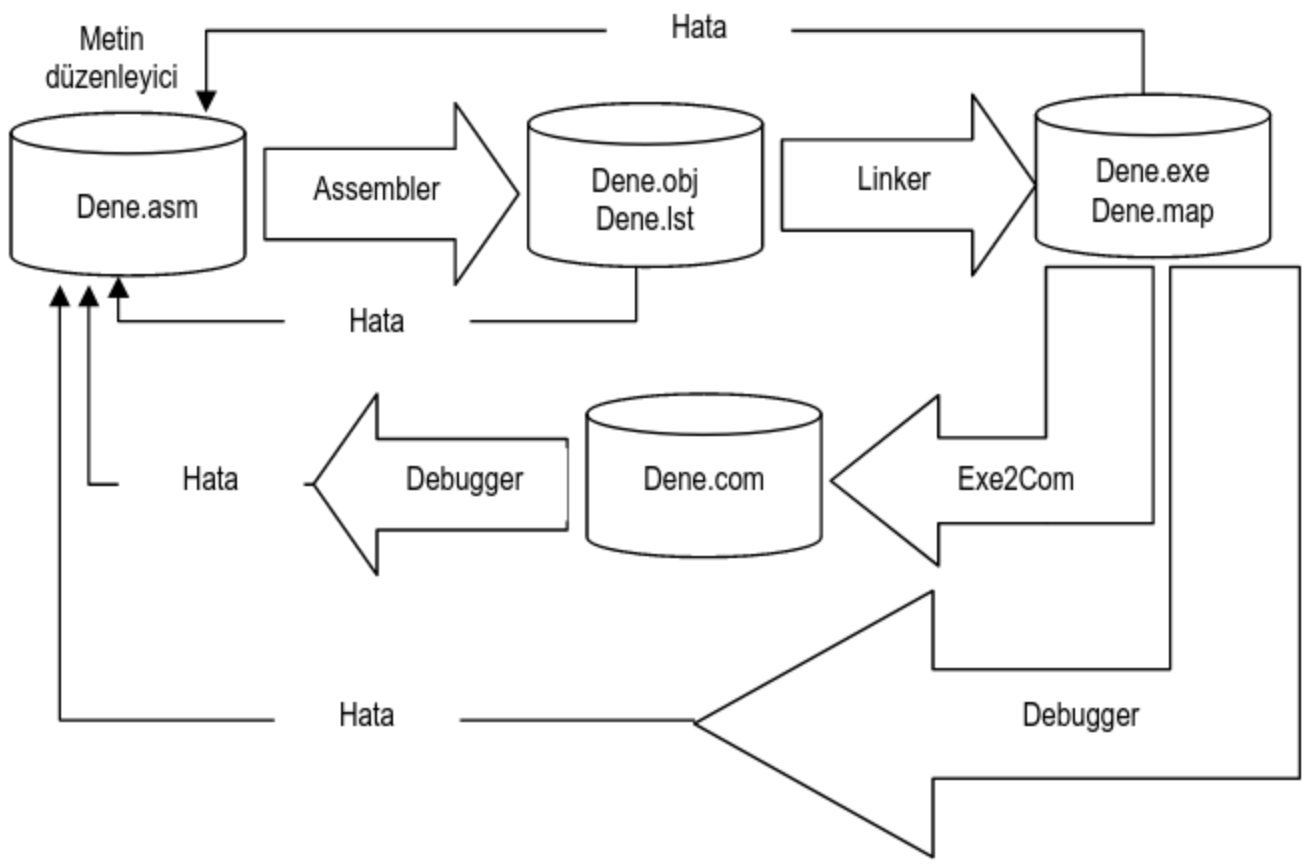
6. **Sistem Dosyaları:** 80x86 işlemci ve yaygın kullanılan işletim sistemlerinden birine sahip bilgisayar sistemleri üzerinde çalışma ortamını tesis edilerek assembly dilinde program yazmak, derlemek, yazılan kodu incelemek mümkündür. Buna rağmen kullanıcılar bilgisayarı açmayı sağlayacak kadar basit bir işletim sistemini (örneğin DOS) disket veya USB bellek gibi ikincil bellek birimleri üzerine kurup, çalışma için gerekli metin düzenleyici, assembly dili derleyicisi, linker, EXE-COM dönüştürücü ve Debugger programlarını kopyalayarak en basit, en düşük donanımlı bilgisayar sisteminde bile assembly kodlama yapılabilir.

Genel olarak assembly dilinde program yazmak için yukarıda tarif edilen araçların uygun sırada kullanılması büyük kolaylıklar sağlamaktadır. Ancak assembly komutlarının işlevlerini, yazım kurallarını ve oluşan sonuçlarını incelemek için daha önceki bölümlerde verilen bazı örneklerin küçük program parçaları olarak debug ortamında yazılıp denenmesi ve işleyişin anlaşılması faydalı bir çalışmadır. Diğer yandan, debug ortamında değişken tanımlamak yerine bellek adresinin, koşul ifadelerinde ve çevrimlerde kullanılan etiketlerin yerine ilgili komut satırının adresinin yazılması yeni başlayanlar için bazı zorluklar yaratacaktır.

Assembly programlama dili küçük/büyük harf duyarlı değildir. Önemli olan herhangi bir metin düzenleyicisi kullanarak assembly yazım kurallarına uygun[104] şekilde kodun yazılması ve **ASM** uzantılı olarak ikincil bellek birimine saklanmasıdır. Assembly dilinde yazılan programların çalıştırılabilmesi için her şeyden önce **ASM** uzantılı dosyanın derleyiciden hatasız bir şekilde geçirilmesi gerekmektedir. Derleyici genel olarak programın yazımına ilişkin hatalar ve işlenenler arasındaki tip uyuşmazlıklardan (type missmatch) kullanıcıyı haberdar edecektir. Programın derleyiciden hatasız olarak çıkması ile **OBJ** uzantılı bir dosya elde edilir. Benzer şekilde, programın derleyici tarafından düzenlenmiş

[104] İlerideki bölümlerde anlatılacak olan EXE veya COM organizasyonun gereklerine ve bu yapılarda kullanılacak sözde komutların (pseudo code) kullanımına dikkat edilmelidir. Anımsatıcıların (mnemonic) komut kümesinde tarif edildiği biçimde kullanılması, komut işlenenlerinin tiplerinin uyumlu, sayı sistemlerinin belirgin ve değişken isimlerinin uygun olması (harf ile başlayıp harf ve/veya sayı ile devam etmesi, içinde Türkçe de kullanılan “ı”, “ğ”, “ü”, ”ş”, “ç”, “ö”, ”İ”, ”Ğ”, “Ü”, “Ş”, “Ç” ve “Ö” harflerinin kullanılmaması) dikkat edilmesi gereken en temel kurallardır.

olan, **LST** uzantılı dosyası da oluşacaktır. **OBJ** uzantılı dosya linker programına verilerek **EXE** uzantılı dosya elde edilmelidir. Bu aşamada kullanıcı isterse, EXE uzantılı dosyanın yanı sıra kesimlerin bellekte ne büyüklükte yer tuttuğunu ve hangi isimler ile erişildiklerini gösteren **MAP** uzantılı bir dosyayı da üretebilir. Eğer link işlemi sırasında bir hata oluşursa bu hatanın ne olduğuna dair bazı bilgiler ekranda görünecektir. Oluşan hata mesajları ışığında programcının yazdığı program üzerinde düzeltmeler yapması gerekir. Linker programının hata vermeden sonlanmasını takiben **EXE** uzantılı dosya oluşmuş olacaktır. Ancak programın COM tipinde tasarlanmış olması durumunda önce EXE2BIN veya EXE2COM isimli programlar yardımıyla **COM** uzantılı olacak şekilde dönüştürülmelidir. EXE veya COM uzantılı dosyalar elde edildikten sonra programın istenilen sonucu üretip üretmediği, var ise mantık hataları ve uç nokta problemleri debugger programı (Debug ile ilgili olarak 16.2. bölüme bakınız) kullanılarak test edilir.



Şekil 7-1: Assembly programın derlenmesi ve çalıştırılma aşamaları

Kaynak koddan test aşamasına varıncaya kadar geçen her adımda oluşan hata için programın koduna dönülerek gerekli düzeltmelerin yapılması ve aynı işlemlerin sırası ile tekrarlanması gerekir. Assembly dilinde programlama için geçilmesi gerekli tüm aşamalar Şekil 7-1 de gösterilmiştir.

Buna göre komut satırından yazılması gerekli komutlar Tablo 7-1'de sıralanmıştır.

Tablo 7-1: Assembly programın derlenme ve çalıştırılma aşamaları

	İşlem	Açıklama
1	A:\>**masm** *dene.asm*	Verilen komutu takiben derleyici OBJ, LST ve CRF uzantılı dosyaların isimlerinin ne olması gerektiğini kullanıcıya sorar[105]. Ardından programın kaynak kodundaki olası yazım hataları ve tip uyuşmazlıkları incelenecek ve hata var ise bu hataların hangi satırlarda ve ne tip hatalar oldukları ekranda görüntülenir. Hata oluşmuş ise OBJ uzantılı dosya yaratılmaz. Kaynak kodun (ASM uzantılı dosyanın) gözden geçirilmesi ve ekranda beliren hata mesajlarına göre düzeltilmesi gereklidir.
2	A:\>**link** *dene.obj*	Verilen komutu takiben linker EXE, MAP ve LIB uzantılı dosyaların ne olması gerektiğini kullanıcıya sorar[106]. Ardından link işlemi başlar ve var ise hatalar ekranda görüntülenir. Bu aşamanın hatasız tamamlanmasını takiben EXE uzantılı program oluşmuş olacaktır. Programın EXE organizasyonuna uygun olarak kodlanmış ise **5**. adımdan, COM organizasyonuna uygun kodlanmış "**no stack segment**" (yığın tanımlanmamış) hata mesajı alınacaktır. Ancak COM programın yapısı gereği bu mesaj göz ardı edilerek işleme 3. adımdan devam edilir. Hata olması durumunda tekrar başa dönülerek ASM uzantılı dosyanın gözden geçirilmesi ve hatayı oluşturan satır(lar)ın düzeltilmesi gereklidir.
3	A:\>**exe2com** *dene.exe dene.com*	COM organizasyonuna uygun kod yazılmış olması durumunda bir önceki adımda elde edilen EXE uzantılı programın bir EXE2COM isimli dönüştürücü yazılım ile COM uzantılı hale getirilmesi gereklidir.
4	A:\>erase *dene.exe*	COM organizasyonunda kod üretildiğine göre EXE uzantılı dosya hiçbir zaman kullanılmayacaktır. Bu dosyanın ikincil bellek üzerinde boşuna yer tutmaması için silinmesi faydalı olacaktır.
5	A:\>**debug** [*dene.exe* | *dene.com*]	Elde edilen çalıştırılabilir kodun nasıl çalıştığının incelenebilmesi amacıyla debug programından yararlanılır. Üretilen çalıştırılabilir dosya hangi tipte ise ona göre tam olarak ismi DEBUG programının yanına tam olarak yazılmalıdır.[107]

[105] OBJ uzantılı dosyanın ismi, derleyiciye verilen ASM uzantılı dosyanın ismi ile aynı kabul edilir. Kullanıcının LST ve CRF uzantılı dosyaları üretmek istemeyebileceği düşünülerek bunların isimleri NUL olarak belirlenmiş olarak görüntülenir. Kullanıcı bunlar için bir isim yazarsa LST ve CRF dosyaları oluşacaktır. LST uzantılı dosya, derleme aşamasında ekranda beliren mesajların da içinde yer alacağı bir dosyadır.

[106] EXE uzantılı dosyanın ismi, linker'a verilen OBJ uzantılı dosyanın ismi ile aynı kabul edilir. Kullanıcının MAP ve LIB uzantılı dosyaları üretmek istemeyebileceği düşünülerek bunların isimleri NUL olarak görüntülenir. Kullanıcı bunlar için bir isim yazarsa MAP dosyası oluşacaktır. MAP uzantılı dosya, link işlemini takiben uygulamanın kullanacağı kesimlerin isimleri ve büyüklükleri ile ilgili bilgileri barındırır.

[107] COM yapısında program yazılmış ilk üç adımın hatasız tamamlanmasına rağmen 4.adımda EXE uzantılı dosya silinmemiş ve 5.adımda debug programına hatalı olarak EXE uzantılı dosyanın adını yazmışsanız anlamlı bir kod görmeniz mümkün olmayacaktır.

8. SÖZDE (PSEUDO) KOMUTLAR/İŞLEMLER

Sözde komutlar, assembly programın düzenlemesine yönelik olup anımsatıcılar (mnemonic) gibi doğrudan makine kodu karşılığı olmayan komutlardır. Makine kodu karşılıkları olmadığı için bayraklar üzerinde bir etkileri olmadığı gibi debug ortamında da kullanılmaları mümkün değildir. Bu komutlar derleyici tarafından üretilen LST uzantılı dosyanın düzeni, program kesim düzenlemesi ve veri tanımlamaları için kullanılırlar. Bu komutlardan özellikle sayfa düzenlemesine yönelik olarak kullanılanlar seçimlik olup kullanılıp kullanılmamaları LST uzantı dosyanın yapısı dışında bir etki yaratmaz. Ancak kesim ve veri tanımlamaya yönelik olarak kullanılan sözde komutlar her programın tipinde uygun parametreler ile kullanılırlar. Sözde komutlar derleyici tarafından değerlendirilerek gerekli düzenlemeler yapılmaktadır. Sözde komutlar genel amaçlı ve koşullu kullanılanlar olmak üzere iki ana başlık altında incelenebilir.

8.1 Genel Amaçlı Sözde Komutlar

Genel amaçlı sözde komutlar, dosya düzenleme, kesim düzenleme, veri tanımlama, yordam düzenleme, makro düzenleme ve diğer sözde komutlar olarak gruplanabilir.

8.1.1 Dosya Düzenlemede Kullanılan Sözde Komutlar

PAGE

Açıklama:
Assembly programın derlenmesini takiben üretilen LST uzantılı dosyanın kaç satıra kaç sütun olarak düzenleneceğini belirtmek için kullanılan sözde komuttur. **PAGE** seçimlik türde bir komuttur Kullanılmaması durumunda üretilecek LST uzantılı dosya 66 satır, 80 sütun olarak tanımlı (varsayılan) değerler kullanılarak üretilecektir. **PAGE** sözde komutu kullanılırken parametrelerinin, satır için 10–255, sütun için de 60–132 değerleri arasında verilmesi gereklidir.

Örnek:

PAGE *60, 80*	Derleyicinin oluşturacağı LST uzantılı dosyanın 60 satır, 80 sütun şeklinde düzenlenmesini sağlar.

TITLE

Açıklama:
Derleyici tarafından üretilen LST uzantılı dosyanın her sayfanın başına yazılması istenen bir başlık var ise bu **TITLE** sözde komutu kullanılarak yazılır. Bu şekilde **PAGE** ile ebadı ayarlanan her sayfanın başına **TITLE** sözde komutunu takip eden başlık yazılır.

Örnek:

TITLE *ilk assembly programım*	Derleyicinin oluşturacağı LST uzantılı dosya yazıcıdan basıldığında her sayfanın üstünde *ilk assembly programım* yazısının bulunduğu görülecektir.

8.1.2 Kesim Düzenlemede Kullanılan Sözde Komutlar

80x86 işlemcileri gerçek kipte kullanılırken belleğe kesim adı verilen 64 KB büyüklüğündeki bloklar aracılığıyla erişir. Kesimler, kullanıcının üzerinde çalışacağı veriyi, yığını ve kodu barındıran bellek alanları olup, assembly programın tipine özgü olarak tanımlanmak zorundadır[108]

SEGMENT / ENDS

Açıklama:

SEGMENT sözde komutu kullanılacak kesimin ne amaçla kullanılacağına bağlı olarak farklı seçenekler ile kullanılabilir. En genel hali ile bir kesim tanımı aşağıdaki şekilde yapılır.

```
kesim-ismi      SEGMENT { seçenekler }
                ........
                ;kesim  içinde yer alması gereken tanımlar/komutlar.
                ........
kesim-ismi      ENDS
```

Burada dikkat edilecek olursa **SEGMENT** tanımı **ENDS** (end of segment) sözde komutu ile bitmektedir. Kesim tanımlarken kullanılabilecek olan seçenekler şu genel başlıklar altında incelenebilir;

a) Hizalama (Alignment) tipi:

Tanımlanan kesimin hangi sınırdan (boundary) başlayacağını belirtmek üzere kullanılır. Geçerli olan değerler;

BYTE	Kesimin herhangi bir adresten başlamasını sağlar.
WORD	Kesimin çift adresten başlamasını sağlar.
PARA	Kesimin 16 sayısına tam bölünebilir bir adresten başlamasını sağlar.
PAGE	Kesimin 256 sayısına tam bölünebilir bir adresten başlamasını sağlar.

Bazı yüksek seviyeli diller ile bağlantılı kesim tanımlamaları yapılmasının gerekli olduğu durumlarda yüksek seviyeli programlama dilinin özelliğine göre uygun hizalama tipi kullanmak gereklidir[109].

b) Birleştirme (Combine) tipi:

Link işleminden sonra aynı isim verilmiş olan kesimlerin fiziksel bellek üzerinde nasıl yer alacaklarını, birbirlerine göre durumlarını, eğer bu kesim yığın olarak kullanılacak ise SS ve SP yazmaçlarının nasıl ayarlanacağını belirlemek için kullanılmaktadır. Geçerli olan değerler;

PUBLIC	Aynı isim verilmiş olan kesimlerin bellekte birbirinin devamında yer alacak şekilde yerleştirilmesini sağlar.
COMMON	Aynı isim verilmiş olan kesimlerin bellekte birbirinin üzerine denk gelecek şekilde yerleştirilmesini sağlar.
STACK	Tanımlanan kesimin yığın mantığından (LIFO) kullanılmasını sağlar.
AT #####	Tanımlanan kesimin bellek üzerinde bizim belirlediğimiz fiziksel adresten başlatılmasını sağlar[110].

c) Sınıf (Class) tipi:

PUBLIC olarak tanımlanmış aynı sınıf tipine sahip kesimlerin Link işlemi sırasında birbirinin peşi sıra yerleştirilmesinde kullanılır. Bu değer her zaman tırnak (') içinde yazılır.

---

[108] Yazılan assembly programın EXE veya COM şeklinde düzenlenmesine bağlı olarak kullanılan kesim sayıları ve kullanım şekillerinde değişiklikler olmaktadır.

[109] Kullanılacak hizalama tipinin ne olacağını tespit etmek için ilgili yüksek seviyeli dil derleyicinin kullanım kitaplarına bakmak gereklidir.

[110] Kesimin başlaması istenen yerin 20 bit'lik fiziksel adresi ##### şeklinde verilmelidir.

Örnek:

Stacksg SEGMENT PARA **STACK** '*STACK*' ..... *Stacksg* ENDS	*Stacksg* ismiyle tanımlanmış olan bu kesim kullanılan birleştirme tipinin **STACK** olması nedeniyle yığın düzeninde kullanılacaktır.
Aticode SEGMENT PARA '*CODE*' ..... *Aticode* ENDS	Kesim tanımı, kesimin ne amaçla kullanılacağını (yığın hariç) belli eden bir ipucu içermez. Dolayısıyla örneğimizdeki kesim tanımı program kodunu tutmak üzere tanımlanıp CS yazmacı ile adreslenecek bir kod kesimi olabileceği gibi DS veya ES yazmaçları ile gösterilecek ve programın üzerinde işlem yapacağı veriyi barındıracak kesim de olabilir. Bu nedenle **CLASS** tipine mümkün olduğunca kesimin kullanım amacını belirleyecek bir isim verilmesi anlaşılırlığı arttırmak için faydalı olacaktır.
Memo SEGMENT BYTE **AT 012A0H** ..... *Memo* ENDS	**012A0H** fiziksel adresinden başlamak üzere 64 KB boyutunda yer kaplayacak bir kesim tanımı yapılmıştır.
Aticode SEGMENT PARA **PUBLIC** '*CODE*' ..... *Aticode* ENDS	**PUBLIC** kullanılması Linker'a bu kesimin başka bir kesim ile birleştirilerek diğerinin devamı olacak biçimde belleğe yerleştirilmesi gerektiğini bildirir.

ORG (Origine)

Açıklama:
COM tipi assembly programlarda tasarım gereği kodun 100H adresinden başlaması gerekmektedir[111]. Bunu sağlamak üzere sadece COM tipindeki programların **SEGMENT** tanımı içinde ORG pseudo komutu kullanılır.

Örnek:

ORG 100H	COM tipindeki assembly programın 100H adresinden başlamasını sağlar.

ASSUME

Açıklama:
Daha öncede belirtildiği gibi SS **yığın**, CS **kod**, DS **veri** kesimlerinin başlangıç adresini göstermek için kullanılan kesim yazmaçlarıdır. Genellikle bir program içinde birden fazla **SEGMENT** tanımı yapılabilir. Derleyicinin, kesimlere verilen isimlerden ve sınıf tiplerinden (class type) ne amaçla tanımlandıklarını ve kullanılacaklarını anlaması beklenemez. Bu nedenle programımızda **SEGMENT** sözde komutu ile tanımladığımız kesimlerimizin ne amaçla kullanılacaklarının ve nereden başladıklarının belirtilmesi için kod kesimi içinde **ASSUME** sözde komutunu kullanarak gerekli tanımlamaları yaparız.

Örnek:

ASSUME SS:*SS_adi*,DS:*DS_adi*,CS:*CS_adi*	EXE tipindeki programlarda birden fazla kesim kullanılacağı için her kesim yazmacının hangi kesim tanımı ile ilintili olduğu belirlenmektedir[112].
ASSUME SS:*S_adi*,DS:*S_adii*,CS:*S_adi*	COM tipindeki programlar tek kesimden oluşmaktadır. Bu nedenle program içinde tek SEGMENT tanımı yer alacak ve tüm kesim yazmaçları aynı kesimi gösterecek şekilde ayarlanacaktır.

[111] Detaylar, 9.2. bölümde COM Yapısındaki Assembly Programların Genel Özellikleri başlığı altında incelenmiştir.

[112] Detaylar, 9.1. bölümde EXE Tipindeki Assembly Programların Genel Özellikleri başlığı altında incelenmiştir. EXE tipindeki assembly programların dönüş kesim adresi tasarımın bir gereği olarak, DS yazmacı üzerinde saklanmaktadır. Buna göre ASSUME sözde komutu yardımıyla DS yazmacı kullanıcı tanımlı bir kesim ile ilişkilendirilmiş bile olsa DS yazmacının kullanıcı tarafından tanımlanan veri alanına erişebilmesi için ana yordam içinde yapılması gereken bazı ek işlemler olacaktır.

8.1.3 Veri Tanımlamada Kullanılan Sözde Komutlar

Veri kesimi içerisinde bellek alanı ayırmak ve değişken/veri tanımlamak amacıyla kullanılan sözde komutlar bu grup altında toplanmıştır. Assembly program yazılırken yüksek seviyeli dillerde olduğu gibi farklı veri tipleri kullanılabilir. Verinin hangi tipte olması gerektiğine programcı karar vererek sözde komutları kullanıp uygun tanımları yapacaktır.

DB (Define Byte)

Açıklama:

DB sözde komutu veri kesimi içerisinde bir byte bellek alanı ayırmak ve değişken/veri tanımlamak amacıyla kullanılır. Bir byte ile ifade edilen sayılar **işaretli** veya **işaretsiz** olabilir. Byte tanımlı bir değişkenin alabileceği değerler aşağıda verilmiştir.

İşaretsiz sayı olarak: 00-0FFH aralığı
İşaretli sayı olarak: 00-7FH aralığı pozitif sayılar
80-0FFH aralığı negatif sayılar

Örnek:

Sayıb1 **DB** 25	*Sayıb1* isimli değişkene 25 değeri verilmiştir.
Sayıb2 **DB** 1EH	*Sayıb2* isimli değişkene 1EH değeri verilmiştir.
Sayıb3 **DB** ?	*Sayıb3* isimli değişken bellekte kendine ayrılan alanda o anda bulunan değeri alır.
Dizib1 **DB** 1, 2, 3FH, 0A4H, 5	*Dizib1* 5 tane byte elemanı olan bir dizi gibi görünmektedir.
Strb **DB** 'Assembly Dersi'	ASCII karakterlerden oluşmuş *Strb* isimli dizi tanımlanmıştır.

DW (Define Word)

Açıklama:

DW sözde komutu veri kesimi içerisinde 2 byte (word) bellek alanı ayırmak ve değişken/veri tanımlamak amacıyla kullanılır. Bir word ile ifade edilen sayılar **işaretli** veya **işaretsiz** olabilir. Word tanımlı bir değişkenin alabileceği değerler aşağıda verilmiştir..

İşaretsiz sayı olarak, 0000H-0FFFFH aralığı
İşaretli sayı olarak, 0000H-7FFFH aralığı pozitif sayılar
8000H-0FFFFH aralığı negatif sayılar

Bilgisayarın belleği byte olarak düzenlenmiştir. Bu nedenle **DW** ile tanımlanmış olan değişkenler belleğe yerleştirilirken önce düşük anlamlı (low) byte düşük adresli, daha sonra yüksek anlamlı (high) byte yüksek adresli belleğe yerleştirilecektir.

Örnek:

Sayıw1 **DW** 693FH	7 0 0251H .. 0252H 3FH ← *Sayıw1* 0253H 69H	*Sayıw1* isimli değişken 0252H ve 0253H adreslerini işgal etmektedir. Düşük anlamlı byte olan 3FH değeri 0252H adresli bellek alanına yerleştirilirken yüksek anlamlı byte olan 69H ise 0253H adresine yerleşir.
Diziw **DW** 5, 10, 20	7 0 0510H 05H ← *Diziw* 0511H 00H 0512H 0AH 0513H 00H 0514H 14H 0515H 00H	0510H adresinden başlayarak her word değerinin düşük anlamlı byte'ını küçük adrese, yüksek anlamlı byte'ı büyük adrese gelecek şekilde yerleştiğinde *Diziw* değişkeni bellekte yanda görüldüğü gibi yerleşmiş olur.

DD (Define Doubleword)

Açıklama:
DD sözde komutu veri kesimi içerisinde 4 byte (dword) bellek alanı ayırmak ve değişken/veri tanımlamak amacıyla kullanılır. Bir dword ile ifade edilen sayılar **işaretli** veya **işaretsiz** olabilir. Dword tanımlı bir değişkenin alabileceği değerler aşağıda verilmiştir..

İşaretsiz sayı olarak,	00000000H-0FFFFFFFFH	aralığı
İşaretli sayı olarak,	00000000H-7FFFFFFFH	aralığı pozitif sayılar
	80000000H-0FFFFFFFFH	aralığı negatif sayılar

Bilgisayarın belleği byte olarak düzenlenmiştir. Bu nedenle **DD** ile tanımlanmış olan değişkenler belleğe yerleştirilirken önce düşük anlamlı (low) word düşük adresli, daha sonra yüksek anlamlı (high) word yüksek adresli belleğe yerleştirilecektir. Ancak her word, **DW** tanımına uygun olarak önce düşük anlamlı (low) byte düşük adresli, daha sonra yüksek anlamlı (high) byte yüksek adresli belleğe yerleştirilir **DD** ile tanımlanmış bir değişkeni basit bir byte dizisi olarak düşünecek olursak byte'ların belleğe ters sırada yerleştirildiği görülecektir.

Örnek:

Sayıdd **DD** 12897F3CH

	7 … 0	
0251H	3CH	← *Sayıdd*
0252H	7FH	
0253H	89H	
0254H	12H	
0255H	....	

Sayıdd değişkeni belleğe yerleştirildiğinde yandaki bellek haritasında görünen sonuç elde edilmiş olur. Byte'ların düşük adresten büyük adrese doğru ters sırada dizildiğine dikkat ediniz.

DQ (Define Quadword)

Açıklama:
DQ sözde komutu veri kesimi içerisinde 8 byte (Qword) bellek alanı ayırmak ve değişken/veri tanımlamak amacıyla kullanılır Bir qword ile ifade edilen sayılar **işaretli** veya **işaretsiz** olabilir. Bu sözde komut, 64 bit çift duyarlı kayan noktalı ve tam sayı değişkenleri tanımlamak üzere kullanılır. **DQ**'yu oluşturan byte'lar **DD** ve **DW** tanımlarında olduğu gibi bellek üzerinde ters sırada yer alacaktır.

DT (Define Ten Bytes)

Açıklama:
DT sözde komutu veri kesimi içerisinde 10 byte bellek alanı ayırmak ve değişken/veri tanımlamak amacıyla kullanılır. Bu on byte ile ifade edilen sayılar **işaretli** veya **işaretsiz** olabilir. Özellikle 80x87 (math coprocessor) ailesi tarafından ihtiyaç duyulan 10 byte'lık BCD değerleri ve genişletilmiş duyarlılığa sahip (80 bit) kayan noktalı değerleri ifade edebilmek için kullanılır.[113]

EQU (Equal)

Açıklama:
EQU sözde komutu bir değişken tanımlamaz, program içinde hiçbir şekilde **bellek işgal etmez**. Ancak tanımladığı değer başka komutlar içinde yer alan **bir sabit gibi değerlendirilir**. Bu değerlendirme ise derleyici tarafından yapılmaktadır.

Örnek:

Eleman **EQU** 100 *Dizi* DB *Eleman* DUP (0)	*Dizi* tanımı içinde *Eleman* sabiti değer olarak kullanılmıştır. Bu tanım uyarınca 100 tane byte elemanlı bir dizi tanımı yapılmıştır.

[113] DT sözde komutu kullanılan assembly dili derleyicisi ve sürümüne bağlı olarak farklı yorumlanabilmektedir. Bu nedenle kullanmadan önce denenmesinde fayda vardır.

DUP (Duplication Factor)

Açıklama:
Veri tanımlamalarında **DUP** sözde komutu kendinden sonra parantez içinde verilen değeri kendinden önce belirlenen rakam kadar tekrarlamak üzere kullanılır. Dizi tanımlamalarında ilk değer atanmasını yapmak veya bellek üzerinde belirli bir miktarda yer ayırmak üzere kullanılır. Veri tanımlamada **DB**, **DW**, **DD**, **DQ** ve **DT** sözde komutlarıyla birlikte kullanılır.

Örnek:

Dizib2 DB 15 **DUP** (0)	15 byte'lık alan ayrılmış ve ilk değerleri 0 ile doldurulmuştur.
Mb1 DB 3 **DUP** (4 **DUP**(8))	3x4 byte uzunluğunda bir alan ayrılmış ve ilk değerleri 8 ile doldurulmuştur.
Diziw DW 10 **DUP** (?)	10 word elemanlı dizinin ilk değerleri verilmemiştir. '?' sembolü o anda bellekte olan değerlerin olduğu gibi kabul edileceği anlamına gelmektedir.

REAL4, REAL8, REAL10

Açıklama:
Bu sözde komutlar, **DD**, **DQ** ve **DT**'ye benzer olarak sırası ile bellekte 4, 8 ve 10 byte'lık yer işgal eder ve kayan noktalı değişkenlerin tanımlanmasına imkân sağlar. Kullanımları diğer tanımlama komutlarında olduğu gibidir. Kayan noktalı işlemleri daha hızlı yapabilmek için ana işlemci ile uyumlu bir matematik yardımcı işlemci bulunmalıdır.
MASM, **DD**, **DQ** ve **DT** ile kayan noktalı değişkenler tanımlanmasına izin vermektedir. Zira bunlar bellek üzerinde gerekli büyüklükte alan ayrılması ve gerekli olması durumunda ilk değer atamasını yapmaktan başka bir amaç için kullanılmazlar. Ancak bu şekilde yapılan kayan noktalı tanımlamaların bazı mahsurları da vardır. Bunların ilki CodeView programının sadece **REAL4**, **REAL8** ve **REAL10** ile tanımlanan değişkenlerin değerlerini doğru gösterebilmesidir. Diğeri ise **DD**, **DQ** ve **DT** ile hem tam hem kayan noktalı sayı tanımının yapılabilmesidir. Eğer yanlışlıkla 1.0 yazılması gereken bir bellek gözüne ilk değer olarak 1 yazılmış ve bu değişken kesirli sayı işlemlerinde kullanılmış ise, tam sayı ve kayan noktalı sayıların ifade farklılıklarından kaynaklı olarak işlem hatası oluşacaktır. Bu mahsurlardan ötürü kesirli sayılar ile işlem yapılacağı zaman mutlaka **REAL4**, **REAL8** veya **REAL10** tanımları kullanılmalıdır[114].

Örnek:

Sayı4 **REAL4** 1.5 *Sayı8* **REAL8** 0e–25 *Sayı10* **REAL10** -1.2594e+10	Değişkenler tanımlanırken kayan noktalı oldukları unutulmamalı ve kesri belli eden nokta kullanılarak değerler verilmelidir.
Hatalı4 **REAL4** 1	Burada yapılan tanım hatalıdır. Değer 1.0 şeklinde verilmiş olmalıdır.

TYPEDEF

Açıklama:
Assembly dilinde program yazarken C veya Pascal gibi yüksek seviyeli programlama dillerinde kullandığınız veri tiplerine benzer tanımlamaları yapmanız da mümkündür. **TYPEDEF** sözde komutu ile MASM 6.x sürümünden itibaren veri tiplerinizi istediğiniz gibi veya daha önce kullanmaya alışkın olduğunuz herhangi bir yüksek seviyeli programlama dilininkine benzer biçimde tanımlayabilirsiniz.

Örnek:

integer **TYPEDEF** WORD *char* **TYPEDEF** BYTE *boolean* **TYPEDEF** BYTE *float* **TYPEDEF** REAL4 *colors* **TYPEDEF** BYTE	Yandaki tanımlamalar yapıldıktan sonra değişkenlerinizi yüksek seviyeli dillerde olduğu şekli ile tanımlayabilirsiniz. i integer ? ch char ? Var boolean ? x float ? Delta colors ?

[114] Bu kitap işlemcinin kesirli sayılar ile çalışmasını açıklamak üzere kaleme alınmamış olup bu tanımlamalar da sadece bilgi amaçlı verilmiştir.

PTR (Pointer)

Açıklama:

Herhangi bir bellek alanına tanımlandığı veri tipinden (byte, word, dword) bağımsız olarak erişilmek isteniyorsa **PTR** sözde komutundan yararlanılır. Böylece yüksek seviyeli dillerde olduğu gibi veriye tip değişimi ile erişme (casting) assembly dilinde de mümkün olmaktadır. Ayrıca PTR sözde komutu **CALL** ve **JMP** komutları ile birlikte de kullanılabilir.

Örnek:

Sayıb DB 15H DB 25H *Sayıw* DW 3545H MOV AX, WORD **PTR** *Sayıb* MOV AH, BYTE **PTR** *Sayıw* MOV AL, BYTE **PTR** *Sayıw*+1	7 … 0 0001H [15H] ←*Sayıb* 0002H [25H] 0003H [45H] ←*Sayıw* 0004H [35H] Tanımlamaların sonucunda bellek üzerinde sayıların yerleşimi yandaki bellek haritasında görüldüğü şekilde olacaktır. *Sayıb* byte tanımlı olmasına rağmen word olarak erişilmek istenmektedir. Sonuçta: AX←2515H olacaktır. *Sayıw* değişkenine byte olarak erişilmek istenmektedir. Sonuçta: AH←45H olacaktır. *Sayıw+1* ile belirlenen yerden bir byte alınarak AL←35H olacaktır.
CALL FAR **PTR** *Toplama*	*Toplama* isimli yordam farklı bir kesimdedir. Bu nedenle FAR CALL ile yordam çağırılmaktadır. CALL komutunu takip eden komut satırını gösteren **IP** ve **CS** yazmaç değerleri dönüş değeri olarak yığında saklanır ve *Toplama* isimli yordamının **RETF** komutunun çalıştırılması ile bu değerler yığından alınarak geri dönülür.
CALL NEAR **PTR** *Cikarma*	*Cikarma* yordamı aynı kesim içerisinde olmakla birlikte farklı bir göreli konumdadır. Bu durumda yığında sadece **IP** yazmaç değeri dönüş adresi olarak saklanır ve *Cikarma* yordamının en sonundaki **RET** komutu işlendikten sonra geri dönülür.

LABEL

Açıklama:

PTR sözde komutu yardımıyla veri kesimi içinde tanımladığımız bazı değişkenlere tanımlandıkları tipten farklı bir tip ile erişebilirken **LABEL** sözde komutu da veri kesiminde tanımlı değişkenlere farkı veri tipileri ile erişebilmek için yeni etiketler tanımlar.

Örnek:

Yenib **LABEL** BYTE
Sayıw DW 2535H
Yeniw **LABEL** WORD
Sayıb DB 45H
Sayıc DB 55H
 	
MOV AX, *Yeniw*
MOV BL, *Yenib*
	7 … 0
	0000H [35H] ← *Sayıw* ←*Yenib*
	0001H [25H]
	0002H [45H] ←*Sayıb* ←*Yeniw*
	0003H [55H] ←*Sayıc*
	0004H [..]
	Tanımlanan değişkenlere ilişkin değerler yandaki bellek haritasında görüldüğü gibi yerleşecektir. *Yeniw* isimli LABEL'ı kullanarak byte tipinde tanımlanmış *Sayıb* değişkenine word olarak erişme hakkı elde edilmiştir. Buna göre işlemin sonucunda AX←5545H olacaktır. Dikkat edilecek olursa burada *Sayıc* değişkeni de işin içine ister istemez dâhil olmuştur.
	Yenib isimli LABEL'ı kullanarak word olarak tanımlanmış *Sayıw* değişkenine byte olarak erişme hakkı elde edilmiştir. Buna göre işlemin sonucunda BL←35H olacaktır.

8.1.4 Yordam Düzenlemede Kullanılan Sözde Komutlar

Diğer programlama dillerinde olduğu gibi assembly dilinde de işlemler yordamlar yardımıyla gerçekleştirilir. Assembly programın kod kesimi içinde birden fazla

yordam olabilir. Ancak bunlardan sadece bir tanesi ana yordam[115] olarak tüm işleyişi kontrol eder, uygun parametre aktarma yöntemlerini kullanarak bulunduğu kesim içindeki (intrasegment) ve/veya farklı bir programın kod kesimi içindeki (intersegment) yordamları çağırarak işlemleri yaptırır. Genel olarak her yordam **RET/RETF** komutunun çalıştırılması ile kendisini çağıran noktaya geri döner[116].

PROC (Procedure) / ENDP (End of procedure)

Açıklama:

PROC tanımının peşinden yordamın tipini belirlemek üzere **FAR** (uzak) veya **NEAR** (yakın) sözde komutu gelmelidir. **FAR** tanımlı **PROC** dönüş adresi olarak yığına 2 byte kesim, 2 byte göreli konum, **NEAR** tanımlı **PROC** sadece 2 byte göreli konum değerini yığında dönüş adresi olarak saklar. Yordamlar, tanım tipine göre **RET** veya **RETF** komutu ile sonlanır[117]. Ancak en sonda yordam tanımının bittiğini gösteren **ENDP** sözde komutu kullanılmalıdır. Genel olarak yordam tanımı şu şekilde yapılır.

```
yordam-ismi     PROC { NEAR | FAR }
                    .....
                    ; komutlar....
                    ....
                { RET | RETF }
yordam-ismi     ENDP
```

EXTRN

Açıklama:

EXTRN sözde komutu referans verilen herhangi bir yordam veya verinin başka bir programa ait (kod veya veri) kesim içerisinde tanımlanmış olduğunu Link programına anlatmak üzere kullanılmaktadır. Bu sayede assembly derleyicisi referansın başka bir kesim içinde olduğunu anlar ve link işlemi sırasında gerekli adres aktarım işlemlerini yapar. Ancak **EXTRN** sözde komutunun kullanılabilmesi için kendine ait yordamı veya veriyi başkaları ile paylaşacak olan programda da **PUBLIC** tanımlamasının yapılması gereklidir.[118] EXTRN sözde komut, harici bir yordama referans vermek için kullanılmış ise FAR, harici bir değişkene referans vermek için kullanılmış ise o değişkenin tipini belirleyen tanım (byte, word, dword gibi) ile birlikte kullanılmalıdır.

Örnek:

EXTRN *Myproc*:FAR, *MyStr*:Byte	*Myproc* isimli farklı kod kesiminde bulunan yordamın çağırılacağı ve yine başka bir veri kesiminde byte olarak tanımlanmış, *MyStr* isimli bir değişkenin kullanılacağı derleyiciye belirtilmektedir.
EXTRN *Mydata*:Byte, *Carpan*:Word	Byte tipinde *Mydata* ve word tipinde *Carpan* isminde iki değişkene referans verileceği (kullanılacağı) derleyiciye belirtilmektedir.

---

[115] Modüler uygulama geliştirmenin gereği olarak karmaşık işlemler, daha kolay yönetilebilen, giriş ve çıkış değerleri iyi tarif edilmiş küçük program parçalarına ayırılmaktadır. Bu durumda kod alanı içinde birden fazla yordam oluşur. Derleyicinin bu yordamlardan hangisinin işleri diğer yordamlara yaptırarak akışı kontrol altında tutan ana yordam olacağını yordam ismine, tanım şekline veya kesim içindeki sırasına/konumuna bakarak belirlemesi mümkün değildir. Bu problem, kontrolü elinde bulunduracak olan ana yordamın isminin END sözde komutundan sonra yazılması ile çözülmektedir.

[116] EXE ve COM tipindeki programların yapısal farklılıklarından ötürü ana yordamın dönüş adres(ler)ini saklanmasında farklılıklar vardır.

[117] Yordamlar bazı durumlarda INT komutu kullanılarak da sonlandırılabilir. Ana yordamların sonlanmasını sağlamak üzere yığına saklanan geri dönüş adresleri aslinda DOS'un 20H (Terminate Program) numaralı kesmesini çalıştırmaktadır. Yardımcı yordamların INT 20H, INT 21H (fonksiyon 00H veya 4CH) ile sondandırılması halinde kontrol doğrudan işletim sistemine döner.

[118] Bir kod içerisinde EXTRN kullanılmış ise yine başka bir kod içerisinde bunun karşılığı olan bir PUBLIC komutunun da bulunması gereklidir. EXTRN ve PUBLIC sözde komutlarının vaad ettikleri işi yapabilmeleri için ilgili OBJ uzantılı dosyaların Link programında birlikte işlenmesi gerekmektedir.

PUBLIC

Açıklama:
PUBLIC sözde komutu birbirleri ile bağlantılı olarak kullanılacak olan programlarda, ortaklaşa kullanılacak yordamları ve veriyi tanımlamak amacıyla **EXTRN** sözde komutu ile karşılıklı olarak kullanılır. EXTRN sözde komutu program içindeki bazı referansların farklı yerlerde tanımlı olduğunu belirlemek için kullanılırken, PUBLIC sözde komutu ise kullanıldığı program içinde bazı yordam ve/veya değişkenlere dışarıdan da erişilebileceğini belirlemek amacıyla kullanılır. EXTRN ve PUBLIC tanımları link programına gerekli adres aktarımlarını yapabilmesi için gerekli referansları sağlamakta kullanılır. **EXTRN** ve **PUBLIC** sözde komutları kullanılmaksızın hiçbir program kendi kontrolündeki kesimlerde tanımlı olmayan yordamlara veya veriye isimlerini kullanarak erişemez.

Örnek:

PUBLIC *Mydata, MyProc*	*Mydata* ve *MyProc* isimli etiketlere başka bir program tarafından erişilebileceğini belirten tanımdır.

8.1.5 Makro Düzenlemede Kullanılan Sözde Komutlar

Modüler programlamanın bir gereği olarak, assembly programlarda yordamlara ek olarak makrolardan da yaygın şekilde yararlanılır. Yordamlar, çağırılma (CALL) ve geri dönme (RET/RETF) esasına göre çalışırken makrolarda çağırma ve geri dönüş kavramları **kullanılmaz**. Hal böyle olunca makroyu kullanmak için sadece makronun isminin yazılması yeterli olduğu gibi makronun sonunda RET/RETF komutları da kesinlikle yer almayacaktır. Makro, kullanıldığı (isminin yazıldığı) yerden önce mutlaka tanımlanmış veya programa bir şekilde dâhil edilmiş olmalıdır. Yordam ve makroların işleyiş, parametre aktarım ve kullanımlarında büyük farklılıklar vardır. Bunlar, 10.3. bölümde Yordam ve Makro Karşılaştırılması başlığı altında ayrıntılı biçimde açıklanacaktır.

MACRO (Macro) / ENDM (End of macro)

Açıklama:
Makrolar, **MACRO** sözde komutu kullanılarak tanımlanırlar ve eğer parametre kabul ediliyor ise bunlar virgül ile birbirlerinden ayrılmak kaydı ile MACRO sözde komutunu takip ederek yazılır.

```
macro_ismi      MACRO { parametre1, { parametre2, { parametre n } } }
                    .....
                    ; işlemler
                    ......
                ENDM
```

Her MACRO sözde komutu makronun başladığını belirtirken **ENDM** (end of macro) sözde komutu da sona erdiğini belirtmek için kullanılır. **ENDM** komutun başına *macro_ismi* yazılması gerekmez.

INCLUDE

Açıklama:
Makrolar kullanıldıkları yerden önce kod kesimi içerisinde tanımlanabileceği gibi metin tipinde ayrı bir dosyaya yazılarak programın başında koda dâhil edilebilirler. Bunu sağlamak üzere **INCLUDE** sözde kodundan yararlanılır. Bu sayede assembly dili derleyicisi makro kodunu bu dosyadan alarak kod içinde makro adının geçtiği noktaya ekler. Birden fazla dosyanın INCLUDE edilmesi durumunda makrolar sırası ile verilen dosyalar içinde aranacaktır.

Örnek:

INCLUDE *DENEME.LIB*	*DENEME.LIB* isimli dosya programa dâhil edilir. Program içinde kullanılacak olan makrolar bu dosyanın içinde bulunmalıdır.

LOCAL

Açıklama:
Hem makro içinde hem de makronun dâhil olduğu program içinde yapılacak dallanmalarda kullanılmak üzere benzer etiketlerin kullanılması probleme neden olur. Bu nedenle makro içerisinde yerel etiket (label) kullanılacak ise bunun **LOCAL** sözde komutu ile derleyiciye tanıtılması gerekmektedir.

Örnek:

Dene MACRO { parametreler } **LOCAL** *atla1, atla2* ASSUME *CS: ..,DS: ...,SS* .. *atla1*: JMP *atla2* *..; işlemler* *atla2:* .. ENDM	*Dene* isimli makro içinde kullanılan *atla1* ve *atla2* isimli etiketlerin makronun kullanıldığı program içinde tekrarlanması durumunda problem yaratmaması için **LOCAL** olarak tanımlanmıştır.

EXITM

Açıklama:
EXITM komutu ile karşılaşıldığında işleme otomatik olarak **ENDM** sözde komutundan sonraki komut ile devam edilir.

Örnek:

Sola_it MACRO *num*, *val*
IFE *val*
EXITM
ENDIF
PUSH CX
MOV CL, val
SHL num, CL
POP CX
ENDM
	Örneğimizdeki *Sola_it* makrosu, *num* ve *val* olarak adlandırılmış iki parametre almaktadır. **IFE** koşullu sözde komutu ile *val* isimli değişkenin değeri kontrol edilir, değerin sıfır olması durumunda **EXITM** çalıştırılarak **ENDM** sözde komutunu takip eden komut ile işlemlere devam edilir. Ancak koşul sağlanmadı ise **ENDIF** koşullu sözde komutundan sonraki satırlar işletilecektir. Burada *num* değişkeni ile verilen değerin *val* ile belirlenen sayı kadar sola ötelenmesi amaçlanmıştır. Öteleme sayısı CL yazmacı üzerinden SHL komutuna verilecek olup işlem öncesinde PUSH, sonrasında da POP komutları ile CX'in önceki değeri korunmuştur.

8.1.6 Diğer Sözde Komutlar

Dosya düzenlemesi, kesim düzenlemesi, veri tanımlamaları, yordam düzenlemesi ve makro düzenlemesi için kullanılan sözde komut gruplarında yer veremediğimiz ancak kullanıcıya kolaylıklar sağlayan diğer sözde komutlar bu grup altında toplanmıştır.

LENGTH

Açıklama:
DUP sözde komutu ile tanımlanmış değişkenlerde, **LENGTH** sözde komutu, **DUP**'dan önceki sayısal değeri getirir, aksi halde 1 değeri dönecektir.

Örnek:

Tablo DW 15 DUP (0) MOV AX, **LENGTH** *Tablo*	Yapılan tanım uyarınca AX←000FH olacaktır.
Sayı DB DUP (0) MOV AX, **LENGTH** *Sayı*	Yapılan tanım uyarınca AX←0001H olacaktır.

TYPE

Açıklama:

TYPE sözde komutu değişkenin kaç byte'lık bellek alanı kapladığını bulmak için kullanılmaktadır. Buna göre;

DB ise 1
DW ise 2
DD ise 4
DQ ise 8
DT ise 10 değeri dönecektir.

Örnek:

Tablo DW 15 DUP (0) MOV AX, **TYPE** *Tablo*	Yapılan tanım uyarınca AX←0002H olacaktır.
Sayı DD 3412ABCFH MOV AX, **TYPE** *Sayı*	Yapılan tanım uyarınca AX←0004H olacaktır.

SIZE

Açıklama:

SIZE sözde komutu **LENGTH*TYPE** sonucunda oluşan değeri verir.

Örnek:

Tablo DW 20 DUP (?) MOV AX, **SIZE** *Tablo*	Yapılan tanım uyarınca AX←0028H olacaktır. (20*2 = 40 olur. Bunun on altılı sayı sistemindeki karşılığı verilmiştir.)

OFFSET

Açıklama:

OFFSET sözde komutu bir değişkenin tanımlı olduğu kesimin başından itibaren kaçıncı byte'ta olduğunu gösteren değeri (adresi) döndürür. Bu göreli konum değeri 16 bitlik bir değer olduğu için **OFFSET** sözde komutunun varış işleneni mutlaka word tipinde olmak durumundadır. Ürettiği sonuç **LEA** komutu ile benzerlik göstermektedir. Ancak **LEA** program çalışırken adres değerinin alınmasını sağlarken, **OFFSET** sözde komutu derleme/link işlemi sırasında değerini almaktadır.

Örnek:

Mov DX, **OFFSET** *Tablo*	7 0 0251H 3CH ← *Tablo* 0252H 7FH 0253H 89H	Tablo isimli değişkenin adresi (EA) ilgili yamaca aktarılacak, sonuç olarak DX←0251H olacaktır.

SEG (Segment)

Açıklama:

SEG sözde komutu bir etiketin içinde bulunduğu kesim adresini döndürür. Kesim değeri 16 bit ile ifade edilen bir değer olması sebebiyle 16 bit'lik bir işlenene aktarılmalıdır.

Örnek:

Mov ES, SEG *Sayıw*	*Sayıw* isimli değişkenin kesim adresi ES yazmacına konur. *Sayıw* , veri kesiminde tanımlı ise ES←DS olmuş olacaktır.
Mov AX, SEG *Arama*	*Arama* isimli bir yordamın CS içerisinde tanımlı olduğunu düşünecek olursak AX←CS işlemi gerçekleştirilmiş olacaktır.

REPT (Repetition)

Açıklama:
Ön eklerin yaptığı işlemi yapmaya sağlayan bir sözde komuttur. **REPT** ile **ENDM** sözde komutları arasında kalan komutlar **REPT** sözde komutunu takiben yazılan değer kadar tekrarlanır.

Örnek:

REPT 5 MOVSB **ENDM**	**REPT** ile **ENDM** arasında yer alan komut bloğu 5 defa tekrarlanacaktır. Bu komutun eşdeğer karşılığı olarak aşağıdaki kod parçaları da kullanılabilir. Tek işlemli blok için MOV CX,5 REP MOVSB Çok işlemli blok için MOV CX,5 L1: MOVSB ; tekrarlanacak blok LOOP L1

IRP (Indefinite Repeat)

Açıklama:
Genel kullanımı;

IRP *degisken*, < *argüman(lar)* >

şeklinde olup '**<**' ve '**>**' işaretleri arasında verilen değerler *degisken*'e sırası ile verilir. **REPT** komutuna benzer olarak işlem **ENDM** sözde komutunu görünceye kadar tekrarlanacaktır.

Örnek:

IRP N, < 2, 4, 6 > DB N **ENDM**	Bu komuta karşılık assembler sırası ile DB 2, DB 4, DB 6 komutlarını yaratmış olur.

IRPC (Indefinite Repeat Character)

Açıklama:
IRP komutuna benzer bir özellik göstermesine rağmen argümanlar bir dizgi şeklinde verilmektedir.
Genel kullanımı;

IRPC *degisken, dizgi*

şeklindedir. Dizgideki her karakter için ayrı işlem yapılacaktır.

Örnek:

IRPC N, 246 DB N **ENDM**	Bu komuta karşılık assembler sırası ile DB 2, DB 4, DB 6 komutlarını yaratmış olur.

END

Açıklama:
Ana programın bittiğini belirlemek amacıyla kullanılan, assembly programın en son sözde komutudur. Her program, modüler programlamanın bir gereği olarak birden fazla yordamdan oluşabilir. Bu yordamlardan biri program başlatıldığında kontrolü elinde bulunduran ana yordam, diğerleri ise bunun tarafından çağırılan ve yardımcı olarak nitelenebilecek yordamlardır. Dolayısıyla program başladığında **ilk çalıştırılacak yordamın hangisi olduğunun belirlenmesi** gereklidir. Bu nedenle **END** sözde komutundan sonra yazılacak yordam ismi, program yüklendiğinde ilk olarak çalıştırılacak olan yordamı belirlemektedir.

Örnek:

END *Baslangic_Noktasi*	Program yüklendikten sonra kontrol *Baslangic_Noktasi* isimli yordama verilecektir.

8.2 Koşullu Sözde Komutlar

Özel durumların kontrol edilerek bir takım işlemlerin yapılması için genel ifadesi aşağıda verilen koşullu sözde komutlardan yararlanılır.

IFxx (koşul)
... ;işlem bloğu
ELSE
... ;işlem bloğu
ENDIF

Her **IFxx** için bir **ENDIF** bulunmalıdır. Koşulun sağlanması durumunda **ELSE** satırına kadar olan, aksi halde **ELSE**'i takip eden işlemler **ENDIF** satırına kadar yürütülür. Null-Else olması halinde ELSE ifadesi kullanılmaz.

IF

Açıklama:
Koşul ifadesi sıfırdan farklı bir değer almış ise takip eden işlemleri gerçekleştirir.

Örnek:

IF *debug* CALL *routine1* **ENDIF**	*debug* sıfırdan farklı bir değer almış ise *routine1* çağırılmaktadır.

IFE

Açıklama:
Koşul ifadesi sıfır değerini almış ise takip eden işlemleri gerçekleştirir.

IF1

Açıklama:
İki geçişli derleme (2 pass compiler) işlemlerinde **IF1** sözde komutunu takip eden işlemler ilk geçişte devreye girer.

IF2

Açıklama:
İki geçişli derleme (2 pass compiler) işlemlerinde **IF2** sözde komutunu takip eden işlemler ikinci geçişte devreye girer.

IFDEF

Açıklama:
Bir sembol program içinde veya **EXTRN** olarak erişilebilir bicimde tanımlanmış ise takip eden işlemler gerçekleştirilir.

Örnek:

IFDEF *Sonuc*
MOV *Sonuc, DX*
ELSE
MOV AX, DX
ENDIF
	Sonuc isimli sembol tanımlanmış ise DX yazmacındaki değer *Sonuc* değişkenine atanmakta, aksi halde DX yazmacındaki değer AX yazmacına yerleştirilmektedir.

IFNDEF

Açıklama:
Bir sembol program içinde veya **EXTRN** olarak erişilebilir bicimde tanımlanmamış ise takip eden işlemler gerçekleştirilir.

Örnek:

IFNDEF *Eleman* *Eleman* EQU *100* **ENDIF**	*Eleman* isimli sembol tanımlanmamış ise bunu **EQU** sözde komutu ile tanımlamaktadır.

IFB

Açıklama:
IFB (blank) koşullu sözde komutu makroya yollanan argümanın boş olması durumunda kendisini takip eden komutların işletilmesini sağlar. Değeri kontrol edilecek argümanın "**<**" ve "**>**" işaretleri arasında verilmesi gereklidir.

Örnek:

Yaz **MACRO** *Mesaj*
IFB <*Mesaj***>**
; Mesaj boş ise yapılacak
; işlemler
ELSE
; Mesaj dolu ise yapılacak
; işlemler
ENDIF
ENDM
	Yaz isimli makroya yollanan *Mesaj* isimli argüman boş olarak iletilmiş ise **IFB** koşullu sözde komutu ile bu durum belirlenerek farklı işlemler yapılmaktadır.

IFNB

Açıklama:
IFNB (not blank) koşullu sözde komutu da **IFB** (blank) koşullu sözde komutuna benzer özellik göstermek ile birlikte makroya yollanan argüman dolu olması durumunda kendisini takip eden komutların işletilmesini sağlar.

IFIDN

Açıklama:
IFIDN koşullu sözde komutu makroya yollanan argümanları küçük büyük harf duyarlı olarak karşılaştırır ve aynı olmaları durumunda kendisini takip eden komutların işletilmesini sağlar. Karşılaştırma işleminde argümanların "**<**" ve "**>**" işaretleri arasına yazılması gereklidir.

Örnek:

Test **MACRO** *arg1, arg2*
IFIDN <*arg1*>,<*arg2*>
; *arg1*=*arg2* ise yapılacak
ELSE
; *arg1*≠*arg2* ise yapılacak
ENDIF
ENDM
	Test isimli makroya yollanan *arg1* ve *arg2* isimli argümanların birbirinin eşi olup olmadığı **IFIDN** koşullu sözde komut ile tespit edilerek makro içinde koşula bağlı olarak farklı işlemler gerçekleştirilmektedir.

IFDIF

Açıklama:
IFDIF koşulu sözde komutu **IFIDN** koşulu sözde komutu gibi makroya yollanan argümanları küçük büyük harf duyarlı olarak karşılaştırır. Ancak argümanların farklı olmaları durumunda kendisini takip eden komutların işlenmesini sağlar.

9. ASSEMBLY PROGRAM TİPLERİ

Assembly dilinde programlar, genel kodlama tarzına, kesim tanımlamalarına ve kesimlerin düzenlerine bağlı olarak iki farklı tipte yazılabilirler. Bu tipler, üretilen çalıştırılabilir dosyaların uzantılarından adlarını almakta olup **EXE** ve **COM** olarak bilinirler. Her iki tip assembly programda da aynı komut seti kullanılmakla birlikte birbirlerinden oldukça farklı özelliklere sahiptirler.

İlerleyen bölümlerde EXE ve COM tipi programların genel özelliklerinden söz edilerek her tip için verilen örnekler detaylı olarak açıklanacak, EXE ve COM programların özellikleri bir karşılaştırma tablosu yardımıyla özetlenecektir.

9.1 EXE Tipindeki Assembly Programların Genel Özellikleri

EXE tipindeki programlar ayrık kesim düzeninde çalışmak üzere tasarlanmışlardır. Diğer deyişle bir EXE programın bellekte kendisine ait yığın, kod ve veri kesimleri (eğer tanımlanırsa ek kesim de) olabilir. EXE programın bellekte aynı anda erişebileceği kesim sayısı, kesim yazmaçlarının sayısı ile sınırlıdır. Ancak EXE programların boyutları kesim boyu ile sınırlı olmadığı için COM tipindeki programlara nazaran çok daha fazla bellek kullanırlar. EXE programlarda, ikincil bellek birimleri üzerinde saklanan kodlarının başında 512 byte büyüklüğünde EXE Header[119] olarak adlandırılan bir kontrol bloğu bulunmaktadır. Kesimlerin ayrı ayrı tanımlanabilmesinin bir sonucu olarak daha fazla veri üzerinde işlem yapabilen daha uzun assembly kodlarının yazılmasına olanak verir.

EXE programın iskelet yapısı Tablo 9-1'de verilmiştir. Program yazarken yaygın kullanılan kodlama tekniklerine uymak her zaman faydalıdır. Böylece rahat anlaşılır ve kolay takip edilebilir kodlar üretilebilir. Yazılan programa daha sonra bakıldığında yapılanların anlaşılabilmesi ise program içinde bulunan açıklama satırlarının sayısıyla orantılıdır. Programın ne amaçla yazıldığı, ne tür veri üzerinde işlem yaptığı, başka bir kesimde bulunan özel amaçlı kodlardan yararlanıyorsa bunların ne yaptıkları, kod içerisinde bir problemi çözmek için özel bir uygulama yapılmış ise bunun neden yapıldığı gibi bilgilerin açıklama satırları ile koda dâhil edilmesi faydalı olacaktır. Açıklama satırlarının derleyici tarafından kod olarak yorumlanmasını engellemek amacıyla "**;**" sembolünden yararlanılır. Bu sembolden sonra yazılacak olan her türlü metin açıklama satırı olarak değerlendirilecektir (Tablo 9-1. 14, 18, 21. satırlar).

---

[119] EXE header bloğu içinde programın EXE tipinde olduğunu belirten bir kod, kodun boyutu (ve 512 byte'lık kaç parçadan oluştuğu), kullanabileceği en az ve en çok paragraf büyüklükleri, SP ve IP yazmaç değerini belirleyen alanlar mevcuttur.

Tablo 9-1: EXE tipindeki programın iskelet yapısı

```
1                 PAGE      60,80
2                 TITLE     exe tipinde yazılmış bir assembly programın iskelet yapısı
3     Stacksg     SEGMENT PARA STACK 'STACK'
4                 DW        32 DUP (?)
5     Stacksg     ENDS
6     Datasg      SEGMENT PARA 'DATA '
7     sayi        DB ?
8     Eleman      DW ?
9     ...
10    Datasg      ENDS
11    Codesg      SEGMENT PARA 'CODE '
12                ASSUME CS:Codesg, DS:Datasg, SS:Stacksg
13    Basla       PROC FAR
14    ; Programın geri dönüş adreslerini yığın olarak tanımlanmış kesimde sakla
15                PUSH DS
16                XOR AX, AX
17                PUSH AX
18    ; Değişkenlerin tanımlandığı kesime erişmek için gereken işlemler
19                MOV AX, Datasg
20                MOV DS, AX
21    ; Programın esas kodu bu satırdan itibaren başlamaktadır
22                ..
23                ..
24                RETF[120]
25    Basla       ENDP
26    Codesg      ENDS
27                END       Basla
```

Programın derleyici tarafından derlendikten sonra bir çıkışının alınması gerekiyor ise bunun sayfa üzerinde nasıl basılması istendiği PAGE sözde komutu ile belirlenir. Bu komut tercihe bağlı olarak kullanılır (satır 1). Benzer biçimde derleyicinin ürettiği LST uzantılı dosyanın her sayfanın başında bir başlık satırı yazılması istenirse (satır 2) TITLE sözde komutundan yararlanılır.

Daha önce de vurguladığımız gibi EXE uzantılı programlarda ayrık kesim düzenini kullanılmaktadır. Buna göre yığın (satır 3), veri (satır 6) ve kod (satır 11) kesimleri ayrı ayrı tanımlanmıştır[121]. Tablo 9-1'de görülen iskelet yapıda *Stacksg* isimli kesim PARA tanımı uyarınca on altıya tam bölünebilir bir adresten başlayacak, STACK tanımının bir gereği olarak da yığın organizasyonunda kullanılacaktır (satır 3). Tanımlanan kesim içinde yığın için kullanılacak bellek alanı programımızın ihtiyaçları göz

[120] RETF yerine RET komutu yazılmış da olsa derleyici bunu PROC tanımının tipi ile uyumlu hale getirerek makine kodunu üretecektir.

[121] Eğer assembly dilinde yazılmış olan bir programın yüksek seviyeli dillerden herhangi biri ile yazılmış başka bir program ile birlikte çalışması gerekiyor ise assembly programın kesim tanımlarının yüksek seviyeli dilde kullanılan kesim tanımlamalarıyla uyumlu olması gerekir. Bu gibi durumlarda yüksek seviyeli dil derleyicilerinin ne tür kesim tanımları kullandıklarına ait bilgiler kullanıcı kitaplarından öğrenilmelidir.

önüne alınarak belirlenmelidir[122]. Yığın üzerinde yapılan işlemlerin word temelli olduğu düşünüldüğünde yığın boyutunun **DW** kullanılarak tanımlanması daha doğru olacaktır (satır 4). Her kesim kendi içinde bir blok olarak değerlendirilmektedir. Bu nedenle her kesim sonu **ENDS** sözde komutu ile belirlenmelidir (satır 5, 10, 26).

Yüksek seviyeli programlama dilleri kullanılarak geliştirilen uygulamalarda olduğu gibi, üzerinde işlem yapılacak değişkenlerin uygun veri tipleri ve uygun büyüklükler ile tanımlanması gereklidir. Değişken, ister dışarıdan bir okuma/yazma işlemi ile alınsın/verilsin[123], isterse veri alanı içinde ilk değer ataması yapılmış olsun, veri kesiminde uygun şekilde tanımlanmalıdır (satır 6–10).

Yüksek seviyeli programlama dillerinde olduğu gibi assembly dilinde yazılmış programlarda da en az bir yordam bulunmalıdır. Ancak programın ihtiyaçları, modüler olabilmesi, rahat takip edilebilir ve anlaşılabilir olmasını sağlamak üzere yapı bir ana yordam ve bunun tarafından çağırılarak kullanılan diğer yardımcı yordamlardan oluşacak şekilde düzenlenebilir. Buna göre programın, kod kesimi içinde (satır 11-26) en az bir **PROC** tanımı yapılmalıdır (satır 13). EXE tipindeki programlarda bir tane yordam var ise bu kontrolü elinde bulunduran temel parça olarak değerlendirilir ve **FAR** tipinde tanımlanır. Bir program içinde birden fazla –özellikle– FAR tanımlı yordam olması durumda hangisinin ana yordam olduğu nasıl belirlenecektir? Gerek bu problemi çözmek, gerek ise programın başlangıç noktasını belirlemek üzere assembly programın tipine, sahip olduğu yordam tiplerine ve sayılarına bakılmaksızın **END** sözde komutunu takiben ilk çalıştırılacak olan yordamın ismi yazılır (satır 27).

EXE tipindeki programlarda her kesimin ayrı ayrı tanımlanması nedeni ile bunlardan hangisinin hangi kesim yazmacı tarafından gösterileceğinin belirtilmesi gereklidir. **ASSUME** sözde komutu (satır 12) bu amaçla kullanılır. Örnek yapımızda CS yazmacı *Codesg*, DS yazmacı *Datasg* ve nihayet SS yazmacı *Stacksg* isimleri ile tanımlanmış kesimleri gösterecek şekilde ayarlanmıştır.

Yazdığımız program derlenip link edildikten sonra elde edilen EXE uzantılı dosyanın adı komut satırına yazılarak program çalıştırılır. Programın görevini yerine getirme-

---

[122] Yığın paylaşılan bir alan olup parametre aktarmak, dönüş adreslerini saklamak ve yerine göre herhangi bir yazmaç üzerinde bulunan değeri geçici olarak saklamak amacıyla kullanılır. Ayrıca bilgisayar sisteminin işlemesi için vazgeçilmez olan kesme mekanizması da (mevcut bayrak değerlerinin ve dönüş adresinin saklanmasının yanı sıra keme servis yordamı içinde kullanılan yazmaçların önceki değerlerinin saklanması için de kullanılır) yığını kullanmaktadır. Bu kullanım yoğunluğu göz önünde bulundurularak yığın büyüklüğü belirlenmelidir. Yığın hiçbir zaman ucu ucuna yetecek kadar az veya ne olur ne olmaz düşüncesiyle hiçbir zaman kullanılamayacak kadar büyük tanımlanmamalıdır. Yığının büyüklüğüne bağlı olarak SP yazmacının ilk değeri belirlenir.

[123] Assembly komut kümesinde, yüksek seviyeli programlama dillerinde olduğu gibi veri girişi için kullanılan input/read veya veriyi çıkışı için kullanılan print/write tarzı komutlar yoktur. Giriş/çıkış işlemleri için bu amaçla yazılmış assembly yordamlardan/makrolarından yararlanılabilir (bkz. 16.6.6. bölüm). Ancak bu kitap kapsamında giriş değişkenleri veri kesiminde ilk değer ataması yapılmış olarak verilecek, sonuç ise veri alanı içinde tanımlanmış değişkenlerde ve/veya yazmaç(lar)da oluşturulacaktır.

sini takiben bizim onu çağırdığımız noktaya (genellikle komut satırına) geri dönmesi gerekir. Yordamın/programın geri dönebilmesi için yığın üzerinde dönüş adresinin saklanması gerekir. Yığın üzerinde saklanan dönüş adresleri, çağırılan yordamın son komutu olan **RETF** komutunun (satır 24) işletilmesi ile yığından çekilir. Diğer deyiş ile programın komut satırından çalıştırılması kesimler arası (**intersegment**) yordam çağırma işlemi gibi değerlendirilir. **FAR** tanımlı (EXE tipinde) yordamların dönüş adresi kesim ve göreli konum ikilisinden oluşacaktır. Bu ikili sayesinde işlemcinin adresleyebileceği bellek aralığında herhangi bir noktaya dönüş yapılabilir. Dönüş adresini belirleyen kesim ve göreli konum değerlerinin yığına yerleştirilmesi programın kontrolünü elinde bulunduran ana yordamın sorumluluğudur.

EXE tipindeki programların tasarımları nedeniyle ana yordam çalışmaya başlatıldığında DS yazmacı dönüş için gerekli olan kesim adresini barındırmaktadır. Öncelikli olarak bu adresin dönüş kesim adresi olarak yığında saklanması gerekir (satır 15). Daha sonra dönüş için gereken göreli konum adresi yığına konmalıdır. Bu değer de tasarım uyarınca 0000H olacaktır (satır 16-17). Yığına göreli konum adresini uygun değer ile koyabilmek üzere PUSH işleminden önce AX yazmacı XOR işlemine tabii tutularak sıfırlanır.

EXE tipindeki programlarda **ASSUME** komutu her ne kadar kesim yazmaçları ile programcının tanımladığı kesimleri eşlese de, DS kesim yazmacı özel durumundan ötürü uygun kesimi göstermez. Bu durumda ana yordam içinde programcının DS yazmaç değerini kendi tanımladığı veri kesimini gösterecek biçimde düzeltmesi gerekir. DS yazmacının *Datasg* isimli kesimi göstermesini sağlamak üzere ilgili kesim adresi AX yazmacından yararlanılarak DS yazmacına aktarılır (satır 19-20). DS kesim yazmacına ilgili kesim adresinin tek bir komut ile aktarılamamasının arkasında komut kümesindeki kısıt yatmaktadır. Tanımlı komut kümesi, kesimi belirleyen etiketin doğrudan bir kesim yazmacına aktarılmasına izin vermediği için AX yazmacından yararlanılır. İlk işlemde AX yazmacına aktarılan *Datasg* isimli veri kesim adresi (satır 19) ikinci işlemde DS yazmacına atanmaktadır (satır 20).

Bu temel işlemlerin tamamlanması ile yordamın yapması gereken diğer işlemlerin programcı tarafından kodlanmasına geçilir. Her yordam bir **RET/RETF** komutu ile sonlanacaktır. Böylece daha önce (satır 15–17) yığına koyduğumuz dönüş adresleri yığından alınarak program sonlandırılacaktır. Tanımlı yordam **ENDP** (satır 25) ve kod kesimi **ENDS** sözde komutları (satır 26) ile bitirilir. Programın bittiğini gösterir **END** sözde kodu ise programdaki en son komuttur. END sözde komutunu takiben programın hangi noktadan (yordamdan) çalışmaya başlayacağını belirten etiket yazılmalıdır (satır 27).

9.1.1 Örnek EXE Programlar

Bu kısımda bazı basit uygulamalar EXE program tipinde kodlanacak ve adım adım nasıl derlendiği, ne şekilde sonuç ürettiği ve nasıl değerlendirilmesi gerektiği açıklanacaktır.

Örnek 9-1

16 bit uzunluğunda pozitif iki tam sayı toplayan programı EXE tipinde yazalım. Tahmin edilebileceği gibi pozitif 16 bit'lik iki sayının toplamı, 16 bit ile ifade edilebilenden daha büyük bir değer olabilir. Bu durumda sonucun bir üst birim cinsinden, yani 32 bit olması gerekecektir. Buna göre programınızda toplanacak sayıları *ASAYISI* ve *BSAYISI*, sonucu ise *TOPLAM* isimli değişkenlerde tutmak için gerekli veri tanımlamalarını yapalım. Toplama işleminin AX yazmacı kullanılarak yapılması daha etkin bir program yazmamızı sağlayacaktır. *TOPLAM* isimli değişkenin 32 bit olacağı ve 8086 işlemcisi ile bir seferde en fazla 16 bit işlem yapılabildiği dikkate alındığında yazmaç dolaylı, baz göreli veya doğrudan indisli adresleme tekniklerinden birinin kullanılması uygun olacaktır.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 9-1.ASM
; 16 bit tanımlı ASAYISI ve BSAYISI isimli değişkenleri toplayarak elde
; edilen sonucu TOPLAM isimli 32 bitlik değişkene yerleştiren EXE program
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE 16 BITLIK POZITIF IKI SAYININ TOPLANMASI
STACKSG SEGMENT PARA STACK 'STACK'
        DW 10 DUP(?)
STACKSG ENDS
DATASG  SEGMENT PARA 'DATA'
ASAYISI DW  1234H       ; Giriş/Çıkış değerlerinin veri alanı üzerinde ilk
BSAYISI DW  0ABCDH      ; değer ataması yapılmaktadır.
TOPLAM  DD  0H          ; TOPLAM değişkeninin ilk değeri sıfır yapılır
DATASG  ENDS
CODESG  SEGMENT PARA 'CODE'
        ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
ANA     PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        SUB  AX, AX
        PUSH AX
;---------------------------------------------------------------------
; DATASG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DATASG
        MOV DS, AX
;---------------------------------------------------------------------
; Toplama işleminin gerçekleştirildiği kod bloğu
;---------------------------------------------------------------------
        LEA SI, TOPLAM   ; SI TOPLAM değişkenin başladığı adresi belirler
        MOV AX, ASAYISI  ; AX yazmacı ASAYISI'nın değerini tutar
        MOV BX, BSAYISI  ; BX yazmacı BSAYISI'nın değerini tutar
        ADD AX, BX       ; toplamın sonucu AX yazmacında bulunur
        MOV [SI], AX     ; sonuç TOPLAM'ın en az anlamlı word'ünde
                         ; oluşan eldeyi yüksek anlamlı word'e yaz
        ADC WORD PTR [SI+2], 0
        RET
ANA     ENDP
CODESG  ENDS
        END ANA         ; Programın başlangıç noktası
```

Programın derlenmesi için Microsoft firmasının Macro Assembler isimli derleyicisinden yararlanılmıştır. Buna göre program aşağıdaki biçimde derlenir[124];

```
C:\assembly>masm Örnek 9-1.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Object filename [Örnek 9-1.OBJ]: Örnek 9-1.obj
Source listing  [NUL.LST]: Örnek 9-1.lst
Cross-reference [NUL.CRF]: Örnek 9-1.crf
  50248 + 384248 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors
```

Derleme işleminin hatasız olarak tamamlanması ile Örnek 9-1.*OBJ* isimli bir çıkış elde edilir. Ancak kullanıcı arzu ederse LST ve CRF isimli iki dosya daha üretilebilmektedir. Bunlardan CRF (cross reference) uzantılı olan, kesimlerin tanımlarının hangi adreslerde yer aldığı ve ne kadar yer tuttuğuna dair bilgileri içermektedir. LST uzantılı dosya ise programın derleyici tarafından düzenlenmiş halidir. Aşağıda LST uzantılı dosyanın çıktısı görülmektedir.

```
Microsoft (R) Macro Assembler Version 5.00                  6/13/01
16 BITLIK IKI SAYININ TOPLANMASI                            Page     1-1
                        PAGE 60,80
                        TITLE 16 BITLIK POZITIF IKI SAYININ TOPLANMASI
 0000                STACKSG SEGMENT PARA STACK 'STACK'
 0000  000A[                    DW 10 DUP(?)
    ????
           ]

 0014                STACKSG ENDS
 0000                DATASG SEGMENT PARA 'DATA'
 0000  1234              ASAYISI DW  1234H
 0002  ABCD              BSAYISI DW  0ABCDH
 0004  00000000          TOPLAM  DD  0H
 0008                DATASG ENDS
 0000                CODESG SEGMENT PARA 'CODE'
                        ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
 0000                ANA     PROC FAR
 0000  1E                    PUSH DS
 0001  2B C0                 SUB AX, AX
 0003  50                    PUSH AX
 0004  B8 ---- R             MOV AX, DATASG
 0007  8E D8                 MOV DS, AX
 0009  8D 36 0004 R        LEA SI, TOPLAM
 000D  A1 0000 R             MOV AX, ASAYISI
 0010  8B 1E 0002 R        MOV BX, BSAYISI
 0014  03 C3                 ADD AX, BX
 0016  89 04                 MOV [SI], AX
 0018  83 54 02 00         ADC WORD PTR [SI+2], 0
 001C  CB                    RET
 001D                ANA     ENDP
 001D                CODESG  ENDS
                               END ANA
```

[124] Derleyiciye parametre olarak aktarılan ASM uzantılı dosyayı takiben ";" simgesi konacak olursa derleyici OBJ, LST ve CRF isimli dosya isimlerini sormadan doğrudan OBJ uzantılı dosyayı üretecektir.

```
Microsoft (R) Macro Assembler Version 5.00                  6/13/01
16 BITLIK IKI SAYININ TOPLANMASI                            Symbols-1

Segments and Groups:
                N a m e         Length    Align   Combine Class

CODESG . . . . . . . . . . . . .     001D PARA NONE 'CODE'
DATASG . . . . . . . . . . . . .     0008 PARA NONE 'DATA'
STACKSG  . . . . . . . . . . . .     0014 PARA STACK    'STACK'

Symbols:
                N a m e         Type  Value    Attr
ANA  . . . . . . . . . . . . . .     F PROC   0000 CODESG   Length = 001D
ASAYISI  . . . . . . . . . . . .     L WORD   0000 DATASG
BSAYISI  . . . . . . . . . . . .     L WORD   0002 DATASG
TOPLAM . . . . . . . . . . . . .     L DWORD  0004 DATASG
@FILENAME  . . . . . . . . . . .     TEXT  Örnek 9-1

     44 Source  Lines
     44 Total   Lines
     12 Symbols
  50800 + 383696 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors
```

Örnek 9-1'de görüldüğü gibi TITLE sözde komutunun yanında yazılı metin LST uzantılı dosyanın her sayfasının başında yer almaktadır. LST uzantılı dosyanın ilk bölümü soldan sağa doğru incelendiğinde 3 ayrı sütun olarak organize edilmiş olduğu görülmektedir. Soldan sağa doğru sırasıyla; adres bilgisi, anımsatıcıların makine kodu karşılıkları ve programınızda kodladığınız şekliyle anımsatıcılar yer almaktadır. Makine kodu içerisinde **R** harfleri, adresin bağıl (**R**elative) olduğunu ifade etmektedir. Örneğin;

```
0004  B8 ---- R            MOV AX, DATASG
```

satırı incelendiğinde bu komutun kod kesimi içinde 0004H göreli adresinde bulunduğu, makine kodu karşılığının 0B8H olduğu, *DATASG* değerinin program belleğe yükleninceye kadar belirli olmadığı (---- sembolü bu anlamdadır) ve adreslemenin bağıl (Relative) olarak yapıldığı anlaşılır. Buna göre komut 3 byte uzunluğundadır. Diğer yandan;

```
0009  8D 36 0004 R         LEA SI, TOPLAM
```

satırı incelendiğinde bu komutun kod kesimi içinde 0009H göreli adresinde bulunduğu, makine kodu karşılığının 8DH 36H olduğu, 0004H değerinin *TOPLAM* isimli değişkenin *DATASG* isimli kesim tanımına bağlı olarak 0004H adresindeki byte'ta yer aldığı anlaşılır. Buna göre komut 4 byte uzunluğundadır.

Takip eden sayfalarda kesim ve grup tanımlamaları ile ilgili açıklamalar, kesim uzunlukları ve tanımlanış şekillerine ait bilgiler verilmektedir. Ayrıca yordam ve değişkenler için kullanılan sembollerin hangi tipte, hangi kesim içinde tanımlandıklarını açıklayan satırlar da yer almaktadır. Programda hata var ise bunlar ile ilgili bilgiler de LST uzantılı dosya içerisinde yer alır.

Örnek 9-2

32 bit uzunluğunda pozitif iki tam sayı toplayan programı EXE tipinde yazalım.

```
; ---------------------------------------------------------------------
; PROGRAM :Örnek 9-2.ASM
; 32 Bit olarak tanımlanmış olan ASAYISI ve BSAYISI isimli değişkenleri
; toplayarak elde edilen sonucu TOPLAM isimli 64 bitlik değişkene
; yerleştiren EXE tipinde program
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE 32 BITLIK POZITIF IKI SAYININ TOPLANMASI
STACKSG SEGMENT PARA STACK 'STACK'
        DW 10 DUP(?)
STACKSG ENDS
DATASG  SEGMENT PARA 'DATA'
ASAYISI DD      12345678H
BSAYISI DD      9ABCDEF0H
TOPLAM  DQ      0H
DATASG  ENDS
CODESG  SEGMENT PARA 'CODE'
        ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
ANA     PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        SUB  AX, AX
        PUSH AX
;---------------------------------------------------------------------
; DATASG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DATASG
        MOV DS, AX
;---------------------------------------------------------------------
; Toplama işleminin gerçekleştirildiği kod bloğu
;---------------------------------------------------------------------
        LEA SI, ASAYISI  ; Sayılarla bir seferde 64 bit toplama yapılmadığı
        LEA DI, BSAYISI  ; için 16 bitlik parçalar halinde okunacaktır
        LEA BX, TOPLAM   ; Bu nedenle sayılara SI/DI/BX Yazmaçları
                         ; yardımıyla erişilecektir.
        MOV AX, [SI]     ; AX yazmacına ASAYISI'nın düşük anlamlı word'ü
        MOV DX, [DI]     ; DX yazmacına BSAYISI'nın düşük anlamlı word'ü
        ADD AX, DX
        MOV [BX], AX     ; Elde edilen TOPLAM'ın düşük anlamlı word'üne
        MOV AX, [SI+2]   ; AX yazmacına ASAYISI'nın yüksek anlamlı word'ü
        MOV DX, [DI+2]   ; DX yazmacına BSAYISI'nın yüksek anlamlı word'ü
        ADC AX, DX       ; Eldeyi de işleme dâhil ediyoruz ve sonucu
        MOV [BX+2], AX   ; TOPLAM değişkenine yazıyoruz
        ADC WORD PTR [BX+4], 0
                         ; iki sayının 32 bit toplamında elde oluşmuş ise
                         ; bu değeri TOPLAM değişkeninin yüksek anlamlı
                         ; DWord alanına aktarabilmek için eldeli toplama
                         ; işlemi yapıyoruz.
        RET
ANA     ENDP
CODESG  ENDS
        END     ANA      ; programın başlangıç noktası
```

Daha önceki örneğimizdekine benzer olarak pozitif 32 bit'lik iki sayının toplamı, 32 bit ile ifade edilebilenden daha büyük bir değer olabilir. Bu durumda *TOPLAM* isimli sonuç değerini bir üst birim cinsinden, yani DQ olarak (64 bit) tanımlanması

gerekecektir. Şimdi bu programın derleyiciden geçtikten sonra oluşan LST uzantılı dosyayı inceleyelim.

```
Microsoft (R) Macro Assembler Version 5.00                  6/14/01
32 BITLIK IKI SAYININ TOPLANMASI                            Page     1-1
                        PAGE 60,80
                        TITLE 32 BITLIK POZITIF IKI SAYININ TOPLANMASI
 0000                STACKSG SEGMENT PARA STACK 'STACK'
 0000  000A[                     DW 10 DUP(?)
    ????
          ]

 0014                STACKSG ENDS
 0000                DATASG SEGMENT PARA 'DATA'
 0000  78563412          ASAYISI DD      12345678H
 0004  F0DEBC9A          BSAYISI DD      9ABCDEF0H
 0008  0000000000000000 TOPLAM  DQ      0H
 0010                DATASG ENDS
 0000                CODESG SEGMENT PARA 'CODE'
                        ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
 0000                ANA     PROC FAR
 0000  1E                    PUSH DS
 0001  2B C0                 SUB AX, AX
 0003  50                    PUSH AX
 0004  B8 ---- R             MOV AX, DATASG
 0007  8E D8                 MOV DS, AX
 0009  8D 36 0000 R        LEA SI, ASAYISI
 000D  8D 3E 0004 R        LEA DI, BSAYISI
 0011  8D 1E 0008 R        LEA BX, TOPLAM
 0015  8B 04                 MOV AX, [SI]
 0017  8B 15                 MOV DX, [DI]
 0019  03 C2                 ADD AX, DX
 001B  89 07                 MOV [BX], AX
 001D  8B 44 02         MOV AX, [SI+2]
 0020  8B 55 02         MOV DX, [DI+2]
 0023  13 C2                 ADC AX, DX
 0025  89 47 02         MOV [BX+2], AX
 0028  83 57 04 00         ADC WORD PTR [BX+4], 0
 002C  CB                    RET
 002D                ANA     ENDP
 002D                CODESG  ENDS
                               END     ANA

Segments and Groups:
                N a m e          Length    Align   Combine Class

CODESG . . . . . . . . . . . . .      002D PARA NONE 'CODE'
DATASG . . . . . . . . . . . . .      0010 PARA NONE 'DATA'
STACKSG  . . . . . . . . . . . .      0014 PARA STACK    'STACK'

Symbols:
                N a m e          Type  Value    Attr
ANA  . . . . . . . . . . . . . .      F PROC   0000 CODESG   Length = 002D
ASAYISI  . . . . . . . . . . . .      L DWORD  0000 DATASG
BSAYISI  . . . . . . . . . . . .      L DWORD  0004 DATASG
TOPLAM . . . . . . . . . . . . .      L QWORD  0008 DATASG
@FILENAME  . . . . . . . . . . .      TEXT

     55 Source  Lines
     55 Total   Lines
     12 Symbols
  50800 + 383696 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors
```

Örnek 9-3

10 elemanı olan, word olarak tanımlı bir dizi içinde kaç tane tek, kaç tane çift sayı olduğunu bulan programı EXE tipinde yazalım.

Bu örnekte *DIZI* isimli, 10 elemanlı bir diziden yararlanacağız. *DIZI* elemanlarının tek veya çift olduğunu anlamak için birkaç yöntem kullanmak mümkündür. Bunlardan bir tanesi sayıyı sağa doğru bir bit kaydırarak CF bayrağının durumunu kontrol ederek tek veya çift ayrımını yapmaktır. Şimdi problemin bu teknikle çözülmesi durumunda üretilen kodu inceleyelim.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 9-3.ASM
; 10 elemanlı bir dizideki tek ve çift sayıların adedini bulan program
; Bu programda SHIFT metodu kullanılır
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Tek Çift / SHIFT kullanarak
SSEG    SEGMENT  PARA STACK 'STACK'
        DW 12 DUP(0)
SSEG    ENDS
DSEG    SEGMENT  PARA 'DATA'
;---------------------------------------------------------------------
; Değişken tanımları
;---------------------------------------------------------------------
DIZI    DW 10,21,43,71,199,67,88,234,0,467
ELEMAN  DW 10
TEK_S   DW 0
CIFT_S  DW 0
DSEG    ENDS
CSEG    SEGMENT PARA 'CODE'
TC      PROC    FAR
        ASSUME CS:CSEG, SS:SSEG, DS:DSEG
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        XOR AX, AX
        PUSH AX
;---------------------------------------------------------------------
; DSEG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DSEG
        MOV DS, AX
;---------------------------------------------------------------------
; Kontrol işleminin gerçekleştirildiği kod bloğu
;---------------------------------------------------------------------
        MOV CX, ELEMAN   ; Çevrim değişkenine eleman sayısı konur
        MOV SI, 0        ; DIZI, sıfırıncı indisli elemandan başlar
CEVRIM: MOV AX, DIZI[SI] ; Elemanlar sırası ile AX yazmacına aktarılır
        SHR AX, 1        ; Sağa doğru öteleme ile en az anlamlı bit CF'ye
        ADC TEK_S, 0     ; CF, 1 ise tektir ve tek sayı adedine eklenir
        ADD SI, 2        ; DIZI word tanımlı olduğu için bir sonraki dizi
                         ; elemanı 2 byte ileride olacak,
        LOOP CEVRIM      ; işlemi dizinin son elemanına gelinceye kadar
                         ; tekrarlamayı sağlayan çevrim komutu
        MOV AX, ELEMAN   ; sonuç olarak tek sayı olan eleman sayısını
                         ; biliyoruz. Bunu ELEMAN sayısından çıkartalım
        SUB AX, TEK_S    ; böylece kalanların çift olduğunu bulduk
        MOV CIFT_S, AX   ; bulunan değeri çift sayı adedi olarak atanır
        RET
TC      ENDP
CSEG    ENDS
        END TC           ; programın başlangıç noktası
```

Yazdığımız programda bazı önemli noktaları vurgulamak faydalı olacaktır.

1. *DIZI* ismi ile tanımlanan verilere erişmek için aşağıdaki komut satırlarından yararlanılmıştır;

```
        MOV SI, 0
CEVRIM: MOV AX, DIZI[SI]
```

Burada yapılan işlem ile *DIZI*'ye yüksek seviyeli dillerde olduğuna benzer bir şekilde erişilmektedir. Ancak bu işlem arka planda

$AX \leftarrow [SI + EA_{DIZI}]$

şeklinde gerçekleştirilmekte yani *DIZI* değişkeninin veri kesiminde bulunduğu göreli konum adresi bulunmakta ve bu adrese SI yazmacındaki sayı kadar artım verilerek elde edilen adresin bellekte gösterdiği veriye erişilmektedir. Bu satırları

```
         LEA SI, DIZI
CEVRIM:  MOV AX, [SI]     ; AX←[SI]
```

ile değiştirirsek her seferinde *DIZI* isimli değişkenin göreli konum adresi ve SI yazmaç değeri toplama işlemine girmediği için daha etkin bir kod üretmiş oluruz.

2. Tek sayının tespiti için SHR komutundan yararlanılmaktadır. Ancak

```
         MOV SI,0                                  MOV SI,0
CEVRIM:  MOV AX, DIZI[SI]     yerine   CEVRIM:  SHR DIZI[SI], 1
         SHR AX, 1                              ADC TEK_S, 0
         ADC TEK_S, 0
```

veya

```
        LEA SI,DIZI                                LEA SI,DIZI
CEVRIM: MOV AX, [SI]          yerine   CEVRIM:  SHR [SI], 1
        SHR AX, 1                              ADC TEK_S, 0
        ADC TEK_S, 0
```

yazılmış olsa veri kesiminde *DIZI* olarak tanımlanan değişkenin bütün elemanları değişmiş, yani orijinal dizi bozulmuş olacaktır. Eğer program içindeki değişkenlerin değerlerinin bozulması istenmiyorsa doğrudan elamanlar üzerinde işlem yapmak yerine yazmaçları veya ara değişkenleri kullanmak en doğru yöntem olacaktır. Genel olarak programınızı tasarlarken giriş verisinin bozulmamasına dikkat etmemiz uygun olacaktır.

3. Dizi işlemlerinde bir sonraki elemana erişmek üzere dizinin tipine bağlı olarak indis değişkenlerinin ayarlanması gereklidir. Çevrim içindeki indis değişkenlerinin doğru değiştirilememesi programın doğru çalışmasını etkiler. Biz de programımızda indis olarak kullandığımız SI yazmacındaki değeri dizinin tipi-

ne bağlı olarak değiştirmeliyiz. Dizimiz **DW** (word) olarak tanımlandığı için artımın değeri **2** olacaktır.

```
ADD SI, 2
```

Alternatif bir çözüm olarak, **TEST** komutunu sayının tek/çift olmasını belirleyen en az anlamlı bit dışındakileri 0 yapacak bir maske ile kullanıp ZF bayrağının durumuna göre karar verebiliriz. Buna göre programımızı tekrar yazarsak;

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 9-3.ASM
; 10 elemanlı bir dizideki tek ve çift sayıların adedini bulan program
; Bu programda TEST komutu kullanılmıştır.
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Tek Çift / TEST kullanarak
SSEG    SEGMENT PARA STACK 'STACK'
        DW 12 DUP(0)
SSEG    ENDS
DSEG    SEGMENT PARA 'DATA'
;---------------------------------------------------------------------
; Değişken tanımları
;---------------------------------------------------------------------
DIZI    DW 10,21,43,71,199,67,88,234,0,467
ELEMAN  DW 10
TEK_S   DW 0
CIFT_S  DW 0
DSEG    ENDS
CSEG    SEGMENT PARA 'CODE'
TC      PROC    FAR
        ASSUME CS:CSEG, SS:SSEG, DS:DSEG
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        XOR AX, AX
        PUSH AX
;---------------------------------------------------------------------
; DSEG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DSEG
        MOV DS, AX
;---------------------------------------------------------------------
; Kontrol işleminin gerçekleştirildiği kod bloğu
;---------------------------------------------------------------------
        MOV CX, ELEMAN   ; Çevrim değişkenine eleman sayısı konur
        LEA SI, DIZI     ; DIZI'nin başlangıç adresi SI yazmacında
CEVRIM: TEST WORD PTR [SI], 0001H
                         ; TEST komutu işlenenleri bozmaz
        JZ CIFT          ; Sonuç sıfır ise sayı çifttir ilgili satıra git
TEK:    INC TEK_S        ; sayı tekdir, tek sayı adedini arttır
CIFT:   ADD SI, 2        ; bir sonraki elemana erişmek için SI arttırılır
                         ; DIZI word tanımlı olduğu için bir sonraki dizi
                         ; elemanı 2 byte ileride olacaktır.
        LOOP CEVRIM      ; işlemi dizinin son elemanına gelinceye kadar
                         ; tekrarlamayı sağlayan çevrim komutu
        MOV AX, ELEMAN   ; çevrim sonucunda sadece tek olan kaç tane eleman
                         ; olduğunu biliyoruz. ELEMAN sayısından çıkartır
        SUB AX, TEK_S    ; böylece kalanların çift olduğunu bulabiliriz
        MOV CIFT_S, AX   ; bulduğumuz değeri tanımlı değişkene yaz
        RET
TC      ENDP
CSEG    ENDS
        END TC           ; programın başlangıç noktası
```

Yazdığımız bu programda da vurgulanması gereken bazı özelliklere değinelim;

1. TEST komutu işlenenler üzerinde bir değişikliğe neden olmadığından işlemi giriş değişkeni olan *DIZI* üzerinde uygulanmasında herhangi bir sakınca yoktur. Ancak TEST işlemine tabii tutulacak olan değerlerin aynı büyüklükte (byte, word gibi) olması gereklidir. Biz bir bellek alanının içeriğini [SI] biçiminde ifade ettiğimizde bunun ne büyüklükte bir veri ifade ettiğini belirlemiş olmayız. Bu nedenle TEST işlemi kullanılırken **WORD PTR** ifadesi ile SI yazmacının gösterdiği bellek adresinden bir word'ün 0001H ile AND işlemine tabii tutulmasını sağlıyoruz.

```
        LEA SI, DIZI
CEVRIM: TEST WORD PTR [SI], 0001H
```

2. TEST işleminin sonucunda bayraklar değişmektedir. Herhangi bir sayının 0001H ile AND işlemine sokulması aslında sayının en az anlamlı biti dışında kalan bitlerin 0 yapılması demektir. Sayıların ikili tabanda gösterimlerinden hatırlanacağı gibi bir sayının en az anlamlı biti 1 ise o sayı tek, aksi halde çifttir. O zaman TEST işleminin sonucunda ZF bayrağının bir olması (yani sayının çift olması) durumunda programın bir sonraki sayı üzerinde işlem yapabilmek için gerekli indis ayarlama işlemlerini yapması gerekir. Aksine sonucun sıfırdan farklı olması (bu örnekte tek bir olasılık söz konusu olup o da sonucun bir olması, yani ZF bayrağının sıfır olması) durumunda ise sayının tek olduğuna karar verilerek tek sayıların adedini tutan değişkenin arttırılması sağlanacaktır. Ancak unutulmaması gereken nokta sayı ister tek, isterse çift olsun her durumda çevrim içinde indis değişkeninin bir sonraki elemanı gösterecek şekilde ayarlanması gerektiğidir. Bu işlem yüksek seviyeli programlama dillerinde kullanılan NULL-ELSE, yani else kolunda ayrı bir işlem yapılmadığı durumdur.

```
     JZ CIFT
TEK: INC TEK_S
CIFT:ADD SI, 2
     LOOP CEVRIM
```

Örnek 9-4

10 elemanlı, elemanları word tanımlı bir dizi verilmektedir. Bu dizinin elemanlarının on altılı sayı sistemindeki gösterimlerindeki hanelerini (hex-digit) tersyüz eden programı EXE tipinde yazalım.

Bizden istenen 0ABCDH olarak verilen bir sayının işlem sonucunda 0DCBAH şeklini almasıdır. Bunu yapabilmek için döndürme komutlarından yararlanılması uygun olacaktır. Ancak sayıyı 16 defa sağa veya sola döndürecek olursak aynı değeri elde ederiz. Benzer biçimde sayıyı sağa doğru 4 defa döndürsek 0DABCH, 8 defa döndürsek 0CDABH değerlerini elde ederiz. Bunlardan hiçbiri bizi istediği-

miz sonuca ulaştırmaz. Oysa önce sayının yüksek anlamlı byte'ı ile düşük anlamlı byte'ının yerini değiştirerek 0CDABH değerini elde edebiliriz. Ardından bu değerin hem yüksek anlamlı, hem de düşük anlamlı byte'larını sağa veya sola doğru 4 defa döndürürsek 0DCBAH değerini elde etmemiz mümkün olur. Fakat kodlama sırasında assembly diline has bir durumu da göz önüne almak gereklidir. Tekrarlanan işlemlerde CX yazmacı tekrar sayısını tutmak üzere kullanılır. Bizim problemimizde tarif edilen döndürme işlemleri 10 eleman için ayrı ayrı tekrarlanacaktır. Ayrıca döndürme işlemini de 4 defa yapacağımız düşünülecek olursa CX yazmacı iki farklı değeri tutacak şekilde kullanılmalıdır.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 9-4.ASM
; 16 Bit'lik bir sayının on altılık tabandaki ifadesindeki hanelerinin
; tersten yazılmasını sağlayan EXE tipinde program
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE HEX Digitleri tersyüz etme
SSEG    SEGMENT PARA STACK 'STACK'
        DW 12 DUP(0)
SSEG    ENDS
DSEG    SEGMENT PARA 'DATA'
;---------------------------------------------------------------------
; Dizi elemanları ve eleman sayısı tanımlanmıştır.
;---------------------------------------------------------------------
ELEMAN  DW 10
DIZI    DW 1234H,2345H,3456H,4567H,5678H,6789H,789AH,89ABH,9ABCH,0ABCDH
DSEG    ENDS
CSEG    SEGMENT PARA 'CODE'
INDIAN  PROC   FAR
        ASSUME CS:CSEG,SS:SSEG,DS:DSEG
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        SUB AX, AX
        PUSH AX
;---------------------------------------------------------------------
; DSEG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DSEG
        MOV DS, AX
;---------------------------------------------------------------------
; Kontrol işleminin gerçekleştirildiği kod bloğu
;---------------------------------------------------------------------
       MOV CX, 10        ; Eleman sayısı CX'e yerleştirildi
       LEA SI, DIZI      ; Elemanlara erişmek için SI indis olacak
CEVRIM:MOV AX, [SI]      ; Sayı üzerindeki işlemi hızlı yapabilmek için
                         ; AX yazmacı üzerinden yapıyoruz
       XCHG AL, AH       ; Düşük ve yüksek anlamlı byte'ları değiştir
       PUSH CX           ; Tekrar sayısını belirleyen CX yazmaç değerini
                         ; yığında sakla
       MOV CL, 4         ; döndürme işleminin tekrar sayısı CL de
       ROR AL, CL        ; AL ve AH değerlerini kendi içlerinde CL ile
       ROR AH, CL        ; belirlenen değer kadar (4) sağa döndür
       MOV [SI], AX      ; AX üzerinde oluşan yeni değeri yerine koy
       ADD  SI,2         ; SI'yı sonraki word'ü gösterecek biçimde ayarla
       POP CX            ; Yığında saklanan değeri alarak işleme devam et
       LOOP CEVRIM       ; işlemi CX de belirlenen sayı kadar tekrarla
       RET               ; dön
INDIAN ENDP
CSEG   ENDS
       END INDIAN        ; programın başlangıç noktası
```

Programımızda dikkat edilmesi gerekli olan noktaları birlikte gözden geçirelim;

1. Assembly programlarda CX yazmacı genel olarak tekrar sayılarını tutmak için kullanılır. Programlarda iç içe tekrarlanan işlemler var ise CX yazmacındaki değerlerin karıştırılmaması gerekir. Programımızda dizi üzerinde yapılacak işlemin tekrar sayısını belirleyen değerin, içte döndürme işlemi için kullanılan değer ile karıştırılmaması için, önce çevrim değeri yığında saklanmış, ardından CL yazmacına döndürme işlemi için kullanılacak tekrar değeri konulmuştur.

```
PUSH CX
MOV CL, 4
```

   Kodun devamında döndürme işlemlerinin tamamlanmasını takiben elde edilen değer bellekte yerine yerleştirilmiş, indis olarak kullanılan SI yazmaç değeri bir sonraki dizi elemanını gösterecek şekilde ayarlanmış, POP CX komutu ile çevrimin tekrar değeri CX yazmacına geri alınarak LOOP komutu ile çevrim tamamlanmıştır[125].

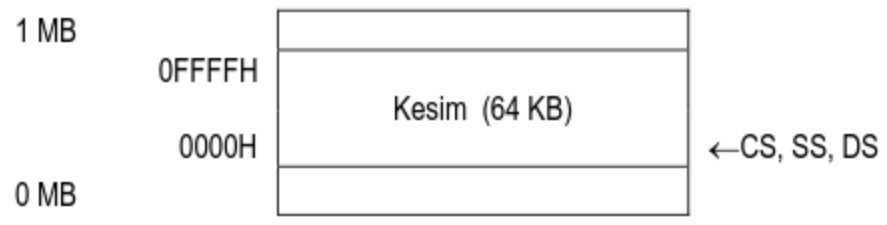
2. Öteleme ve döndürme komutlarında işlem birden fazla tekrarlanacak ise kolaylık olması açısından işlem tekrar sayısı CL yazmacı ile verilmektedir. Bunun için CL yazmacına tekrar sayısı atandıktan sonra;

```
ROR AL, CL
ROR AH, CL
```

   komutları kullanılacaktır. Döndürme komutlarından ikinci işlenen olarak kullanılan CL yazmacının değeri yapılan işlem ile değişikliğe uğramaz.

9.2 COM Yapısındaki Assembly Programların Genel Özellikleri

COM yapısındaki programlar, EXE yapısındakilerden farklı olarak, çakışık kesim organizasyonundadır.[126]. Diğer deyişle bir COM program için bellekte bir tane 64 KB kesim ayrılmıştır. Bu durum bütün kesim yazmaçlarının aynı yeri göstermesi olarak da özetlenebilir (Şekil 9-1).



Şekil 9-1: COM yapısındaki programının bellekteki yerleşimi

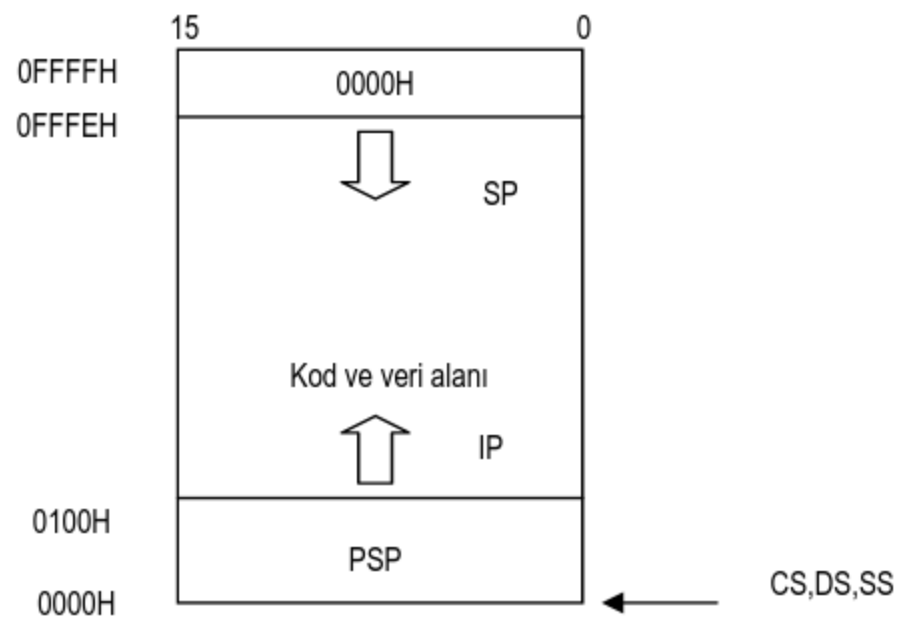
[125] Eğer daha önce yığına saklanan CX değeri POP komutu ile yığından alınmadan işleme devam edilecek olursa, MOV CL, 4 komutundan ötürü CX hiçbir zaman sıfır olamayacak, sonsuz döngüye girildiği için program hiçbir zaman sonlanmayacaktır.

[126] COM yapısındaki programın bir segment kullanması CP/M işletim sistemi ile uyumluluğunu sağlamak içindir.

COM programlarda sadece kod kesimi tanımlanmaktadır. Ancak yığın ve veri kesimleri de bu yapı içerisinde yer alır. Kesimlerin çakışık olduğu

`ASSUME CS:`*`kesim-ismi`*`, SS:`*`kesim-ismi`*`, DS:`*`kesim-ismi`*

tanımı ile belirlenmektedir. Yığın kullanımı yüksek adresten düşük adrese doğru (0FFFEH adresinden azalarak) ilerlemekte ve ilk değer (dönüş değeri) olarak yığında 0000H değeri ile başlanmaktadır. Diğer taraftan programı oluşturan komutlar ise **0100H** adresinden itibaren yerleşmektedir. 0000H-0100H arasında kalan 256 byte PSP (Program Segment Prefix) adını alır. PSP, programın sonlandırılması durumunda gidilecek adreslerin, programa aktarılan parametrelerin ve yardımcı bir takım değerlerin tutulduğu bir alandır. Genel olarak COM programın kesim yapısı ve bu yapı içinde kesim ve kesim içi işaretçiler (yazmaçlar) arasındaki ilişki Şekil 9-2'de gösterilmiştir;



Şekil 9-2: COM yapısındaki programın kesim organizasyonu

COM programlarda ana yordam aynı kesim içinde işleyeceği için **NEAR** olarak tanımlanır ve EXE programlarda olduğu gibi dönüş değerleri programcı tarafından yığına yerleştirilmez. Dönüş adresi programa başlarken kesimin 0FFFEH adresine yerleştirilen 0000H değeridir. COM programlarda verilerin tanımlanacağı alan çok önemlidir. Veriler istenirse programın hemen baş kısmında koddan önce veya yordamın sonu ile kesim tanımı bitirilmeden hemen önce yapılır. Buna göre COM yapısındaki program için iki ayrı iskelet yapısı olduğunu söyleyebiliriz (Tablo 9-2, Tablo 9-3).

COM yapısındaki programlarda kesimlerin çakışık olması nedeniyle bir tek kod kesimi tanımlaması yapılmaktadır (Tablo 9-2, satır 3). Tanımlanan kesim alanının en başındaki 256 byte'lık bölgede PSP yer alacağı için kodumuz 100H göreli konum adresinden başlamalıdır. Bunu belirtmek üzere **ORG 100H** ibaresi kullanılmaktadır (satır 4). Yığın ve veri kesimleri için ayrı tanımlamalar bulunmadığı için

ASSUME sözde komutu ile tüm kesimleri tanımlı olan kod kesimi ile çakıştırılarak tanımlanır (satır 5).

Tablo 9-2: COM yapısındaki programın iskelet yapısı – Model 1

1		PAGE 60,80
2		TITLE *com türünde yazılmış bir assembly programın iskelet yapısı*
3	*Codesg*	SEGMENT PARA 'CODE'
4		**ORG 100H**
5		**ASSUME** CS:*Codesg*, SS:*Codesg*, DS:*Codesg*, ES:*Codesg*
6	*Bilgi*:	**JMP** Basla
7	*Sayib*	**DB** ?
8	*Sayiw*	**DW** ?
9		*;tanımlamak istenilen diğer veriler*
10	*Basla*	**PROC NEAR**
11		*;Yapılması gereken işlemler*
12		.....
13		**RET**
14	*Basla*	**ENDP**
15	*Codesg*	**ENDS**
18		**END** *Bilgi*

EXE yapısındaki programlarda bağımsız veri kesimi içinde yapılan tanımlamalar, COM yapısında kod kesimi içinde, ancak kod ve yığın tarafından bozulmayacak (ezilmeyecek) biçimde yapılmalıdır. Bunu sağlamak üzere kod alanı içinde kullanılan koşulsuz dallanma komutu ile ana yordama atlama yapılırken dallanma komutu ile ana yordam arasında kalan alan veri tanımlamak üzere koruma altına alınmıştır. Bu teknik, tanımlanan değişkenlerin sürekli izlenmesinin gerektiği durumlarda göreli konum adreslerinin daha çabuk bulunmasına da yardımcı olur. Yani bizim değişkenlerimiz 100H adresinden itibaren kullanılan JMP komutu ile PROC tanımı arasına yerleşecektir (satır 7, 8, 9).

COM yapısındaki ana yordamda, EXE tipindekilerden farklı olarak geri dönüş adreslerinin yığında saklanmasına, veri kesim yazmacının (DS) uygun değer ile kullanıma hazırlanmasına gerek yoktur. Zira dönüş adresi olarak programın belleğe yüklenmesi ile birlikte yığına (0FFFFH adresine 0000H değerinde) bir word yerleştirilmiştir. COM programın ana yordamının NEAR tipinde olduğu da göz önüne alındığında yığına yerleştirilen word'ün dönüş için gerekli olan adres değeri olduğu anlaşılacaktır. Kesim yazmacı değişmediği için yığında bulunan göreli konum değeri de aslında PSP alanının başını göstermektedir.

Tablo 9-2'de verilen iskelet yapısı uyarınca kodlanan COM programlarda programın bitişini belirleyen **END** komutundan sonra ise değişkenleri tanımlamak üzere ayırılan alanın başındaki koşulsuz dallanma komutunun (satır 6) etiketi yazılarak başlangıç noktası belirlenmektedir. Bu sayede program belleğe yüklendiği zaman JMP komutunun olduğu noktadan başlayacak, veri bloğunun üzerinden atlayarak NEAR tanımlı yordamı çalıştıracaktır. Böylece verilerin yer aldığı bellek alanı ko-

mut olarak değerlendirilmediği gibi tanımlı değişkenlere erişmek için ek işlem de yapılmayacaktır.

Diğer iskelet modelimizde ise (Tablo 9-3) değişkenler yordam tanımının sonuna, kesim tanımı sonlandırılmadan hemen önce yerleştirilmiştir. Bu sayede ilk modelde (Tablo 9-2) kullanılan JMP komutuna gerek kalmaz. Ancak tanımladığımız değişkenlerin hangi göreli konum adresinde olduklarını bilmek için programın kaç byte uzunlukta olduğunu bilmemiz veya RET komutunu takip eden byte'tan itibaren veri tanımlamalarını incelememiz gerekir. Programların, geliştirme aşamasında eklenen ve/veya çıkarılan her komut satırı ile uzayıp/kısaldığını düşünecek olursak verilerin bulunduğu göreli konum adresinin tespit edilmesi analiz aşamasında bazı güçlüklere neden olabilir.

Tablo 9-3: COM yapısındaki programın iskelet yapısı – Model 2

```
1                PAGE 60,80
2                TITLE com türünde yazılmış bir assembly programın iskelet yapısı
3     Codesg     SEGMENT PARA 'CODE'
4                ORG 100H
5                ASSUME CS:Codesg, SS:Codesg, DS:Codesg, ES:Codesg
6     Basla      PROC NEAR
7                ;Yapılması gereken işlemler
8                .....
9                RET
10    Basla      ENDP
11    Sayib      DB ?
12    Sayiw      DW ?
13               ;tanımlamak istenilen diğer veriler
14    Codesg     ENDS
15               END Basla
```

9.2.1 Örnek COM Programlar

COM yapısındaki programların işleyişini bazı örnekler ile inceleyelim;

Örnek 9-5

Word tipinde, 10 tane işaretli sayıdan oluşan bir dizinin en büyük elemanını bulan COM yapısındaki programı yazalım.

Bilindiği gibi sayıların işaretli veya işaretsiz olduklarını değerlendirmek programcının sorumluluğundadır. Sayının en anlamlı biti, işaret bit'i kabul edilir. İşaret bit'i 1 ise sayı negatif olarak değerlendirilir. Buna göre karşılaştırma işleminin sonucu bayraklar üzerinden değerlendirirken işaretli sayılarla kullanılması gereken türde koşullu dallanma komutlarından yararlanmamız gerekecektir. Bu komutlar JG, JL ve benzerleridir. Buna göre problemimizin çözümü için aşağıdaki kod parçası yazılmalıdır.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 9-5.ASM
; 10 elemanlı word olarak tanımlanmış bir dizi içindeki en büyük
; elemanın bulunması için COM yapısında bir program
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE En büyük elemanın bulunması
CDSEG   SEGMENT PARA 'CODE'
        ORG 100H        ; COM program 100H den başlar
        ASSUME CS:CDSEG, SS:CDSEG, DS:CDSEG
BASLA:  JMP MAXI
;---------------------------------------------------------------------
; Eleman sayısı, Dizi elemanları ve Büyük sayı için gerekli değişkenler
;---------------------------------------------------------------------
ELEMAN  DW 10
DIZI    DW -4,23,456,4,8,67,-78,9,-99,256
BYKSAY  DW ?
;---------------------------------------------------------------------
; işleminin gerçekleştirildiği yordam
;---------------------------------------------------------------------
MAXI    PROC NEAR
        MOV CX, ELEMAN   ; Eleman sayısı CX'e yerleştirildi
        LEA SI, DIZI     ; Elemanlara erişmek için SI indis olacak
        DEC CX           ; Eleman sayısından bir tane eksik işlem yap
        MOV AX, [SI]     ; ilk sayı büyük kabul edilerek AX'e konuyor
CEVRIM: ADD SI, 2        ; bir sonraki indisli elemana erişmek için +2
        CMP AX, [SI]     ; sayıları karşılaştırıyoruz
        JG DEVAM         ; AX büyük, bir sonraki eleman ile karşılaştır
        MOV AX, [SI]     ; AX küçük, yeni sayıyı AX yazmacına yerleştir
DEVAM:  LOOP CEVRIM      ; işlemi CX de belirlenen sayı kadar tekrarla
        MOV BYKSAY, AX   ; AX yazmacındaki değeri BYKSAY'a yerleştir.
        RET              ; dön
MAXI    ENDP
CDSEG   ENDS
        END BASLA        ; programın başlangıç noktası
```

Yazdığımız kodu derlemek için aşağıdaki işlemler yapılır;

```
C:\assembly>masm Örnek 9-5.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Object filename [Örnek 9-5.OBJ]: Örnek 9-5.obj
Source listing  [NUL.LST]: Örnek 9-5.lst
Cross-reference [NUL.CRF]: Örnek 9-5.crf

  50362 + 384134 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

Derleme aşamasının hatasız olarak geçmesi ile oluşan LST uzantılı dosyayı inceleyerek dikkat edilmesi gereken bazı noktaları açıklamaya çalışalım.

```
Microsoft (R) Macro Assembler Version 5.00                  6/15/01
En büyük elemanın bulunması                                 Page     1-1

                        PAGE 60,80
                        TITLE En büyük elemanın bulunması
  0000              CDSEG   SEGMENT PARA 'CODE'
  0100                      ORG 100H
                        ASSUME CS:CDSEG, SS:CDSEG, DS:CDSEG
  0100  EB 19 90             BASLA:  JMP MAXI
  0103  000A             ELEMAN  DW 10
  0105  FFFC 0017 01C8 0004  DIZI    DW -4,23,456,4,
        0008 0043 FFB2 0009  8,67,-78,9,
        FF9D 0100                     -99,256
```

```
0119  ????            BYKSAY  DW ?
011B              MAXI    PROC NEAR
011B  8B 0E 0103 R      MOV CX, ELEMAN
011F  8D 36 0105 R      LEA SI, DIZI
0123  49                  DEC CX
0124  8B 04               MOV AX, [SI]
0126  83 C6 02    CEVRIM: ADD SI, 2
0129  3B 04               CMP AX, [SI]
012B  7F 02               JG DEVAM
012D  8B 04               MOV AX, [SI]
012F  E2 F5       DEVAM:  LOOP CEVRIM
0131  A3 0119 R           MOV BYKSAY, AX
0134  C3                  RET
0135              MAXI    ENDP
0135              CDSEG   ENDS
                   END BASLA

Microsoft (R) Macro Assembler Version 5.00                  6/15/01
En büyük elemanın bulunması                                 Symbols-1

Segments and Groups:
                N a m e          Length    Align   Combine Class
CDSEG  . . . . . . . . . . . . .      0135 PARA NONE 'CODE'

Symbols:
                N a m e          Type  Value    Attr
BASLA  . . . . . . . . . . . . .      L NEAR   0100 CDSEG
BYKSAY . . . . . . . . . . . . .      L WORD   0119 CDSEG
CEVRIM . . . . . . . . . . . . .      L NEAR   0126 CDSEG
DEVAM  . . . . . . . . . . . . .      L NEAR   012F CDSEG
DIZI . . . . . . . . . . . . . .      L WORD   0105 CDSEG
ELEMAN . . . . . . . . . . . . .      L WORD   0103 CDSEG
MAXI . . . . . . . . . . . . . .      N PROC   011B CDSEG    Length = 001A
@FILENAME  . . . . . . . . . . .      TEXT  Örnek 9-5

    35 Source  Lines
    35 Total   Lines
    11 Symbols

 50368 + 384128 Bytes symbol space free

     0 Warning Errors
     0 Severe  Errors
```

1. LST uzantılı dosyanın en sol yanında programın belleğe yerleştirildiği zaman yer alacağı göreli konum adresleri verilmektedir. Buna göre aşağıdaki satırdan da anlaşıldığı gibi ilk çalıştıracağımız komut olan JMP *MAXI* komutu 0100H adresine yerleşmiştir.

   ```
   0100  EB 19 90      BASLA:  JMP MAXI
   ```

2. 0100H adresine yerleşen JMP komutu 3 byte yer kaplamaktadır. Buna göre birinci modele göre yazdığımız COM programımızda tanımladığımız değişkenler ancak **0103H** adresinden itibaren yerleşebilirler.

   ```
   0103  000A                          ELEMAN  DW 10
   0105  FFFC 0017 01C8 0004  DIZI      DW -4,23,456,4,
    0008 0043 FFB2 0009                8,67,-78,9,
    FF9D 0100                              -99,256
   0119  ????                          BYKSAY  DW ?
   ```

3. *MAXI* isimli yordam ise veri tanımlarını takiben, 011BH adresinden başlamaktadır.

```
011B      MAXI     PROC NEAR
```

4. Değişkenlere erişirken sol tarafta görünen **R** harfi erişimin bağıl adres olarak yapıldığını göstermektedir. R harfinde önce yer alan sayı o değişkenin bellekte hangi göreli konum adresinde yer aldığını göstermektedir. Buna göre aşağıdaki satırdan *ELEMAN*'ın 0103H, *DIZI*'nin ise 0105H göreli konum adreslerine yerleştiği görülmektedir.

```
011B  8B 0E 0103 R  MOV CX, ELEMAN
011F  8D 36 0105 R  LEA SI, DIZI
```

Şimdi, başarılı derleme sonucunda oluşan OBJ uzantılı dosyanın Link programından geçirilmesi işleminin nasıl yapıldığı görelim.

```
C:\assembly>link Örnek 9-5.obj

Microsoft (R) Overlay Linker  Version 3.61
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [Örnek 9-5.EXE]: Örnek 9-5.exe
List File [NUL.MAP]: Örnek 9-5.map
Libraries [.LIB]:
LINK : warning L4021: no stack segment
```

LINK programı, işlemini tamamlarken "**no stack segment**" (yığın kesimi yok) şeklinde bir mesaj vermiştir. Bunun nedeni COM program için bağımsız bir yığın kesim tanımı yapılmamış olmasıdır. Bu gerçekten bir hata değil sadece bir uyarı (warning) mesajıdır. Eğer ikincil bellek birimi üzerindeki dosyalar incelenecek olursa EXE uzantılı bir dosyanın da oluşmuş olduğu görülecektir. Yazdığımız program COM tipinde olup bir ara işlemden daha geçirilmesi gereklidir. Aksi halde programımızın çalışması mümkün değildir. Sözünü ettiğimiz son kademe işlemi eski DOS sürümlerinde bulunan EXE2BIN gibi EXE'den COM'a dönüştürücü bir program veya Internet'ten bulabileceğiniz benzer özellikli herhangi bir dönüştürücü program ile gerçekleştirilebilir. Bizim kullanacağımız program EXE2COM[127] isimli bir dönüştürücüdür. Programın çalıştırılması sırasında verilmesi gereken parametreler aşağıda gösterilmiştir.

```
C:\assembly>exe2com Örnek 9-5.exe Örnek 9-5.com
exe2com 1.04 by Chris Dunford/The Cove Software Group
```

[127] EXE2COM isimli program Chris Dunford/The Cove Software Group tarafından yazılmış olup Internet üzerinden ücretsiz olarak temin edilebilmektedir.

Bütün bu işlemlerden sonra disk üzerinde yer alan dosyalar incelediğinde aşağıdaki listede isim, uzantı, boyut ve tarih/saat bilgisi verilenlere benzer dosyalarla karşılaşılacaktır.

```
C:\assembly>dir Örnek 9-5.*

Örnek 9-5   ASM         1.736  15/06/01  16:56
Örnek 9-5   OBJ           133  15/06/01  16:57
Örnek 9-5   LST         3.887  15/06/01  16:57
Örnek 9-5   CRF           287  15/06/01  16:57
Örnek 9-5   MAP           181  15/06/01  16:58
Örnek 9-5   EXE           818  15/06/01  16:58
Örnek 9-5   COM            50  15/06/01  16:59
         7 file(s)          7.092 bytes
```

Disk üzerinde aynı isimde hem EXE hem de COM uzantılı dosyaların olduğu görülmektedir. Bu durumda DOS komut satırından Örnek 9-5 yazarak ENTER tuşuna bastığımızda her iki çalıştırılabilir koddan hangisi çalışacaktır ? Bu sorunun cevabı "*COM uzantılı olan*" dır. Zira COM kesim organizasyonu manasında EXE'den daha basittir ve daha önce devreye girmektedir. Ancak ileride herhangi bir karışıklığa neden olmaması ve ikincil bellek üzerinde boşu boşuna yer kaplamaması için EXE uzantılı dosyayı silmek faydalı olacaktır. Benzer şekilde CRF, LST, OBJ ve MAP uzantılı dosyaların kullanılması düşünülmüyor ise ikincil bellek alanını doldurmaması amacıyla silinmesi faydalı olur (ASM uzantılı dosya derleyici, linker ve EXE2COM tipinde dönüştürücü bir program elimizde olduğu sürece silinen dosyaların yeniden üretilmesi mümkündür).

Örnek 9-6

Ekrana bizim belirleyeceğimiz bir mesajı yazdırmayı sağlayan COM yapısındaki programı yazalım.

Bilgisayarın dış dünya ile ilişkisini sağlamak üzere kullanılan okuma/yazma işlemleri assembly programlamada BIOS/DOS kesmelerinden yararlanılarak yapılmaktadır. Şu ana kadar bu konular üzerinde bir açıklama yapılmamış olsa da yazacağımız programda ekrana mesaj yazdırmak üzere INT 21H'in 09H fonksiyonundan yararlanacağız. Bu kesmenin düzgün çalışabilmesi için tanım gereği DX yazmacına yazdırılacak olan mesajın başlangıç (göreli konum) adresi[128], AH yazmacında fonksiyon numarasını belirleyen 09H değeri konarak INT 21H kullanılmalıdır. Buna göre programımız aşağıdaki gibi olacaktır.

[128] Ekrana yazdırılacak olan mesajın uzunluğu mesaja göre değişeceği için mesajın sonunda kullanılan "$" karakteri ile mesajın sonu belirlenmektedir.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 9-6.ASM
; INT 21H - fonksiyon 9H ile ekrana mesaj yazmayı sağlayan
; COM yapısında bir program
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Ekrana mesaj yazma
CDSEG   SEGMENT  PARA 'CODE'
        ORG 100H        ; COM program 100H den başlar
        ASSUME CS:CDSEG, SS:CDSEG, DS:CDSEG
;---------------------------------------------------------------------
; işleminin gerçekleştirildiği yordam
;---------------------------------------------------------------------
YAZDIR  PROC NEAR
        LEA DX, MESAJ    ; DX yazdırılacak mesajın başını göstermeli
        MOV AH, 09H      ; AH, fonksiyon numarasını tutar
        INT 21H          ; 21H numaralı kesmeyi çağır
        RET              ; dön
YAZDIR  ENDP
;---------------------------------------------------------------------
; Değişkenleri tanımladığımız alan
;---------------------------------------------------------------------
MESAJ   DB 'COM yapisinda bir program calisti','$'
CDSEG   ENDS
        END YAZDIR      ; programın başlangıç noktası
```

COM uzantılı programımız, veri tanımlamalarının yapılışı itibariyle ikinci model olarak isimlendirdiğimiz yapıda (veriler yordam ile kesim sonlandırmayı sağlayan sözde komutlar arasında) yazılmıştır. Program derlendikten sonra oluşan LST uzantılı dosya aşağıda verilmiştir.

```
Microsoft (R) Macro Assembler Version 5.00                  6/15/01
Ekrana mesaj yazma                                          Page     1-1

                        PAGE 60,80
                        TITLE Ekrana mesaj yazma
 0000                CDSEG   SEGMENT  PARA 'CODE'
 0100                    ORG 100H
                         ASSUME CS:CDSEG, SS:CDSEG, DS:CDSEG
 0100                YAZDIR  PROC NEAR
 0100  8D 16 0109 R       LEA DX, MESAJ
 0104  B4 09                MOV AH, 09H
 0106  CD 21                INT 21H
 0108  C3                   RET
 0109                YAZDIR  ENDP
 0109  43 4F 4D 20 79 61 73 MESAJ DB 'COM yapsinda bir program calisti','$'
       70 73 69 6E 64 61 20
       62 69 72 20 70 72 6F
       67 72 61 6D 20 63 61
       6C 69 73 74 69 24
 012B                CDSEG   ENDS
                               END YAZDIR

Microsoft (R) Macro Assembler Version 5.00                  6/15/01
Ekrana mesaj yazma                                          Symbols-1


Segments and Groups:
                N a m e          Length    Align   Combine Class
CDSEG  . . . . . . . . . . . . .     012B PARA NONE 'CODE'

Symbols:
                N a m e          Type  Value    Attr
MESAJ  . . . . . . . . . . . . .     L BYTE   0109 CDSEG
YAZDIR . . . . . . . . . . . . .     N PROC   0100 CDSEG    Length = 0009
```

```
@FILENAME  . . . . . . . . . . .     TEXT  Örnek 9-6

     25 Source  Lines
     25 Total   Lines
      6 Symbols

  50992 + 383504 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

Hatasız derleme ve link işlemini takiben EXE uzantılı dosyadan COM uzantılı dosyaya dönüşüm işlemini gerçekleştirelim.

```
C:\assembly>exe2com Örnek 9-6.exe Örnek 9-6.com
exe2com 1.04 by Chris Dunford/The Cove Software Group
```

Gereksiz olan dosyaları silerek ikincil bellek alanımızı temizleyelim.

```
C:\assembly>Erase Örnek 9-6.exe
C:\assembly>Erase Örnek 9-6.map
C:\assembly>Erase Örnek 9-6.lst
C:\assembly>Erase Örnek 9-6.crf
```

Şimdi komut satırından programımızın ismini yazarak çalıştırabiliriz. Dosya uzantısının yazılmasına gerek yoktur[129].

```
C:\assembly>Örnek 9-6
C:\assembly>COM yapsinda bir program calisti
```

9.3 COM ve EXE Programlar Arasındaki Farklılıklar

Şu ana kadar EXE ve COM tipindeki, çalıştırılabilir (executable) olarak adlandırılan programlara has özellikler gerek iskelet yapıları üzerinden genel hatları ile gerek ise örnekler ile açıklanmaya çalışılmıştır. Bu bilgiler ışığında COM ve EXE yapısındaki programlar arasındaki farklılıklar Tablo 9-4'de özetlenmiştir.

[129] Aynı isimde hem EXE hem de COM uzantılı dosyalar aynı dizinde olsaydı öncelikli olarak COM uzantılı dosya devreye girerek çalıştırılmış olacaktı.

Tablo 9-4: COM ve EXE programların karşılaştırılması

	COM Program	EXE Program
Kullanılabilen bellek miktarı	64 KB - 256 Byte (PSP boyutu) - 2 Byte (dönüş göreli konun adresi)	Boş bellek ile sınırlı
Başlangıçta CS yazmacının değeri	PSP'nin bulunduğu kesim adresi	END komutunu takip eden etiket ile belirlenen kodun bulunduğu kesim adresi
Başlangıçta IP yazmacının değeri	0100H (ORG 100H)	Programın bitimini belirleyen END komutunu takip eden etiket ile belirlenen noktanın göreli konum değeri
Başlangıçta DS yazmacının değeri	PSP'nin bulunduğu kesim adresi	Dönüş için gerekli olan kesim adresi
Tanımlı değişkenlere erişmek için yapılması gereken işlemler	Yok	MOV AX, *verikesimi_ismi* MOV DS, AX
Değişkenlerin tanımlandığı yer	Kesim içinde JMP komutu ile ana yordam arasında veya ENDP komutundan sonra ENDS den önce olmak üzere iki ayrı şekilde	Veri kesimi
Başlangıçta ES yazmacının değeri	PSP'nin bulunduğu kesim adresi	PSP'nin bulunduğu kesim adresi
Başlangıçta SS yazmacının değeri	PSP'nin bulunduğu kesim adresi	Tanımlanan yığın kesiminin adresi
Başlangıçta SP yazmacının değeri	0FFFEH	Tanımlanan yığın kesiminin boyutu ile belirlenen değer
Başlangıçta yığın içeriği	0000H (dönüş adresi olarak programa başlandığında otomatik yerleştirilir)	Yığın boştur (Eğer başka bir program bloğu içinde çağırılmadıysa)
Yığın büyüklüğü	64 K - 256 Byte (PSP boyutu) - X Byte (kodun boyutu) - Y Byte (değişkenlerin boyutu)	Tanımlanan yığın kesiminin boyutu ile belirlenen değer
Ana yordam tipi	NEAR	FAR
Ana yordamdan dönebilmek için yapılması gerekenler	Yok	PUSH DS XOR AX, AX PUSH AX
Alt program çağırma şekli	NEAR	NEAR veya FAR
Programı bitirmek için	RET veya INT 21H fonksiyon 4CH	Eğer dönüş adresleri ana yordam içinde yığına atılmış ise RETF aksi hallerde INT 21H fonksiyon 4CH
Dosyanın ikincil bellek üzerindeki büyüklüğü	Programın byte cinsinden karşılığı kadar	Z Byte (Programın boyutu) + 512 Byte EXE header
Çalışma önceliği	EXE'den önce	COM'dan sonra

10. ASSEMBLY PROGRAMIN TEMEL TAŞLARI: YORDAM VE MAKRO

Programlama tekniği gereğince programı oluşturacak kodların mümkün olduğunca modüler yapıda tasarlanması ve buna göre kodlamanın yapılması gerekir. Giriş ve çıkış değerleri iyi tarif edilmiş modüller kullanarak program geliştirmek hem zaman hem de programın sürekliliğinin sağlanması ve özelliklerinin geliştirilebilmesi açısından son derece önemlidir. Bu amaçla assembly dilinde kodlama yaparken yordam ve makrolardan yararlanılmaktadır. Yordamlar ve makrolar assembly dilinin komut kümesi kullanılarak oluşturulmuş işlevsel kod parçalarıdır.

Daha önceki bölümlerde gerek EXE gerekse COM yapısındaki programların nasıl yazıldığını incelerken yordamlardan yararlandık, her program içerisinde en az bir tane ana yordam olması gerektiğini vurguladık. Ancak aynı tür bir genellemeyi makrolar için yapmak mümkün değildir. Yordam ve makro kullanımları arasında bazı farklılıklar, birbirlerine göre üstün ve zayıf oldukları yönler vardır. Bunları ayrı ayrı incelemeye çalışalım.

10.1 Yordam Özellikleri

Yordamlar **PROC** sözde komutu ile başlayıp **ENDP** sözde komutu ile sonlanan bir şekilde tanımlanmaktadır. Yordamların aynı kesim içinden (**intrasegment**) ve farklı kesim içerisinden (**intersegment**) kullanılabilmeleri için sırasıyla **NEAR** veya **FAR** olarak tanımlanmaları gereklidir. Yordamların çağırıldıkları yere dönebilmeleri için en son komut olarak **RET** (NEAR tanımlı yordamlarda) veya **RETF** (FAR tanımlı yordamlarda) kullanılmaktadır. Ancak assembly dili derleyicisi, FAR olarak tanımlı yordamlar RET ile sonlandırılsa bile bunu RETF olarak değerlendirecek ve gerekli düzenlemeleri kendi içinde gerçekleştirecektir. Bu nedenle yordam tanımının ne olduğuna bakılmaksızın RET ile yordamın sonlandırılması bir alışkanlık olmuştur. Yordamın tanımı en genel şekli ile aşağıda gösterilmiştir.

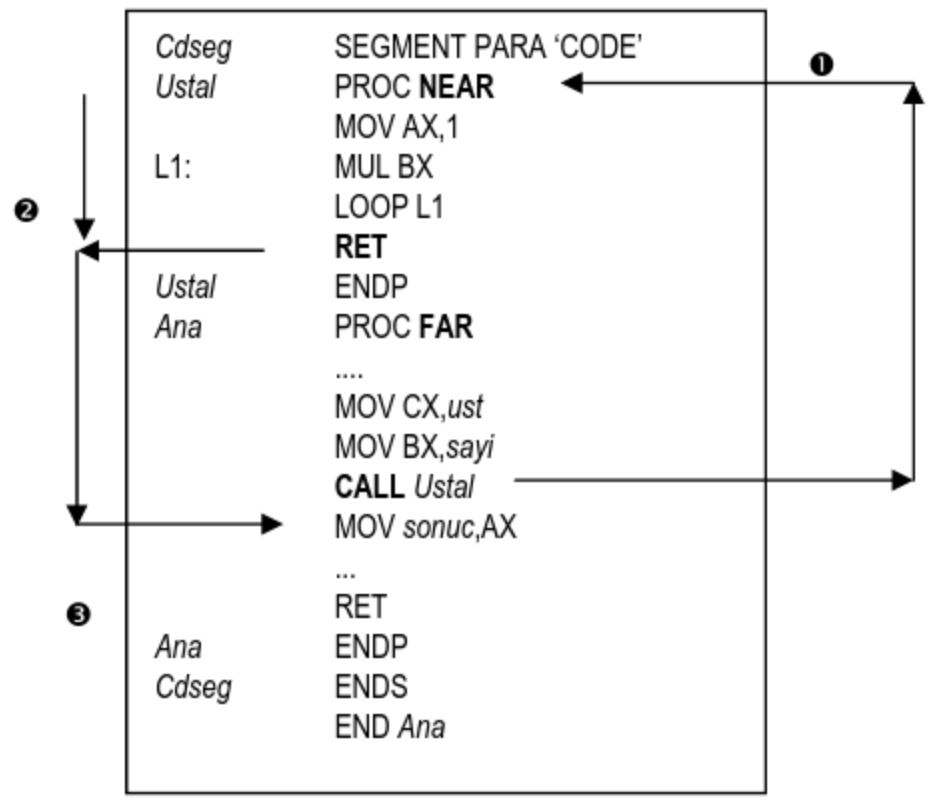
Tablo 10-1: Yordam tanımı

Yordam_ismi	**PROC** { NEAR | FAR }
	; yordamın kodu
	RET
Yordam_ismi	**ENDP**

10.1.1 Kesim İçi Yordam Kullanımı

Kesim için yordam çağırma (intrasegment call) olarak adlandırılan bu tip kullanımda çağıran ve çağırılan aynı kesim içinde yer alır. Bunu örnek üzerinden inceleyebilmek için aynı kesim içinde tanımlanmış iki yordama sahip olduğumuzu,

bunlardan *Ana* isimli olanının ilk olarak çalıştırılan (yani kontrolü elinde bulunduran yordam) olduğunu, *Ustal* isimli olanın ise *Ana* tarafında çağırılan ve NEAR olarak tanımlanmış bir yordam olduğunu düşünelim (Şekil 10-1).



Şekil 10-1: Kesim içindeki yordamın çağırılması (intrasegment call)

Çağırılan yordam, $sayi^{ust}$ (*sayı*'nın *ust* ile belirlenen kuvvetini hesaplayan) işlemini yapmakta ve **CALL** *Ustal* komutu yardımıyla çağırılmaktadır. Çağırma işlemiyle programın akışı *Ustal* isimli yordamın olduğu adrese aktarılmakta (❶), işlemler gerçekleştirildikten sonra (❷) RET komutunun çalıştırılmasıyla birlikte çağırma işleminin yapıldığı kod parçası içinde yer alan ve CALL komutunu takip eden satıra geri dönülmektedir. Böylece *Ana* isimli yordamın kaldığı noktadan işlemlere devam edebilmesi sağlanmaktadır (❸). Kesim içi çağırma işleminin gerçekleşmesi sırasında bize doğrudan işlem olarak görünmeyen, daha doğrusu kullanılan CALL komutunun gereği olarak otomatik olarak yerine getirilen bazı ara işlemler de vardır. Bu işlemlerin neler olduğu ve ne amaçla yapıldığını maddeler halinde açıklayalım;

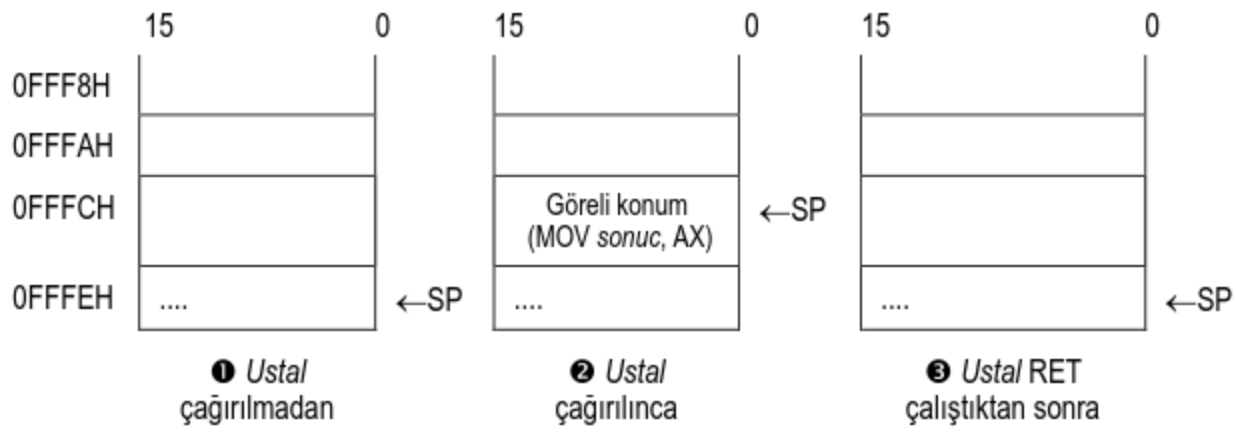
1. *Ana* isimli yordam içinde yer alan CALL komutu çalıştırıldığı zaman program kontrolünün *Ustal* isimli yordama verilmesi, onun işini bitirmesini takiben çalışan RET komutu ile tekrar *Ana* isimli yordama dönülmesi ve CALL satırının altındaki satırından (MOV *sonuc*, AX) işlemeye devam edilmesi gereklidir. O halde CALL komutunun dönüş adresini (bu adres CALL komutundan sonra yürütülmesi gereken komut satırının adresi olmalıdır) bir şekilde saklaması gereklidir. Dönüş adresini saklamak için yığından yararlanılır.

2. Yığında dönüş adresi nasıl saklanacaktır? Çağıran ve çağırılan yordamlar aynı kesim içinde oldukları ve çağırılan yordam NEAR olarak tanımlandığından, sadece göreli konum değerinin saklanması yeterli olacaktır (Şekil 10-2).

3. Kontrolün *Ustal* isimli yordama aktarılması ancak IP yazmacının *Ustal* yordamının kesim içi adresini göstermesi ile mümkün olacaktır. Bu nedenle IP←Göreli konum (*Ustal*) işlemi arka planda otomatik olarak gerçekleştirilir.

4. Artık kontrol, *Ustal* isimli yordamdadır ve onun komut satırları sırasıyla işlenecektir. RET komutuna gelindiği zaman dönüş yapılması gerektiği, bunun için daha önceden CALL komutunun yığında sakladığı dönüş adresinin kullanılacağı hatırlanır. Doğru dönüş adresi ancak SP yazmacının yığın üzerinde dönüş adreslerinin saklandığı bölgeyi göstermesi durumunda temin edilebilir. Eğer çağrılan *Ustal* isimli yordam içinde uygulamanın gereği olarak PUSH işlemi yapılmış ve yığın üzerinde bir takım veriler saklanmak istenmiş ise bu verilerin dönüş adresinin alınmasından önce yığından alınarak SP yazmacının doğru değerleri göstermesi sağlanmalıdır[130]. Aksi bir durum söz konusu olursa programın kontrolü bizim ne şekilde çalıştığını bilmediğimiz bir bellek alanına aktarılacak, kontrol kaybedilecek hatta bilgisayar sonsuz bir çevrim içine girerek kilitlenebilecektir (Şekil 10-2).

5. RET komutu içinde yer aldığı yordamın tipine (NEAR veya FAR olmasına) bağlı olarak yığından dönüş adresini ne şekilde alması gerektiğini tespit eder. NEAR tanımlı yordamdan dönebilmek için sadece göreli konum değerine ihtiyaç vardır. Buna göre yığından alınan değer IP yazmacına yerleştirilir ve dönülmesi gereken noktaya geri gelinmiş olur (Şekil 10-2).

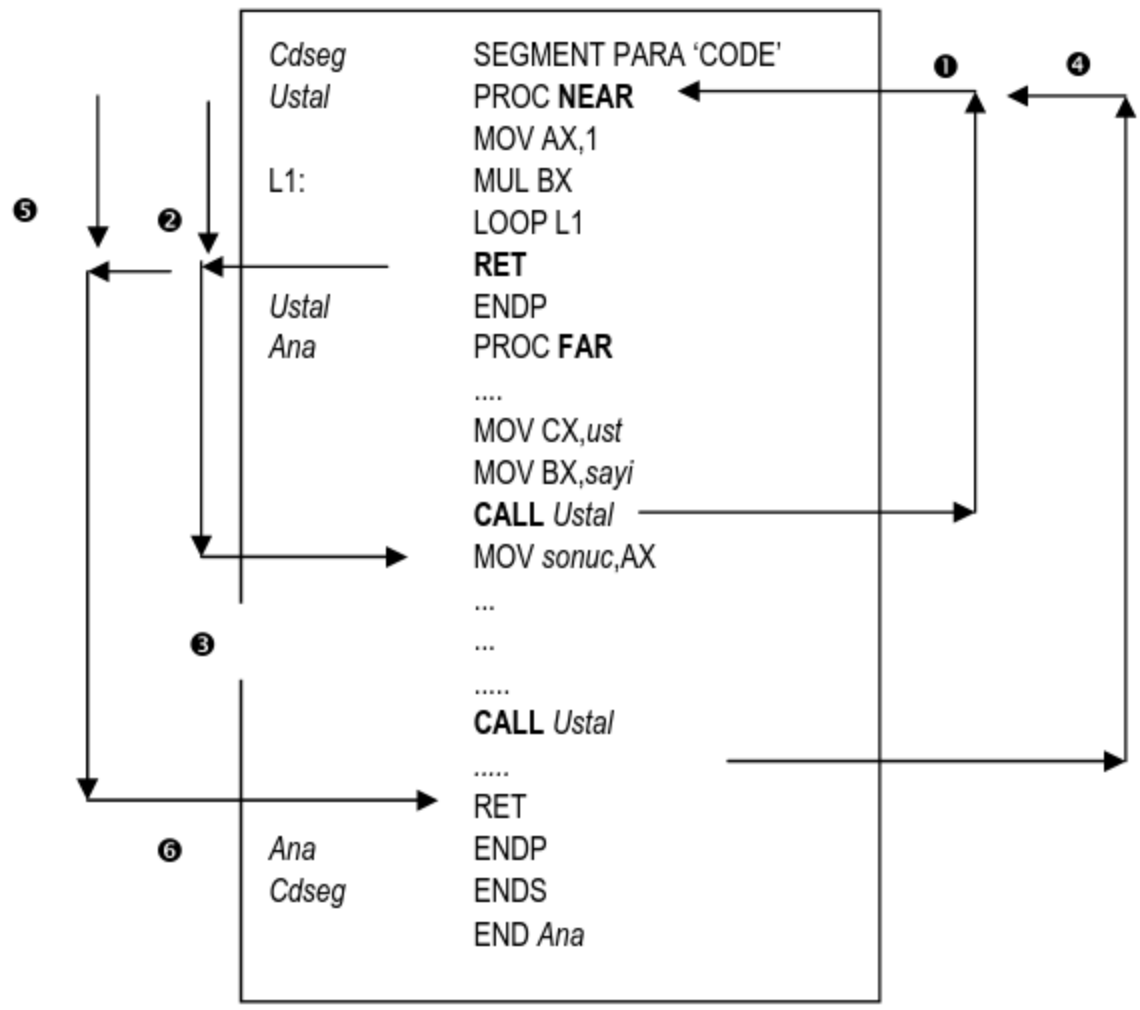


Şekil 10-2: Yordam kullanılırken yığının[131] durumu

[130] PUSH, matematik ifadedeki parantez açma, POP ise parantez kapama işlemine karşı gelir. Matematiksel ifadelerde açılan ve kapanan parantezlerin kullanımında uyulması gereken kurallar (uygun şekillerde ve aynı sayıda) assembly programlarda kullanılan PUSH ve POP işlemleri için de geçerlidir.

[131] 8086 işlemcisinde SP yazmacı yığın üzerinde bulunan dolu gözü gösterir. Oysa 80286 ve daha üst işlemcilerde SP yazmacı yığın üzerindeki boş olan gözü gösterir.

Yordamın aynı program içinde birden fazla yerde çağırılması durumundaki akış incelediğinde her çağırmada kontrolün doğrudan *Ustal* isimli yordamın olduğu adrese aktarıldığı, işlemlerin tamamlanmasını takiben RET komutu ile geri dönüldüğü görülür. Bu sayede yordamlar kaç kere çağırılırsa çağırılsın hep aynı adresten çalıştırıldıkları için programın kodunu büyütmez. Ancak her çağırma işlemi, geri dönebilmek için gerekli adres(ler)in yığına atılması, kontrolün aktarılacağı göreli konum değerinin IP yazmacına atanması ve işlemin bitimini takiben dönüş için gerekli adres değerinin yığından çekilmesi gibi işlemleri de içermesi nedeniyle ister istemez belli bir gecikmeye neden olmaktadır (Şekil 10-3).



Şekil 10-3: Aynı yordamın kod içinde birden fazla yerden çağırılması

Kesim içi yordam kullanımı konusundaki bilgilerimizi pekiştirmek ve durumu çalışan program parçaları üzerinden incelemek üzere Örnek 10-1'den yararlanalım.

Örnek 10-1

Şekil 10-1'de verilen ve bir sayının kuvvetini almayı sağlayan *USTAL* isimli yordamı EXE tipinde bir program içinde kullanalım. *USTAL* isimli yordam giriş verilerini BX ve CX yazmaçları üzerinden almakta, işlem sonucunda hesaplanan değeri

ise AX yazmacı üzerinde tutmaktadır[132]. Üst almak için çarpma işleminden yararlanılacaktır.

Programımızdaki *SAYI*, *UST* ve *SONUC* isimli değişkenler word olarak tanımlanmıştır. Burada *SAYI* ve *UST* değerleri büyük seçilecek olursa elde edilecek değer *SONUC* isimli değişkene sığmayacak kadar büyük olacaktır. Buna göre, programımızı test ederken giriş değerlerini özenle seçmek gerekecektir.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 10-1.ASM
; Kesim  içindeki yordamı çağıran EXE tipinde program
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Sayinin kuvvetini hesaplar
STACKSG SEGMENT PARA STACK 'STACK'
        DW 10 DUP(?)
STACKSG ENDS
DATASG  SEGMENT PARA 'DATA'
;---------------------------------------------------------------------
; Kullanacağımız değişkenler
;---------------------------------------------------------------------
SAYI    DW 2
UST     DW 10
SONUC   DW 0
DATASG  ENDS
CODESG  SEGMENT PARA 'CODE'
        ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
ANA     PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        SUB AX, AX
        PUSH AX
;---------------------------------------------------------------------
; DATASG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DATASG
        MOV DS, AX
;---------------------------------------------------------------------
; işleminin gerçekleştirildiği kod bloğu
;---------------------------------------------------------------------
        MOV CX, UST     ; UST ve SAYI parametreleri yazmaçlarda
        MOV BX, SAYI
        CALL USTAL      ; Yordam çağırma
        MOV SONUC, AX   ; İşlemin sonucu AX yazmacı üzerinden aktarılıyor
        RET             ; dön
ANA     ENDP
;---------------------------------------------------------------------
; üst alma işlemini gerçekleştiren yordam
;---------------------------------------------------------------------
USTAL   PROC NEAR
        MOV AX, 1
L1:     MUL BX          ; çarpma işlemi AX üzerinden yapılıyor
        LOOP L1
        RET             ; çağırıldığın noktaya dön
USTAL   ENDP
CODESG  ENDS
        END ANA         ; programın başlangıç noktası
```

[132] Çağıran ve çağırılan yordamların aynı veri kesimini paylaştıkları düşünülecek olursa giriş ve çıkış verilerinin bu ortak alandan tanımlanması ve paylaşılması mümkündür. Ancak örneğimizde yazmaç kullanarak parametre aktarımı yapılacaktır.

Programımızı derleyip EXE uzantılı dosyayı elde ettikten sonra DEBUG programı yardımıyla program akışını inceleyelim.

```
c:\assembly>debug örnek 10-1.exe

-u
1C5E:0000 1E            PUSH    DS
1C5E:0001 2BC0          SUB     AX, AX
1C5E:0003 50            PUSH    AX
1C5E:0004 B85D1C        MOV     AX, 1C5D
1C5E:0007 8ED8          MOV     DS, AX
1C5E:0009 8B0E0200      MOV     CX, [0002]
1C5E:000D 8B1E0000      MOV     BX, [0000]
1C5E:0011 E80400        CALL    0018
1C5E:0014 A30400        MOV     [0004], AX
1C5E:0017 CB            RETF
1C5E:0018 B80100        MOV     AX, 0001
1C5E:001B F7E3          MUL     BX
1C5E:001D E2FC          LOOP    001B
1C5E:001F C3            RET
```

Tablo 10-2: Örnek 10-1'de yazılan programın adım adım çalıştırılması

	Kısa Açıklama	DEBUG programının ürettiği değerler
1	Programın dönüş adresleri yığına yerleştiriliyor. Yığın 10 word tanımlandığı için SP yazmacının ilk değeri 0014H (20) olarak görülmektedir. Her PUSH işlemi ile bu sayı 2 birim azalır.	AX=0000 BX=0000 CX=0050 DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=**1C4B** ES=1C4B SS=1C5B CS=1C5E IP=0000 NV UP EI PL ZR NA PE NC 1C5E:0000 1E **PUSH DS**
2		AX=0000 BX=0000 CX=0050 DX=0000 SP=0012 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5E IP=0001 NV UP EI PL ZR NA PE NC 1C5E:0001 2BC0 **SUB AX,AX**
3		AX=**0000** BX=0000 CX=0050 DX=0000 SP=0012 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5E IP=0003 NV UP EI PL ZR NA PE NC 1C5E:0003 50 **PUSH AX**
4	Yığına yerleştirilen DS ve AX yazmaçlarının değerleri görülmektedir.	-d ss:0000 1C5B:0000 00 00 00 00 00 00 00 00-00 00 04 00 5E 1C AC 15 1C5B:0010 **00 00 4B 1C**
5	Veri kesimine erişebilmek için gerekli tanımlar yapılıyor.	AX=0000 BX=0000 CX=0050 DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5E IP=0004 NV UP EI PL ZR NA PE NC 1C5E:0004 B85D1C **MOV AX,1C5D**
6		AX=1C5D BX=0000 CX=0050 DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5E IP=0007 NV UP EI PL ZR NA PE NC 1C5E:0007 8ED8 **MOV DS,AX**
7	CX ve BX yazmaçlarına giriş değerleri atanıyor.	AX=1C5D BX=0000 CX=0050 DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=0009 NV UP EI PL ZR NA PE NC 1C5E:0009 8B0E0200 **MOV CX,[0002] DS:0002=000A**
8		AX=1C5D BX=0000 CX=000A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=000D NV UP EI PL ZR NA PE NC 1C5E:000D 8B1E0000 **MOV BX,[0000] DS:0000=0002**
9	Veri kesiminin durumu.	-d ds:0 1C5D:0000 **02 00 0A 00** 00 00 00 00-00 00 00 00 00 00 00 00
10	*Ustal* isimli yordam çağırılıyor. Dönüş adresi olarak MOV [0004], AX komutunun göreli konum değeri (0014H) yığına konacaktır.	AX=1C5D BX=0002 CX=000A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=0011 NV UP EI PL ZR NA PE NC 1C5E:0011 E80400 **CALL 0018**

11	Dönüş adresi yığına yerleştirildikten sonra yığının durumu	-d ss:0 1C5B:0000 00 00 00 00 5D 1C 00 00-18 00 5E 1C AC 15 **14 00** 1C5B:0010 00 00 4B 1C
12	Üst alabilmek için CX yazmacı sıfır değerine gelene kadar işlemler tekrarlanır.	AX=1C5D BX=0002 CX=000A DX=0000 **SP=000E** BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=0018 NV UP EI PL ZR NA PE NC 1C5E:0018 B80100 **MOV AX,0001**
13		AX=0001 BX=0002 CX=000A DX=0000 SP=000E BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=001B NV UP EI PL ZR NA PE NC 1C5E:001B F7E3 **MUL BX**
14		AX=0002 BX=0002 CX=000A DX=0000 SP=000E BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=001D NV UP EI PL ZR NA PE NC 1C5E:001D E2FC **LOOP 001B**
15	Dönüş adresi SP yazmacının gösterdiği adresten alınır.	AX=0400 BX=0002 CX=0000 DX=0000 **SP=000E** BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=001F NV UP EI PL ZR NA PE NC 1C5E:001F C3 **RET**
16	Dönüş adresi yığından alınırken yığının durumu.	-d ss:0 1C5B:0000 00 00 00 00 5D 1C 00 00-18 00 5E 1C AC 15 **14 00** 1C5B:0010 00 00 4B 1C
17	Ana yordama dönüldükten sonra. AX yazmacındaki değer sonuç değişkenine yerleştiriliyor.	AX=0400 BX=0002 CX=0000 DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=0014 NV UP EI PL ZR NA PE NC 1C5E:**0014** A30400 **MOV [0004],AX DS:0004=0000**
18	Veri kesiminin durumu.	-d ds:0 1C5D:0000 02 00 0A 00 **00 04** 00 00-00 00 00 00 00 00 00 00
19	FAR yordam olduğu için dönüş adresi 4 byte'tır. SP yazmacının belirlediği yerden çekilerek CS:IP ikilisine yerleştirilir.	AX=0400 BX=0002 CX=0000 DX=0000 **SP=0010** BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=0017 NV UP EI PL ZR NA PE NC 1C5E:0017 CB **RETF**
20	Dönüş adresi yığından alınırken yığının durumu	-d ss:0 1C5B:0000 00 00 00 00 5D 1C 00 00-18 00 5E 1C AC 15 14 00 1C5B:0010 **00 00 4B 1C**
21	CS←1C4BH IP←0000H yapıldı ve program sonlandı.	AX=0400 BX=0002 CX=0000 DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C4B IP=0000 NV UP EI PL ZR NA PE NC **1C4B:0000** CD20 **INT 20**

10.1.2 Kesimler Arası Yordam Kullanımı

Kullanmak istediğimiz yordamlar her zaman aynı kesim içinde bulunmayabilir. Bu durumda kesim dışındaki bir yordama erişmek ve onu kullanmak ihtiyacı doğabilir. Bulunulan kesim dışındaki bir yordamın çağırılarak kullanılmasına kesimler arası çağırma (intersegment call) adı verilir. Kesimler arası yordamların kullanımının iki önemli nedeni vardır. Bunlar;

1. Bir kesim ile sınırlı kalmaksızın programımızın kodunu istediğimiz kadar büyütebilmek
2. Başka birisi tarafından yazılmış, test edilmiş giriş ve çıkış değerleri belli bir yordamdan yararlanabilmek

Her ne kadar bu nedenler birbirinden farklı gibi görünse de genel olarak modüler program yazma ihtiyacının doğal bir sonucu olarak ortaya çıkmaktadırlar. Kullanmak istediğimiz yordamın farklı bir kesimde yer aldığının derleyiciye bir şekilde belirtilmesi gerekmektedir. Aksi halde derleyici çağırılan yordamı aynı kod kesimi içinde arayacak, adı geçen yordamı bulamayınca da hata verecektir. Bu problemi

aşmak üzere çağıran tarafta çağırılan yordamın ismi ve tipi **EXTRN**, çağırılan tarafta ise yordamın ismi **PUBLIC** sözde komutlarından sonra yazılmalıdır.

Kesim içi yordam çağırmadan farklı olarak kesimler arası yordam çağırmada, çağıran ve çağırılan yordamlar aynı veri kesimini paylaşmazlar. Bunun için aralarında parametre aktarımı yapmaları gerekir. Parametre aktarımı için aktarılacak parametre sayısına bağlı olarak farklı yöntemler[133] kullanılmalıdır.

Bu durumu kesimler arası yordam çağırma işlemine vereceğimiz bir örnek üzerinden incelemeye çalışalım.

Örnek 10-2

Öncelikle byte tipinde iki değeri BL ve BH yazmaçları üzerinden alan ve bunların toplamını hesaplayarak AX yazmacı üzerinden döndüren *TOPLAMA* isminde bir yordamın var olduğunu varsayarak bu yordamdan servis alacak olan programı yazalım. *TOPLAMA* ismindeki yordamın giriş parametreleri byte olduğu için *SAYI1* ve *SAYI2* isimli değişkenler byte, toplam değeri AX yazmacı üzerinden döndüğü için *SONUC* değeri word olarak veri kesiminde tanımlanmalıdır.

Bu örneğimizde **EXTRN** komutunun kullanılmaması durumunda ne tip hata mesajları ile karşılaşılacağını inceleyeceğiz. Alınan hata mesajları doğrultusunda kodumuza EXTRN tanımını ilave ederek hemen ardından *TOPLAMA* isimli yordamı barındıran kodu yazacağız. Son olarak iki farklı assembly kaynak kodundan ne şekilde çalıştırılabilir bir nesne kodu üretileceğini göreceğiz

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 10-2.ASM
; Toplama isimli yardımcı yordamın çağırılması
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE yardımcı yordam çağırma
STSEG   SEGMENT PARA STACK 'STACK'
        DW 12 DUP(0)
STSEG   ENDS
DTSEG   SEGMENT PARA 'DATA'
;---------------------------------------------------------------------
; Kullanılacak değişkenler tanımlanır.
;---------------------------------------------------------------------
SAYI1   DB 12H
SAYI2   DB 78H
SONUC   DW 0H
DTSEG   ENDS
CDSEG   SEGMENT PARA 'CODE'
        ASSUME CS:CDSEG, SS:STSEG, DS:DTSEG
BASLA   PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        XOR AX, AX
        PUSH AX
```

[133] Aktarılacak parametre sayısına bağlı olarak yazmaç, yığın veya ortak veri kesimi kullanılır. Parametre aktarma ile ilgili ayrınlar için 12. Bölümü inceleyiniz.

```
;-----------------------------------------------------------------------
; DATASG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;-----------------------------------------------------------------------
        MOV AX, DTSEG
        MOV DS, AX
;-----------------------------------------------------------------------
; işlem bloğu başlıyor
;-----------------------------------------------------------------------
        MOV BH, SAYI1     ; Değerler BH ve BL Yazmaçları üzerinden
        MOV BL, SAYI2     ; çağırılacak olan yardımcı yordama aktarılır
        CALL TOPLAMA      ; yordam çağırılır
        MOV SONUC, AX     ; toplama işleminin sonucunu AX yazmacında
        RET
BASLA   ENDP
CDSEG   ENDS
        END BASLA        ; programın başlangıç noktası
```

Bu programımızı derlediğimiz zaman gelen mesajları inceleyim.

```
C:\assembly>masm Örnek 10-2.asm;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Örnek 10-2. asm(37): error A2009: Symbol not defined: TOPLAMA

51534 + 382962 Bytes symbol space free

      0 Warning Errors
      1 Severe  Errors
```

Hemen görüleceği gibi programın 37. satırında, *TOPLAMA* isimli sembolün tanımsız olduğuna ilişkin bir hata mesajı alıyoruz. Bu hata, *TOPLAMA* isimli yordamın kod kesimi içinde yer almaması veya bu yordamın harici bir kesimde olduğunun belirtilmemesinden kaynaklanmaktadır. **EXTRN** sözde komutu programımız içinde atıfta bulunacağımız herhangi bir sembolün (bu bir yordam ismi olabileceği gibi bir değişken ismi de olabilir) bizim programımızın dışındaki bir yerde tanımlı olduğunu derleyiciye bildirmek için kullanılmaktadır. Benzer şekilde **PUBLIC** sözde komutu da sembolün tanımlandığı noktada bu sembole dışarından bir erişim olacağını derleyiciye belirtmek amacıyla kullanılır. Buna göre; Örnek 10-2.ASM isimli programın başına;

```
EXTRN TOPLAMA:FAR
```

yazılacak olursa derleyici *TOPLAMA* isimli sembolün kesim dışında tanımlı ve FAR özellikte bir yardımcı yordam olduğunu anlayacaktır. EXTRN sözde konutunda kullanılan FAR ifadesi daha önceden de belirtildiği gibi dönüş adreslerinin kesim ve göreli konum ikilisinden oluştuğunu belirler. Buna göre programımıza EXTRN satırını ekleyerek tekrar derleyelim;

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 10-2.ASM
; Toplama isimli yardımcı yordam çağırılması
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE yardımcı yordam çağırma
        EXTRN TOPLAMA:FAR
STSEG   SEGMENT PARA STACK 'STACK'
        DW 12 DUP(0)
STSEG   ENDS
DTSEG   SEGMENT PARA 'DATA'
;---------------------------------------------------------------------
; kullanılacak değişkenler tanımlanır.
;---------------------------------------------------------------------
SAYI1   DB 12H
SAYI2   DB 78H
SONUC   DW 0H
DTSEG   ENDS
CDSEG   SEGMENT PARA 'CODE'
        ASSUME CS:CDSEG, SS:STSEG, DS:DTSEG
BASLA   PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        XOR AX, AX
        PUSH AX
;---------------------------------------------------------------------
; DATASG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DTSEG
        MOV DS, AX
;---------------------------------------------------------------------
; işlem bloğu başlıyor
;---------------------------------------------------------------------
        MOV BH, SAYI1     ; Değerler BH ve BL yazmaçları üzerinden
        MOV BL, SAYI2     ; çağırılacak olan yardımcı yordama aktarılır
        CALL TOPLAMA      ; yordam çağırılır
        MOV SONUC, AX     ; toplama işleminin sonucunu AX yazmacında
        RET
BASLA   ENDP
CDSEG   ENDS
        END BASLA       ; programın başlangıç noktası

C:\assembly>masm Örnek 10-2.asm;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

  51534 + 382962 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors


Microsoft (R) Macro Assembler Version 5.00                  6/21/01
yardımcı yordam çağıma                                  Page     1-1

                        PAGE 60,80
                        TITLE yardımcı yordam çağırma
                        EXTRN TOPLAMA:FAR
 0000               STSEG   SEGMENT PARA STACK 'STACK'
 0000  000C[                    DW 12 DUP(0)
    0000
           ]

 0018               STSEG   ENDS
 0000               DTSEG   SEGMENT PARA 'DATA'
 0000  12           SAYI1   DB 12H
 0001  78           SAYI2   DB 78H
 0002  0000         SONUC   DW 0H
```

```
0004                DTSEG   ENDS
0000                CDSEG   SEGMENT PARA 'CODE'
                       ASSUME CS:CDSEG, SS:STSEG, DS:DTSEG
0000                BASLA   PROC FAR
0000  1E                    PUSH DS
0001  33 C0                 XOR AX, AX
0003  50                    PUSH AX
0004  B8 ---- R             MOV AX, DTSEG
0007  8E D8                 MOV DS, AX
0009  8A 3E 0000 R        MOV BH, SAYI1
000D  8A 1E 0001 R        MOV BL, SAYI2
0011  9A 0000 ---- E        CALL TOPLAMA
0016  A3 0002 R             MOV SONUC, AX
0019  CB                    RET
001A                BASLA   ENDP
001A                CDSEG   ENDS
                              END BASLA

Microsoft (R) Macro Assembler Version 5.00                  6/21/01
yardımcı yordam çağırma                                   Symbols-1

Segments and Groups:
                N a m e          Length    Align   Combine Class
CDSEG  . . . . . . . . . . . . .     001A PARA NONE 'CODE'
DTSEG  . . . . . . . . . . . . .     0004 PARA NONE 'DATA'
STSEG  . . . . . . . . . . . . .     0018 PARA STACK 'STACK'

Symbols:
                N a m e          Type  Value    Attr
BASLA  . . . . . . . . . . . . .     F PROC    0000 CDSEG    Length = 001A
SAYI1  . . . . . . . . . . . . .     L BYTE    0000 DTSEG
SAYI2  . . . . . . . . . . . . .     L BYTE    0001 DTSEG
SONUC  . . . . . . . . . . . . .     L WORD    0002 DTSEG
TOPLAMA  . . . . . . . . . . . .     L FAR     0000     External
@FILENAME  . . . . . . . . . . .     TEXT  Örnek 10-2
```

Örnek 10-2. LST isimli dosya incelendiğinde bazı dikkat çekici özellikler görülmektedir. Bunlar;

1. Bazı atıflar bağıl adresler içermektedir. Bunları belirlemek üzere **R** harfi kullanılmıştır. Harfin önündeki sayı ise göreli konum değerini göstermektedir.

   ```
   0009  8A 3E 0000 R        MOV BH, SAYI1
   000D  8A 1E 0001 R        MOV BL, SAYI2
   ```

2. Bazı atıflar **E** harfi kullanılarak yapılmıştır. **E** harfi erişilmek istenen sembolün harici (external) bir değer olduğunu belirlemektedir. E harfinin önünde yer alan `----` sembolü ise kesim adresin henüz belli olmadığını, ancak link işleminden geçip programlar belleğe yerleştirildiğinde kesim adresinin bu noktaya yazılacağını göstermektedir. Bu kesim adresi içinde çağırılacak yordamın göreli konum değeri ise 0000 olarak belirlenmiştir.

   ```
   0011  9A 0000 ---- E        CALL TOPLAMA
   ```

EXTRN tanımı sayesinde derleme aşamasında karşılaşılan hata ortadan kalkmıştır. Bu aşamadan sonra *TOPLAMA* isimli yordamın içinde bulunacağı diğer programı da yazmamız gerekmektedir. Bu programda dikkat edilmesi gereken önemli nokta *TOPLAMA* yordamının çağırılış şekline uygun olarak, **FAR** tanımlanmasıdır. An-

cak *TOPLAMA* yordamını bünyesinde bulunduran programda da yordamın başka programlar tarafından kullanılmasına imkân veren **PUBLIC** sözde komutu kullanılmalıdır. Bu kurallar uyarınca *TOPLAMA* yordamının içinde bulunduğu programı yazacak olursak aşağıdaki kodu elde ederiz.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 10-2-1.ASM
; Toplama isimli yardımcı yordamın içinde bulunduğu program
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE yardımcı procedure çağırma
        PUBLIC  TOPLAMA
CODES   SEGMENT PARA 'CODE'
        ASSUME CS:CODES
TOPLAMA PROC FAR
;-----------------------------------------------------------------------
; işlem bloğu başlıyor
;---------------------------------------------------------------------
        XOR AX, AX       ; AX yazmacını sıfırla
        ADD AL, BL       ; AL üzerinde toplamaya başla
        ADD AL, BH
        ADC AH, 0        ; CF oluşması durumunda bunu AH'ya aktar
        RET              ; çağırıldığın yere don
TOPLAMA ENDP
CODES   ENDS
        END
```

Örnek 10-2-1.ASM isimli dosya da alışa geldiğimiz kodlamanın dışında kalan bazı özelliklerin vurgulanması gereklidir. Bunlar;

1. Yığın ve veri kesim tanımlamaları yapılmamıştır. Yığın, *TOPLAMA* yordamını çağıran program tarafında tanımlandığı için tekrar tanımlanmasına gerek yoktur. Ayrıca *TOPLAMA* yordamı içinde ayrı bir yığın tanımı yapılacak olursa dönüş adreslerine erişim mümkün olamayacaktır. Benzer biçimde *TOPLAMA* yordamının içinde bulunduğu programa has değişken de olmadığı için veri kesimi tanımlanmamıştır.[134]

2. EXE tipindeki programlarda kontrolü elinde bulunduran yordam **FAR** olarak tanımlanırken, geri dönüşü sağlamak üzere gerekli adresleri yığına yerleştirilebilmek için;

   ```
   PUSH DS
   XOR AX, AX
   PUSH AX
   ```

   komutları, ardından veri kesimine erişebilmek için;

   ```
   MOV AX, veri_kesim_ismi
   MOV DS, AX
   ```

[134] Ayrı bir veri kesimi tanımlaması yapılmadığı için DS yazmacında bulunan değer hali hazırda yordamı çağıran program bloğunda tanımlı olan veri kesimini göstermektedir. Bu durumda programcı her iki program parçasının aynı veri kesimini paylaştığı hissine kapılarak veriye yazmaç dolayı adresleme teknikleriyle erişmek isteyebilir. Oysa ortak veri kesimi tanımı yapılmadığı sürece bu tür erişimin kullanılması sağlıklı olmayıp kullanılmaması tavsiye edilir.

komutları kullanılmalıdır. Oysa bu komut satırları Örnek 10-2-1.ASM içinde bulunmaz. Bunun arkasında yatan neden *TOPLAMA* isimli yordamının kendi başına çalışacak, yani kontrolü elinde bulunduracak türde olmamasıdır. O zaman dönüş adreslerinin (işletim sistemine dönmek için) yığında saklanması, *TOPLAMA* isimli yordamı çağıranın sorumluluğunda olmalıdır.

3. Programın bitişini belirleyen **END** sözde komutunun sonunda başlangıç noktası olarak hiçbir yordam isminin yazılmamış olması da dikkat çekicidir. Örnek 10-2-1.ASM isimli program hiçbir zaman kendi başına çalışmak üzere tasarlanmadığı için başlangıç noktası olarak bir yordam ismi yazmaya gerek yoktur.

4. *TOPLAMA* yordamının başka programlar tarafından da çağırılabilmesi için

   PUBLIC TOPLAMA

   sözde komutundan yararlanılmıştır.

Örnek 10-2-1.ASM isimli programı tekrar derleyip durumu inceleyelim;

```
C:\assembly>masm Örnek 10-2-1;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

  51578 + 382918 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors

Microsoft (R) Macro Assembler Version 5.00                  6/21/01
yardımcı yordam çağırma                                   Page     1-1

                        PAGE 60,80
                        TITLE yardımcı yordam çağırma
                        PUBLIC  TOPLAMA
 0000           CODES   SEGMENT PARA 'CODE'
                        ASSUME CS:CODES
 0000           TOPLAMA PROC FAR
 0000  33 C0            XOR AX, AX
 0002  02 C3            ADD AL, BL
 0004  02 C7            ADD AL, BH
 0006  80 D4 00        ADC AH, 0
 0009  CB               RET
 000A                TOPLAMA ENDP
 000A                CODES   ENDS
                             END

Segments and Groups:
                N a m e          Length    Align   Combine Class
CODES  . . . . . . . . . . . .      000A PARA NONE 'CODE'

Symbols:
                N a m e          Type  Value    Attr
TOPLAMA  . . . . . . . . . . . F PROC   0000 CODES    Global   Length = 000A
@FILENAME  . . . . . . . . . . .     TEXT  Örnek 10-2-1
```

Görüldüğü gibi Örnek 10-2.ASM isimli program, Örnek 10-2-1.ASM isimli program içinde yer alan bir yordamı kullanmak istemektedir. Programlarda kullanılan **EXTRN/PUBLIC** sözde komutları yardımıyla kimin, kimden hangi yordamı kulla-

nacağı belirlenmiş, derleme işlemleri sonucunda Örnek 10-2.OBJ ve Örnek 10-2-1.OBJ isimli nesne kodları elde edilmiştir. Bundan sonrası Link programının sorumluluğundadır. Link programı Örnek 10-2.OBJ içinde EXTRN tanımı yardımıyla harici *TOPLAMA* yordamına olan atfı, Örnek 10-2-1.ASM içinde *TOPLAMA* yordamının başkaları tarafından kullanılmasına imkân sağlayan PUBLIC tanımı ile eşleştirerek farklı bir kesim içinde olan yordamın kullanılmasını sağlayacaktır[135]. Bunun için yapılması gereken Örnek 10-2.OBJ ile Örnek 10-2-1.OBJ nesne kodlarının aşağıdaki komut satırında olduğu gibi + sembolü kullanılarak link programından geçirilmesidir.

```
C:\assembly>Link örnek 10-2.obj + örnek 10-2-1.obj

Microsoft (R) Overlay Linker  Version 3.61
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [örnek 10-2.EXE]: örnek 10-2.exe
List File [NUL.MAP]: örnek 10-2.map
Libraries [.LIB]:

C:\assembly>dir örnek 10-2.exe

örnek 10-2   EXE          602  21/06/01  20:46
```

Link işleminin hatasız olarak tamamlanması ile Örnek 10-2.EXE isimli çalıştırılabilir bir kod elde edilmiştir. Buna ek olarak **MAP** uzantılı bir dosya daha oluşturulmuştur[136]. Bu dosya da her iki programda tanımlı kesimlerin yerleşimlerine ait bilgiler bulunmaktadır. MAP uzantılı dosya aşağıdaki şekilde oluşmuş olacaktır.

```
Start  Stop   Length Name                    Class
00000H 00017H 00018H STSEG                   STACK
00020H 00023H 00004H DTSEG                   DATA
00030H 00049H 0001AH CDSEG                   CODE
00050H 00059H 0000AH CODES                   CODE

Program entry point at 0003:0000
```

Bu dosyada da her iki programda da tanımlanmış kesimlerin hangi adresten başladığı, hangi adreste bittiği, ne büyüklükte yer kapladığı ve kesimlerin sınıfları (class) belirlenmektedir. Bunlardan ilk üç tanesi Örnek 10-2.ASM ve sonuncusu da Örnek 10-2-1.ASM isimli programlarda tanımlanmış olan kesimlerdir. Kısaca bu

---

[135] Ayrı ayrı derleme sonucunda elde edilen OBJ uzantılı dosyaların Link programı ile ilişkilendirilmesiyle oluşan çalıştırılabilir kod, yordamların bellekte yerleştirildikleri adreslerin çağırma sırasındakullanılabilmesine olanak verecek şekilde düzenlenmiştir. Programlar belleğe çalıştırılmak üzere yükleninceye kadar yordamların hangi adreste yer alacağı belirli değildir. Ayrıca oluşturulan çalıştırılabilir kod belleğe her yüklendiğinde farklı bir bellek adresi üzerinde yer alabileceğinden çağıran ile çağırılan yordamların arasındaki adres alışverişinin yükleme işleminden (loading) sonra yapılması gereklidir. Bu özellik dikkate alındığında oluşturulan koda yeri tekrar belirlenebilen (re-allocatable) kod adı da verilmektedir.

[136] Derleme (MASM) ve ilişkilendirme (LINK) aşamalarında komut satırının sonuna ";" işareti kullanılmadan komut çalıştırılacak olursa kullanıcıya farklı uzantılı (LST, MAP, CRF gibi) yardımcı dosyaların üretilmesini isteyip istemediği sorulacaktır.

program bellekte dört farklı kesim işgal etmektedir. Yapılacak işlemlerin gereği olarak bu kesim sayıları daha da artabilir.

Çalıştırılabilir bir kod elde edildiğine göre bunun işleyişini debug programı yardımıyla inceleyelim;

```
c:\assembly>debug örnek 10-2.exe

-u
1C5E:0000 1E            PUSH    DS
1C5E:0001 33C0          XOR     AX, AX
1C5E:0003 50            PUSH    AX
1C5E:0004 B85D1C        MOV     AX, 1C5D
1C5E:0007 8ED8          MOV     DS, AX
1C5E:0009 8A3E0000      MOV     BH, [0000]
1C5E:000D 8A1E0100      MOV     BL, [0001]
1C5E:0011 9A0000601C    CALL    1C60:0000
1C5E:0016 A30200        MOV     [0002], AX
1C5E:0019 CB            RETF
```

Tablo 10-3: Örnek 10-2. EXE programın adım adım çalıştırılması

	Kısa Açıklama	DEBUG programının ürettiği değerler
1	Programın dönüş adresleri yığına yerleştiriliyor. Yığın 12 word tanımlandığı için SP yazmacının ilk değeri 0018H (24) olarak belirlenmiştir. Her PUSH işlemi ile bu sayı iki birim azalacaktır.	AX=0000 BX=0000 CX=005A DX=0000 **SP=0018** BP=0000 SI=0000 DI=0000 DS=**1C4B** ES=1C4B SS=1C5B CS=1C5E IP=0000 NV UP EI PL NZ NA PO NC 1C5E:0000 1E **PUSH DS**
2		AX=0000 BX=0000 CX=005A DX=0000 SP=0016 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5E IP=0001 NV UP EI PL NZ NA PO NC 1C5E:0001 33C0 **XOR AX,AX**
3		AX=**0000** BX=0000 CX=005A DX=0000 SP=0016 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5E IP=0003 NV UP EI PL ZR NA PE NC 1C5E:0003 50 **PUSH AX**
4	Yığına yerleştirilen DS ve AX yazmaçlarının değerleri görülmektedir..	-d ss:0000 1C5B:0000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 04 00 1C5B:0010 5E 1C AC 15 **00 00 4B 1C**
5	Veri kesimine erişebilmek için gerekli tanımlar yapılıyor.	AX=0000 BX=0000 CX=005A DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5E IP=0004 NV UP EI PL ZR NA PE NC 1C5E:0004 B85D1C **MOV AX,1C5D**
6		AX=1C5D BX=0000 CX=005A DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5E IP=0007 NV UP EI PL ZR NA PE NC 1C5E:0007 8ED8 **MOV DS,AX**
7	BH←[0000H], *SAYI1* değişkeni (12H) BH yazmacına alınıyor.	AX=1C5D BX=0000 CX=005A DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=0009 NV UP EI PL ZR NA PE NC 1C5E:0009 8A3E0000 **MOV BH,[0000] DS:0000=12**
8	BL←[0001H], *SAYI2* değişkeni (78H) BL yazmacına alınıyor	AX=1C5D BX=**12**00 CX=005A DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=000D NV UP EI PL ZR NA PE NC 1C5E:000D 8A1E0100 **MOV BL,[0001] DS:0001=78**
9	Veri kesiminin durumu.	-d ds:0 1C5D:0000 **12 78** 00 00 00 00 00 00-00 00 00 00 00 00 00 00

10	*TOPLAMA* isimli yordam 1C60H:0000H adresinde bulunmaktadır. Bu adresten çağırılacaktır.	AX=1C5D BX=12**78** CX=005A DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B **CS=1C5E IP=0011** NV UP EI PL ZR NA PE NC 1C5E:0011 9A0000601C **CALL 1C60:0000**
11	Dönüş adresi yığına yerleştirildikten sonra yığının durumu	-d ss:0 1C5B:0000 00 00 00 00 00 00 5D 1C-00 00 00 00 60 1C AC 15 1C5B:0010 **16 00 5E 1C** 00 00 4B 1C-
12	*TOPLAMA* yordamına ait işlemler gerçekleştiriliyor.	AX=1C5D BX=1278 CX=005A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C60 IP=0000 NV UP EI PL ZR NA PE NC 1C60:0000 33C0 **XOR AX,AX**
13		AX=0000 BX=1278 CX=005A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C60 IP=0002 NV UP EI PL ZR NA PE NC 1C60:0002 02C3 **ADD AL,BL**
14		AX=0078 BX=1278 CX=005A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C60 IP=0004 NV UP EI PL NZ NA PE NC 1C60:0004 02C7 **ADD AL,BH**
15		AX=008A BX=1278 CX=005A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C60 IP=0006 OV UP EI NG NZ NA PO NC 1C60:0006 80D400 **ADC AH,00**
16	Dönüş adresi, SP yazmacının gösterdiği adresten 2 word olarak alınıyor.	AX=008A BX=1278 CX=005A DX=0000 **SP=0010** BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C60 IP=0009 NV UP EI PL ZR NA PE NC 1C60:0009 CB **RETF**
17	Dönüş adresi yığından alınırken yığının durumu.	-d ss:0 1C5B:0000 00 00 00 00 00 00 8A 00-00 00 09 00 60 1C AC 15 1C5B:0010 **16 00 5E 1C** 00 00 4B 1C
18	Ana yordama dönüldü. AX yazmacındaki değer sonuç değişkenine yerleştiriliyor.	AX=008A BX=1278 CX=005A DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=0016 NV UP EI PL ZR NA PE NC 1C5E:0016 A30200 **MOV [0002],AX DS:0002=0000**
19	Veri kesimi durumu.	-d ds:0 1C5D:0000 12 78 **8A 00** 00 00 00 00-00 00 00 00 00 00 00 00
20	Dönüş adresi 2 word'dür. SP yazmacının belirlediği yerden çekilerek CS:IP yazmaç ikilisine konur.	AX=008A BX=1278 CX=005A DX=0000 **SP=0014** BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5E IP=0019 NV UP EI PL ZR NA PE NC 1C5E:0019 CB **RETF**
21	Dönüş adresi yığından alınırken yığının durumu.	-d ss:0 1C5B:0000 00 00 00 00 00 00 8A 00-00 00 8A 00 00 00 19 00 1C5B:0010 5E 1C AC 15 **00 00 4B 1C**
22	CS←1C4BH IP←0000H oldu ve program sonlandı.	AX=008A BX=1278 CX=005A DX=0000 SP=0018 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C4B IP=0000 NV UP EI PL ZR NA PE NC **1C4B:0000** CD20 **INT 20**

10.2 Makro Özellikleri

Makrolar, yordamlar gibi assembly program parçalarıdır. Ancak yordamlardan farklı olarak derleme aşamasında isimlerinin geçtiği her yere kendilerine ait kod bloğunu kopyalarlar. Makro kullanılacağı zaman, sadece

macro_ismi { parametre listesi }

ifadesi de kod içine yazılmalıdır. Önemli olan makronun adının geçtiği yerden önce tanımlanmış olmasıdır. Bu makronun kod kesimi içinde ilk olarak kodlanması gereken parça olması demektir. Eğer istenirse makrolar ayrı bir metin dosyası olarak yazılıp programa **INCLUDE** sözde komutu yardımıyla da dâhil edilebilirler. Makronun en genel hali Tablo 10-4'de gösterilmiştir.

Tablo 10-4: Makro Tanımı

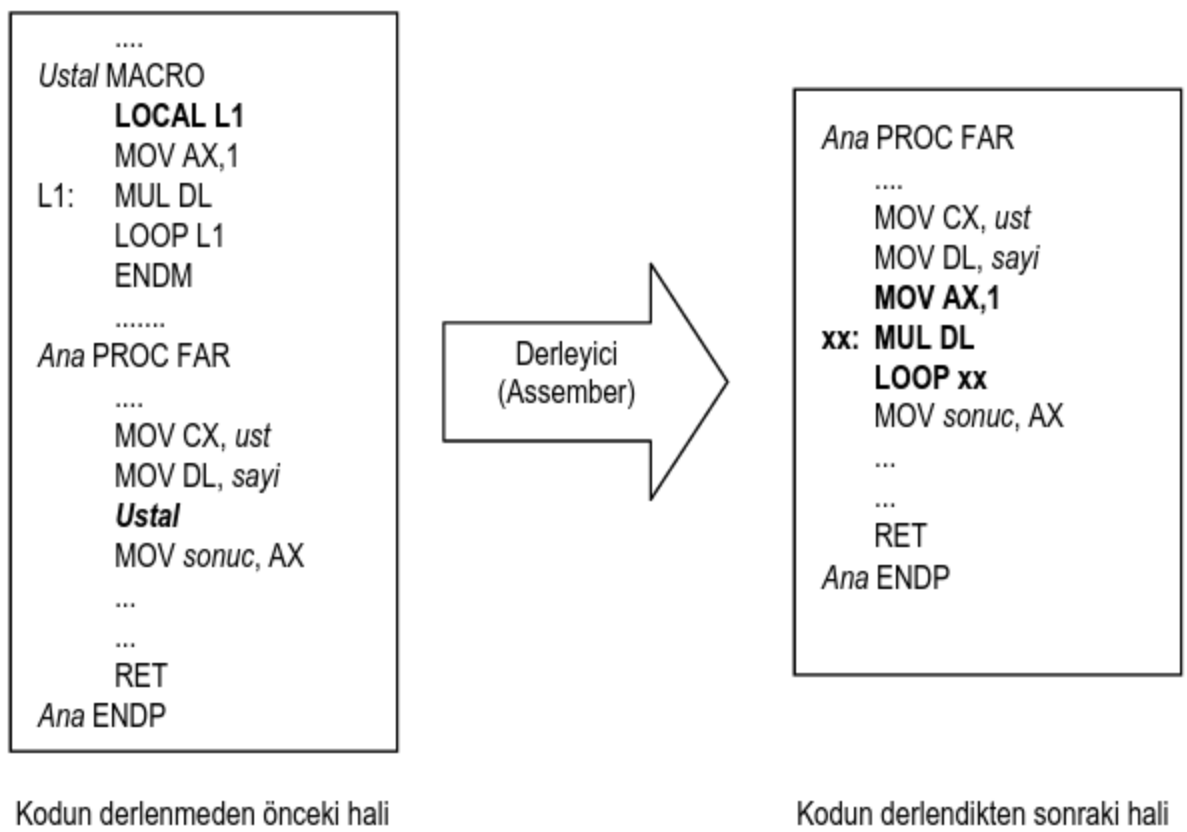
Macro_ismi	**MACRO** { *parametre listesi* }
	LOCAL ....
	ASSUME .....
	; makro kodu
	ENDM

Makrolar, yerine getirecekleri işin bir parçası olarak içinde çeşitli etiketler (label) barındırabilir. Makroların isimlerinin geçtiği yerlere kendi kodlarını yerleştirdikleri bilindiğine göre kullanılan etiketlerin dâhil oldukları programlarda da kullanılmış olması karışıklıklara neden olur. Hatta aynı makronun bir kod kesiminde birden fazla kullanılması durumunda da bu problem ile karşı karşıya kalınır. Kullanılan etiketlerin tek (unique) olması **LOCAL** sözde komutu ile sağlanır. Böylece makro içinde kullanılan etiketlerin dâhil oldukları kod blokları içindeki aynı isimli etiketler ile karışmasının önüne geçilir.

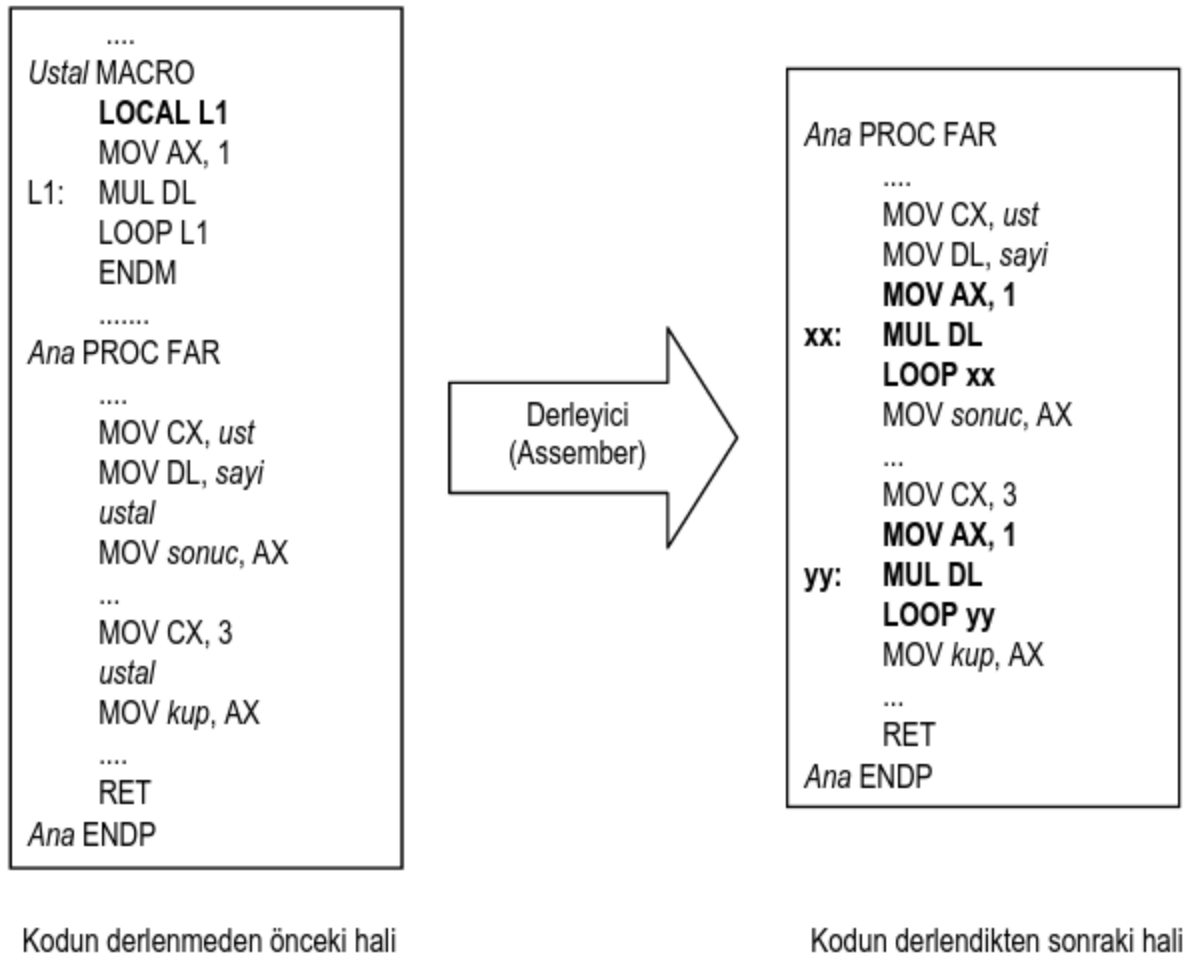
Makro içinde ihtiyaç duyulması durumunda **ASSUME** sözde komutu ile kesim eşlemelerinde de değişiklik yapılabilir.

Makrolara parametre aktarılmasının gerekli olduğu durumda parametreler makro ismini takiben virgül ile ayrılmak kaydıyla verilebilir[137]. Şekil 10-4'de *ustal* makrosunun program içinde bir yerde kullanılması, Şekil 10-5'de ise aynı makronun programın içinde farklı iki yerde kullanılması durumları ve bu programların derlenmesini takiben oluşacak kod yapısı gösterilmektedir.

[137] Yordamlar ve makrolarda farklı parametre aktarma teknikleri kullanılmaktadır. Bunlar ile ilgili detaylar için 12. Bölümü inceleyiniz. Makrolara aktarılacak olan parametre listesi içinde doğrudan bir sayı veya yazmaç ismi kullanılamaz. Parametre listesinde sadece kullanıcı tarafından tanımlanmış değişkenlerin isimleri, virgül ile yarılmış olarak yer alabilir.



Şekil 10-4: Makronun program içinde kullanılması



Şekil 10-5: Makronun program içinde birden fazla yerde kullanılması

Görüldüğü gibi *ustal* isimli macro, isminin her geçtiği yere derleyici tarafından yerleştirilmiştir. Bu sırada makro içinde kullanılan *L1* etiketi derleyici tarafından koda dâhil edilirken (xx ve yy ile gösterilen sembolik) adres olarak ifade edilmiştir. Böylece program içerisinde başka bir etiket ile karışması gibi bir olasılık kalmamıştır. Makroların isimlerinin geçtiği her yere kendilerini yerleştirmeleri nedeniyle derleme sonrasında program kodunun büyüdüğü de dikkat çekmektedir.

10.2.1 Makro Örnekleri

Makro kullanımını ve parametre aktarma işlemlerini yazacağımız basit programlar ile birlikte inceleyelim.

Örnek 10-3

Örneğimiz, bilgisayar ekranı temizleyip bir mesaj yazdırmak olsun. Programımız, daha az kesim tanımı yapılması ve sade yapısı nedeni ile COM tipinde olup, ekranı temizlemek üzere *SIL* ve mesaj yazdırmak için ise *YAZDIR* isimli makrolardan yararlanacaktır.

Makroların ekranı silmesi ve mesaj yazdırabilmesi için mevcut BIOS/DOS kesme fonksiyonlarından[138] **INT** komutunu kullanarak faydalanacağız. Her ne kadar şu ana kadar olan kısımda kesme (interrupt) kavramından söz edilmemiş dahi olsa kesme yordamları da assembly komutlardan oluşmuş program parçalarıdır. Kesme yordamları, yaptıkları işe göre, çağırılmadan önce ilgili yazmaçların ilk değerlerinin uygun şekilde ayarlanması ile kullanılabilir, yaptıkları işin niteliğine bağlı olarak da bir sonuç değeri döndürebilirler. Bu özellikleri dikkate alındığında aslında kesimler arası (intersegment) yordam çağırmaya çok benzer özellikte çalışırlar.

Ekranı silmek için kullanacağımız INT 10 H-Fonksiyon 06H, CX, DX, AH ve BH yazmaçlarından parametre alır. Ekrana mesaj yazdırmak için kullanılacak olan INT 21H-Fonskiyon 09H ise DX ve AH yazmaçlarından parametre almaktadır. Bu kesme yordamlarını kullanacak olan *SIL* isimli makro parametre almayacak ancak *YAZDIR* isimli makro ekrana yazdırılacak olan mesajı parametre olarak alacaktır.

[138] Kesme konusu 14. Bölümde detaylı olarak ele alınmıştır. BIOS kesme listesine 16.4. bölümde, DOS kesme listesine 16.5. bölümde yer verilmiştir.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 10-3.ASM
; COM yapısındaki programda macro kullanımı
; Program ekranı temizledikten sonra bir mesaj yazacaktır
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Ekrana mesaj yazma
CDSEG   SEGMENT  PARA 'CODE'
        ORG 100H        ; COM program 100H den başlar
        ASSUME CS:CDSEG, SS:CDSEG, DS:CDSEG
;---------------------------------------------------------------------
; INT 10H yardımıyla ekranı silmeyi sağlayan macro kodu
;---------------------------------------------------------------------
SIL     MACRO
        MOV CX, 0000H    ; Ekranın sol üst köşesi satır/sütun adresi
        MOV DX, 184FH    ; Ekranın sağ alt köşesi satır/sütun adresi
        MOV BH, 07H      ; öznitelik değeri (attribute byte)
        MOV AX, 0600H    ; AH = 06H pencereyi yukarı kaydırma
        INT 10H          ; 10H numaralı kesmeyi çağır
        ENDM
;---------------------------------------------------------------------
; INT 21H yardımıyla ekrana mesaj yazdıran macro
;---------------------------------------------------------------------
YAZDIR  MACRO text
        LEA DX, text     ; DX yazdırılacak mesajın başını göstermeli
        MOV AH, 09H      ; AH = 09H Ekrana mesaj yazma
        INT 21H          ; 21H numaralı kesmeyi çağır
        ENDM
;---------------------------------------------------------------------
; Macro'ları kullanarak işlemi gerçekleştiren yordam
;---------------------------------------------------------------------
BASLA   PROC NEAR
        SIL              ; SIL isimli macro'yu kullan
        YAZDIR MESAJ     ; YAZDIR makrosunu MESAJ parametresi ile kullan.
        RET
BASLA   ENDP
;---------------------------------------------------------------------
; Mesajın tanımlandığı alan
;---------------------------------------------------------------------
MESAJ   DB 'Ekran silindi ve bu mesaj yazildi...','$'
CDSEG   ENDS
        END BASLA       ; programın başlangıç noktası
```

Yazdığımız programın derlenmesi sırasında üretilen LST uzantılı dosyayı inceleyelim.

```
Microsoft (R) Macro Assembler Version 5.00                  6/17/01
Ekrana mesaj yazma                                          Page     1-1

                        PAGE 60,80
                        TITLE Ekrana mesaj yazma
 0000               CDSEG   SEGMENT PARA 'CODE'
 0100                       ORG 100H
                        ASSUME CS:CDSEG, SS:CDSEG, DS:CDSEG
                SIL     MACRO
                        MOV CX, 0000H
                        MOV DX, 184FH
                        MOV BH, 07H
                        MOV AX, 0600H
                        INT 10H
                        ENDM
```

```
                YAZDIR  MACRO text
                        LEA DX, text
                        MOV AH, 09H
                        INT 21H
                        ENDM
 0100                BASLA   PROC NEAR
                        SIL
 0100  B9 0000                            MOV CX, 0000H
 0103  BA 184F                            MOV DX, 184FH
 0106  B7 07                              MOV BH, 07H
 0108  B8 0600                            MOV AX, 0600H
 010B  CD 10                              INT 10H
                        YAZDIR MESAJ
 010D  8D 16 0116 R                       LEA DX, MESAJ
 0111  B4 09                              MOV AH, 09H
 0113  CD 21                              INT 21H
 0115  C3                   RET
 0116                BASLA   ENDP
 0116  45 6B 72 61 6E 20 73  MESAJ   DB 'Ekran silindi ve bu mesaj yazildi...
                ','$'
       69 6C 69 6E 64 69 20
       76 65 20 62 75 20 6D
       65 73 61 6A 20 79 61
       7A 69 6C 64 69 2E 2E
       2E 24
 013B                CDSEG   ENDS
                               END BASLA

Microsoft (R) Macro Assembler Version 5.00                  6/17/01
Ekrana mesaj yazma                                          Symbols-1

Macros:
      N a m e           Lines
SIL  . . . . . . . . . . . . . .        5
YAZDIR . . . . . . . . . . . . .        3

Segments and Groups:
                N a m e          Length    Align   Combine Class
CDSEG  . . . . . . . . . . . . .     013B PARA NONE 'CODE'

Symbols:
                N a m e          Type  Value    Attr
BASLA  . . . . . . . . . . . . .     N PROC   0100 CDSEG    Length = 0016
MESAJ  . . . . . . . . . . . . .     L BYTE   0116 CDSEG
@FILENAME  . . . . . . . . . . .     TEXT  Örnek 10-3

     43 Source  Lines
     51 Total   Lines
      8 Symbols
  50990 + 383506 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors
```

Makroların kullanımı ile ilgili örneğimiz üzerinde vurgulanması gereken noktalar;

1. Makrolar mutlaka kullanılmadan önce tanımlanmış olmalıdır. Bu nedenle, eğer aynı kod kesimi içinde yer alacaklarsa, yordamlardan önce onların tanımlarının yapılması gerekmektedir. Makrolar istenirse ayrı bir metin dosyası içine konarak **INCLUDE** sözde komutu ile programa dâhil edilmelidir.

2. Makrolar her ne kadar yordamlardan önce tanımlanmış olsa da LST uzantılı dosyanın sol yanındaki bellek adresleri incelendiğinde tanımlandıkları yerde herhangi bir adrese sahip olmadıkları görülecektir.

```
0100                ORG 100H
                     ASSUME CS:CDSEG, SS:CDSEG, DS:CDSEG
        SIL         MACRO
                    MOV CX, 0000H
                    MOV DX, 184FH
                    MOV BH, 07H
                    MOV AX, 0600H
                    INT 10H
                    ENDM
        YAZDIR      MACRO text
                    LEA DX, text
                    MOV AH, 09H
                    INT 21H
                    ENDM
0100    BASLA       PROC NEAR
```

3. Makrolar ancak yordam içinde adlarının geçtiği yerlere kendileri için tanımlanmış kod satırlarını yazarak bellek üzerinde yer tutmaya başlarlar. LST uzantılı dosyada MACRO adının yerine ona ait kod yerleştirilmiştir. İşte bu nedenle programlarımız içinde farklı noktalarda çok sayıda makro kullanılması programımızın koduna her seferinde makronun kod bloğunun eklenmesi anlamına gelir ki bu da programımızın kodunun büyümesine neden olur.

```
                                SIL
0100  B9 0000                                MOV CX, 0000H
0103  BA 184F                                MOV DX, 184FH
0106  B7 07                                  MOV BH, 07H
0108  B8 0600                                MOV AX, 0600H
010B  CD 10                                  INT 10H
                            YAZDIR MESAJ
010D  8D 16 0116 R                           LEA DX, MESAJ
0111  B4 09                                  MOV AH, 09H
0113  CD 21                                  INT 21H
```

4. *YAZDIR* isimli makro, yordam içinde çağırılırken yanına *MESAJ* isimli değişken parametre olarak verilmiştir. Ancak bu parametreye makro içinde *text* isimli bir ara tanımlayıcı ile değerlendirilmektedir.

```
YAZDIR  MACRO text
        LEA DX, text
        MOV AH, 09H
        INT 21H
        ENDM
  YAZDIR MESAJ
010D  8D 16 0116 R                           LEA DX, MESAJ
0111  B4 09                                  MOV AH, 09H
0113  CD 21                                  INT 21H
```

Örnek 10-4

Örnek 10-3'de yazdığımız programda basit bir değişiklik yaparak makroların harici bir dosyadan alınması durumunda yapılacakları inceleyelim.

```
; ---------------------------------------------------------------------
; PROGRAM :Örnek 10-4.ASM
; COM yapısındaki programda macro kullanımı
; Program ekranı temizledikten sonra bir mesaj yazacaktır
; Macro'lar harici bir dosyadan alınarak kullanılacaktır.
; ----------------------------------------------------------------------
        PAGE 60,80
        TITLE Ekrana mesaj yazma
;---------------------------------------------------------------------
; Kullanılacak Macro'lar ORNEK.MAC isimli dosyadan alınmaktadır
;---------------------------------------------------------------------
        INCLUDE Ornek.mac
CDSEG   SEGMENT  PARA 'CODE'
        ORG 100H        ; COM program 100H den başlar
        ASSUME CS:CDSEG, SS:CDSEG, DS:CDSEG
;---------------------------------------------------------------------
; Macro'ları kullanarak işlemi gerçekleştiren yordam
;---------------------------------------------------------------------
BASLA   PROC NEAR
        SIL             ; SIL isimli macro'yu kullan
        YAZDIR MESAJ    ; YAZDIR makrosunu MESAJ parametresi ile kullan
        RET
BASLA   ENDP
;---------------------------------------------------------------------
; Mesaj tanımı
;---------------------------------------------------------------------
MESAJ   DB 'Macro'lar Ornek.mac Dosyasından alındı','$'
CDSEG   ENDS
        END BASLA      ; programın başlangıç noktası
```

Makrolar programımız ile aynı dizinde bulunan *Ornek.mac* isimli dosyada yer almaktadır. *Ornek.mac* isimli dosya aşağıdaki şekilde yazılacaktır.

```
;---------------------------------------------------------------------
; DOSYA: ORNEK.MAC
; Bu dosya harici macro'ların yer aldığı bir dosyadır.
;---------------------------------------------------------------------

;---------------------------------------------------------------------
; INT 10H yardımıyla ekran silmeyi sağlayan macro kodu
;---------------------------------------------------------------------
SIL     MACRO
        MOV CX, 0000H    ; Ekranın sol üst köşesi satır/sütun adresi
        MOV DX, 184FH    ; Ekranın sağ alt köşesi satır/sütun adresi
        MOV BH, 07H      ; attribute byte
        MOV AX, 0600H    ; AH = 06H pencereyi yukarı kaydır
        INT 10H          ; 10H numaralı kesmeyi çağır
        ENDM
;---------------------------------------------------------------------
; INT 21H yardımıyla ekrana mesaj yazdıran macro
;---------------------------------------------------------------------
YAZDIR  MACRO text
        LEA DX, text     ; DX yazdırılacak mesajın başını göstermeli
        MOV AH, 09H      ; AH = 09H ekrana mesaj yaz
        INT 21H          ; 21H numaralı kesmeyi çağır
        ENDM
```

10.3 Yordam ve Makro Karşılaştırılması

Yordam ve makro kullanımları hakkında verdiğimiz temel bilgilerden sonra bu iki modüler program yazma aracını birbirleri ile Tablo 10-5'de verilen özelliklere göre karşılaştıralım.

Tablo 10-5: Yordam ve macro karşılaştırması

Özellik	Yordam	Makro
Çalışma Şekli	Derlenen kod içindeki **CALL** komutunun işlenmesi ile devreye girer. Bu açıdan bakıldığında yürütme süresi (run time) bir kullanım söz konusudur.	Makro kodları derleyici tarafından derleme sırasında kullanıldıkları kodun içine yerleştirilir ve çalıştırılabilen bir kod elde edilir. Bu açıdan derleme süresi (compile time) bir kullanım söz konusudur.
Çalışma Süresi	Yordamlar her çağırılışlarında aynı adres alanından çalıştırılırlar. Çağıran programın kontrolü çağırılana bırakabilmesi ve işin bitimini takiben geri dönebilmesi için göreli konum adresini yığına saklaması, çağırılan yordamın adresinin IP yazmacına aktarması, **RET** komutun çalıştırılmasıyla da dönüş adresinin yığından alınması gerekecektir. Buna göre her yordam çağırma işlemi arka planda doğrudan görünmeyen PUSH, atama ve POP işlemlerinin yapılmasını gerektirir. Bu komutların çalıştırılması da doğal olarak çalışma süresini uzatacaktır.[139]	Makrolar derleme sırasında programın koduna eklendiğinden ötürü herhangi bir adresin saklanması, kontrolün farklı bir adreste bulunan kod parçasına aktarılması ve daha sonra saklanan adreslerin alınarak geri dönülmesi söz konusu olmaz. Bu sayede yordamlarda olduğu gibi arka planda ek olarak yapılması gereken işlemler yoktur ve zaman kaybına neden olmaz. Bu açıdan değerlendirildiğinde aynı işi yapan makroların, yordamlardan daha hızlı çalıştığını rahatlıkla söyleyebiliriz.
Kod büyüklüğü	Yordamlar hep aynı kod alanından çalıştırıldığı için bir program içinde aynı yordamın birden fazla yerde çağırılması programın kodunda herhangi bir büyüme neden olmaz.	Makrolar isimlerinin geçtiği her yere eklendikleri için program içinde ne kadar çok kullanılırlarsa programın kodunun o derece büyümesine neden olurlar.
Parametre aktarımı	Yordamlara yazmaçlar, yığın ve ortak veri kesimleri üzerinden parametre aktarılabilir.	Makrolar isimlerini takiben, kullanıcı tarafında tanımlanmış, virgül ile ayrılmış değişkenleri parametre listesi olarak kabul eder.

Unutulmamalıdır ki yordamların da, makroların da kendilerine has özellikleri vardır. Bu özellikler dikkate alındığında herhangi birinin diğerinden üstün olduğunu söylemek zordur. Önemli olan her iki yapının da getirilerinin neler olduğunun bilinmesi ve hangisi mevcut durum için daha hızlı, daha az bellek kullanarak bir çözüm üretecek ise onun tercih edilmesidir.

[139] FAR tanımlı yordamlar için IP yazmacının yanı sıra CS yazmacının da değerinin değişmesi gereklidir. Bu durumda arka planda yapılması gereken işlem sayısı iki katına çıkmış olacaktır.

11. ANA VE ALT PROGRAMLARDAKİ KESİMLERİN BİRLEŞTİRİLMESİ

Geliştirdiğimiz uygulamaların gereği olarak bazen yararlanacağımız yordamlar farklı programlara ait kod kesimleri içinde yer alabilir. Bu durumlarda, gerek çağıran programda (**EXTRN** *yordam_ismi*:FAR) gerek ise çağırılan programlar tarafında (**PUBLIC** *yordam_ismi*) bazı ek tanımlamaların yapılmış olması ve yararlanılacak yordamların içinde bulundukları programların belleğe yerleştirilmiş (yüklenmiş) olması gerekir.

Yordamlar yerine getireceklere işlevin karmaşıklığına bağlı olarak değişik büyüklüklerde bellek kullanırlar. Yordamı oluşturan kod ister 10-15 byte isterse 64 KB büyüklüğünde olsun, kod kesimi için bellek üzerinde en az 64 KB'lık bir bölge kullanılır. Yordamlar, çoğu zaman program için ayrılmış olan kod kesimini tamamen kullanamazlar. Kod kesimi içinde yordamlar tarafından kullanılamayan bellek alanı da başka bir amaç için kullanılamayacağından bellek etkin bir şekilde kullanılamamış olur.

Diğer yandan uygulama farklı kesimlerde bulunan yordamları kullanırken kesimler arası yordam çağırma (intersegment call) yapmak zorunda kalacak;

- yordamlar arasındaki gidiş/gelişin sağlanabilmesi için kesim ve göreli konum adreslerini yığında saklanma,
- uzak yordama ulaşabilmek için CS ve IP yazmaçlarına çağırılacak olan yordamın adreslerini yerleştirme,
- RETF komutunun işlenmesi ile dönüş adreslerini yığından alarak IP ve CS yazmaçlarına aktarma

işlemleri de göreceli bir gecikme yaşanmasına neden olacaktır.

Bu tespitler, assembly programlar için baştan beri vurguladığımız, sistem kaynaklarını mümkün olduğunca az ve etkin kullanma ile hızlı çalışma özellikleriyle çelişmektedir. Tespitlerimizin temelindeki problemler, ana ve alt programların kod kesimlerinin birleştirilmesi ile kolaylıkla ortadan kaldırılabilir.

11.1 Birleştirilmiş (Combined) Kod Kesimi Kullanımı

Farklı programlar içindeki kod kesimlerin birleştirilmesi, kesim isimlerinin, kesimin başlangıç adresini belirlemek üzere kullanılan hizalama (allignment) tiplerinin ile kesimleri gruplamak için kullanılan sınıf (class) tiplerinin aynı olması ve kesim birleştirme (combine) tiplerinin **PUBLIC** tanımlanması ile sağlanır.

Tablo 11-1'de verilen *ANA.ASM* isimli program içindeki *Basla* isimli yordam, Tablo 11-2'de verilen *ALT ASM* isimli program içinde yer alan *Altprg* isimli yordamı kullanmak üzere tasarlanmıştır. *Altprg* isimli yordamın farklı bir kesimde olduğu **EXTRN** sözde komutu ile (satır 3) belirtilmiştir. Her iki programın da kod kesim-

leri ***Codesg*** isminde, kesim hizalama tipileri **PARA**, birleştirme tipleri **PUBLIC** ve sınıf tipleri '**code**' olarak (satır 6) belirlenmiş olup bu sayede her iki programın kod kesimlerinin birleştirilmesi sağlanmıştır.

ANA.ASM programı içindeki *Basla* isimli yordam, adının programın sonundaki END komutunu takiben yazılmış olması nedeni ile (satır 18) kontrolü elinde bulunduran ana yordam konumundadır. *Basla* isimli yordamın öncelikli görevi geri dönüş için gerekli adresleri yığında saklamaktır (satır 9-11). Daha sonra CALL komutunu kullanarak *ALT.ASM* içinde tanımlı *Altprg* isimli yordamı kesimler arası yordam çağırma tekniğine uygun olarak çağırmaktadır. (satır 13).

Tablo 11-1: Birleştirilmiş (combined) kod kesimi – ANA.ASM

```
ANA.ASM
 1              PAGE 60,80
 2              TITLE Birleştirilmiş code kesim – iskelet yapı
 3              EXTRN Altprg: FAR
 4              ......          ; Diğer kesim tanımları
 5   Basla      PROC FAR
 6   Codesg     SEGMENT PARA PUBLIC 'code'
 7              ASSUME CS:Codesg, SS:.....,DS:.....
 8   Basla      PROC FAR
 9              PUSH DS         ; Dönüş adreslerinin yığında saklanması
10              XOR AX,AX
11              PUSH AX
12              ......          ; Diğer işlemler
13              CALL Altprg
14              .....
15              RET
16   Basla      ENDP
17   Codesg     ENDS
18              END Basla       ; Programın başlangıç noktası
```

Tablo 11-2: Birleştirilmiş (combined) kod kesimi – ALT.ASM

```
ALT.ASM
 1              PAGE 60,80
 2              TITLE Birleştirilmiş code kesim – iskelet yapı
 3              PUBLIC Altprg
 4              ......
 5
 6   Codesg     SEGMENT PARA PUBLIC 'code'
 7              Assume CS:Codesg
 8   Altprg     PROC FAR
 9
10              .....           ; Diğer işlemler
11              RET             ; Çağırıldığı noktaya geri dön
12   Altprg     ENDP
13   Codesg     ENDS
14              END
```

Diğer yandan Tablo 11-2'de verilen *ALT.ASM* programı içindeki *Altprg* isimli yordamın başkaları tarafından kullanılabilmesini sağlamak üzere **PUBLIC** tanımı yapılmıştır (satır 3). Kesim tanımı *ANA.ASM* programında yapılan kod kesim tanımı ile aynıdır (satır 6). Daha önceki örneklerimizde de olduğu gibi *ALT.ASM* isimli program kendi başına çalışmak yerine, gelen isteklere hizmet vermek üzere tasarlandığından dolayı END komutundan sonra hiçbir başlangıç noktası belirtilmemiştir (satır 14).

Görüldüğü gibi uygun tanımlar yaparak gerek çağıran, gerekse çağırılan yordamı tek bir kod kesimi içinde toplayarak (diğer değişle kod kesimlerini birleştirerek) mevcut belleği daha etkin kullanmak mümkündür. Bu noktada programcıların aklına "*eğer çağıran ve çağırılan yordamların içinde bulunduğu kesimleri birleştirerek tek kesim haline getirmek mümkün ise, çağırılan yordamı tanımlamak için kullanılan EXTRN ve yordamın çağırılabileceğini tanımlamak için kullanılan PUPLIC sözde komutlarının kullanılmasına gerek kalır mı?*" sorusu gelebilir. Sorunun cevabı "*evet, kesimler birleştirilse bile EXTRN/PUBLIC tanımlamalarına ihtiyaç vardır*" olacaktır. Zira derleme aşamasında CALL satırında verilen isminde bir yordamın söz konusu kesim içinde bulması veya ona ait bir harici referansın (EXTRN) derleyiciye verilmiş olması beklenir. Aksi halde derleyici hata verecektir. Kesimlerin birleştirilmesi, derlemenin ardından link işlemini takiben kod belleğe yüklenirken gerçekleşeceği için, EXTRN/PUBLIC tanımlarının kullanılması hatasız derleme yapabilmek için gerekmektedir.

Ancak bundan sonra da aklımıza "*kesimler birleşeceğine göre kesimler arası yordam çağırma yerine, kesim içi yordam çağırma işlemi yapıyor olacağız. Bu durumu göz önüne alarak çağırılan yordamı FAR yerine NEAR tanımlamak gerekmez mi?*" sorusu gelecektir. Bu doğru bir tespittir. EXTRN sözde komutunu takiben yordamı tanımlamak için kullanacağımız FAR veya NEAR tamamen çağırılacak yordamın tipi ile ilgilidir. Buna göre çağıracağımız yordamı;

yordam_ismi PROC **FAR**	şeklinde tanımlanmış ise	EXTRN *yordam_ismi*: **FAR**
yordam_ismi PROC **NEAR**	şeklinde tanımlanmış ise	EXTRN *yordam_ismi*: **NEAR**

şeklinde tanımlayarak kullanmak mümkündür.

Buna göre *ANA.ASM* isimli programın 3. satırını, **EXTRN Altprg:NEAR** ve *ALT.ASM* isimli programın 8. satırını **Altprg PROC NEAR** olarak değiştirerek kesimleri birleştirmenin yanı sıra kesimler arası yordam çağırma yerine kesim içi yordam çağırma yaparak daha etkin çalışan bir program yazmış oluruz. Ancak çağırılacak olan yordamın FAR olarak tanımlandığı ve bunun değiştirilmesinin mümkün olmadığı (sadece OBJ uzantılı dosyanın bulunduğu) durumlarda **EXTRN *yordam_ismi*:FAR** tanımlamasının yapılması mecburidir. Bu durumda kesimlerin birleştirilmesi ile bellek kullanımı iyileştirilirken kesimler arası yordam çağırmadan doğan gecikmeye razı olunacaktır.

Örnek 11-1

Word tanımlı, pozitif sayılardan oluşan,10 elemanlı bir dizinin en büyük elemanı, *BUYUK* isimli harici bir yordam yardımıyla bulan programı ve söz konusu harici yordamın içinde yer alacağı yardımcı programı yazalım.

Örneğimizde yazmamız beklenen iki programdan ilki, 10 elemanlı dizinin tanımlandığı ve genel olarak kontrolü elinde bulunduracak olan Örnek 11-1.ASM isimli program, ikincisi ise *BUYUK* isimli yordamı bulunduracak olan Örnek 11-1-1.ASM isimli programdır. *BUYUK* isimli yordamın FAR olarak tanımlandığını kabul ederek ilk programımızı kodlayacak olursak;

```
; ---------------------------------------------------------------------
; PROGRAM :Örnek 11-1.ASM
; Kullanılacak harici bir yordam ile dizi içindeki en büyük elemanı
; bulma. kod kesimleri birleştiriliştir.
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Code kesimleri birleşik - yardımcı yordam çağırma
; ---------------------------------------------------------------------
; BUYUK harici bir yordamdır. FAR olarak tanımlanmıştır.
; DIZI ise başka bir program tarafından kullanılacaktır.
; ---------------------------------------------------------------------
        EXTRN BUYUK:FAR
        PUBLIC DIZI
STSEG   SEGMENT  PARA STACK 'STACK'
        DW 12 DUP(0)
STSEG   ENDS
DTSEG   SEGMENT  PARA 'DATA'
;---------------------------------------------------------------------
; kullanılacak değişkenler tanımlanır.
;---------------------------------------------------------------------
DIZI    DW 12,1,5,64,21,65,127,512,0,1024
ELEMAN  DW 10
BYK_EL  DW ?
DTSEG   ENDS
CDSEG   SEGMENT  PARA PUBLIC 'CODE'      ; Kesim  birleştirilecek
        ASSUME CS:CDSEG,SS:STSEG,DS:DTSEG
BASLA   PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        XOR AX,AX
        PUSH AX
;---------------------------------------------------------------------
; DTSEG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DTSEG
        MOV DS,AX
;---------------------------------------------------------------------
; işlem bloğu başlıyor
;---------------------------------------------------------------------
        MOV CX, ELEMAN  ; Eleman sayısını CX ile BUYUK isimli yordama aktar
        CALL BUYUK      ; BUYUK isimli yordamı çağırılır
        MOV BYK_EL,AX   ; En büyük değer AX yazmacı üzerinden geri döner
        RET             ; DOS'a dön
BASLA   ENDP
CDSEG   ENDS
        END BASLA      ; programın başlangıç noktası
```

Örnek 11-1.ASM isimli ilk programı derleyerek Örnek 11-1.LST dosyasını oluşturalım. Bu dosyayı birlikte inceleyerek bazı önemli noktaları vurgulayalım

```
Microsoft (R) Macro Assembler Version 5.00                    6/24/01
Code kesimleri birleşik - yardımcı yordam çağırma      Page     1-1

                        EXTRN BUYUK:FAR
                        PUBLIC DIZI
 0000                STSEG   SEGMENT  PARA STACK 'STACK'
 0000  000C[                     DW 12 DUP(0)
    0000
           ]

 0018                STSEG   ENDS
 0000                DTSEG   SEGMENT  PARA 'DATA'
 0000  000C 0001 0005 0040   DIZI    DW 12,1,5,64,21,65,127,512,0,1024
       0015 0041 007F 0200
       0000 0400
 0014  000A              ELEMAN  DW 10
 0016  ????              BYK_EL  DW ?
 0018                DTSEG   ENDS
 0000                CDSEG   SEGMENT  PARA PUBLIC 'CODE'
                        ASSUME CS:CDSEG,SS:STSEG,DS:DTSEG
 0000                BASLA   PROC FAR
 0000  1E                    PUSH DS
 0001  33 C0                 XOR AX,AX
 0003  50                    PUSH AX
 0004  B8 ---- R             MOV AX, DTSEG
 0007  8E D8                 MOV DS,AX
 0009  8B 0E 0014 R        MOV CX, ELEMAN
 000D  9A 0000 ---- E      CALL BUYUK
 0012  A3 0016 R             MOV BYK_EL,AX
 0015  CB                    RET
 0016                BASLA   ENDP
 0016                CDSEG   ENDS
                               END BASLA


Microsoft (R) Macro Assembler Version 5.00                    6/24/01
Code kesimleri birleşik - yardımcı yordam çağırma      Symbols-1

Segments and Groups:
                N a m e          Length   Align    Combine Class
CDSEG  . . . . . . . . . . . . . . .  0016    PARA PUBLIC   'CODE'
DTSEG  . . . . . . . . . . . . . . .  0018    PARA NONE 'DATA'
STSEG  . . . . . . . . . . . . . . .  0018    PARA STACK    'STACK'

Symbols:
                N a m e          Type Value    Attr
BASLA  . . . . . . . . . . . . . . .F PROC    0000 CDSEG    Length = 0016
BUYUK  . . . . . . . . . . . . . . .L FAR     0000 External
BYK_EL . . . . . . . . . . . . . . .L WORD    0016 DTSEG
DIZI . . . . . . . . . . . . . . . .L WORD    0000 DTSEG    Global
ELEMAN . . . . . . . . . . . . . . .L WORD    0014 DTSEG
@FILENAME  . . . . . . . . . . . . .TEXT          Örnek 11-1

     48 Source  Lines
     48 Total   Lines
     13 Symbols

  50984 + 383512 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

LST uzantılı dosyada, daha önceki örneklerimizde de olduğu gibi, bazı atıfların adresleri program belleğe yerleştirilinceye kadar belli olmadığı için **R** (relative - bağıl) ve **E** (external - harici) harfleri kullanılarak belirtilmiştir. *BUYUK* isimli yordamın çağırıldığı komut satırında görüldüğü gibi E harfinden önce **----** sembolü ile boş bırakılan alana o yordama ait adres gelecektir.

```
000D  9A 0000 ---- E     CALL BUYUK
```

E ile belirlenen adresler ancak her iki programın hatasız derlenmesi ile elde edilecek OBJ dosyalarının birlikte link işlemine tabii tutulmasından sonra belirlenecektir. LST uzantılı dosyanın kesim tanımları ile ilgili satırlarında CDSEG isimli kesimin **PUBLIC** tanımlandığı görülmektedir.

```
Segments and Groups:
                N a m e         Length Align  Combine Class
CDSEG  . . . . . . . . . . . . .0016 PARA PUBLIC   CODE'
```

Şimdi programın ikinci kısmı olan ve en büyük sayıyı bulacak *BUYUK* isimli yordamın içinde bulunduğu diğer program parçasını (Örnek 11-1-1.ASM) kodlayalım.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 11-1-1.ASM
; Örnek 11-1.ASM isimli program tarafından çağırılan yardımcı yordam
; Kod kesimleri birleştirilmiştir.
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Code kesimleri birleşik - yardımcı yordam çağırma
; ---------------------------------------------------------------------
; BUYUK isimli yordam başka programlar tarafından da kullanılabilir.
; DIZI ise başka bir kesim de word tanımlı harici veridir.
; ---------------------------------------------------------------------
        EXTRN DIZI: WORD
        PUBLIC BUYUK
CDSEG   SEGMENT PARA PUBLIC 'CODE'      ; Kesim  birleştirilecek
        ASSUME CS: CDSEG
BUYUK   PROC FAR
;---------------------------------------------------------------------
; işlem bloğu başlıyor
; CX üzerinden dizinin eleman sayısı bu yordama aktarılıyor.
;---------------------------------------------------------------------
        XOR SI, SI       ; SI dizi indisi olarak kullanılacak
        MOV AX, DIZI[SI] ; ilk elemanı en büyük kabul et
        SUB CX, 1        ; geri kalan elemanlar üzerinden işlem yapılacak
L1:     ADD SI, 2        ; DW tanımlı bir sonraki elaman 2 byte ileride
        CMP AX, DIZI[SI]
        JA DEVAM
        MOV AX, DIZI[SI] ; dizinin elemanı daha büyük
DEVAM:  LOOP L1          ; cevrimi tekrarla
        RET              ; çağıran noktaya geri don
BUYUK   ENDP
CDSEG   ENDS
        END
```

Örnek 11-1-1.ASM isimli programın derleyerek oluşan LST uzantılı dosyayı inceleyelim;

```
Microsoft (R) Macro Assembler Version 5.00                  6/24/01
Code kesimleri birleşik - yardımcı yordam çağırma      Page     1-1

                        EXTRN DIZI:WORD
                        PUBLIC BUYUK

 0000                CDSEG   SEGMENT  PARA PUBLIC 'CODE'
                        ASSUME CS:CDSEG
 0000                BUYUK   PROC FAR
 0000  33 F6                XOR SI, SI
 0002  8B 84 0000 E       MOV AX, DIZI[SI]
 0006  83 E9 01         SUB CX, 1
 0009  83 C6 02L1:      ADD SI, 2
 000C  3B 84 0000 E      CMP AX, DIZI[SI]
 0010  77 04                JA DEVAM
 0012  8B 84 0000 E       MOV AX, DIZI[SI]
 0016  E2 F1             DEVAM:  LOOP L1
 0018  CB                   RET
 0019                BUYUK   ENDP
 0019                CDSEG   ENDS
                           END

Microsoft (R) Macro Assembler Version 5.00                  6/24/01
Code kesimleri birleşik - yardımcı yordam çağırma      Symbols-1

Segments and Groups:
                N a m e         Length    Align   Combine Class
CDSEG  . . . . . . . . . . . . .     0019 PARA PUBLIC   'CODE'

Symbols:
                N a m e         Type  Value    Attr
BUYUK  . . . . . . . . . . . . . F PROC   0000 CDSEG    Global   Length     =
0019
DEVAM  . . . . . . . . . . . . . L NEAR   0016 CDSEG
DIZI . . . . . . . . . . . . . . V WORD   0000 External
L1 . . . . . . . . . . . . . . . L NEAR   0009 CDSEG
@FILENAME  . . . . . . . . . . . TEXT     Örnek 11-1-1

     33 Source  Lines
     33 Total   Lines
      8 Symbols

  50864 + 383632 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

Örnek 11-1-1.LST dosyasında, Örnek 11-1.LST dekine benzer bir kesim tanımı görülmektedir.

```
Segments and Groups:
                N a m e         Length    Align   Combine Class
CDSEG  . . . . . . . . . . . . .     0019 PARA PUBLIC   'CODE'
```

Ayrıca Örnek 11-1.ASM içinde tanımlı olan *DIZI* isimli değişkene

```
EXTRN DIZI:WORD
```

tanımı sayesinde farklı kod kesimi içindeki yordam tarafından, sanki kendi veri kesiminde tanımlı bir değişken gibi erişilebilmektedir. LST uzantılı dosyada *DIZI* is-

minin geçtiği her yerde **E** harfi ile bu değişkenin başka bir programın veri alanında olduğu vurgulanmaktadır.

```
0002  8B 84 0000 E       MOV AX,DIZI[SI]
000C  3B 84 0000 E       CMP AX,DIZI[SI]
0012  8B 84 0000 E       MOV AX,DIZI[SI]
```

Programlarımız hatasız derlendikten sonra her program için oluşturulan OBJ uzantılı dosyalar Link işleminden geçirilerek EXE uzantılı dosyayı oluşturulur.

```
C:\assembly>link Örnek 11-1 + Örnek 11-1-1

Microsoft (R) Overlay Linker  Version 3.61
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [Örnek 11-1.EXE]:Örnek 11-1.exe
List File [NUL.MAP]:Örnek 11-1.map
Libraries [.LIB]:
```

Link işlemi sırasında üretilen MAP uzantılı dosya, oluşturulan çalıştırılabilir kodun kullandığı kesimlerin nasıl yapılandırıldığını göstermektedir.

```
Start  Stop   Length Name                   Class
00000H 00017H 00018H STSEG                  STACK
00020H 00037H 00018H DTSEG                  DATA
00040H 00078H 00039H CDSEG                  CODE

Program entry point at 0004:0000
```

MAP dosyasında görüldüğü gibi her iki programda (Örnek 11-1.ASM ve Örnek 11-1-1.ASM) kullanılan kod kesim tanımlamalarının aynı olması sonucu kesimler birleştirilerek *CDSEG* isimli tek bir kod kesimi oluşturulmuştur. Burada görünen *STSEG* ve *DTSEG* isimli kesimler ise Örnek 11-1.ASM içinde tanımlı olan yığın ve veri kesimleridir. Şimdi bu açıklamanın doğruluğunu kanıtlamak üzere Örnek 11-1.LST isimli dosyadaki kesim tanımlarına tekrar bakalım;

```
Segments and Groups:
                N a m e          Length   Align    Combine Class
CDSEG  . . . . . . . . . . . . . . 0016    PARA PUBLIC    'CODE'
DTSEG  . . . . . . . . . . . . . . 0018    PARA NONE      'DATA'
STSEG  . . . . . . . . . . . . . . 0018    PARA STACK     'STACK'
```

Hemen dikkat çekeceği gibi *DTSEG* ve *STSEG* isimli kesimlerin boyutları hem Örnek 11-1.LST dosyasında hem de MAP uzantılı dosyada aynıdır. Oysa *CDSEG* isimli kesim boyutu Örnek 11-1.LST dosyasında 0016H, Örnek 11-1-1.LST dosyasında 0019H iken MAP uzantılı dosyada 0039H olarak görünmektedir. Her iki programın kod kesimleri birleştirildiğinde oluşan kesimin boyutunun bu iki kesimin boyutunun toplamı olan (0016H+0019H) 002FH olması beklenirken MAP dosyasında bu değer nasıl ve neden 0039H olarak görünmektedir?. Bu sorunun cevabı kesimlerin hizalama tipini belirleyen tanımda gizlidir. Kesimin başlayacağı adresi belirlemek üzere kullanılan PARA ifadesi kesim adresinin 16'ya tam bölünebilir bir sayı olmasını sağlamak üzere kullanılır. Buna göre Örnek 11-1.ASM'de tanımlı *CDSEG* isimli kesim 16'ya tam bölünebilir bir noktadan başlar, hemen ardından Örnek 11-1-1.ASM'de tanımlı *CDSEG* yine takip eden ilk 16'ya tam bölü-

nebilir adresten başlar. Bu durum Tablo 11-3'de gösterilmektedir. Tablodan da görüldüğü gibi her iki yordamın arasında kullanılmayan bir bölge oluşmaktadır. Bu bölge, PARA hizalama tipi kullanıldığında 1-15 byte büyüklüğe ulaşabilir. PARA yerine BYTE hizalama tipi kullanılırsa bu bölge en aza indirilir.

Tablo 11-3: Birleştirilmiş kesimlerin yerleştiği bellek adresleri

Başlangıç adresi (16'ya tam bölünebilir ilk adres)	Bitiş adresi	Kesim uzunluğu	Dâhil olduğu program
0040H	0055H	0016H	Örnek 11-1.ASM
	0055H-0060H arasında kalan PARA tanımı uyarınca kullanılamayan bölge.		
0060H	0078H	0019H	Örnek 11-1-1.ASM

Programımızın işleyişini debug ortamında görmeye çalışalım;

```
C:\assembly>debug örnek 11-1.exe

-u
1C5F:0000 1E            PUSH    DS
1C5F:0001 33C0          XOR     AX, AX
1C5F:0003 50            PUSH    AX
1C5F:0004 B85D1C        MOV     AX, 1C5D
1C5F:0007 8ED8          MOV     DS, AX
1C5F:0009 8B0E1400      MOV     CX, [0014]
1C5F:000D 9A20005F1C    CALL    1C5F:0020
1C5F:0012 A31600        MOV     [0016], AX
1C5F:0015 CB            RETF
1C5F:0016 0000          ADD     [BX+SI], AL
1C5F:0018 0000          ADD     [BX+SI], AL
```

Gördüğümüz kod parçasında 0000H-0015H adresleri arasında yer alan kod, Örnek 11-1.ASM isimli programda yazılan koddur. RETF komutunu takip eden

```
1C5F:0016 0000          ADD     [BX+SI], AL
```

komut satırı bizim programlarımızın hiç birine ait olmayıp, Tablo 11-3 de açıklamaya çalıştığımız ve kullanılamayan alan olarak ifade ettiğimiz bellek alanındaki byte'ların komut olarak değerlendirilmesi sonucunda görünen değerlerdir. Programın devamını oluşturan Örnek 11-1-1.ASM deki program ise 0020H adresinden[140] itibaren yerleşmiş olacaktır. Bu komut bloğunu görebilmek için debug ortamında **U CS:0020** yazılmalıdır.

```
-u cs:0020

1C5F:0020 33F6          XOR     SI, SI
1C5F:0022 8B840000      MOV     AX, [SI+0000]
1C5F:0026 83E901        SUB     CX, +01
1C5F:0029 83C602        ADD     SI, +02
1C5F:002C 3B840000      CMP     AX, [SI+0000]
```

[140] 16'ya tam bölünebilir ilk adres 0020H'dir. Ayrıca yordam çağırmayı sağlayan 1C5F:000D adresindeki CALL komutunun göreli konum değeri de (0020H) bu adresi tespit etmemize yardımcı olacaktır.

```
1C5F:0030 7704          JA      0036
1C5F:0032 8B840000      MOV     AX, [SI+0000]
1C5F:0036 E2F1          LOOP    0029
1C5F:0038 CB            RETF
```

Tablo 11-4: Örnek programın adım adım çalıştırılması (Örnek 11-1)

	Kısa Açıklama	DEBUG programının ürettiği değerler
1	Programın dönüş adresleri yığına yerleştiriliyor. Yığın 12 word tanımlandığı için SP yazmacının değeri başlangıçta 0018H (24) dir. Her PUSH işlemi ile bu sayı iki azalır.	AX=0000 BX=0000 CX=0079 DX=0000 **SP=0018** BP=0000 SI=0000 DI=0000 **DS=1C4B** ES=1C4B SS=1C5B CS=1C5F IP=0000 NV UP EI PL NZ NA PO NC 1C5F:0000 1E **PUSH DS**
2		AX=0000 BX=0000 CX=0079 DX=0000 SP=0016 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5F IP=0001 NV UP EI PL NZ NA PO NC 1C5F:0001 33C0 **XOR AX, AX**
3		**AX=0000** BX=0000 CX=0079 DX=0000 SP=0016 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5F IP=0003 NV UP EI PL ZR NA PE NC 1C5F:0003 50 **PUSH AX**
4	Yığına yerleştirilen DS ve AX yazmaçlarının değerleri görülmektedir.	-d ss:0 1C5B:0000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 04 00 1C5B:0010 5F 1C AC 15 **00 00 4B 1C-**
5	Veri kesimine erişebilmek için gerekli tanımlar yapılıyor.	AX=0000 BX=0000 CX=0079 DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5F IP=0004 NV UP EI PL ZR NA PE NC 1C5F:0004 B85D1C **MOV AX, 1C5D**
6		AX=1C5D BX=0000 CX=0079 DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5F IP=0007 NV UP EI PL ZR NA PE NC 1C5F:0007 8ED8 **MOV DS, AX**
7	Veri kesiminin durumu.	-d ds:0 1C5D:0000 0C 00 01 00 05 00 40 00-15 00 41 00 7F 00 00 02 1C5D:0010 00 00 00 04 **0A 00** 00 00-00 00 00 00 00 00 00 00
8	CX←ELEMAN	AX=1C5D BX=0000 CX=0079 DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0009 NV UP EI PL ZR NA PE NC 1C5F:0009 8B0E1400 **MOV CX, [0014] DS:0014=000A**
9	Yordam aynı kod kesimi içinde olmasına rağmen **FAR** tanımdan dolayı kesim/göreli konum ikilisi ile erişilmektedir.	AX=1C5D BX=0000 **CX=000A** DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B **CS=1C5F** IP=000D NV UP EI PL ZR NA PE NC 1C5F:000D 9A20005F1C **CALL 1C5F:0020**
10	Dönüş kesimi olarak CS değeri (1C5FH), dönüş göreli konumu olarak CALL komutunu takip eden MOV [0016], AX komutun adresi (0012H) yığına yerleşir.	-d ss:0 1C5B:0000 00 00 00 00 00 00 5D 1C-00 00 20 00 5F 1C AC 15 1C5B:0010 **12 00 5F 1C** 00 00 4B 1C-
11	CS ve IP yazmaçları, CALL komutunda belirlenen değerleri almış ve dallanma gerçekleşmiştir.	AX=1C5D BX=0000 CX=000A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B **CS=1C5F IP=0020** NV UP EI PL ZR NA PE NC **1C5F:0020** 33F6 **XOR SI, SI**
12	EXTRN olarak tanımlı *DIZI* değişkenine veri kesimi üzerinden erişilmektedir.	AX=1C5D BX=0000 CX=000A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0022 NV UP EI PL ZR NA PE NC 1C5F:0022 8B840000 **MOV AX, [SI+0000] DS:0000=000C**
13	CX değeri ayarlanıyor.	AX=000C BX=0000 CX=000A DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0026 NV UP EI PL ZR NA PE NC 1C5F:0026 83E901 **SUB CX, +01**

14	Tekrarlan işlem bloğu. 16. satırdaki koşul sağlanmaz ise 17. satır (verilen örnekte bu satır daha sonraki adımlarda çalıştığı için olası farklı değerleri göstermek için DS:**xxxx=xxxx** ifadesi kullanılmıştır. Aksi halde program 0036H adresinden devam edecektir (satır 18).	AX=000C BX=0000 CX=0009 DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0029 NV UP EI PL NZ NA PE NC 1C5F:**0029** 83C602 **ADD SI, +02**
15		AX=000C BX=0000 CX=0009 DX=0000 SP=0010 BP=0000 SI=0002 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=002C NV UP EI PL NZ NA PO NC 1C5F:002C 3B840000 **CMP AX, [SI+0000]** DS:0002=0001
16		AX=000C BX=0000 CX=0009 DX=0000 SP=0010 BP=0000 SI=0002 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0030 NV UP EI PL NZ NA PO NC 1C5F:0030 7704 **JA 0036**
17		AX=xxxx BX=0000 CX=xxxx DX=0000 SP=0010 BP=0000 SI=xxxx DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0032 1C5F:0032 8B840000 MOV AX, [SI+0000] **DS:xxx=xxxx**
18		AX=000C BX=0000 CX=0009 DX=0000 SP=0010 BP=0000 SI=0002 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0036 NV UP EI PL NZ NA PO NC 1C5F:**0036** E2F1 **LOOP 0029**
19	Çağıran programa dönmek üzere RETF çalıştırılmıştır.	AX=0400 BX=0000 **CX=0000** DX=0000 **SP=0010** BP=0000 SI=0012 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0038 NV UP EI NG NZ NA PE CY 1C5F:0038 CB **RETF**
20	Dönüş adresi yığından alınır. IP←0012H, CS←1C5FH	-d ss:0 1C5B:0000 00 00 00 00 00 00 40 00-00 00 38 00 5F 1C AC 15 1C5B:0010 **12 00 5F 1C** 00 00 4B 1C-
21	Elde edilen sonuç veri kesiminde tanımlı olan değişkene yazılıyor.	AX=0400 BX=0000 CX=0000 DX=0000 **SP=0014** BP=0000 SI=0012 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0012 NV UP EI NG NZ NA PE CY **1C5F:0012** A31600 **MOV [0016], AX** **DS:0016=0000**
22	DOS'a dönüş yapılıyor.	AX=0400 BX=0000 CX=0000 DX=0000 **SP=0014** BP=0000 SI=0012 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C5F IP=0015 NV UP EI NG NZ NA PE CY 1C5F:0015 CB **RETF**
23	Dönüş adresi IP←0000H, CS←1C4BH	-d ss:0 1C5B:0000 00 00 00 00 00 00 40 00-00 00 00 04 00 00 15 00 1C5B:0010 5F 1C AC 15 **00 00 4B 1C**
24	Program sona ermiştir.	AX=0400 BX=0000 CX=0000 DX=0000 SP=0018 BP=0000 SI=0012 DI=0000 DS=1C5D ES=1C4B SS=1C5B CS=1C4B IP=0000 NV UP EI NG NZ NA PE CY **1C4B:0000** CD20 **INT 20**

11.2 Birleştirilmiş (Combined) Veri Kesimi Kullanımı

Farklı programların içinde tanımlanmış veri alanına erişmek için de **EXTRN/PUBLIC** tanımlarından yararlanılır. Ancak kod kesimlerinin birleştirilmesi örneğinde olduğu gibi uygun tanımları yapmak kaydıyla farklı iki programın veri kesimleri de tek bir 64 KB'lık bellek alanı içinde toplanabilir. Kod kesimlerinin birleştirilmesi için geçerli kurallar veri kesimi için de aynen geçerlidir.

Birleştirilmiş veri kesimi kullanımı ile ilgili tanımlar Tablo 11-5 ve Tablo 11-6'de verilen iskelet yapı üzerinde gösterilmiştir.

Tablo 11-5: Birleştirilmiş (combined) veri kesimi – BIR.ASM

```
BIR.ASM
                TITLE Birleştirilmiş data kesim  – iskelet yapı
                EXTRN Altprg:FAR, I_Word:Word
                PUBLIC B_Word
Datasg          SEGMENT BYTE PUBLIC 'DATA'
I_Byte1         DB ?
I_Byte21        DB ?
B_Word          DW 0ABCDH
Datasg          ENDS
Codesg          SEGMENT PARA 'CODE'
                ASSUME CS:Codesg, DS:Datasg,SS:……
Basla           PROC FAR
                PUSH DS             ; Dönüş adreslerinin yığında saklanması
                XOR AX, AX
                PUSH AX
                ……                  ; Diğer işlemler
                CALL Altprg
                RET
Basla           ENDP
Codesg          ENDS
                END Basla           ; Programın başlangıç notası
```

Tablo 11-6: Birleştirilmiş (combined) veri kesimi – IKI.ASM

```
IKIASM
                TITLE Birleştirilmiş data kesim  – iskelet yapı
                PUBLIC Altprg, I_Word
                EXTRN B_Word: Word
Datasg          SEGMENT BYTE PUBLIC 'DATA'
B_Byte1         DB ?
B_Byte21        DB ?
I_Word          DW 1234H
Datasg          ENDS
Cdseg           SEGMENT PARA  'code'
                ASSUME CS:Cdseg, DS:Datasg, SS:…..
Altprg          PROC FAR

                …..             ; Diğer işlemler
                RET             ; Çağırıldığı noktaya geri döner
Altprg          ENDP
Cdseg           ENDS
                END
```

Örnek 11-2

Örnek 11-2.ASM isimli programdaki yordam *I_Word* isimli harici değişkeni, kendi veri kesiminde tanımlı olan *I_Byte1* ve *I_Byte2* isimli byte tanımlı değişkenlere yerleştirmekte, ardından *ALTPRG* isimli harici bir yordamı (Örnek 11-2-1.ASM isimli program içinde yer alan) çağırmaktadır. Örnek 11-2.ASM ve Örnek 11-2-1.ASM isimli programların veri kesimlerini birleştirmek üzere her iki programın

veri kesimleri de aynı şekilde tanımlanacaktır. Veri alanında kullanılamayan bölgenin en aza indirilmesi için kesim hizalama tipi olarak BYTE kullanılacaktır. Şimdi bu bilgiler ışığında Örnek 11-2.ASM kodunu yazalım.

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 11-2.ASM
; Örnek 11-2-1.ASM ile veri kesimleri birleştirilmiştir.
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Veri kesimleri birleşik
; ---------------------------------------------------------------------
; ALTPRRG ve I_WORD Örnek 11-2-1.ASM içinde tanımlıdır.
; B_WORD ise Örnek 11-2-1.ASM içindeki ALTPRG isimli yordam erişecektir.
; ---------------------------------------------------------------------
        EXTRN ALTPRG:FAR, I_WORD:WORD
        PUBLIC B_WORD
STSEG   SEGMENT  PARA STACK 'STACK'
        DW 12 DUP(0)
STSEG   ENDS
; ---------------------------------------------------------------------
; Veri kesimi PUBLIC tanımlanmıştır. Ayrıca BYTE tanımı ile tüm alanın
; etkin kullanılması amaçlanmıştır.
; ---------------------------------------------------------------------
DTSEG   SEGMENT  BYTE PUBLIC 'DATA'
I_BYTE1 DB ?
I_BYTE2 DB ?
B_WORD  DW 0ABCDH
DTSEG   ENDS
CDSEG   SEGMENT  PARA 'CODE'
        ASSUME CS:CDSEG,SS:STSEG,DS:DTSEG
BASLA   PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
        PUSH DS
        XOR AX, AX
        PUSH AX
;---------------------------------------------------------------------
; DTSEG ismiyle tanımlı veri kesimine erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
        MOV AX, DTSEG
        MOV DS, AX
;---------------------------------------------------------------------
; işlem bloğu başlıyor
;---------------------------------------------------------------------
        MOV AX, I_WORD   ; Örnek 11-2-1.ASM içindeki değeri al
        MOV I_BYTE1, AL  ; Parçalayarak kendi içinde tanımlı değişkenlere
        MOV I_BYTE2, AH  ; yerleştir
        CALL ALTPRG      ; harici yordamı çağır
        RET              ; DOS'a dön
BASLA   ENDP
CDSEG   ENDS
        END BASLA      ; programın başlangıç noktası

C:\assembly>masm Örnek 11-2.asm;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

     51 Source  Lines
     51 Total   Lines
     14 Symbols
  50988 + 383508 Bytes symbol space free

      0 Warning Errors
```

```
      0 Severe  Errors

Microsoft (R) Macro Assembler Version 5.00                    6/24/01
Veri kesimleri birleşik                                     Page     1-1

                        PAGE 60,80
                        TITLE Veri kesimleri birleşik
                        EXTRN ALTPRG:FAR, I_WORD:WORD
                        PUBLIC B_WORD
 0000                STSEG   SEGMENT  PARA STACK 'STACK'
 0000  000C[                     DW 12 DUP(0)
    0000
          ]
 0018                STSEG   ENDS
 0000                DTSEG   SEGMENT  BYTE PUBLIC 'DATA'
 0000  ??            I_BYTE1 DB ?
 0001  ??            I_BYTE2 DB ?
 0002  ABCD          B_WORD  DW 0ABCDH
 0004                DTSEG   ENDS
 0000                CDSEG   SEGMENT  PARA 'CODE'
                        ASSUME CS:CDSEG,SS:STSEG,DS:DTSEG
 0000                BASLA   PROC FAR
 0000  1E                    PUSH DS
 0001  33 C0                 XOR AX, AX
 0003  50                    PUSH AX
 0004  B8 ---- R             MOV AX, DTSEG
 0007  8E D8                 MOV DS, AX
 0009  A1 0000 E             MOV AX, I_WORD
 000C  A2 0000 R             MOV I_BYTE1, AL
 000F  88 26 0001 R        MOV I_BYTE2, AH
 0013  9A 0000 ---- E      CALL ALTPRG
 0018  CB                    RET
 0019                BASLA   ENDP
 0019                CDSEG   ENDS
                           END BASLA

Microsoft (R) Macro Assembler Version 5.00                    6/24/00
Veri kesimleri birleşik                                     Symbols-1

Segments and Groups:
                N a m e          Length    Align   Combine Class
CDSEG  . . . . . . . . . . . . .     0019 PARA NONE 'CODE'
DTSEG  . . . . . . . . . . . . .     0004 BYTE PUBLIC   'DATA'
STSEG  . . . . . . . . . . . . .     0018 PARA STACK    'STACK'

Symbols:
                N a m e          Type  Value    Attr
ALTPRG . . . . . . . . . . . . .  L FAR   0000 External
BASLA  . . . . . . . . . . . . .  F PROC  0000 CDSEG    Length = 0019
B_WORD . . . . . . . . . . . . .  L WORD  0002 DTSEG    Global
I_BYTE1  . . . . . . . . . . . .  L BYTE  0000 DTSEG
I_BYTE2  . . . . . . . . . . . .  L BYTE  0001 DTSEG
I_WORD . . . . . . . . . . . . .  V WORD  0000 External
@FILENAME  . . . . . . . . . . .  TEXT  Örnek 11-2

     51 Source  Lines
     51 Total   Lines
     14 Symbols

  50988 + 383508 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

Örnek 11-2-1.ASM isimli program ise *B_Word* isimli harici değişkeni, kendi veri kesiminde tanımlı *B_Byte1* ve *B_Byte2* isimli değişkenlere yerleştiren *ALTPRG* isimli bir yordama sahiptir. Bu yordam, Örnek 11-2.ASM isimli program tarafından harici yordam olarak çağırılmaktadır. Veri kesimlerinin de birleştirileceğini dikkate alarak Örnek 11-2-1.ASM isimli programı yazalım;

```
; ---------------------------------------------------------------------
; PROGRAM : Örnek 11-2-1.ASM
; Örnek 11-2.ASM ile veri kesimleri birleştiriliştir.
; ---------------------------------------------------------------------
        PAGE 60,80
        TITLE Veri kesimleri birleşik
; ---------------------------------------------------------------------
; ALTPRRG ve I_WORD Örnek 11-2.ASM tarafından kullanılacaktır.
; B_WORD, Örnek 11-2.ASM içinde tanımlıdır
; ---------------------------------------------------------------------
        PUBLIC ALTPRG, I_WORD
        EXTRN B_WORD: WORD
; ---------------------------------------------------------------------
; Veri kesimi PUBLIC tanımlanmıştır. Ayrıca BYTE tanımı ile tüm alanın
; etkin kullanılması amaçlanmıştır.
; ---------------------------------------------------------------------
DTSEG   SEGMENT  BYTE PUBLIC 'DATA'
B_BYTE1 DB ?
B_BYTE2 DB ?
I_WORD  DW 01234H
DTSEG   ENDS
CODESG  SEGMENT  PARA 'CODE'
        ASSUME CS:CODESG, DS:DTSEG
ALTPRG  PROC FAR
;---------------------------------------------------------------------
; işlem bloğu başlıyor
;---------------------------------------------------------------------
        MOV AX, B_WORD   ; Örnek 11-2.ASM içindeki B_WORD değerini al
        MOV B_BYTE1, AL  ; Kendi veri kesiminde tanımlı değişkenlere koy
        MOV B_BYTE2, AH
        RET              ; çağıran noktaya geri dön
ALTPRG  ENDP
CODESG  ENDS
        END
```

Örnek 11-2-1.ASM isimli program derlendikten sonra elde edilen LST uzantılı dosya aşağıda verilmiştir.

```
C:\assembly>masm Örnek 11-2-1.asm;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.
     36 Source  Lines
     36 Total   Lines
     11 Symbols
  51014 + 383482 Bytes symbol space free
      0 Warning Errors
      0 Severe  Errors

Microsoft (R) Macro Assembler Version 5.00                  6/24/01
Veri kesimleri birleşik                                   Page     1-1

                        PAGE 60,80
                        TITLE Veri kesimleri birleşik
                        PUBLIC ALTPRG, I_WORD
                        EXTRN B_WORD:WORD
```

```
0000                DTSEG   SEGMENT  BYTE PUBLIC 'DATA'
0000  ??            B_BYTE1 DB ?
0001  ??            B_BYTE2 DB ?
0002  1234              I_WORD  DW 01234H
0004                DTSEG   ENDS
0000                CODESG  SEGMENT  PARA 'CODE'
                       ASSUME CS:CODESG,DS:DTSEG
0000                ALTPRG  PROC FAR
0000  A1 0000 E                 MOV AX,B_WORD
0003  A2 0000 R                 MOV B_BYTE1,AL
0006  88 26 0001 R          MOV B_BYTE2,AH
000A  CB                        RET
000B                ALTPRG  ENDP
000B                CODESG  ENDS
                              END

Microsoft (R) Macro Assembler Version 5.00                  6/24/01
Veri kesimleri birleşik                                   Symbols-1

Segments and Groups:
                N a m e          Length    Align   Combine Class
CODESG . . . . . . . . . . . . .     000B PARA NONE 'CODE'
DTSEG  . . . . . . . . . . . . .     0004 BYTE PUBLIC   'DATA'

Symbols:
                N a m e          Type  Value    Attr
ALTPRG . . . . . . . . . . . . .F PROC   0000  CODESG   Global  Length = 000B
B_BYTE1  . . . . . . . . . . . .L BYTE   0000  DTSEG
B_BYTE2  . . . . . . . . . . . .L BYTE   0001  DTSEG
B_WORD . . . . . . . . . . . . .V WORD   0000  External
I_WORD . . . . . . . . . . . . . L WORD   0002  DTSEG Global
@FILENAME  . . . . . . . . . . .     TEXT  Örnek 11-2-1

     36 Source  Lines
     36 Total   Lines
     11 Symbols

  51014 + 383482 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

Şimdi ayrı ayrı üretilen OBJ uzantılı dosyaları kullanarak Link işlemini gerçekleştirelim.

```
C:\assembly>Link Örnek 11-2.obj + Örnek 11-2-1.obj

Microsoft (R) Overlay Linker  Version 3.61
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [Örnek 10-2.EXE]: Örnek 11-2.exe
List File [NUL.MAP]: Örnek 11-2.map
Libraries [.LIB]:
```

Link işleminden geçtikten sonra oluşan **MAP** uzantılı dosya oluşturduğumuz programın kesimlerinin belleğin hangi adreslerinde ve ne şekilde yerleştirildiğini gös-

termektedir. Bu dosyayı incelediğimizde bellekte 4 kesim kullanılmakta olduğunu görürüz.

```
Start     Stop     Length   Name                    Class
00000H    00017H   00018H   STSEG                   STACK
00018H    0001FH   00008H   DTSEG                   DATA
00020H    00038H   00019H   CDSEG                   CODE
00040H    0004AH   0000BH   CODESG                  CODE

Program entry point at 0002:0000
```

Burada *STSEG* ve *CDSEG* isimli kesimler Örnek 11-2.ASM programı içinde tanımlı yığın ve kod kesimlerini, *CODESG* ise Örnek 11-2-1.ASM içinde tanımlı kod kesimdir. *DTSEG* ise hem Örnek 11-2.ASM hem de Örnek 11-2-1.ASM isimli programlarında tanımlanmış ve birleştirilmiş veri kesimini göstermektedir. *DTSEG* tanımında alışıla gelen PARA sözde komutu yerine BYTE kullandığımız için her iki program parçasındaki tanımlı veri birbirinin peşi sıra veri kesimi içine yerleştirilmiştir. Ayrıca BYTE tanımlandığı için *DTSEG*, *STSEG* tanımını takip eden byte sınırından itibaren yerleştirilmiştir. Her iki programda da 4 byte büyüklüğünde veri tanımı yapıldığı için kesim birleştirildiğinde büyüklüğü bu ikisinin toplamı yani 8 byte olarak görülmektedir.

Programımızı debug ortamında incelediğimizde Tablo 11-7'de gösterilen sonuçlar elde edilecektir.

Tablo 11-7: Örnek 11-2'deki programın adım adım çalıştırılması

	Kısa Açıklama	**DEBUG programının ürettiği değerler**
1	Programın dönüş adresleri yığına yerleştiriliyor. Yığın 12 word tanımlandığı için SP yazmacının değeri başlangıçta 0018H (24) dir. Her PUSH işlemi ile bu sayı iki azalır.	AX=0000 BX=0000 CX=004B DX=0000 **SP=0018** BP=0000 SI=0000 DI=0000 **DS=1C4B** ES=1C4B SS=1C5B CS=1C5D IP=0000 NV UP EI PL NZ NA PO NC 1C5D:0000 1E **PUSH DS**
2		**AX=0000** BX=0000 CX=004B DX=0000 SP=0016 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5D IP=0001 NV UP EI PL NZ NA PO NC 1C5D:0001 33C0 **XOR AX,AX**
3		AX=0000 BX=0000 CX=004B DX=0000 SP=0016 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5D IP=0003 NV UP EI PL ZR NA PE NC 1C5D:0003 50 **PUSH AX**
4	Yığına yerleştirilen DS ve AX yazmaçlarının değerleri görülmektedir.	-d ss:0 1C5B:0000 00 00 00 00 00 00 00 00-00 00 5C 1C 00 00 07 00 1C5B:0010 5D 1C AC 15 **00 00 4B 1C**-
5	Veri kesimine erişebilmek için gerekli tanımlar yapılıyor.	AX=0000 BX=0000 CX=004B DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C4B ES=1C4B SS=1C5B CS=1C5D IP=0004 NV UP EI PL ZR NA PE NC 1C5D:0004 B85C1C **MOV AX,1C5C**
6		**AX=1C5C** BX=0000 CX=004B DX=0000 SP=0014 BP=0000 SI=0000 DI=000 DS=1C4B ES=1C4B SS=1C5B CS=1C5D IP=0007 NV UP EI PL ZR NA PE NC 1C5D:0007 8ED8 **MOV DS,AX**

7	Veri kesiminin durumu	-d ss:0018 1C5B:0010 **00 00 CD AB 00 00 34 12** -d ds:0 1C5C:0000 5D 1C AC 15 00 00 4B 1C-**00 00 CD AB 00 00 34 12**
8	Örnek 11-2. ASM içindeki işlemler.	AX=1C5C BX=0000 CX=004B DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5D IP=0009 NV UP EI PL ZR NA PE NC 1C5D:0009 A10E00 **MOV AX,[000E] DS:000E=1234**
9		**AX=1234** BX=0000 CX=004B DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5D IP=000C NV UP EI PL ZR NA PE NC 1C5D:000C A20800 **MOV [0008],AL DS:0008=0**
10		AX=1234 BX=0000 CX=004B DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5D IP=000F NV UP EI PL ZR NA PE NC 1C5D:000F 88260900 **MOV [0009],AH DS:0009=00**
11	Veri kesiminin durumu.	-d ds:8 1C5C:0000 **34 12** CD AB 00 00 34 12
12	Yordam çağırma.	AX=1234 BX=0000 CX=004B DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5D IP=0013 NV UP EI PL ZR NA PE NC 1C5D:0013 9A00005F1C **CALL 1C5F:0000**
13	Dönüş adresleri yığında.	-d ss:0 1C5B:0000 00 00 00 00 00 00 34 12-00 00 00 00 5F 1C AC 15 1C5B:0010 **18 00 5D 1C** 00 00 4B 1C-
14	Örnek 11-2-1. ASM içindeki işlemler.	AX=1234 BX=0000 CX=004B DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5F IP=0000 NV UP EI PL ZR NA PE NC **1C5F:0000** A10A00 **MOV AX,[000A] DS:000A=ABCD**
15		AX=ABCD BX=0000 CX=004B DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5F IP=0003 NV UP EI PL ZR NA PE NC 1C5F:0003 A20C00 **MOV [000C],AL DS:000C=00**
16		AX=ABCD BX=0000 CX=004B DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5F IP=0006 NV UP EI PL ZR NA PE NC 1C5F:0006 88260D00 **MOV [000D],AH DS:000D=00**
17	Veri kesiminin durumu.	-d ds:8 1C5C:0000 34 12 CD AB **CD AB** 34 12
18	Dönüş değerlerini yığından al.	AX=ABCD BX=0000 CX=004B DX=0000 SP=0010 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5F IP=000A NV UP EI PL ZR NA PE NC 1C5F:000A CB **RETF**
19	IP←0018H, CS←1C5DH	-d ss:0 1C5B:0000 00 00 00 00 00 00 CD AB-00 00 0A 00 5F 1C AC 15 1C5B:0010 **18 00 5D** 1C 00 00 4B 1C-
20	Dönüş değerlerini yığından al.	AX=ABCD BX=0000 CX=004B DX=0000 SP=0014 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C5D IP=0018 NV UP EI PL ZR NA PE NC **1C5D:0018** CB **RETF**
21	IP←0000H, CS←1C4BH	-d ss:0 1C5B:0000 00 00 00 00 00 00 CD AB-00 00 0A 00 5F 1C AC 15 1C5B:0010 18 00 5D 1C **00 00 4B 1C-**
22	Program biti.	AX=ABCD BX=0000 CX=004B DX=0000 SP=0018 BP=0000 SI=0000 DI=0000 DS=1C5C ES=1C4B SS=1C5B CS=1C4B IP=0000 NV UP EI PL ZR NA PE NC **1C4B:0000** CD20 **INT 20**

Veri kesimi BYTE olarak tanımlandığı için *STSEG* isimli kesimin hemen sonundan başlamıştır. Bu durumda veriyi görmek için 2 yol vardır. Bunlardan biri yığının en son adresinden itibaren yığın alanını görüntülemektir (Tablo 11-7, satır 7). Programa başladığımız anda SP yazmacının değeri bizim için yığının sonunu belirlemektedir (Tablo 11-7, satır 1). Görüldüğü gibi SP←0018H değerini almıştır. O zaman veri SS:0018H adresinden itibaren görünebilecektir. Bu durumda 1C5B:0018H adresine karşı gelen fiziksel adreste tanımladığımız değişkenlerin yer alıyor olması gerekmektedir. Bu kesim ve göreli konum ikilisi 1C5C8H fiziksel adresini belirlemektedir. Diğer taraftan (Tablo 11-7, satır 9) 0008H adresindeki veriye erişilmektedir. Bu adres Örnek 11-2.ASM isimli programda tanımlı ilk değişkenin adresidir (*I_Byte1*). O zaman bizim veri tanımlarımız aslında DS:0008H adresinden başlamaktadır. DS değeri (Tablo 11-7, satır 6) MOV DS, AX işleminin sonunda 1C5CH değerini aldığına göre veri 1C5C:0008 kesim ve göreli konum ikilisi ile ifade edilen fiziksel adreste bulunacaktır. Bunun karşılığı olan fiziksel adres de 1C5C8H'dır. Görüldüğü gibi he iki durumda da aynı fiziksel adrese bakıldığı için aynı değerleri görmek mümkün olabilmektedir.

12. YORDAMLAR VE PARAMETRE AKTARMA YÖNTEMLERİ

Yazdığımız yordamlar arasında bir takım değerlerin verilip alınması işlemi parametre aktarma olarak adlandırılmaktadır. Parametre aktarımında dikkat edilmesi gereken en önemli nokta aktarılması düşünülen parametrenin sayısıdır. Bu sayıya bağlı olarak parametre aktarma yöntemlerinden uygun olanının seçilmesi gerekir. Eğer az sayıda parametre aktarımı söz konusu olacak ise bunlar değer (by value) olarak aktarılabilir. Ancak çok miktarda parametre aktarımı söz konusu ise bunları değer olarak aktarmak her zaman mümkün olmayacaktır. Bu gibi durumlarda adres (by reference) ile karşı tarafa parametre aktarmak daha uygundur.

Yordamlara parametre aktarmak için üç temel yöntem vardır. Bunlar;

1. Yazmaç üzerinden parametre aktarma (değer aktarımı)
2. Yığın üzerinden parametre aktarma (değer aktarımı)
3. PUBLIC/EXTRN tanımları yardımıyla ortak veri tanımı yaparak parametre aktarma (adres aktarımı)

12.1 Yazmaç Üzerinden Parametre Aktarma

Aktarılacak parametre sayıca az ise yazmaçlardan yararlanmak en hızlı ve kolay yollardan birisidir. Parametre aktarmak için genelde veri yazmaçları (AX, BX, CX, DX) ve indis yazmaçları (SI, DI) kullanılır. Bunlar dışından kalan yazmaçların parametre aktarımı için kullanılması alışıldık bir durum olmadığı gibi kesim ve işaretçi grubu yazmaçlar özel görevleri nedeniyle parametre aktarımında kullanılmaya uygun değillerdir. Bununla birlikte veri ve indis yazmaçlarından da bazılarının programda yapılan işlemlerin gereği olarak kullanımda olabileceği de düşünülmelidir. Kısaca yazmaç üzerinden parametre aktarımı 2–3 parametre için oldukça hızlı bir çözümdür. Zira yazmaç kümesi işlemci üzerinde çalıştırılan her türlü program tarafından paylaşılan niteliktedir. Genel olarak aktarılması düşünülen parametre yazmaca yerleştirildikten sonra gerekli yordam (veya kesme[141]) çağırılacaktır. Benzer şekilde çağırılan yordam da (veya kesme de) kendisini çağıran koda dönüş değerini yazmaç(lar) kullanarak aktarabilir. Daha önce incelediğimiz Örnek 10-1, Örnek 10-2, Örnek 10-3 ve Örnek 11-1 de yazmaç kullanılarak parametre aktarılmıştır.

12.2 Yığın Üzerinden Parametre Aktarma

Değer olarak parametre aktarma yollarından biri de yığını kullanmaktır. Yığının yordamlar arasında paylaşılır nitelikte bir alan olması onun parametre ak-

[141] Kesmeler de yordamlara benzer kullanım özelliklerine sahiptir. Kesmeler ile ilgili ayrıntıları 14.bölümde bulabilirsiniz.

tarımında kullanılmasına imkân sağlamaktadır. Ancak yığının parametre aktarmak amacıyla kullanılabilmesi için aktarılması düşünülen parametreleri barındırabilecek büyüklükte tanımlanması gereklidir. Ayrıca yığının yordam çağırma işlemlerinde dönüş adreslerini tuttuğu da düşünüldüğünde, yığın üzerinde tutulacak (unutulacak) herhangi bir değer, çağırılan noktaya dönüşü imkânsız hale getirecektir. Benzer şekilde yığından gereğinden fazla değerin alınması da aynı sonucu doğurur. Kısaca yığın kullanımı sırasında her PUSH işleminin karşılığı olarak bir POP işlemi veya buna eşdeğer sonuç üretecek türde başka işlemlerin[142] yapılması gerekmektedir. Yığın üzerinden aktarılacak parametrelerin yordam çağırılmadan önce **PUSH** komutları yardımıyla yığına yerleştirilmesi gereklidir. Ancak bundan sonra CALL komutu kullanılarak programın akışı ilgili yordamdan devam edecek şekilde değiştirilebilir. Çağırılan yordamın içinden yığında bulunan parametrelere erişmek için **POP** komutu kesinlikle kullanılmamalıdır. SP yazmacının değeri kullanıcı tarafından değiştirilerek yığın üzerinde herhangi bir alana erişilmeye çalışılması daha da büyük problemlere[143] neden olacaktır. Bu durumda yığın alanı üzerindeki parametrelere erişim için BP yazmacından yararlanılmalıdır.

Yığın üzerinden parametre aktarma işlemi için kullanılacak iskelet program yapısı Tablo 12-1'de verilmiştir. *ILK.ASM* isimli programdan *Word1* ve *Word2* isimli değişkenler yığına **PUSH** komutları yardımıyla yerleştirilmişlerdir (satır 18-19). Parametrelerin yığına yerleşmesini takiben onlar üzerinde işlem yapması planlanan *Altprg* isimli, FAR olarak tanımlanmış harici yordam (satır 2) CALL komutu (satır 20) ile çağırılmıştır. Çağırılan yordamın FAR tanımlanmış olması nedeniyle dönüş adresi olarak yığında kesim ve göreli konum ikilisinin saklanması gerekecektir. Buna göre yığının büyüklüğünün ihtiyacı karşılayacak biçimde belirlenmesi gerekmektedir. *ILK.ASM* isimli program komut satırından çağırıldığı için geri dönebilmek üzere yığında 2 word dönüş adresi zaten yerleştirilmiş durumdadır (satır 14, 16). Buna ek olarak yığına parametre olarak yerleştirilmiş olan 2 word daha vardır. Çağırılan FAR yordamdan geri dönülebilmesi için gerekli olan 2 word'lük dönüş adresi de düşünüldüğünde toplam 6 word'lük yığın alanının kullanıldığı açıkça görülmektedir. Bu kullanım miktarına bakarak yığını ucu ucuna yetecek kadar tanımlamak doğru bir yaklaşım değildir. Zira ilerleyen başlıklar altında anlatacağımız -sistemin sürekliliğini sağlamak üzere belirli aralıklarla devreye girip çıkan- kesmeler de yığın alanını

---

[142] Bazı durumlarda dönüş işlemini gerçekleştiren RET komutunun sonuna yazılacak olan rakam ile SP yazmacının kaç byte arttırılacağı belirlenebilir.

[143] SP yazmacının gösterdiği adresin altında kalan –yani daha büyük adrese sahip olan yığın gözlerine denk gelen– adreslerdeki değerler koruma altındadır. SP yazmacı kullanıcı tarafından daha büyük bir adres alanına alışılagelmiş şekillerin dışında ayarlanacak olursa SP yazmacının eski ve yeni değerleri arasında kalan yığın içeriğinin geçerliliği kalmaz. Bu nedenle SP yazmacının kullanıcı tarafından doğrudan değiştirilmemesi gereklidir.

kullanmaktadır. Kesmeler de dikkate alınarak yığının ihtiyacın üzerinde bir değer verilerek tanımlanması alışılmış bir durumdur. Ancak bu değeri çok büyük seçmek de doğru bir yaklaşım değildir. Programcı, yığın büyüklüğünü zaman içerisinde kazanacağı deneyim ile uygulamanın gereklerine dikkat ederek belirlemelidir.

Tablo 12-1: Yığın üzerinden parametre aktarımı – ILK.ASM

ILK.ASM

```
        TITLE yığın üzerinden parametre aktarımı
        EXTRN Altprg: FAR
Stseg   SEGMENT PARA STACK 'STACK'
        DW 16 DUP(?)
Stseg   ENDS
Datasg  SEGMENT BYTE 'DATA'
Word1   DW 1234H
Word2   DW 5678H
B_Word  DW 0
Datasg  ENDS
Codesg  SEGMENT PARA 'CODE'
        ASSUME CS:Codesg, DS:Datasg, SS: Stseg......
Basla   PROC FAR
        PUSH DS         ; Dönüş adreslerinin yığında saklanması
        XOR AX, AX
        PUSH AX
        ......          ; Diğer işlemler
        PUSH Word1      ; Parametreler yığına basılır
        PUSH Word2
        CALL Altprg
                        ; Diğer işlemler
        RET
Basla   ENDP
Codesg  ENDS
        END Basla       ; Programın başlangıç noktası
```

Parametreleri aktaracağımız yordam çağırıldığı anda yığının durumu incelendiğinde Tablo 12-2'de gösterilen yapı ile karşılaşılacaktır.

Tablo 12-2: Yığının Durumu - Altprg çalışmadan önce

Adres (SP)	15 … 0	
0010H		
0012H		
0014H	Dönüş (göreli konum)	←SP
0016H	Dönüş (kesim)	
0018H	*Word2*	
001AH	*Word1*	
001CH	Komut satırına dönüş (göreli konum)	
001EH	Komut satırına dönüş (kesim)	

ILK.ASM isimli programda (Tablo 12-1, satır 4) yığın 16 adet word olarak tanımlanmıştır (20H adet byte). Buna göre yığın kesimi içinde işaretçi olarak kullanılan SP yazmacının değeri 001EH olacaktır (yığına ilk yerleştirilen değer 001EH-001FH adresli byte'lara yerleşmiştir). Komut satırına dönüşü sağlamak için dönüş kesim adresi ve göreli konum adresleri de yığına yerleştirilmiştir. Ardından *Word1* ve *Word2* isimli word tipindeki değişkenler yığına parametre olarak yerleştirilmiştir (satır 18–19). FAR olarak tanımlı harici yordamın çağrılmasıyla programın 21. satırdaki komutu işlemek için dönebilmesini sağlamak üzere gerekli kesim ve göreli konum değerleri de yığına yerleştirilecektir (satır 20).

Şimdi parametrelere erişerek bunlar üzerinde işlem yapacak olan program parçasının iskelet yapısını Tablo 12-3 üzerinde inceleyelim. Yığın üzerinden parametrelere erişmek isteyen yordam bu işlemi yaparken BP yazmacından yararlanmaktadır. BP yazmacı, SP yazmacı gibi yığın alanının başlangıcını gösteren SS kesim yazmacı ile birlikte kullanılır. Ancak BP yazmacının, SP yazmacından farklı olarak, değeri kullanıcı tarafından değiştirilebilmektedir. Bu sayede kullanıcı yığını bir dizi gibi kullanabilmekte ve yığın üzerinde istediği veriye rahatlıkla ulaşabilmektedir.

Tablo 12-3: Yığın üzerinden parametre aktarımı – IKI.ASM

```
IKIASM
 1                TITLE yığın üzerinden parametreleri alan program
 2                PUBLIC Altprg
 3    Codesg      SEGMENT PARA  'code'
 4                ASSUME CS: Codesg
 5    Altprg      PROC FAR
 6                PUSH BP
 7                MOV BP, SP
 8                MOV AX; [BP+6]      ; AX←Word2
 9                MOV BX, [BP+8]      ; BX←Word1
10                .....               ; Diğer işlemler
11
12                POP BP
13                RET 4               ; Çağırıldığı noktaya geri dön
14    Altprg      ENDP
15    Codesg      ENDS
16                END
```

Tablo 12-3'de verilen yordamın hemen başında BP yazmacının da yığın üzerinde saklandığı görülmektedir (satır 6). Çağrılan yordamın görevini yerine getirmek üzere kullandığı yazmaçların o anda sahip oldukları değerler çağıran yordam açı-

sından önemli olabilir. Bu yazmaçlardaki değerlerin bozulmasını engellemek üzere yığında saklanmaları ve çağıran yordama geri dönülmeden hemen önce yığına konuş sırasının tersinde ilgili yazmaçlara aktarılmaları gereklidir. Bu sayede çağırılan kodlar, kendilerini çağıran kodların çalışmalarını engellemez ve hemen hemen her türlü program ile birlikte kullanılabilirler. BP yazmacının değeri de PUSH komutu ile yığın üzerinde geldiği yerde kullanımda olabileceği düşüncesi ile saklanmaktadır[144]. *Altprg* isimli yordama gelindiğinde yığının durumunu Tablo 12-4 üzerinde inceleyecek olursak;

Tablo 12-4: Yığının durumu – PUSH BP yapıldıktan sonra

Adres (SP)	15 … 0	
0010H		
0012H	BP	←SP[145] ←BP
0014H	Dönüş (göreli konum)	←BP +2
0016H	Dönüş (kesim)	←BP +4
0018H	*Word2*	←BP +6
001AH	*Word1*	←BP +8
001CH	Komut satırına dönüş (göreli konum)	
001EH	Komut satırına dönüş (kesim)	

BP yazmacı PUSH edildikten sonra SP yazmacının değeri bu yazmaca yüklenmektedir (satır 7). Böylece BP yazmacının değeri kullanıcı kontrolünde belirlenerek yığına bir dizi gibi erişme şansı elde edilmiştir. Yığın adresleri incelendiğinde yığının altında kalan değerlere erişmek için BP yazmacındaki değerin arttırılması gereklidir. Bunu yaparken erişilmesi düşünülen parametrenin adresi hesaplanmalıdır. *Word1* isimli değişkene erişebilmek için [BP+8] adresli[146] bellek içeriğine bakmak yeterli olacaktır (satır 9). Yığın alanındaki parametrelere BP yazmacı ile erişilmesinin en büyük avantajı, herhangi bir PUSH işlemi yapılması durumunda bile BP yazmacının değerinin değişmemesidir. Böylece yığındaki parametrelere erişmek için yeniden adres hesabı yapılması gerekmez.

[144] Benzer şekilde çağırılan yordam içinde kullanılan diğer yazmaçların da çağırıldıkları yerdeki değerlerinin kayıp olmaması için yığında saklanmaları gereklidir. Hatta bazı özel durumlarda bayrak değerlerinin de yığında saklanması gerekebilir. Kesme kullanımında bayrakların saklanması gerekli olacağı için INT komutu tarafından bu işlem otomatik olarak yapılmaktadır. Diğer yandan eğer herhangi bir yazmaç sonuç değerini geri döndürmek için kullanılacak ise önceki değerlerinin yığında saklanması gerekmez.

[145] 80286 ve üst model işlemcilerde SP yığın üzerinde bulunan boş gözü göstermektedir. 80286 veya daha üst model işlemciler için üretilmiş kodlarda buna dikkat ederek gerekli bağıl adres hesaplamaları yapılmalıdır. PUSH ve POP komutlarının çalışmasında bir değişiklik olmayıp kullanıcıları etkilemez.

[146] Bağıl adres (+8 değeri), yordam içinde yığında saklanan başka değerlerin olması durumunda değişecektir. Programcı bu değerleri göz önüne alarak gerekli hesaplamayı yapmalı ve bağıl adresi tespit etmelidir.

Çağırılan yordamdan dönülmeden önce programın başında kullanılan PUSH BP komutunun karşılığı olan POP BP komutunun[147] kullanılması (satır 12) gerekir. Bu sayede SP yazmacı dönüş adreslerinin olduğu noktayı gösterir hale gelmiş olur ve RET (satır 13) komutuyla çağıran yordama dönülür. RET komutu doğası gereği dönüş adreslerini ilgili yazmaçlara yerleştirdikten sonra SP yazmacının değerini de ayarlayacaktır. Bu işlemden sonra yığının durumu Tablo 12-5'te gösterildiği gibi olacaktır. Bu noktada sorulması gereken önemli bir soru vardır. "*Parametre olarak aktarılmak üzere yığına yerleştirilen değerleri oradan çekmek/almak kimin sorumluluğunda olacaktır ?*" Bu sorunun cevabı uygulamanın niteliğine bağlı olarak değişebilir. Eğer çağırılan yordam, aktarılan parametreler üzerinde işlem yapmış ve bunun sonuçlarını yine yığında parametreler[148] için ayrılmış alana yazarak kendisini çağıran programa aktarmayı amaçlamışsa bu değerleri yığından çekmek yine çağıran tarafın görevidir. Çağıran taraf bunları uygun şekilde POP ederek gerekli değişkenlere veya yazmaçlara aktararak işlemlerine devam edecektir. Ancak çağırılan yordamdan dönüldüğünde yığın üzerindeki değerler önemli değillerse iki olasılık oluşmaktadır. İlk olasılık bu değerleri yığına koyanın kaldırmasıdır. Parametreleri yığına koymak için kullanılan PUSH komutu kadar POP komutu ile bunlar yığından geri alınırlar. Bu işlem programın yavaşlamasına neden olacaktır. Zira her POP işlemi için yığına erişilmekte, bir word alınarak bir yazmaç veya bellek alanına aktarılmakta, ardından SP yazmacının değeri (POP komutu tarafından) 2 byte arttırılmaktadır. Üstelik bu işlem hiç kullanılmayacak olan veriyi çekmek (kısaca programın doğru bir şekilde sonlanmasını sağlayacak dönüş adreslerine ulaşmak) için yapılmaktadır. Diğer olasılık hem daha kolay hem de başarımı ilkine göre daha iyi bir çözüm sunmaktadır. Bu çözüm, çağırılan yordamın geri dönmesini sağlayan RET komutunun yanına yazılan değer ile SP yazmacının değerinin arttırılmasıdır. Örneğimizde (satır 13) **RET 4** komutu, önce dönüş adreslerini yığından alınarak (SP←0018H değerini alır) ardından **SP←SP + 4** işlemini yapılmaktadır (SP←001CH değerini alır). Bu basit bir toplama işlemi olması nedeniyle ilk olasılıktan daha etkin bir sonuç verecektir. Dikkat edilmesi gereken nokta RET komutundan sonra yazılan sayının aktarılan parametre sayısı ile orantılı olmasıdır. Yığın üzerindeki işlemlerin hep word olarak yapıldığı düşünüldüğünde burada verilecek olan sayının çift olması gerektiği hemen söylenebilir. Eğer ***n*** adet parametre aktarımı gerçekleşmiş ise ve bunların çağırılan yordam tarafından yığından kaldırılması isteniyorsa RET komutunun yanına yazılacak sayı ***2*n*** ile elde edilen bir değer olmalıdır.

---

[147] Benzer amaçla başka yazmaç değerleri de yığın üzerinde saklanmış ile bunlarından yordamdan dönülmeden önce yığından alınması gerekir.

[148] Dönüş değerlerinin işgal edeceği yer, giriş değerlerinin işgal ettiği yere eşit büyüklükte veya küçük ise bu yöntem kullanılabilir.

Tablo 12-5: Yığın durumu - Altprg'den dönüldükten sonra

Adres (SP)	15 … 0	
0010H		
0012H		
0014H		
0016H		
0018H	*Word2*	←SP
001AH	*Word1*	
001CH	Komut satırına dönüş (göreli konum)	
001EH	Komut satırına dönüş (kesim)	

12.3 EXTRN/PUBLIC Kullanımı ile Parametre Aktarımı

Daha önceki örneklerimizde EXTRN/PUBLIC kullanımı ile nasıl parametre aktarıldığı konusu üzerinde durmuştuk. Bu şekilde parametre aktarımında adres kullanıldığı (by reference) için daha fazla veri, daha az zahmet ile karşı tarafa aktarılabilmektedir. Tablo 12-6'da verilen iskelet kod parçasında *Ack* isimli 100 byte büyüklüğünde bir dizi ve *Eleman* isimli değişken **PUBLIC** tanımlanarak (satır 2) başka bir kod tarafından kullanılabilir kılınmıştır.

Tablo 12-6: EXTRN/PUBLIC kullanımı ile parametre aktarımı - veri içeren program

```
1                   TITLE EXTRN/PUBLIC ile parametre aktarımı
2                   PUBLIC Ack, Eleman
3   Stcksg1         SEGMENT PARA STACK 'STACK'
4                   DW 12 DUP (?)
5   Stcksg1         ENDS
6   Datasg1         SEGMENT BYTE 'DATA'
7   Ack             DB 100 DUP (?)
8   Eleman          DW 100
9   Datasg1         ENDS
10  Codesg1         SEGMENT PARA 'CODE'
11                  ASSUME CS:Codesg1, DS:Datasg1, SS:Stcksg1
12  Basla           PROC FAR
13                  PUSH DS         ; Dönüş adreslerinin yığında saklanması
14                  XOR AX, AX
15                  PUSH AX
16                  ......          ; Diğer işlemler
17                  RET
18  Basla           ENDP
19  Codesg1         ENDS
20                  END Basla       ; Programın başlangıç notası
```

Benzer şekilde Tablo 12-7'deki program parçasında, **EXTRN** tanımı ile *Ack* isimli byte, *Eleman* isimli word tipinde (satır 2) harici veri alanlarına erişilmek istendiği belirtilmiştir.

Dikkat edilecek olursa Tablo 12-6'da verilen iskelet kod parçasında tanımlı olan *Ack* isimli dizi yapısındaki değişkene Tablo 12-7'de verilen iskelet kod parçasındaki *Altprg* isimli yordamın erişerek bir takım işlemler yapması hedeflenmiştir. Ancak *Altprg* isimli yordam, üzerinde işlem yapacağı dizinin başlangıç adresini ve veri tipini bilmesine rağmen dizinin uzunluğu (eleman sayısı) hakkında bir bilgiye sahip değildir. O zaman eleman sayısının da *Altprg* isimli yordama bir şekilde parametre olarak aktarılması gerekecektir. Hatırlanacağı üzere az sayıda parametre aktarmak için yazmaçların kullanılması en uygun yöntemdir. Ancak bu örneğimizde bu yöntemin kullanılması mümkün değildir. Zira *Altprg* isimli yordam ile Tablo 12-6'da verilen program arasında (*Ack* isimli değişkeni paylaşmak dışında) bir kod ilişkisi yoktur. Kısaca Tablo 12-6'da ki program dilimi içinde *Altprg* isimli yordam çağırılmadığı için çağırma işlemi öncesinde bir yazmaç yardımı ile *Eleman* sayısının parametre olarak aktarılması da mümkün olamaz. O zaman dizinin boyutunu belirleyen *Eleman* değişkenin de EXTRN/PUBLIC tanımları kullanılarak paylaşılması ve *Altprg* isimli yordamın ihtiyaç duyduğu anda bu değişkeni kullanması sağlanmalıdır.

Tablo 12-7: EXTRN/PUBLIC kullanımı ile parametre aktarımı – veri kullanan program

```
1                   TITLE EXTRN/PUBLIC ile parametre aktarımı
2                   EXTRN Ack:Byte, Eleman: Word
3   Stcksg2         SEGMENT PARA STACK 'STACK'
4                   DW 10 DUP (?)
5   Stcksg2         ENDS
6   Datasg2         SEGMENT BYTE 'DATA'
7   B_Byte1         DB ?
8   B_Byte21        DB ?
9   Datasg2         ENDS
10  Codesg2         SEGMENT PARA 'CODE'
11                  ASSUME CS:Codesg2, DS:Datasg2, SS:Stcksg2
12  Altprg          PROC FAR
13                  ......          ; Dönüş ve veri kesimi erişimi için gerekli kodlar
14                  LEA SI, Ack     ; Acık isimli değişke erişiliyor.
15                  .....           ; Diğer işlemler
16                  RET             ; Çağırıldığı noktaya geri dön
17  Altprg          ENDP
18  Codesg2         ENDS
19                  END Altprg      ; Programın başlangıç notası
```

Her iki kod parçası kendi başına çalışabilir olmakla beraber Tablo 12-7'deki programın, Tablo 12-6'da tanımlı *Ack* isimli veri alanını kullanabilmesi için her iki koda ait OBJ uzantılı dosyaların birlikte Link işleminden geçirilmesi gereklidir[149].

[149] Her iki programda da END komutunu takiben bir başlangıç noktası belirlenmiştir. Bu nedenle Link işlemine parametre olarak verilecek olan OBJ dosyalarının sırası da oluşturulacak EXE uzantılı programın hangi yordam ile çalışmaya başlayacağını belirleyecektir. *Altprg* isimli yordamın öncelikli olarak çalışması ve diğer programda yer alan veriyi kullanarak işlem yapabilmesini sağlamak üzere link işlemine *Altprg* isimli yordamı barındıran OBJ dosyasının daha sonra ise içinde veri tanımını barındıran OBJ uzantılı dosyanın verilmesi gerekecektir.

13. SİSTEMİN ÇALIŞMAYA BAŞLAMASI (START UP)

8088/8086 işlemcili bir bilgisayar sistemi açıldığı zaman fiziksel olarak FFFF0H adresinde bulunan komutu işlemek üzere tasarlanmıştır. Bilgisayar ilk durumuna döndürüldüğünde veya kapatılıp/açıldığında bayraklar, IP ve kesim yazmaçları aşağıdaki değeri alacaktır.

PSW←bayrakların hepsi sıfırlanır (clear)

IP←0000H

CS←0FFFFH

DS←0000H

ES←0000H

SS←0000H

Bilgisayar açıldığında gerçekleştirilen işlemler basitçe söyle sıralanabilir;

1. Bilgisayarın açılması ile birlikte dâhili güç kaynağı devreye girerek kısa bir süre içinde bilgisayar sisteminin parçalarının düzgün çalışabilmesi için gerekli gücü sağlar. Ana kart üzerindeki elektronik devre elemanlarının çalışma karakteristiklerine uygun güç sağlanamaz ise bir problem olduğu düşünülerek işlemci yollanan sinyal ile ilk durumuna döndürülür. Bu işlem ana kartın güç kaynağından gelen sinyal doğru oluncaya (power good) kadar devam eder.

2. Bilgisayar sistemlerinde kullanılan bellek üniteleri (DRAM) beslendiği ve belli zamanlarda tazelendiği (refresh) sürece veri tutma özelliğindedir. Bu nedenle sisteme ilk defa güç verildiği anda bellekte yürütülebilecek herhangi bir kod veya anlamlı bir veri bulunmamaktadır. İşlemci üreticileri, sisteme her güç verildiğinde işlemcilerini aynı bellek alanına erişerek kod çalıştıracak şekilde tasarlamışlardır. CS ve IP yazmaçlarının ilk değerlerinden de anlaşılacağı gibi ilk erişilen fiziksel adres FFFF0H olacaktır. Bu adres ROM üzerinde yer alan BIOS alanına denk gelmekte ve bu nokta sistemin çalıştırılması için gerekli kodun olduğu adrese dallanmayı sağlayan bir JMP komutu yer almaktadır.

3. JMP komutu ile dallanılan adreste, BIOS üzerinde yer alan POST (Power On Self Test) olarak adlandırılan bir takım test programları yer almaktadır. Bu test programlarının yapısı ve içeriğinde yer alan testler BIOS ve ana kart üreticisine göre farklılıklar gösterebilir. Ancak ortak noktaları yürüttükleri testlerin başarısız olması durumunda bir takım sesli uyarılar vermeleri ve sistemin açılışını durdurmalarıdır. Yürütülen testler başarı ile tamamlanmış ise

INT 19H (sistem warm start) kodu çağırılarak ana karta bağlı olan çevre birimler tespit edilmeye çalışılır.

4. BIOS (INT 19H) öncelikle görüntü kartını arar ve bu karta ait olan özel kodları çalıştırmak üzere kart üzerinde yer alan BIOS'daki kodu çalıştırarak görüntü kartının kullanıma hazır hale gelmesini sağlar (initialization). Bu aşamada görüntü kartı ile ilgili bazı bilgiler ekranda belirecektir.

5. Sisteme bağlı bulunan ve kendisine has BIOS ROM'ları olan diğer çevre birimleri aranır ve kullanıma hazır hale gelmeleri sağlanır. Normalde sistemde bulunan disket sürücü (floppy drive) 0000:7C00 adresinde, IDE/ATA sabit disk ise C800:0000 adresinde yer almaktadır.

6. Mevcut çevre birimler tespit edildikten sonra ekranda BIOS ve sistemde bulunan çevre birimlerine ait bilgiler belirir.

7. BIOS yapması gerekenleri tamamladıktan sonra görevi işletim sistemine bırakmak üzere çalıştırabilir bir işletim sistemi kodu aramaya başlar[150]. Bunun için öncelikle disket sürücü, sabit disk sürücüsü, SCSI sürücü, CDROM sürücüsü, ağ bağlantısı ve USB sürücüler aranmaya başlar. Bunların hangi sırada devreye girecekleri BIOS üzerindeki bazı parametrelerin ayarlanması ile kullanıcı tarafından belirlenir[151].

8. Açılışı sağlayacak sürücü (boot drive) üzerinde yer alan işletim sistemi kodu belleğe yerleştirir. Bu sayede BIOS kontrolü işletim sistemine bırakmış olur. Eğer sürücüler üzerinde geçerli bir işletim sistemi bulunmuyor ise sistem çalışmaya devam edemeyeceği için var ise ROM BASIC çalıştırılacak, eğer BIOS'da bununla ilgili bir kod yer almıyor ise sistemin durdurulduğunu ve işleme devam edilemeyeceğini gösterir bir mesaj çıkacaktır.

Bilgisayar ilk açıldığı zaman mutlaka POST aşamasından geçer ve bu işlem en baştan başlatmak (**cold start**) olarak adlandırılır. Ancak Ctrl-Alt-Del tuşlarına basılarak bilgisayar ilk durumuna döndürülmüş ise sıcak başlatma (**warm start**) kabul edilir ve POST aşamasından geçilmez. Bilgisayar, başlatmayı takiben POST aşamasından geçip geçmeyeceğine 0040:0072H adresine bakarak karar verir. Bu adreste 1234H değeri var ise sıcak başlatma, farklı bir değer var ise baştan başlatma yapılması gerektiğine karar verecektir. Test işlemleri tamam-

---

[150] Önyükleme tutanağı (boot record) 0 numaralı sürücünün (bootable drive) 0 numaralı yüzünün (side), 0 numaralı izinin (track) 1 numaralı kesiminde (sector) bulunan kayıttır. Üreticilerin ürünlerinden bağımsız olarak çalışabilmek amacıyla işletim sistemleri her zaman aynı adreste bulunan kendilerine has kodun belleğe yüklenmesi ile devreye girer ve sistemin kontrolünü ele alırlar.

[151] BIOS üreticisi firmaya göre değişiklik göstermek ile birlikte BIOS ayarlarına girmek için yeni nesil bilgisayarlarda F1, F2 gibi fonksiyon tuşları veya Esc, Del gibi tuşlar kullanılabilir. F10 genel olarak işletim sisteminin hangi birim üzerinden yükleneceğini belirlemek için kullanılmaktadır Daha eski bilgisayar sistemlerinde (Ctrl+Alt+Esc), (Ctrl+Alt+Ins), (Ctrl+Alt+Enter) veya (Ctrl+Alt+S) gibi karmaşık tuş kombinasyonları da kullanılmıştır.

landıktan sonra sistemi başlatmak üzere önyükleme tutanağını (boot record) belleğin 007C:0000H adresine yerleştirir. Bu adresteki kodun çalıştırılması ile kontrolü ele alan işletim sistemidir. Örneğin MS-DOS[152] tabanlı sistemlerde önyükleyici (bootstrap) kontrolü ele aldıktan sonra IO.SYS ve MSDOS.SYS isimli dosyalar yüklenir, disk yapısı, gerekli çevre birim sürücüleri (device drivers) bu dosyalardan okunan bilgiler sayesinde belirlenir. Her işletim sistemi kendine ait bir açılış kodu içermektedir. Kullanıcılar bazen kullandıkları sistemin kontrolünü kendi ellerine geçirmek isteyebilirler. Böyle bir istek genelde yeni bir işletim sistemi yazmak veya işletim sistemini devre dışı bırakmak ihtiyacından doğar. Bunun için POST biter bitmez kontrolü ele almak en uygun zamandır. Tabii kontrolü ele alacak olan kodun önyükleyici kodu şeklinde yazılması gerekmektedir.

13.1 DOS Bellek Haritası

Bilgisayar sistemi üzerinde çalıştırılması istenen her kodun belleğe yüklenmiş olması gereklidir. Bu nedenle kesme servis programları da (gerek BIOS, gerek işletim sistemine ait olanlar) sistem belleğine yerleştirilmelidir. BIOS kendine ait olan kod kısımlarını ROM bütünleşik devresi üzerinde tutar ve bu alana bellek alanı gibi erişilmesine imkân sağlar. İşletim sisteminin çevre birimler ile iletişimini sağlamak üzere kullanılan sürücüler IO.SYS isimli, kesme servis programları da MSDOS.SYS isimli dosyalar yardımıyla işletim sisteminin yüklenmesi sırasında belleğe aktarılır.

Kesme servis programlarına ihtiyaç duyuldukça rahat bir şekilde erişilebilmesi için kesim ve göreli konum (4 byte) adresleri vektör tablosu adı verilen bir bellek alanında saklanır. Vektör tablosu belleğin başından itibaren (fiziksel olarak 00000H adresinden) yerleşmiş, 1024 byte uzunluğunda bir tablodur.

İşletim sistemi de (COMMAND.COM) bellek üzerinde iki parça şeklinde yer almaktadır. Bunlardan biri yerleşik parça (resident part) olarak isimlendirilen kısımdır. Yerleşik parça INT 22H (Terminate Process Exit Address), INT 23H (Ctrl-C Handler address), INT 24H (Critical Error Handler Address) ve COMMAND.COM'un geçici (transient part) olarak adlandırılan diğer parçasını[153] yükleyebilmek için gerekli kod parçasını barındırır. COMMAND.COM'un geçici parçası belleğin yüksek adreslerinde yer alır. Geçici parça içinde, işletim sisteminde kullanılan dâhili komutlar (dir, cd, md gibi), komut yorumlayıcısı (command interpreter) ve toplu (batch) işlemeye imkân veren kodları barındırır. DOS'a ait

[152] IBM-DOS için IBMIO.COM ve IBMDOS.COM isimli dosyalar yüklenir.

[153] Boş bellek alanının mevcut istekleri karşılayamadığı özel durumlarda işletim sistemine ait geçici parça bellekten uzaklaştırılarak uygulamalara yer açılabilir. Uygulamaların sona ermesi ile işletim sisteminin yerleşik parçası, geçici parçayı, açılış sırasında kullanılan sürücüden (boot drive) yeniden yükler.

EXE ve COM uzantılı yardımcı programlar ise geçici program alanına yüklenmeleri halinde çalıştırılabilirler. En yüksek adres alanında ise ROM ve video işlemlerinde kullanılan ara bellek yer almaktadır. Kullanıcı programları da geçici program alanı olarak tanımlanan alana yüklendikten sonra çalıştırılabilir. Çalışmanın sona ermesini takiben işletim sistemi tarafından bu alan boşaltılarak daha sonra gelecek istekler için kullanılır (Tablo 13-1).

Tablo 13-1: DOS bellek haritası

FFFFFH	ROM ve video ara belleği
	COMMAND.COM (geçici parçası)
	Program alanı (geçici) İşletim sisteminin harici komutları ve diğer yardımcı programlar
	COMMAND.COM (yerleşik parçası)
	DOS ara belleği, kontrol alanları ve yüklü çevre birimi sürücüleri
	MSDOS.SYS
	IO.SYS ve çevre birim sürücüleri (resident)
00000H-003FFH	Kesme vektör tablosu

PC-XT olarak adlandırılan bilgisayar sistemlerinde bellek alanlarının hangi adresten başladığı ve bittiği konusunda basit bir fikir vermesi açısından Tablo 13-2 incelenebilir.

Tablo 13-2: DOS Bellek adresleri

Blok	Adres	Tipi	Kullanım Amacı
0	00000-0FFFFh	RAM	Kesme vektör tablosu
1-9	10000-9FFFFh	RAM	DOS ve kullanıcı için ayrılmış bellek alanı
10	A0000-AFFFFh	RAM	Video belleği için ayrılmış alan
11	B0000-BFFFFh	RAM	Mono ve renkli video belleği
12	C0000-CFFFFh	ROM	BIOS – Sabit dik programları
13-15	D0000-FFFFFh	ROM	BIOS

14. KESME (INTERRUPT) NEDİR? NE AMAÇLA KULLANILIR?

Kesmelerin temel kullanım amacı, yavaş birimler ile hızlı birimler arasındaki bağlantıyı sağlarken işlemcinin bekleyerek zaman kaybetmesini engellemektir. Hepimizin bildiği gibi, çevre birimleri bazı mekanik kısıtlardan dolayı işlemciden daha yavaş çalışmaktadır. Buna göre işlemci ile yavaş birimler arasındaki ilişki aşağıda açıklanan üç temel stratejiden biri kullanılarak kurulur.

1. **Çevre birimi beklemek:** İşlemcinin çevre birimi, çevre birimin ihtiyaç duyduğu anda servis vermek için beklemesidir. Bu işleme meşgul beklemek (**busy waiting)** adı verilmektedir. Çevre birimin işlemciye ihtiyaç duyup duymadığı sürekli kontrol edilerek tespit edileceği için işlemcinin arada başka bir iş ile ilgilenmeye fırsatı olmayacaktır. Beklenen birimin işlemciye ne zaman ihtiyaç duyacağı da bilinmediğine göre bu yöntemin başarımı son derece düşük, servis verilmesi muhtemel birim sayısı ise kabul edilemeyecek kadar azdır. Gerçek hayatta banka vb. ortamlarda oluşan kuyruklar buna en güzel örnektir. Bankadaki memur sadece kendi masasında ne zaman geleceği belli olmayan müşterileri bekleyecek olursa başka işler ile uğraşması mümkün olmayacaktır.

2. **Çevre birimi beklemek yerine, başka işler ile ilgilenmek ve zaman zaman çevre birimlerin servis isteyip istemediğinin kontrol etmek (polling-yoklama):** İşlemci başka işlerini yerine getirirken, kendi belirleyeceği (düzenli veya düzensiz) aralıklarla çevre birimlerin kendinden servis isteyip istemediklerini kontrol eder. İşlemcinin kontrol ettiği anda çevre birim servis istiyor ise işlemci servis verecektir. Ancak işlemci bir başka işle uğraşır iken veya herhangi bir çevre birime servis vermekteyken bir diğer çevre birim servis isteyecek olursa işlemci bunu o anda kontrol etmediği için istek olduğunun farkında olamayacak ve doğal olarak isteğe cevap veremeyecektir. Bu yöntem birinciden daha iyi sonuç verecek, hızlı ve yavaş birim arasındaki çalışmanın daha etkin bir şekilde yürütülmesini sağlayacaktır. Ancak çevre birimlerden bazılarının servis isteklerinin zaman zaman kaçırılmasına da neden olabilecektir. Gerçek hayatta bir otomobil sürücüsü göstergeleri, önünde akan trafiği, dikiz aynaları yardımıyla arkasındaki trafiği, motorun sesini kontrol ederek ve diğer yolcular ile sohbet ederek aracını kullanır. Bu işlemler sırasında bazen kendisini geçmek isteyen başka bir aracı zamanında fark edemediği için yol vermez. Bu örnekte olduğu gibi işlemci de farklı pek çok işle uğraşırken bazılarını gözden kaçırabilir.

3. **Servis isteyenin, servis alacağı kaynağı ihtiyaç duyduğunda uyarması (interrupt-kesme):** İhtiyaç duyan birimin bir sinyal mekanizması ile işlemciyi uyarması esasına dayanır. Bu durumda işlemci o anda yapmakta olduğu işi geri döndüğünde kaldığı yerden devam edebilecek biçimde bir süre (servis tamamlanıncaya kadar) askıya alır ve servis isteyen çevre birim ile ilgilenir. İşlemci kendinden istenen servisin gereğini yerine getirdikten sonra daha önceden yapmakta

olduğu işleme geri dönerek kaldığı noktadan devam etmelidir. Gerek servis alan, gerekse servis veren açısından değerlendirildiğinde en etkin kullanım kesme mekanizması adı verilen bu yöntemle sağlanır. Günlük hayatımızda buna en iyi örneklerden birisi zil sesidir. Her çevre birim (telefon, sokak kapısı, yangın alarmı, çalar saat vb.) farklı bir tonda çalarak bizden servis istendiğini belirtecektir. Telefon çaldığı zaman yapmakta olduğumuz işi bir süreliğine bırakır ve telefona bakarız. Telefon konuşmamızın bitimini takiben de daha önce yapmakta olduğumuz işe kaldığımız noktadan devam etmek üzere geri döneriz. Ancak geri döndüğümüzde işimize kaldığımız noktadan devam edebilmek için bir takım işlemleri de servis vermeye başlamadan önce yapmamız gereklidir. Örneğin telefon çalmadan önce kitap okuyor isek okuduğumuz sayfanın yerini kolay bulmak adına o sayfayı belirleyecek bir ayıraçtan yararlanabiliriz. Telefon konuşmasını tamamladıktan sonrada daha önce belirlediğimiz sayfadan itibaren kitabımızı okumaya devam edebiliriz. Ancak tüm uyaranların (zillerin) aynı anda oluşması durumunu da göz önünde tutarak kendimize has bir öncelik dâhilinde hareket tarzı belirleriz.

Bu stratejilerin birbirlerine göre avantaj ve dezavantajlarını basit ama daha teknik bir örnek üzerinden inceleyelim. İki bilgisayarın telefon hattı üzerinden haberleştiği bir uygulamada bilgisayarlardan alıcı durumunda olan, seri iskele üzerinden muntazam olmayan (eş zamansız) aralıklarla gelen verileri alır ve bunları bir bellek alanına yerleştirir. Bu bilgisayarın gelecek veriyi beklemek ve geleni bir bellek alanına kopyalamaktan başka bir işi olmadığı düşünülürse, bir çevrim içinde seri iskele üzerinden verinin okunduğunu, bir bellek gözüne yazıldığını ve yeni veri gelinceye kadar beklediğini (busy wait) söyleyebiliriz. Bilgisayarın yapması gereken başka bir iş olmadığı durumlarda bu tip bir kullanım çok rahatsız edici olmayabilir. Ancak kullanıcı arka planda bir başka işlemi de yapmak mesela bir e-posta yazılımı ile bir posta iletisi hazırlamak isterse ne olacaktır? Durumu daha rahat inceleyebilmek için e-posta yazılımı kullanımı sırasında yapılması gereken işlemleri de sırası ile inceleyelim;

1. Seri iskele üzerinden gelen verilerin alınarak bir bellek alanına depolanması
2. Klavye kullanılarak basılan her harfin e-posta yazılım programına aktarılması ve ileti içerisinde görüntülenmesi
3. İlk adımdan itibaren işlemlerin tekrarlanması

Aslında yapılan işler böyle maddeler halinde gösterildiğinde çok karışık bir durum olmadığı izlenimi yaratmaktadır. Fakat burada zamanlama ile ilgili olarak göz önüne almadığımız bazı noktalar vardır. Öncelikle haberleşme hızımızın 33600 bps olduğunu düşünelim. Bu basit bir hesap ile saniyede 3360 byte[154] verinin seri iskele

[154] Eş zamansız seri iletimde her btye'ın bir start ve bir stop biti ile yollandığını, bu durumda her byte'ın 10 bit olarak gönderildiği kabulü ile hesaplama yapılmıştır.

üzerinden bilgisayarımıza akacağı manasına gelir. Diğer deyiş ile yaklaşık 0,29 milisaniyede bir byte seri iskeleye erişmektedir. Eğer her 0,29 milisaniye sonunda seri iskeleden bir byte veri okunamaz ise kayıp olacaktır[155]. O zaman bu çevrimde yapılacak işlemlerin tamamının 0,29 milisaniye içinde bitirilmesi gereklidir. Buna göre yukarıda verdiğimiz iş sırasını biraz daha detaylandırarak 0,29 milisaniyelik zaman içinde hangi adımların tamamlanması gerektiğini bir kez daha inceleyelim;

1. Seri iskele üzerinden gelen verilerin alınarak bir bellek alanına depolanması
   a. Seri iskeleden gelen veriyi oku.
   b. Veriyi bellekte koyacağın yeri belirle.
   c. Veriyi belirlenen yere yerleştir.
   d. Kontrolü arka plandaki e-posta uygulamasına aktar.
2. Klavye kullanılarak basılan her harfin e-posta yazılım programına aktarılması ve ileti içerisinde görüntülenmesi
   a. Klavyeden basılan karakterin ne olduğunu algıla.
   b. Karakterin e-posta programı içerisinde görünmesini sağla.
   c. Eğer satır sonuna gelmiş ise ekrandaki verileri bir satır kaydırarak alta yeni bir satır ekle.
   d. Eğer sayfa sonuna gelinmiş ise ekrandaki verileri uygun şekilde kaydırarak sayfa düzenlemelerini yap.
   e. Eğer basılan karakter görüntülenebilir karakterler arasında değil ise ne yapılacağını belirle ve bu durum için kullanıcıyı uyar.
3. İlk adımdan itibaren işlemlerin tekrarlanması

Bu işlemlerin 0,29 milisaniye içinde ve hatasız olarak gerçekleştirilmesi gereklidir. Tabii bunlar bilgisayarın sadece basit iki işi yapması durumunda bize görünen yüzüdür. Bunlara ek olarak sistem saniyede 18,2 defa zamanlama ile ilgili fonksiyonları yerine getirebilmek üzere[156] bir başka kesme servisi tarafından kullanılmaktadır. Bu servis, sistem zamanının ve bir takım kontrol fonksiyonlarının (özellikle disk motor kontrol fonksiyonlarının) yerine getirilmesini sağlayan zamanlayıcı kesmesidir.

Günümüz bilgisayar sistemleri (yeni nesil işlemciler ve işletim sistemlerini kullanan) çok işlemcili (multi processor), çoklu işlem (multi processing), çok görevlilik

[155] 0,29 milisaniye sonra gelen yeni byte bir öncekinin kayıp olmasına neden olur.

[156] Bilgisayarın çalışması sırasında son derece önemli olan zamanlayıcı kesmesi (timer interrrupt) saniyede 18,2 defa -her 55 milisaniyede bir- kullanıcıya hissettirmeden devreye girerek sistem saati ve sistem işleyişi için vazgeçilmez öneme sahip fonksiyonları yerine getirmektedir.

(multi tasking) ve çok izleklilik (multi threading) özelliklerine sahip olsalar bile bazen yaptıkları işler nedeni ile cevap verme sürelerinin uzadığı hatta sistemin bazen cevap vermez hale geldiği hepimiz tarafından gözlemlenmiş bir durumdur. Bu durumdan kurtulmak adına, kullanıcıların klavyedeki tuşlara rasgele basmalarının veya fareyi oynatıp tıklamalarının bir faydası olmadığı gibi bu çevre birimlerin gereksiz olarak yarattıkları servis istekleri, işlemlerin daha da uzamasına ve problemin düzelmek yerine daha da kötüye gitmesine neden olacaktır.

14.1 Kesme Çeşitleri

Temel olarak kesmeler

1. Donanım kesmesi (Hardware interrupt)
2. Yazılım kesmesi (Software interrupt)

olarak iki gruba ayrılabilirler. Donanım kesmeleri çevre birimleri ile işlemci arasındaki uyumu sağlamak üzere oluşan kesme türleridir ve işlemci üzerinde bulunan özel bacaklara yollanan sinyaller olarak karşımıza çıkarlar. Diğer yandan yazılım kesmeleri ise kullanıcının **INT** komutu ile yarattığı, işlemci üzerindeki herhangi bir kontrol bacağına sinyal yollamayan ancak donanım kesmesi yaratılmış gibi çalışan kesmelerdir. Kesmeler ister donanım isterse kullanıcı (yazılım) tarafından tetiklensin, o kesme için hazırlanmış olan kesme servis (interrupt handler) programı çalıştırılır.

Kesmeler için bir diğer sınıflama şekli ise kesmenin oluşmasına neden olan durumdur. Bu kapsamda incelendiğinde kesmeler dâhili (internal) ve harici (external) kesmeler olarak değerlendirilebilir.

14.1.1 Dâhili Kesmeler

Dâhili kesmeler işlemcinin yaptığı işlemlerin sonucunda meydana gelir. Aynı zamanda mantıksal kesme (logical interrupt) olarak da bilinirler. Sıfıra bölme (divide by zero) ve adım adım çalıştırma (single step) buna en iyi örnekleri oluştururlar.

14.1.2 Harici Kesmeler

Bunlar donanım elemanları tarafından kullanılan ve işlemciyi uyarmak amacıyla oluşturulan kesmelerdir. Kullanıcının klavyenin tuşlarına basması veya fareye tıklanması ile oluşan kesmeler ilk etapta akla gelecek olanlardır. İşlemcinin donanım kesmelerini algılamak üzere iki tane kontrol bacağı mevcuttur. Bunlar INTR (Interrupt Request) ve NMI (Non-Maskable Interrupt) olarak adlandırılıp kullanımlarında belirgin farklılıklar vardır.

14.1.2.1 Maskelenemez Kesmeler (NMI)

İşlemci üzerindeki NMI bacağı, çevre birimlerinde oluşan kritik durumların işlemciye bildirilmesinde kullanılır. Bu bacağa gelen sinyaller (kesme istekleri) hemen değerlendirilir. Örneğin bellekte oluşan bir hata bu bacak üzerinden gönderilen işaret ile işlemciye iletilmekte ve ekrana "**memory parity error**" gibi bir hata mesajını çıkartmaktadır. Genel olarak bellekte böyle bir hata oluşması durumunda sistemin çalışmaya devam etmesi mümkün değildir. Adından da anlaşılacağı gibi bu gruba dâhil olan kesmelerin oluşması hiçbir şekilde engellenemez.

14.1.2.2 Maskelenebilir Kesmeler (INTR)

İşlemci üzerindeki INTR (Interrupt Request) bacağı, 8259A (Programmable Interrupt Controller) bütünleşik devresi üzerinden sürülmektedir. 8259A bütünleşik devresinin görevi, kullanıcının belirleyeceği durumlara bağlı olarak bazı çevre birimlerden gelen kesmelerin göz ardı edilebilmesini sağlamak, yani gelen istekleri belirlenen işlem politikası dâhilinde işlemciye yansıtmamaktır. IF (Interrupt Flag) değerine bakarak bu tipteki kesmeler dikkate alınır (IF←1-enable) veya alınmaz (IF←0-disable).

14.1.3 Kesme Öncelikleri

Kesmeler belirsiz zamanlarda, gerek çevre birimlerden gelen servis istekleri, gerekse kullanıcı tarafından oluşturulan istekler ile tetiklenip, çalışmakta olan işi bir süreliğine bölerler. Bu tür farklı kanallardan istek oluşması demek, zaman zaman kesmelerin aynı anda oluşabilir olması anlamına da gelir. Bu gibi durumlarda kesme üreten birimlerin de bir öncelik sırasına göre işlemciden servis alması gerekecektir. Bu öncelikler Tablo 14-1'de verilmiştir.

8259A programlanabilir kesme kontrol bütünleşik devresi sekiz adet dış birimin kesme isteklerini karşılamak üzere kullanılmaktadır. Bu 8259A bütünleşik devre, IF (Interrupt flag) bayrağındaki değere göre sürdüğü çevre birimlerden gelen kesmeleri engellerken, NMI bacağı üzerinden gelen kesmeler üzerinde hiçbir etkisi yoktur.

8259A bütünleşik devresi, 20H ve 21H numaralı iki iskele (port) kullanılarak programlanabilmektedir. 21H numaralı iskele, 8259A için maske bilgisini tutmak üzere kullanılırken, 20H adresindeki iskeleden de kontrol bilgisi yollanmaktadır. PC-XT'lerde bir adet 8259A bütünleşik devresi ile sadece 8 harici kesme kontrol edilebilirken (Tablo 14-2), PC-AT'lerde peş peşe bağlanmış (cascade) iki 8259A kullanılarak 15 adet harici kesme (Tablo 14-3) kontrol edilebilmektedir.

Tablo 14-1: Kesme öncelikleri

Öncelik	Kesme	Açıklama
1	Bölme taşması (Divide overflow)	INT 00H Divide by Zero – İşlemci tarafından bir sıfıra bölme işleminin sonuncunda oluşur. Genelde BIOS tarafından IRET komutu ile bu kesmeden dönüş sağlanırken DOS "Divide overflow" şeklinde bir mesaj verecek şekilde kesme servis programını değiştirir. Bu kesmenin oluşması ile verilen mesajı takiben hataya neden olan program sonlandırılır. Çalışmanın beklenmedik bir şekilde sonlanması nedeniyle sistem güvenilirliğini yitirmiş olacaktır. Bu durumunda sistemin yeniden başlatılması veya işlemin güvenli bir şekilde sonlandırılmasını sağlayacak özel bir kesme servis programı yazılması uygun olacaktır.
2	INT #	Yazılım olarak numarası verilen kesmenin çağırılması
3	INTO	INT 04H-Arithmetic Overflow kesmesinin anımsatıcı karşılığıdır. OF bayrağının durumuna bağlı olarak aritmetik işlemlerin (MUL/IMUL) sonuçlarını değerlendirmek üzere kullanılır. JO ve JNO koşullu dallanma komutları da bu amaçla kullanılır. Aritmetik taşma pek çok program için çok önemli bir hata olarak değerlendirilmediğinden dolayı bunun için yazılmış olan kesme servis programı IRET komutundan ibarettir.
4	NMI	INT 02H – Bellek üzerinde oluşan kritik hatayı belirlemek için kullanılır.
5	INTR	İşlemcinin INTR bacağı üzerinden bağlantılı bulunan 8259A kesme kontrol bütünleşik devresi üzerinden oluşturulan donanım kesmeleridir.
6	Single step	INT 01H Adım adım çalıştırmaya imkân sağlar. TF bayrağının 1 olması durumunda her komutun çalışmasını takiben bu kesme devreye girecektir.

Tablo 14-2: XT'ler için IRQ ve kontrol ettikleri dış bağlantılar

Giriş hattı	INT #	Vektör Adresi	Harici birim
IRQ0	08H	0020H	Zamanlayıcı kanalı
IRQ1	09H	0024H	Klavye
IRQ2	0AH	0028H	Renkli grafik adaptörü
IRQ3	0BH	002CH	Seri–2
IRQ4	0CH	0030H	RS–232-C (Seri–1)
IRQ5	0DH	0034H	Sabit disk
IRQ6	0EH	0038H	Disket sürücü
IRQ7	0FH	003CH	Yazıcı

Tablo 14-3: AT'ler için IRQ ve kontrol ettikleri dış bağlantılar

Giriş hattı	INT #	Vektör Adresi	Harici birim
IRQ0	08H	0020H	Zamanlayıcı kanalı
IRQ1	09H	0024H	Klavye
IRQ2	0AH	0028H	İkinci 8259A
IRQ3	0BH	002CH	Seri–2
IRQ4	0CH	0030H	RS–232-C (Seri–1)
IRQ5	0DH	0034H	Yazıcı–2
IRQ6	0EH	0038H	Disket sürücü
IRQ7	0FH	003CH	Yazıcı–1
IRQ8	70H	01C0H	Gerçek zaman saati-RTC
IRQ9	71H	01C4H	INT 0AH
IRQ10	72H	01C8H	Yedek
IRQ11	73H	01CCH	Yedek
IRQ12	74H	01D0H	Yedek
IRQ13	75H	01D4H	Matematik işlemci
IRQ14	76H	01D8H	Sabit disk
IRQ15	77H	01DCH	Yedek

8259A bütünleşik devresini programlamak için ilgili bit değerinin 0 (izinli – enabled) veya 1 (izinsiz – masked) olarak belirlenmesi gereklidir. Buna göre ilgili bit'lerden oluşan byte (PC-XT için) 8259A bütünleşik devresinin 21H adresli iskelesine yazılır. Bir örnek olması açısından klavye ve zamanlama kesmesinin izinli diğerlerinin ise maskelenmiş olduğu bir maske Şekil 14-1'de gösterilmiştir.

7	6	5	4	3	2	1	0		
1	1	1	1	1	1	0	0	→	FCH

Şekil 14-1: 8259A için maske değerinin belirlenmesi

Şekil 14-1'de oluşturulan maske değeri 0FCH olup 8259A bütünleşik devresinin 21H numaralı iskelesi üzerinden programlanmalıdır. Bu amaçla aşağıda yer alan kodun çalıştırılması gerekir.

```
MOV AL, 0FCH
OUT 21H, AL
```

Maskenin yerine yazıldığını ve işlemin bittiğini ve bundan sonra verilen maske değerinin geçerli olacağını belirtmek için ise kontrol iskelesi olarak adlandırılan 20H numaralı iskeleye işlemin sona erdiğini bildiren özel kodun (20H) yazılması gereklidir.

```
MOV AL, 20H
OUT 20H, AL
```

14.1.4 Kesme Oluştuğunda Yapılan İşlemler

Kesme ister kullanıcı tarafından yazılım olarak (INT komutu kullanılarak), ister sistemin genel işleyişi içinde dâhili veya harici etkenler tarafından oluşmuş olsun hemen ardından gerçekleşen adımlar şunlardır;

1. Bayraklar yığında saklanır (PUSHF)
2. TF←0
3. IF←0 (CLI)
4. CS yazmacı yığına atılır (PUSH CS)
5. IP yazmacı yığına atılır (PUSH IP)
6. INT komutunun yanındaki numara 4 ile çarpılarak[157] vektör tablosundan bu kesme servis programının bulunduğu bellek adresi bilgisine erişilir. Bu alandan alınan ilk word değer IP yazmacına takip eden adresten alınan ikinci word değer CS yazmacına yerleştirilerek kontrol CS:IP ikilisinin belirlediği noktaya aktarılır.

Dikkat edildiği gibi kesme servis programı çağırılmadan önce bayrak değerlerinin yığın üzerinde saklanması gereklidir. Zira kesme servis programı tarafından gerçekleştirilecek işlemler bayrakların değerlerinin değişmesine neden olabilecektir. Program kendi işlemini gerçekleştirmek için kullanacağı her yazmacı işi bittiğinde bulduğu değer ile bırakmalıdır. Bu amaçla değerini değiştirmeyi düşündüğü yazmaçların eski değerlerinin de kesme servis programını yazan tarafından mutlaka yığında saklaması, kesme servis programının IRET komutu ile sonlandırılmasından hemen önce de yığından saklandıkları sıranın tersinde geri alınması gerekmektedir.

Kesme servis programı devreye girdiğinde IF bayrağının değeri 0 olacağı için kesme servis programının içinde kesme oluşması engellenmiş olacaktır. Ancak sistemin genel işleyişini etkilememek adına bu durumun düzeltilmesi gerekir ve kesme servis programında STI komutu ile kesmelere izin verilir.

---

[157] Her kesme servis programının bellek üzerinde nerede olduğunu belirleyecek olan kesim ve göreli adres değerleri vektör tablosunda ikişer byte'tan 4 byte olarak saklanmaktadır.

14.2 Vektör Tablosunun Görevi ve Komumu

Kesme servis programları gerek BIOS gerekse işletim sistemi (DOS) tarafından sağlanan, değişik uzunluktaki kod parçaları olup bilgisayar sisteminin açılışı sırasında belleğin farklı bölgelerine yerleştirilirler. Bellek üzerinde farklı alanlarda bulunan bu kod parçalarına erişebilmek üzere vektör tablosu adı verilen merkezi bir içindekiler tablosu tutulmaktadır. Tablonun görevi sadece ve sadece kesme servis programlarının bulunduğu başlangıç adresini belirlemek olup vazgeçilmez öneme sahiptir[158]. Bu tablo ile ilgili özellikler şunlardır;

1. 00000H-003FFH fiziksel adresleri arasında bulunan 1024 byte'lık alanda bulunur.
2. Her servisin adresi tablo üzerinde 2 word ile (4 byte) ifade edilir. Bunlardan düşük adresteki 2 byte'ta göreli konum, takip eden 2 byte'ta ise kesim adresi yer alır.
3. Vektör tablosunda her kesme için 4 byte adres bilgisi kullanıldığı bilindiğine göre 1024 byte'lık uzunluğundaki tabloda 256 tane kesme için adres bulunabilir. Buna göre herhangi bir kesme servis programının belleğin neresinde olduğunu bulmak üzere ilgili kesme numarası 4 ile çarpılıp, elde edilen indisten alınan ilk word değer göreli konum adresi, takip eden (+2 byte ilerideki adresten alınan) word değer ise kesim adresi olarak kullanılır[159].

Zaman zaman kullanıcıların da var olanların dışında, kendilerine has kesme servis programları yazmaları gerekebilir. Bunun için vektör tablosunda INT 60H - INT 67H arasında kalan kısmı (00180H fiziksel adresinden itibaren) ayrılmıştır. Kullanıcılar, belleğe yerleştirdikleri (TSR-Terminate and Stay Resident) programların adreslerini vektör tablosundaki uygun adreslere koymak şartıyla INT komutu kullanarak çağırılabilen kodlar yazabilirler. Benzer şekilde mevcut bir kesme servis programının yerini tamamen alacak veya var olandan önce çalışacak şekilde bir kod parçası da yazmak gerekebilir. Bu işlemler ister istemez belek üzerinde yer alan vektör tablosundaki değerlerin değiştirilmesini gerektirecektir. Ancak dikkatsiz ve kurallara uymadan yapılan değişiklikler sistemin kararlı çalışmasını engelleyebilir.

14.2.1 Vektör Tablosunda Değişiklik Neden ve Nasıl Yapılır?

Kullanıcı kendi yazdığı bir kodun, mevcut bir kesme servis programının yerini almasını isteyebileceği gibi var olan kesme servis programından önce devreye gire-

[158] Vektör tablosu, kitaplarda gördüğümüz içindekiler tablosuna benzetilebilir. İçindekiler tablosu nasıl kitap içinde yer alan konuların kitabın hangi sayfalarında yer aldığını gösteriyor ise vektör tablosunda kesme servis programının belleğin hangi adresinde yer aldığını göstermektedir. Okuyucuların kitabın içindekiler tablosunda aradıkları konuyu bulduktan sonra ilgili sayfada konunun detaylarını okudukları gibi bilgisayarlarda kesme adreslerini vektör tablosundan aldıktan sonra ilgili adresten kesme servis programını çalıştırırlar.

[159] Örneğin INT 21H için yazılmış olan kesme servis programının bellek üzerindeki yerine bulmak üzere 84H (21H*4) adresinden 1 word IP'ye, 86H (84H+02H) adresinden 1 word CS yazmaçlarına yerleştirilmelidir.

rek çalışmasını, hemen ardından da orijinal kesme servis programının devreye girmesini isteyebilir. Her iki durumda da kullanıcı tarafından yazılmış olan kodun adresinin vektör tablosundaki ilgili alanlara yerleştirilmiş olması gerekmektedir. Bunun için yapılması gereken işlemler sırasıyla;

1. Kesme servis programının yerini almak üzere kullanıcı tarafından yazılmış kodun işletim sistemince başka bir program parçası tarafından ezilmeyecek şekilde belleğe yerleştirilmiş olması gereklidir (TSR program).
2. Kullanıcı tarafından yazılan kodun devreye girebilmesi için onun adresinin (kesim ve göreli konum değerlerinin) vektör tablosundaki ilgili alanlara yazılması gereklidir. Bunu sağlamak üzere vektör tablosu üzerinde bir değiştirme işlemi yapılacaktır. Değiştirme sırasında adresi değiştirilmek istenen kesme servis programının çağırılmasını engellemek üzere kesmeler değişiklik tamamlanıncaya kadar engellenmelidir (disable interrupts)[160].
3. Her ne neden ile olursa olsun, adresi değiştirilecek olan kesme servis programının orijinal adresi saklanmalıdır (Orijinal kesme vektör değeri vektör tablosunda o vektör için ayrılmış bellek alanından okunabileceği gibi INT 21H Fonksiyon 35H - Get Interrupt Vector kullanılarak da okunabilir).
4. Kullanıcı tarafından yazılan ve belleğe yerleştirilmiş olan kesme kodunun adresi vektör tablosunda ilgili adreslere yerleştirilir (Bu işlem vektör tablosuna doğrudan erişilerek yapılabileceği gibi INT 21H - Fonksiyon 25H- Set Interrupt Vector kullanılarak da yapılabilir).
5. Vektör tablosu üzerindeki değişiklikler tamamlanır tamamlanmaz sistemin işleyişini engellememek üzere kesmelere izin verilmelidir (enable interrupts).
    a. Eğer kullanıcının yazdığı kod orijinal kesme servis programının yerini alacak ise, kesme servis programının çalışma prensiplerine uygun yazılmış[161] ve işi bittiğinde geri dönebilir (IRET) nitelikte olmalıdır.

---

[160] Vektör tablosundaki bir kesmenin adresinin değitirilebilmesi kesim ve göreli konum değerlerinden oluşan 2 word'ün tabloda ilgili adrese yazılması gerekecektir. Ancak 8086 işlemcinin bir seferde en fazla 16 bit'lik bir işlemi yapabileceği düşünüldüğünde 2 word'ü aynı anda değiştirmek mümkün değildir. Ayrıca ilgili kesmeyi değiştirmek üzere ilk word vektör tablosuna yazıldığı anda adresi değiştirilmek istenen kesme bir neden ile tetiklenecek olursa, adresin yarısı eski, diğer yarısı yeni adres değerleri olacağı için kullanıcı kontrolü kayıp edilecektir. Bu problemin tek çözümü vektör tablosu üzerinde değişiklik yaparken bir şekilde kesme oluşmasının engellenmesidir. Ancak kesmelerin engellenmesi NMI türündeki çok kritik durumları bildiren kesmelerin oluşmasına engel olamaz.

[161] Gerek yordam çağırmada gerek ise kesme kullanımında son derece önemli bir temel prensip vardır. *Bulduğun gibi bırak*" Bu prensip uyarınca yapılan işlem daha önce yapılmış ve/veya yapılmakta olan işlemlerin durumunu bozacak nitelikte olmamalıdır. İşte bu nedenledir ki INT komutu çalıştırıldığı anda bayrak değerleri otomatik olarak yığında saklanmaktadır. Benzer biçimde kesme servis programlarında da işleyişin gereği olarak değeri değişecek yazmaçların işleme başlamadan önceki değerlerinin saklanması ve iş bitimini takiben eski hallerine getirilerek bırakılması sorumluluğu programcıya bırakılmıştır. Yazılan kesme servis yordamı yapması gereken işlemleri en etkin şekilde ve en kısa sürede tamamlamalıdır.

b. Eğer kullanıcının yazdığı kod orijinal kesme servis programından önce devreye girecek şekilde tasarlanmış ise, kendi işlerini en kısa zamanda tamamlayacak, orijinal kesme vektörünün bulunduğu adrese[162] kesimler arası atlama (intersegment jump) yaparak kontrolü orijinal kesme servis programına bırakacaktır.

6. Kullanımın sona ermesini takiben kullanıcı tarafından yazılan kod parçası bellekten çıkartılacak ve orijinal kesme kodunun daha önce saklanan vektör adresi tablodaki uygun yerine geri konacaktır.

14.2.2 Interrupt Servis Programı Yazarken Nelere Dikkat Edilmelidir?

Genel olarak vektör tablosu üzerinde değişiklik yapılacağı zaman nelere dikkat edilmesi gerektiğini gördükten sonra kesmenin oluşması ile devreye girecek olan kodun yazımında nelere dikkat edilmesi gerektiğini de vurgulamaya çalışalım;

1. Kesmenin oluşması sırasında yapılan işlemlerden biriside IF değerinin 0 yapılmasıdır (bkz. 14.1.4. bölüm). Kesmelerin kullanılabilir hale gelmesi için kesme kodu içinde IF değerinin 1 yapılarak kesme oluşmasına izin verilmelidir (enable interrupt).

2. Kesme kodu içinde hedeflenen amaç uyarınca yapılacak işlemlerin bir sonucu olarak bazı yazmaçlardan yararlanılacaktır. Bu yazmaçların daha önceki değerlerinin korunması kesme servis programını yazanın sorumluluğundadır. Kesme servis yordamından geri dönüldüğü zaman işlemin kaldığı noktadan devam edebilmesi buna bağlıdır. INT komutu çalıştırıldığında bayrak değerleri otomatik olarak saklanmaktadır. Bunun için kullanıcın bayraklar ile ilgili olarak ek bir işlem yapması gerekmez.

3. Kesme programının kullanacağı farklı kesimler var ise ilgili kesim yazmaçlarına bu değerler yüklenmelidir.

4. Yazılan yeni kesme kodu bir donanım (BIOS) kesmesinin yerini alacak ise kod içinde hiçbir şekilde işletim sistemine (DOS) ait bir fonksiyon kullanılmamalıdır.

5. Kesme programı 8259A programlanabilir kesme denetleyicisinin durumuna dikkat etmelidir.

6. Kesme programının adresi vektör tablosuna yerleştirilmeden önce bellek üzerinde başka bir kod parçası tarafından ezilmeyecek şekilde yerleşmiş olmalıdır (TSR). Ancak kesme kodunun yazılım amacına göre;

---

[162] Kesme adresinin vektör tablosundaki eski değerinin saklanması bu gibi durumlar için gerekli olduğu gibi kullanıcının orijinalin yerini alacak olan kesme servis yordamını sonladırması durumunda da sistemin eski haline getirilebilmesi için gereklidir.

a. Eğer yazılan kod mevcut orijinal kesme kodunun yerini alacak ise **IRET** komutu ile geri dönülmelidir (INT komutu kullanıldığında bayrak ve geri dönüş adresleri CS:IP çifti olarak yığında saklandığı için IRET komutu bu değerleri geri alarak dönüşü problemsiz bir şekilde tamamlayacaktır).

b. Eğer yazılan kod orijinal kesme kodundan önce çalışıp daha sonra kontrolü ona verecek ise vektör tablosu değerleri üzerinde işlem yaparken sakladığımız orijinal kesme kodunun bulunduğu bellek alanına JMP (kesimler arası atlama) komutu ile atlamamız gereklidir.

7. Genel olarak kesme kodu, yığında sakladığı geçici değerlerin yığından kaldırılmasından da sorumludur. Yığın üzerinde fazladan kalan her türlü veri genel işleyişi sekteye uğratacaktır.

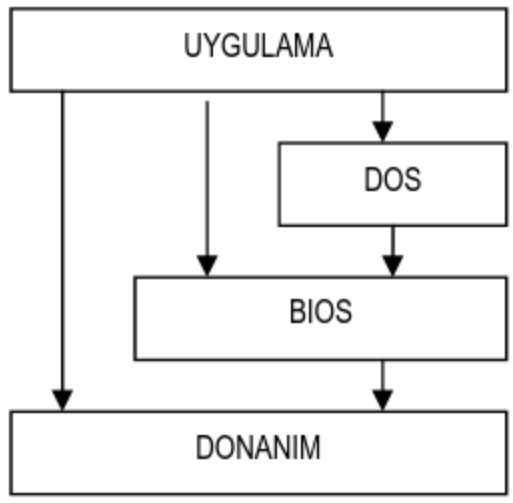
14.2.3 Vektöre Tablosunun Eski Haline Getirilmesi

İhtiyaca bağlı olarak kesmeye ait orijinal vektör değeri kullanıcı tarafından değiştirilebilmektedir. Bu işlem her ne amaçla yapılmış olursa olsun kullanıcının işlemini bitirilmesi ve yazdığı programı bellekten kaldırılması haline mutlaka ilgili kesme vektörünün de eski haline getirilmesi gerekir. Aksi halde bellek üzerinde artık yer almayan bir kod alanını gösteren değer vektör tablosunda kalacak ve söz konusu kesme her oluştuğunda sistemin çalışması sekteye uğrayacaktır. Bu probleme neden olmamak için "*bulduğun gibi bırak*" prensibine bağlı kalınmalı, orijinal kesmenin adresi vektör tablosundaki yerine yazılmalıdır.

14.3 DOS ve BIOS Ara Yüzleri

Kullanıcının yazdığı uygulamanın üzerinde çalıştığı donanım ile irtibatın sağlanması açısından BIOS ve DOS arayüzleri son derece önemlidir. Programlar ihtiyaç duydukları noktalarda (ekrana veri yazmak, klavyeden veri almak gibi) gerekli BIOS kesmelerini (yordamlarını) çağırmaktadır. Çağırılan bu kesmelerin gereği olarak parametre aktarımları da yapılmaktadır. Bu tür ara yüzler programcıları pek çok zahmetten kurtardığı gibi doğrudan donanıma erişilmesi nedeniyle uygulamanın daha hızlı çalışmasını da sağlarlar. Ayrıca uygulama programlarının denenmiş, doğru çalıştığından şüphe edilmeyen standart yordamları kullanmaları sayesinde uyumluluk sağlanmış olur. DOS ve BIOS kesme fonksiyonları, programları, üzerinde çalışmakta oldukları donanımdan izole ederler. Grafik kartları incelendiğinde karşımıza çok farklı özelliklerde grafik kipi çıkmaktadır. Bir programın bütün bu grafik kartları ile uyumlu olabilmesi için programcının programını bunların özelliklerini göz önünde bulundurarak kodlaması gerekmektedir. Oysa kullanılacak BIOS kesme fonksiyonları bu kartlar ile uyumlu arayüzler sağladığı için ekran kartının kullanımı sadece bu fonksiyonların çağırılması ile gerçekleştirilebilecektir.

Uygulama açısından bakıldığında donanımın kullanımı üç katmanlı bir model oluşturmaktadır. Şekil 14-2'de görüldüğü gibi uygulama yazılımları donanıma, duruma göre DOS, BIOS veya doğrudan erişebilmektedir.



Şekil 14-2: Katman Model

Kesmeler ister yazılım ister donanım olsunlar 8 bit ile ifade edilen bir numara ile anılırlar. Bu durumda kesme numaraları 0–255 (00H-FFH) arasında değişebilir. Bunların bir kısmı BIOS, diğer bir kısmı ise DOS kesmelerini ifade etmek için kullanılmaktadır. Teorik olarak 256 tane BIOS ve DOS kesmesi olabilir. Ancak kesmeler özelliklerine göre sınıflandırıldıkları için 256 sayısı, farklı grup adedi olarak algılanmalıdır. Zira bir grup altında farklı fonksiyonlar (hatta alt fonksiyonlar) tanımlanarak gerçekleştirilebilecek kesme fonksiyon sayısı arttırılmaktadır. Örneğin ekran çözünürlük işlemleri ile ilgili kesmenin numarası 10H'dir. Oysa bu kesme altında 00H-1CH arasında 29 tane kadar farklı fonksiyon bulunur. Kesmeler çağırılırken fonksiyon numaraları AH yazmacı, eğer var ise alt-fonksiyon numaraları da AL yazmacı üzerinden aktarılır. Kesme kodunun çalıştırılması için başka parametrelerin de aktarılmasının gerekli olduğu durumlarda diğer yazmaçlardan yararlanılmaktadır. Kesmeler yaptıkları işleme göre geriye bazı değerler de döndürebilirler. Kesme yordamlarından yararlanmak isteyen programcıların yazmaçları kullanırken dönüş değerlerinin hangi yazmaçlar üzerinde olacağını hesaba katarak işlemlerini yapmaları gereklidir.

14.3.1 BIOS (Basic Input Output System) Kesme Fonksiyonları

Numaraları 00H-1FH arasında kalan kesmeler BIOS grubuna dâhil kabul edilmektedir. BIOS[163] kesme fonksiyonları bilgisayarın donanımının etkin kullanımını sağlar. BIOS kesme fonksiyonları, DOS tarafından kendi işlemlerini yapmak üzere de çağırılıp kullanılmaktadır. Programcılar yüksek seviyeli programlama dillerinin

[163] PC ve PC uyumlu bilgisayar sistemlerinde BIOS, ROM da bulunur. Bundan dolayı bazı kaynaklarda ROM BIOS ifadesi de kullanılmaktadır.

sağladığı imkânlar ile problemi çözmenin başka bir yolunu bulamadıkları zaman kesme fonksiyonlarına başvururlar. Çok nadiren de olsa bazen BIOS kesme fonksiyonları da yeterli olmayabilir ve programcı donanıma kadar inerek kendi ihtiyacını karşılayacak kesme yordamlarını yazabilir.

Bilgisayar sistemlerinde yer alan BIOS, sistemi üreten firmalar tarafından sağlanmakla birlikte kodları genelde bu işte uzmanlaşmış olan firmalar (AMI, Award, Pheonix vb.) tarafından yazılır. Bilgisayar sistemlerine eklenen her yeni donanım ve mevcut olanlarda karşılaşılan hataların düzeltilmesi ile zaman içinde BIOS kodları yenilenmektedir. BIOS fonksiyonları genel olarak video kartı, seri iskele, bellek, paralel iskele, disket, klavye, sabit disk ve pil ile çalışan saat (Real Time Clock) birimlerine erişimi sağlamaktadır. BIOS kesme fonksiyonları[164] hakkında genel bilgi için 16.4. bölümde yer alan BIOS Kesme Listesi'nden yararlanılabilir.

14.3.2 DOS Kesme Fonksiyonları

Numaraları 20-2FH arasında kalan kesmeler DOS grubuna dâhildir. Bunlara, 33H fare (mouse) ve 67H genişletilmiş bellek yönetimi (expanded memory management) için kullanılan fonksiyonlar eklenebilir. BIOS fonksiyonlarının yanı sıra DOS fonksiyonları da donanıma erişimi sağlamak üzere tasarlanmıştır. Ancak DOS fonksiyonları BIOS fonksiyonları gibi fiziksel değil mantıksal arayüzü sağlamaktadır. Örneğin BIOS sabit diske erişmek istediği zaman iz (track), kesim (sector), kafa (head) gibi fizikler özellikleri kullanarak erişirken, DOS bunlara dosya (file), dizin (path), sürücü (drive) gibi mantıksal seviyede erişmektedir. DOS kesmeleri donanıma BIOS üzerinden eriştiği gibi gerekli durumlarda doğrudan da erişebilmektedir.

DOS'un sağladığı kesmeler tek kullanıcılı (single user) ve tek iş (single task) düzenine göre çalışacak şekilde düzenlenmiştir. Bu yüzden DOS kesmeleri birbirlerinin içerisinden çağırılamazlar[165]. DOS bellekte **INDOS** adında, bir byte büyüklüğünde bir değer tutar. Bu değerin durumuna göre DOS fonksiyonlarına güven içinde erişilip erişilemeyeceği belirlenmektedir. INDOS bayrağı DOS kesme fonksiyonlarına erişimde semafor (semaphore) vazifesi görerek DOS kesmesi içinden bir başka DOS kesmesi içine girme (re-entrency) problemini engellemektedir. DOS kesme fonksiyonları hakkında genel bilgi için 16.5. bölümde yer alan DOS Kesme Listesi'nden yararlanılabilir.

---

[164] http://www.ctyme.com/rbrown.htm adresinden kesmelerin farklı şekillerde gruplandırılmış açıklamalarına ve http://www.bioscentral.com/ adresinden de BIOS ile ilgili pek çok farklı linke erişmek mümkündür.

[165] DOS kesmelerinin birbirinin içerisinden çağırılamaması re-entrency olarak ifade edilir. Eğer bu kural bozulacak olursa, DOS tarafından kullanılan dâhili değişkenler (internal variables) ve yığın, birbiri içinden çağırılan DOS kesme fonksiyonları yüzünden bozulacak ve iş akışı bozulduğundan dolayı sistemin çalışması etkilenecektir.

15. YÜKSEK SEVİYELİ DİLLER VE ASSEMBLY BAĞLANTISI

Zaman zaman yüksek seviyeli dillerden assembly program parçaları çağırılarak bir takım işlemlerin daha alt seviyede ve daha hızlı olarak yapılması gerekebilir. Benzer şekilde Pascal ve C gibi yüksek seviyeli programlama dillerinde yazılmış kodların arasında assembly program parçacıklarına da rastlanabilmektedir. Yüksek seviyeli diller içinde assembly kullanılması gerektiği zamanlarda iki farklı yöntemden yararlanmak mümkündür. Bunlardan ilki ilgili yüksek seviyeli dilin derleyicisinin imkân verdiği ölçüler içerisinde assembly kodların yüksek seviyeli dilin kodunun içine yerleştirilmesi (in-line), diğeri ise bağımsız olarak yazılıp, derlendikten sonra elde edilen assembly nesne kodlarının (object code) yüksek seviyeli bir dil ile kodlanan program içinden çağırılmasıdır. İkinci tür kullanım ilkine göre biraz daha karmaşık gibi görünmekle birlikte daha etkin bir yapı sunar. Ancak bu amaçla üretilmiş olan assembly kodlarda uyulması gereken bazı tanımlama ve parametre aktarma kurallarına dikkat edilmelidir.

Bu bölümde Turbo Pascal ve C programlama dili ile assembly dilli arasındaki kullanımın nasıl yapılacağını göstermek üzere basit örnekler verilecektir. Kullanılan yüksek seviyeli dil ve derleyicisine bağlı olarak bazı tanım ve kullanım farklılıkları olabileceği unutulmamalıdır. Bu nedenle assembly dili ile bağlantılı kullanılacak programlama diline ait derleyicinin kullanıcı kitaplarından ne tür tanımlar yapılması gerektiği kontrol edilerek program yazılmalıdır.

15.1 Turbo Pascal'dan Assembly Yordam Çağırma

Turbo Pascal da yazılmış programdan çağıracağımız bir assembly program ile imleci (cursor) ekran üzerinde bir yere getirerek geri dönen ve ekrana bir mesaj yazan örnek bir çalışma yapalım. Programın kontrolü Pascal' da yazılmış kodda olacağı için öncelikle Pascal programımızı yazalım;

Örnek 15-1

Tablo 15-1'de görüldüğü gibi Pascal programda farklı bir kesim içinde yer alacak olan assembly yordamı çağırabilmek için gerekli derleyici komutu **{$F+}** yardımıyla verilmektedir (satır 1). Benzer biçimde kullanılacak *Move_Cursor* isimli yordamın içinde bulunduğu OBJ uzantılı dosyanın adı da **{$L MOVECR}** komutu ile linker programına bildirilmektedir (satır 3). Pascal program içinden çağırılacak olan *Move_Cursor* isimli harici yordam ve buna yollanacak parametreler ise satır 5'de tanımlanmıştır. Pascal programdan *Move_Cursor* isimli harici yordam çağırıldığında (satır 12) ilgili parametreler yığın üzerinden aktarılacaktır. Parametreler veriliş sırasında göre (soldan sağa doğru) yığına yerleştirilirler. Böylece *Move_Cursor* isimli yordam, imleci 12. satırın 39. sütununa yerleştirip **Cursor...** yazısını yazacak (satır 13) sonra kullanıcının herhangi bir tuşa basması beklenecektir (satır 14).

Tablo 15-1: Pascal program içinde assembly yordam çağırma

```
 1  Program Callasm (input, output);
 2  {$F+}
 3  {$L MOVECR }
 4  Uses Crt;
 5  Procedure Move_Cursor (Row: Integer; Col: Integer);External;
 6  Var
 7          trow, tcol: Integer;
 8  Begin
 9          Clrscr;
10          trow:= 12;
11          trow:= 39;
12          Move_Cursor (trow, tcol);
13          Write ('Cursor...');
14          Repeat Until Keypressed;
15  End.
```

MOVECR.ASM isimli assembly program (Tablo 15-2) içinde yer alacak olan *Move_Cursor* isimli yordam ise BIOS kesmelerinden 10H (Video Interrupt)'in 02H (Set Cursor Position) fonksiyonunu kullanarak imleci bizim belirlediğimiz noktaya getirecektir. Gerek Pascal, gerekse assembly dilleri büyük/küçük harf duyarlı olmadıkları için harici yordam ismini ne şekilde yazdığımızın önemi yoktur. Ancak bir alışkanlık elde edebilmek ve büyük/küçük harf duyarlı diller ile assembly kullanırken hata yapma olasılığını azaltmak amacıyla yordam isimlerinin her iki kod parçasında da aynı şekilde yazılmış olmasına dikkat edilmelidir. Buna göre *MOVECR.ASM* isimli programımızı Tablo 15-2'de ki gibi kodlamamız gerekecektir.

Tablo 15-2: Pascal tarafından çağırılan assembly yordam

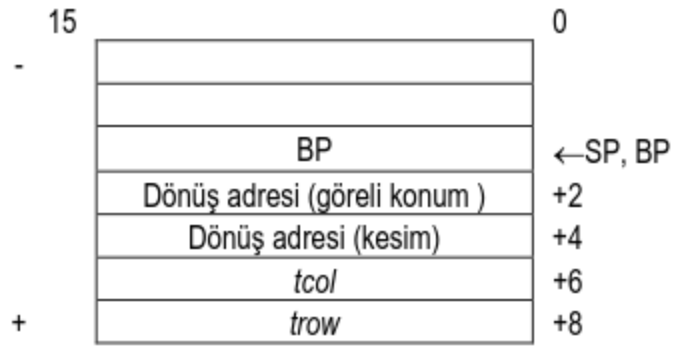
```
 1                  TITLE Combine Turbo Pascal and Assembly
 2                  Public Move_Cursor
 3  Code            SEGMENT BYTE PUBLIC
 4  Move_Cursor     PROC FAR
 5                  ASSUME CS: CODE
 6                  PUSH BP
 7                  MOV BS, SP
 8                  MOV AX, [BP+8]
 9                  MOV BX, [BP+6]
10                  MOV DH, AL
11                  MOV DL, BL
12                  XOR BX, BX
13                  MOV AH, 2
14                  INT 10H
15                  POP BP
16                  RET 4
17  Move_Cursor     ENDP
18  Code            ENDS
19                  END
```

Assembly programımız içinde yer alan *Move_Cursor* isimli yordamın Pascal'da kodlanmış program içerisinden çağırılmasını sağlamak üzere **PUBLIC** tanımından

yararlanılmaktadır (Tablo 15-2, satır2). Pascal programı içinden assembly programı kullanabilmek için kod kesim ismi ve tanımlama şekli son derece önemlidir. Örneğimizden de görüldüğü gibi (satır 3) kesim ismi **Code** olarak verilmiş, **PUBLIC** ve **BYTE** özellikleri ile tanımlanmıştır. Pascal programdan **FAR** yordam olarak çağırıldığı için *Move_Cursor* isimli yordam da **FAR** olarak tanımlıdır (satır 4).

Pascal programdan aktarılan parametreler diziliş sırasına göre yığına yerleştirilmekte ve ardından *Move_Cursor* isimli yordam çağırılmaktadır. Buna göre yığınında oluşan değerler Tablo 15-3'de gösterilmiştir.

Tablo 15-3: Pascal-Assembly kullanımında yığının durumu



Yığın üzerinden aktarılan parametrelere erişmek için BP yazmacından yararlanılacaktır. BP yazmacının eski değerinin kayıp olmaması için yığına saklanması (satır 6) ve yığına BP üzerinden erişebilmek amacıyla SP yazmacının o anki değerinin de BP yazmacına atanması gereklidir (satır 7). Bu aşamadan sonra yığındaki verilere istenilen sıra ile erişilebilmesi için [BP+6] ve [BP+8] adresleri kullanılacak, AX yazmacına *trow* değeri, BX yazmacına ise *tcol* değeri konulacaktır (satır 8–9). Kullanılacak olan INT 10H kesmesinin 02H fonksiyonu, DH yazmacında satır (*trow*), DL yazmacında ise sütun (*tcol*) değerlerini beklemektedir (satır 10–11). İşlemin hangi aktif sayfa üzerinde gerçekleştirileceği ise BX yazmacına yerleştirilen değer ile belirlenir (satır 12). Ardından işlemi gerçekleştirecek fonksiyon numarası AH yazmacına (satır 13) konularak INT 10H çağırılır (satır 14). Bu işlemin sonucunda *Move_Cursor* isimli yordam bizim *trow* ve *tcol* değişkenleri ile ekranda yerini belirlediğimiz noktaya imleci yerleştirip, kendisini çağıran Pascal programa geri dönerek (satır 16) *Write ('Cursor...')* komutu ile belirlenen mesajı ekrana yazacaktır.

Pascal program, parametre olarak yığına yerleştirdiği değerleri kaldırmaz. Bu sorumluluk assembly kod parçasına bırakılmıştır[166]. Bu nedenle aktarılan her parametre için 2 byte olmak üzere SP değeri assembly program tarafından değiştirilmelidir[167] (satır

[166] Pascal ve assembly kullanımında yığından parametreleri kaldırmak çağırılan yordamın görevidir.

[167] Pascal program tarafından yığına yerleştirilen 2 parametre vardır. Her parametre 2 byte yer kapladığı için dönüş işlemi sırasında RET 4 komutundan yararlanılarak SP yazmacının değeri otomatik olarak 4 arttırılmış olacaktır. Böylece SP yazmacının değeri, parametrelerin yığından çekilmesi sonucunda olması gereken değere ulaşacaktır. Her ne kadar aktarılan parametre değerleri fiziksel olarak POP benzeri bir işlem ile yerinden kaldırılmamış dahi olsa, SP yazmacının sahip olduğu değerden düşük adres değerine sahip olan yığın alanlarında bulunan değerler bizim için anlam ifade etmezler.

16). Ancak bu aşamada daha önce değeri yığın üzerinde saklanan BP yazmacının eski değerinin geri alınması (satır 15) gerekir. Aksi halde assembly koddan geri dönülmesi mümkün olamayacağı gibi programın kontrolü de bilemeyen bir adreste bulunan koda geçecek ve sonuç alınamayacaktır.

İki farklı program parçasının birlikte kullanılabilmesi için öncelikle assembly programın derlenerek *MOVECR.OBJ* dosyanın yaratılması gereklidir. Hemen ardından Turbo Pascal ortamından Pascal programın çalıştırılabilir halinin oluşturulması gerekir. Bu işlem sırasında Pascal derleyicisi verilen komut uyarınca *MOVECR.OBJ* isimli nesne kodunu da kullanarak EXE uzantılı bir dosya oluşturacaktır. Yaratılan EXE uzantılı dosyanın komut satırından çalıştırılması ile ekranın 12. satır ve 39. sütununda 'Cursor...' yazısı çıkacaktır.

15.2 Assembly'den Turbo Pascal Yordam Çağırma

Yüksek seviyeli dillerde hızlı işlem yapabilmek veya donanıma daha alt seviyelerden müdahale edebilmek amacıyla assembly yordamlar çağrıldığı gibi assembly programlardan da yüksek seviyeli dillerde yazılmış yordamları çağırmak mümkündür.

Örnek 15-2

Dizi elemanlarının belirlediğimiz aralıkta olup olmadıklarını test eden uygulamayı Pascal ve assembly dillerini birlikte kullanarak gerçekleştirelim. Verilerin Pascal programı tarafından okunarak *Buffer* isimli bir dizide saklandığını, Pascal'dan çağırılacak *CheckRange* isimli assembly yordam ile bu sayıların verilen aralık içinde olup olmadıklarının test edildiğini ve koşulu sağlamayan sayı tespit edildiğinde ise assembly kod içinden çağırılan *RangeError* isimli Pascal yordam ile durumun kullanıcıya bildirildiğini varsayalım.

Tablo 15-4: Assembly tarafından çağırılan Pascal program

CHECKNUMBER.PAS

```
 1  Program Sayi (input, output);
 2  {$L CHECK }
 3  Var
 4          Buffer: array [1..100] of integer;
 5          Count: integer;
 6  Procedure RangeError (N:integer);
 7  Begin
 8          Writeln ('Sayı tanımlı aralığın dışında...',N);
 9  End;
10  Procedure CheckRange (min, max: integer); external;
11  Begin
12          Count=0;
13          While (not Eof and (count < 100)) do
14          Begin
15                  Count:= Count +1
16                  Readln (Buffer [Count]);
17          End;
18          CheckRange (-10,10);
19  End.
```

Programımız *CHECK.OBJ* isimli assembly kod ile birlikte link edileceği için **{$L CHECK}** komutu Pascal derleyicisini bilgilendirmek üzere verilmiştir (Tablo 15-4 satır 2). Kullanıcının gireceği veriler ve bu verilerin sayısı *Buffer* ve *Count* isimli değişkenlerde tutulmaktadır (satır 4–5).

RangeError isimli Pascal yordam, dizi elemanının verilen aralıkta olmaması durumunda assembly program tarafından çağırılacak ve ekrana kullanıcıyı bilgilendiren bir mesaj yazılmasını sağlayacaktır (satır 6–9).

Pascal program içinden çağırılacak olan *CheckRange* isimli assembly yordam harici (external) olarak tanımlanmıştır (satır 10). Bu yordama ait kod *CHECK.OBJ* isimli dosyanın içinde bulunur ve link programına bu durum satır 2'de yapılan tanım ile bildirilmiştir.

Pascal programın ana kod bloğu içinde (satır 11–19) dışarıdan kullanıcının vereceği miktarda sayı (100 den az olmak kaydıyla) alınarak *Buffer* isimli diziye yerleştirilmektedir. Kullanıcı, veri girişinde CTRL+ Z tuşlarına basacak olursa dizi o ana kadar girilen kadar elemana sahip olacaktır (satır 13). Okuma işlemin bitmesini takiben girilen verinin değerlendirilmesi için *CheckRange* isimli harici assembly yordam çağırılmaktadır (satır18). Pascal ve assembly programlar arasındaki parametre aktarımı daha önceki örneğimizde de olduğu gibi yığın üzerinden yapılmaktadır. Yığına konan parametrelerin yığından kaldırılması görevi ise çağırılan yordama bırakılmıştır. Bu nedenle Pascal program içinde *CheckRange* yordamı –10 ve 10 değerleri ile çağırıldığında bu değerler yığına Pascal program tarafından geliş sırasında göre (soldan sağa doğru) yerleştirirken (Tablo 15-6-a) işlem bitip geri dönülmeden önce assembly yordam tarafından yığından kaldırılacaktır.

Daha önceki örneğimizden (Tablo 15-1 ve Tablo 15-2) farklı olarak bu sefer Pascal program içinde **{$F+}** komutu kullanılmamış, *CheckRange* isimli assembly yordam da NEAR olarak tanımlanmıştır. Bunun sonucu olarak derleme sonrasında elde edilecek çalışabilir kod, Pascal ve assembly yordamların içinde birlikte bulundukları tek bir kod kesimine[168] sahip olacaktır.

Yazdığımız Pascal program (Tablo 15-4) ile birlikte çalışacak olan assembly programımızda (Tablo 15-5) öncelikle *CheckRange* isimli assembly yordamın Pascal programdan çağırılmasını sağlayabilmek üzere **PUBLIC** (satır 2) ardından assembly program içinden çağırılacak *RangeError* isimli Pascal yordamı EXTRN olarak tanımlanmıştır (satır 3).

[168] Bu örneğimizde Pascal ve assembly yordamlar birbirlerini kesim içi yordam çağırma (intrasegment call) kuralları uyarınca çağırıp kullanacaktır.

Tablo 15-5: Pascal programı çağıran assembly kodu

```
CHECK.ASM
 1                  TITLE Pascal and Assembly programların birlikte kullanılması
 2                  PUBLIC CheckRange
 3                  EXTRN RangeError: NEAR
 4  DATA            SEGMENT WORD PUBLIC
 5                  EXTRN Buffer: WORD, Count: WORD
 6  DATA            ENDS
 7  CODE            SEGMENT BYTE PUBLIC
 8                  ASSUME CS: CODE, DS: Buffer
 9  CheckRange      PROC NEAR
10                  PUSH BP
11                  MOV BP, SP
12                  MOV AX, [BP+6]
13                  MOV DX, [BP+4]
14                  XOR BX, BX
15                  MOV CX, Count
16                  JCXZ SD4
17  SD1:            CMP Buffer[BX], AX
18                  JL SD2
19                  CMP Buffer[BX], DX
20                  JLE SD3
21  SD2:            PUSH AX
22                  PUSH BX
23                  PUSH CX
24                  PUSH DX
25                  PUSH Buffer[BX]
26                  CALL RangeError
27                  POP DX
28                  POP CX
29                  POP BX
30                  POP AX
31  SD3:            ADD BX, 2
32                  LOOP SD1
33  SD4:            POP BP
34                  RET 4
35  CheckRange      ENP
36  CODE            ENDS
37                  END
```

Assembly kodu Pascal ile birlikte kullanılırken veri kesiminin **DATA** adında, **WORD** ve **PUBLIC** olarak tanımlanması gereklidir[169] (satır 4). Bu kesim içinde

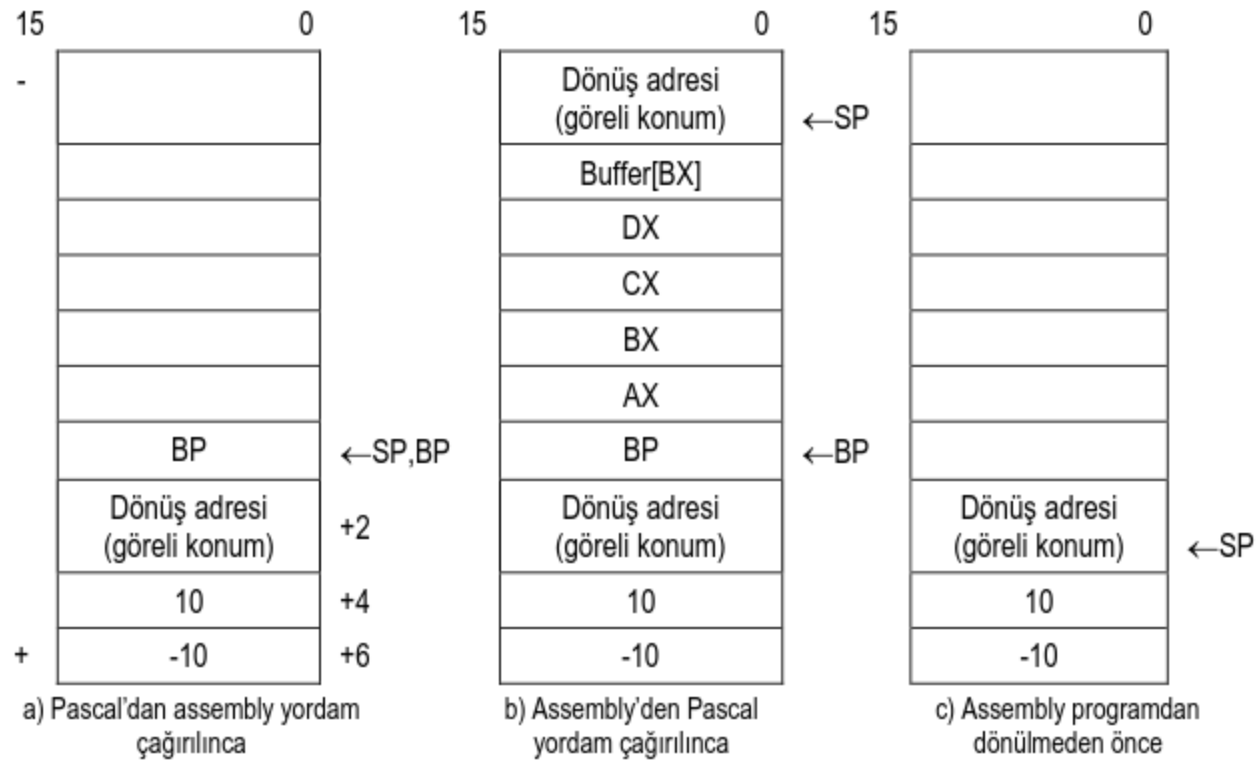
[169] Assembly programda kesim tanımında **WORD** ve **PUBLIC** kullanılması ile tanımlanan veri alanı her iki program tarafından paylaşılabilir olmuştur. Böylece assembly program, veri girişi Pascal program tarafından yapılan, *Buffer* isimli dizi üzerinde işlem yapabilmektedir. Üzerinde işlem yapılacak dizinin 100 elemanı olacağı da düşünüldüğünde parametrelerin bu şekilde aktarılması en uygun yoldur. Bu işlemler 11.2. bölümde açıklanan ortak veri alanı tanımlaması kuralları dahilinde yapılmıştır.

EXTRN komutu ile *Buffer* (word) ve *Count* (byte) isimli değişkenlerin tanımlandığı (satır 5) görülmektedir.

Assembly kod kesimi, Pascal programın gereği olarak **CODE** isminde, **BYTE** ve **PUBLIC** olarak tanımlanmıştır (satır 7). Veriye erişebilmek için **ASSUME** komutuyla DS yazmacı *Buffer* isimli alan ile ilişkilendirilmiştir (satır 8).

Parametrelerin sayısının az olması nedeniyle *CheckRange* isimli yordama parametreler yığın kullanılarak aktarılmaktadır (Tablo 15-6-a). Çağırılan yordamın yığın üzerinden aktarılan parametrelere erişmesi için -önceki değeri yığın üzerinde saklanmak kaydıyla- BP yazmacından yararlanılmaktadır. Böylelikle AX yazmacına -*10* (satır 12), DX yazmacına ise *10* değerleri (satır 13) BP yazmacı kullanılarak aktarılır. Kullanıcının veri girişi sırasında CTRL + Z tuşlarına basarak diziye eleman vermemiş olabileceği de göz önüne alınarak CX yazmacına atanan *Count* değişkeninin değeri kontrol edilmekte (satır 15), CX yazmacının değeri sıfır ise yordamı sonlandırmak üzere *SD4* etiketi ile belirlenen noktadan koda (satır 16) devam edilmektedir.

Tablo 15-6: Turbo Pascal – Assembly program arasında yığın üzerinden parametre aktarımı



Buffer isimli diziye sıra ile erişebilmek amacıyla BX yazmacı indis olarak kullanılmaktadır (satır 14). Dizi elemanlarının belirlenen değerler arasında olup olmadıkları AX ve DX yazmaçlarındaki değerler ile kontrol edilmekte, bu aralık içinde olmaları durumunda *SD3* ile belirlenen noktaya atlanıp aynı kontroller dizinin takip eden elemanları için gerçekleştirilmektedir. Koşulun sağlanmaması durumunda *SD2* ile belirlenen noktadaki kod çalıştırılarak Pascal'da yazılmış olan *RangeCheck*

isimli yordam çağırılıp ekrana mesaj yazdırılmaktadır (satır 26). Ancak Pascal'da yazılmış yordam çağırılmadan önce kullanılmakta olan AX, BX, CX ve DX yazmaçlarının değerlerinin bozulmaması için yığında saklandığı[170] (satır 21–24), koşulu sağlamayan elemanın değerinin yine yığın üzerine konarak (satır 25) Pascal yordamın çağırıldığı (satır 26) görülmektedir. Pascal program içinde yer alan *RangeError* isimli yordam da ekrana yazacağı parametreyi yığın üzerinden alacaktır. Pascal-assembly kullanımında yığın üzerinden aktarılan parametrelerin temizlenmesi çağırılanın sorumluluğunda olduğu için Pascal bu değeri yığından otomatik olarak kaldıracaktır. Pascal yordamdan dönülmesinden sonra (satır 27), önceki değerlerini yığında sakladığımız yazmaçlar yığına konuş sırasının tersinde (DX, CX, BX ve AX olarak) geri alınmaktadır (satır 27–30). İşlemler sırasında yığının durumu Tablo 15-6'da ayrıntılı olarak gösterilmiştir.

Assembly programdaki işlem ister normal, isterse kullanıcının diziye eleman vermemiş olmasından ötürü sonlanmış olsun, BP yazmacının yordamın başında yığın üzerinde saklanan değeri dönüşten hemen önce geri alınmalıdır (satır 33). Bu aşamadan sonra yığın assembly yordam çağırılmadan önceki halini alacaktır (Tablo 15-6-c). Pascal programa dönmek için RET komutu kullanıldığında Pascal yordam tarafından yığına yerleştirilen parametreler assembly program tarafından yığından kaldırılacaktır (satır 34).

Pascal ve assembly yordamlardan oluşan bu uygulamanın çalıştırılması için öncelikle *CHECK.ASM* isimli assembly program derlenip *CHECK.OBJ* isimli dosyanın oluşturulması, ardından Turbo Pascal ile *CHECKNUMBER.PAS* isimli programının derlenerek EXE uzantılı dosyanın elde edilmesi gereklidir. Turbo Pascal derleyicisi derleme işlemini takiben link işleminde otomatik olarak yapacak, bu sırada daha önce yapılmış olan **{$L CHECK}** tanımı uyarınca *CHECK.OBJ* isimli kodu kullanacaktır.

15.3 Turbo Pascal ve Assembly Kullanımında Dikkat Edilmesi Gerekenler

Turbo Pascal'dan assembly program çağırabilmek için **{$L}** komutu kullanılmalı, çağırılacak olan assembly yordam veya fonksiyon da harici olarak tanımlanmalıdır. Bunun için Pascal kod içinde;

Procedure ABC (x:integer); External;

Function XYZ (x:integer):integer; External;

tanımlarından yararlanılır. Çağırılacak olan assembly yordamın FAR olması durumunda **{$F}** direktifi ile Turbo Pascal'a durum bildirilmelidir.

Diğer taraftan assembly program içinde yer alacak bütün yordamlar CODE isimli kesim içinde, veriler ise DATA isimli kesim içinde tanımlanmalıdır. Kesimler

[170] Bu işlem, kontrolün ekrana mesaj yazmak için Pascal programa aktarılmasını takiben assembly yordamda kullanılan yazmaçların değerlerinin bozulmasını engeller.

BYTE veya WORD olarak tanımlanabilir. Ancak hiçbir şekilde sınıf tipi (Class Type) belirtilmemelidir. DATA ismi ile tanımlanan veri kesiminde EXTRN tanımı kullanılarak Pascal programa ait her yerden erişilebilen (global) değerlere ulaşılabilir (satır 5).

Pascal programdan assembly programa parametre aktarmak gerektiğinde soldan sağa doğru ilk gelen parametre ilk olarak yığına yerleştirilir. Yığındaki parametreler de çağırılan yordam tarafından yığından kaldırılır.

15.4 Turbo C'den Assembly Yordam Çağırılması

Her ne kadar C programlama dili orta seviyeli dil sınıfında yer alan bir programlama dili de olsa bazı uygulamalarda C ile assembly programların birlikte kullanılması da sık karşılaşılan bir durumdur.

Örnek 15-3

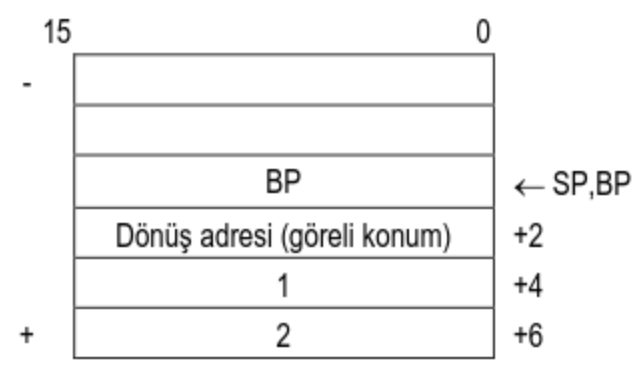
Turbo C'de tanımlı iki sayının toplamını assembly program içinde yer alan yordamın hesapladığı basit bir uygulamayı gerçekleştirelim. Uygulama gerçekleştirilirken C programla dilinin büyük/küçük harf duyarlı olduğu unutulmamalıdır.

Tablo 15-7: C'den assembly yordam çağırma

```
YAZ.C
1   #include <stdio.h>
2   main()
3   {
4           printf(' 1+ 2 = %d \n ', add(1,2));
5   }
```

Turbo C kullanarak yazdığımız bu basit program *add* isimli bir assembly yordamı kullanarak (Tablo 15-7, satır 4) toplama işlemini gerçekleştirecektir. Assembly yordama aktarılacak olan parametreler yığın üzerinden, ters sırada yerleştirilecektir. Yani sağdaki parametre ilk, soldaki ise son olarak yığına yerleşecektir (Tablo 15-8).

Tablo 15-8: Turbo C - Assembly program arasında yığın üzerinden parametre aktarımı



C programlama dilinin büyük/küçük harf duyarlı olduğundan daha önce bahis etmiştik. Bu nedenle assembly program içinde yer alacak olan *add* isimli yordamın da küçük harf ile yazılması gereklidir. Ayrıca C dilinden kullanılacak her türlü harici atıflarda değişkenlerin veya etiketlerin başına bir alt çizgi, ('_' sembolü) yerleştirilmelidir. Buna göre *TOPLA.ASM* isimli program aşağıdaki gibi olacaktır.

Tablo 15-9: C'den çağırılan assembly program

```
TOPLA.ASM
 1   _TEXT     SEGMENT BYTE PUBLIC 'CODE'
 2             ASSUME CS:_TEXT
 3   _add      PROC NEAR
 4             PUSH BP
 5             MOV BP,SP
 6             MOV AX,[BP+4]
 7             ADD AX,[BP+6]
 8             POP BP
 9             RET
10   _add      ENDP
11   _TEXT     ENDS
12             END
```

C ile birlikte kullanılacak olan assembly programda (Tablo 15-9) yordamın içinde bulunacağı kesim **_TEXT** adı ile **BYTE** ve **PUBLIC** olarak tanımlanacaktır. Sınıf tipi olarak **'CODE'** kullanılacaktır (satır 1). C programında tanımlı her türlü harici atfın önünde alt çizgi olması gerektiği kuralı uyarınca yordam *_add* olarak tanımlanmıştır (satır 3). Yığın üzerinden aktarılan parametrelere daha önceki örneklerimizde olduğu gibi BP yazmacı yardımıyla ulaşılmaktadır. Yordamın NEAR tanımlandığı dikkate alındığında yığın üzerinden (Tablo 15-8) parametrelere [BP+4] ve [BP+6] adreslerinden erişilmesi gerektiği görülmektedir (satır 6–7). *_add* isimli yordam bir fonksiyon şeklinde çalışmakta, yani bir tek değer döndürmektedir. Bu tür tek değerin döndürüldüğü durumlarda AX yazmacı dönüş değerini taşımaktadır. C programlama dilinde yığına yerleştirilen değerlerin kaldırılmasından çağıran sorumludur. Bu nedenle örneğimizde yığından değerleri almak *YAZ.C* isimli programın görevidir.

Yazdığımız programları birlikte çalıştırabilmek için Turbo C derleyicisinden yararlanabiliriz. Bunun için şu satırın yazılması yeterli olacaktır;

TCC –IC:\TURBOC\INCLUDE –LC:\TURBOCLIB *YAZ.C TOPLA.ASM*

15.5 Assembly'den Turbo C Yordam Çağırma

C program içinden assembly dilinde yazılan yordamlar çağırılabildiği gibi gerekli olması durumunda tam tersini de yapmak mümkündür. Böylece yüksek seviyeli

programlama dilinin sağladığı imkânlardan assembly kod içinden de yararlanmak mümkün olacaktır. Bu amaçla yazılmış olan Örnek 15-4'ü inceleyelim;

Örnek 15-4

C programlama dilinde yazılmış *long dcalc (int *fact1, int fact2, int fact3);* şeklinde tanımlı, tamsayı tipinde bir işaretçi ve tam sayı tipinde iki değişkeni parametre olarak alan bir yordamı[171] assembly yordam içinden çağırmak için yazılması gereken kod Tablo 15-10'da verilmiştir.

Tablo 15-10: Assembly programdan C program çağırma

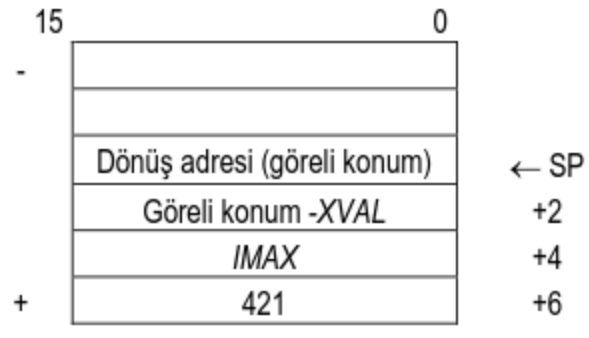
```
TOPLA.ASM
 1                EXTRN  _dcalc: NEAR
 2   _DATA        SEGMENT WORD PUBLIC 'DATA'
 3   IMAX         DW ?
 4   XVAL         DW 10 DUP(?)
 5   ANS          DW 2 DUP(?)
 6   _DATA        ENDS
 7   _TEXT        SEGMENT BYTE PUBLIC 'CODE'
 8                ASSUME CS:_TEXT
 9   BASLA        PROC NEAR
10                MOV AX, 421
11                PUSH AX
12                PUSH IMAX
13                LEA AX, XVAL
14                PUSH AX
15                CALL _dcalc
16                ADD SP,6
17                MOV ANS, AX
18                MOV ANS+2, DX
19                INT 20H
20   BASLA        ENDP
21   _TEXT        ENDS
22                END BASLA
```

Assembly programın C'de yazılmış olan *dcalc* isimli yordama (fonksiyona) erişebilmesi için **EXTRN** ile gerekli tanım yapılmıştır (Tablo 15-10, satır 1). Bu harici yordam tanımı NEAR olarak yapıldığı için çağırma işlemi kesim içinde (intrasegment call) gerçekleşecektir. Diğer yandan kesim tanımlarında, veri kesimi **_DATA** isminde (satır 2), kod kesimi ise **_TEXT** isminde (satır 7) tanımlanmıştır. Assembly yordam içinden aktarılacak parametreler yığına PUSH komutları ile yerleştirilmektedir (satır 10–14). C'de yazılmış *dcalc* isimli yordamı (fonksiyonu) ça-

[171] Bir tek dönüş değeri olduğu için fonksiyon olarak adlandırmak daha doğru olacaktır.

ğırmak için **CALL _*dcalc*** (satır 15) komutu kullanılır. Assembly ve C birlikte kullanıldığında yığına yerleştirilen parametreleri kaldırmak çağıran tarafın görevi olup bu işlem assembly program tarafından SP yazmaç değerinin değiştirilmesi ile yapılmaktadır (satır 16). *dcalc* isimli yordamın (fonksiyonun) dönüş değeri *long* yani *dword* olduğu için AX ve DX yazmaçları dönüş değerini taşımaktadırlar. Bunlar da sırası ile *ANS* isimli veri alanına yerleştirilmektedir (satır 17–18).

Tablo 15-11: Assembly – C arasında yığın üzerinden aktarılan parametreler



15.6 Turbo C ve Assembly Kullanımında Dikkat Edilmesi Gerekenler

Turbo C'de herhangi bir harici atıfla erişilmek istendiğinde derleyici onun başına bir alt çizgi sembolü yerleştirmektedir. Buna göre assembly tarafında kullanılacak değişkenler, etiketler vs. başına da bu sembol eklenmelidir. Buna göre;

- Assembly programda, C'de yazılmış yordama erişebilmek için yapılması gereken tanımlar:

  EXTRN _cfunc1: NEAR, _cfunc2:FAR

  şeklinde olmalıdır.

- C programda tanımlanmış veriye erişmek için assembly kod içinde yapılması gereken tanımlar Tablo 15-12'de verilmiştir.

Tablo 15-12: C ve Assembly bağlantısında dikkat edilmesi gerekenler

C	Assembly
int i, dizi[10]; char ch; long result;	EXTRN _i: WORD, _dizi: WORD EXTRN _ch: BYTE EXTRN _result: DWORD

- C'den assembly programa aktarılan parametreler yığın üzerinde ters sırada yerleştirilir. Yani soldan sağa doğru son parametre ilk, ilk parametre son olarak yerleşir.
- Yığın üzerinden aktarılan parametrelerin kaldırılması onları yığına koyan tarafın sorumluluğudur.

16. EKLER

Bu kitap kapsamında kullanılan uygulamaların kurulumu, debug kullanımı, ve kesmeler ile ilgili bazı bilgiler ve örnek programlar ek olarak verilmiştir.

16.1 Çalışma Ortamının Kurulması

Gerçek kipte assembly programlama yapabilmek için kullanıcıların öncelikle kendilerine uygun çalışma ortamını hazırlamaları gereklidir. Çalışma ortamının iki önemli parçası vardır. Birinci parça bilgisayar sisteminin donanım unsurları ile kullanıcı arasındaki ilişkiyi tesis edecek olan işletim sistemi, ikinci parça ise assembly dilinde programlama yapmak için kullanılacak yazılımlardır.

İşletim sistemi olarak tüm 80x86 tabanlı bilgisayarlarda gerçek kipte çalışan, bellek ve sabit diskte az yer kaplayan, kolay ve çabuk öğrenilebilir, düşük donanımlı bilgisayarlarda çalışabilme gibi avantajları nedeniyle DOS[172] (Disk Operating System) kullanılabilir.

Programlamada kullanılacak yazılımlar ise; metin düzenleyici (editor), assembler (assembly dili derleyicisi), linker (link işlerinde kullanılacak olan program), tip dönüşümü yapan uygulama (COM uzantılı programlar üretebilmek amacıyla) ve hata bulucudan (debugger) oluşur.

Kitap kapsamındaki örnekler ve işleyiş, işlemcinin gerçek programlama kipinde, Microsoft Macro Assembler[173] kullanılarak anlatılmıştır. Doğal olarak kullanılan derleyici, Linker gibi yazılım unsurları da Microsoft firmasının lisanslı ürünleri arasından seçilmiştir. Ancak assembly programla yapmak için özgür belgeleme lisansı (GNU GPL) kapsamında temin edilebilen ücretsiz assembly derleyicileri[174] ve ilgili araçlardan da yararlanılabilir.

Buna göre her platformda kullanacağımız temel yazılım listesi Tablo 16-1'de verilmiştir. Bunlara ek olarak kullanılması gereken metin düzenleyici, EXE'den COM'a dönüştürücü ve hata bulucu yazılımları işletim sisteminin parçası olabileceği gibi ihtiyaç olması durumunda üçüncü parti sağlayıcılardan rahatlıkla temin edilebilirler.

---

[172] DOS, tek görevli, grafik kullanıcı arabirimi kullanmayan bir işletim sistemi olmak ile birlikte gerçek kipte assembly programlama yapmak için yeterlidir. Assembly programlar, günümüz 32 bit'lik modern işletim sistemlerinde halen komut satırından (cmd) DOS siyah ekran uygulaması (DOS black screen) olarak çalışabilmektedir. MS-DOS, PC-DOS ve IBM-DOS olarak bilinen lisanslı dağıtımların yanı sıra FreeDOS olarak bilinen GNU-GPL kapsamında kullanılabilen (http://www.freedos.org/) bir seçenek de vardır.

[173] Microsoft firmasının ürünlerinin kullanımı için lisans sahibi olunması gereklidir.

[174] FreeDOS projesi kapsamındaki JWASM http://www.freedos.org/software/?prog=jwasm adresinden, FASM http://www.freedos.org/software/?prog=fasm adresinden, NASM http://www.freedos.org/software/?prog=nasm adresinden, WASM http://www.freedos.org/software/?prog=wasm adresinden ve YASM http://www.freedos.org/software/?prog=yasm adresinden temin edilebilir.

Tablo 16-1 Temel yazılım listesi

Yazılım	Açıklama
MASM.EXE	Microsoft Macro Assembler
LINK.EXE, ML.ERR, ML.EXE	Microsoft Linker
EXE2COM.EXE[175] veya EXE2BIN.COM[176]	EXE'den COM'a dönüştürücü
DEBUG.COM[177]	Hata bulucu program

16.1.1 Mevcut İşletim Sistemini Kullanmadan Çalışma Ortamının Kurulması

80x86 işlemcili bir bilgisayar sistemi (ne kadar eski veya yeni olduğuna bakılmaksızın) basit bir DOS açılış disketi veya USB bellek oluşturularak assembly programla yapmak için rahatlıkla kullanılabilir. Bu sayede bilgisayar sisteminin üzerinde çalışmakta olan işletim sistemi üzerinde ek ayarlar yapmak gerekmeyeceği gibi bu ayarlamaları yaparken ortaya çıkabilecek problemlerden de korunmuş olacaktır.

Günümüzde 80x86 donanım platformu üzerinde Microsoft firmasının Windows olarak adlandırdığı işletim sistemleri, farklı Linux dağıtımlarının yanı sıra donanım platformunu değiştiren Apple firmasının Mac OS işletim sistemi de çalışmaktadır. Bu çeşitlilik karşısında mevcut işletim sistemlerinde kurulum ve/veya ayar değişiklikleri yapmadan, gerçek çalışma kipinde assembly programlama yapmak için bilgisayar sistemini açmayı sağlayacak nitelikte bir açılış (boot) disketi veya USB bellek oluşturmak daha hızlı bir çözüm olabilmektedir. Kullanılacak olan yazılımların lisans sözleşmelerine uygun olarak veya özgür belgeleme lisansı (GNU GPL) kapsamında kullanılması gerekir. Buna göre, sistem açılışında kullanılacak olan disket veya USB bellek, kullanıcının sahip olduğu lisanslı işletim sistemi[178] ile yaratılabileceği gibi FreeDOS kullanan bazı uygulamalar kullanılarak da yaratılabilir. Buna göre;

- Açılış disketi yaratmak için:

---

[175] ftp://ftp.columbia.edu/pub/packet-drivers/bin/exe2com.exe adresinden Chris Dunford/The Cove Software Group tarafından yazılmış program temin edilebilir.

[176] EXE2BIN isimli program MS-DOS 5.00 kadar işletim sisteminin bir parçası olarak dağıtılmış, daha sonraki sürümlerinde ise standart dağıtım listesinden çıkartılmıştır. Eğer MS-DOS 5.00 dan ileri bir sürüm kullanılacak ise EXE2BIN.COM isimli program ftp://ftp.columbia.edu/pub/packet-drivers/bin/exe2com.exe adresinden, kullanılan işletim sistemi FreeDOS olacak ise EXE2BIN.COM isimli program http://www.freedos.org/software/?prog=exe2bin adresinden temin edilebilir.

[177] DEBUG.COM genel olarak, (MS-DOS, IBM-DOS, Windows 9x, Windows XP gibi) işletim sistemlerinde bulunmaktadır. FreeDOS ile yaratılan açılış disketi veya USB bellekte bu program yok ise, FreeDOS projesi dahilinde http://www.freedos.org/software/?prog=debug adresinden temin edilebilir.

[178] Windows 9.x ve Windows XP işletim sistemlerinde disket biçimlendirirken (format) sunulan seçeneklerden "MS-DOS başlangıç disketi oluştur" seçilerek başlangıç disketi oluşturmak mümkündür. Ancak bu işletim sistemlerinde mevcut sabit disk üzerinde gerekli kurulumları yapmak da oldukça kolaydır.

  - http://www.bootdisk.com/bootdisk.htm/
  - http://www.freedos.org/
- Açılış USB belleği yaratmak için:
  - http://rufus.akeo.ie/

adreslerinden ücretsiz indirilebilen ve GNU GPL kapsamında kullanılabilen uygulamaları kullanarak FreeDOS işletim sistemi ile sistemi açan (boot) bir disket veya USB bellek oluşturabiliriz. Bu işlemi takiben açılış için hazırladığımız ikincil bellek birimi üzerindeki bir dizin altına (örneğin *\assembly*) temel yazılımlarımızı kopyalayarak çalışma ortamının hazırlanması işini tamamlamış oluruz. *\assembly* dizini altına geçilerek 7. bölümdeki açıklamalara uygun olarak programlama yapılabilir. FreeDOS işletim sistemi ile gelen EDIT.COM isimli metin düzenleyicisi ihtiyaçlarımızı karşılamaya yeterli olacağı için ek bir metin düzenleyici yazılımı arayışına girmeye gerek yoktur.

16.1.2 Microsoft İşletim Sistemlerinde Çalışma Ortamının Oluşturulması

Günümüzde masaüstü ve taşınabilir sistemlerde kullanılan en yaygın işletim sistemleri Microsoft firması tarafından geliştirilmiş olan "Windows" olarak adlandırılan işletim sistemleridir. İlerleyen bölümlerde bu işletim sistemlerinde uygulanabilecek yapılandırmalar açıklanacaktır.

16.1.2.1 Windows 9x

Bilgisayarımızda hali hazırda yüklü olan Windows 9x işletim sistemi üzerinde çalışmalar yapabilmek için Tablo 16-1'de yer alan temel yazılımlara ek olarak;

- Bellek yönetimi için DOSXMSF.EXE, uygulamaların DOS penceresinden çalışması için DOSXNT.386 ve DOSXNT.EXE isimli programların sabit disk üzerindeki uygun bir dizin altına (örneğin C:\assembly) kopyalanır
- SISTEM.INI dosyasının metin düzenleyici program ile açılarak [386Enh] bölümüne "*device=C:\assembly\dosxnt.386*" satırı eklenir
- Sistem tekrar başlatılır.

Sistem açıldıktan sonra MS-DOS kutusundan[179] *C:\assembly* dizini altına geçilerek 7. bölümdeki açıklamalara uygun olarak programlama yapılabilir. Windows ortamında çalışılıyor olması nedeniyle işletim sisteminde var olan Notepad uygulaması metin düzenleyici olarak kullanabilir. Ancak kullanıcılar kullanımını daha rahat buldukları başka metin düzenleyicilerinden de yararlanabilir.

[179] Start (Başlat)→Run (Çalıştır) seçilerek içine "*command*" yazılarak MS-DOS kutusu açılabilir.

16.1.2.2 Windows XP

Bilgisayarımızda hali hazırda yüklü olan Windows XP işletim sistemi üzerinde çalışmalar yapabilmek için aşağıdaki adımların gerçekleştirilmesi gereklidir.

- Tablo 16-1'de yer alan temel yazılımlar sabit disk üzerindeki uygun bir dizin altına (örneğin *C:\assembly*) kopyalanır.
- Start (Başlat)→Run (Çalıştır) seçilerek içine "*Regedit*" yazılır ve sistemin kayıt defteri düzenleyicisi (registery editor) çalıştırılır[180].
- *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage* sekmesini bulunur.
- *ACP* anahtarının değeri *1254* yapılır.
- *OEMCP* anahtarının değeri *857* yapılır.
- Sistem tekrar başlatılır.

Sistem açıldıktan sonra Start (Başlat)→Run (Çalıştır) seçilerek içine "*cmd*" yazılarak MS-DOS kutusundan "*CD C:\assembly"* komutu ile ilgili dizin altına geçip Notepad veya benzer bir metin işleme programı ile 7. bölümdeki açıklamalara uygun olarak programlama yapılır.

16.1.2.3 Windows 7

Bilgisayarımızda hali hazırda yüklü olan Windows 7 işletim sisteminin 32 bit'lik veya 64 bit'lik olması izlenecek yolun değişik olmasına neden olmaktadır. 32 bit'lik Window 7 işletim sisteminde Tablo 16-1'de yer alan temel yazılımların bir dizin altına (örneğin *C:\assembly*) kopyalanması ve DOS penceresinden "*CD C:\assembly"* komutu ile ilgili dizin altına geçip Notepad veya benzer bir metin işleme programı ile 7. bölümdeki açıklamalara uygun olarak programlama yapılması mümkündür.

Ancak 64 bit'lik Windows işletim sisteminde bu şekilde bir çalışma yöntemi izlenemez. Bu durumda sanallaştırma (virtualization) yazılımları veya emülator uygulamaları kullanarak[181] gerekli olan çalışma ortamı oluşturulabilir.

Windows işletim sistemi altında (eğer donanım kaynakları yeterli düzeyde ise) sanal (virtual) bilgisayarlar oluşturarak, uygulama geliştirmek ve bazı yazılımları mevcut işletim sistemimize hiçbir zarar vermeden, tamamen farklı bir bilgisayar

[180] Sistem kayıt defteri (registry) üzerinde yapılan değişiklikler sistemin çalışmasını etkileyebilir. Bu tamamen sizin sorumluluğunuzdadır.

[181] Emülatör uygulamaları farklı işletim sistemleri üzerinde çalışabildikleri için ilerleyen bölümde ayrı bir başlık olarak açıklanacaktır.

sistemi üzerinde çalışıyormuş gibi test etmek mümkündür. Bu amaçla kullanılabilen sanallaştırma yazılımları şunlardır;

- Vmware Workstation[182]
- Microsoft Virtual PC 2007[183]
- Microsoft Windows Virtual PC[184]

Söz konusu yazılımlardan herhangi birini mevcut işletim sistemimize kurduğumuzda, herhangi bir Windows uygulaması gibi farklı bir pencere içinde ve kendisine verilmiş olan kaynakları kullanarak çalışan bir sanal kişisel bilgisayar elde etmiş oluruz. Bu sanal bilgisayar üzerine doğrudan assembly çalışmaya imkân sağlayan bir işletim sistemi (DOS, Windows 9.x veya Windows XP gibi) kurmak ve assembly için gerekli ayarları daha önceki bölümlerde anlatılan şekilde yapmak kaydı ile çalışma ortamımızı oluşturmuş oluruz.

16.1.3 Linux İçin Çalışma Ortamını Oluşturma

Linux işletim sisteminde çalışma ortamını oluşturabilmek için iki seçenek bulunmaktadır. Bunlardan ilki Windows işletim sistemlerinde olduğu gibi lisans ücretline tabii Vmware Workstation[185] uygulamasının kurulması ile oluşturulacak sanal bilgisayar üzerine doğrudan assembly çalışmaya imkân sağlayan bir işletim sistemi (DOS, Windows 9.x veya Windows XP gibi) kurmak ve assembly için gerekli ayarları daha önceki bölümlerde anlatılan şekilde yapmak kaydı ile çalışma ortamımızı oluşturmaktır.

İkincisi seçenek ise kullanılmakta olan Linux dağıtımında çalışabilecek, ücretsiz DOSEMU[186] uygulamanın kurulmasıdır. DOSEMU kurulduktan sonra;

- Linux işletim sistemindeki bir kullanıcıya bağlanılır (login).

---

[182] Vmware Workstation ürünü 32 ve 64 bit'lik, Windows 8, Windows 7, Windows Vista, Windows XP SP3, Windows Server 2012, Windows Server 2008, Windows Server 2003 Standard, Windows XP Home Edition SP2 işletim sistemleri üzerinde çalışabilen ücretli bir yazılımdır. 30 günlük deneme sürümü http://www.vmware.com/products/workstation/overview.html adresinden indirilebilmektedir.

[183] Virtual PC 2007, Microsoft firmasının 32 ve 64 bit'lik, Windows Vista (Business, Enterprise, Ultimate; Windows Server 2003 (Standard Edition, Standard x64 Edition), Windows XP (Professional, Professional x64 Edition, Tablet PC Edition) işletim sistemlerini lisanslı kullananların http://www.microsoft.com/en-us/download/details.aspx?id=4580 adresinden indirebileceği bir yazılımdır.

[184] Microsoft Virtual PC, Microsoft firmasının 32 ve 64 bit'lik, Windows 7 (Home Basic, Home Premium, Professional, Ultimate ve Enterprise) işletim sistemini lisanslı kullananların http://www.microsoft.com/en-us/download/details.aspx?id=3702 adresinden indirebileceği bir yazılımdır.

[185] Vmware Workstation ürünü, Ubuntu 8.04 ve üstü, Red Hat Enterprise Linux 4.5 ve üstü, CentOS 5.0 ve üstü, Oracle Linux 5.0 ve üstü, openSUSE 10.2 ve üstü, SUSE Linux 10 ve üstü işletim sistemleri üzerinde çalışabilen ücretli bir yazılımdır. 30 günlük deneme sürümü http://www.vmware.com/products/workstation/overview.html adresinden indirilebilmektedir.

[186] GNU-GPL kapsamında kullanılabilen dosemu, http://www.dosemu.org/stable/ adresinden indirilebilir.

- Bir terminal penceresi açılır ve "*dosemu"* yazılarak uygulamanın kullanıcı kopyası oluşturulur (Bu aşamada ekranda çıkan mesajlar iyice okunarak verilen uyarılar doğrultusunda işlemler yapılmalıdır). Kurulum tamamlandıktan sonra ekranda "*C:\>* " şeklinde bir satır belirecektir.
- "*C:\>mkdir assembly*" yazılarak yeni bir dizin yaratılır.
- "*C:\>cd assembly*" komutu ile yaratılan dizine geçilir.
- Yeni bir teminal penceresi açılır ve "*cd dosemu/freedos/assembly*" komutu ile bu dizin altına geçilir ve Tablo 16-1'de yer alan temel yazılımlar bu alana (*dosemu/freedos/assembly*) kopyalanır.
- İçinde bulunulan terminal penceresi kapatılır.
- İlk terminal penceresinden "*C:\>cd assembly*" yazılarak ilgili dizine geçilir. 7. bölümdeki açıklamalara uygun olarak programlama yapılır.

Programlarınızı yazmak için Linux işletim sisteminde bulunan vi veya gedit isimli metin düzenleyici programlarından yararlanılabilir.

16.1.4 Emülatör Kullanımı

Günümüzde farklı işletim sistemleri üzerinde çalışabilen DOS emülatörleri de bulunmaktadır. Bunlardan en bilindik olanı DOSBox[187] isimli, tamamen ücretsiz ve açık kaynak kodlu yazılımdır. İşletim sistemi ile uyumlu DOSBox uygulaması kurulduktan sonra yapılması gereken adımlar şunlardır;

- Tablo 16-1'de yer alan temel yazılımlar bilgisayarın diski üzerinde bir dizine (örneğin c:\assembly) kopyalanır.
- DOSBox uygulaması simgesine tıklanarak başlatılır. Uygulama, açılan siyah DOSBox penceresi içinde "Z:\>" komut satırının görünmesi ile başlamıştır.
- "*Z:\>mount y c:\assembly*" komutu verilerek DOSBox uygulamasının yazılımların olduğu dizine erişimi sağlanır. Komut satırından "*Z:\>y:*" yazıldıktan sonra 7. bölümdeki açıklamalara uygun olarak programlama yapılır.

16.2 Debug Komutları

Assembly dili ile program yazarken en çok kullanılan araçlardan birisi hata bulucu (debugger) programıdır. Her ne kadar günümüzde çok değişik, zengin grafik arayüzleri bulunan, kullanıcıya ek kolaylıklar sağlayan pek çok yazılım (Turbo Debugger, CodeView vb.) bulunuyor olsa da hemen hemen her zaman ilk akla ge-

[187] DOSBox, Microsoft Window, FreeBSD, Fedora Core (Linux), Gentoo (Linux), Mac OSX, OS/2, BeOS ve RiscOS gibi farklı işletim sistemlerin üzerinde çalışabilmektedir. Yazılım, http://www.dosbox.com/download.php?main=1 adresinden ücretsiz olarak indirilebilir.

len DEBUG.COM isimli programdır. DEBUG.COM, assembly komutlarının nasıl çalıştıklarını deneyerek görmek, yazılan programların amacına uygun çalışıp çalışmadığını anlamak ve varsa hatalarını düzeltmek için kolay bulunulabilir ve kullanılabilirliğinden ötürü tercih nedenidir.

DEBUG programını kullanarak var olan çalıştırılabilir kodu incelemek için o programın isminin (EXE veya COM uzantısıyla birlikte) parametre olarak programa verilmesi gereklidir. Diğer yandan DEBUG ortamında basit kodlar yazarak denemeler yapmak isterseniz sadece DEBUG yazmak yeterlidir. DEBUG komutu çalıştırıldıktan sonra sadece bu program içinde geçerli olan komut kümesinden (Tablo 16-2) yararlanmak mümkündür.

DEBUG ortamında yer alan bütün gösterimlerde on altılı tabanda (hexadecimal) sayılar kullanılmaktadır. Bu özellik herkes tarafından bilindiği için sayıların gösterimlerinin sonunda ek bir H (Hex) eki bulunmaz.

DEBUG programı çalışmaya başladığını ekrandaki **'-'** işareti ile belirler. Bu işaretten sonra DEBUG programı içinde tanımlı olan komutlar verilerek (Tablo 16-2) işlemler gerçekleştirilir.

Tablo 16-2: Debug komutları ve basit açıklamaları

Komut	Açıklama
R	Yazmaç ve bayrakların içeriğinin görüntülenmesi ve değiştirilmesi için kullanılır
D	Belirlenen bellek adresinin içeriğinin görüntülenmesi için kullanılır
A	Assembly kod yazmak için kullanılır
T	Assembly komutların çalıştırılması için kullanılır
U	Belirlenen bellek adresinden itibaren yer alan nesne kodların anımsatıcı karşılıklarını göstermek için kullanılır
E	Belirlenen bir bellek adresinden itibaren değerlerin değiştirilmesi için kullanılır
F	Belirlenen bellek bölgesinin verilen değer ile doldurulması için kullanılır
C	Belirlenen bellek bölgelerinin içeriklerini karşılaştırmak için kullanılır
G	Assembly program sonlanıncaya kadar (RET/RETF komutuna veya programı sonlandırmayı sağlayan INT komutuna kadar) olan kodları çalıştırmak için kullanılır
H	On altılı tabanda toplama ve çıkarma işlemi yapmak için kullanılır
L	Belleğe programı tekrar yükleyip baştan başlamak için kullanılır
P	Assembly komutların çalıştırılması için kullanılır. T komutundan farklı olarak INT, CALL ve LOOP anımsatıcısı ile karşılaştığında işlem tamamlandıktan sonraki durumu gösterir
Q	Debug ortamından çıkıp komut satırına dönmek için kullanılır.

En sık kullanılan komutlar ve kullanımlarına ilişkin örnekler aşağıda verilmiştir.

16.2.1 R (Register-Yazmaç)

Bu komut o anda yazmaçlar ve bayrakların durumunu göstermek için kullanılır. R komutunun yazılıp ENTER tuşuna basılmasını takiben ekranda 3 satır görünecektir. Bunlardan ilki genel amaçlı veri yazmaçları, işaret ve indis grubu yazmaçlarının değerlerini gösterir. İkinci satır kesim yazmaçları, IP ve o anda bayrakların sahip olduğu değerleri ifade eden kısaltmaları ve nihayet üçüncü satırda o anda işletilmek üzere bekleyen komutun ne olduğunu ve komut, veri kesimi ile ilgili bir işlem yapıyor ise veri kesimindeki ilgili bellek gözünün içeriğini göstermektedir (Tablo 16-3).

Tablo 16-3: Debug (-R) yazmaçlar ve bayrakların görüntülenmesi

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2AF2  ES=2AF2  SS=2AF2  CS=2AF2  IP=0100   NV UP EI PL NZ NA PO NC
2AF2:0100 8B24          MOV     SP,[SI]                     DS:0000=20CD
-
```

Herhangi bir yazmacın değerinin değiştirilmesinin gerektiği durumlarda R komutunu takiben yazmacın ismi yazılıp ENTER tuşuna basılır. Böylelikle ekranda o anda o yazmaçta olan değer belirecek, hemen alt satırın başında da "**:**" işareti çıkacaktır. Bu aşamada yazmaca on altılı sayı sisteminde bir değer verilebilir. Örneğimizde AX yazmacının değeri 12AF olarak değiştirilmiştir (Tablo 16-4).

Tablo 16-4: Debug (-R AX) yazmaç değerinin değiştirilmesi

```
-R AX
AX 0000
:12AF

-R
AX=12AF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2AF2  ES=2AF2  SS=2AF2  CS=2AF2  IP=0100   NV UP EI PL NZ NA PO NC
2AF2:0100 8B24          MOV     SP,[SI]                     DS:0000=20CD
-
```

Bayrakların durumlarını değiştirmek istersek R komutundan sonra F yazmamız yeterlidir. Böylece bayrakların mevcut durumları gösterilecek ve beliren '–' sembolünden sonra yeni değerleri ifade eden kısaltmalar (

Tablo 16-6) kullanılabilecektir. Yeni değerleri yazarken herhangi bir sıra takip etme zorunluluğu yoktur. Sadece değerini değiştirmek istediğimiz bayrakları ifade eden kısaltmaların boşluklar ile ayrılarak yazılması yeterlidir.

Tablo 16-5: Debug (-R F) bayrak değerlerinin değiştirilmesi

```
-R F
NV UP EI PL NZ AC PO CY  -
```

Tablo 16-6: Debuger'da kullanılan bayrak kısaltmaları

Bayrak İsmi	Set (1)	Clear (0)
Overflow - OF	OV (overflow)	NV (not overflow)
Direction - DF	DN (decrement - down)	UP (increment - up)
Interrupt - IF	EI (enable)	DI (disable)
Sign - SF	NG (negative)	PL (plus)
Zero – ZF	ZR (zero)	NZ (not zero)
Auxilary Carry- AF	AC (auxilary carry)	NA (not auxilary)
Parity – PF	PE (even parity)	PO (odd parity)
Carry – CF	CY (carry yes)	NC (not carry)

16.2.2 D (Dump / Display – Döküm / Görüntüleme)

Bu komut belleğin istediğimiz bir noktasından (kesim ve göreli konum ikilisi ile adresini belirtmek kaydıyla) içeriğini görüntülemek için kullanılır.

Tablo 16-7: Debug (-D DS:0) veri kesiminin içeriği

```
-D DS:0
2AF2:0000  CD 20 FF 9F 00 9A EE FE-1D F0 4F 03 5A 25 8A 03   . ........o.Z%..
2AF2:0010  5A 25 17 03 5A 25 14 24-01 01 01 00 02 FF FF FF   Z%..Z%.$........
2AF2:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 3B 25 4E 01   ............;%N.
2AF2:0030  16 2A 14 00 18 00 F2 2A-FF FF FF FF 00 00 CE 0B   .*.....*........
2AF2:0040  06 1E 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2AF2:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20   .!...........
2AF2:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20           .....
2AF2:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00           ........
-
```

Tablo 16-7'de veri kesiminin (DS), 0000H göreli konumundan itibaren olan kısmının dökümü ekranda belirmiştir. Eğer adres verilirken herhangi bir aralık verilmemiş ise komut **128 byte** büyüklüğünde veriyi ekranda gösterecektir. Ekranda beliren veri her satırda 16 byte olacak şekilde düzenlenmiştir. Her satır soldan sağa doğru incelendiğinde kesim adresi, göreli konum adresi, 16 byte bellek içeriği (ilk 8 byte ile ikinci 8 byte arasında beliren – sembolü okumayı kolaylaştırmak amacıyla kullanılır) ve nihayet her satırda yer alan 16 byte'ta karşılık gelen ASCII karakterler görülecektir. Kullanıcının bellek üzerinde sınırlı bir alanı görmek istemesi durumunda komut satırının göreli konum adres bilgisi Tablo 16-8'de olduğu gibi de verilebilir.

Tablo 16-8: Debug (-D DS:0110, 0115) bellek içeriği

```
-D DS:0100,0115
2AF2:0100  8B 24 17 5A 18 BA 4A 29-A2 4F F6 C9 07 AE 07 09   .$.Z..J).O......
2AF2:0110  20 8C 07 BA 4F 80                                  ...O.
-
```

16.2.3 A (Assembly)

Bu komut DEBUG programı yardımıyla basit programlar yazmayı sağlar. DEBUG içinde yazılan programlar COM uzantılı programlarda olduğu gibi **0100H** göreli konum adresinden başlarlar. Bu nedenle program çalıştırılacağı zaman mutlaka IP yazmacının değerinin R komutu ile 0100H'e getirilmesi gerekmektedir.

Tablo 16-9: Debug (-A 100) assembly program yazmak

```
-A
2AF2:0100 MOV AX,5
2AF2:0103 ADD AX,4
2AF2:0106 SHR AX,1
2AF2:0108 ^C
-
```

A komutu verildikten sonra yazılacak kod, CS ve IP yazmaçlarının gösterdiği bellek alanından başlayarak yerleştirilecektir. Her geçerli komut girildiğinde bir sonraki komutun hangi göreli konum adresinden itibaren yazılacağı otomatik olarak hesaplanmaktadır. Yazım sırasında kurallara uygun olmayan bir komut satırı yazılacak olursa ekranda hatalı giriş yapıldığını gösteren bir mesaj belirecek ve göreli konum adresinde bir değişiklik olmayacaktır[188]. Böylece kullanıcı komutun doğru şeklini yazmak zorunda kalacaktır. Kodun sonuna gelindiğinde ise yeni gelen satırda ENTER tuşuna basılarak kodlama sonlandırılır.

16.2.4 T (Trace / İzleme)

T komutu yardımıyla program içindeki assembly komut satırları teker teker çalıştırıp işlem sonunda yazmaç ve bayrakların aldıkları değerler ekranda gösterilir. T komutu çalışmaya IP yazmacının gösterdiği noktadan itibaren başlar ve her komut işlendikten sonra durur.

Tablo 16-10'de önce R komutu kullanılarak yazmaç, bayrak değerleri ve çalıştırılmak üzere bekleyen komut satırı olan MOV AX, 0005 görüntülenmiştir. T komutunun kullanılması ile ilgili komut çalıştırılmış ve hemen arkasından yazmaç, bayrak değerleri ve bir sonraki seferde çalıştırılacak olan komut görüntülenmiştir. MOV komutu doğası gereği bayraklar üzerinde bir değişikliğe neden olmaz. Ancak takip eden diğer komutlar T komutu ile çalıştırıldıktan sonra gerek AX yazmacının değeri, gerekse bayrakların değerlerinde değişiklikler olmuştur. T komutu, her anımsatıcıyı çalıştırdıktan sonra yapılan işlemi sonucunu gösterdiği için kullanıcıların adım adım program işleyişini incelemesine imkân sağlamaktadır.

---

[188] Özellikle Linux işletim sistemindeki dosemu uygulaması altında çalışan DEBUG programı bazı komutlarda işlenen tiplerindeki uyuşmazlıktan kaynaklanan hataları kullanıcıya göstermeden uygun bir şekilde düzeltmektedir. Örneğin MOV AL, BX gibi bir komut normalde işlenenlerinin tiplerinin uyuşmaması (ilki byte, ikincisi word) nedeni ile hatalı olması gerekirken bu komut DEBUG tarafından otomatik olarak MOV AX, BX şekline dönüştürülerek hata ortadan kaldırılmaktadır. Bu özellik dikkate alınarak kullanıcıların yazdıkları kodu ve bununun sistem tarafından nasıl kabul edildiğini izlemelerinde sayısız fayda vardır.

Tablo 16-10: Debug (-T) komutların adım adım çalıştırılması

```
-R
AX=12AF  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2AF2  ES=2AF2  SS=2AF2  CS=2AF2  IP=0100   NV UP EI PL NZ NA PO NC
2AF2:0100 B80500        MOV     AX,0005
-T
AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2AF2  ES=2AF2  SS=2AF2  CS=2AF2  IP=0103   NV UP EI PL NZ NA PO NC
2AF2:0103 050400        ADD     AX,0004
-T
AX=0009  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2AF2  ES=2AF2  SS=2AF2  CS=2AF2  IP=0106   NV UP EI PL NZ NA PE NC
2AF2:0106 D1E8          SHR     AX,1
-T
AX=0004  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=2AF2  ES=2AF2  SS=2AF2  CS=2AF2  IP=0108   NV UP EI PL NZ AC PO CY
2AF2:0108 A24FF6        MOV     [F64F],AL                   DS:F64F=11
-
```

Kullanıcı isterse T komutunun yanına bir sayı da verebilir. Bu durumda verilen sayı kadar komut satırı arka arkaya çalıştırılır, her seferinde yazmaç ve bayrak değerleri ekrana yazılır. Benzer şekilde kullanıcı hangi göreli konum adresinden başlamak kaydıyla kaç satır komut işletileceğini de belirtebilmektedir. Tablo 16-11'de gösterilen ilk kullanım IP yazmacının gösterdiği göreli konum adresinden itibaren 5 komut satırının çalıştırılması, ikinci kullanımda ise 0200H göreli adresinden itibaren 5 komut satırı çalıştırılmasını ifade eder. Her iki durumda da ekranda 5 komut satırı sanki adım adım çalıştırılmış gibi ara adımların sonuçları ile görüntülenecektir.

Tablo 16-11: Debug (-T 5 / -T=0200,5) belirlenen sayıda komutun çalıştırılma

```
- T 5


-T=0200,5
```

16.2.5 U (Unassembly)

U komutu kesim ve göreli konum değerleri verilen adresten itibaren yer alan nesne kodunu kullanıcının anlayacağı formdaki anımsatıcılara çevirerek göstermek için kullanılır. Her çağırılışında **32 byte** uzunluğunda nesne koduna karşılık gelen anımsatıcılar ekranda belirir. Tablo 16-12'da CS:0112 adresinde yer alan nesne kodunun anımsatıcı karşılıkları görülmektedir. Ekranda görünen bilgi, satır olarak incelendiğinde soldan sağa doğru kesim adresi, göreli konum adresi, nesne kodu ve anımsatıcı karşılıkları görülmektedir. Kullanıcı zaman zaman kendi yazdığı programı liste halinde görmek istediğinde de bu komuttan yararlanır.

İstenirse U komutuna parametre olarak verilen kesim, göreli konum ve uzunluk bilgisi ile görüntülenecek anımsatıcı sayısı sınırlandırılabilir. Tablo 16-13'da CS:0100 adresinden itibaren sadece 8 byte uzunluğunda nesne kodunun anımsatıcı karşılığının görüntülenmesi istenmektedir.

Tablo 16-12: Debug (-U CS:0112) nesne kodlarının anımsatıcı karşılıkları

```
-U CS:0112
2AF2:0112 07          POP     ES
2AF2:0113 BA4F80      MOV     DX,804F
2AF2:0116 FC          CLD
2AF2:0117 807463BA    XOR     BYTE PTR [SI+63],BA
2AF2:011B C6418208    MOV     BYTE PTR [BX+DI-7E],08
2AF2:011F 83745BBA    XOR     WORD PTR [SI+5B],-46
2AF2:0123 D208        ROR     BYTE PTR [BX+SI],CL
2AF2:0125 847453      TEST    DH,[SI+53]
2AF2:0128 BADE04      MOV     DX,04DE
2AF2:012B 0908        OR      [BX+SI],CX
2AF2:012D 85744B      TEST    SI,[SI+4B]
2AF2:0130 BAEA08      MOV     DX,08EA
-
```

Tablo 16-13: Debug (-U CS:0100L8) istenen uzunlukta nesne kodun anımsatıcı karşılığı

```
-U CS:0100L8
```

16.2.6 E (Edit / Düzenle)

E komutu bellekte herhangi bir adresteki değerleri değiştirmek amacıyla kullanılan bir komuttur. Tablo 16-14'de DS yazmacının gösterdiği kesim adresinin ilgili göreli konum adresindeki (0010H) byte değiştirilmek istenmektedir. Komutun çalışması ile birlikte mevcut değer ekranda belirecek ve hemen ardından yeni byte değerinin verilmesi için imleç yanıp sönmeye başlayacaktır. Eğer bir seferde birden fazla birbirini takip eden byte'ın değeri değişecek ise ara çubuğuna (space bar) basılarak işlemlerin tekrarlanması sağlanacaktır. İşlemi sonlandırmak üzere ENTER tuşuna basmak yeterli olacaktır.

Tablo 16-14: Debug (-E DS:10) bellek alanında değişiklik

```
-E DS:0010
2AF2:0010  5A.
```

16.2.7 F (Fill / Doldurma)

F komutu yardımıyla başlangıç ve bitiş adresleri belirlenen bir bellek alanının verilen değer ile doldurulması sağlanır. Doldurulmak için kullanılacak değer byte ile ifade edilebilecek bir büyüklükte olmalıdır. Tablo 16-15'de DS yazmacı ile belirlenen kesimin 0100H-0120H göreli konum adreslerindeki byte'lar ABH değeri ile doldurulmuştur. İşlemin sonucu –D DS:0100 ile görüntülenmiştir.

F komutuna parametre olarak verilen kesim, göreli konum ve uzunluk bilgisi ile doldurulacak olan byte sayısı sınırlandırılabilir. Tablo 16-16'da DS:0200 adresinden itibaren sadece 8 byte uzunluğunda ABH değeri nesne kodunun anımsatıcı karşılığının görüntülenmesi istenmektedir.

Tablo 16-15: Debug (-F DS:0100, 0120, AB) bellek alanının doldurulması

```
-F DS:0100,0120,AB
-D DS:0100
2AF2:0100  AB AB AB AB AB AB AB AB-AB AB AB AB AB AB AB AB   ................
2AF2:0110  AB AB AB AB AB AB AB AB-AB AB AB AB AB AB AB AB   ................
2AF2:0120  AB 5B BA D2 08 84 74 53-BA DE 04 09 08 85 74 4B   .[....tS......tK
2AF2:0130  BA EA 08 86 74 43 12 28-BA F6 08 87 74 3B BA 02   ....tC.(....t;..
2AF2:0140  50 08 90 20 88 74 33 BA-0E 08 89 74 2B BA 1A 41   P.. .t3....t+..A
2AF2:0150  82 08 8A 74 23 BA 26 08-8B 74 1B BA 32 54 24 08   ...t#.&..t..2T$.
2AF2:0160  8D A2 BA 3E 08 8E 48 00-74 0B BA 86 08 FF 74 03   ...>..H.t.....t.
2AF2:0170  BA 4A 50 E9 1D A0 C0 C9-BA 56 50 E8 17 75 E8 AB   .JP......VP..u..
-
```

Tablo 16-16: Debug (-F DS:0200L8,AB) bellek alanının doldurulması

```
-F DS:0200L8,AB
-D DS:0200
0C8B:0200  AB AB AB AB AB AB AB AB-19 55 06 E7 19 56 06 57   .........U...V.W
0C8B:0210  1A 68 06 A2 1A 69 06 DA-1A 7C 06 5F 1B 90 06 A5   .h...i...|._....
0C8B:0220  1B A4 06 C4 1B B8 06 84-1C CC 06 CE 1C CD 06 3C   ...............<
0C8B:0230  1D E0 06 C2 1D F4 06 15-1E 08 07 8C 1E 09 07 E0   ................
0C8B:0240  1E 1C 07 34 1F 1D 07 88-1F 30 07 1C 20 44 07 76   ...4.....0.. D.v
0C8B:0250  20 45 07 CE 20 46 07 09-21 47 07 7F 21 48 07 15    E.. F..!G..!H..
0C8B:0260  22 49 07 73 22 4A 07 D9-22 58 07 36 23 59 07 B4   "I.s"J.."X.6#Y..
0C8B:0270  23 5A 07 4D 24 5B 07 A2-24 6C 07 49 25 80 07 5A   #Z.M$[..$l.I%..Z
```

16.2.8 C (Compare / Karşılaştırma)

C komutu yardımıyla bellek üzerinde farklı bölgelerde yer alan belirli sayıda byte'ın karşılaştırılması yapılarak ekranda farklılıklar görüntülenir. Tablo 16-17'de DS yazmacı ile belirlenen kesimin 0100H göreli konumundan 16 byte veri, yine DS yazmacı ile belirlenen kesimin 0110H göreli konumundaki veri ile karşılaştırılmakta ve farklı olan değerler görüntülenmektedir.

Tablo 16-17: Debug (-C 0100, 0110, 10) bellek içeriklerinin karşılaştırılması

```
-C DS:0100,DS:0110,10
2AF2:0100  AB  5A  2AF2:0010
2AF2:0101  AB  25  2AF2:0011
2AF2:0102  AB  17  2AF2:0012
2AF2:0103  AB  03  2AF2:0013
2AF2:0104  AB  5A  2AF2:0014
2AF2:0105  AB  25  2AF2:0015
2AF2:0106  AB  14  2AF2:0016
2AF2:0107  AB  24  2AF2:0017
2AF2:0108  AB  01  2AF2:0018
2AF2:0109  AB  01  2AF2:0019
2AF2:010A  AB  01  2AF2:001A
2AF2:010B  AB  00  2AF2:001B
2AF2:010C  AB  02  2AF2:001C
2AF2:010D  AB  FF  2AF2:001D
2AF2:010E  AB  FF  2AF2:001E
2AF2:010F  AB  FF  2AF2:001F
2AF2:0110  AB  FF  2AF2:0020
```

Ekranda görünen satırlar soldan sağa doğru incelendiğinde ilk bellek bloğunu belirleyen kesim ve göreli konum adresi ve bu bellek alanındaki byte ile -tam ters sırada- diğer bellek içeriği ve bu içeriğin yer aldığı kesim ile göreli konum adresi görülmektedir.

C komutunun sonucunda sadece farklılıklar raporlanacaktır. Tablo 16-18'de her iki bellek bölgesi de aynı adresten başladığından dolayı hiçbir farklılık bulunamayacak ve ekranda ek bir bilgi görünmeyecektir. Bu karşılaştırma sırasında kesim adresi veya bir kesim yazmacı verilmemiş ise bu işlem veri kesimi içinde yapılacaktır.

Tablo 16-18: Debug (-C 0100, 0100, 100) bellek içerik karşılaştırılması

```
-C 0100,0110,100
-
```

C komutunda farklı kesim değerleri veya yazmaçları verilerek de karşılaştırma işlemi yapılabilir. Tablo 16-19'te DS yazmacının belirlediği kesim alanının 0100H-0110H göreli konumu arasında kalan değerleri CS yazmacının belirlediği kesimin 0300H göreli konumundan itibaren karşılaştırılmakta ve farklılıklar görüntülenmektedir.

Tablo 16-19: Debug (-C DS:0100, 0110, CS:0300) bellek içerik karşılaştırılması

```
-C DS:0100,0110,CS:0300
2AF2:0100  AB  25  2AF2:0300
2AF2:0101  AB  44  2AF2:0301
2AF2:0102  AB  68  2AF2:0302
2AF2:0103  AB  20  2AF2:0303
2AF2:0104  AB  DD  2AF2:0304
2AF2:0105  AB  20  2AF2:0305
2AF2:0106  AB  91  2AF2:0306
2AF2:0107  AB  CF  2AF2:0307
2AF2:0108  AB  3A  2AF2:0308
2AF2:0109  AB  1C  2AF2:0309
2AF2:010A  AB  04  2AF2:030A
2AF2:010B  AB  1C  2AF2:030B
2AF2:010C  AB  6B  2AF2:030C
```

16.2.9 G (Go / Git)

G komutu programın verilen bir adrese veya sonlanana kadar çalışmasını sağlamak amacıyla kullanılır. G komutuna parametre olarak göreli konum adresi verilecek olursa[189] program o adrese kadar otomatik olarak çalışacaktır. Parametre verilmeyecek olursa programın sonladığını belirten RET/RETF veya INT komutuna kadar işlemler yürütülecektir.

Tablo 16-20: Debug (-G) programın sonlandırılması

```
-G
Program terminated normally
```

Program eğer düzgün çalıştı ise Tablo 16-20'deki mesaj ekrana gelecektir. Programın dönüş adreslerinde bir hata varsa veya programda bir mantık hatası yapılmış

[189] Göreli konum adresinin bir anımsatıcının başlangıç adresine denk geldiğinden emin olunuz. Bunun için U komutundan yararlanarak göreli konum adresini belirlemek daha uygun olacaktır.

ise program hiçbir zaman bitmez. Bu durumda debug uygulamasının tekrar başlatılması gerekecektir.

16.2.10 H (Hex Calculator / On Altılı Hesaplayıcı)

H komutu zaman zaman ihtiyaç duyulan basit toplama ve çıkarma işlemlerini on altılı tabanda yapmak için kullanılır. Tablo 16-21'de 2E ve 3 değerleri parametre olarak verilmiştir. Elde edilen sonuçtan ilki sayıların toplamını, ikincisi ise farkını göstermektedir.

Tablo 16-21: Debug (-H) on altılı tabanda hesaplayıcı

```
-H2E,3
0031  002B
-
```

16.2.11 L (Load / Yükle)

L komutu, debug programının yanına parametre olarak verilen programın tekrar yüklenmesini sağlar. Böylece debug ortamından çıkmadan işleme baştan başlama imkânı sağlanmış olur.

16.2.12 M (Move / Aktar)

M komutu, adresi verilen bir blok verinin başka bir bellek alanına aktarılması için kullanılır. Tablo 16-22'de DS yazmacı ile belirlenen kesimin, 0100H-0110H göreli konum adresleri arasında kalan değerleri CS yazmacı ile belirlenen kesimin 0500H göreli konumdan başlayarak aktarılmaktadır.

Tablo 16-22: Debug (-M DS:0100, 0110, CS:0500) bellek alanına aktarım

```
- M DS:0100,0110,CS:0500
```

16.2.13 P (Proceed / Devam Etmek)

P komutu genel kullanımı itibari ile T komutuna benzer. Her çalışından sonra yazmaç ve bayrak değerlerini ve bir sonraki turda işletilecek olan assembly kodunu gösterir. Ancak **LOOP**, **CALL**, **INT** gibi komutların işlenmesinde kullanılması durumunda T komutundan farklılığı ortaya çıkar. LOOP, CALL veya INT komutlarından herhangi biri P komutu ile çalıştırılacak olursa o işlemin detaylarına girmeden sonucuna ulaşmak mümkündür. İşletilen komut LOOP ise çevrim sonuçlanana kadar, CALL ise işlemler yerine getirilip geri dönülene kadar veya INT ise kesme kodunun yapması gereken tüm işlemler tamamlanana kadar işlemler yürütülür ve devamında yazmaç ve bayrakların durumu ekranda görüntülenir. Bu özellik kullanıcıların bazı kod bloklarını hızlı bir şekilde analiz edebilmesine imkân sağlar.

16.2.14 Q (Quit / Çıkış)

Her programı sonlandırmak için bir komut olduğu gibi DEBUG programını da sonlandırmak için Q komutu vardır. Bu komutun çalıştırılması ile DEBUG programı başlatıldığı komut satırına geri döner (Tablo 16-23).

Tablo 16-23: Debug (-Q) programın sonlandırılması

```
-Q
C:\>
```

16.3 ASCII Karakter Tablosu

Tablo 16-24: ASCII karakter tablosu

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
000	00H	Null	032	20H	sp	064	40H	@	096	60H	`
001	01H	Start Heading	033	21H	!	065	41H	A	097	61H	a
002	02H	Start Text	034	22H	"	066	42H	B	098	62H	b
003	03H	End Text	035	23H	#	067	43H	C	099	63H	c
004	04H	End Transmit	036	24H	$	068	44H	D	100	64H	d
005	05H	Enquiry	037	25H	%	069	45H	E	101	65H	e
006	06H	Acknowledge	038	26H	&	070	46H	F	102	66H	f
007	07H	Bell	039	27H	'	071	47H	G	103	67H	g
008	08H	Back Space	040	28H	(	072	48H	H	014	68H	h
009	09H	Horizontal Tab	041	29H	)	073	49H	I	105	69H	ı
010	0AH	Line Feed	042	2AH	*	074	4AH	J	106	6AH	j
011	0BH	Vertical Tab	043	2BH	+	075	4BH	K	107	6BH	k
012	0CH	Form Feed	044	2CH	,	076	4CH	L	108	6CH	l
013	0DH	Carriage Return	045	2DH	-	077	4DH	M	109	6DH	m
014	0EH	Shift Out	046	2EH	.	078	4EH	N	110	6EH	n
015	0FH	Shift In	047	2FH	/	079	4FH	O	111	6FH	o
016	10H	Data line Esc	048	30H	0	080	50H	P	112	70H	p
017	11H	Dev Ctl 1	049	31H	1	081	51H	Q	113	71H	q
018	12H	Dev Ctl 2	050	32H	2	082	52H	R	114	72H	r
019	13H	Dev Ctl 3	051	33H	3	083	53H	S	115	73H	s
020	14H	Dev Ctl 4	052	34H	4	084	54H	T	116	74H	t
021	15H	Negatif Ack	053	35H	5	085	55H	U	117	75H	u
022	16H	Synch Idle	054	36H	6	086	56H	V	118	76H	v
023	17H	End Tran. Blk.	055	37H	7	087	57H	W	119	77H	w
024	18H	Cancel	056	38H	8	088	58H	X	120	78H	x
025	19H	End of Medium	057	39H	9	089	59H	Y	121	79H	y
026	1AH	Substitute	058	3AH	:	090	5AH	Z	122	7AH	z
027	1BH	Escape	059	3BH	;	091	5BH	[	123	7BH	{
028	1CH	File Separator	060	3CH	<	092	5CH	\	124	7CH	|
029	1DH	Group Separator	061	3DH	=	093	5DH	]	125	7DH	}
030	1EH	Rec. Separator	062	3EH	>	094	5EH	^	126	7EH	~
031	1FH	Unit Separator	063	3FH	?	095	5FH	_	127	7FH	DEL

16.4 BIOS Kesme Listesi

Tablo 16-25: BIOS kesme listesi

Interrupt	Fonksiyon	Açıklama
00H		Divide by Zero
01H		Single Step
02H		Non Maskable
03H		Breakpoint
04H		Arithmetic Overflow
05H		Print Screen
06H		Reserved
07H		Reserved
08H		System Timer
09H		Keyboard
0AH		Reserved
0BH		Communications
0CH		Communications
0DH		Hard Disk Controller
0EH		Floppy Disk Management
0FH		Printer Management
10H		Video
	00H	Set Video Mode
	01H	Set Cursor Type
	02H	Set Cursor Position
	03H	Read Cursor Position and Configuration
	04H	Real Light Pen Position
	05H	Select Active Display Page
	06H	Scroll Windows Up
	07H	Scroll Windows Down
	08H	Read Character And Attribute
	09H	Write Character And Attribute
	0AH	Write Character At Cursor
	0BH	Set Color Palette
	0CH	Write Graphics Pixel
	0DH	Read Graphics Pixel
	0EH	Write Text in Teletype Mode

	0FH	Get Current Display Mode
	10H	Set Palette Register
	11H	Character Generator
	12H	Alterneate Select
	13H	Write String
	1AH	Read/Write Display Codes
	1BH	Get Display State
	1CH	Save/Restore Display State
11H		Get Equipment Status
12H		Get Memory Size
13H		Diskette
	00H	Reset Disk System
	01H	Get Disk Sistem Status
	02H	Read Disk Sectors
	03H	Write Disk Sectors
	04H	Verify Disk Sectors
	05H	Format Disk Track
	06H	Format Cylinder and Set Bad Sector Flags
	07H	Format Drive From Specified Cylinder
	08H	Return Disk Drive Parameter
	09H	Initialize Fixed Disk Table
	0AH	Read Long Sector
	0BH	Write Long Sector
	0CH	Seek Cylinder
	0DH	Alternate Disk Reset
	0EH	Read Sector Buffer
	0FH	Write Sector Buffer
	10H	Test Fixed Disk System Status
	11H	Recalibrate Fixed Disk Drive
	12H	Diagnose Controller RAM
	13H	Diagnose Drive
	14H	Diagnose Conrtroller
	15H	Return DASD Type
	16H	Read Disk Change Line Status
	17H	Set DASD Type For Disk Format

	18H	Set Media Type For Format
	19H	Park Heads
	1AH	Format ESDI Unit
14H		Asynchronous Communication
	00H	Initialize Communications Port
	01H	Write Character to Communicaitions Port
	02H	Read Character From Communications Port
	03H	Request Communications Port Status
	04H	Extenden Initialization (PS2)
	05H	Extenden Communications Port Control
15H		System Services
	00H	Turn On Casette Motor
	01H	Turn Off Casette Motor
	02H	Read Data Blocks From Casette Drive
	03H	Write Data Blocks To Casette Drive
	0FH	ESDI Unit Format Periodic Interrupt
	21H	Power On Self Test Error Log
	4FH	Keyboard Intercept
	80H	Drive Open
	81H	Drive Close
	82H	Program Termination
	83H	Set/Cancel Wait İnterval
	84H	Joystick Support
	85H	System Request Key Pressed
	86H	Delay
	87H	Move Block
	88H	Extenden Memory Size Determination
	89H	Switch Processor To Protected Mode
	90H	Device Wait
	91H	Interrupt Complete
	C0H	Return System Configuration Parameters
	C1H	Return Extended BISO Data Area Segment Address
	C2H	Pointing Device BIOS Interface
	C3H	Enable/Disable Watchdog Timeout
	C4H	Programmable Option Select

16H		Keyboard
	00H	Read Keyboard Character
	01H	Read Keyboard Status
	02H	Return Keyboard Flags
	03H	Adjust Keyboard Repeat Rate
	04H	Key-Click On/Off
	05H	Write to Keyboard Buffer
	10H	Get Keystroke
	11H	Get Keyboard
	12H	Get Keyboard Status Flags
17H		Printer
	00H	Write Character To Printer
	01H	Initialize Printer Port
	02H	Request Printer Port Status
18H		Execute ROM BASIC
19H		System Warm Boot
1AH		System Timer / Real-Time Clock Services
	00H	Get Clock Counter
	01H	Set Clock Counter
	02H	Read Real-Time Clock
	03H	Set Real-Time Clock
	04H	Read Date of Real-Time Clock
	05H	Set Date of Real-Time Clock
	06H	Set System Alarm
	07H	Disable Real-Time Clock Alarm
	09H	Read Real-Time Clock Alarm
	0AH	Get Day Count
	0BH	Set Day Count
	80H	Set Sound Service
1BH		Ctrl-Break Address
1CH		Timer Tick Interrupt
1DH		Video Initialization Parameter Table
1EH		Disk Initialization Parameter Table
1FH		Graphics Display Character Bit-Map Table
70H		Real-Time Clock Interrupt

16.5 DOS Kesme Listesi

Tablo 16-26: DOS kesme listesi

Interrupt	Fonksiyon	Açıklama
20H		Terminate Program
21H	00H	Terminate Program
	01H	Read character from standard input with echo
	02H	Write character to standard output
	03H	Read character from STDAUX
	04H	Write character to STDAUX
	05H	Write character to printer
	06H	Direct console output
	07H	Direct character input without echo
	08H	Character output without echo
	09H	Write string to standard output
	0AH	Buffered input
	0BH	Get STDIN status
	0CH	Flush buffer and read standard input
	0DH	Disk reset
	0EH	Select default drive
	0FH	Open file using FCB
	10H	Close file using FCB
	11H	Find first matching file using FCB
	12H	Find next matching file using FCB
	13H	Delete file using FCB
	14H	Sequential read from FCB file
	15H	Sequential write to FCB file
	16H	Create or truncate file using FCB
	17H	Rename file using FCB
	18H	Null function for CP/M compatibility
	19H	Get current default drive
	1AH	Set disk transfer area address
	1BH	Get allocation information for default drive
	1CH	Get allocation information on specific drive
	1DH	Null function for CP/M compatibility
	1EH	Null function for CP/M compatibility

	1FH	Get drive parameter block for default drive
	20H	Null function for CP/M compatibility
	21H	Read random record from FCB file
	22H	Write random record to FCB file
	23H	Get file size for FCB
	24H	Set random record number for FCB
	25H	Set interrupt vector
	26H	Create new program kesim prefix
	27H	Random block read from FCB file
	28H	Random Block write to FCB file
	29H	Parse filename into FCB
	2AH	Get system date
	2BH	Set system date
	2CH	Get system time
	2DH	Set system time
	2EH	Set verify flag
	2FH	Getdisk transfer area address
	30H	Get DOS version
	31H	Terminate and stay resident
	32H	Get DOS drive parameter block for specific drive
	33H	Extended break checking
	34H	Get address of INDOS flag
	35H	Get interrupt vector
	36H	Get free disk space
	37H	Switch char
	38H	Get country specific information
	39H	Create subdirectory
	3AH	Remove subdirectory
	3BH	Set current directory
	3CH	Create or truncate file
	3DH	Open existing file
	3EH	Close file
	3FH	Read from file or device
	40H	Write to file or device
	41H	Delete file
	42H	Set current file position

	43H	Get/Set file attributes
	44H	IO Controle
	45H	Duplicate file handle
	46H	Force duplicate file handle
	47H	Get current directory
	48H	Allocate memory
	49H	Free memory
	4AH	Resize memory block
	4BH	Load and/or Execute program
	4CH	Terminate with return code
	4DH	Get return code
	4EH	Find first matching file
	4FH	Find next matching file
	50H	Set current process id
	51H	Get current process id
	52H	Get lists of list
	53H	Translate BIOS parameter block to drive parameter block
	54H	Get verify flag
	55H	Create child PSP
	56H	Rename file
	57H	Get/Set file date and time
	58H	Get/Set memory allocation strategy
	59 H	Get extended error information
	5AH	Create temporary file
	5BH	Create newfile
	5CH	Record locking
	5DH	Server function call
	5EH	Get/Set machine name - network related
	5FH	Redirection
	60H	Canonicalize filename or path
	61H	Unused
	62H	Get current PSP address
	63H	Country specific
	64H	Set device driver lockahead flag
	65H	Get extended country information
	66H	Get/Set global code page

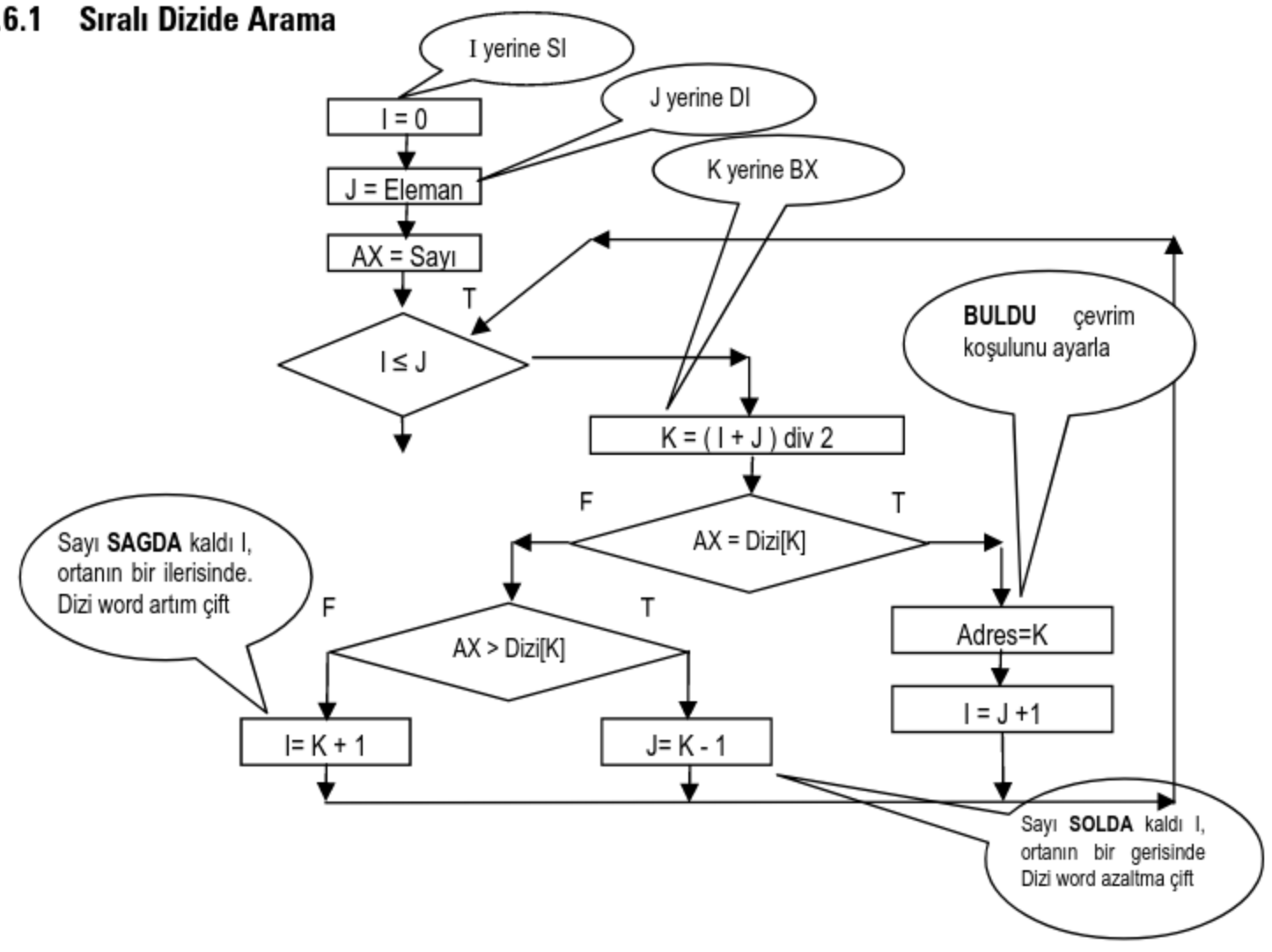
	67H	Set handle count
	68H	Commit file
	69H	Get/Set disk serial number
	6AH	Unknown
	6BH	Unknown
	6CH	Extended open/create
	80H	Execute program in background
	87H	Get PID
	89H	Unknown
22H		Program termination address
23H		Control-C / Control Break handler
24H		Critical error handler
25H		Absolute disk read
26H		Absolute disk write
27H		Terminate and stay resident
28H		DOS idle interrupt
29H		Fast console output
2BH		Reserved
2CH		Reserved
2DH		Reserved
2EH		Pass command to command interpreter for execution
2FH		Multiplex interrupt
30H		Far JMP instruction for CP/M style calls
6CH		Realtime clock update

16.6 Ek Örnek Kodlar

Bu bölümde gerek COM gerek ise EXE tipinde yazılmış assembly program örnekleri verilmiştir. Assembly öğrenmek için bu örneklerde de olduğu gibi sık karşılaşılan algoritmalar assembly dilinde kodlanarak da deneyim kazanılabilir.

Hangi programlama dili olursa olsun, program yazmadan önce algoritmanın akış diyagramı teknikleri kullanılarak çizilmesi ve uç nokta problemlerinin titiz bir şekilde analiz edilmesi gereklidir. Bu çalışma ile tasarlanan algoritmanın doğru olduğuna emin olduktan sonra kodlama aşamasına geçilebilir. Ancak assembly kodlamaya başlamadan önce hangi yazmaçların akış diyagramındaki hangi değişkenin (sembolün) yerine kullanılacağının belirlenmesi (özellikle dizi türündeki veri yapıları ile çalışılması durumunda indis olarak kullanılacak olan yazmaçların değerlerinin ne şekilde değişeceğine dikkat edilmesi) ve koşul ifadelerinde dallanma için kullanılacak olan etiketlerin akış diyagramı üzerine yazılması kodlama aşamasında programcıya büyük kolaylık sağlayarak, olası hataların daha çabuk bulunmasına ve programın rahat takip edilebilmesine imkân verecektir. Biz de sadece ilk örneğimizi bu çalışma yöntemine uygun olarak, daha sonrakileri ise oldukça ayrıntılı açıklama satırları bulunan kod parçaları olarak vereceğiz.

16.6.1 Sıralı Dizide Arama



Şekil 16-1 Sıralı dizide arama akış diyagramı

```
; ---------------------------------------------------------------------
; 10 word elemanlı işaretli sayılardan olan sıralı bir dizide verilen
; X (word) sayısının hangi adreste olduğunu bulan COM program
; Not: Dizi sıralı olduğu için yarıya bölme ile aranan elemanın en az
; karşılaştırma ile bulunması sağlanacak, X sayısının dizi içinde olmaması
; durumunu göstermek üzere adres değeri olarak FFFFh kullanılacaktır.
; ---------------------------------------------------------------------
MYSEG      SEGMENT PARA 'CODE'
           ORG 100H
           ASSUME CS:MYSEG,SS:MYSEG, DS:MYSEG
BILGI:     JMP MAIN
; ---------------------------------------------------------------------
; Kullanılan değişkenlerin tanımlanması
; ---------------------------------------------------------------------
DIZI       DW -21,-8, -3, -1, 0 , 4, 14, 21, 45, 67
ELEMAN     DW 10
SAYI       DW 45
ADRES      DW 0FFFFH     ; Aranan sayının dizide olmadığı var sayılarak
                         ; adres değişkeni 0FFFFH yapılmıştır.
MAIN       PROC NEAR
           XOR SI,SI     ; dizinin ilk elemanının göreli adresi
           MOV DI,ELEMAN ; Word tanımlı dizinin son elemanını takip eden
                         ; bellek gözünün adresi 2 * eleman kadar uzakta
                         ; olacaktır. Dizi 0. adresten başladığı için
           SHL DI,1      ; son eleman da 2 * elaman -2 adresinde yer alır.
           SUB DI,2      ; dizinin son elemanının göreli adresi
           MOV AX,SAYI   ; aranacak olan sayı AX'e konur.
KONT:      CMP SI,DI     ; dizinin ilk elemanının göreli adresi, son elemanın
           JG SON        ; göreli adresinden büyük ise işlem sonlanır
           MOV BX,SI     ; dizinin ortasına denk gelen adresin bulunması için
           ADD BX,DI     ; baş ve son adres toplanıp 2'ye bölünüyor.
           SHR BX,1
           CMP AX,DIZI[BX]; ortadaki elemanı ile aranan sayıyı karşılaşır.
           JE BULDU      ; aranan sayı bulundu
           JG SAGDA      ; AX büyük ise,sayı dizinin ortası ile sonu arasında
           MOV DI,BX     ; değil ise başı ile ortası arasında olacaktır.
           SUB DI,2      ; DI←BX yaparak dizinin ortasını sonu kabul et
                         ; karşılaştırmayı zaten orta eleman ile yaptığımız
                         ; için ondan önceki elamanı kullanacağız
           JMP KONT
SAGDA:     MOV SI,BX     ; SI←BX yaparak dizinin başını ortası kabul et
           ADD SI,2      ; karşılaştırmayı zaten orta eleman ile yaptığımız
                         ; için ondan sonraki elamanı kullanacağız
           JMP KONT
BULDU:     MOV ADRES, BX ; BX yazmacında saklanan adres değerini belleğe koy
           MOV SI,DI     ; kont isimli çevrinden (while) düzgün çıkmak için
                         ; koşul ifadesini çevrinden çıkacak şekilde ayarla
           ADD SI,2      ; Word dizi olduğu için indisi 2 arttır
           JMP KONT      ; çevrim koşulunu kontrol etmek üzere dallan
SON:       RET
MAIN       ENDP
MYSEG      ENDS
           END BILGI
```

16.6.2 Dizi İşlemleri

```
; ---------------------------------------------------------------------
; Bir hastanın 10 gün boyunca ölçülmüş büyük ve küçük tansiyon değerleri
; verilmiş olsun. Büyük ile küçük tansiyon değerleri arasındaki farkın en
; fazla olduğu günü bulan ve GUN isimli değişkende saklayan program
; Büyük Tansiyon > Küçük Tansiyon olacaktır.
; BYK : Büyük tansiyon değerleri
; KCK : Küçük tansiyon değerleri
; GUN : Farkın en fazla olduğu gün
; ---------------------------------------------------------------------
MYSTACK    SEGMENT PARA STACK 'STACK'
           DW 32 DUP (?)
MYSTACK    ENDS

MYDATA     SEGMENT PARA 'DATA'
BYK        DB 12,13,14,15,15,16,13,12,10,9
KCK        DB 5 , 6, 7, 8, 9, 3, 2, 5, 5,6
GUN        DW 0
ELEMAN     DW 10
MYDATA     ENDS

MYCODE     SEGMENT PARA 'CODE'
           ASSUME CS:MYCODE, SS:MYSTACK, DS:MYDATA
MAIN       PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
           PUSH DS
           XOR AX,AX
           PUSH AX
;---------------------------------------------------------------------
; MYDATA ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
           MOV AX,MYDATA
           MOV DS,AX
           MOV CX,ELEMAN      ; tekrar sayısı CX yazmacına konur
           XOR SI,SI          ; dizi indisi olacak
           XOR BX,BX          ; tansiyon arasındaki en büyük farkı tutar
L1:        MOV AL,BYK[SI]     ; Büyük tansiyon değeri AL'de
           SUB AL,KCK[SI]     ; Küçük tasniyon değeri BL'de
           CMP AL, BL
           JB SONRAKI         ; sayılar işaretsiz
           MOV BL,AL          ; fark daha büyük ise BL'ye koy
           MOV GUN,SI         ; indisi GUN olarak sakla
SONRAKI:   INC SI             ; dizi byte, artım tek olmalı
           LOOP L1
           RET
MAIN       ENDP
MYCODE     ENDS
           END MAIN
```

16.6.3 Dizi Elemanları Üzerinden Bit İşlemleri

```
; ---------------------------------------------------------------------
; 10 adet word tipinde elemanı olan bir dizi elemanlarının binary (ikili
; sayı sisteminde) gösteriminde en çok 1 barındıran elemanın adresini
; bulan ve bunu bellek üzerindeki bir değişkende saklayan COM program
; ---------------------------------------------------------------------
MYCODESEGMENT PARA 'CODE'
      ORG 100H
      ASSUME CS:MYCODE, SS:MYCODE, DS:MYCODE
MAIN  PROC NEAR
      XOR DI,DI     ; DI en  fazla 1 barındıran elemanın indisini tutar
      XOR SI,SI     ; SI işlenecek elemanın adresini tutar.
      XOR DX,DX     : DL elemanın ikili gösterimdeki 1 sayısını tutar
                    ; DH ise en fazla 1'in kaç defa geçtiğini tutar
      MOV CX,ELEMAN
L2:   MOV AX,DIZI[SI]
      PUSH CX       ; Dış cevrimin tekrar sayısı bozulmasın diye saklanır
      MOV CX,16     ; dizi word olduğu için öteleme işlemi 16 defa tekrarlı
      XOR DL,DL     ; 1'leri saymak için sayısı sıfırla
L1:   SHR AX,1
      ADC DL,0      ; öteleme sonucunda CF'deki değerleri topla
      LOOP L1       ; eleman içindeki 1'lerin sayısı DL'de bulundu
      CMP DH,DL     ; DH ≥ DL ise değişen bir şey yok
      JGE NEXT
      MOV DH,DL     ; DL, şu ana kadar bulunmuşlardan daha büyük
      MOV DI,SI     ; indisi ayarla
NEXT: POP CX        ; Dış çevrimin saklanan değerini yığından al
      ADD SI,2      ; dizi word olduğu için artım çift
      LOOP L2
      MOV YER,DI    ; sonucu belleğe yaz
      RET
MAIN  ENDP
ELEMANDW 10
DIZI  DW 88H,45H,71H,77H,63H,0AAH,56H,12H,92H,0BFH
YER   DW ?
MYCODEENDS
      END MAIN
```

16.6.4 Karekök Hesabı – SQROOT (Harici Yordam)

```
; ---------------------------------------------------------------------
; Verilen pozitif bir tam sayının karekökünü hesaplamak için sayıdan
; sırası ile 1,3,5.. gibi tek sayılar çıkartılarak sonuç sıfırdan küçük
; çıkana kadar işleme devam edilir. Gerçekleştirilen işlem sayısı verilen
; sayının karekök değerini (tam sayı olarak) verir.
; Bu program yığın üzerinden 1 word olarak aldığı parametrenin karekök
; değerini AX yazmacı üzerinden döndürür. Yığındaki parametreyi kaldırır.
; ---------------------------------------------------------------------
          PUBLIC SQROOT ; Farklı programların kullanmasına izin ver
MYCODE    SEGMENT PARA 'CODE'
          ASSUME CS:MYCODE
SQROOT    PROC FAR
; ---------------------------------------------------------------------
; İşlem sırasında kullanacağımız yazmaçların daha önce sahip olduğu
; değerleri kayıp etmemesi için yığında saklamalıyız. Benzer şekilde yığın
; kullanarak parametre aktaracağımız için BP'in de saklanması gerekir.
; ---------------------------------------------------------------------
          PUSH DX
          PUSH BX
          PUSH BP
          MOV BP,SP     ; yığın üzerinden yollanan parametreye BP ile eriş
          MOV DX,1      ; Çıkarılacak tek sayılar DX'de olacak
          XOR AX,AX     ; kaç tane çıkarma işlemi yapıldığı AX'de tutulacak
          MOV BX,[BP+10]; parametreyi BX yazmacına alıyoruz
          SUB BX,DX
L1:       JL SON        ; çıkarma işleminin sonucu negatif ise bitir
          INC AX        ; İşlem sayısı için AX'i arttır
          ADD DX,2      ; Bir sonraki tek sayısı hesapla
          SUB BX,DX     ; çıkarma işlemini yap
          JMP L1        ; kontrol etmek için Dallan
SON:      POP BP        ; Yığında saklananları ters sırada geri al
          POP BX
          POP DX
          RET 2         ; yığındaki parametreyi al ve bitir.
SQROOT    ENDP
MYCODE    ENDS
          END           ; kendi başına çalışması beklenen bir program
                        ; olmadığı için Başlama noktası yok
```

16.6.5 Harici Karekök (SQROOT) Yordamını Kullanmak

```
; ---------------------------------------------------------------------
; SQROOT isimli harici yordamı kullanarak, 5 word elemanlı dizinin
; elemanlarının karekök değerlerini hesaplayıp yeni bir diziye yerleştirme
; ---------------------------------------------------------------------
          EXTRN SQROOT:FAR  ; Kullanılacak olan harici yordamı tanımla
MYSS      SEGMENT PARA STACK 'STACK'
          DW 12 DUP (?)
MYSS      ENDS
MYDS      SEGMENT PARA 'DATA'
DIZI      DW 16,2,4,8,9
KOK       DW 5 DUP (?)
ELEMAN    DW 5
MYDS      ENDS
MYCD      SEGMENT PARA 'CODE'
          ASSUME CS:MYCD, SS:MYSS, DS:MYDS
MAIN      PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
          PUSH DS
          XOR AX,AX
          PUSH AX
;---------------------------------------------------------------------
; MYDS ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
          MOV AX, MYDS
          MOV DS,AX
          MOV CX,ELEMAN     ; Çevrim sayısını ayarla
          XOR SI,SI         ; Her iki dizi içinde SI indis olacak
L1:       PUSH DIZI[SI]     ; parametreyi yığına koy/çağırılan kaldıracak
          CALL SQROOT       ; yordamı çağır
          MOV KOK[SI],AX    ; karekök değerini yeni diziye koy
          ADD SI,2          ; diziler word bir sonraki eleman iki ileride
          LOOP L1
          RET
MAİN      ENDP
MYCD      ENDS
          END  MAIN
```

16.6.6 Klavyeden Okuma, Hesaplama ve Ekrana Yazdırma İşlemleri

```
; ---------------------------------------------------------------------
; Okunan işaretli iki sayının toplamını hesaplayıp ekrana yazdırır.
; ANA      : Ana yordam
; PUT_STR  : Ekrana sonu 0 ile belirlenmiş dizgeyi yazdırır.
; PUTC     : AL deki karakteri ekrana yazdırır.
; GETC     :Klavyeden basılan karakteri AL'ye alır.
; PUTN     : AX'deki sayeyi ekrana yazdırır.
; GETN     : Klavyeden okunan sayeyi AX'e koyar
; ---------------------------------------------------------------------
SSEG  SEGMENT PARA STACK 'STACK'
      DW 32 DUP (?)
SSEG  ENDS
DSEG  SEGMENT PARA 'DATA'
CR    EQU 13
LF    EQU 10
MSG1  DB 'ilk sayiyi veriniz: ',0
MSG2  DB CR, LF, 'ikinci sayiyi veriniz: ', 0
HATA  DB CR, LF, 'Dikkat !!! Sayi vermediniz yeniden giriş yapiniz.!!!  ', 0
SONUC DB CR, LF, 'Toplam ', 0
SAYI1 DW ?
SAYI2 DW ?
DSEG  ENDS
CSEG  SEGMENT PARA 'CODE'
      ASSUME CS:CSEG, DS:DSEG, SS:SSEG
ANA   PROC FAR
;---------------------------------------------------------------------
; Dönüş için gerekli olan değerler yığında saklanıyor
;---------------------------------------------------------------------
      PUSH DS
      XOR AX,AX
      PUSH AX
;---------------------------------------------------------------------
; DSEG ismiyle tanımlı kesim alanına erişebilmek için gerekli tanımlar
;---------------------------------------------------------------------
      MOV AX, DSEG
      MOV DS, AX
      MOV AX, OFFSET MSG1
      CALL PUT_STR          ; MSG1'i göster
      CALL GETN             ; SAYI1'i oku
      MOV SAYI1, AX
      MOV AX, OFFSET MSG2
      CALL PUT_STR          ; MSG2'i göster
      CALL GETN             ; SAYI2'i oku
      MOV SAYI2, AX
      MOV AX, OFFSET SONUC
      CALL PUT_STR          ; Sonucu göster
      MOV AX, SAYI1         ; AX = SAYI1
      ADD AX, SAYI2         ; AX = AX + SAYI2
      CALL PUTN             ; Toplam değerini yaz
      RETF
ANA   ENDP

GETC  PROC NEAR
;---------------------------------------------------------------------
; Klavyeden basılan karakteri AL yazmacına alır ve ekranda gösterir.
; işlem sonucunda sadece AL etkilenir.
;---------------------------------------------------------------------
      MOV AH, 1h
      INT 21H
      RET
GETC  ENDP
```

```
PUTC   PROC NEAR
;-----------------------------------------------------------------------
; AL yazmacındaki değeri ekranda gösterir. DL ve AH değişiyor. AX ve DX
; yazmaçlarının değerleri korumak için PUSH/POP yapılır.
;-----------------------------------------------------------------------
       PUSH AX
       PUSH DX
       MOV DL, AL
       MOV AH,2
       INT 21H
       POP DX
       POP AX
       RET
PUTC   ENDP
GETN   PROC NEAR
;-----------------------------------------------------------------------
; Klavyeden basılan sayiyi okur, sonucu AX yazmacı üzerinden dondurur.
; DX: sayının işaretli olup/olmadığını belirler. 1 (+), -1 (-) demek
; BL: hane bilgisini tutar
; CX: okunan sayının islenmesi sırasındaki ara değeri tutar.
; AL: klavyeden okunan karakteri tutar (ASCII)
; AX zaten dönüş değeri olarak değişmek durumundadır. Ancak diğer
; yazmaçların önceki değerleri korunmalıdır.
;-----------------------------------------------------------------------
       PUSH BX
       PUSH CX
       PUSH DX
GETN_START:
       MOV DX, 1     ; sayının şimdilik + olduğunu varsayalım
       XOR BX, BX    ; okuma yapmadı Hane 0 olur.
       XOR CX,CX     ; ara toplam değeri de 0 dır.
NEW:
       CALL GETC     ; klavyeden ilk değeri AL'ye oku.
       CMP AL,CR
       JE FIN_READ   ; Enter tuşuna basilmiş ise okuma biter
       CMP  AL, '-'  ; AL ,'-' mi geldi ?
       JNE  CTRL_NUM ; gelen 0-9 arasında bir sayı mı?
NEGATIVE:
       MOV DX, -1    ; - basıldı ise sayı negatif, DX=-1 olur
       JMP NEW       ; yeni haneyi al
CTRL_NUM:
       CMP AL, '0'   ; sayının 0-9 arasında olduğunu kontrol et.
       JB error
       CMP AL, '9'
       JA error      ; değil ise HATA mesajı verilecek
       SUB AL,'0'    ; rakam alındı, haneyi toplama dâhil et
       MOV BL, AL    ; BL'ye okunan haneyi koy
       MOV AX, 10    ; Haneyi eklerken *10 yapılacak
       PUSH DX       ; MUL komutu DX'i bozar işaret için saklanmalı
       MUL CX        ; DX:AX = AX * CX
       POP DX        ; işareti geri al
       MOV CX, AX    ; CX deki ara değer *10 yapıldı
       ADD CX, BX    ; okunan haneyi ara değere ekle
       JMP NEW       ; klavyeden yeni basılan değeri al
ERROR:
       MOV AX, OFFSET HATA
       CALL PUT_STR  ; HATA mesajını göster
       JMP GETN_START ; o ana kadar okunanları unut yeniden sayı almaya başla
FIN_READ:
       MOV AX, CX    ; sonuç AX üzerinden dönecek
       CMP DX, 1     ; İşarete göre sayıyı ayarlamak lazım
       JE FIN_GETN
       NEG AX        ; AX = -AX
FIN_GETN:
       POP DX
       POP CX
```

```
      POP DX
      RET
GETN  ENDP
PUTN  PROC NEAR
;-----------------------------------------------------------------------
; AX de bulunan sayiyi onluk tabanda hane hane yazdırır.
; CX: haneleri 10'a bölerek bulacağız, CX=10 olacak
; DX: 32 bölmede işleme dâhil olacak. Soncu etkilemesin diye 0 olmalı
;-----------------------------------------------------------------------
      PUSH CX
      PUSH DX
      XOR DX,  DX   ; DX 32 bit bölmede soncu etkilemesin diye 0 olmalı
      PUSH DX       ; haneleri ASCII karakter olarak yığında saklayacağız
                    ; kaç haneyi alacağımızı bilmediğimiz için yığına 0
                    ; değeri koyup onu alana kadar devam edecelim
      MOV CX, 10    ; CX = 10
      CMP AX, 0
      JGE CALC_DIGITS
      NEG AX        ; sayı negatif ise AX pozitif yapılır.
      PUSH AX       ; AX sakla
      MOV AL, '-'   ; işareti ekrana yazdır.
      CALL PUTC
      POP AX        ; AX'i geri al
CALC_DIGITS:
      DIV CX        ; DX:AX = AX/CX  AX = bölüm DX = kalan
      ADD DX, '0'   ; kalan değerini ASCII olarak bul
      PUSH DX       ; yığına sakla
      XOR DX,DX     ; DX = 0
      CMP AX, 0     ; bölen 0 kaldı ise sayının işlenmesi bitti demek
      JNE CALC_DIGITS   ; işlemi tekrarla
DISP_LOOP:
                    ; yazılacak tüm haneler yığında. En anlamlı hane üstte
                    ; en az anlamlı hane en alta ve onu altında da
                    ; sona vardığımızı anlamak için konan 0 değeri var.
      POP AX        ; sırayla değerleri yığından alalım
      CMP AX, 0     ; AX=0 olursa sona geldik demek
      JE END_DISP_LOOP
      CALL PUTC     ; AL deki ASCII değeri yaz
      JMP DISP_LOOP     ; işleme devam
END_DISP_LOOP:
      POP DX
      POP CX
      RET
PUTN  ENDP
PUT_STR   PROC NEAR
;-----------------------------------------------------------------------
; AX de adresi verilen sonunda 0 olan dizgeyi karakter karakter yazdırır.
; BX dizgeye indis olarak kullanılır. Önceki değeri saklanmalıdır.
;-----------------------------------------------------------------------
      PUSH BX
      MOV BX,  AX           ; Adresi BX'e al
      MOV AL, BYTE PTR [BX] ; AL' de ilk karakter var
PUT_LOOP:
      CMP AL,0
      JE  PUT_FIN           ; 0 geldi ise dizge sona erdi demek
      CALL PUTC             ; AL 'deki karakteri ekrana yazar
      INC BX                ; bir sonraki karaktere geç
      MOV AL, BYTE PTR [BX]
      JMP PUT_LOOP          ; yazdırmaya devam
PUT_FIN:
      POP BX
      RET
PUT_STR   ENDP

CSEG  ENDS
      END ANA
```

Kaynaklar

- 16 Bit Microprocessor Handbook, Osborn /Mc Graw Hill
- 3 Volt System Logic for Personal Computers Data Book, AMD
- 64 bit Computing, John R. Mashey, Byte, Sept/1991
- A History of Microprocessors http://www.intel.com/intel/intelis/museum/online/hist_micro/index.htm
- Advanced Assembly Language on The IBM PC, Holzner, S., Prentice Hall, ISBN 0–89303–633–1
- Advanced MS-DOS, Duncan, R., Microsoft Press, ISBN 0–914845–77–2
- AM386 Microprocessors for Personal Computers Data Book, AMD
- AMD vs. Superman, Halfhill, T.R., Byte, Nov/1994
- Assembly Language Step-by-Step: Programming with DOS and Linux, Second Edition, Jeff Duntemann, ISBN: 0–471–375233
- Assembly Language Tools & Techniques for the IBM Microcomputers, Sanchez, J., Prentice Hall, ISBN 0–13–0504493–9
- Bilgisayar Dünyası Süper Hızlı P6 ya Hazır mı? BT/Haber 24–30 Nisan 1995 Sayı: 14
- Bilişim Sözlüğü – 2005, Bülent Sankur, Pusula Yayıncılık
- Bilişim Terimleri Sözlüğü, Türkiye Bilişim Derneği, Mayıs–1996
- Cover Story, Halfhill, T.R., Byte, JUNE/1994
- CPU World http://www.cpu-world.com/CPUs/index.html
- DOS Programmers Reference, Dettmann, H., Que Corp., ISBN 0–88022–458–4
- http://www.bixnet.com/sddddddddramc.html
- IBM PC Assembler Language and Programming, Abel, P., Prentice Hall, ISBN 0–13–449067–3
- Intel® 64 and IA-32 Architectures Software Developer's Manual, Order Number: 253665-042US, March 2012
- Korumalı kip nedir?, http://my.execpc.com/~geezer/os/pm.htm
- Microcomputer Systems, The 8086/8088 Family, Liu, Y.C., Gibson, G.A., Prentice-Hall, ISBN 0–13–580944–4

- Microprocessors VOL I /II Intel, ISBN 1–55512–150–0
- Microsoft MS-DOS Programmer's Reference Microsoft
- Microsoft’s 80386/80486 Programming Guide, Nelson, R.P., Microsoft Press, ISBN 1–55615–343–0
- NSTL Lab. Report: Notebook CPU’s, Byte, Oct/1993
- PC Assembly Language, Carter, P.A., http://www.drpaulcarter.com/pcasm
- PC Intern, Tischer, M. ABACUS, ISBN 1–55755–145–6
- PS/2-PC Assembly Language, Holzner, S., Prentice Hall, ISBN 0–13–731878–2
- Smart Cache Family Performance Brief, Intel Corp., Order # 297018–001
- Supercharging C with Assembly Language, Chesley, H.R., Waite, M., Addision Wesley, ISBN 0–201–18349–8
- T5, Brute Force, Halfhill, T.R., Byte, Nov /1994
- The 80386, 80486 and Pentium Processor, Hardware, Software and Interfacing, Triebel, W.A., Prentice Hall, ISBN 0–13–533225–7
- The AM386-AM486 CPU Family PCs and Portables, AMD
- The Am486 CPU Family for PCs and Portables, AMD
- The Art of Assembly Language, Hyde, R., 1996, http://wwebster.cs.ucr.edu/
- The Cyrix 386 to 486 Upgrade Microprocessor Technical Biref, Cyrix
- The Intel 386 Family – An Architectural Solution, Intel Corp, Order # 240231–001
- Undocumented DOS, Schulman, A., Michels, A., Kyle, J., Paterson, T., Maxey, D., Addision-Wesley ISBN 0–201–57064–5
- Upgrading and Repairing PCs, 20th Edition, Scott Mueller, Pearson Education, ISBN-13: 978-0-7897-4710-3
- Wikipedia Free Encyclopedia http://wikipedia.org/

Türkçe İngilizce Terimler Dizini

Anabilgisayar : Mainframe
Anımsatıcı : Mnemonic
Ara çubuğu : Space bar
Arama : Search
Ardışık düzen : Pipeline
Aygıt sürücüsü : Device driver
Ayrıcalık : Privilege
Ayrık : Discrete
Bacak : Pin
Bağlantı yuvası : Socket
Başarım : Performance
Başvuru : Reference
Bayrak : Flag
Bellek : Memory
Biçim katsayısı : Form factor
Bilgisayar : Computer
Birlemek : Set
Birleştirme : Combine
Bütünleşik : Integrated
Çalışırken değiştirme : Hot swap
Çapraz Derleyici : Cross Compiler
Çentik : Notche
Çift eşilik : Even parity
Çizge : Graph
Çok basit : Primitive
Çok görevlilik : Multitasking
Çok işlem : Multiprocessing
Çok işlemcili : Multiprocessor
Çok izlekli/kullanımlı : Multithreading
Çoklu ortam : Multimedia
Çoklu programlama : Multiprogramming

Derleme : Compile
Derleyici : Compiler
Devre dışı bırakma : By pass
Devre takımı : Chipset
Dizgi : String
Dizin : Path
Donanım : Hardware
Dosya : File
Döndürme : Rotate
Dörtlü : Nibble
Duyarlılık : Precision
En baştan başlamak : Cold start
En iyi : Optimal
Eş zamanlı : Synchronous
Eş zamansız : Asynchronous
Eşlik : Parity
Etiket : Label
Etkileşimli : Interactive
Geçici : Transient
Geleneksel : Conventional
Genişletilmiş : Extended
Genişletme yuvası : Expention slot
Gerçek zamanlı : Real time
Geriye dönük : Backtracking
Geriye doğru uyumluluk : Backward compatibility
Gevşek bağlı : Loosely coupled
Göbek : Hub
Göreli konum : Offset / Effective Address
Hata ayıklama : Debug
Hata : Error
Her yerden erişilebilen : Global
Hız : Cycle rate
Hizalama : Alignment

İkili : Binary
İlişkilendirme : Link
İmleç : Cursor
İskele : Port
İş : Job
İşlenen : Operand
İşletim sistemi : Operating system
İşlev : Function
İz : Track
Kafa : Head
Kapı : Gate
Kaydırma : Shift
Kaynak kapma yarışı : Race condition
Kesim : Segment
Kesme : Interrupt
Kişisel bilgisayar : Personal Computer
Komut alma : Fetch
Komut çözme : Decode
Komut işleme : Execute
Komut satırı : Command prompt
Makine kodu : Machine code
Makro işlemci : Macro processor
Mantis : Mantisa
Meşgul bekleme : Busy waiting
Nesne kodu : Object code
On altılı : Hexadecimal
Ölçeklenebilirlik : Scaleability
Ön bellek : Cache memory
Ön ek : Prefix
Önyükleme tutanağı : Boot record
Örneksel : Analog
Özgür belgeleme lisansı : General Public License (GPL)
Özyineli : Recursive

Papatya zinciri : Daisy chain
Rasgele : Random
Saat darbesi : Clock cycle
Sanal bellek : Virtual memory
Sanallaştırma : Virtualization
Sayfalı bellek kullanımı : Paging
Sayısal : Digital
Seçimli : Optional
Sekizli : Octal
Sıcak başlama : Warm start
Sıfırlamak : Reset
Sıkı bağlı : Tightly coupled
Sınıf : Class
Sınır : Boundary
Sözde : Pseudo
Sürücü : Drive
Sürüm : Version
Şekillendirme : Configuration
Tampon : Buffer
Tek eşlik : Odd parity
Toplu işleme : Batch processing
Tümleşik : Integrated
Uçucu : Volitile
Uyarı : Warning
Üst : Exponent
Varsayılan : Default
Veri : Data
Yazılım : Software
Yazım kuralları : Syntax
Yazmaç : Register
Yerel : Local
Yerleşik : Resident
Yol : Bus

Yordam : Procedure
Yorumlayıcı : Interpreter
Yükleyici : Loader
Yüz : Side
Zaman dilimi : Time slice
Zaman paylaşımı : Time sharing
Zamanlayıcı : Scheduler