

# ALT SEVİYE PROGRAMLAMA

Hafta 1

Dr. Öğr. Üyesi Erkan USLU

# MİKROİŞLEMCİLERİN TARİHSEL GELİŞİMİ

# Mikroişlemci Çağı

- İlk mikroişlemci Intel firmasının geliştirdiği 4004'tür
  - 4 bitlik mikroişlemci
  - Adresleme kapasitesi: 4096 x 4 bit
  - Komut seti 45 komuttan oluşuyor
  - 30 gram ağırlığında
  - Saniyede 50000 işlem (30 ton ENIAC saniyede 100000 işlem)
  - Oyun ve küçük kontrol sistemlerinde kullanıldı
  - RTL (direnç –transistör lojisi ile tasarlanmış)
- Sonrasında daha yüksek frekanslı 4040 mikroişlemci geliştirildi

# Mikroişlemci Çağı

- 1971'de Intel 8008 mikroişlemciyi tanıttı
  - 8-bitlik bir mikroişlemci
  - 16KB adresleme kapasitesi
  - Toplamda 48 farklı komut yürütebiliyordu
- Mikroişlemcilerin daha karmaşık sistemlerde kullanımı mümkün oldu

# Mikroişlemci Çağı

- 1973 yılında Intel 8080 mikroişlemciyi tanıttı
- İlk modern 8 bitlik mikroişlemci olarak kabul edilir
- 8080
  - 64KB adresleme kapasitesi
  - 8008'e göre yaklaşık 10 kat daha hızlı
  - TTL (transistör- transistör lojiği ile tasarlamış)

# Mikroişlemci Çağı

- 8080'in sunumundan 6 ay sonra Motorola MC6800 mikroişlemciyi sundu
- Diğer firmalar tarafından da 8 bitlik mikroişlemciler piyasaya sunuldu
- Fairchild – F8, MOS tech – 6502, National Semiconductors – IMP8, Zilog – Z8
- 1974'te MITS Altair 8800 sunuldu
  - 1975'te Bill Gates ve Paul Allen Altair 8800 için BASIC dilini geliştirdi

# Mikroişlemci Çağı

- 1977 yılında Intel 8085 mikroişlemciyi sundu
- Intel'in son 8 bitlik mikroişlemcisi
- Saniyede 769230 işlem
- Dahili saat üretici kullanımı
- Entegre komponent sayısında artış

# Modern Mikroişlemciler

- 1978 yılında 8086 ve bir yıl sonra 8088 mikroişlemciler tanıtıldı
- 16 bitlik mikroişlemciler
- Komut yürütme süresi 400 ns (saniyede 2,5 milyon işlem)
- Adresleme kapasitesi 1MB
- 4 veya 6 byte'lık komut kuyruğu mevcut (sıradaki birkaç komutun birlikte okunması)
- Çarpma bölme gibi komutların sunulması
- Varyasyonları ile 20000'i bulan komut sayısı



# Modern Mikroişlemciler

- 8086/8088 CISC (complex instruction set computers) mimarisindedir
- Yazmaç sayısında artış söz konusu
- 8086 ve 8088: 20 adet adres ucuna sahip
- 8086: 16 veri ucuna sahip
- 8088: 8 veri ucuna sahip

# Modern Mikroişlemciler

- 1983 yılında 80286 tanıtıldı
- 16MB adresleme kapasitesine sahip
- Komutlar 8086'ya benzer şekilde olmakla birlikte 16MB hafıza için komutlarda güncelleme var
- Saat frekansı 8MHz → saniyede 4 milyon işlem

# Modern Mikroişlemciler

- 1986 yılında 80386 sunuldu
- 32 bit adres yolu, 32 bit veri yolu
- 4GB adresleme kapasitesi

# Modern Mikroişlemciler

- 1989 yılında 80486
- 32 bit adres yolu, 32 bit veri yolu
- Cache kullanımı
- DX modelinde matematik işlemci ana işlemci ile birleştirilmiştir.
- Pentium, Celeron, Itanium, Core, Core 2, Core i

# SAYI SİSTEMLERİ VE DÖNÜŞÜM KURALLARI

# SAYI SİSTEMLERİ

- Mikroişlemciler ile çalışılırken sayı sistemlerinin
  - İkili taban
  - Onluk taban
  - Onaltılık taban

Ve bunlar arasındaki dönüşümlerin nasıl sağlandığının bilinmesini gerektirir

# Rakamlar

- Onluk tabanda kullanılan rakamlar 0-9 arasındadır
- Sekizli tabanda kullanılan basamaklar 0-7 arasındadır
- İkili tabanda kullanılan basamaklar 0,1'dir
- Onluk tabandan daha büyük tabanlarda eklenen her bir basamak alfabe A ve sonraki harfler ile gösterilir
  - Oniki tabanında kullanılan basamaklar 0-9,A,B'dir
- Bilgisayar sistemlerinde onluk, ikili ve onaltılı tabanlar yaygın olarak kullanılır

# Sayıların Gösterimi

- Sayı gösterimlerinde iki önemli kavram var
  - Basamak
  - Basamakların sayıdaki konumsal gösterimi
- Onluk tabanda yazılan 132 sayısı;
  - Soldan sağa 1,3,2 basamaklarından oluşmakta
  - 2 sayısı 1'ler basamağında ( $10^0$ )
  - 3 sayısı 10'lar basamağında ( $10^1$ )
  - 1 sayısı 100'ler basamağında ( $10^2$ )
  - $132 = 10^2 * 1 + 10^1 * 3 + 10^0 * 2$



# Sayıların Gösterimi

- İkili tabanda
  - $(11)_2 = 2^1 * 1 + 2^0 * 1 = 2 + 1 = 3$
- Sekizli tabanda
  - $(11)_8 = 8^1 * 1 + 8^0 * 1 = 8 + 1 = 9$
- Onluk tabanda
  - $(11)_{10} = 10^1 * 1 + 10^0 * 1 = 10 + 1 = 11$

# Sayıların Gösterimi

- Sayıların gösteriminde virgülden sonraki sayılar ise negatif üs ile ifade edilir
- Onluk tabanda
  - $(0.1)_{10} = 10^0 * 0 + 10^{-1} * 1 = 0 + 0.1 = 0.1$
- İkili tabanda
  - $(0.1)_2 = 2^0 * 0 + 2^{-1} * 1 = 0 + 0.5 = 0.5$

# Sayı Tabanları Arası Dönüşüm

- Onluk tabandan ikili tabana dönüşüm
  - $(10)_{10} = (?)_2$
  - $10/2 = 5$ , kalan 0
  - $5/2 = 2$ , kalan 1
  - $2/2 = 1$ , kalan 0
  - Kalan 1
  - $\rightarrow (1010)_2$

# Sayı Tabanları Arası Dönüşüm

- Onluk tabandan sekizli tabana dönüşüm
  - $(10)_{10} = (?)_8$
  - $10/8 = 1$ , kalan 2
  - Kalan 1
  - $\rightarrow (12)_8$

# Sayı Tabanları Arası Dönüşüm

- Onluk tabandan onaltılık tabana dönüşüm (onaltılık tabanda 0-9,A,B,C,D,E,F basamakları kullanılır)
  - $(109)_{10} = (?)_{16}$
  - $109/16 = 6$ , kalan D
  - Kalan 6
  - $\rightarrow (6D)_{16}$

# Sayı Tabanları Arası Dönüşüm

- Virgüllü sayılarda tabanlar arası dönüşüm
  - Sayı basamak değeri ile çarpılır
  - 0 elde edilene kadar işlem tekrarlanır
  - Bazı virgüllü sayılar için sonuçta 0 elde edilemeyebilir

# Sayı Tabanları Arası Dönüşüm

- $(0.625)_{10} = (?)_2$
- 0.625, tam kısım 0
- $0.625 * 2 = 1.25$ , tam kısım 1
- $0.25 * 2 = 0.5$ , tam kısım 0
- $0.5 * 2 = 1.0$ , tam kısım 1
- Virgülden sonra 0 değerine ulaşıldı
- $\rightarrow (0.101)_2$

# Sayı Tabanları Arası Dönüşüm

- $(0.2)_{10} = (?)_2$
- 0.2, tam kısım 0
- $0.2 * 2 = 0.4$ , tam kısım 0
- $0.4 * 2 = 0.8$ , tam kısım 0
- $0.8 * 2 = 1.6$ , tam kısım 1
- $0.6 * 2 = 1.2$ , tam kısım 1
- $0.2 * 2 = 0.4$ , tam kısım 0
- ...
- Virgülden 0 değerine asla ulaşamaz
- $\rightarrow (0.00110011 \dots)_2$



# İşaretli ve İşaretsiz Sayılar

- İşaretsiz sayılarda tüm basamaklar konumsal gösterimine göre değerlendirilerek sayının genliği hesaplanır
- İkili işaretli sayılarda 3 farklı gösterim söz konusudur
  - İşaretli genlik gösterimi
  - 1'e tümleyen işaretli sayı
  - 2'ye tümleyen işaretli sayı

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

# Bilgisayarlarda Veri Formatları

- Bilgisayarlarda alfanümerik verinin tutulması için farklı formatlar kullanılmaktadır
- ASCII
- Unicode
- BCD
- İşaretli, işaretsiz sayılar
- Kayan noktalı sayılar

# Bilgisayarlarda Veri Formatları

- ASCII (American Standard Code for Information Interchange)
- Alfaniümeric karakterler 7 bitlik gösterim ile ifade edilirler

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0 ^@ NUL NULL	1 ^A SOH START OF HEADING	2 ^B STX START OF TEXT	3 ^C ETX END OF TEXT	4 ^D EOT END OF TRANSM.	5 ^E ENQ ENQUIRY	6 ^F ACK ACKNOWL- EDGE	7 ^G BEL BELL	8 ^H BS BACKSP.	9 ^I HT CHARACT. TAB'TION	10 ^J LF LINE FEED	11 ^K VT LINE TAB'TION	12 ^L FF FORM FEED	13 ^M CR CARRIAGE RETURN	14 ^N SO SHIFT OUT	15 ^O SI SHIFT IN
1	16 ^P DLE DATALINK ESCAPE	17 ^Q DC1 DEVICE CONTROL1	18 ^R DC2 DEVICE CONTROL2	19 ^S DC3 DEVICE CONTROL3	20 ^T DC4 DEVICE CONTROL4	21 ^U NAK NEG.ACK- NOWLEDGE	22 ^V SYN SYNCHR. IDLE	23 ^W ETB END OF TRANS.	24 ^X CAN CANCEL	25 ^Y EM END OF MEDIUM	26 ^Z SUB SUBS- TITUTE	27 ^[ ESC ESCAPE	28 ^\ FS INFO. SEP. 4	29 ^] GS INFO. SEP. 3	30 ^^ RS INFO. SEP. 2	31 ^_ US INFO. SEP. 1
2	32 SPACE	33 ! EXCLAM. MARK	34 " QUOT. MARK	35 # NUMBER SIGN	36 \$ DOLLAR SIGN	37 % PERCENT SIGN	38 & AMPERSAND	39 ' APOS- TROPHE	40 ( LEFT PAREN.	41 ) RIGHT PAREN.	42 * ASTERISK	43 + PLUS SIGN	44 , COMMA	45 - HYPHEN- MINUS	46 . FULL STOP	47 / SOLIDUS
3	48 0 DIGIT ZERO	49 1 DIGIT ONE	50 2 DIGIT TWO	51 3 DIGIT THREE	52 4 DIGIT FOUR	53 5 DIGIT FIVE	54 6 DIGIT SIX	55 7 DIGIT SEVEN	56 8 DIGIT EIGHT	57 9 DIGIT NINE	58 : COLON	59 ; SEMI- COLON	60 < LS. -THAN SIGN	61 = EQUALS SIGN	62 > GR. -THAN SIGN	63 ? QUEST- ION MARK
4	64 @ COMM'IAL AT	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
5	80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 [ LEFT SQ. BRACKET	92 \ REVERSE SOLIDUS	93 ] RT. SQR. BRACKET	94 ^ CIRCUM'X ACCENT	95 _ LOW LINE
6	96 , GRAVE ACCENT	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
7	112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 { L. CURLY BRACKET	124   VERTICAL LINE	125 } R. CURLY BRACKET	126 ~ TILDE	127 ^? DEL DELETE

# Bilgisayarlarda Veri Formatları

- Unicode: alfanümerik karakterler 16 bitlik bir sayı ile kodlanır
- Farklı alfabeler ve yazı sistemlerini destekler
- *<http://www.unicode.org>*

# Bilgisayarlarda Veri Formatları

- BCD (Binary-Coded Decimal): Onluk gösterimdeki her bir basamak 4-bit ile ifade edilir
- BCD gösterim üzerinden işlemleri destekleyen assembly komutları mevcuttur
- Karmaşık işlemler için çok uygun değildir
- $(526)_{10} = (0101\ 0010\ 0110)_{BCD}$

# Bilgisayarlarda Veri Formatları

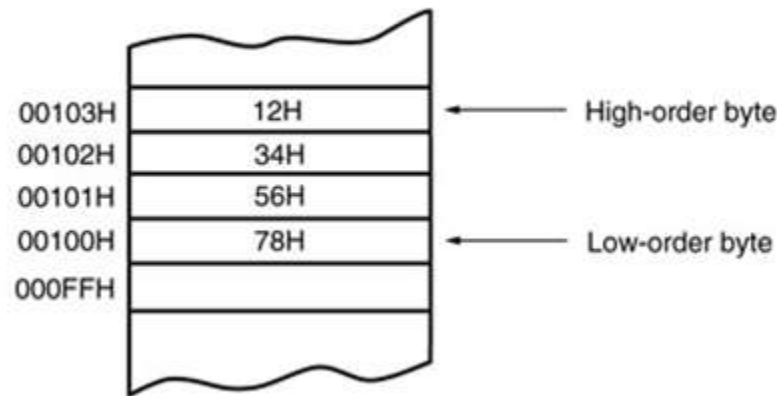
- 8-bitlik ikili sayılar
  - İşaretli, işaretsiz sayıların gösteriminde kullanılır
- En anlamlı bit
  - işaretsiz sayılarda  $2^7 = 128$  kuvvetindedir
  - İşaretli sayılarda  $-2^7 = -128$  kuvvetindedir
- 8 bit işaretsiz sayılar 0 ile 255 arasındaki değerleri gösterebilir
- 8 bit işaretli sayılar -128 ile 127 arasındaki değerleri gösterebilir
- Bilgisayarlarda işaretli sayılar 2'ye tümleyen yapıdadır



# Bilgisayarlarda Veri Formatları

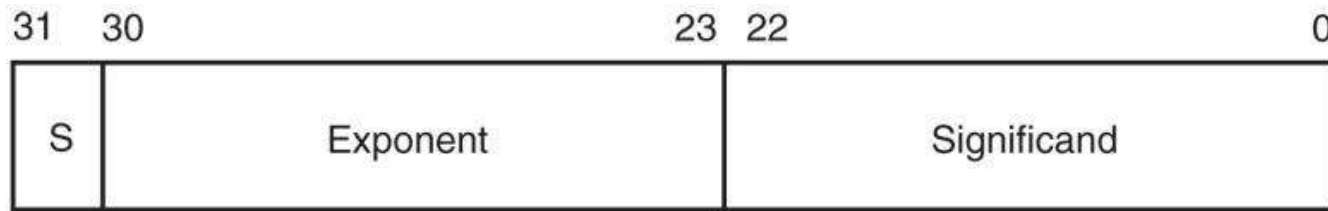
- 16-bitlik ikili sayılar (Word sized data)
- 32-bitlik ikili sayılar (Double word sized data)
- 16-bit, 32-bit sayılarda herbir byte değerinin saklanma yerine göre → little endian, big endian

Little endian →

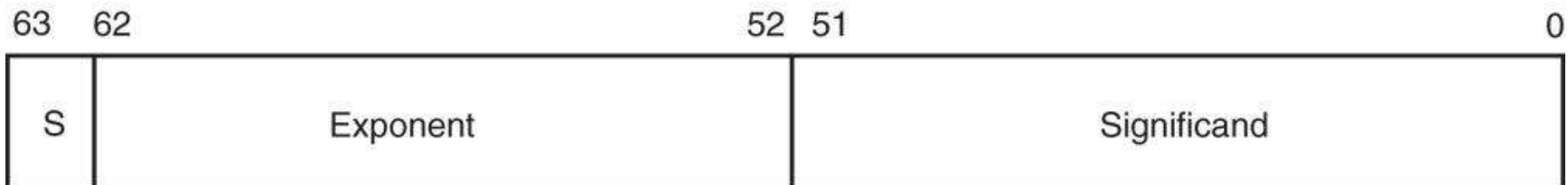


# Bilgisayarlarda Veri Formatları

- Gerçel sayılar aşağıdaki 2 hassasiyette ifade edilebilir:
  - 4-byte single precision
  - 8-byte double precision



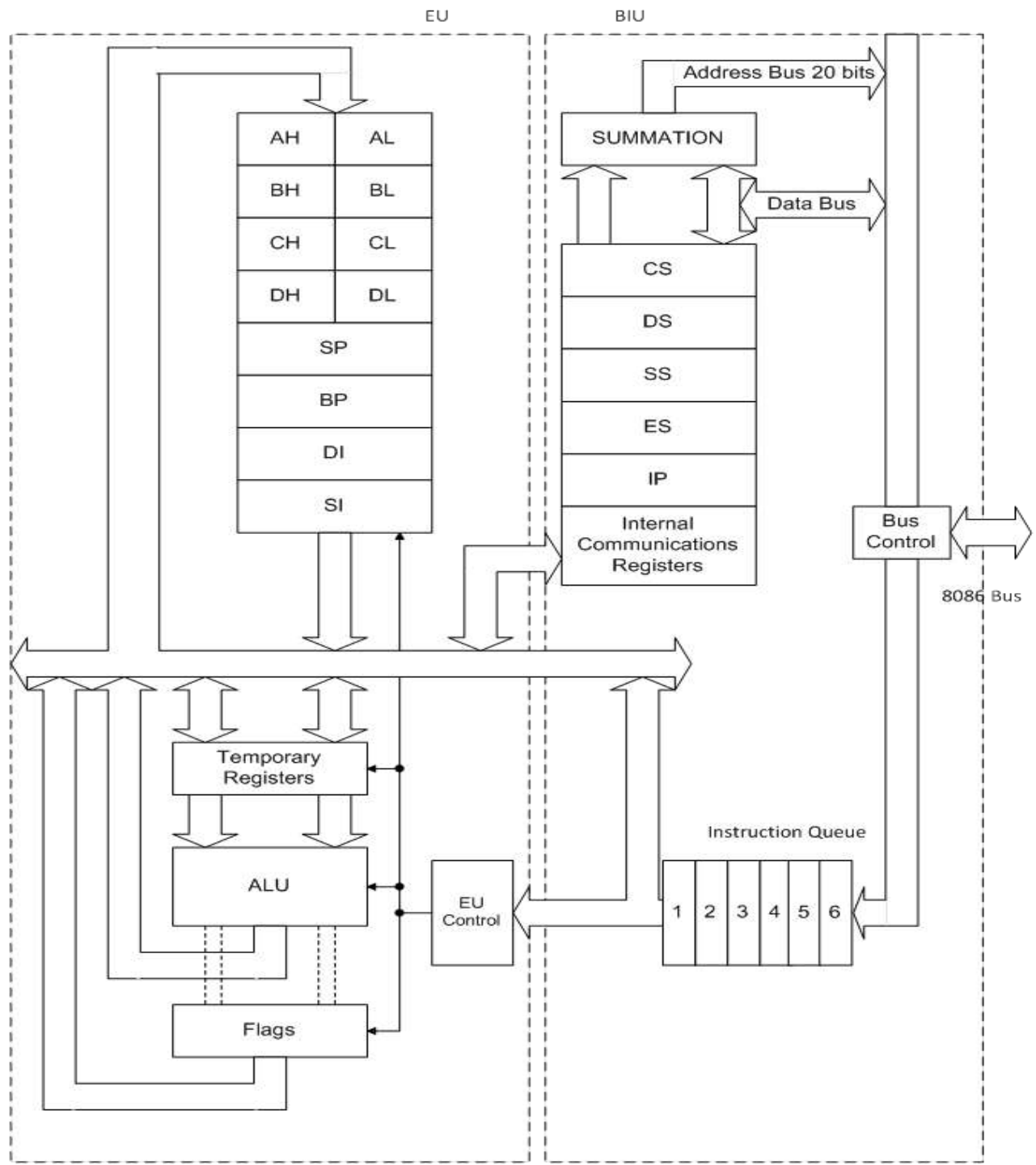
(a)



(b)

# 8086 İÇ YAPISI

# 8086 iç Yapısı



# 8086 Yazmaçları

- AX
  - AL
  - AH
- BX
  - BL
  - BH
- SI
- DI
- CX
  - CL
  - CH
- DX
  - DL
  - DH
- SP
- BP

Genel amaçlı yazmaçlar

Segment yazmaçları

- SS
- CS
- DS
- ES
- IP

Özel amaçlı yazmaç

Bayrak yazmacı

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

# 8086 Yazmaçları – AX, AL, AH

- AX : 16 bitlik akümülatör yazmaç
- AH, AL : 8 bitlik akümülatör yazmaçlar
- Aritmetik, lojik ve veri transferi işlemlerinde kullanılabilir
- Çarpma ve bölme işlemlerinde gizli operand olark kullanılır
- Giriş çıkış komutlarında kullanılır

# 8086 Yazmaçları – BX, BL, BH

- BX : 16 bitlik genel amaçlı yazmaç, (base register)
- BL, BH : 8 bitlik genel amaçlı yazmaçlar
- Dizi şeklindeki veri erişiminde kullanılır

# 8086 Yazmaçları – CX, CL, CH

- CX : 16 bitlik genel amaçlı yazmaç
- CL, CH : 8 bitlik genel amaçlı yazmaçlar
- Tekrarlı işlemlerde tekrar sayısını saklar (CX)
- Öteleme ve kaydırma işlemlerinde tekrar sayısını saklar (CL)



# 8086 Yazmaçları – DX, DL, DH

- DX : 16 bitlik genel amaçlı yazmaç
- DL, DH : 8 bitlik genel amaçlı yazmaçlar
- Çarpma ve bölme komutlarında bölünen sayıyı oluşturmak için kullanılır
- Giriş çıkış işlemlerinde port numarasını saklar

# 8086 Yazmaçları – SP

- SP : yığın yazmacı (stack pointer)
- Yığının en üst adresini işaretlemek için kullanılır
- SS ile birlikte kullanılır
- Her zaman çift bir değer gösterir
- WORD tipinde veriyi gösterir

# 8086 Yazmaçları – BP

- BP : Base pointer
- Fonksiyona parametre aktarılırken kullanılır
- SS ile birlikte kullanılır

# 8086 Yazmaçları – SI

- SI : kaynak indisi yazmacı (source index)
- Dizi komutlarında kaynak indisini tutar
  - DS ile birlikte kullanılır

# 8086 Yazmaçları – DI

- DI : hedef indisi yazmacı (destination index)
- Dizi komutlarında hedef indisini tutar
  - ES ile birlikte kullanılır

# 8086 Kesim (Segment) Yazmaçları

- CS : Kod segment, IP ile kullanılır
  - DS : Data segment, BX, SI, DI ile kullanılır
  - ES : Extra segment, DS gibi
  - SS : Stack segment, BP ve SP ile kullanılır
- 
- DS=1230H, SI=0045H ikilisi ile erişilen fiziki adres  
 $12300H + 0045H = 12345H$

# 8086 Yazmaçları – IP

- IP : Instruction pointer
  - Sıradaki işlenecek komutu gösterir
  - CS ile birlikte kullanılır
- 
- Efektif program adresi :
  - $CS \times 10H + IP$

# 8086 Bayrak Yazmacı

- **Carry Flag (CF)** : İşaretsiz işlemlerde taşma olursa 1 değerini alır
- **Parity Flag (PF)** : İşlem sonucunda 1 olan bitlerin sayısı tek ise 0, çift ise 1 değerini alır
- **Auxiliary Flag (AF)** : 4 bitlik kısımların toplama-çıkarma sonucu elde değerini tutar
- **Zero Flag (ZF)** : İşlem sonucu 0 ise ZF=1 olur
- **Sign Flag (SF)** : İşlem sonucu negatif ise SF=1 olur



# 8086 Bayrak Yazmacı

- **Trap Flag (TF)** : Her komuttan sonra kesme oluşmasını sağlar
- **Interrupt enable Flag (IF)** : Kesme kaynaklarının kesme oluşturmalarına izin verir
- **Direction Flag (DF)** : Dizi işlemlerinde başlangıç adresinden itibaren arttırarak/azaltarak sıradaki göze erişimi belirler
- **Overflow Flag (OF)** : İşaretli işlemlerde taşma durumunda 1 değerini alır

# 32 bit $\mu$ P Yazmaçları (80386)

- EAX
  - AX
    - AL
    - AH
- ECX
  - CX
    - CL
    - CH
- SS
- CS
- DS
- ES
- EIP
- FS
- GS
- EFLAGS
- EBX
  - BX
    - BL
    - BH
- EDX
  - DX
    - DL
    - DH
- ESI
- ESP
- EBP
- EDI

# 64 bit $\mu$ P Yazmaçları

- RAX
  - EAX
    - AX
      - AL
      - AH
- RBX
  - EBX
    - BX
      - BL
      - BH
- RSI
- RDI
- RCX
  - ECX
    - CX
      - CL
      - CH
- RDX
  - EDX
    - DX
      - DL
      - DH
- RSP
- RBP
- SS
- CS
- DS
- ES
- FS
- GS
- RIP
- R8-R15
- RFLAGS

REAL – PROTECTED MOD

# Real Mod Hafıza Adresleme

- 8086 real modda hafıza adresleme yapar
- Real modda sadece 1MB alan adreslenebilir
- 8086 hafıza uzayı 1MB (20 adres ucu → 1MB)
- Tüm bilgisayarlar açıldığında real modda açılır

# Real Mod Hafıza Adresleme

- Real modda segment adres ve ofset adres değerlerinin birleşimiyle hafızada istenen alana erişilir
- Bir segment değeri 64KB'lık alanı gösterir (NEDEN?)
- Ofset değeri 64KB'lık alan içinde bir yeri gösterir
- Örnek: Segment değeri 1000H, ofset değeri 2000H ise mikroişlemcide erişilen fiziki adres ne olur?
- $1000H \times 10H + 2000H = 12000H$  (1000H:2000H)

# Real Mod Hafıza Adresleme

- Varsayılan Segment ve Ofset yazmaçları
- Program hafızasına erişimde CS:IP birlikte kullanılır
- Yığın (stack) erişiminde SS:SP veya SS:BP kullanılır
- Veri erişiminde DS:BX, DS:DI, DS:SI kullanılır
- String işlemlerinde ES:DI ile kullanılır

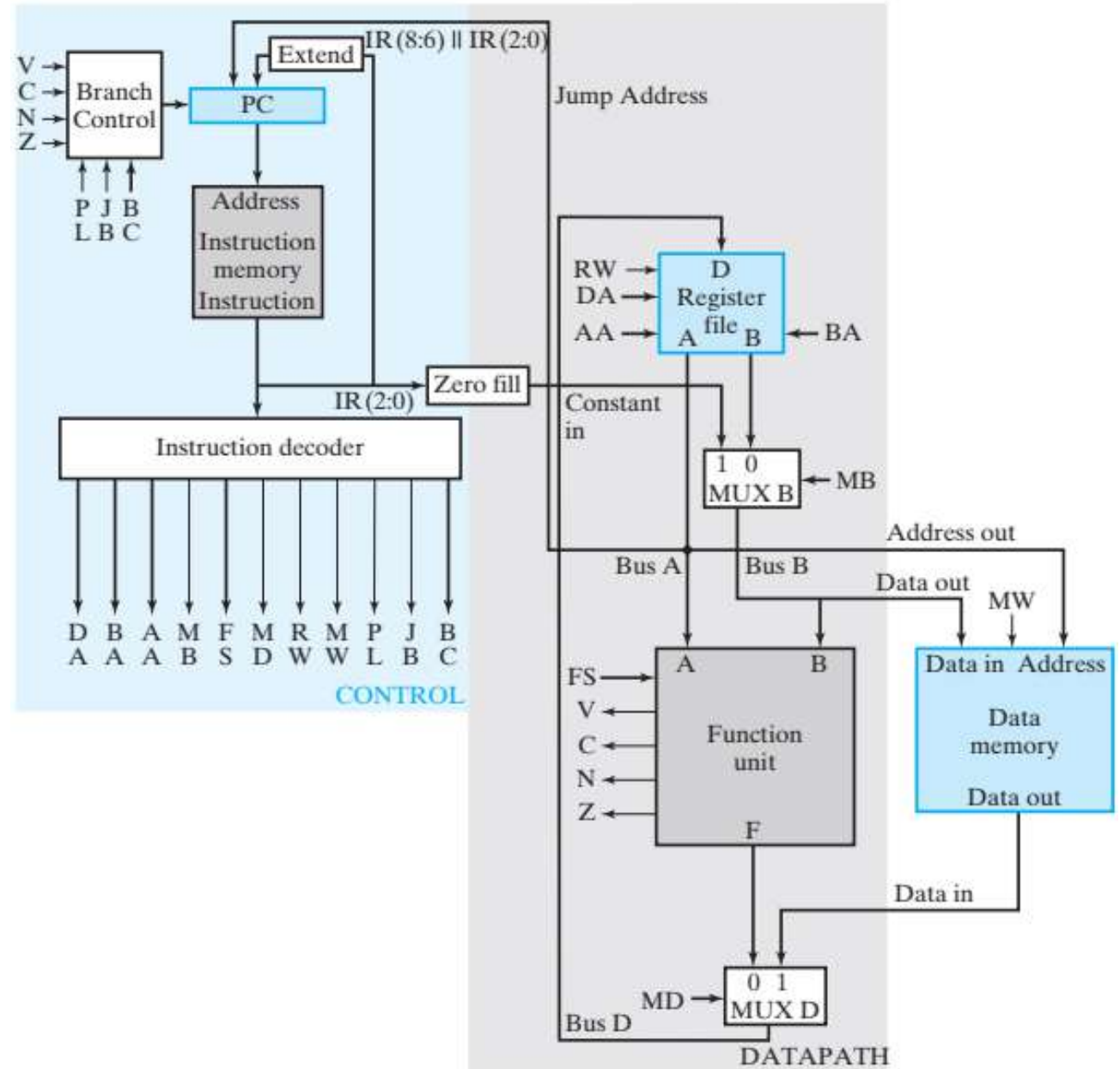
# Protected Mod Hafıza Adresleme

- 1MB'tan daha geniş hafızayı adreslemede kullanılır
- Segment adres değeri bir tablonun (descriptor table) bir satırını gösterir
- Bu tablonun herbir gözünde kullanım için ayrılmış segmentin boyu, yeri, erişim izinleri yazılıdır
- Tabloların global ve yerel versiyonları vardır
- Global descriptor tablosunda tüm programlar için kullanılabilecek segment bilgisi yer alır
- Yerel descriptor tablosunda ise uygulamaya özel segment bilgisi yer alır

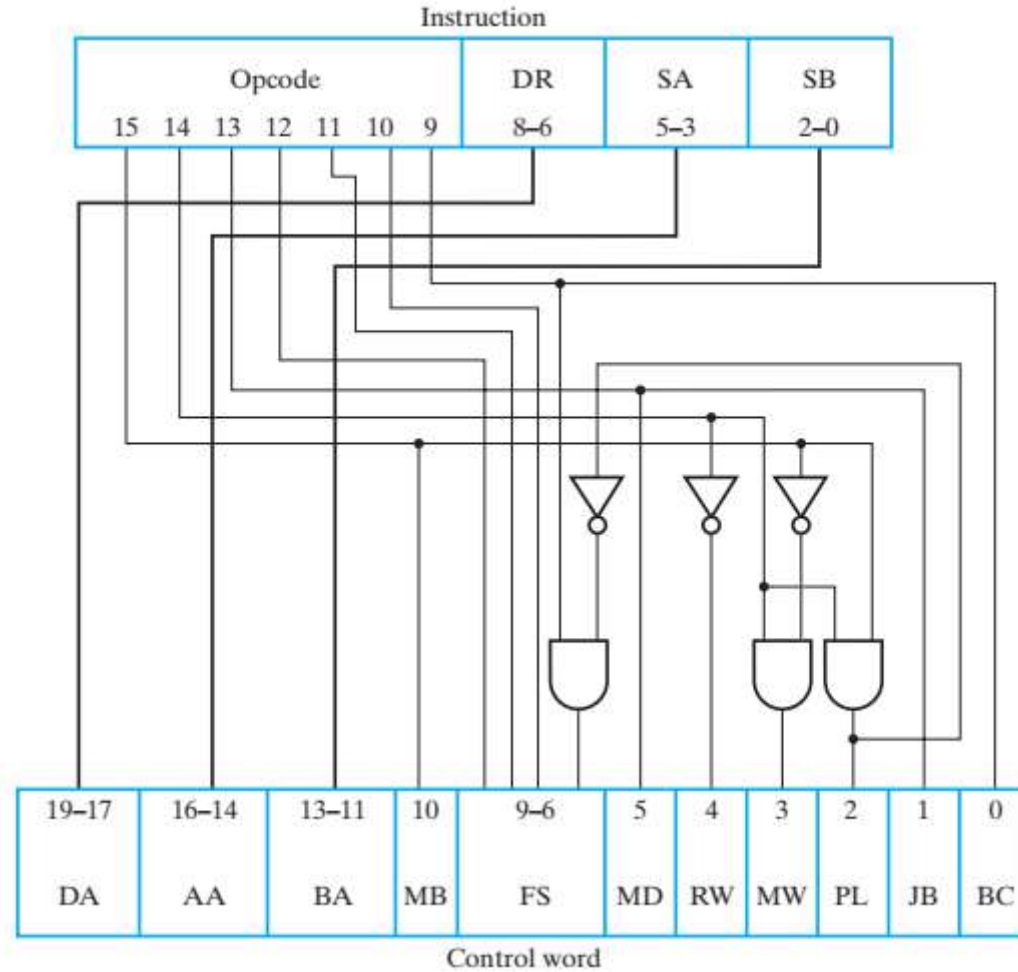


EK – Komut nasıl değerlendirilir:

# Basit bir işlemci iç yapısı



# Basit bir işlemciye ilişkin komut çözümleme devresi



# ALT SEVİYE PROGRAMLAMA

Hafta 2

Dr. Öğr. Üyesi Erkan USLU

# 80x86 KOMUT YAPISI

# 80x86 Komut Yapısı

*{Etiket :}* **Mnemonic**

*{{Operand1,} Operand2}*

*{; Açıklamalar}*

# 80x86 Komut Yapısı

- 80x86'da komutlar:
  - Operand almayan
  - Tek operand alan
  - İki operand alan
  - Üç operand (80x86 ailesi üst serilerinde)
  - $\text{Operand1} \leftarrow \text{Operand1} \text{ **işlem** } \text{Operand2}$
  - Operandlar aynı tipte olmalı veya uygun sözde komut dönüşümü yapılmalı

# 80x86 VERİ AKTARIM KOMUTLARI



# Komutlardaki Kısaltmalar

- acc : akümülatör
- reg : 8/16 bitlik yazmaç
- regb : 8 bitlik yazmaç
- regw : 16 bitlik yazmaç
- sreg : segment (kesim) yazmacı
- mem : bellek adresi
- idata : 8/16 bitlik sabit değer
- disp8/disp16 : [-128...0...127]/[-32768...0...32767]
- dest/scr : hedef/kaynak

# Veri Aktarım Komutları

- MOV
  - LEA
  - LDS
  - LES
  - XCHG
  - XLAT/XLATB
- 
- Bayraklar etkilenmez

# Veri Aktarım Komutları

- MOV : move data
- MOV dest, src
- $\text{dest} \leftarrow \text{src}$
- dest=mem ve src=mem olamaz
- dest=sreg ve src=sreg olamaz
- dest=sreg ve src=idata olamaz
- dest ve src aynı tipte olmalıdır

# Veri Aktarım Komutları

- 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

# Veri Aktarım Komutları

- LEA : load effective address
- LEA regw, mem
- regw  $\leftarrow$  offset(mem)

# Veri Aktarım Komutları

- LDS : load data segment register
- LDS regw, mem
- regw  $\leftarrow$  [mem]
- DS  $\leftarrow$  [mem+2]

# Veri Aktarım Komutları

- LES : load extra segment register
- LES regw, mem
- regw  $\leftarrow$  [mem]
- ES  $\leftarrow$  [mem+2]

# Veri Aktarım Komutları

- XCHG : exchange
- XCHG dest, src
- dest <-> src
- XCHG reg,reg
- XCHG reg, mem
- XCHG mem, reg



# Veri Aktarım Komutları

- XLAT/XLATB : translate byte (XLAT = XLATB)
- Doğrudan operandı olmayan bir komut, AL gizli operand
- $AL \leftarrow DS:[BX+AL]$
- DS:[BX] adresindeki tablonun AL numaralı indisindeki değeri AL yazmacına kopyalar

# ARİTMETİK KOMUTLAR

# Aritmetik Komutlar

- ADD
- ADC
- SUB
- SBB
- INC
- DEC
- NEG
- CMP
- MUL
- IMUL
- DIV
- IDIV

# Aritmetik Komutlar

- ADD : addition
- ADD dest, src
- $\text{dest} \leftarrow \text{dest} + \text{src}$
- ADD reg, idata
- ADD mem, idata; PTR gerekli
- ADD reg, reg
- ADD reg, mem
- ADD mem, reg

# Aritmetik Komutlar

- ADC : add with carry
- ADC dest, src
- $\text{dest} \leftarrow \text{dest} + \text{src} + \text{CF}$
- ADC reg, idata
- ADC mem, idata; PTR gerekli
- ADC reg, reg
- ADC reg, mem
- ADC mem, reg

# Aritmetik Komutlar

- SUB : subtraction
- SUB dest, src
- $\text{dest} \leftarrow \text{dest} - \text{src}$
- SUB reg, idata
- SUB mem, idata; PTR gerekli
- SUB reg, reg
- SUB reg, mem
- SUB mem, reg

# Aritmetik Komutlar

- SBB : subtraction with borrow
- SBB dest, src
- $\text{dest} \leftarrow \text{dest} - \text{src} - \text{CF}$
- SBB reg, idata
- SBB mem, idata; PTR gerekli
- SBB reg, reg
- SBB reg, mem
- SBB mem, reg

# Aritmetik Komutlar

- INC : increment
- INC dest
- $\text{dest} \leftarrow \text{dest} + 1$
- INC reg
- INC mem; PTR gerekli



# Aritmetik Komutlar

- DEC : decrement
- DEC dest
- $\text{dest} \leftarrow \text{dest} - 1$
- DEC reg
- DEC mem; PTR gerekli

# Aritmetik Komutlar

- NEG : negate / two's complement
- NEG dest
- $\text{dest} \leftarrow 0 - \text{dest}$
- NEG reg
- NEG mem; PTR gerekli

# Aritmetik Komutlar

- CMP : compare
- CMP dest, src
- $\text{dest} - \text{src}$
- Sonuç saklanmaz, bayraklar etkilenir
- Komut sonrasında koşullu dallanma komutları kullanılabilir
  - $\text{dest} > \text{src}$ ,  $\text{dest} \geq \text{src}$ ,
  - $\text{dest} = \text{src}$ ,
  - $\text{dest} \leq \text{src}$ ,  $\text{dest} < \text{src}$

# Aritmetik Komutlar

- CMP reg, idata
- CMP mem, idata
- CMP reg, reg
- CMP reg, mem
- CMP mem, reg

# Aritmetik Komutlar

- MUL : unsigned multiplication
- MUL src
- Eğer src 8 bit ise :  $AX \leftarrow AL \times src$
- Eğer src 16 bit ise :  $DX:AX \leftarrow AX \times src$
- MUL reg
- MUL mem; PTR gerekli

# Aritmetik Komutlar

- IMUL : integer signed multiplication
- IMUL src
- Eğer src 8 bit ise :  $AX \leftarrow AL \times src$
- Eğer src 16 bit ise :  $DX:AX \leftarrow AX \times src$
- Sonuç ve operandlar işaretli sayı olarak değerlendirilir
- IMUL reg
- IMUL mem; PTR gerekli

# Aritmetik Komutlar

- DIV : unsigned divide
- DIV src
- Eğer src 8 bit ise :  $AL \leftarrow AX / src, AH \leftarrow AX \% src$
- Eğer src 16 bit ise :

$AX \leftarrow DX:AX / src, DX \leftarrow DX:AX \% src$

- DIV reg
- DIV mem; PTR gerekli

# Aritmetik Komutlar

- IDIV : integer signed divide
- IDIV src
- Eğer src 8 bit ise :  $AL \leftarrow AX / src, AH \leftarrow AX \% src$
- Eğer src 16 bit ise :

$AX \leftarrow DX:AX / src, DX \leftarrow DX:AX \% src$

Sonuç ve operandlar işaretli sayı olarak değerlendirilir

- IDIV reg
- IDIV mem; PTR gerekli



DALLANMA  
KOMUTLARI

# Dallanma Komutları

- Koşullu Dallanma
  - Basit Koşullu Dallanma
  - İşaretsiz Sayı İşlemlerinde Dallanma
  - İşaretli Sayı İşlemlerinde Dallanma
- Koşulsuz Dallanma
- Çağırma Komutları
- Dönüş Komutları

# Dallanma Komutları (1)

## Koşullu Dallanma

- Basit Koşullu Dallanma

- JE/JZ
- JNZ/JNE
- JS
- JNS
- JO
- JNO
- JP/JPE
- JNP/JPO

- İşaretsiz Sayı İşlemlerinde Dallanma

- JB/JNAE/JC
- JA/JNBE
- JAE/JNB/JNC
- JBE/JNA

- İşaretli Sayı İşlemlerinde Dallanma

- JL/JNGE
- JNL/JGE
- JLE/JNG
- JG/JNLE

# Basit Koşullu Dallanma

- JE/JZ: jump zero/equal, ZF=1 ise dallan
- JNZ/JNE: jump not zero/not equal, ZF=0 ise dallan
- JS: jump if sign, SF=1 ise dallan
- JNS: jump no sign, SF=0 ise dallan
- JO: jump if overflow, OF=1 ise dallan
- JNO: jump if no overflow, OF=0 ise dallan
- JP/JPE: jump on parity/even parity, PF=1 ise dallan
- JNP/JPO: jump no parity/odd parity, PF=0 ise dallan

# İşaretsiz Sayı İşlemlerinde Dallanma

- İşaretsiz sayılarda CMP işlemi sonrasında koşullu dallanma oluşturmak için kullanılır
- JB/JNAE/JC: jump if below
- JA/JNBE: jump if above
- JAE/JNB/JNC: jump if above or equal
- JBE/JNA: jump if below or equal

# İşaretli Sayı İşlemlerinde Dallanma

- İşaretli sayılarda CMP işlemi sonrasında koşullu dallanma oluşturmak için kullanılır
- JL/JNGE: jump if less
- JNL/JGE: jump if greater or equal
- JLE/JNG: jump if less or equal
- JG/JNLE: jump if greater

# Aritmetik Komutlar- İşlemci koşullara nasıl karar verir

	İşaretsiz Sayılarda	İşaretli Sayılarda
$\text{dest} > \text{src}$	$\text{CF}=0$ VE $\text{ZF}=0$	$\text{ZF}=0$ VE $\text{SF}=\text{OF}$
$\text{dest} \geq \text{src}$	$\text{CF}=0$	$\text{SF}=\text{OF}$
$\text{dest} = \text{src}$	$\text{ZF}=1$	$\text{ZF}=1$
$\text{dest} \leq \text{src}$	$\text{CF}=1$ VEYA $\text{ZF}=1$	$\text{ZF}=1$ VEYA $\text{SF} \neq \text{OF}$
$\text{dest} < \text{src}$	$\text{CF}=1$	$\text{SF} \neq \text{OF}$

# Dallanma Komutları (2)

- Koşulsuz Dallanma

- JMP
- JMP FAR PTR

- Çağırma Komutları

- CALL
- CALL FAR PTR
- INT
- INTO

- Dönüş Komutları

- RET
- RETF
- IRET



# Koşulsuz Dallanma

- JMP: near jump
- JMP disp
- $IP \leftarrow IP + disp$
- JMP reg
- $IP \leftarrow reg$
- JMP mem
- $IP \leftarrow [mem]$

# Koşulsuz Dallanma

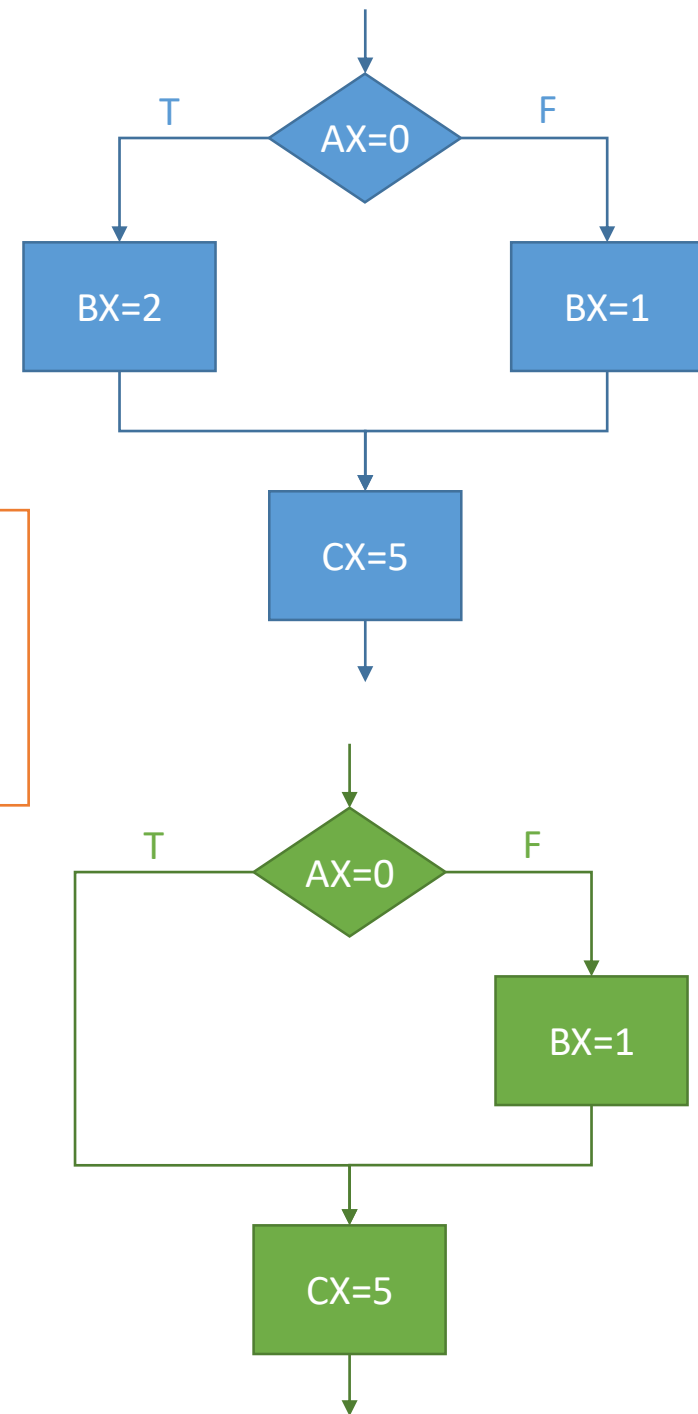
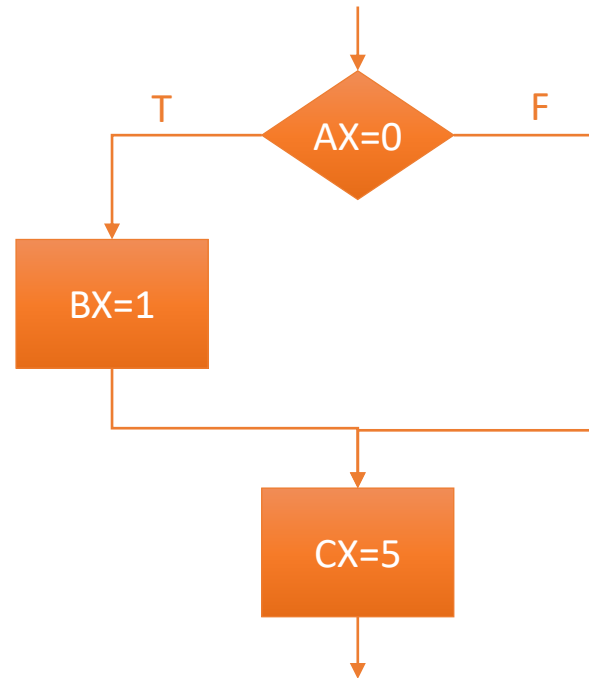
- JMP FAR PTR: far jump
- JMP FAR PTR idata1:idata2
- $CS \leftarrow idata1, IP \leftarrow idata2$
  
- JMP FAR PTR mem
- $IP \leftarrow [mem], CS \leftarrow [mem+2]$

# Dallanma Şekilleri

- If then else

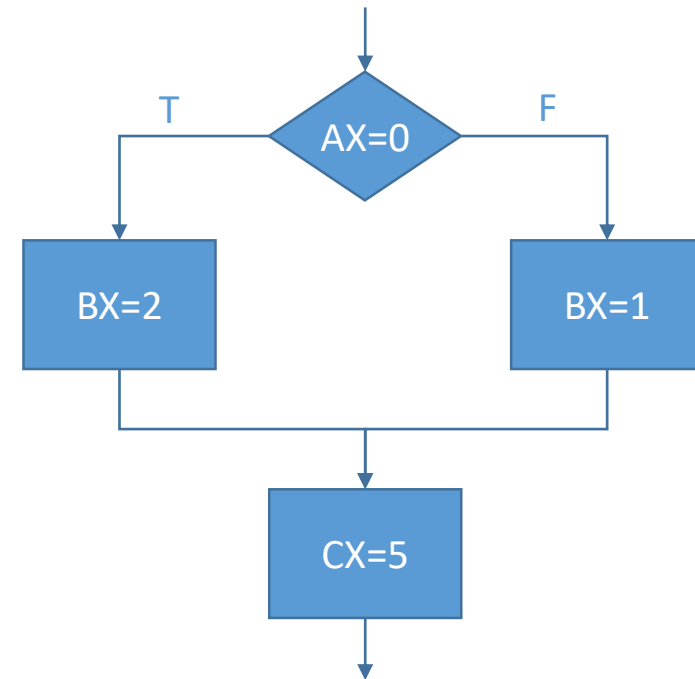
- If null else

If null then



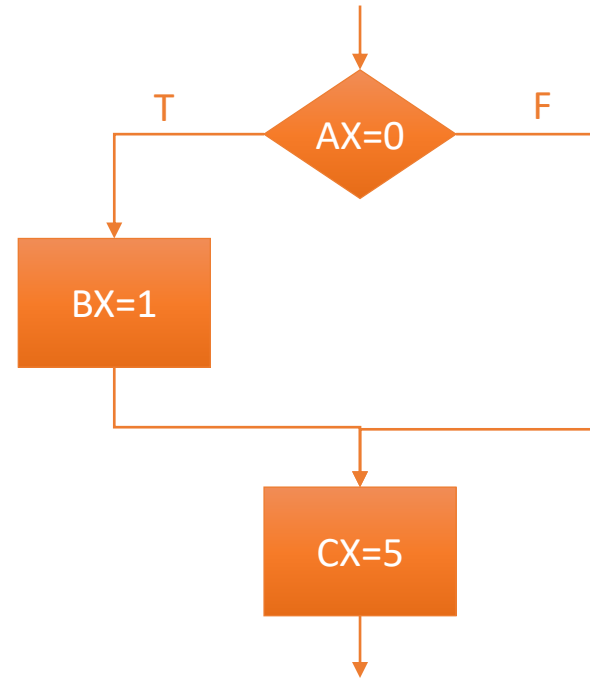
# Dallanma Şekilleri

- `CMP AX, 0`
- `JZ true`
- `MOV BX, 1`
- `JMP next`
- `true: MOV BX, 2`
- `next: MOV CX, 5`



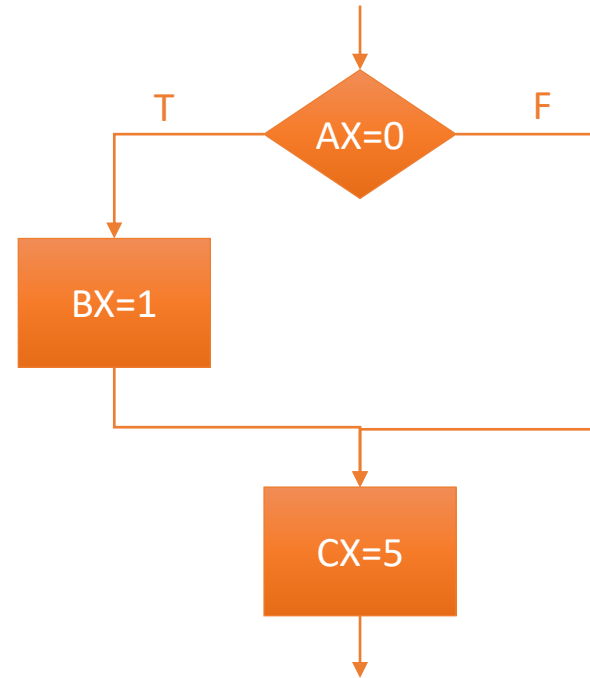
# Dallanma Şekilleri

- `CMP AX, 0`
- `JZ true`
- `JMP next`
- `true: MOV BX, 1`
- `next: MOV CX, 5`



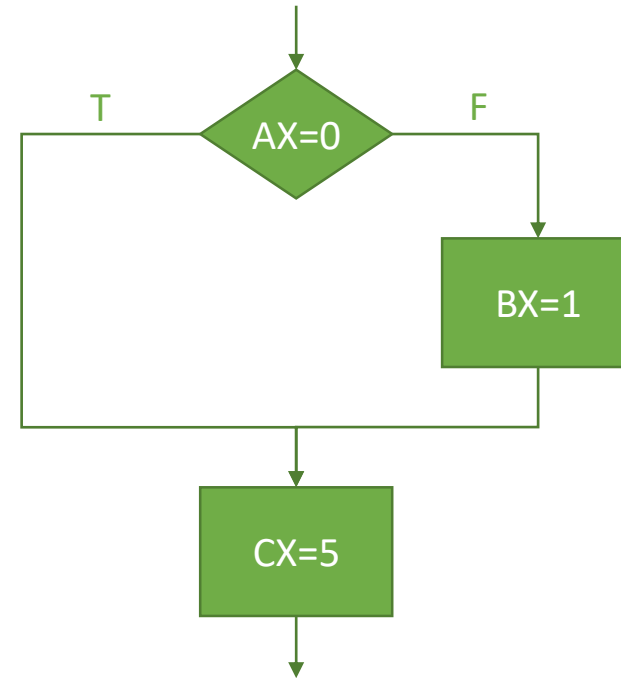
# Dallanma Şekilleri

- `CMP AX, 0`
- `JNZ false`
- `MOV BX, 1`
- `false: MOV CX, 5`



# Dallanma Şekilleri

- `CMP AX, 0`
- `JZ true`
- `MOV BX, 1`
- `true: MOV CX, 5`



# Çağırma Komutları

- CALL: near procedure call, (prosedür dönüşü RET ile)
- Yığına sadece IP atılır
- CALL disp
- $IP \leftarrow IP + disp$
- CALL mem
- $IP \leftarrow reg$
- CALL reg
- $IP \leftarrow reg$



# Çağırma Komutları

- CALL FAR PTR: far procedure call, (prosedür dönüşü RETF ile)
- Yığına CS ve IP atılır
- CALL FAR PTR idata1:idata2
- $CS \leftarrow idata1, IP \leftarrow idata2$
- CALL FAR PTR mem
- $IP \leftarrow [mem], CS \leftarrow [mem+2]$

# Çağırma Komutları

- INT: software interrupt
- INT idata
- İşlemci aşağıdaki işlemleri otomatik yapar
  - PUSHF
  - PUSH CS
  - PUSH IP
  - $TF \leftarrow 0$
  - $IF \leftarrow 0$
  - $IP \leftarrow 0000:[idata*4]$
  - $CS \leftarrow 0000:[idata*4+2]$

# Çağırma Komutları

- INTO: interrupt on overflow
- INTO
- INTO = INT 04H

# Dönüş Komutları

- RET: near return from procedure
  - Yığından sadece IP çekilir
- RETF: far return from procedure
  - Yığından IP ve CS çekilir
- IRET: return from interrupt
  - Yığından IP, CS ve FLAG çekilir

EK - Tüm dallanma komutları ve  
bayrak koşulları

Opcode	Description	CPU Flags
JA	Above	CF = 0 and ZF = 0
JAE	Above or equal	CF = 0
JB	Bellow	CF
JBE	Bellow or equal	CF or ZF
JC	Carry	CF
JE	Equality	ZF
JG	Greater <sup>(s)</sup>	ZF = 0 and SF = OF
JGE	Greater of equal <sup>(s)</sup>	SF = OF
JL	Less <sup>(s)</sup>	SF ≠ OF
JLE	Less equal <sup>(s)</sup>	ZF or SF ≠ OF
JNA	Not above	CF or ZF
JNAE	Neither above nor equal	CF
JNB	Not bellow	CF = 0
JNBE	Neither bellow nor equal	CF = 0 and ZF = 0
JNC	Not carry	CF = 0
JNE	Not equal	ZF = 0
JNG	Not greater	ZF or SF ≠ OF
JNGE	Neither greater nor equal	SF ≠ OF
JNL	Not less	SF = OF
JNLE	Not less nor equal	ZF = 0 and SF = OF

Opcode	Description	CPU Flags
JNO	Not overflow	OF = 0
JNP	Not parity	PF = 0
JNS	Not negative	SF = 0
JNZ	Not zero	ZF = 0
JO	Overflow <sup>(s)</sup>	OF
JP	Parity	PF
JPE	Parity	PF
JPO	Not parity	PF = 0
JS	Negative <sup>(s)</sup>	SF
JZ	Null	ZF

(s): signed numbers

EK - Dallanma komutlarının  
eşdeğerliliği

JB ≡ JC ≡ JNAE

JAE ≡ JNB ≡ JNC

JE ≡ JZ

JNE ≡ JNZ

JBE ≡ JNA

JA ≡ JNBE

JP ≡ JPE

JNP ≡ JPO

JL ≡ JNGE

JGE ≡ JNL

JLE ≡ JNG

JG ≡ JNLE



# ALT SEVİYE PROGRAMLAMA

Hafta 3

Dr. Öğr. Üyesi Erkan USLU

# YIĞIN KOMUTLARI

# Yığın Komutları

- POP
- POPF
- PUSH
- PUSHF

# Yığın Komutları

- POP: pop word from stack
- POP dest
- $\text{dest} \leftarrow \text{SS}:[\text{SP}]$
- POP regw
- POP mem

# Yığın Komutları

- POPF: pop flag from stack
- POPF
- FLAG  $\leftarrow$  SS:[SP]

# Yığın Komutları

- PUSH: push word to stack
- PUSH src
- $SS:[SP] \leftarrow src$
- PUSH idata
- PUSH regw
- PUSH mem
- PUSH sreg

# Yığın Komutları

- PUSH: push flag to stack
- PUSHF
- $SS:[SP] \leftarrow FLAG$

DÖNGÜ  
KOMUTLARI



# Döngü Komutları

- LOOP
- LOOPZ/LOOPE
- LOOPNZ/LOOPNE
- JCXZ

# Döngü Komutları

- LOOP: loop until complete
- LOOP disp
- Her loop geçişinde CX azaltılır, sonrasında disp ile gösterilen adrese gidilir
- CX≠0 olduğu sürece tekrarlanır

# Döngü Komutları

	toplam	CX	SI	
MOV toplam, 0;	0	?	?	
MOV CX, 5;	0	5	?	
MOV SI, 1;	0	5	1	
L1: ADD toplam, SI;	1	15	1	5
INC SI;	1	15	1	6
LOOP L1;	1	15	0	6

# Döngü Komutları

- LOOPZ/LOOPE: loop zero/loop equal
- LOOPZ disp
- Her loop geçişinde CX azaltılır, sonrasında disp ile gösterilen adrese gidilir
- $CX \neq 0$  ve  $ZF=1$  olduğu sürece tekrarlanır

# Döngü Komutları

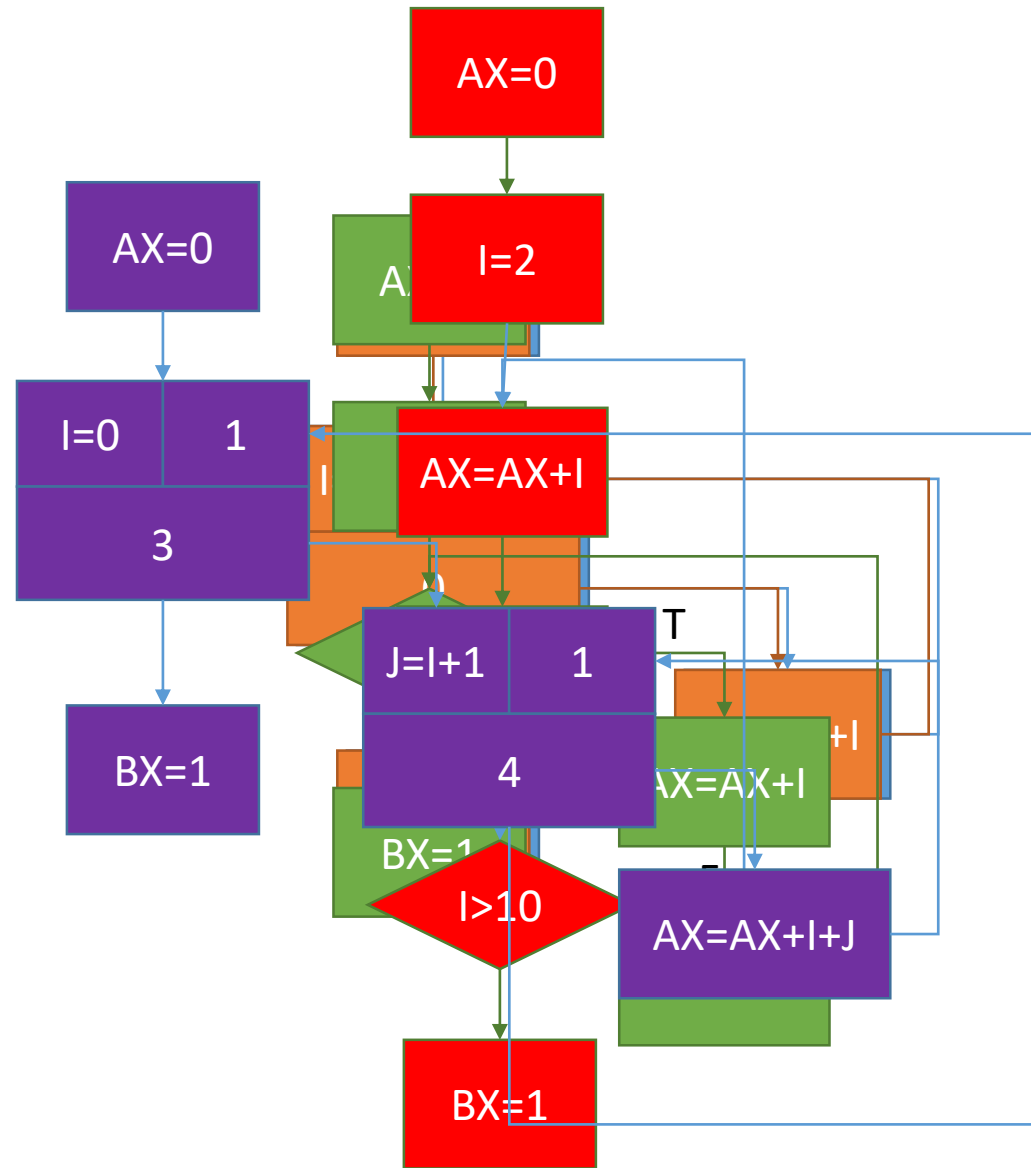
- LOOPNZ/LOOPNE: loop not zero/loop not equal
- LOOPNZ disp
- Her loop geçişinde CX azaltılır, sonrasında disp ile gösterilen adrese gidilir
- $CX \neq 0$  ve  $ZF=0$  olduğu sürece tekrarlanır

# Döngü Komutları

- JCXZ: jump cx zero
- JCXZ disp
- CX=0 ise dallan, CX değeri kontrolü kullanıcıda

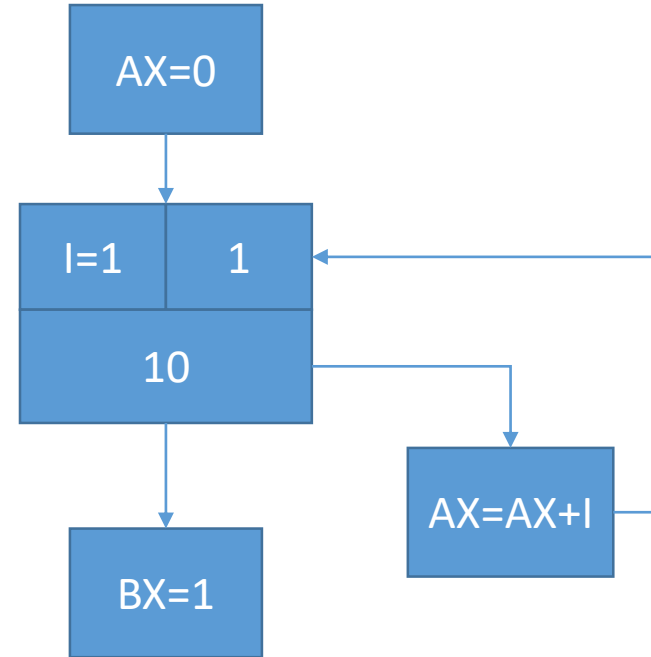
# Döngü Şekilleri

- For to
- For downto
- While
- Repeat until
- For for



# Döngü Şekilleri

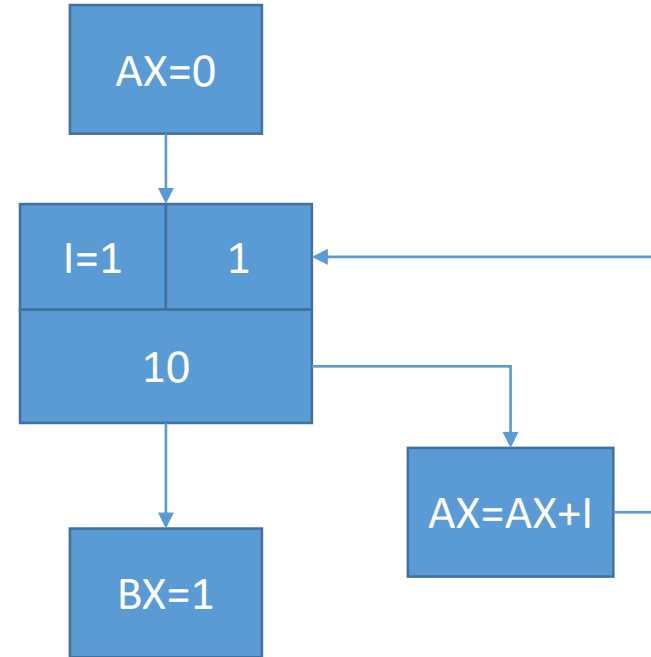
- XOR AX, AX
- MOV SI, 1
- MOV CX, 10
- tekrar: ADD AX, SI
- INC SI
- LOOP tekrar





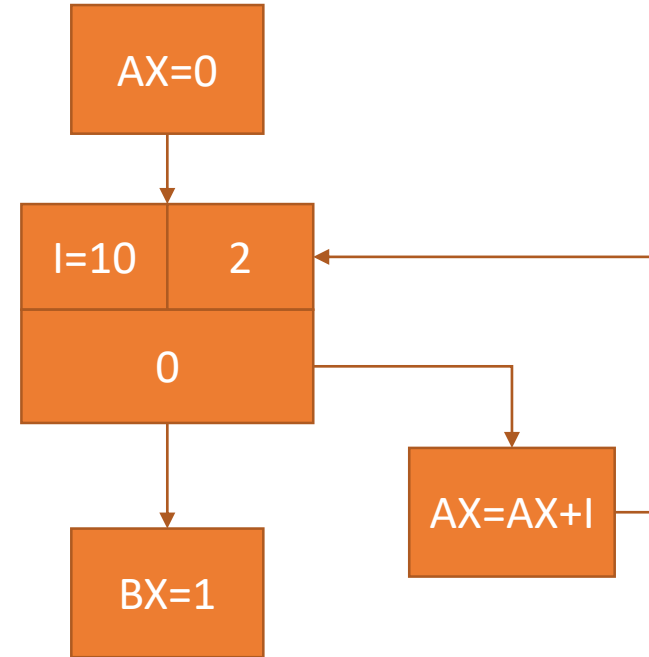
# Döngü Şekilleri

- XOR AX, AX
- MOV CX, 10
- tekrar: ADD AX, CX
- LOOP tekrar



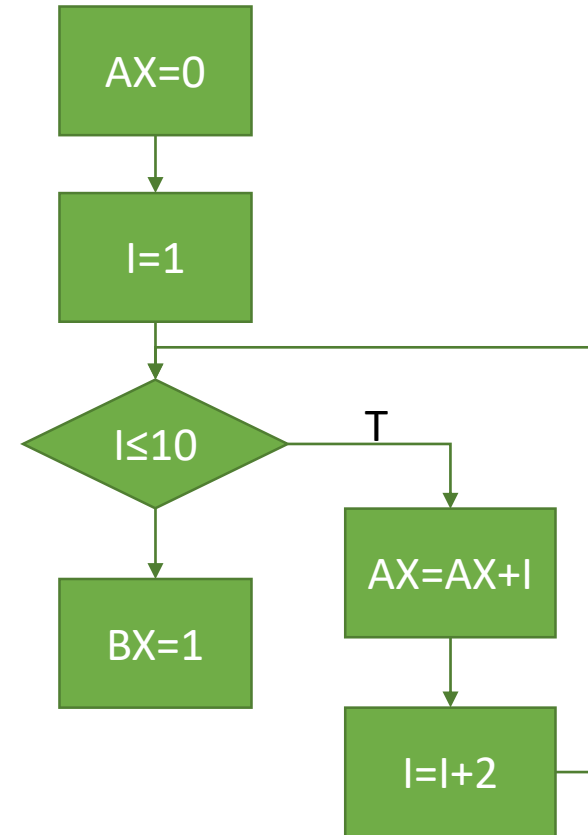
# Döngü Şekilleri

- XOR AX, AX
- MOV CX, 10
- tekrar: ADD AX, CX
- SUB CX, 2
- JCXZ devam
- JMP tekrar
- devam:



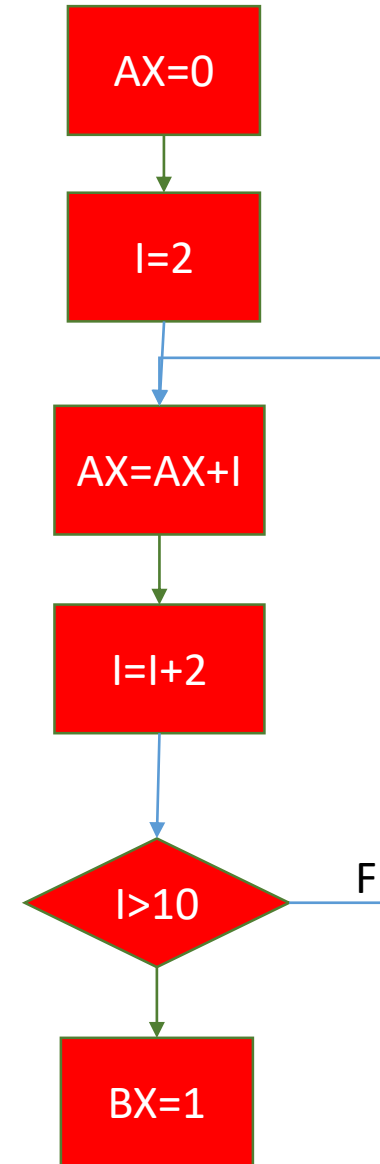
# Döngü Şekilleri

- XOR AX, AX
- MOV SI, 1
- tekrar: CMP SI, 10
- JA devam
- ADD AX, SI
- ADD SI, 2
- JMP tekrar
- devam: MOV BX, 1



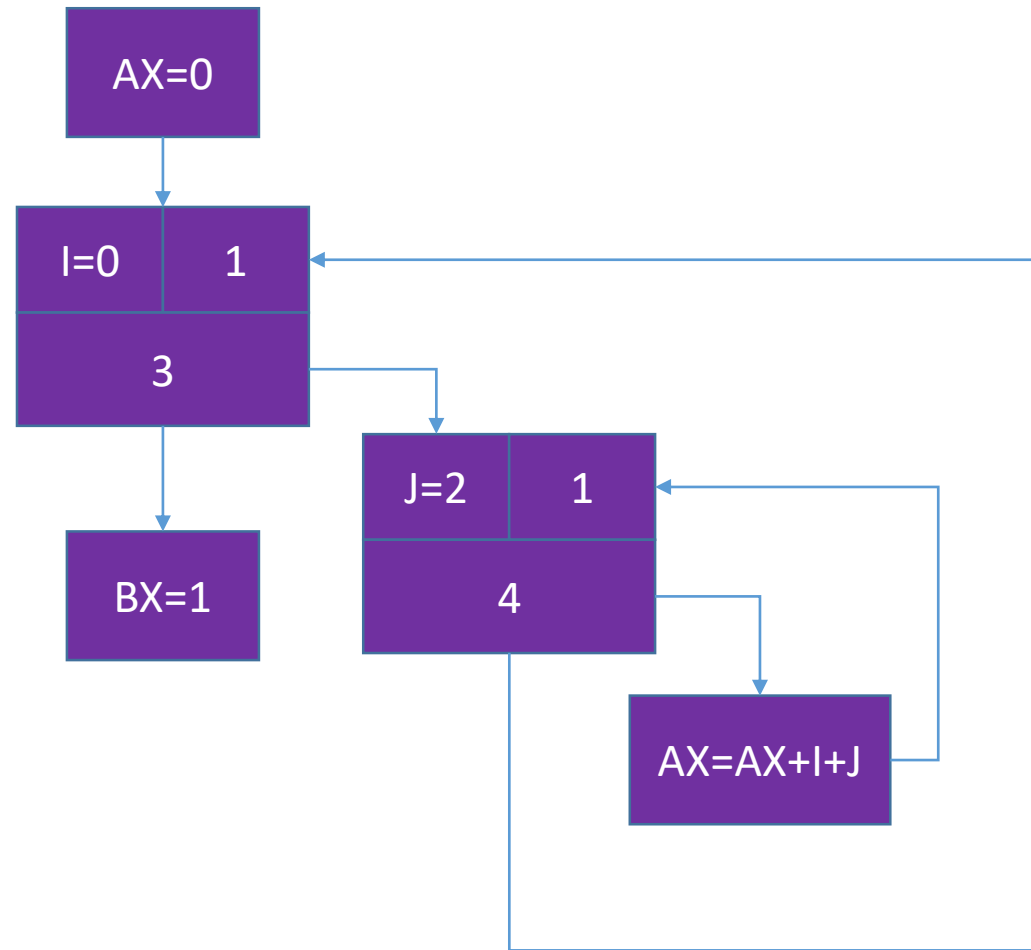
# Döngü Şekilleri

- XOR AX, AX
- MOV SI, 2
- tekrar: ADD AX, SI
- ADD SI, 2
- CMP SI, 10
- JBE tekrar
- MOV BX, 1



# Döngü Şekilleri

- MOV AX, 0
- MOV CX, 4
- MOV DI, 0
- L2: PUSH CX
- MOV CX, 3
- MOV SI, 2
- L1: ADD AX, DI
- ADD AX, SI
- INC SI
- LOOP L1
- POP CX
- INC DI
- LOOP L2
- MOV BX, 1



# BAYRAKLARLA İLGİLİ KOMUTLARI

# Bayraklarla İlgili Komutlar

- CLC: clear carry f.
- CMC: complement carry f.
- STC: set carry f.
- CLD: clear direction f.
- STD: set direction f.
- STI: set interrupt f.
- CLI: clear intrerrupt f.
- LAHF: load AH with flag:  $AH \leftarrow SF, ZF, ?, AF, ?, PF, ?CF$
- SAHF: store AH to flag:  $SF, ZF, ?, AF, ?, PF, ?CF \leftarrow AH$

# MANTIKSAL KOMUTLAR



# Mantıksal Komutlar

- NOT
- OR
- AND
- XOR
- TEST

# Mantıksal Komutlar

- NOT : bitwise not
- NOT dest
- $\text{dest} \leftarrow (\text{dest})'$
- NOT reg
- NOT mem; PTR gerekli

# Mantıksal Komutlar

- OR : bitwise or
- OR dest, src
- $\text{dest} \leftarrow \text{dest OR src}$
- OR reg, idata
- OR mem, idata; PTR gerekli
- OR reg, reg
- OR reg, mem
- OR mem, reg

# Mantıksal Komutlar

- AND : bitwise and
- AND dest, src
- $\text{dest} \leftarrow \text{dest AND src}$
- AND reg, idata
- AND mem, idata; PTR gerekli
- AND reg, reg
- AND reg, mem
- AND mem, reg

# Mantıksal Komutlar

- XOR : bitwise exclusive or
- XOR dest, src
- $\text{dest} \leftarrow \text{dest XOR src}$
- XOR reg, idata
- XOR mem, idata; PTR gerekli
- XOR reg, reg
- XOR reg, mem
- XOR mem, reg

# Mantıksal Komutlar

- XOR AL, 0FFH ; AL'de 1'ler 0, 0'lar 1 yapılır
- XOR AX, AX ;  $AX \leftarrow 0$ , 4 clock cycle, 3 byte
- MOV AX, 0 ; 3 clock cycle, 2 byte

# Mantıksal Komutlar

- TEST : test bits
- TEST dest, src
- dest AND src
- Sadece bayraklar değişir, sonuç saklanmaz
- TEST reg, idata
- TEST mem, idata; PTR gerekli
- TEST reg, reg
- TEST reg, mem
- TEST mem, reg

# ÖTELEME VE DÖNDÜRME KOMUTLARI

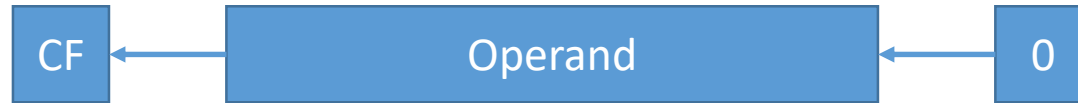


# Öteleme ve Döndürme Komutları

- SHL
- SAL
- SHR
- SAR
- RCL
- RCR
- ROL
- ROR

# Öteleme ve Döndürme Komutları

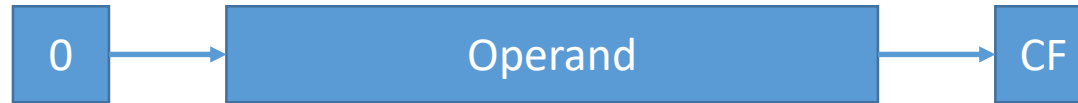
- SHL : shift left logical



- SHL reg, 1
- SHL mem, 1
- SHL reg, CL
- SHL mem, CL
- **SAL = SHL**

# Öteleme ve Döndürme Komutları

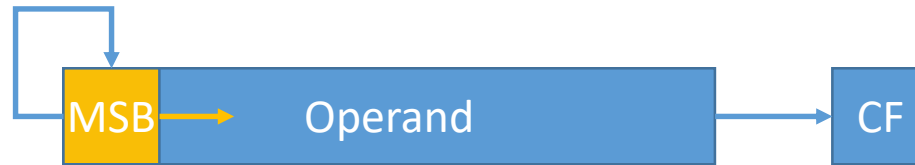
- SHR : shift right logical



- SHR reg, 1
- SHR mem, 1
- SHR reg, CL
- SHR mem, CL

# Öteleme ve Döndürme Komutları

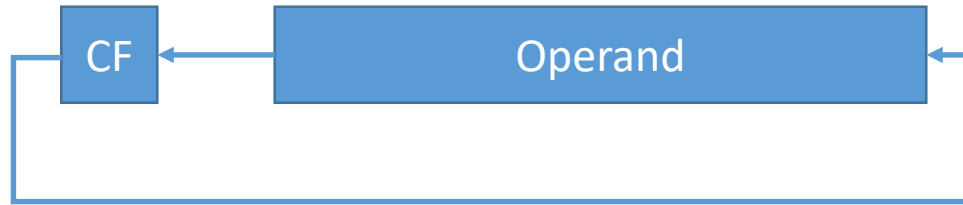
- SAR : shift arithmetic right



- Öteleme işleminde sayının işaret bitini dikkate alır
- SAR reg, 1
- SAR mem, 1
- SAR reg, CL
- SAR mem, CL

# Öteleme ve Döndürme Komutları

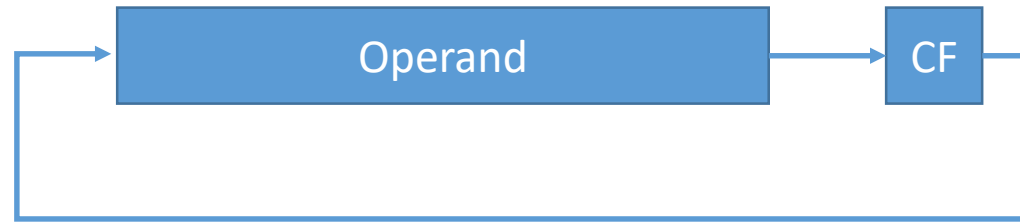
- RCL : rotate through carry left



- RCL reg, 1
- RCL mem, 1
- RCL reg, CL
- RCL mem, CL

# Öteleme ve Döndürme Komutları

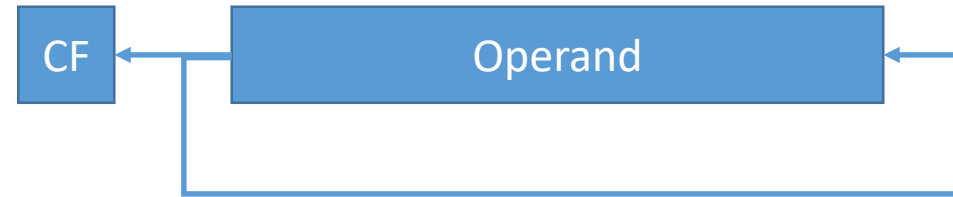
- RCR : rotate through carry right



- RCR reg, 1
- RCR mem, 1
- RCR reg, CL
- RCR mem, CL

# Öteleme ve Döndürme Komutları

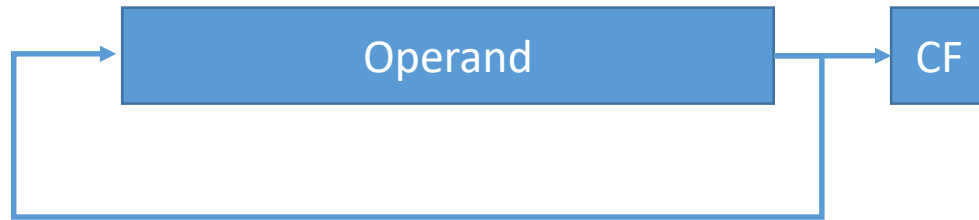
- ROL : rotate left



- ROL reg, 1
- ROL mem, 1
- ROL reg, CL
- ROL mem, CL

# Öteleme ve Döndürme Komutları

- ROR : rotate right



- ROR reg, 1
- ROR mem, 1
- ROR reg, CL
- ROR mem, CL



# ALT SEVİYE PROGRAMLAMA

Hafta 4

Dr. Öğr. Üyesi Erkan USLU

# DİZİ KOMUTLARI

# Dizi (string) Komutları

- MOVSB
- MOVSW
- CMPSB
- CMPSW
- SCASB
- SCASW
- LODSB
- LODSW
- STOSB
- STOSW
- CBW
- CWD
- REP
- REPE/REPZ
- REPNE/REPNZ

# Dizi (string) Komutları

- Tekrarlamalı işlemler için tasarlanmış komutlardır
- İşlem tekrar sayısı CX yazmacında tutulur
- Her işlem adımında CX değeri bir otomatik olarak 1 azaltılır
- Tekrarlı dizi işlemlerini belirtmek için önekler kullanılır (REP, REPE/REPZ, REPNE/REPNZ)

# Dizi (string) Komutları

- İşlem yönü DF bayrağı ile belirlenir
- DF=0 ise her adımında dizinin bir sonraki adresi için işlem tekrarlanır
- DF=1 ise her adımında dizinin bir önceki adresi için işlem tekrarlanır

# Dizi (string) Komutları

- MOVS: bir bellek bölgesinin bir başka bellek bölgesine kopyalanması
- CMPS: farklı iki bellek bölgesinin içeriklerinin karşılaştırılması
- SCAS: bellek bölgesinin içeriğinin AX/AL ile karşılaştırılması
- LODS: bellek bölgesindeki değerin AX/AL ye yükelnmesi
- STOS: bellek bölgesinin AX/AL değeri ile doldurulması
- CBW: işaret bitini koruyarak byte'ı word'e genişletme
- CWD: işaret bitini koruyarak word'u double word'e genişletme

# Dizi (string) Komutları

- MOVSB: move string byte
  - $ES:[DI] \leftarrow DS:[SI]$
  - $DI \leftarrow DI+1$ ; (DF=1 olsaydı  $DI \leftarrow DI-1$ )
  - $SI \leftarrow SI+1$ ; (DF=1 olsaydı  $SI \leftarrow SI-1$ )
- 
- MOVSW: move string word
  - $ES:[DI] \leftarrow DS:[SI]$
  - $DI \leftarrow DI+2$ ; (DF=1 olsaydı  $DI \leftarrow DI-2$ )
  - $SI \leftarrow SI+2$ ; (DF=1 olsaydı  $SI \leftarrow SI-2$ )

# Dizi (string) Komutları

- CMPSB: compare string byte
  - DS:[SI]-ES:[DI]
  - $DI \leftarrow DI+1$ ; (DF=1 olsaydı  $DI \leftarrow DI-1$ )
  - $SI \leftarrow SI+1$ ; (DF=1 olsaydı  $SI \leftarrow SI-1$ )
- 
- CMPSW: compare string word
  - DS:[SI]-ES:[DI]
  - $DI \leftarrow DI+2$ ; (DF=1 olsaydı  $DI \leftarrow DI-2$ )
  - $SI \leftarrow SI+2$ ; (DF=1 olsaydı  $SI \leftarrow SI-2$ )



# Dizi (string) Komutları

- SCASB: scan string byte
  - AL-ES:[DI]
  - $DI \leftarrow DI+1$ ; (DF=1 olsaydı  $DI \leftarrow DI-1$ )
- SCASW: scan string word
  - AX-ES:[DI]
  - $DI \leftarrow DI+2$ ; (DF=1 olsaydı  $DI \leftarrow DI-2$ )

# Dizi (string) Komutları

- LODSB: load string byte
  - $AL \leftarrow DS:[SI]$
  - $SI \leftarrow SI+1$ ; (DF=1 olsaydı  $SI \leftarrow SI-1$ )
- LODSW: load string word
  - $AX \leftarrow DS:[SI]$
  - $SI \leftarrow SI+2$ ; (DF=1 olsaydı  $SI \leftarrow SI-2$ )

# Dizi (string) Komutları

- STOSB: store string byte
  - $ES:[DI] \leftarrow AL$
  - $DI \leftarrow DI+1$ ; (DF=1 olsaydı  $DI \leftarrow DI-1$ )
- STOSW: store string word
  - $ES:[DI] \leftarrow AX$
  - $DI \leftarrow DI+2$ ; (DF=1 olsaydı  $DI \leftarrow DI-2$ )

# Dizi (string) Komutları

- CBW: convert byte to word
- $AX \leftarrow AL$
- CWD: convert word to double word
- $DX:AX \leftarrow AX$

# Dizi (string) Komutları

- LEA DI, dizi
  - CLD
  - XOR AL, AL
  - MOV CX, size
  - REP STOSB
- 
- dizi  $\leftarrow$  0

# Dizi (string) Komutları

- LEA SI, dizi1
  - LEA DI, dizi2
  - CLD
  - MOV CX, size
  - REP MOVSB
- 
- dizi2  $\leftarrow$  dizi1

# Dizi (string) Komutları - Önekler

- REP: repeat
- CX=0 olana kadar tekrarla
  
- REPE/REPZ: repeat equal
- CX=0 veya ZF=0 olana kadar tekrarla
  
- REPNE/REPNZ: repeat not equal
- CX=0 veya ZF=1 olana kadar tekrarla

GİRİŞ ÇIKIŞ KOMUTLARI



# Giriş Çıkış Komutları

- IN
- OUT

# Giriş Çıkış Komutları

- IN: input from port adress
- IN acc, idata; 0-255 arası portlara erişim
- IN acc, DX; 0-65535 arası portlara erişim

# Giriş Çıkış Komutları

- OUT: output to port address
- OUT idata, acc; 0-255 arası portlara erişim
- OUT DX, acc; 0-65535 arası portlara erişim

# DURDURMA ve BEKLETME KOMUTLARI

# Durdurma ve Bekletme Komutları

- HLT: halt, işlemci durur, kesme ile işlemci kaldığı komuttan devam eder
- NOP: no operation
- WAIT: wait until not busy, yardımcı işlemcinin sonucunu beklemek için kullanılır

# BCD DÜZENLEME KOMUTLARI

# BCD Düzenleme Komutları

- AAA
- AAD
- AAM
- AAS
- DAA
- DAS

# BCD Düzenleme Komutları

- AAA: ASCII adjust after addition
- ASCII değeri olarak tutulan iki sayının toplamını BCD olarak düzenler
- XOR AX, AX
- MOV AL, "6"
- ADD AL, "7"
- AAA ; AX  $\leftarrow$  0103H , BCD olarak 13 sayısı



# BCD Düzenleme Komutları

- AAD: ASCII adjust before division
- BCD olarak tutulan iki basamaklı sayının bölme öncesi düzenlenmesi için kullanılır
- MOV AH, 04H
- MOV AL, 05H; AX'te BCD 45 sayısı var
- AAD;  $AX \leftarrow 002DH$

# BCD Düzenleme Komutları

- AAM: ASCII adjust after multiplication
- ASCII değer olarak tutulan iki sayının toplamını BCD olarak düzenler
- MOV AL, 4
- MOV AH, 8
- MUL AH; AX'te 0020H var
- AAM ; AX  $\leftarrow$  0302H , BCD olarak 32 sayısı

# BCD Düzenleme Komutları

- AAS: ASCII adjust after subtraction
- ASCII değer olarak tutulan iki sayının farkını BCD olarak düzenler
- MOV AL, "5"
- SUB AL, "7"; AL'de FEH var
- AAS;  $AL \leftarrow 08H$ ,  $CF=1$ ;  $-10+8=-2$

# BCD Düzenleme Komutları

- DAA: decimal adjust AL after addition
- Sıkıştırılmış BCD formatlı sayıların toplanması sonucu AL'deki değeri sıkıştırılmış BCD olarak düzenler
- MOV AL, 75H
- ADD AL, 19H; AL'de 8EH var
- DAA; AL  $\leftarrow$  94H

# BCD Düzenleme Komutları

- DAS: decimal adjust AL after subtraction
- Sıkıştırılmış BCD formatlı sayıların çıkarılması sonucu AL'deki değeri sıkıştırılmış BCD olarak düzenler
- MOV AL, 42H
- SUB AL, 13H; AL'de 2FH var
- DAS; AL  $\leftarrow$  29H

# ALT SEVİYE PROGRAMLAMA

Hafta 5

Dr. Öğr. Üyesi Erkan USLU

ADRESLEME MODLARI

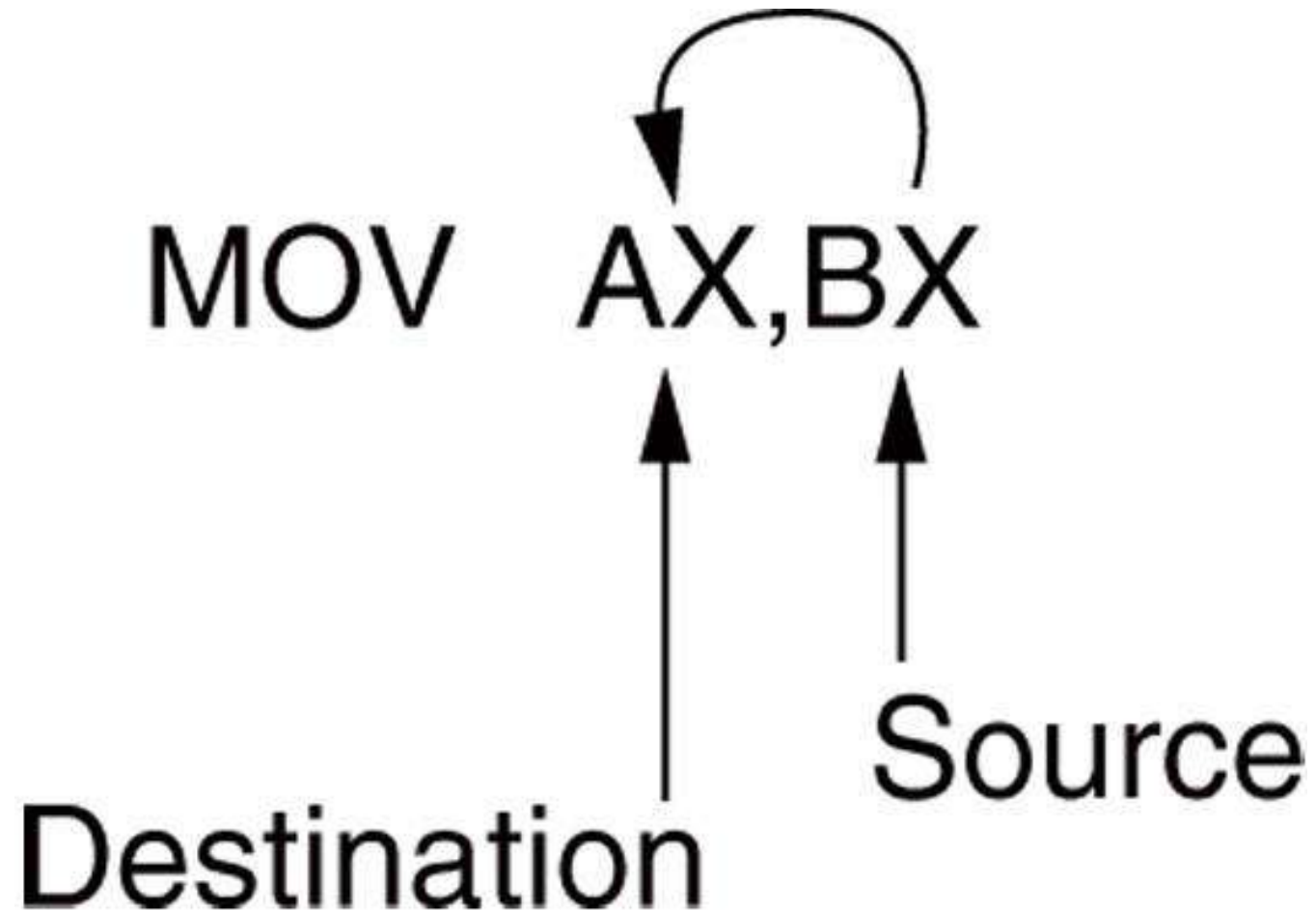
# Adresleme Modları

- Verimli assembly programları geliştirebilmek için komutlar ile birlikte kullanılan adresleme modlarının bilinmesi gerekmektedir.
- Üç tip erişim için adresleme modu mevcuttur:
  - Veri adresleme modları
  - Program hafızası adresleme modları
  - Yığın adresleme modları



# Veri Adresleme Modları

- 8086 assembly genel yapısı



# Veri Adresleme Modları

- 8086 assembly genel yapısı
  - $AX \leftarrow 1234H$
  - $BX \leftarrow ABCDH$
  - $MOV\ AX, BX$
  - $AX \leftarrow ABCDH$
  - $BX \leftarrow ABCDH$
- $MOV\ DST, SRC$ 
  - $DST \leftarrow SRC$

# Yazmaç Adresleme (Register Addressing)

- Yazmaç Adresleme (Register Addressing)
- 8 bitlik AL, AH, BL, BH, CL, CH, DL ,DH yazmaçları kullanılabilir
- 16 bitlik AX, BX, CX, DX, SP, BP, SI, DI yazmaçları kullanılabilir
- Yazmaç adreslemede kullanılan yazmaç genişlikleri uyumlu olmalıdır

MOV BX, CX

# Hemen Adresleme (Immediate Addressing)

- Sabit değer atamayı ifade eder
- 16 bit veya 8 bit sabit değer atama söz konusu olabilir

MOV AL, 0F2H

MOV CX, 100

MOV BL, 01010101B

MOV AH, 'A' ;ASCII A karakteri AH yazmacına atanır

# Doğrudan Adresleme (Direct Addressing)

- Erişilecek hafıza gözünün doğrudan gösterildiği durumdur

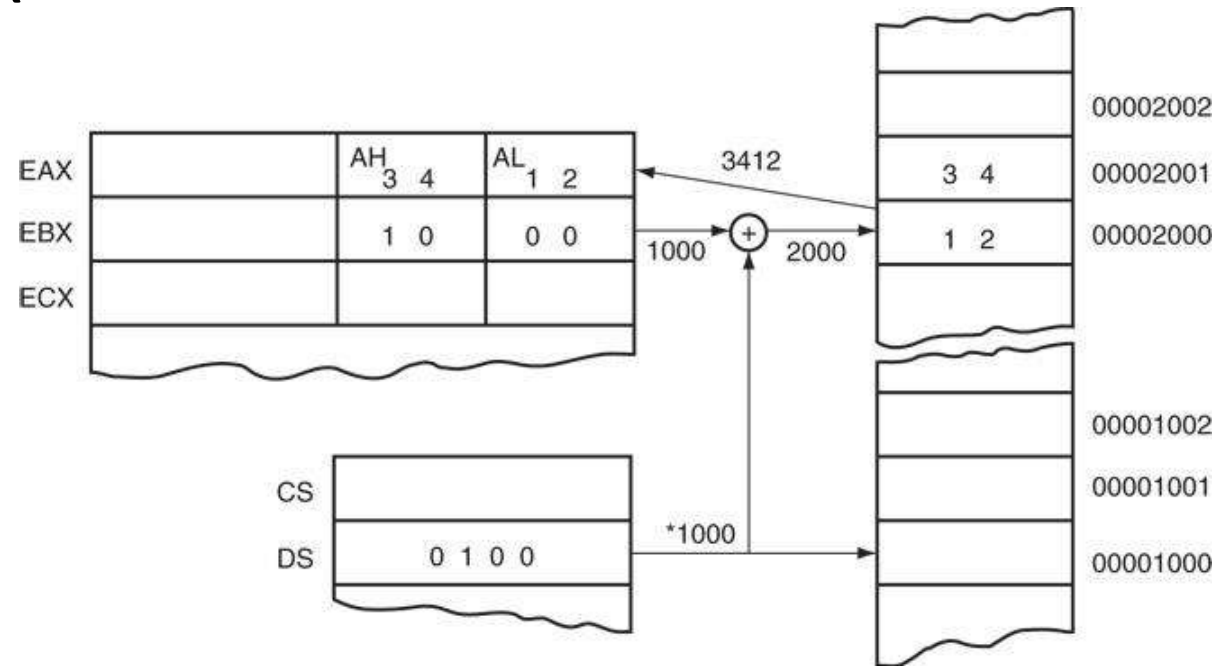
MOV AL, DATA ; DATA bir etiket olup assembler bunu karşılık gelen adres değeri ile değiştirir

MOV BX, [1234H] ; BX  $\leftarrow$  DS:1234H

# Yazmaç Dolaylı Adresleme (Register Indirect Addressing)

- BP (SS ile), BX, DI ve SI (DS ile) yazmaçları ile kullanılabilir
- Hafıza ofset değeri bir yazmaçta saklanır

MOV AX, [BX] ; AX  $\leftarrow$  DS:BX



# Dolaylı Adresleme (Indirect Addressing)

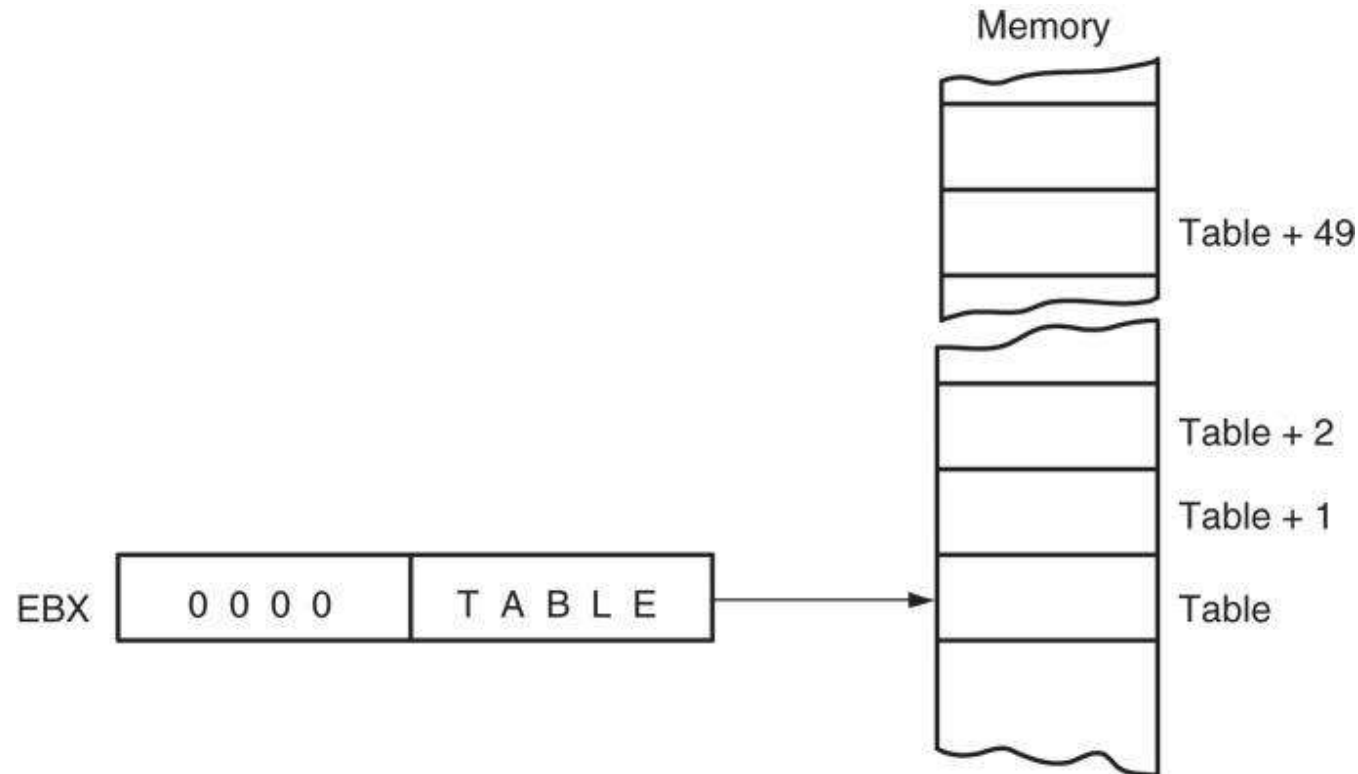
- Dolaylı adreslemelerin bir kısmında BYTE PTR, WORD PTR gibi özel tanımlayıcılar vermek gerekir
- Arttırma komutu olan INC, dolaylı adresleme ile hafızada bir Word mü yoksa Byte değeri mi arttıracığını bilemez

INC WORD PTR [BX]

- Dolaylı adreslemede operandlardan birisi yazmaç ise BYTE PTR, WORD PTR'ye gerek yoktur (NEDEN?)

# Yazmaç Dolaylı Adresleme (Register Indirect Addressing)

- Dizi olarak tutulan veriye sıralı erişimde yazmaç dolaylı adresleme kullanımı uygundur





# Base+Index Addressing

- Temelde bir dolaylı adresleme modudur
- Base yazmacı (BX veya BP) işlem yapılacak hafıza konumunun başlangıcını göstermek için kullanılır
- Index yazmaçları (DI veya SI) verinin bu başlangıç adresine görece yerini tutmak için kullanılır

MOV DX, [BX+DI]

# Yazmaç Göreli Adresleme (Register Relative Addressing)

- Base (BP veya BX) veya Index (DI, SI) yazmaçlarının bir sabit ofset değeri ile kullanılmasını ifade eder

MOV AX, [BX+100H]

# Base Relative + Index Addressing

- İki boyutlu veri adresleme için uygundur

MOV AX,[BX + SI + 100H]

# Program Hafızası Adresleme Modları

- Program akışı sırasında fonksiyon çağırma, koşullu ve koşulsuz dallanma komutları ile farklı program hafızası adresleme modları kullanılır
- Doğrudan (direct)
- Göreli (relative)
- Dolaylı (indirect)

# Doğrudan Program Hafızası Adresleme

- Doğrudan bir program adresine ulaşmak için kullanılır
- Mevcut kod segmentinden farklı bir kod segmentine geçiş sağlayacağı için segmentler-arası bir işlemdir
- Hem CS hem de IP değeri uygun şekilde değiştirilir

JMP 200H:300H ; CS  $\leftarrow$  200H, IP  $\leftarrow$  300H

CALL 200H:300H

# Görelî Program Hafızası Adresleme

- Mevcut IP yazmacı değerine göre hangi program hafızasının adresleneceğini ifade eder
- JMP komutu 1 byte veya 2 byte işaretli sabit değerli operand kabul eder

JMP 100

JMP 0FFH ; IP değeri 1 azalır (NEDEN?)

JMP 1000H

# Dolaylı Program Hafızası Adresleme

- CALL ve JMP komutları ile kullanılır

JMP BX

CALL [BX]

# Yığın Adresleme Modları

- Tüm yazmaçlardan yığına veri basılabilir
- CS hariç tüm yazmaçlara yığından veri çekilebilir

PUSH CS ; çalışır

POP CS ; assembler hatası verir



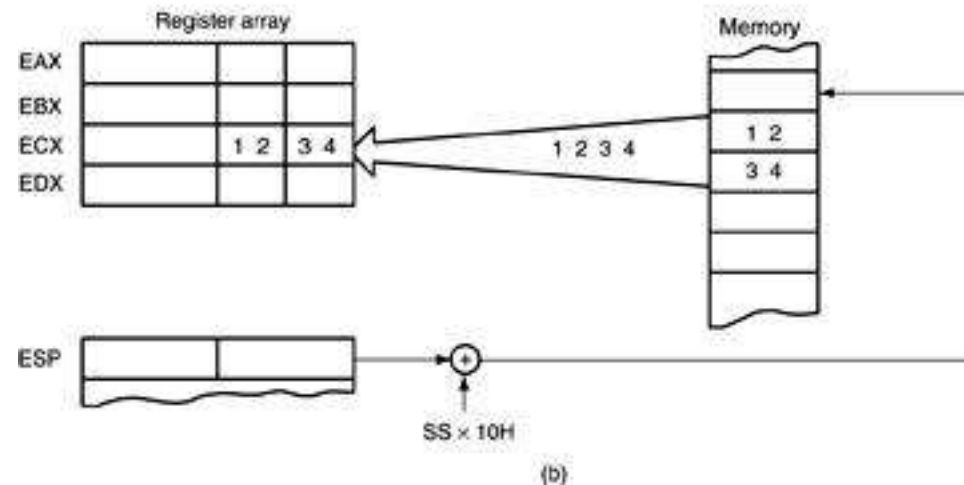
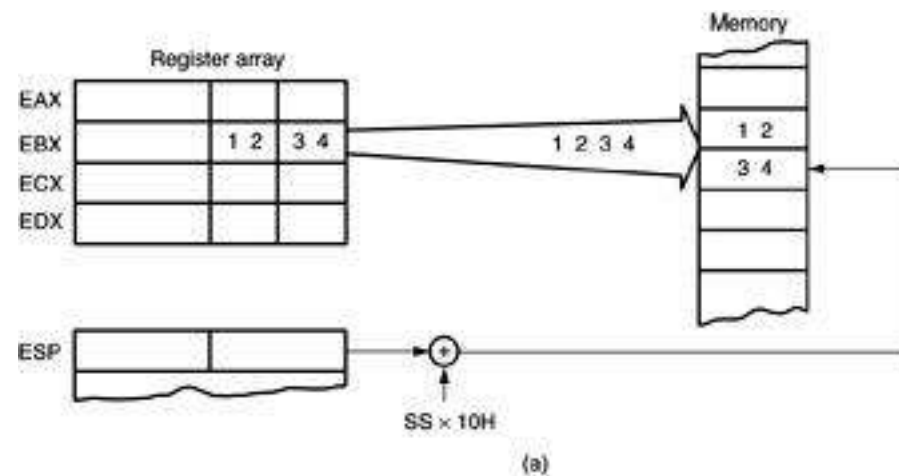
# Yığın Adresleme Modları

- 8086'da yığın geçici veri saklamak için ve fonksiyonlardan dönüşlerde dönüş adreslerini saklamak için kullanılır
  - Yığın LIFO mantığında çalışır (Last In First Out)
  - Yığın ile ilgili PUSH ve POP komutları kullanılır
  - PUSH yığına WORD basar, POP yığından WORD çeker
- PUSH BL ; (DOĞRU MU?)
- Yığın adresleme için SS:SP ikilisi kullanılır

# Yığın Adresleme Modları

- SP yazmacı programcının tanımladığı yığının genişliğini gösterecek şekilde ilk değer alır
- Her PUSH işleminde SP-1 ve SP-2 adreslerine 2 byte veri yazılır ve SP değeri 2 azaltılır
- Her POP işleminde SP+1 ve SP+2 adreslerinden 2 byte veri okunur ve SP değeri 2 arttırılır

# Yığın Adresleme Modları



SÖZDE KOMUTLAR

# Sözde Komutlar

- Assembly programın düzenlemesine yöneliktir
- Doğrudan makine kodu karşılığı yoktur
- Mnemonic'ler için makine kodu karşılığı üretilir

# Sözde Komutlar

- LST uzantılı dosyanın düzenlenmesi
  - PAGE, TITLE
- Program kesim düzenlemesi
  - SEGMENT/ENDS, ORG, ASSUME
- Veri tanımlamaları
  - DB, DW, DD, DQ, EQU, DUP, TYPEDEF, PTR, LABEL
- Yordam düzenleme
  - PROC/ENDP, EXTRN, PUBLIC
- MACRO düzenleme
  - MACRO/ENDM, INCLUDE, LOCAL
- Diğer
  - LENGTH, TYPE, SIZE, OFFSET, SEG, END

# Sözde Komutlar

- PAGE : Derlenen .asm sonucu oluşan .lst uzantılı dosyanın satır sütun genişliğini belirler
- TITLE : Oluşan .lst için her sayfa başına yazılacak başlığı belirler

# Sözde Komutlar

- SEGMENT/ENDS : Kesim tanımlaması için kullanılır

## a) Hizalama :

- a) BYTE : Kesim sıradaki adresten başlar
- b) WORD : Kesim sıradaki çift adresten başlar
- c) PARA : Kesim sıradaki 16'nın tam katından başlar
- d) PAGE : Kesim sıradaki 256'nın tam katından başlar

## b) Birleştirme:

- a) PUBLIC : Aynı isimli kesimlerin peşpeşe yerleşmesini sağlar
- b) COMMON : Aynı isimli kesimlerin aynı adresten başlamasını sağlar
- c) STACK : Yığın mantığında (LIFO) kesim tanımlar
- d) AT ##### : Kesim adresi belirler



# Sözde Komutlar

- ORG : COM tipi programların 100H adresinden başlaması için kullanılır
- ASSUME : Kesim tanımlarının ne amaçla yapıldığı ve başlangıç adreslerini belirler

# Sözde Komutlar

- DB : Define byte → 00H-0FFH arası değerleri tanımlar

Sayi1 DB 25

Sayi2 DB 0FFH

Dizi DB 1, 2, 3, 50H

Str DB 'Assembly'

# Sözde Komutlar

- DW : Define word → 0000H-0FFFFH arası değerleri tanımlar

Dizi DW 5, 10, 20

	7	0	
0510H	05H		← Dizi w
0511H	00H		
0512H	0AH		
0513H	00H		
0514H	14H		
0515H	00H		

- DD : Define double word → 00000000H-0FFFFFFFFFH arası değerleri tanımlar
- DQ : Define quad word → 8 byte bellek alanı ayırmak için kullanılır

# Sözde Komutlar

- EQU : Değişken tanımlamaz, derleme öncesi tüm EQU komutları uygun sabit değer ile değiştirilir
- DUP : Tekrarlı veri tanımı için kullanılır
  - Dizi1 DB 15 DUP (0) ; 15 adet değeri 0 olan byte ayırır
  - Matris DB 3 DUP (4 DUP (8)) ; 3x4 adet değeri 8 olan byte ayırır
  - Dizi2 DW 10 DUP (?) ; 10 adet word ayırır, ilk değer olarak o anki ; bellek içeriğini kullanır

# Sözde Komutlar

- TYPEDEF : Veri tipi tanımlama

integer TYPEDEF WORD

...

i integer 9

- PTR : Bellek alanı erişim veri tipini belirler

WORD PTR

BYTE PTR

FAR PTR

NEAR PTR

# Sözde Komutlar

- LABEL : Veri tanımına farklı tip ile erişim için kullanılır

Yenib LABEL BYTE

Sayıw DW 2535H

Yeniw LABEL WORD

Sayib DB 45H

Sayic DB 55H

MOV AX, Yeniw

MOV BL, Yenib

0000H	35H	← Sayıw	← Yenib
0001H	25H		
0002H	45H	← Sayib	← Yeniw
0003H	55H	← Sayic	
0004H	..		

# Sözde Komutlar

- PROC/ENDP
- EXTRN
- PUBLIC

# Sözde Komutlar

- MACRO/ENDM
- INCLUDE
- LOCAL



# Sözde Komutlar

- LENGTH : DUP ile yapılan veri tanımının boyutunu verir

Tablo DW 15 DUP(0)

MOV AX, LENGTH Tablo ;  $AX \leftarrow 15$

- TYPE : Değişkenin (dizi ise herbir elemanın) kaç byte yer kapladığını verir

MOV BX, TYPE Tablo ;  $BX \leftarrow 2$

- SIZE : Değişken tanımı için toplam kaç byte yer ayrıldığını verir

MOV DX, SIZE Tablo ;  $DX \leftarrow 2 \times 15$

# Sözde Komutlar

- OFFSET : Derleme öncesi değişken için ofset değeri döndürür (LEA program çalışırken ofset değeri döndürür)
- SEG : Etiketin içinde yer aldığı kesim değerini verir
- END : İlk çalıştırılacak yordamı belirler

# ALT SEVİYE PROGRAMLAMA

Hafta 5

Dr. Öğr. Üyesi Erkan USLU

# Assembly Program Tipleri

- Assembly dilinde programlar
  - Kodlama şekli
  - Kesim tanımlamaları
  - Kesim düzenine

göre iki farklı şekilde yazılabilir.

- Derlenen programın uzantısına göre isimlendirme yapılır:
  - EXE
  - COM

EXE Tipi Programlar

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'
```

```
    DW 32 DUP(?)
```

```
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'
```

```
SAYI DB ?
```

```
ELEMAN DW ?
```

```
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'
```

```
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR
```

```
    PUSH DS
```

```
    XOR AX,AX
```

```
    PUSH AX
```

```
    MOV AX, DATASG
```

```
    MOV DS, AX
```

```
    RETF
```

```
BASLA ENDP
```

```
CODESG ENDS
```

```
    END BASLA
```

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

Stack Segment



```
DATASG SEGMENT PARA 'DATA'  
SAYI DB ?  
ELEMAN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX
```

```
    MOV AX, DATASG  
    MOV DS, AX
```

```
    RETF  
BASLA ENDP  
CODESG ENDS  
END BASLA
```

# EXE Tipindeki Programlar

STACKSG SEGMENT PARA STACK 'STACK'

DW 32 DUP(?)

STACKSG ENDS

DATASG SEGMENT PARA 'DATA'

SAYI DB ?

ELEMAN DW ?

DATASG ENDS

CODESG SEGMENT PARA 'CODE'

ASSUME CS:CODESG, DS:DATASG, SS:STACKSG

BASLA PROC FAR

PUSH DS

XOR AX,AX

PUSH AX

MOV AX, DATASG

MOV DS, AX

RET

BASLA ENDP

CODESG ENDS

END BASLA

Stack Segment  
Adı



# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'  
SAYI DB ?  
ELEMAN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX
```

```
    MOV AX, DATASG  
    MOV DS, AX
```

```
    RETF  
BASLA ENDP  
CODESG ENDS  
END BASLA
```

Segmentin Stack  
Segment  
olduğunu belirler

# EXE Tipindeki Programlar

STACKSG SEGMENT PARA STACK 'STACK'

DW 32 DUP(?)

STACKSG ENDS

DATASG SEGMENT PARA 'DATA'

SAYI DB ?

ELEMAN DW ?

DATASG ENDS

CODESG SEGMENT PARA 'CODE'

ASSUME CS:CODESG, DS:DATASG, SS:STACKSG

BASLA PROC FAR

PUSH DS

XOR AX,AX

PUSH AX

MOV AX, DATASG

MOV DS, AX

RETF

BASLA ENDP

CODESG ENDS

END BASLA

DW kullanılmalı  
veya çift sayı ile  
DB kullanılmalı

# EXE Tipindeki Programlar

STACKSG SEGMENT PARA STACK 'STACK'

DW 32 DUP(?)

STACKSG ENDS

DATASG SEGMENT PARA 'DATA'

SAYI DB ?

ELEMAN DW ?

DATASG ENDS

CODESG SEGMENT PARA 'CODE'

ASSUME CS:CODESG, DS:DATASG, SS:STACKSG

BASLA PROC FAR

PUSH DS

XOR AX,AX

PUSH AX

MOV AX, DATASG

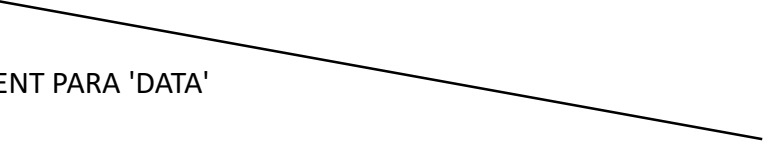
MOV DS, AX

RET

BASLA ENDP

CODESG ENDS

END BASLA



Segmentin  
bitişini işaret  
eder

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'
```

```
    DW 32 DUP(?)
```

```
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'
```

```
SAYI DB ?
```

```
ELEMAN DW ?
```

```
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'
```

```
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR
```

```
    PUSH DS
```

```
    XOR AX,AX
```

```
    PUSH AX
```

```
    MOV AX, DATASG
```

```
    MOV DS, AX
```

```
    RETF
```

```
BASLA ENDP
```

```
CODESG ENDS
```

```
    END BASLA
```

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'  
SAYI  DB ?  
ELEMAN DW ?  
DATASG ENDS
```

Data Segment



```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA  PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX
```

```
    MOV AX, DATASG  
    MOV DS, AX
```

```
    RETF  
BASLA  ENDP  
CODESG ENDS  
END BASLA
```

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'  
SAYI  DB ?  
ELEMAN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA  PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX
```

```
    MOV AX, DATASG  
    MOV DS, AX
```

```
    RETF  
BASLA  ENDP  
CODESG ENDS  
END BASLA
```

Değişken  
tanımları

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'
```

```
    SAYI DB ?  
    ELEMAN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR
```

```
    PUSH DS  
    XOR AX,AX  
    PUSH AX
```


```
    MOV AX, DATASG  
    MOV DS, AX
```

```
    RETF
```

```
BASLA ENDP
```

```
CODESG ENDS
```

```
END BASLA
```



Data segment  
ismi

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'
```

```
    DW 32 DUP(?)
```

```
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'
```

```
SAYI DB ?
```

```
ELEMAN DW ?
```

```
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'
```

```
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR
```

```
    PUSH DS
```

```
    XOR AX,AX
```

```
    PUSH AX
```

```
    MOV AX, DATASG
```

```
    MOV DS, AX
```

```
    RETF
```

```
BASLA ENDP
```

```
CODESG ENDS
```

```
    END BASLA
```



# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'  
SAYI DB ?  
ELEMAN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG  
  
BASLA PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX  
  
    MOV AX, DATASG  
    MOV DS, AX  
  
    RETF  
BASLA ENDP  
CODESG ENDS  
END BASLA
```



Code Segment

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'  
SAYI DB ?  
ELEMAN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX  
  
    MOV AX, DATASG  
    MOV DS, AX  
  
    RETF  
BASLA ENDP
```

```
CODESG ENDS  
    END BASLA
```

EXE programda  
en az 1 tane FAR  
tipinde prosedür  
olmalı

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'  
SAYI DB ?  
ELEMEN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR
```

```
PUSH DS  
XOR AX,AX  
PUSH AX
```

```
MOV AX, DATASG  
MOV DS, AX
```

```
RETF
```

```
BASLA ENDP  
CODESG ENDS  
END BASLA
```

EXE'yi çağıran  
kodun kesim  
değeri DS'de,  
offset değeri 0,  
dönüş için bu  
değerler stack'e  
atılır

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

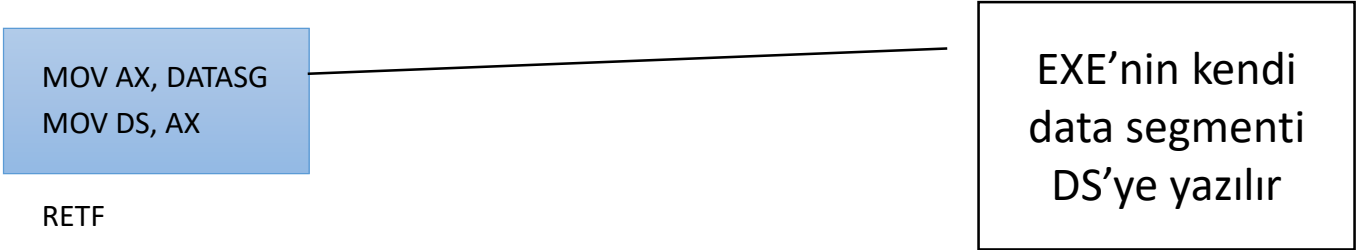
```
DATASG SEGMENT PARA 'DATA'  
SAYI DB ?  
ELEMAN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX
```

```
    MOV AX, DATASG  
    MOV DS, AX
```

```
    RETF  
BASLA ENDP  
CODESG ENDS  
    END BASLA
```



EXE'nin kendi  
data segmenti  
DS'ye yazılır

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'  
SAYI DB ?  
ELEMAN DW ?  
DATASG ENDS
```

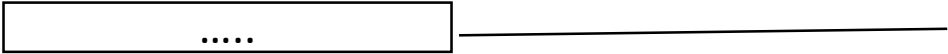
```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX
```

```
    MOV AX, DATASG  
    MOV DS, AX
```

```
    RETF
```

```
BASLA ENDP  
CODESG ENDS  
    END BASLA
```



Assembly  
kodunu  
yazacağımız blok

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'  
    DW 32 DUP(?)  
STACKSG ENDS
```

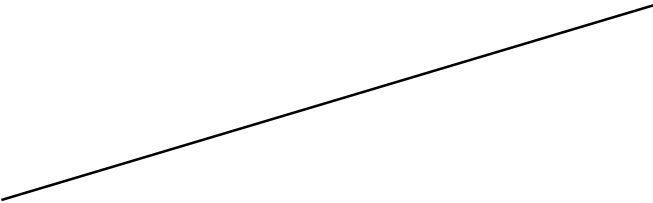
```
DATASG SEGMENT PARA 'DATA'  
SAYI DB ?  
ELEMEN DW ?  
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'  
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR  
    PUSH DS  
    XOR AX,AX  
    PUSH AX
```

```
    MOV AX, DATASG  
    MOV DS, AX
```

```
    RETF  
BASLA ENDP  
CODESG ENDS  
END BASLA
```



EXE  
başlatıldığında  
çağırılacak  
yordamı ve CS'yi  
belirler

# EXE Tipindeki Programlar

```
STACKSG SEGMENT PARA STACK 'STACK'
```

```
    DW 32 DUP(?)
```

```
STACKSG ENDS
```

```
DATASG SEGMENT PARA 'DATA'
```

```
SAYI DB ?
```

```
ELEMAN DW ?
```

```
DATASG ENDS
```

```
CODESG SEGMENT PARA 'CODE'
```

```
    ASSUME CS:CODESG, DS:DATASG, SS:STACKSG
```

```
BASLA PROC FAR
```

```
    PUSH DS
```

```
    XOR AX,AX
```

```
    PUSH AX
```

```
    MOV AX, DATASG
```

```
    MOV DS, AX
```

```
    RETF
```

```
BASLA ENDP
```

```
CODESG ENDS
```

```
    END BASLA
```

PAGE 60,80

TITLE ornek\_\_\_\_\_

STEK SEGMENT PARA STACK 'STK'

DW 30 DUP(?)

STEK ENDS

DSG SEGMENT PARA 'DTS'

SAYI DB ?

ELEMAN DW 10

DIZI DW 12 DUP(0)

DIZI2 DD 10 DUP(?)

DSG ENDS

CSG SEGMENT PARA 'CODE'

ASSUME CS:CSG, DS:DSG, SS:STEK

BASLA PROC FAR

PUSH DS

XOR AX,AX

PUSH AX

MOV AX, DSG

MOV DS, AX

RETF

BASLA ENDP

CSG ENDS

END BASLA



COM Tipi Programlar

# COM Tipi Programlar

```
CODESG SEGMENT PARA 'CODE'
```

```
    ORG 100H
```

```
    ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG
```

```
BASLA  PROC NEAR
```

```
    RET
```

```
BASLA  ENDP
```

```
SAYIB  DB ?
```

```
SAYIW  DW ?
```

```
CODESG ENDS
```

```
    END BASLA
```

# COM Tipi Programlar

```
CODESG SEGMENT PARA 'CODE'  
    ORG 100H  
    ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG  
  
BASLA  PROC NEAR  
  
        RET  
BASLA  ENDP  
  
SAYIB  DB ?  
SAYIW  DW ?  
CODESG ENDS  
END BASLA
```

COM programlar  
için tek bir  
segment tanımı  
yapılır; data,  
stack ve code  
segment olarak  
aynı segment  
kullanılır

# COM Tipi Programlar

```
CODESG SEGMENT PARA 'CODE'
```

```
    ORG 100H
```

```
    ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG
```

```
BASLA PROC NEAR
```

```
    RET
```

```
BASLA ENDP
```

```
SAYIB DB ?
```

```
SAYIW DW ?
```

```
CODESG ENDS
```

```
    END BASLA
```

COM programların ilk 256 byte'lık kısımları header bilgisi içerir. Assembly komutlarının 100H'tan başlayabilmesi için ORG 100H kullanılır

# COM Tipi Programlar

```
CODESG SEGMENT PARA 'CODE'
```

```
    ORG 100H
```

```
    ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG
```

```
BASLA  PROC NEAR
```

```
    RET
```

```
BASLA  ENDP
```

```
SAYIB  DB ?
```

```
SAYIW  DW ?
```

```
CODESG ENDS
```

```
    END BASLA
```

COM program  
tek segment  
kullandığı için en  
az 1 tane near  
prosedür  
içermesi gerekir

# COM Tipi Programlar

```
CODESG SEGMENT PARA 'CODE'
```

```
    ORG 100H
```

```
    ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG
```

```
BASLA  PROC NEAR
```

```
    RET
```

```
BASLA  ENDP
```

```
SAYIB  DB ?
```

```
SAYIW  DW ?
```

```
CODESG ENDS
```

```
END BASLA
```



Değişken  
tanımları

# COM Tipi Programlar

CODESG SEGMENT PARA 'CODE'

ORG 100H

ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG

BASLA PROC NEAR

RET

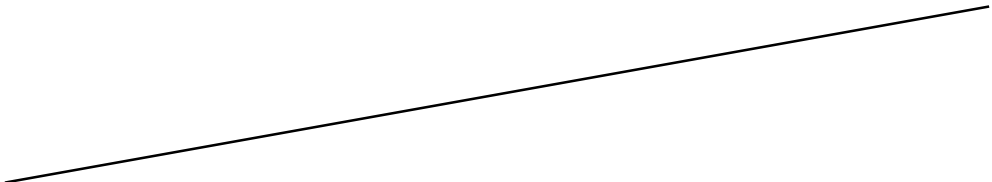
BASLA ENDP

SAYIB DB ?

SAYIW DW ?

CODESG ENDS

END BASLA



COM program  
çalıştırıldığında  
çalıştırılacak  
yordamı belirtir

# COM Tipi Programlar

```
CODESG SEGMENT PARA 'CODE'
```

```
    ORG 100H
```

```
    ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG
```

```
BASLA  PROC NEAR
```

```
    ....
```

```
    RET
```

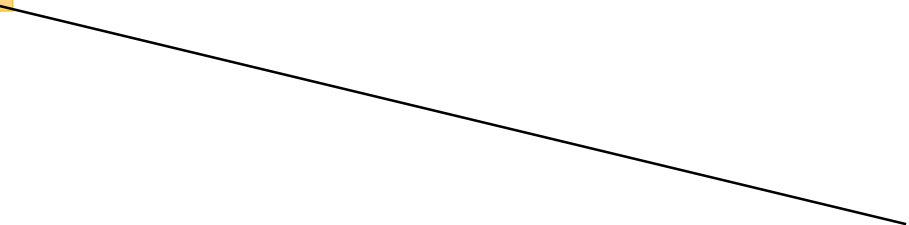
```
BASLA  ENDP
```

```
SAYIB  DB ?
```

```
SAYIW  DW ?
```

```
CODESG ENDS
```

```
    END BASLA
```



Assembly  
kodunu  
yazacağımız blok



# COM Tipi Programlar

```
CODESG SEGMENT PARA 'CODE'
```

```
    ORG 100H
```

```
    ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG
```

```
BASLA  PROC NEAR
```

```
    RET
```

```
BASLA  ENDP
```

```
SAYIB  DB ?
```

```
SAYIW  DW ?
```

```
CODESG ENDS
```

```
    END BASLA
```

PAGE 60,80

TITLE ornek\_\_\_\_\_

CODESG SEGMENT PARA 'CODE'

ORG 100H

ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG

BASLA PROC NEAR

RET

BASLA ENDP

SAYIB DB ?

SAYIW DW ?

CODESG ENDS

END BASLA

COM Şablon 1

PAGE 60,80

TITLE ornek\_\_\_\_\_

CODESG SEGMENT PARA 'CODE'

ORG 100H

ASSUME CS:CODESG, SS:CODESG, DS:CODESG, ES:CODESG

BILGI: JMP BASLA

SAYIB DB ?

SAYIW DW ?

BASLA PROC NEAR

RET

BASLA ENDP

CODESG ENDS

END BILGI

COM Şablon 2

# COM EXE Karşılaştırma

	COM Programı	EXE Programı
Kullanılabilen Bellek	64KB - Header – Dönüş adresi	Boş bellek
Başlangıçta CS	Header'ın bulunduğu kesim	END komutundan sonraki etiketin bulunduğu kesim
Başlangıçta IP	100H	END komutundan sonraki etiketin bulunduğu ofset
Başlangıçta DS	Header'ın bulunduğu kesim	Dönüş için gerekli kesim değeri
Değişkenlere Erişim İçin Yapılması Gerekenler	Yok	MOV AX, verikesimismi MOV DS, AX
Değişkenlerin Tanımlandığı Yer	ENDP'den sonra	Veri kesiminde

# COM EXE Karşılaştırma

	COM Programı	EXE Programı
Başlangıçta SS	Header'ın bulunduğu kesim	Yığın kesimi adresi
Başlangıçta SP	0FFFEH	Yığın kesimi boyutu
Başlangıçta Yığın	0000H	Boş
Yığın Büyüklüğü	64KB – 256 (header) – X byte Kod – Y byte Değişken	Yığın kesimi boyutu
Ana Prosedür Tipi	NEAR	FAR
Dönüş İçin Yapılması Gereken	Yok	PUSH DS XOR AX, AX PUSH AX

# ALT SEVİYE PROGRAMLAMA

Hafta 11

Dr. Öğr. Üyesi Erkan USLU

PROSEDÜR VE MAKRO

# Prosedür (Yordam) - Makro

- Modüler programlama yaklaşımı
  - Giriş ve çıkış değerleri tanımlanmış modüller
  - Sürekliliği sağlama
  - Kod tekrar kullanımı
- 
- Prosedür: Çalışma sırasında çağrılan fonksiyonlar
  - Makro: Derleme sırasında yerine kopyalanan modüller



# Prosedür Tanımı

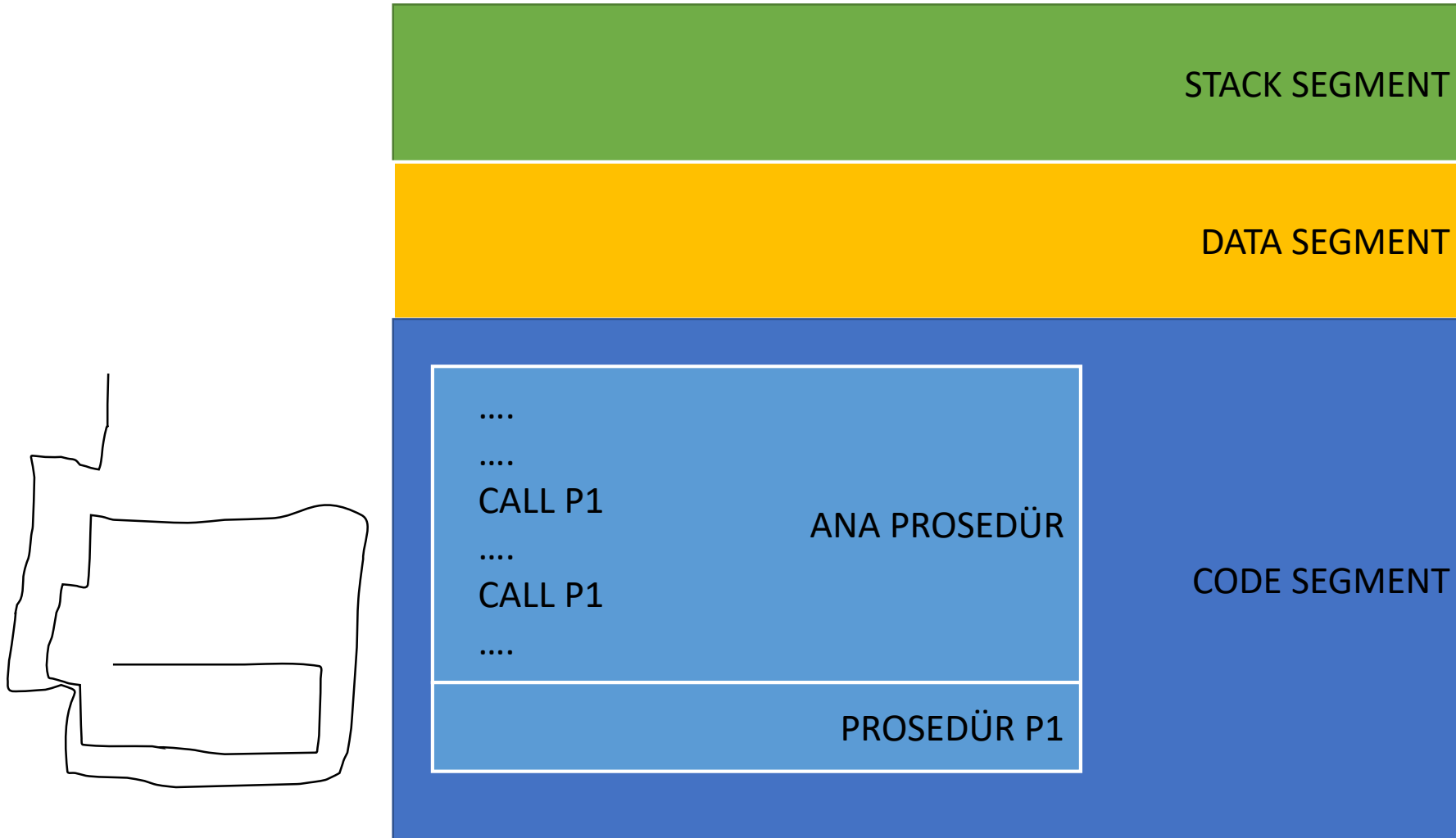
*Yordam\_ismi* PROC {NEAR|FAR}

*yordam\_kodu*

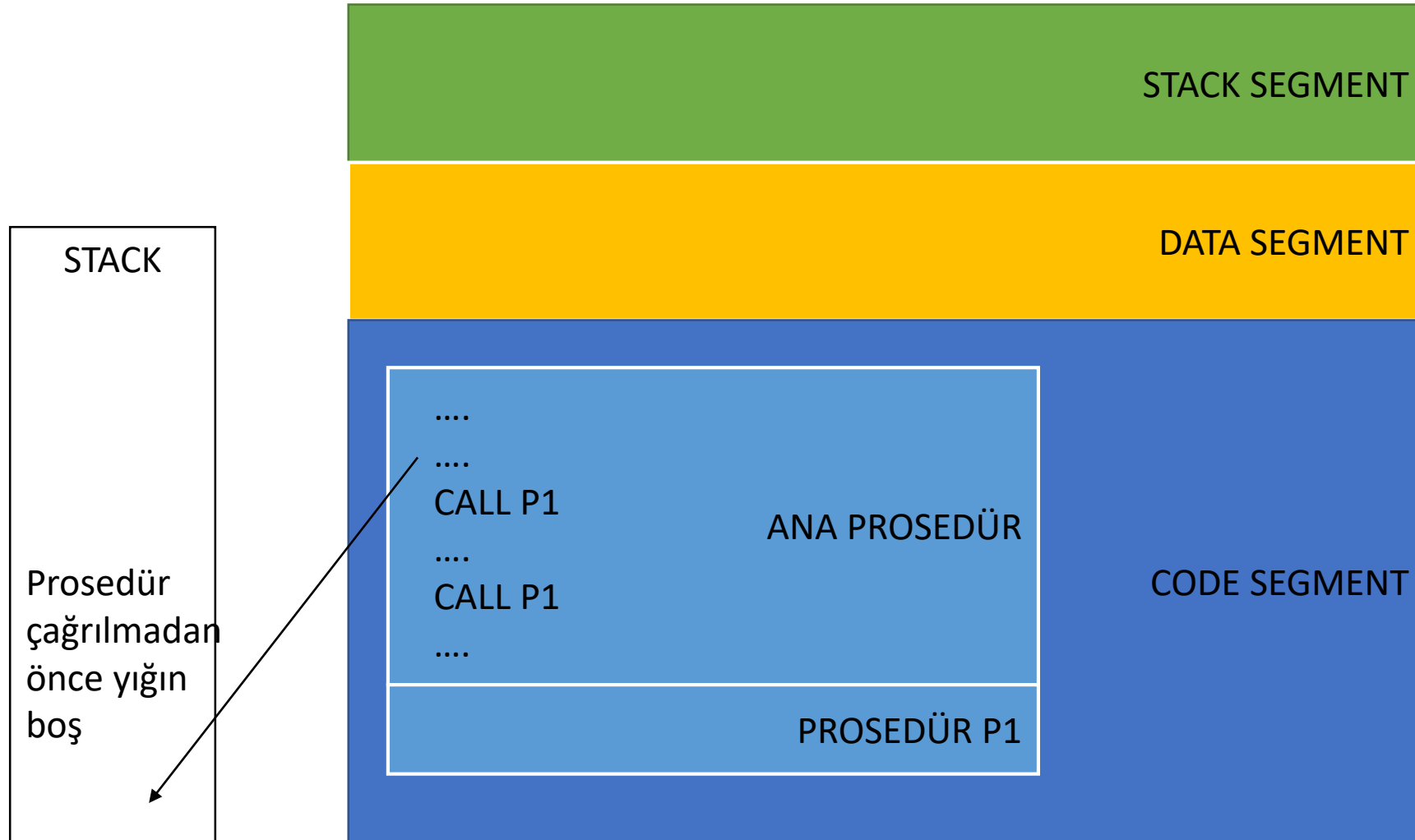
RET

*Yordam\_ismi* ENDP

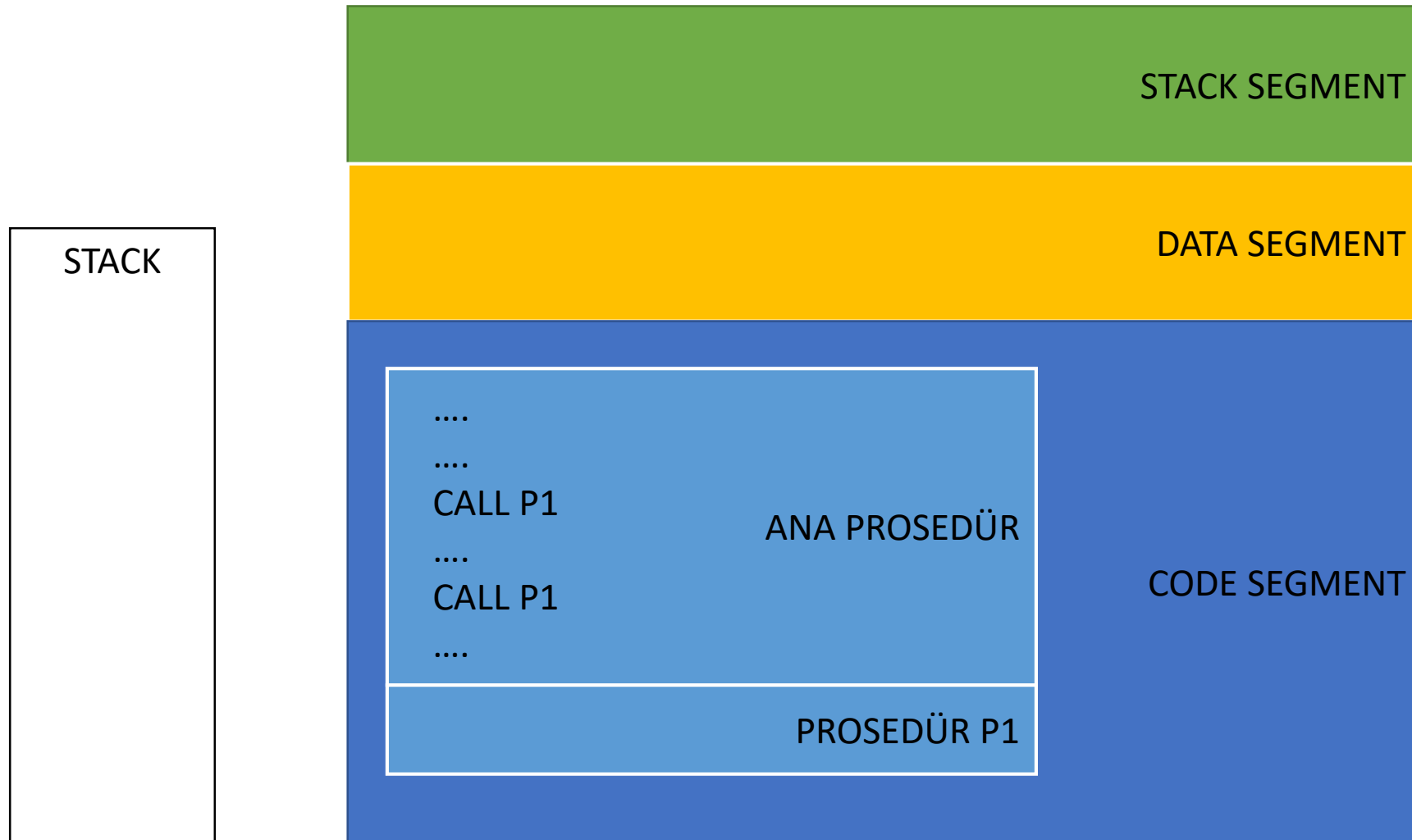
# Prosedür Çağırma



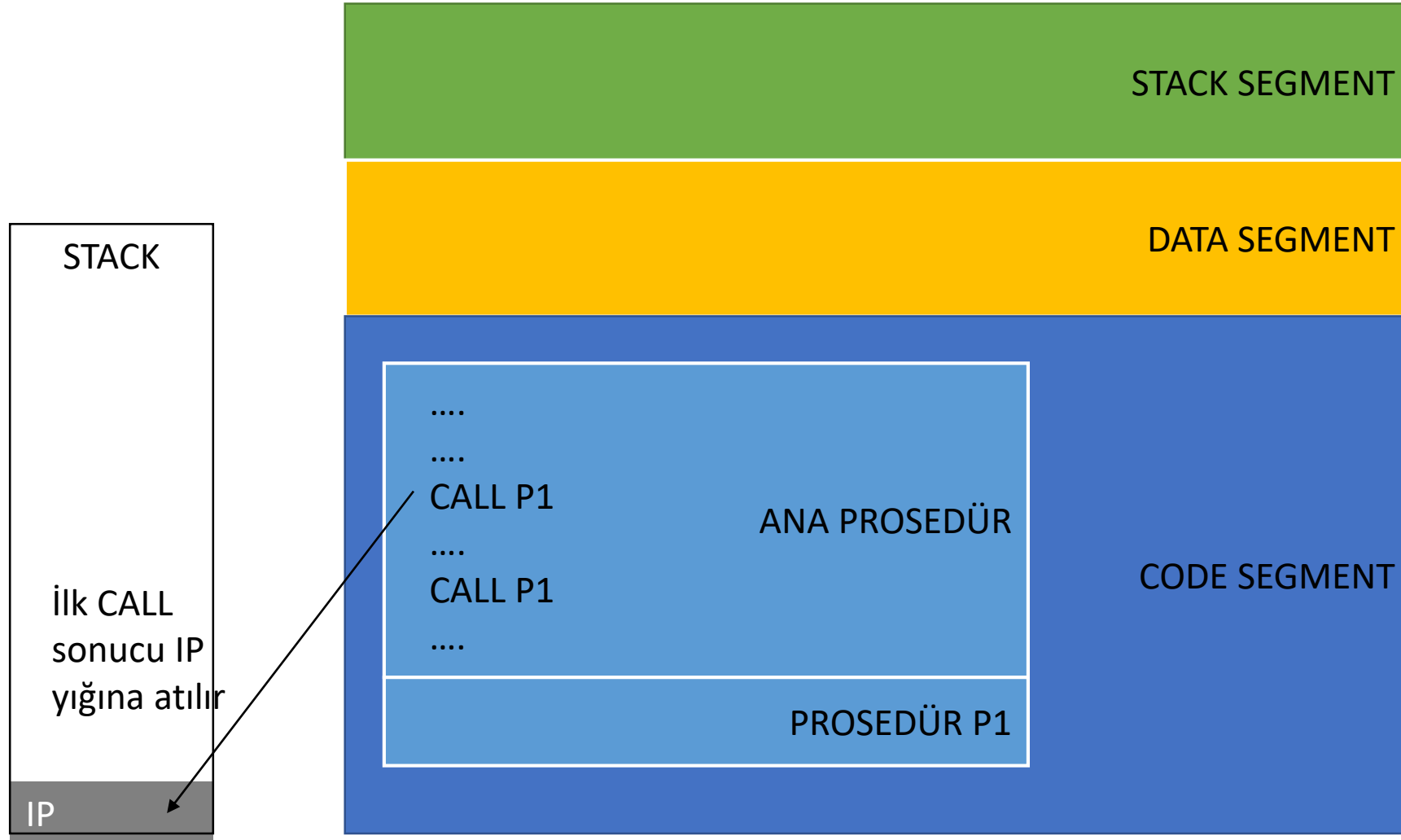
# NEAR Prosedür Çağırma



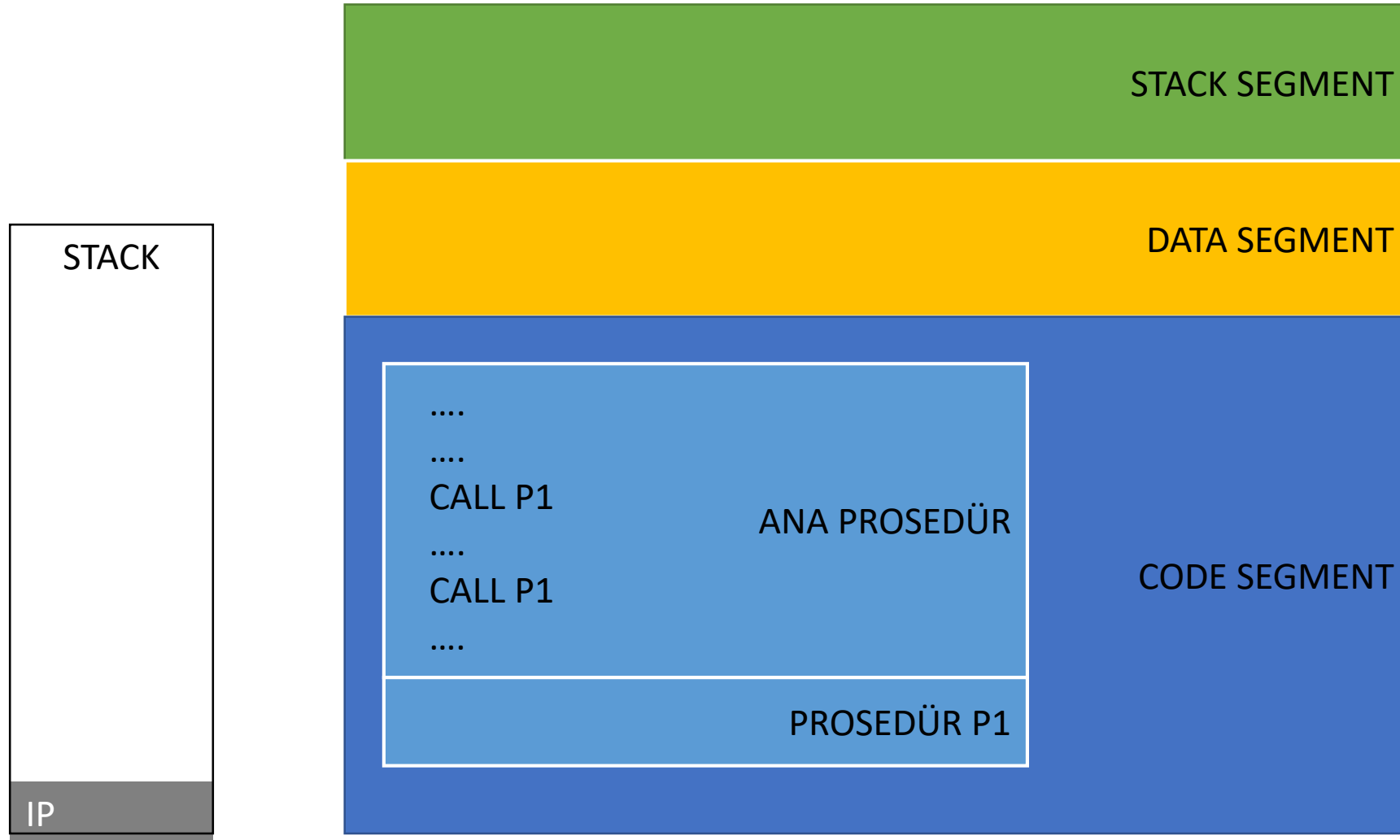
# NEAR Prosedür Çağırma



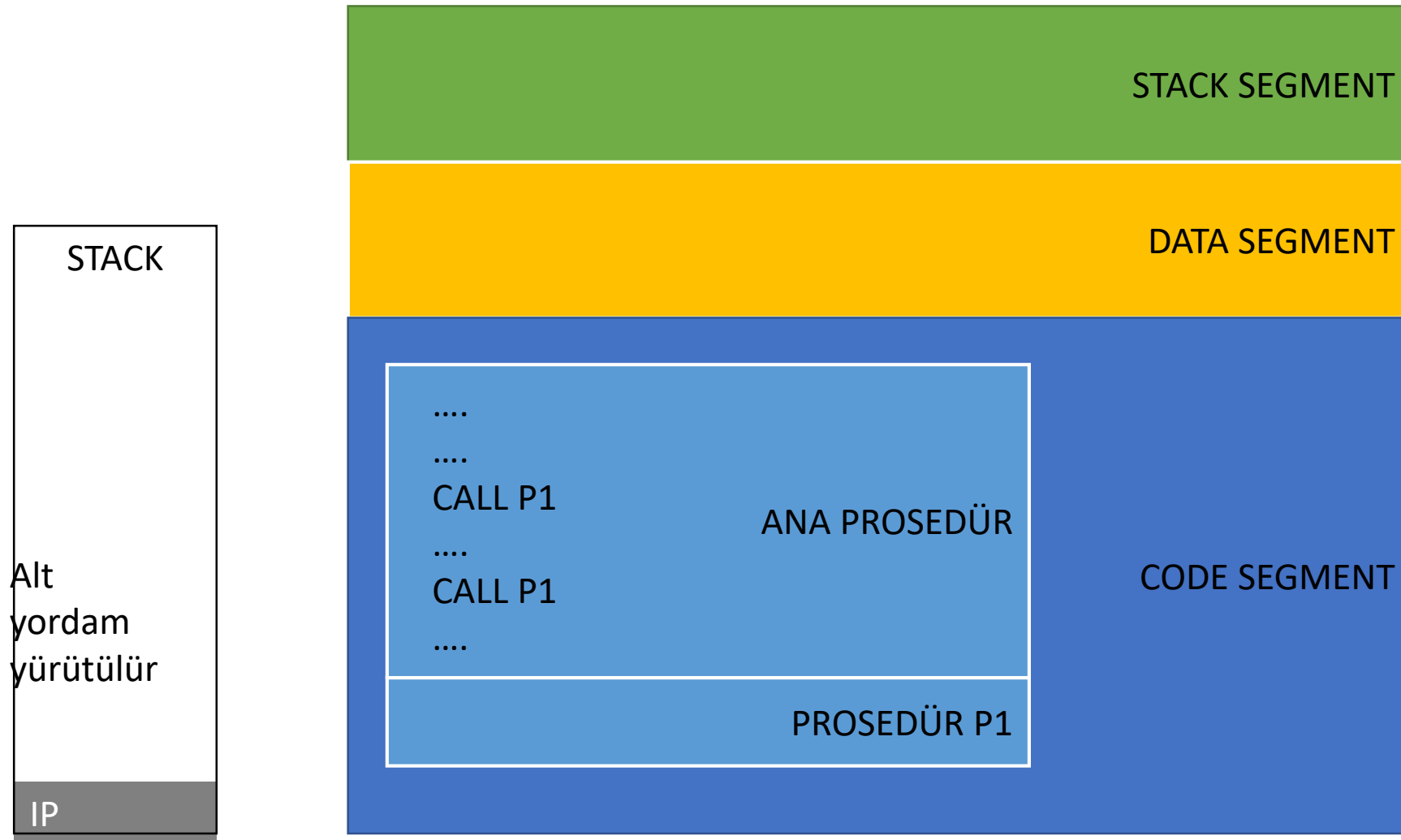
# NEAR Prosedür Çağırma



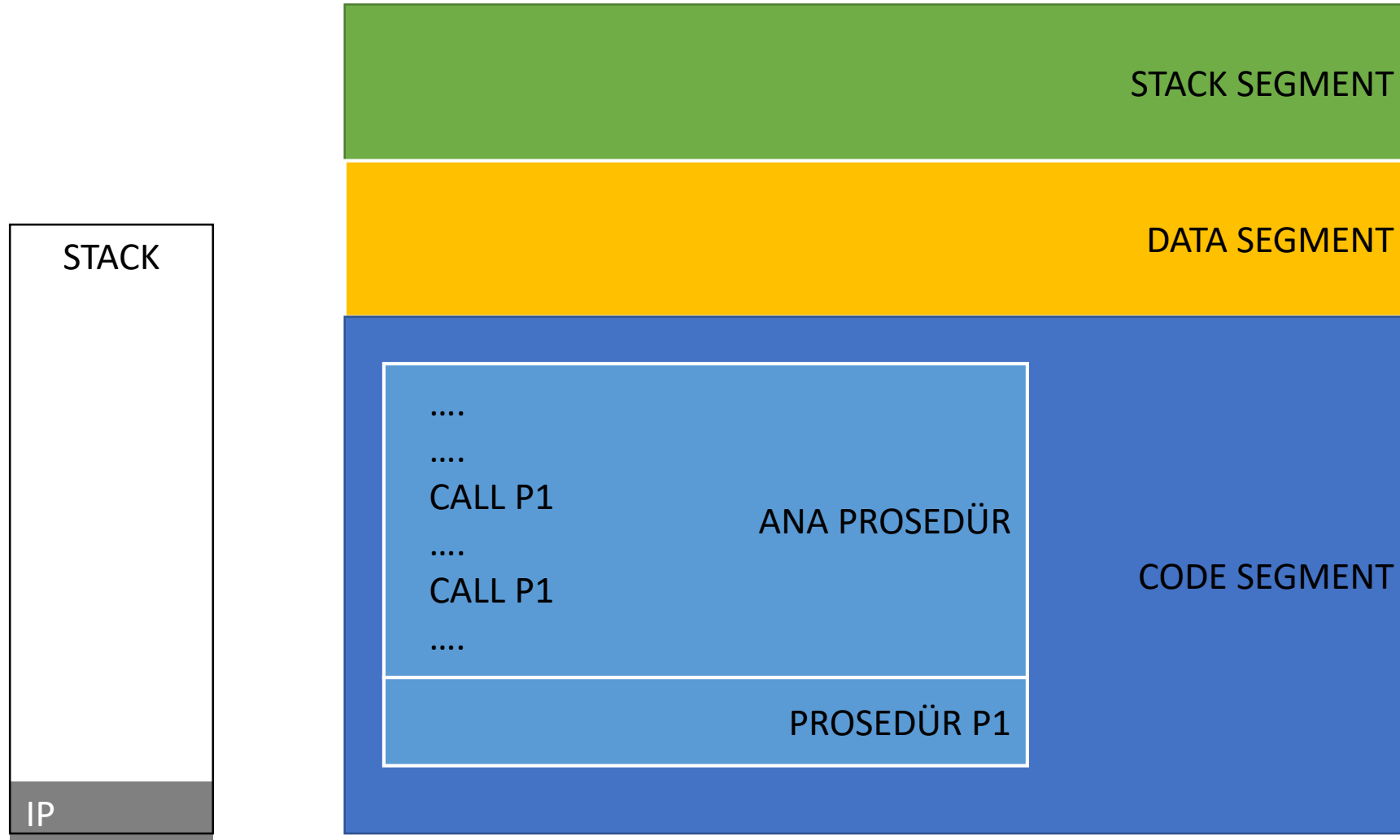
# NEAR Prosedür Çağırma



# NEAR Prosedür Çağırma

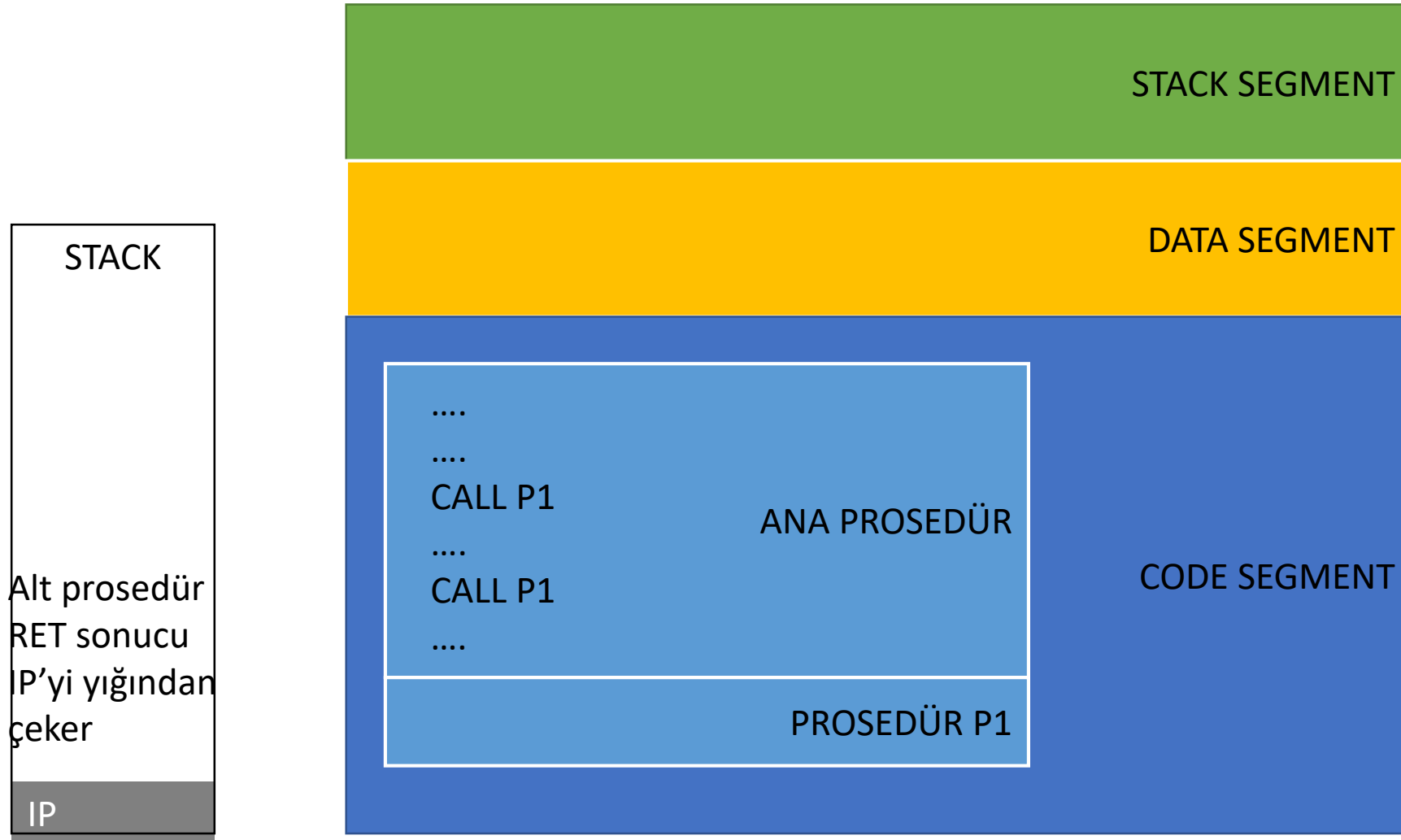


# NEAR Prosedür Çağırma

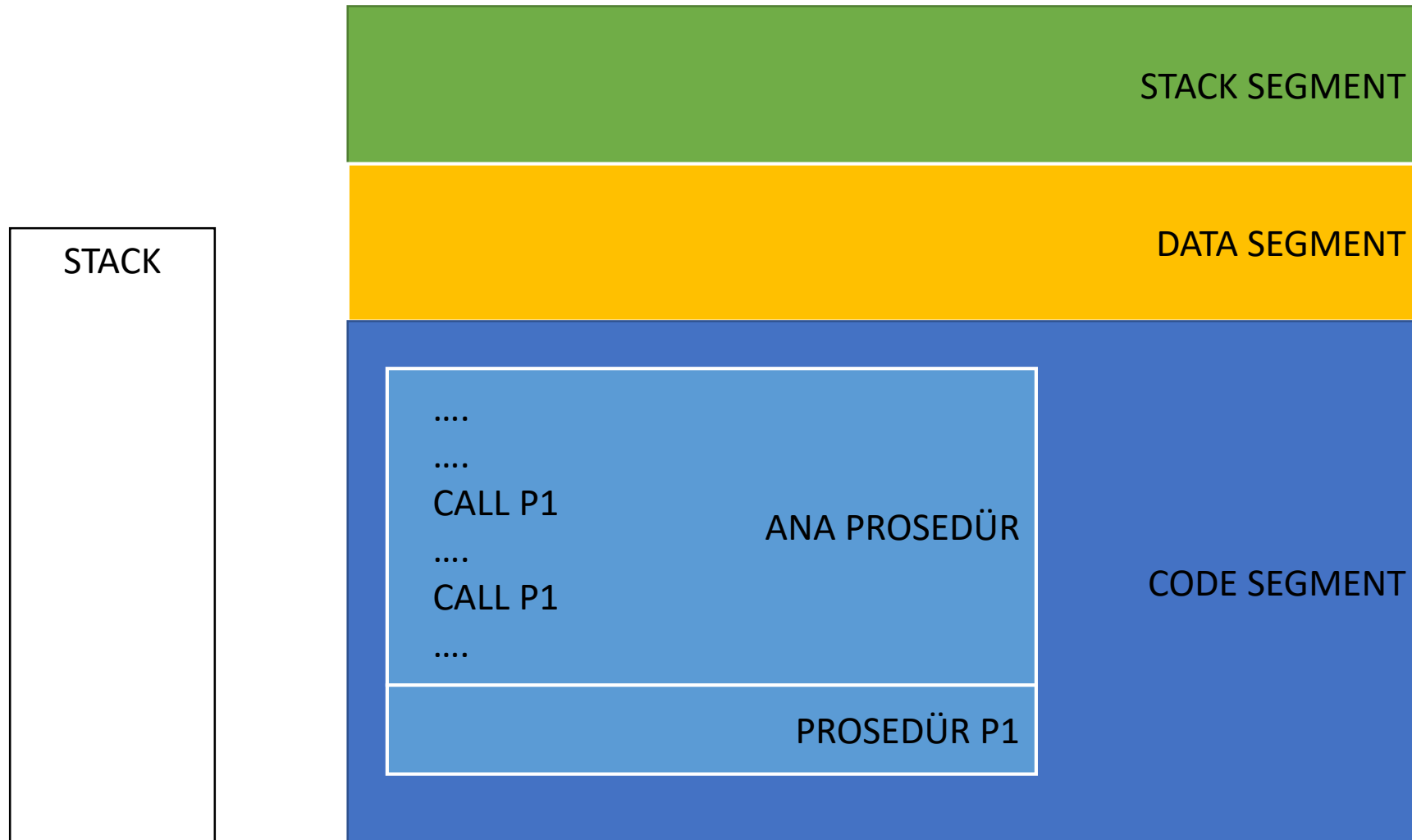




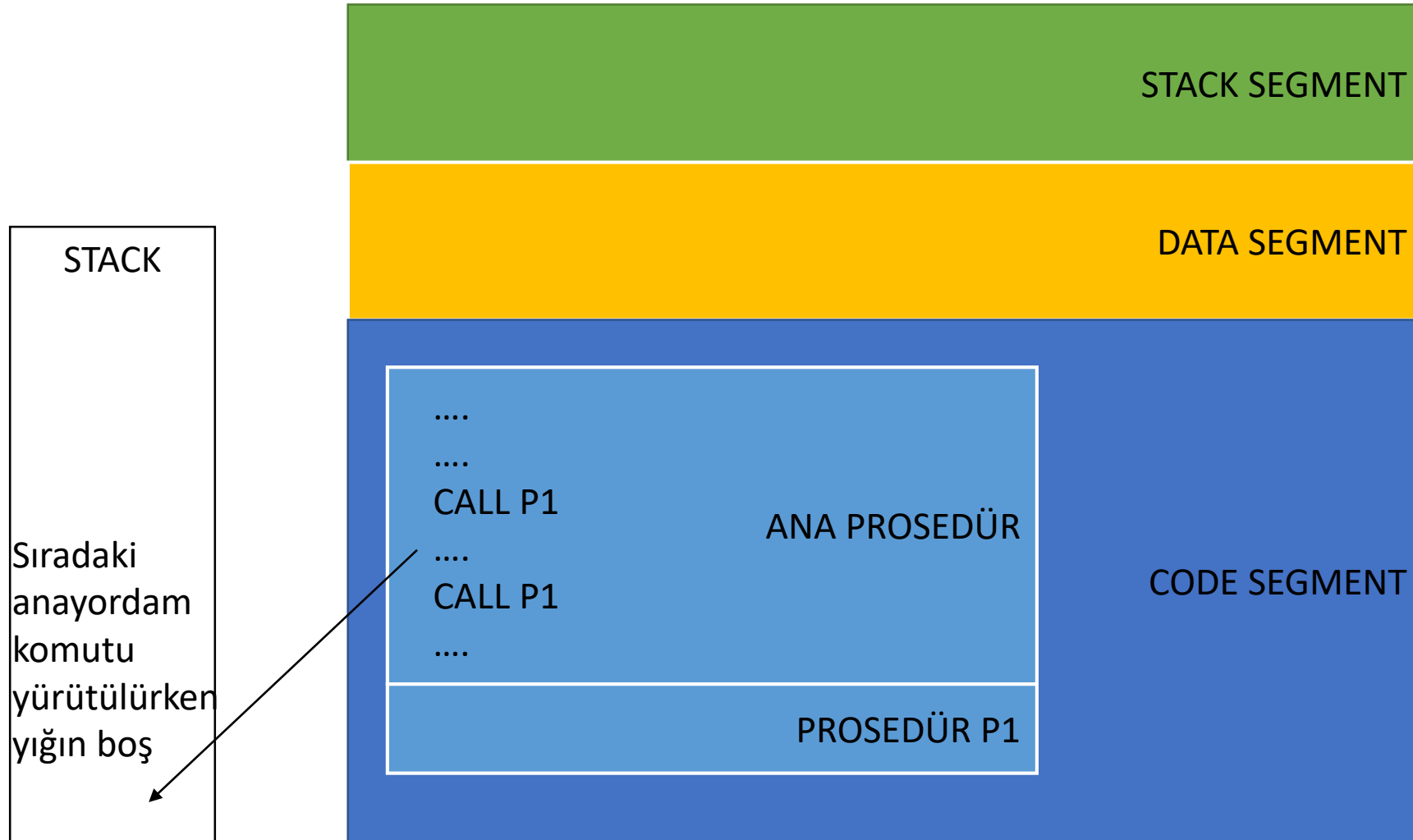
# NEAR Prosedür Çağırma



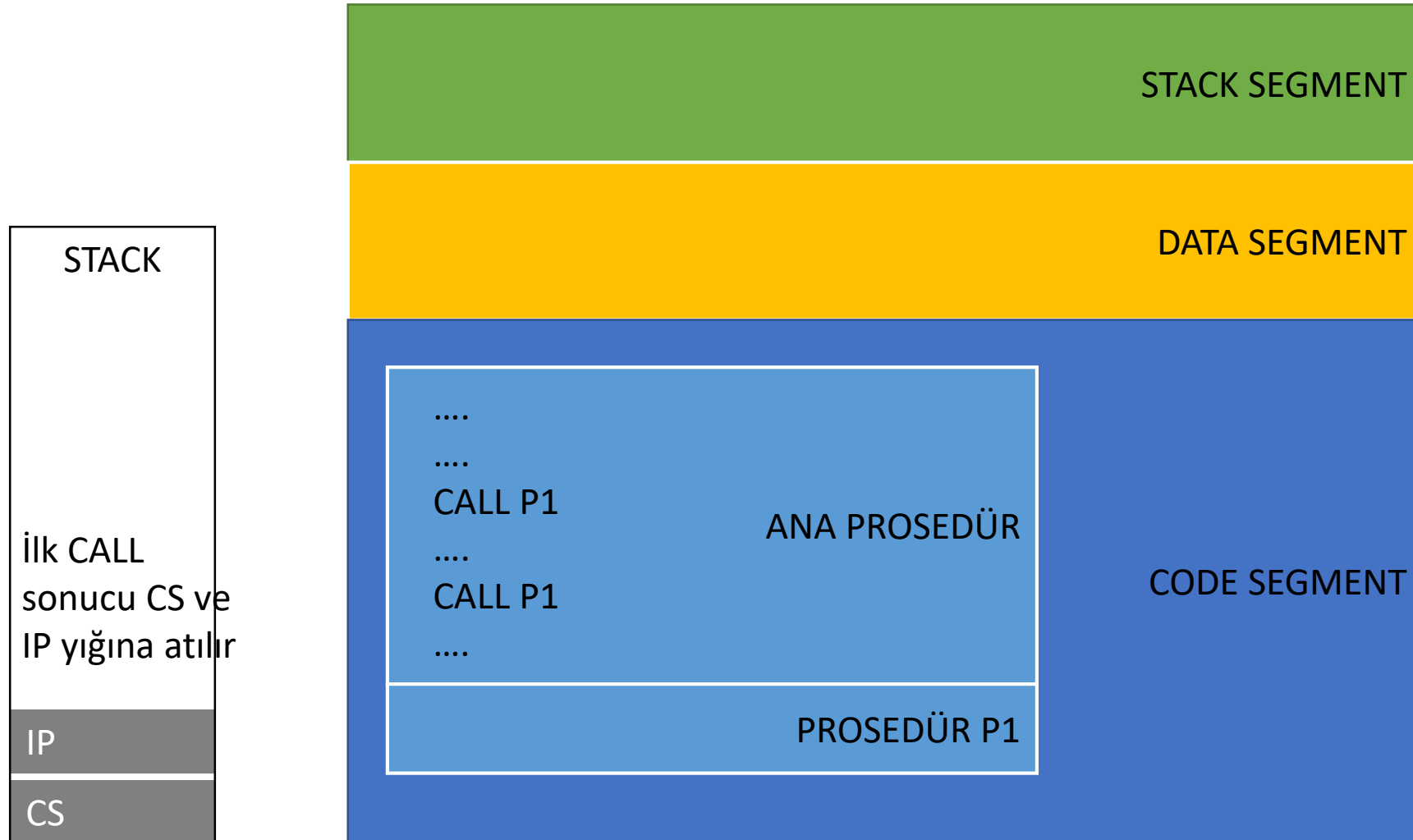
# NEAR Prosedür Çağırma



# NEAR Prosedür Çağırma



# FAR Prosedür Çağırma



# Makro Tanımı

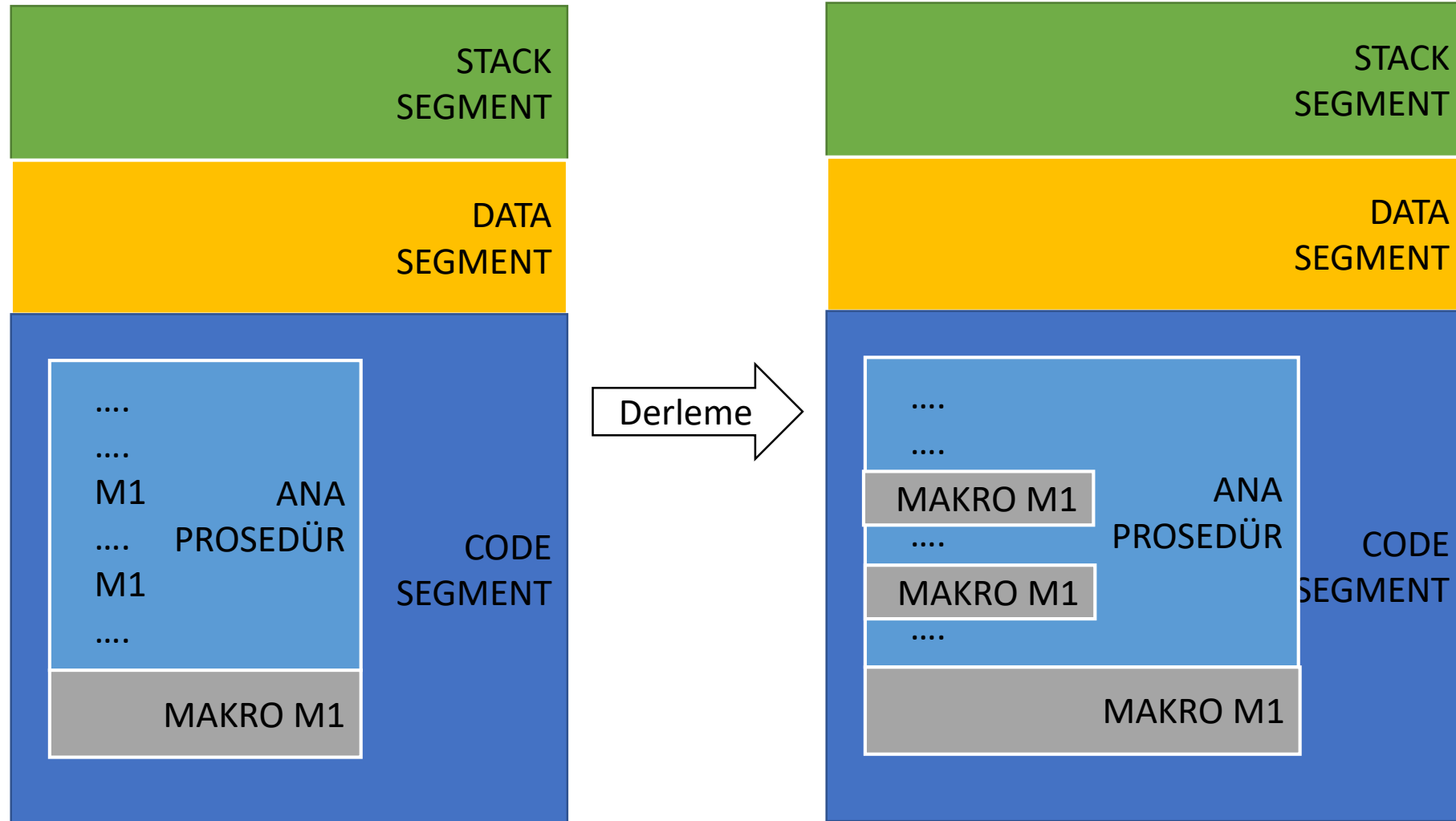
*Makro\_ismi* MACRO {*parametre\_listesi*}

LOCAL ...

*makro\_kodu*

ENDM

# Makro Çağırma



# ALT SEVİYE PROGRAMLAMA

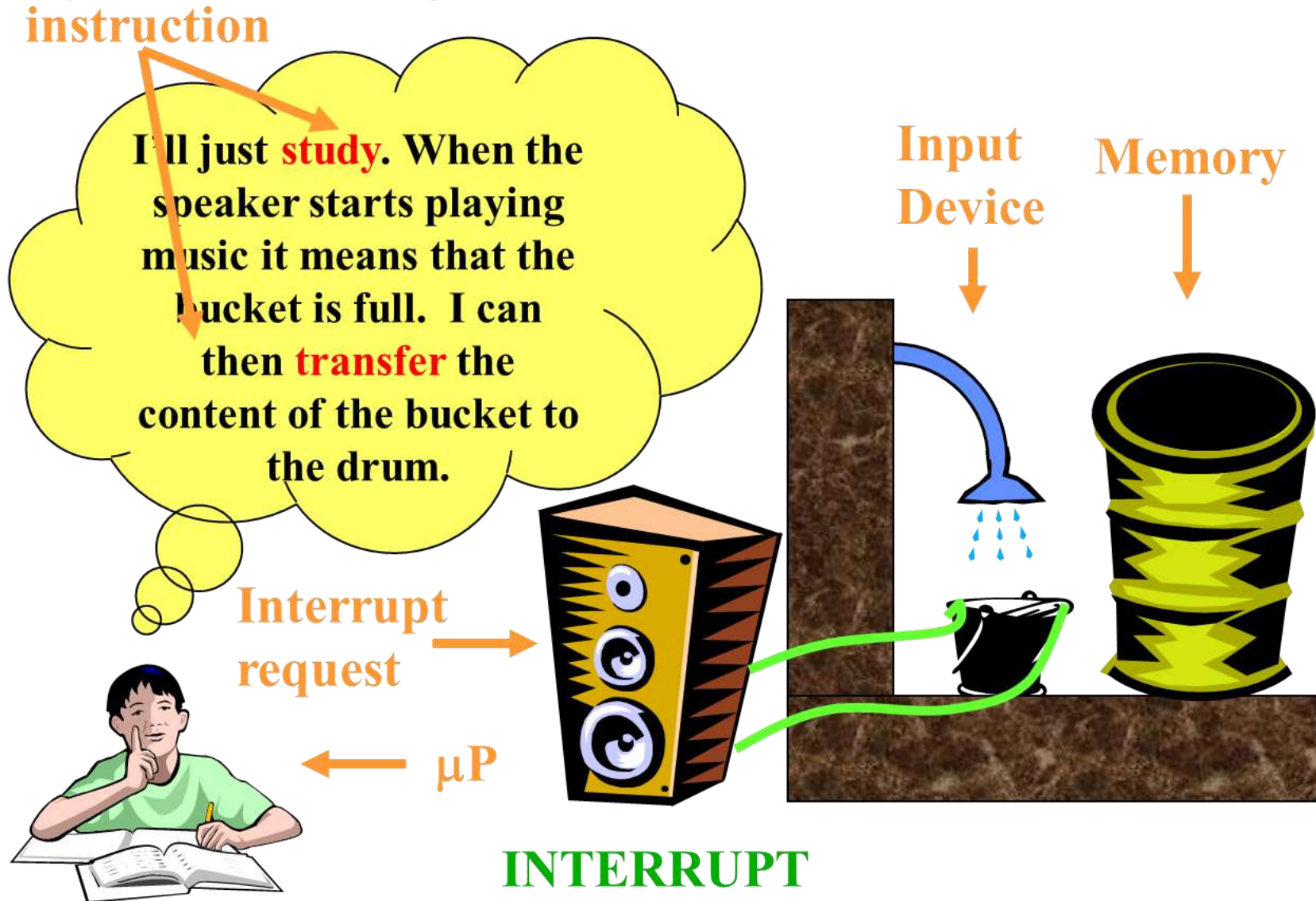
Dr. Öğr. Üyesi Erkan USLU

# Interrupt - Polling





# Interrupt - Polling



# Kesme Kaynakları

- Yazılımsal olarak: INT, INTO, INT 3
- Donanımsal olarak: INTR, NMI
  - Donanım kesmesi ile ilgili acknowledge:  $\overline{INTA}$
- Kesme ile ilgili bayraklar: IF, TF
- Kesme ile ilgili dönüş komutu: IRET

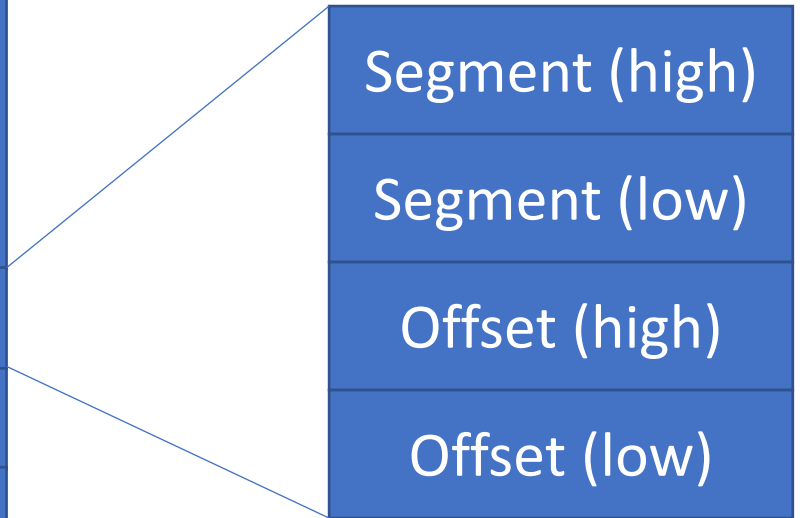
# Kesme Vektör Tablosu

- Kesme oluştuğunda mevcut CS:IP ile gösterilen kodun yürütülmesi bırakılarak, kesmeye ilişkin fonksiyonun yer aldığı kesim ve offset değerindeki kod işlenmelidir.
- Her kesmeye ilişkin fonksiyonların hangi hafıza adresinde yer aldığı Kesme Vektör Tablosu ile tutulur.

# Kesme Vektör Tablosu

- Kesme Vektör Tablosu hafıza uzayında 00000H-003FFH adres aralığındaki 1024 byte'lık alandır.
- Kesme Vektör Tablosu toplamda 256 farklı kesme için ilgili fonksiyonların offset ve kesim değerlerini saklar.

# Kesme Vektör Tablosu

3FFH	Available Interrupt Pointers		
080H			
014H	Reserved Interrupt Pointers		
010H			
00CH	Overflow (TiP4)		Segment (high)
008H	1 byte Breakpoint (TiP3)		Segment (low)
004H	NMI (TiP2)		Offset (high)
000H	Single Step (TiP1)		Offset (low)
	Divide By 0 (TiP0)		

# Kesme İşlemi

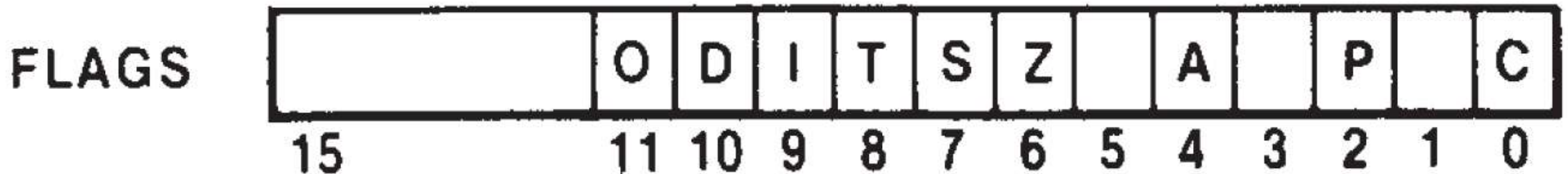
- Sıradaki işlem bittiğinde, işlemci:
  - Komut işleme ile oluşan, single-step, NMI, INTR, INT sırasıyla bir kesme olup olmadığını kontrol eder.
  - Kesme varsa
  - PUSHF
  - $IF \leftarrow 0$ ,  $TF \leftarrow 0$  (INTR, single-step kesmeleri engellenir)
  - PUSH CS
  - PUSH IP
  - $IP \leftarrow \text{Kesme\_vektör\_tablosu}$ ,  $CS \leftarrow \text{Kesme\_vektör\_tablosu}$

# Kesme İşlemi

- Kesme alt programı yürütülür
- Kesme alt programının sonunda IRET ile dönülür
- POP IP
- POP CS (Bazı kesmelerin dönüşü bir sonraki adrese bazısının dönüşü ise kesme oluşturan adressedir)
- POPF (Kesme öncesi  $IF \leftarrow 1$  olarak ayarlanmış olsa, kesme alt programı süresince  $IF \leftarrow 0$  yapılmıştı, kesme dönüşünde POPF ile otomatik olarak kesmeler açık)

# Kesme Bayrakları

- INTR ucu sadece IF=1 ise kesme oluşturabilir
- TF=1 ise her komut işlendikten sonra single-step kesmesi oluşur. (Debug amaçlı)
- **STI** : set interrupt flag, **CLI** : clear interrupt flag, TF için özel komut yok





# Kesme Vektör Tablosunu Değiştirme

- Örnek: DIV0 kesme alt programının adresini değiştirme

# ALT SEVİYE PROGRAMLAMA

Hafta 14

Dr. Öğr. Üyesi Erkan USLU

# YÜKSEK SEVİYELİ DİLLER VE ASSEMBLY BAĞLANTISI

# Yüksek Seviyeli Dil - Assembly

- Inline
  - Yüksek seviyeli dil içerisinde assembly kodlarının yerleştirilmesi
- Object Code
  - Derlenen assembly programından oluşan obje'nin yüksek seviyeli dil ile kodlanan program içerisinden çağırılması

# Yüksek Seviyeli Dil - Assembly

- Yüksek seviyeli dillerden assembly yordamlarını çağırma ✓
- Assembly programlarında yüksek seviyeli dillerde yazılmış yordamları çağırma ✓

# C'de Assembly Yordamı Çağırma

## **YAZ.C**

```
1    #include <stdio.h>
2    main()
3    {
4        printf(' 1+ 2 = %d \n ', add(1,2));
5    }
```

# C'de Assembly Yordamı Çağırma

**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'de Assembly Yordamı Çağırma

- C'de alt yordama aktarılacak parametreler yığında ters sırada yerleştirilir.
- Parametre listesinde en sağdaki parametre yığına ilk atılır.
- Parametre listesinde en soldaki parametre yığına son atılır.
- C büyük küçük harf duyarlıdır.
- Asm'de tanımlı yordam C'deki tanımıyla büyük küçük harf olarak uyumlu olmalıdır.



# C'de Assembly Yordamı Çağırma

- Asm'de C dilinden kullanılacak her türlü etiket ve değişken için başta \_ (underscore) kullanılır.
- C'de yığına yerleştirilen değerlerin kaldırılmasından çağıran sorumludur.
- Asm'den dönen değer AX yazmacı üzerinden C'ye döndürülür.

# Assembly'de C Yordamı Çağırma

# Alt Seviye Programlama BLM2021

Öğr. Grv. Furkan ÇAKMAK





# Ders Tanıtım Formu ve Konular

BLM2021  
Alt Seviye  
Programlama

Hafta 13

Hafta	Tarih	Konular
1	08.10.2020	Alt seviye dilinin özellikleri, sayı ve kodlama sistemleri, 80x86 ailesi işlemcileri, yazmaçları ve bayrakları ile kesim organizasyonu
2	15.10.2020	Komutlar (veri aktarımı, aritmetik ve dallanma)
3	22.10.2020	Komutlar (çevrim, bayraklar, mantıksal, öteleme, döndürme)
4	29.10.2020	Cumhuriyet Bayramı
5	05.11.2020	Komutlar (katar işlemleri, ön ekler)
6	12.11.2020	Adresleme modları, alt seviye programlama araçları, sözde komutlar
7	19.11.2020	Çalışma ortamının hazırlanması ve debug kullanımı
8	26.11.2020	EXE tipinde alt seviye programlama
9	03.12.2020	COM tipinde alt seviye programlama
10	10.12.2020	Yordam ve makro kullanımları
11	17.12.2020	Alt-programlar ve parametre aktarma yöntemleri
12	24.12.2020	Ortak kesim kullanımı ve EXTRN/PUBLIC tanımlamaları
13	31.12.2020	Kesme, vektör tablosu
14	07.01.2020	Alt seviye programlama dilinin yüksek seviyeli diller ile birlikte kullanılması



# Kesme (Interrupt) Nedir?

- Çeşitli kriterler ve önceliklerle program işleyişlerinin kesintiye uğratılarak eş zamanlı çalışmanın sağlanması interrupt mekanizması ile sağlanır.
- Yavaş birimler ile hızlı birimler arasındaki bağlantıyı sağlarken zaman kaybetmeyi engellemek için kullanılır.
- Çevre birimleri işlemciden daha yavaş çalışır.
  - Mouse - Klavye
  - Yazıcı
  - Monitör
- Diğer birimler
  - RAM



# İşlemci ile Yavaş Birimler Arasındaki İlişki

BLM2021  
Alt Seviye  
Programlama

Hafta 13

- Busy waiting
- Polling
- Interrupt
  - Zaman kesmesi
  - Grafik kesmesi





# Kesme Çeşitleri

BLM2021  
Alt Seviye  
Programlama

Hafta 13

- Temel kesmeler
  - Donanım Kesmeleri
  - Yazılım Kesmeleri
- Interrupt Handler
- Dahili kesmeler
  - Logical interrupt
  - Sıfıra bölme
  - Adım adım çalıştırma
  - Vb.
- Harici kesmeler
  - Maskelenemez Kesmeler (NMI)
  - Maskelenebilir Kesmeler (INTR)
    - IF durumu önemli !
    - 8259A bütünleşik entegresi üzerinden sürdürülür.





# Kesme Öncelikleri

BLM2021  
Alt Seviye  
Programlama

Hafta 13

Öncelik	Kesme	Açıklama
1	<i>Divide by Zero</i>	<i>Hatana neden olan program sonlandırılır. Sistem güvenliğini yitirir. Tekrar başlatılması gerekebilir.</i>
2	<i>INT #</i>	<i>Yazılım olarak numarası verilen kesmenin çağırılması</i>
3	<i>INTO</i>	<i>Aritmetik işlemin (MUL/IMUL) sonucunu değerlendirmek içindir. JO/JNO koşullu dallanma komutları da bu amaçla kullanılır.</i>
4	<i>NMI</i>	<i>INT 02H – Bellek üzerinde oluşan kritik hatalarda kullanılır.</i>
5	<i>INTR</i>	<i>8259A entegresi üzerinde oluşan donanım kesmeleridir.</i>
6	<i>Single step</i>	<i>INT 01H programın adım adım çalışmasını sağlayan kesmedir.</i>



# Kesme Oluştuğunda Yapılan İşlemler

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 çalıştırılır (# x 4 sonucunda vektör tablosundaki adrese erişilir.)
    - Alınan ilk word değeri IP yazmacına,
    - İkinci word değeri CS yazmacına yerleştirilir.
- Kesme servis programını yazan;
    - Yazmaçlarla ilgili durumları değerlendirmeli,
    - IRET komutu ile kesmeden dönmelidir.



# Vektör Tablosunun Görevi ve Konumu

- Kesme servis programları gerek BIOS gerekse İşletim Sistemi tarafından sağlanan değişik uzunluktaki kod parçalarıdır.
- Bilgisayar açılışında belleğe yerleştirilirler.
- Farklı bellek alanlarında bulunan bu kodlara erişmek için Vektör Tablosu kullanılır.
- Vektör Tablosunun tek görevi kesmelerin başlangıç adreslerini tutmaktır.
- 00000H - 003FFH fiziksel adresleri arasında bulunan 1024 byte'lık alanda bulunur.
- Her servisin adresi 2 word'le ifade edilir (Offset ve kesim adresi).
- Yani  $1024 / 4 = 256$  tane kesme vardır.
- Kullanıcılar kesme yazacaklarsa;
  - INT 60H - INT 67H arası bu iş için ayrılmıştır.
  - Mevcut olan servisin yerine yeni bir kesme yazılabilir.



# Vektör Tablosunda Değişiklik Nasıl Yapılır?

- Programcı kendi kesmesini yazabilir.
- Var olan bir kesmeden hemen önce devreye girebilecek bir kesme yazabilir.
- Her iki durumda da yazılan kesme vektör tablosuna yerleştirilmelidir.
- Bunun için yapılması gereken işlemler;
  1. Kesmenin TSR ile belleğe yerleştirilmiş olması gerekir. (TSR -> Terminate and Stay Resident)
  2. Yazılan kodun adresinin (offset + kesim) Vektör Tablosunda ilgili alanlara yazılması gerekir (Kesmeler disable edilmelidir. Edilmezse ?)
  3. Değiştirilecek kesmenin adresi saklanmalıdır.
  4. INT 21H - Fonksiyon 15H (Set Interrupt Vector) kullanılarak yazılan kesme kodunun adresi vektör tablosuna yazılmalıdır.
  5. Kesmelere izin verilmelidir.
    1. Kendi kesmemizi yazmışsak IRET ile dönmeliyiz.
    2. Orijinal kesmeden önce çalışmışsak JMP ile orijinal kesme adresine gitmeli ve akışa oradan devam etmeliyiz.
  6. Kullanım tamamlandıktan sonra orijinal kesmenin adresi vektör tablosuna yazılmalıdır.



# Interrupt Servis Programı Yazarken Nelere Dikkat Edilmelidir?

BLM2021  
Alt Seviye  
Programlama

Hafta 13

- Kesme kodu içerisinde girildiğinde kesmelere izin verilmelidir.
- Kesme içerisinde yazmaçların bir önceki değerlerinin korunması gerekmektedir.
- Kesme programı 1'den fazla kesim kullanacak ise bu değerler kesmeye gitmeden önce ilgili yazmaçlara yüklenmelidir.
- Donanım kesmesi yerine (BIOS) geçecek kesme yazılacaksa İşletim Sistemine ait (DOS) hiçbir fonksiyon kullanılmamalıdır.
- Kesme programı TSR ile belleğe yerleştirilmesi gerekmektedir.
- Yığına veri göndermişsek onları kesme içerisinde temizleyecek şekilde hareket sergilemeliyiz.



# Kesmeleri Kullanırken

- 00H-1FH arasında BIOS kesmeleri,
- 20H-2FH arasında DOS kesmeleri bulunur.
- Kesmeler çağırılırken,
  - Fonksiyon numarası AH'a,
  - Eğer varsa alt fonksiyon numarası AL'ye aktarılır.
  - INT komutu kullanılır.
- Örnek;
  - MOV AH, 1H
  - INT 21H



# Sabırla Dinlediğiniz İçin Teşekkürler

BLM2021  
Alt Seviye  
Programlama

Hafta 13





## Assembly Dili'nin Yüksek Seviyeli Bir Dil Olan C İle Bağlantısı

Hazırlayanlar: Arş. Grv. Furkan ÇAKMAK (furkan@ce.yildiz.edu.tr, Yrd. Doç. Dr. Ahmet Tevfik İNAN (tevfik@ce.yildiz.edu.tr)

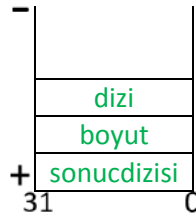
Yıldız Teknik Üniversitesi, Bilgisayar Mühendisliği Bölümü  
(http://www.ce.yildiz.edu.tr)  
İstanbul

Bu kılavuzda Assembly programlama dilinin yüksek seviyeli bir dil olan C ile nasıl birlikte kullanılacağı detaylı ve uygulamalı olarak anlatılmıştır. Uygulamalar farklı platformlar kullanılarak ve farklı örnek problemler ile gerçekleştirilmiştir. Zaman zaman anlatımı kolaylaştırması için ekran görüntülerine, yığın organizasyonlarına ve program parçacıklarına yer verilmiştir.

### Uygulama Öncesi Temel Bilgiler

- Assembly yordamlarının C fonksiyonu ile benzer kullanılabileceği gibi aynı dosya içerisinde hem assembly hem de C kodu yazılabilmektedir. Ancak platform farklılıklarından kaynaklanan yazım değişiklikleri bilinmelidir.
- C ve Assembly dillerinde yazılan kodların aynı tipteki büyüklükte (32-bit, 64-bit, vb.) derlenmesi veya en azından üretecekleri nesne dosyalarının aynı tipte olması sağlanmalıdır.
- Eğer bu kılavuzda ilk iki uygulamasında olduğu gibi fonksiyon kullanımı tercih edilecekse parametre aktarmanın ne şekilde gerçekleştirileceği önem arz etmektedir.
  - C fonksiyonundan Assembly yordamı çağırıldığında aktarılan parametreler; aktarım sırasının tam tersi olacak şekilde (sağdan sola doğru) yığına atılırlar. Yığından aldığı verileri işleyen Assembly yordamı eğer tek bir değer döndürecekse akümülatör yazarı (kullanılan platforma göre değişiklik göstermekle birlikte AX (16-bit), EAX (32-bit) veya RAX (64-bit) üzerinden değer döndürmektedir. Örnek 1'de integer'ın 4 byte olduğu bir platform için yazılmış bir C fonksiyonu prototipi ve bu prototipin çağırılması sonucu parametrelerin ne şekilde yığına atılacağı gösterilmiştir. Fonksiyon bir tamsayı değeri döndürecek için bu da EAX yazarı üzerinden gerçekleştirilecektir.

**Örnek 1:** `int Toplama(int* dizi, int boyut, int* sonucdizisi);`



Şekil 1: Örnek 1 Fonksiyonunun Çağırılması Sonucu Parametrelerin Yığına Konulma Sırası

- Çağırılan yordamların “near” tipinde olduğu bilinmeli ve yığına bu sebeple bir göreceli konum (offset) değerinin atıldığı unutulmamalıdır.
- C programlama dilinin yapısı gereği yığın üzerinden parametre olarak gönderilen değerler, yine C tarafında yığından kaldırılmaktadır. Bu sebeple Assembly yordamından dönerken bu parametrelerin kaldırılması ile ilgili herhangi bir işlem yapılmamalıdır.
- Farklı platformlarda değişiklikler göstermekle birlikte unutulmamalıdır ki; bir assembly program (COM tipinde hepsi bir arada olsa bile) veri, yığın ve kod kesimlerinden oluşur. Eğer bir assembly yordamı, bir C fonksiyonu tarafından çağırıldıysa, yığın olarak çağırılan programın

(yani C programının) yığını kullanılır. Eğer yordamda veri kullanımına ihtiyaç duyulmuşsa, ihtiyaç duyulan veriler, veri kesiminde tanımlandıktan sonra kod kesiminde kullanılabilir.

- Programlar içerisinde kullanılan değişkenler, bellekte birer adresi göstermektedir. Bu sebeple inline assembly kodu yazarken C değişkenleri olduğu gibi kullanılabilir (okunup, yazılabilir). Yani ortak veri kesimi kullanımı gerçekleştirilmektedir.
- Assembly yordamına int'den büyük değerler (double vb.) gönderilmek istenirse, unutulmamalıdır ki, veri int'den büyük tanımlı olsa bile o verinin bellekteki adresi bir int boyutundadır. Bu da, istenilen her büyüklükteki verinin (kaynakların kapasitesi göz önünde bulundurularak) assembly yordamına gönderilebilmesine imkan sağlamaktadır.

## 1 - Linux (Ubuntu) Ortamında Assembly ve C Dillerinin Birlikte Kullanımı

Bu uygulamada farklı bir assembler olan **nasm** (The Netwide Assembler <http://www.nasm.us/>) kullanılarak assembly kodları derlenmiş, ardından C nesne kodları ile birlikte linklenerek çalıştırılabilir dosya üretilmiştir.

Öncelikle kodlama yapılacak dosyaların bulunacağı bir klasör Masaüstü'nde oluşturulmuştur.

```
$ mkdir ~/Dekstop/casm
```

Ardından ornek1.c ve ornek1.asm dosyaları oluşturulmuştur.

```
$ cd ~/Dekstop/casm
$ > ornek1.c
$ > ornek1.asm
```

Bu örnekte C programında tanımlanmış **n** elemanlı bir dizi içerisinde bulunan değerlerin çağırılan bir Assembly yordamı tarafından toplanarak sonucunun döndürülmesi sağlanmıştır. Öncelikle C kodlarını yazmak için gedit uygulamasıyla C dosyasını açalım.

```
$ gedit ornek1.c
```

Yazılan C dosyası aşağıda verildiği gibidir.

```
ornek1.c
#include <stdio.h>
#include <stdlib.h>
extern int Topla(int *, int);

int main(){
    int i, top, n;
    printf("Kaca kadar sayilari toplamak istiyorsunuz: ");
    scanf("%d", &n);
    int dizi[n];
    for(i = 0; i < n; i++)
        dizi[i] = i + 1;
    top = Topla(dizi, n); //Assembly yordam çağırma satırı
    printf("Toplam: %d\n", top);
    system("PAUSE");
    return 0;
}
```

"ornek1.c" dosyası içerisinde "Topla" isimli bir assembly yordamı kullanılmıştır. Yordamın kodu bu dosya içerisinde olmadığından derleyiciyi uyarmak için yordamın prototipi yazılırken "extern" ön eki kullanılmıştır. Toplama işlemini gerçekleştirecek Assembly dosyası ornek1.asm ise;



```
$ gedit ornek1.asm
```

Koduyla oluşturulmuş ve içeriği aşağıdaki gibi yazılmıştır.

#### ornek1.asm

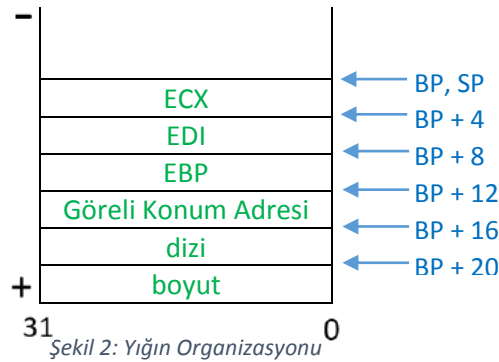
```
section .data

section .text

global Topla

Topla:
    PUSH EBP
    PUSH EDI
    PUSH ECX
    MOV EBP, ESP
    MOV EDI, [EBP+16] ; Dizinin adresine erisilir.
    MOV ECX, [EBP+20] ; Dizinin boyutu olan n degerine erisilir.
    XOR EAX, EAX
L1:   ADD EAX, [EDI]
    ADD EDI, 4 ; Dizi integer (4 byte) tanimli oldugu icin bir sonraki
    LOOP L1 ; elemana erisim icin 4 eklenir.
    POP ECX
    POP EDI
    POP EBP
    RET
```

Assembly kodu içerisinde veri tanımlanmadığı için data kısmı/kesimi boş bırakılmıştır. “Topla” yordamına ‘:’ kullanılarak başlanmış RET ifadesi ile de bitirilmiştir. Yordama girildiğinde öncelikle kullanılacak yazmaçlar yığına atılmış ve göreceli olarak yığın üzerinden gelen parametrelere erişildikten sonra işlem gerçekleştirilmiştir. İşlemin sonucunun akümülatör (EAX) yazmacı üzerinde oluşması sağlanmıştır. Çünkü daha önceden de bahsedildiği gibi RET komutuyla birlikte yordamdan C programına döndüğünde bulunan sonuç akümülatör yazmacı üzerinden beklenmektedir. Kodun daha iyi anlaşılabilmesi için yığın organizasyonu Şekil 2’de verilmiştir.



Kodları derlemek için bir “makefile” oluşturulmuştur.

```
$ gedit makefile
```

“makefile” dosyası içerisinde öncelikle Assembly source kodundan elf (Executable and Linkable Format) formatında obje kodu üretilmesi için;

```
nasm -f elf32 -o ornek1.o ornek1.asm
```

satırı yazılmıştır. Ardından C kodunu derleyecek ve Assembly kodu ile linkledikten sonra çalıştırılabilir dosya üretecek;

```
gcc -m32 ornek1.c ornek1.o -o combine_casm
```

kodu yazılmıştır.

64-bit'lik linux platform üzerinde gcc derleyicisinin nesne kodu üretmesi için m32 parametresi kullanılmıştır. Eğer sisteminizde yüklü değilse öncelikle;

```
$ sudo apt-get install gcc-multilib
```

komutu ile gcc derleyicisinin çoklu derleme opsiyonu yüklenmelidir. Bu sayede m32 parametresi kullanılabilir.

“nasm” ile Assembly kodunu derlerken kullanılan elf32 parametresi ise üreteceği nesne dosyasının 32-bit'lik olmasını sağlamak içindir.

Programların derlenmesi için ilgili dizine gidildikten sonra “make” komutu çalıştırılmalı, ardından oluşan combine\_casm çalıştırılabilir dosyası aşağıda verildiği gibi çalıştırılmalıdır.

```
$ cd ~/Desktop/casm  
$ make  
$ ./combine_casm
```

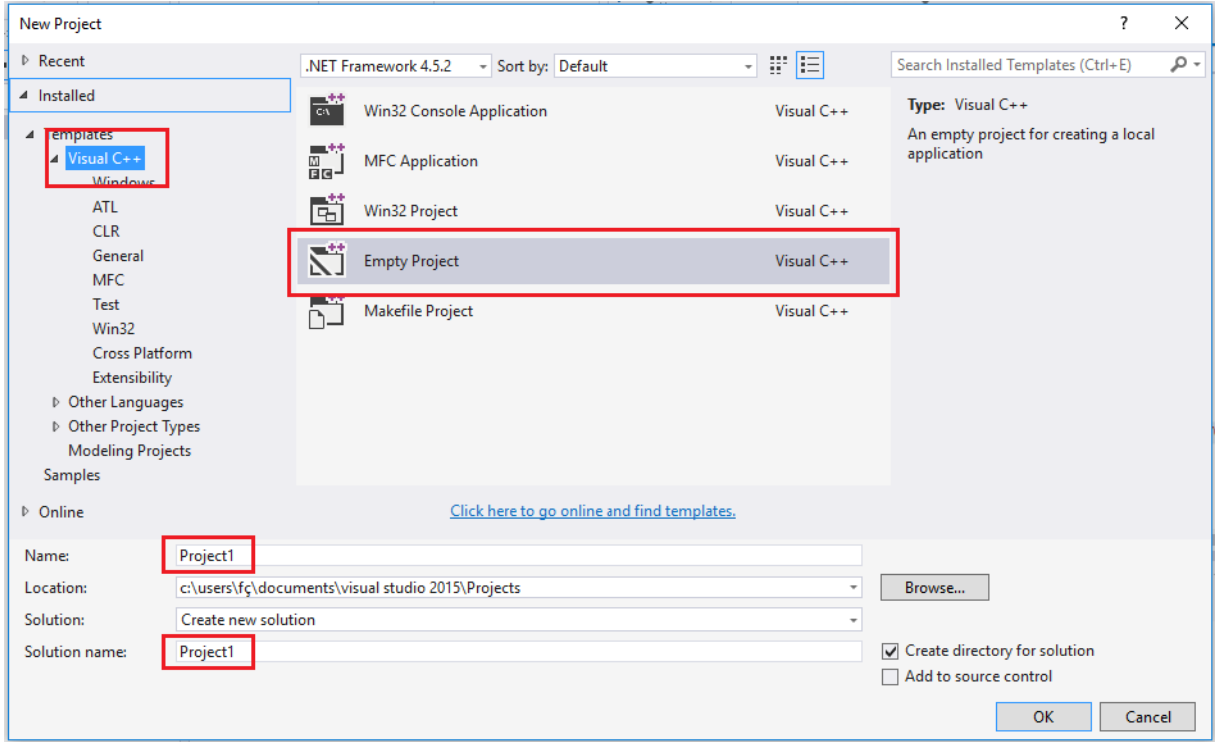
Bu işlem gerçekleştirildiğinde program terminal ekranında çalıştırılacak ve sizden istediği “n” değeri ile işlemi gerçekleştirecektir.

## Windows Ortamında Visual Studio 2015 Kullanılarak C Programı İçerisinden Assembly Yordamı Çağırma

Bu başlıkta, bir önceki başlıkta yapılan işlemler, Windows ortamında kullanıcı dostu bir IDE üzerinde gerçekleştirilecektir.

Örnek olarak Worst-Case durumunda Selection Sort yapan bir C ve bir Assembly fonksiyonu yazılacaktır. C kodu içerisinden her ikisi de çağırılacak ve çalışmaları debug kullanılarak incelenecektir.

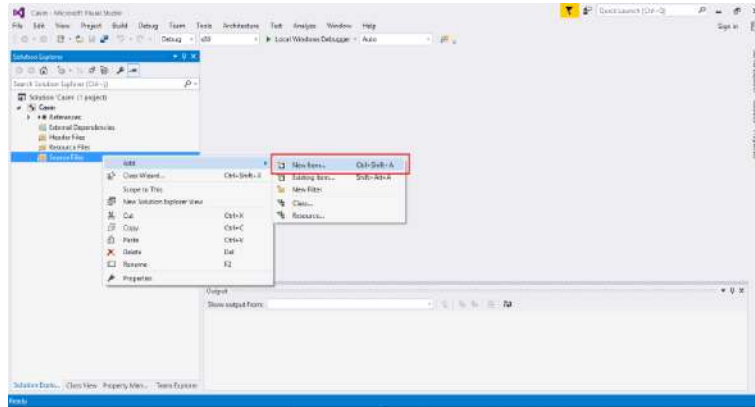
Öncelikle Visual Studio 2015 ortamı çalıştırılır. File -> New -> Project modülü tıklanır ve yeni proje oluşturmak için gerekli ekran ile karşılaşılır (Şekil 3).



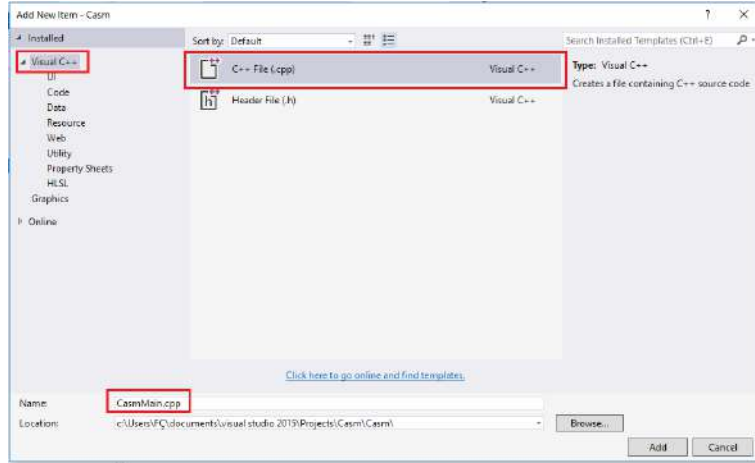
Şekil 3: Yeni Proje Oluşturma Ekranı

Şekil 3'te görüldüğü gibi açılan segmede "Visual C++", ardından "Empty Project" seçildikten sonra proje ve çözüm ismi girilerek boş bir C++ projesi oluşturulur. Unutulmamalıdır ki C++ derleyicileri aynı zamanda C kodlarını da derlemektedir. Bu bakımdan kodunuz C++ olarak kaydedilmesinde ve C++ derleyicisi tarafından derlenmesinde bir sıkıntı olmayacaktır.

Oluşturulan boş projenin "Source Files" kısmı sağ tıklanır ve Şekil 4'te görülebileceği gibi bir C++ kaynak dosyası oluşturulur.



(a)



(b)

Şekil 4: (a) Yeni Madde Seçimi İşlemi, (b) C++ File (.cpp) Dosyası Adıyla Birlikte Oluşturma İşlemi

Oluşturulan ve main fonksiyonu ihtiva eden C programı aşağıdaki gibidir.

#### CasmMain.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

extern "C" void SelSortASM(int *, int);
void SelSortC(int *, int);
const int n = 100000;

int main(){
    int dizi[n];
    int i;
    double sure;
    clock_t bas, bit;
    for (i = 0; i < n; i++)
        dizi[i] = i + 1;
    bas = clock();
    SelSortC(dizi, n);
    //SelSortASM(dizi, n);
    bit = clock();
    sure = (double)(bit - bas) / CLOCKS_PER_SEC;
    printf("Toplam sure: %.2f sn\n", sure);
    printf("Dizinin ilk elemani: %d, son elemani: %d\n", dizi[0], dizi[n - 1]);
    system("PAUSE");
    return 0;
}
```

Kod içerisinde 1’den “n” değerine kadar ardışık elemanları olan bir dizi tanımlanmıştır. Küçükten büyüğe sıralı olan bu dizinin hem “SelSortC()” hem de “SelSortASM()” fonksiyonları ile sıralanması sağlanmıştır. Yazılan fonksiyonların süre olarak karşılaştırılabilmesi için “time.h” kütüphanesinin “clock\_t” veri tipi ve “clock()” fonksiyonları kullanılmıştır.

Öncelikle aşağıda verilen C fonksiyonu ile Selection Sort işleminin gerçekleştirilmesi sağlanmıştır.

```
void SelSortC(int *sdizi, int n) {
    int i, j, tmp, max;
    for (i = 0; i < n - 1; i++) {
        max = i;
        for (j = i + 1; j < n; j++)
            if (sdizi[j] > sdizi[max])
                max = j;
        tmp = sdizi[i];
        sdizi[i] = sdizi[max];
        sdizi[max] = tmp;
    }
}
```

```

    tmp = sdizi[max];
    sdizi[max] = sdizi[i];
    sdizi[i] = tmp;
}
}

```

Prototipten de anlaşılabileceği üzere fonksiyon, sıralanmak istenilen dizinin adresini ve boyutunu alarak, ek bir dizi kullanmadan sıralama sonucunu aynı bellek alanında üretmek üzere yazılmıştır. Bu fonksiyon yukarıda oluşturulmuş “CasmMain.cpp” dosyasının “main()” fonksiyondan sonraki kısmına eklenmiştir.

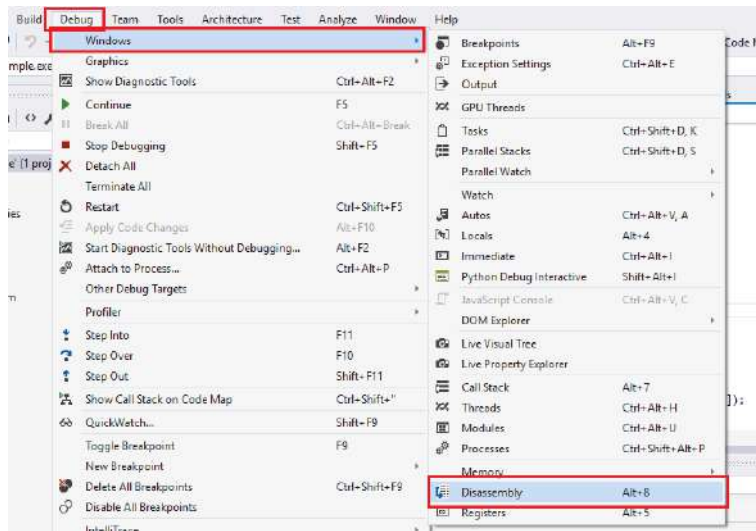
C kodunun Disassembly edilebilmesi için, “main()” fonksiyon içerisinde sıralama fonksiyonunun çağırıldığı satıra bir “break point” konulmuş ve “F5” tuşuna basılarak debug işlemi başlatılmıştır. Debug işlemi “break point” konulan noktada durduktan sonra Debug -> Windows -> Disassembly yolu izlenerek Şekil 5’te gösterildiği gibi C fonksiyonuna karşılık gelen Assembly kodu gözlemlenebilmiştir.

Debug fonksiyonları (F11: Adım adım çalıştır (trace), F10: fonksiyon ve döngüleri atlayarak çalıştır (proceed), F5: Tamamını çalıştır) kullanılarak kod adım adım çalıştırılıp C kodunun derleyici tarafından ne şekilde assembly’e dönüştürüldüğü görülebilir.

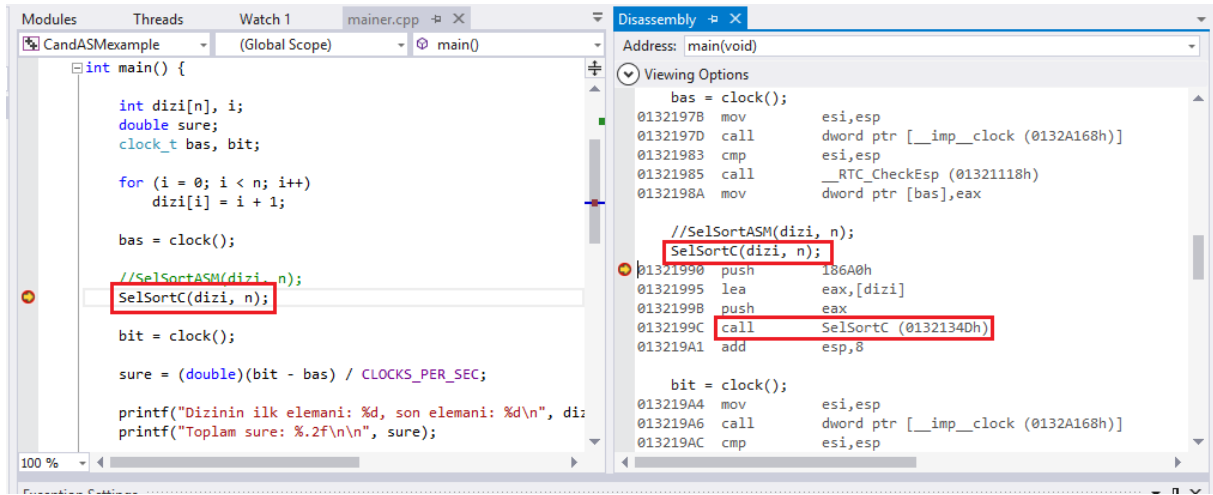
Ancak bu işlemlerin “Debug” modda yapıldığını hatırlatmak ta ve “Release” modda yapıldığında disassembly aşamasında daha az assembly kodu ile karşılaşacağını belirtmekte fayda var.

Şekil 5-b’ye bakıldığında öncelikle kullanılacak parametreler yığına atılmakta ve “call” komutu ile ilgili yordam çağırılmaktadır. Ancak bir diğer dikkat edilmesi gereken husus da fonksiyon çağırımı tamamlandığında IP yazmacının göstereceği koddur ki bu örneğimizde “add esp,8” şeklindedir. Daha önceden de belirtildiği gibi, C de fonksiyon kullanımında, gönderilen parametrelerin, gönderen tarafından kaldırılması gerekmektedir. Burada da SP yazmaç değeri 8 artırılarak bu yapılmaya çalışılmıştır. Burada kesinlikle programcının ekstra bir müdahalesi yoktur, olmamalıdır. C derleyicileri bu işlemleri otomatik olarak gerçekleştirecek şekilde tasarlanmışlardır.

İstenildiği takdirde debug fonksiyonları kullanılarak kod adım adım incelenebilir. Ancak bu işlem uzun süreceği için burada gerçekleştirilmeyecek, yazılan Assembly yordamının incelenmesine geçilecektir.



(a)



(b)

Şekil 5: (a) Disassembly Görüntüleme Başlatma Butonu, (b) C Kodunun Derleyici Tarafından Üretilen Assembly Karşılığı

Assembly yordamının yazılacağı dosyayı oluşturmak için yapılması gerekenler biraz daha farklıdır. Şöyle ki, Visual Studio ortamında projeler build edildiğinde her bir dosya tipinin ne şekilde derleneceği bilinmektedir ve kullanıcının ayrıca bunu bildirmesine gerek yoktur. Ancak Assembly dosyalarının ne şekilde derleneceğini proje ayarlarında bildirmemiz gerekmektedir.

Yine “Source Files” üzerine sağ tıklayarak ismi “casm.asm” olacak şekilde boş bir dosya oluşturulması sağlanmalıdır. Bu dosyanın içerisine “SelSortASM()” yordamını barındıracak aşağıda verilen Assembly kodu yerleştirilecektir.

#### casm.asm

```
.586
.model flat, c
.stack 100h
.data

.code

SelSortASM    PROC NEAR

                PUSH EBP
                PUSH ECX
                PUSH EDI
                PUSH ESI
                PUSH EAX
                PUSH EBX
                PUSH EDX

                MOV EBP, ESP
                MOV EDI, [EBP+32]
                MOV ECX, [EBP+36]

                DEC ECX
                MOV EAX, EDI
dis_don:        MOV EBX, ECX
                MOV ESI, EAX
                MOV EDI, EAX
                MOV EDX, [EDI]
ic_don:         ADD ESI, 4
                CMP [ESI], EDX
```

```

        JGE ATLA
        MOV EDI, ESI
        MOV EDX, [EDI]
ATLA:   DEC EBX
        JNZ ic_don
        MOV EDX, [ESI]
        XCHG EDX, [EDI]
        MOV [ESI], EDX
        LOOP dis_don

        POP EDX
        POP EBX
        POP EAX
        POP ESI
        POP EDI
        POP ECX
        POP EBP
        RET
SelSortASM ENDP
        END

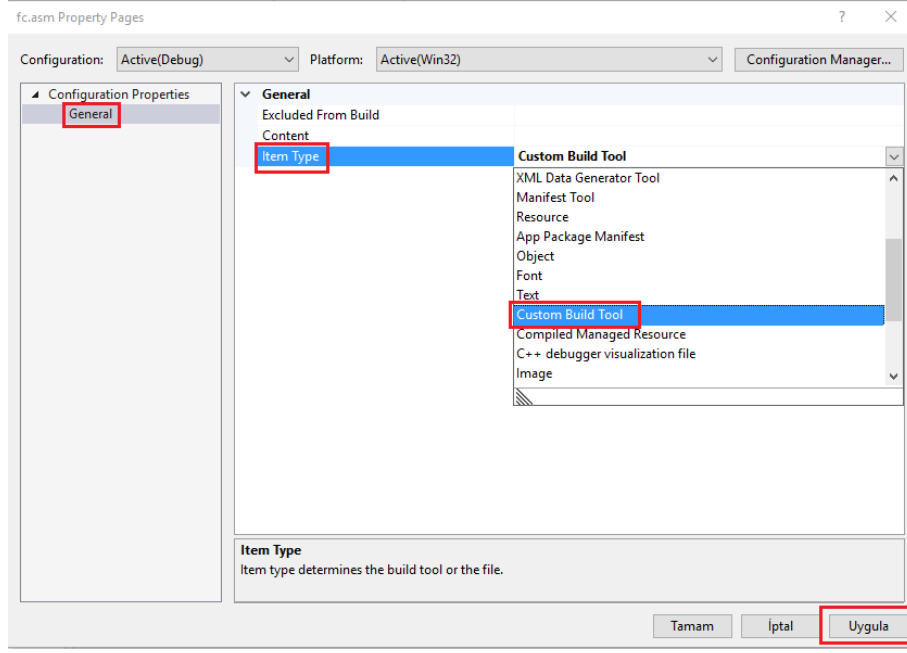
```

Hiç kuşkusuz Selection Sort algoritması farklı iyileştirmeler yapılarak daha optimize bir şekilde çalıştırılabilir. Ancak buradaki amaç varolan bir kodu optimize etmek değil Assembly yordamlarının C programlarından çağırılmasının sağlanmasıdır.

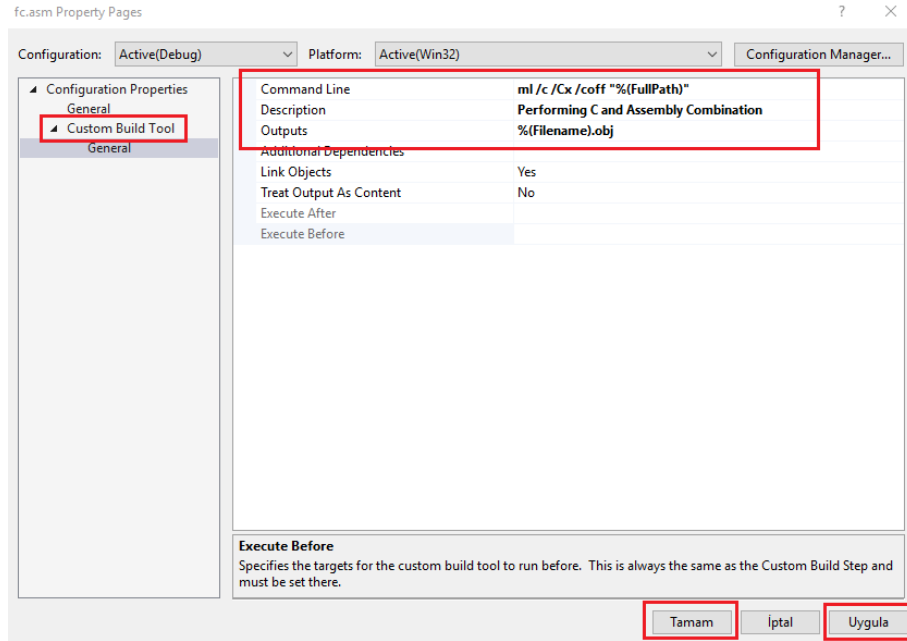
“Main()” fonksiyon içerisinde C’de yazılmış sıralama fonksiyonunu çağırarak satır kapatılıp Assembly’de yazılmış fonksiyon satırı açılmalıdır.

Oluşturulan “casm.asm” dosyasının ne şekilde derleneceğini söylemek için öncelikle proje bölmesindeki dosyaya sağ tıklanır ve “Properties/Özellikler” seçilir. Şekil 6’da gösterildiği gibi açılan ekranda “Item Type” bölümünde “Custom Build Tool” segmesi seçilir ve “Apply/Uygula” butonuna basılır. Bu butona basıldıktan sonra aynı ekranda “Custom Build Tool” segmesi açılacaktır. Bu segmenin altında bulunan “General” segmesine tıklanarak custom build’in ne şekilde gerçekleştirileceğinin söyleneceği ekrana geçilmelidir. Bu ekranın bir görüntüsüne Şekil 7’de yer verilmiştir.

Açılan ekranda “Command Line” kısmına **ml /c /Cx /coff “%(FullPath)”** komutu, “Outputs” kısmına da **%(Filename).obj** ismi yazılmalıdır. Bu komutları yazarak oluşturduğumuz Assembly dosyasının ml (Microsoft Assembler) ile derlenmesini ve sonucunda aynı ismi taşıyan bir obje dosyası oluşturulmasını sağlamış oluyoruz.



Şekil 6: Dosya Özellikleri (Properties) Sayfası



Şekil 7: Custom Build Ayarlarının Yapılacağı Sayfa

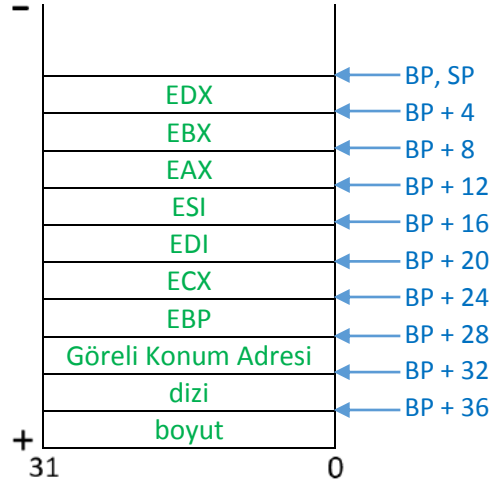
Komut debug edilmeye başlanmadan önce Şekil 8’de verilen yığın organizasyonuna bir göz atılabilir.

Ardından yukarıda anlatıldığı gibi adım adım debug işlemi gerçekleştirilerek yazılan Assembly kodun ne şekilde çalıştığı, disassembly edilmiş C koduyla ne farklılıklar gösterdiği incelenebilir.

Farklı örnek problemler hem C fonksiyonu hem de Assembly yordamı olarak yazılarak hız ve performans karşılaştırması bu şekilde gerçekleştirilebilir.

Bu anlatılanlar öğrenildikten sonra yazılacak aynı işi yapan Assembly yordamlarının en az C fonksiyonları kadar hızlı çalışması beklenmektedir. Çünkü yazılan C fonksiyonları disassembly edildiğinde zaten assembly karşılıkları görülebilecektir.





Şekil 8: Yığın Organizasyonu

## Visual Studio 2015 Ortamında C Dosyası İçerisinde Inline Assembly Komutu Yazma Örneği

Bu başlık altında belki en sık kullanılan C/Assembly kod birleştirme yöntemi olan inline assembly kodu yazma incelenmiştir. Yani aynı dosya içerisinde bir C fonksiyonunda Assembly kodu yazarak işlemlere devam edilmeyi sağlayan yapı incelenmiştir.

Bunun için bir üst başlıkta anlatıldığı şekilde yeni bir proje ve bu proje içerisinde bir tane C++ kaynak kod dosyası oluşturulmuştur.

Visual Studio ortamında c kodunun içerisinde;

```
__asm {
    ;Assembly kodları
}
```

komutu kullanılarak Assembly kodu yazılması sağlanmaktadır. Ancak farklı tür IDE ve platformlarda farklı notasyonların (Intel ve AT&T olarak yaygın kullanılan 2 notasyon avrdır. Ders kapsamında Intel notasyonu kullanılmıştır.) kullanılabileceğini unutmamak gerekir.

C’de bir dizide bulunan elemanları dizinin ortasına göre tersine çevirmek kolay bir işlem sayılır. Ancak integer (32-bit) tanımlı bir dizide, dizinin her bir elemanının ikili (binary) gösterimindeki digit’lerin tersine çevrilmesi o kadar basit bir problem değildir. Bu tarz Assembly dilinde yapılması C diline göre daha kolay olan işlemlerin inline Assembly kodu olarak yazılması hiç kuşkusuz işlevişi oldukça kolaylaştıracaktır. Bu başlık altında yukarıda anlatılan örnek gerçekleştirilmiştir. Sorunun daha iyi anlaşılabilmesi için ikili digit’ler üzerinde tersine çevirme (reverse) işleminin nasıl olduğu aşağıda örnek ile verilmiştir.

**Örnek:**  $11010111100001 \xrightarrow{\text{reverse}} 10000111101011$

Oluşturulan proje içerisinde “inline.cpp” olarak isimlendirilen C++ kaynak dosyası içerisine aşağıda verilen kod yazılmıştır.

Kod içerisinde dizinin elemanlarının bitleri tersine çevrilmeden önce hexadecimal olarak yazdırılması gerçekleşmiş, ardından assembly kodu yazılmış ve elemanlar tekrar ekrana yazdırılmıştır. Böylece yazılan Assembly kodunun doğru çalışıp çalışmadığı ekran çıktılarından anlaşılabilecektir.

Görülebileceği gibi C değişkenleri, Assembly içerisinde kullanılmıştır. Uygulama öncesi temel bilgiler bölümünde de bahsedildiği gibi değişkenler bellekte adres gösteren birimlerdir. Derleyici kodu derlerken onun için bütün değişkenler adreslerden ibarettir. Böyle olunca C değişkenlerinin Assembly içerisinde kullanımı sorun teşkil etmeyecektir.

#### inline.cpp

```
#include <stdio.h>
#include <stdlib.h>
const int n = 10;

int main() {
    int i, dizi[n];
    for (i = 0; i < n; i++) {
        dizi[i] = i + 1;
        printf("%8x, ", dizi[i]);
    }
    printf("\n\n");

    __asm {
        MOV ECX, n
        XOR ESI, ESI

    L2:    MOV EAX, dizi[ESI]
           PUSH ECX
           MOV ECX, 32

    L1:    SHR EAX, 1
           RCL EBX, 1
           LOOP L1
           POP ECX

           MOV dizi[ESI], EBX
           ADD ESI, 4
           LOOP L2
    }

    for (i = 0; i < n; i++)
        printf("%8x, ", dizi[i]);
    system("PAUSE");
    return 0;
}
```

## Neticeler

- C programları içerisinde hıza ihtiyaç duyulduğu zaman Assembly dili rahat bir şekilde kullanılabilir.
- Assembly kodlarının yordamlar olarak hazırlanması ve C dosyalarından fonksiyon çağırır gibi çağırılması her ne kadar modüler programlama açısından elverişli ve tercih sebebi olsa da bazı durumlarda inline assembly kodu yazmanın da sağladığı avantajlar göz ardı edilemez.
- Günümüzde büyük veri analizlerinin gerçekleştirilmesi için günlerce çalışma süresi olan kodlar çalıştırılmaktadır. Bazı durumlarda %10 bile performans artışının çok faydalı olabileceği göz önünde bulundurulmalı ve Assembly kartını masaya koymak için her daim güncel ve hazır olunması iyi olacaktır.
- Assembly dilinin de yüksek seviyeli dillerde olduğu gibi bir çok kütüphanesi olduğu ve bu kütüphanelerde bir çok ihtiyaç duyulan program, yordam seviyesinde kullanıcıya sunulduğu unutulmamalıdır. C içerisinde kullanılan printf ve scanf gibi fonksiyonların da bir kütüphane tarafından sağlandığı, bu kütüphaneyi kullanmadan ekrana bir yazı yazmanın veya okumanın

ne kadar zor olacağı düşünöldüğünde Assembly için hazırlanmış çeşitli kütüphanelerin ihtiyaç duyulduğunda araştırılarak kullanılması belki hayat kurtarabilir ☺