

FURKAN EKİCİ

2017555017

MICROPROCESSORS

LAB.

MOV Komutu :

MOV operand 1, operand 2

→ Görevi : Operand2 değerini operand1'e aktarır.

→ Algoritması : operand1 = operand2

operand 1 { operand 2

Register	{	Memory
Memory		Register
Register		Register
Memory		Immediate → (Verdiğiniz sayılar 3h, 8, DL E h gibi)
Register		Immediate

* Memory'den memory'ye aktarım yapamayız.

→ Etkilediği flagler: C Z S O P A
 Hibritisi

ADD Komutu :

ADD operand 1, operand 2

→ Görevi : operand1 ve operand2'yi topla ve toplamı operand1'e aktar.

→ Algoritması : operand1 = operand1 + operand2

operand 1 { operand 2

Register	{	Memory
Memory		Register
Register		Register
Memory		Immediate
Register		Immediate

* Etkilediği flagler: C Z S O P A

Hepsini etkileyebilir

SUB Komutu :

SUB operand 1, operand 2

→ Görevi : operand1'den operand2'yi çıkar ve değeri operand1'e yaz.

→ Algoritması : operand1 = operand1 - operand2

* Diğer kısımları ADD ile aynıdır.

! Emülatörde görülen her sayı (aksi belirtilmemişti) saatte hexadecimal formattadır. (Execute butonuna bastıktan sonra ekran da)

Registers :

	H → High	L → Low
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

→ Örneğin; AH AL olabilir. Böyle bir durumda Ax registeri 16 bitlik veri tutar.
→ Aslında bunlar işlemcinin kendi değişkenleridir.

* AX, BX, CX, DX değişkenlerinin yanı sıra AH gibi AX'in H kısmını da değişken olarak kullanabiliriz.

AH → 8 bit (byte)

AX → 16 bit (word)

AL → 8 bit (byte)

* ; (noktalı virgül) sıfır satırı olarak kullanılır.

Ex:

sonuna h getince hexadecimal (16'lık taban) oluyor

MOV AL, 5h ; AL'ye 5h değerini ata

SUB AL, 1h ; AL'den 1h çıkart ve AL'ye yaz.

Ex:

Register memory

SUB AL, [0100h] ; 100h adresindeki değeri al AL'den çıkart
; Sonucu AL'ye yaz.

! SUB 5, 3 → Hatalı kullanım (immediate to immediate yok)

! ADD [0100h], [0050h] → Hatalı kullanım (memory to memory yok)

↳ Açıklamadaki operand 1 ve operand 2

* Emülatör bitsin diye RET (return) komutu kullanılır.

Ex:

	H	L
1 MOV AH, 5	05	00
2 MOV AL, 1	05	01
3 ADD AX, 1	05	02
4 RET		

→ 3. satır AX değişkeninin tamamına etki eder. Eğer 3. satırda AX yerine AH yazılısaydı son durum ^H 06 ^L 01 olurdu.

Günümüzde AH ve AL ayrı registerlardır. İkisi birleşip AX'i oluşturur.

* Hexadecimal (16'lık) sayı sisteminde harfler de kullanılır.

(Örneğin; 10 yerine A, 13 yerine D gibi). Immediate dan kisimlara böyle bir sayı verileceği zaman basına sıfır(0) konulmalıdır.

Ex:

MOV AX, OFFh ; FFh'in basına 0 konulur.

Ex:

	H	L
MOV AH, 5	05	00
MOV AL, OFFh	05	FF
ADD AX, 1	06	00

Ex:

	H	L
MOV AH, 5	05	00
MOV AL, OFFh	05	FF
ADD AL, 1	05	00

* AL ve AH ayrı registerlerdir.

Ex:

MOV [0200h], 73 h ; 200h adresine 73 h değerini aktar.

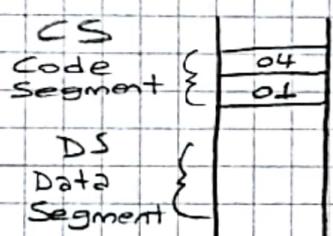
→ Bu kod aslında şu halde dir:

MOV 0100 : [0200h] , 0073 h
(Data) (Adres) (değer)
(Segment)

* 0400 : 0200 adresi emülatörde aratıldığında adresi bize,

01200 diye görünür. Bunun formülü:

Data segmentin sonuna "0" + adres.



* Code Segment, bellekte sıralı bir şekilde bulunduğu komutların sıralı bir şekilde bulunduğu bir blok.

* istenilen verinin depolandığı bellek alanının adresini gösterir.

* Data Segment (DS) değiştirilebilir:

MOV AX, 500h

MOV DS, AX

→ DS, 0100'dü şimdi D500 oldu.

! DS'ye direkt erişemeyiz. Araya AX, BX gibi değişken sokup öyle değiştirebiliriz.

INC komutu:

INC operand

→ Görevi: operandin değerini bir artırır.

→ Algoritması: $\text{operand} = \text{operand} + 1$

Operand

Register

Memory

* Etkilediği flagler: C Z S O P A
değişimez degistirilebilir

DEC komutu:

DEC operand

→ Görevi: Operandın değerini bir azaltır.

→ Algoritması: $\text{operand} = \text{operand} - 1$

* Diğer kısımları INC komutu ile aynı.

LOOP komutu:

LOOP etiket

→ Görevi: CX register'i "0" değilse etikete gider.

→ Algoritması: $CX = CX - 1$ büyük veya küçükse (0'a eşit değilse)

if $CX < > 0$ then

• Jump

else

• no jump, continue.

Ex:

MOV CX, 5h ; CX'ye 5 değerini atı (döngü 5 kere dönecek)

etiket: ; Label (etiket) belirle

INC AX ; Her döngüde AX'yi 1 artır.

LOOP etiket: ; CX, 0 mi kontrol et. Değilse, CX'yi 1 azalt ve etiketeden
; CX=0 ise alt satırda devam et.

* CX loop sonunda kendisi azalıyor biz 221 miyoruz!

! Loop u baza etiketi sona yazma! (yerlerini karıştırma)

★ SI (source index), DI (destination index), BP (base pointer), SP (stack pointer) değerleri de bir registerdir. Bunlara doğrudan değer aktarımı yapabiliyoruz (ES, DS, CS, SS 'den farklı olarak). Bunları adres gibi kullanabiliyoruz. (AX, BX, CX, DX'den farklı olarak). Örneğin;

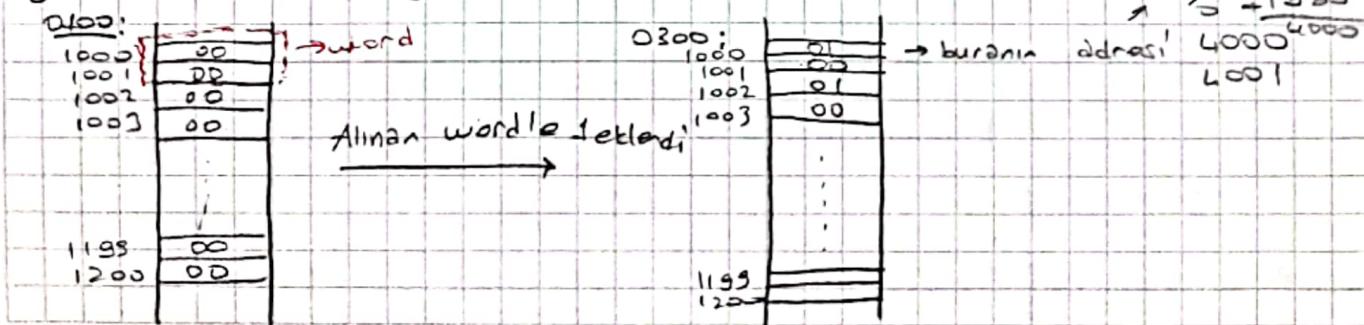
MOV SI, 173h }
 MOV [SI], 5h } Sunulmuş aynıdır: MOV [0173h], 5h.

★ Basında belirtmediğimiz sürece DS ile kullanıyor gibi kabul eder. (Aslında DS:[SI], 5h yazıyor)

Ex: Write a program that reads value from [0100:1000h]-[0100:1200h] memory range. Add 1 to every word before writing back to memory. You have to use DS:SI addressing for reading and ES:DI addressing for writing.

(Belirtilen adreslerdeki değerleri DS:SI kullanarak 1, 1 ekle ve belirtilen adres'e ES:DI kullanarak yaz.)

! Burada dikkat edilmesi gereken şey "word" kelimesidir. 16 bit demektir. Emülatörün adres kısmında her satır 8 bit tır. Yani burada yapılması gereken düşük öncelikli byte'ı 1 artırmak, yüksek öncelikli byte'ı dokunmamak.



MOV AX, 100h ; AX'ın 100n atı
MOV DS, AX ; AX değerini DS'ye atı
MOV SI, 1000h ; SI'yi 1000n yap

MOV BX, 300h ; BX'le Book adı
MOV ES, BX ; BX değerini ES'ye atı
MOV DI, 1000h ; DI'yi 1000n yap

MOV CX, 100h ; Döngü 100 defa calısır. Cunki 2'li gidiyoruz.
; (atlayarak)

etiket: ; etiket (label) döngünün geleceği yer.

MOV DX, DS:[SI] ; DS:SI'deki değer DX'e aktarıldı
INC DX ; DX 1 artırıldı
MOV ES:[DI], DX ; DX değeri ES:DI'ya aktarıldı
ADD SI, 2 ; Bir önceki sebepten dolayı
ADD DI, 2 ; bunlar 2 arttı (word)

LOOP etiket ; CX=0 mı? değilse CX=cx-1 yap etikete git
RET ; Programı sonlandır.

* DS ve ES ayırmayı sorabilin hoca!

! 0100:1000 adresine 1-100 arası sayıların toplamı yazarsak:

02000 : BA	ve o adresle gidersek karşımıza
02001 : 13	
02002 : 00	bu çıkar. Burada 2001inci likli
:	byte'tır ve bu sayı 13BA.

yani SOSO diye okunur.

CMP Komutu:

CMP operand1, operand2

→ Görevi: operand1 ve operand2'yi karşılaştırır.

→ Algoritması: operand1 - operand2

operand1	{	operand2
Register		Memory
Memory		Register
Register		Register
Memory		Immediate
Register		Immediate

* Etkilediği flagler: C Z S O P A

r r r r r r

* Görüldüğü gibi Memory to Memory yok!

* operand1 - operand2 yapar. Sonuç "0" ise $ZF=1$ olur (esit demek)
Sonuç "0" değilse $ZF=0$ olur. (esit değil)

JMP Komutu:

JMP label (or 4bytes address.)

→ Görevi: Bu komut okunduğunda belirtilen etikete kosulsuz sıçrama yapar.

Jcc Komutları:

* Burada koşullu sıçramadan bahsedilir. Örneğin;

JZ label , şeklindeki kullanım eğer ZF (zero flag) 1 ise
label'e sıçra , aksi takdirde devam et demektir.

* Bazi jmp lar signed, bazi lar ise unsigned olarak gelir.

Bunlara ; emu8086 → help → Tutorials → Chapter 7 'den
erisilebilir.

* Signed olanlardan birkaç tane , unsigned olanlardan birkaç
tane ezberle!

* Eğer MOV ile atama işlemi yaparken byte kadar veri aktarıyorsak;

MOV byte ptr [1000h], 34h → 1 veya 2 haneli.

word kadar veri aktarıyorsak;

MOV word ptr [1001h], 1234h → 3 veya 4 haneli.

→ Biz yazmadığımızda bunlar otomatik konur.

Ex: 0100:1500h ve 0100:15FFh adresi arasındaki değerleri kontrol et. Eğer adresdeki değer "0" ise "0" dan başlayıp "FF" e kadar giden değerini o adresse yaz. (örnegin; 0100:1543h adresindeki değer "0" ise bu adresse 43h yaz "0" değilse değerini aynı kalsın).

MOV AX, 100h ; AX'e 100h ata

MOV DS, AX ; AX değerini DS ye aktar.

MOV SI, 1500h ; SI ya 1500h ata

MOV CX, OFFh ; Döngü FFh kadar denecek belli.

ETIKET:

; LOOP dönüs etiketi

CMP DS:[SI], 00h ; Adresdeki değer kontrol et.

JNZ ATLA ; Esit değilse etikete atla.

MOV DS:[SI], BL ; BL (byte) değerini adresse aktar.

ATLA: ; Esit değilse geleceği etiket.

INC BL ; BL'yi 1 artır.

INC SI ; SI'yi 1 artır. (Diger adresler için)

LOOP ETIKET ; CX sifir değilse etikete atla.

RET ; Programı bitir.

* Döngünün kaç kere deneceği belli olmasaydı, bu döngüyü loop yerine JMP ile kurardık!

ADC Komutu:

ADC operand1, operand2

→ Görevi: operand1, operand2 ve CF'yi toplayıp operand1'e yazar.

→ Algoritması: operand1 = operand1 + operand2 + CF

operand1 | operand2

Register	Memory
Memory	Register
Register	Register
Memory	immediate
Register	immediate

† Etkilediği flagler: Cz SO PA

Hepsi

★ Eldeli toplama yapar. (Full adder)

Ex:

MOV AL, OFFh ; AH:00 AL:FF
MOV BL, 003h ; BL:02

ADD AL, BL ; Bunun sonucunda toplamın sonuc 8 bitten büyük oluyor. (CF=1)
; ve bu sonucun sadece son 8 bitlik kısmı dikkate alınıyor.
; örneğin 10000000 + 00000010 = 00000010

ADC AH, 0 ; Normalde hiçbir şey yapmaz ama CF varsa AH:01 olur
; Tüm işlemlerin sonunda sonuc AH:01 AL:02 => AX=0102

Ex: 2 tane 4 byte'lık işaretsiz tam sayıyı 0.100:1000h ve 0.100:1004h

adreslerinde oku. Bu sayıları topla, toplamı 0.100:1008'e yaz.

(1. sayı → 3.000.000.000 (B200 5E00), 2. sayı → 1.000.000.000 (3B9A CA00))

MOV AX, 0100h

MOV DS, AX

MOV BX, 01000h

; DS ayarlandı

; BX değer adreslerle birleştirilmiş şekilde ayarlandı

MOV [BX], 05E00h

; Üç milyarın son dört hanesi (düşük öncelik)

MOV [BX+2], 0B2D0h

; Üç milyarın ilk dört hanesi (yüksek öncelik)

MOV [BX+4], 0CA00h

; Bir milyonun son dört hanesi (düşük öncelik)

MOV [BX+6], 03B9Ah

; Bir milyonun ilk dört hanesi (yüksek öncelik)

MOV AX, [BX]

; 5E00

ADD AX, [BX+4]

; CA00

MOV [BX+8], AX

; Toplanan değerin düşük öncelikli kısmı yazıldı.

MOV AX, [BX+2]

; B2D0

ADC AX, [BX+6]

; 3B9A

MOV [BX+10], AX

; Toplanan değerin yüksek öncelikli kısmı

RET

; Sonuç => E6FB 2800
; 100A 100B 100C 100D 100E 100F

SBB Komutu:

SBB operand1, operand2

→ Görevi: operand1'den operand2 ve CF'yi çıkarıp CF'ye yazar.

→ Algoritması: operand1 = operand1 - operand2 - CF

operand1	operand2
Register	Memory
Memory	Register
Register	Register
Memory	immediate
Register	immediate

★ Etkilediği flagler: CZSOPA
Hepsi.

★ ADC için yapılan örneğin sahce toplama komutlarını değiştiremeye eldeلى ulasma da yapabiliriz (2 tanesi 32 bitlik sayı arasında) (ADD → SUB, ADC → SBB)

★ MBSB, MBSW, LODSB, LODSW, STOSB, STOSW bunlar string

hafta 4

hafta 5

Komutlarıdır. (isimleri böyledir; yine sayılarla işlem yapılır)

→ Yukarıdaki komutlar birbir bayrağı değiştirmez.

STOSB Komutu:

STOSB

→ Görevi: AL'in içerigini ES:[DI]'ya kopyala ve DI'yi güncelle.

→ Algoritması: • ES:[DI] = AL
if DF=0 then
• DI = DI + 1
else
• DI = DI - 1

★ Etkilediği flagler: Hibritisi

STOSW Komutu:

STOSW

→ Görevi: AX'in içerigini ES:[DI]'ya kopyala ve DI'yi güncelle.

→ Algoritma: • ES:[DI] = AX
if DF=0 then
• DI = DI + 2
else
• DI = DI - 2

★ Etkilediği flagler: Hibritisi

LODSB Komutu:

LODSB

→ Görevi: AL'ye DS:[SI] içeriğini yükler ve SI'yi günceller.

→ Algoritması: • AL = DS : [SI]
if DF=0 then
• SI = SI + 1
else
• SI = SI - 1

* Etkilediği flagler: Hiçbirisi

LODSW Komutu:

LODSW

→ Görevi: AX'e DS:[SI] içeriğini yaz ve SI'yi güncelle.

→ Algoritması: • AX = DS : [SI]
if DF=0 then
• SI = SI + 2
else
• SI = SI - 2

* Etkilediği flagler: Hiçbirisi

MOVSB Komutu:

MOVSB

→ Görevi: ES:[DI]'ya DS:[SI]'yı at ve DI ve SI'yi güncelle.

→ Algoritması: • ES:[DI] = DS:[SI]
if DF=0 then
• SI = SI + 1
• DI = DI + 1
else
• SI = SI - 1
• DI = DI - 1

* Etkilediği flagler: Hiçbirisi

MOVSW Komutu:

MOVSW

→ Görevi: ES:[DI]'ya DS:[SI]'yı at ve DI ve SI'yi güncelle.

→ Algoritması: • ES:[DI] = DS:[SI]

if DF=0
• SI = SI + 2
• DI = DI + 2

* Etkilediği flagler: Hiçbirisi

else
• SI = SI - 2
• DI = DI - 2

CLD Komutu:

CLD

* Etkilediği flagler: DF

→ Görevi: DF'nin içini sıfır yapar.

→ Algoritmisi: DF = 0

Digerleri etkilenmez

STD Komutu:

→ Görevi: DF'nin içini set eder (1 yapar)

→ Algoritmisi: DF = 1 * Etkiledigi flagler: DF

REP Komutu:

REP string instruction. (MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW)

→ Görevi: CX ≠ 0 ise string instruction'ı yapar.

→ Algoritmisi: CX = CX - 1 * Etkiledigi flagler: ZF

if CX < 0 :

do instruction

else

continue.

Digerleri etkilenmez.

Örnek: 0100:0300 - 0100:03FF adresine kadar olan kısma ABCDEh yaz.

Daha sonra 0100:0300 - 0100:03FF adresindeki verileri 0100:3500 - 0100:35FF

adreslerine yaz.

MOV AX, 0100h	; AX'e 0100h aktar.
MOV ES, AX	; ES'ye 0100h aktar. (Depolama ES:SI'ye olur)
MOV DI, 0300h	; DI'ya 0300h ata.
MOV CX, 128	; Döngü 256(100h) kadar değil 128 dönecek
MOV AX, ABCDEh	; AX'E belirtilen değeri aktar.
CLD	; DF=0
REP STOSW	; Verileri ES:DI'ya depola (0100:0300 - 0100:03FF)

MOV AX, 0100h	; AX'e 0100h ata.
MOV DS, 0100h	; DS'ye 0100h ata (Veriler buradan (DS:SI) okunur)
MOV SI, 0300h	; SI'ya 0300h ata.
MOV ES, AX	; ES'ye 0100h ata (Veriler buraya (ES:DI) yazılır)
MOV DI, 3500h	; DI'ya 3500h ata
MOV CX, 128	; Döngü 128 kez döner
REP MOVSW	; DS:SI'deki verileri ES:DI'ya aktar.

MUL Komutu:

MUL operand

→ Görevi: işaretsiz sayılarında çarpma işlemi.

→ Algoritması: operand byte ise:

$$AX = AL * \text{operand}$$

operand word ise:

$$(DX; AX) = AX * \text{operand}.$$

Operand

Register
Memory

* Etkilediği flagler: CF, OF

Digerleri degisebilir ama
dikkate alınmaz (undefined)

IMUL Komutu:

IMUL operand

→ Görevi: işaretli sayılarında çarpma işlemi.

→ Algoritması: MUL komutu ile aynı.

Operand

Register
Memory

* Etkilediği flagler: CF, OF

Digerleri degisebilir ama
dikkate alınmaz (undefined)

Örnek:

MOV AL, 20 ;

MOV BL, 4 ;

MUL BL ; $AX = AL * BL$

$$\begin{array}{l} \text{dec/d16} \\ 20d \times 4d = 80d = 50h \Rightarrow AX' 00\ 50 \end{array}$$

Örnek:

MOV AL, -2

MOV BL, -4

MUL BL ; $AX = AL * BL \rightarrow (-2) \times (-4) = 8 \quad AX' = 00\ 08$

$$\begin{array}{l} \text{(8bit)} \rightarrow AL \\ \text{(-2)} = FE \\ \text{(-4)} = FC \rightarrow BL \quad FE = 254 \\ \quad \quad \quad FC = 252 \end{array}$$

254 ve 252 çarpılarak ve hex'e çevrilip AX'e yazılıdı.
Üçüncü unsigned olarak işlem görecek. Ama burada
bizim yapmak istediğimiz signed sayıları çarpmak.

Örnek:

MOV AX, OFFFBh ; -5d
 MOV BX, 00001h ; 1d
 MUL BX ; DX: FFFF AX: FFFB

carpma sonuc negatif.

Örnek:

MOV AX, 0FFFFh
 MOV BX, 0FFFFh
 MUL BX

$$; (DX:AX) = AX * BX \rightarrow DX:FFFF AX:0001$$

yörekti.
örekti.

düzelte
önceki.

* IMUL olsaydı FFFF \rightarrow -1 $(-1) \times (-1) = 1$ olurdu

Sonuç olarak DX:0000 AX:0001 olurdu.

Örnek: işaretsi 2 FEh ve 10h sayılarını çarp sonuc 0100:0400'e yaz

MOV AL, 0FEh ; 254d
 MOV BL, 010h ; 16d
 MUL BL ; AX = AL * BL $(254) \times (16)$

MOV DX, 0100h ;
 MOV DS, DX ; DS'yi 0100 yap.

MOV [6400h], AX; Verilen adrese AX'i yaz

* IMUL yazsaydık $(-2) \times (+16)$ yapacaktı FE = -2 (signed)
 $= 254$ (unsigned)

* Register ile de çarpabileceğimizi söyledik.

MUL byte ptr [1000h] \rightarrow AX = AL * [1000h] (byte)

MUL word ptr [1000h] \rightarrow (DX:AX) = AX * [1000h] [1001h] (word).

→ Bunları yazmazsağ byte ptr olarak alın.

DIV Komutu:

DIV operand

→ Görevi: Tamsayı, işaretsi bölmeye yapar.

→ Algoritmaları: operand byte ise:

$$AL = AX / \text{operand}$$

AH = remainder (kalan)

operand word ise:

$$AX = (DX \ AX) / \text{operand}$$

DX = remainder (kalan)

operand

Register
Memory

* Etkilediği flagler: Hepsi undefined

IDIV Komutu

IDIV operand

operand
Memory
Register.

→ Görevi: Tamsayı, işaretli bölmeye yapar.

→ Algoritması: DIV komutu ile aynı. ~~Etkilediği flagler~~: Hepsinin undefined

Örnek:

MOV AX, 11d
 MOV DL, 2
 DIV DL ; AL = AX / opo
 AL = 11 / 2 = 05 AH: kalan } $\overbrace{\overbrace{01}^{01} \overbrace{05}^{05}}$
 { AX: 01 05

Örnek: $200.0000_d = \underline{\underline{3}} \underline{\underline{0}} \underline{\underline{D}} \underline{\underline{4}} \underline{\underline{0}} h$

MOV DX, 00003h → yüksek öncelik (DX AX)
 MOV AX, 00D40h → düşük öncelik, $\rightarrow 50.000$
 MOV [1000h], 004h
 DIV word ptr [1000h] ; AX = C350 DX = 0000 (kalan yok)

| Buraya byte yazıldı byte bâlmesi yapardı. (E bit)

Örnek:

MOV AX, -203 ; AX=0FF25h
 MOV BL, 4
 IDIV ; AL = -50 (0CEh), AH = -3 (0FDh)

Örnek: $-85(0ABh) / 10(0Ah)$ yap. bâlmi^t 0100.0500'e, kalan 0100.0502.
 (signed)

MOV AX, 0FFFAB ; Sayı negatif ve 16 bitlik soruda sign bit var.
 MOV BX, 00Ah ; Onur için FFBAB yazdırık. DA3h yazsaydılk
 IDIV BL ; AX:00AB yarı pozitif olurdu
 (-111)

MOV DX, 0100h
 MOD DS, DX ; DS'ye 0100h atıa.

MOV [0500h], AL ; Bâlme
 MOV [05022h], AH ; + kalan

| Word ile signed bölmeye yapılanı yapıllıken DX'e FFFF overmeyi unutma. Çünkü (DX AX) bir sayı sayılardan bâlmede yani DX'te 0000 olursa 2. Çünkü sayı negatif bu ýerde FFFF olursa 0'da. (F000 sadece AX kullanılıyorsa sayıların)

INT komutu:

INT imm.

→ Görevi: Interrupt oluşturur. Bu interruptların listesi sitale var.

→ Algoritması: Push to stack, flags, CS, IP

IF = 0

imm Transfer control to interrupt procedure.

immediate (0-255)
(8 bit)

* Önceden yazılmış fonksiyonlar gibi disüntebilir.

* Bu ders INT1 10h / AH=0EH - teletype output interruptini göreceğiz. Bu interrupt sayesinde ekrana karakter bastırabiliyoruz. Bu interrupt AL'in içeriğini ekrana bastırmamızı sağlıyor.

Örnek:

MOV AL, 'A' ; Hex 41 dec 65 AL'ye karakter verilebilir.
MOV AH, 0EH ; INT1 10h/AH=0EH kesmesi
INT 10h ; Bir ekran açılır ve 'A' yazdırılır.

* Birkas tane daha INT 10h yazılırsa yazılım INT 10h kadar A yazılır ekranda.

* Direkt karakteri yazmanıza gerek yok. ASCII tablosundaki hex veya dec karşılıkları da yazılabilir. Yukarıda örnek için MOV AL, 41h veya MOV AL, 65d yazabilirisiz.

* C dilinde yazdığımız int a=5; kodunda ki a'yı mp anımsaz. Sadece a'nın RAMdeki yerini bilir ve içinde tutulduğumuzu bilir.

* Assembly'de kendi değişkenlerimiizi oluşturabiliriz.

DB → define byte DW → define word keywordlerini kullanarak.

Örnek:

MOV AX,00100h
MOV CX,01h
INC CX
RET
ekici DB 73h

in memory →

01000 :	138	}
01001 :	00	
01002 :	01	}
01003 :	B9	
01004 :	01	}
01005 :	00	
01006 :	41	}
01007 :	73	

* INC komutu olmasa değişke 01006'da tutulacaktı.

LEA Komutu :

→ Görevi: oluşturduğumuz değişkenin adresini register'e aktarır (16bit)

→ Algoritması: REG = address of memory

LEA reg16, variable.

* Etkilediği flagler: Hiubirisi

* Yazdığımız değişken kodun ulaşamayacağı bir yere yazmazız.
(Örn RET komutunun altına)

Örnek:

MOV AX,00100h
INC CX
LEA BX, ekici
RET
ekici DB 73h

01000 :	B8	}
01001 :	00	
01002 :	01	}
01003 :	41	
01004 :	BB	}
01005 :	08	
01006 :	00	}
01007 :	C3	
01008 :	73	ekici db 73h

↳ ekici değişkenin adresi (01008)

Kod uzayıktan sonra:

BX:00 08 olur.

DS:0 + BX yapılarında ekici değişkenin adresi bulunmuyor.

* ekici DB 'fırkan' da yazabilirmiz veya ekici DB 72h, 14h, 20h ---- gibi değerler de yazabilirmiz.
(array)

||| Eğer yukarıdaki örnekte bunu yazsaydık BX yine 0008 olurdu. Başlangıç adresini gösteriyor. 'fırkan' da da aynıtı olundu. f'nin tutuldığı adresi gösterirdi.

Örnek: isim yazdırma.

MOV AH, 0EH
MOV CX, 6
LEA BX, ekici

; Teletype - ekrana yazmak için AH=0EH olmalı
; ilk 6 karakterin olduğu için
; ekici değişkenin ilk adresini al

DONGU:

MOV AL, [BX]
INT 10h
INC BX
LOOP DONGU
RET

; etiket

; AL'ye BX adresindeki değeri aktar.
; 10h kesmesi yap ekrana karakteri bastır.
; bir sonraki karaktere geç
; etikete dön
; Sonlandır.

ekici DB 'furkan' ; ekici değişkenine 'furkan' stringini写了.

* isim DW 11h, 22h, 33h yazarsak.

0100: 11
01001: 00
01002: 22
01003: 00
01004: 33
01005: 00

} olur. Çünkü word dedik.

Bu soruyu söyle de çözülebilirdi;

MOV AH, 0EH
LEA BX, ekici

DONGU:

MOV AL, [BX]
CMP AL, 0
JE SON
INT 10h
INC BX
LOOP DONGU

BITIS:

MOV AH, 4CH ; INT 21h/4C kesmesi programı sonlandır.
INT 21h
ekici DB 'furkan', 0

* Bu kodda sonsuz döngü yapılmış. 'furkan' den sonra 0 geleceğinden SON etiketine gidierek program sonlandır. Burada ismin kaç karakterli olduğunu CX'e yazmak gerekmek.

* DS ve CS otomatik olarak 0100 tanımladığı için oluşturduğumuz

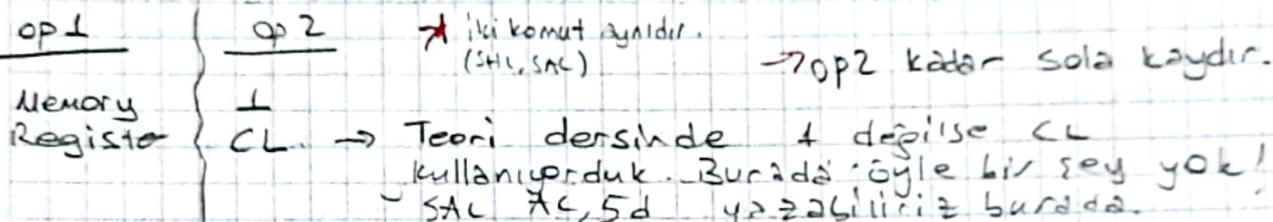
değişkenler kodla karışık yolda bu yüzden RET'in altına yazıydı. Eğer DS'yi değiştirdiğimizde bu sorun olusmaz ve ni deyişkeni istedigimiz yere yazabilirim.

SHL/SAL Komutları

- Görevi: Sola doğru bitleri kaydırır sağda boş kalan yerlere 0 koyar.
- Algoritması:

SHL/SAL, op1, op2

* Etkilediği flagler: CF, OF etkilenir.



Örnek:

MOV AL, 0E0h
 SAL AL, 1
 RET

CF=0 E 0

AL = COh

CF=1 C 0

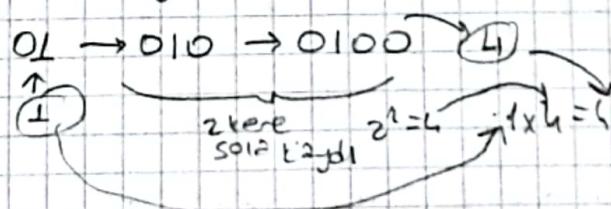
* OF=0 Eğer ilk bit orijinal sign bitinde kalırsa.

* SAL [100bh], 1 böyle yazarsak byte ptr olarak izlenir

↳ SAL byte ptr [1000h], 1 (byte)
 ↳ SAL word ptr [1000h], 1 (word)

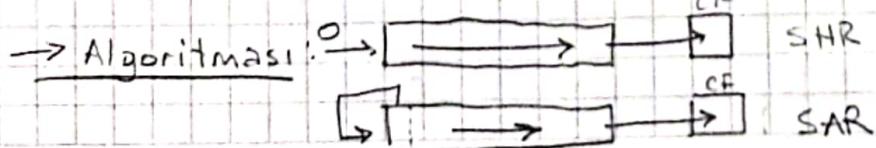
* Sola n kadar kaydirmak sayıyi 2^n ile carpmak demekdir.

Örneğin:



SHR/SAR Komutları

- Görevi: Bitleri sağa doğru kaydırır. Solda boş kalan yere SHR'de 0 konur, SAR'da en soldaki bit tekrar konur.



SHR/SAR op1, op2

op1, op2 yukarıdaki gibi

* Etkilediği flagler: CF, OF

Örnek:

MOV AL, 91h ; 1001 0001 CF=1
 SAR AL, 1 ; 1100 1000 ①
 MOV AL, 91h ; C 8
 SAL AL, 1 ; 0100 1000 ② CF=1
 4 8

* SAR soldaki boş kalan yerlere en yüksek bitini yerlestirirken, SHR 0 kayar.

Örnek: 0100:1000'e 2 yaz bu sayıyi shift islemi ile kullanarak 18 ile çarp sonucu 1002h adresine yaz.

MOV [1000h], 002h ; Doğrudan 18 ile çarpamıyorum $(x16 + x2)$
 MOV BL, [1000h] ; 2 ile çarpıp toplamak için yaptık
 SHL [1000h], 4 ; $x16 \rightarrow 2 \times 16 = 32$
 SHL BL, 1 ; $x2 \rightarrow 2 \times 2 = 4$
 36

MOV AL, [1000h]
 ADD AL, BL ; $x16 + x2 = 36 \rightarrow$ sonuç
 MOV [1002h], AL ; Sonucu yaz.

Örnek: 1000h adresine yazdiğim sayının 12. biti 0 olsun kadar sağa kaydır.

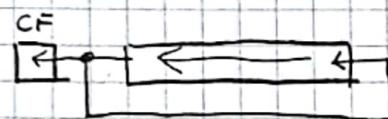
TEST komutu: Bitler arasında and işlemi yapar. 2 operatördür. bunların bitlerini karşılaştırır. Eğer karşıtlı bitlerin herhangi biri 1 olursa ZF=0 olur. (Sonra 0'dan farklı). Eğer 0 bit uymazsa sonuç 0 olacağinden ZF=1 olur.

MOV [1000h], 0FE00h ; 1111 1110 0000 0000
 MOV BX, 0001 0000 0000 0000 ; Test için
 MOV AX, [1000h] ; Veri AX'ye aktarıldı
 DÖNGÜ:
 TEST AX, BX ; Eğer AX'in 12. biti 0 olursa ZF=1
 JZ BITIR ; ZF=1 ise 2+12
 SHR AX, 1 ; Döpsilse bir sağa kaydırır, 12. bitin
 JMP DÖNGÜ ;
 BITIR:
 MOV [1000h], AX ; AX'i tekrar yerine yaz
 RET ; Bitir.

Rotate Komutları :

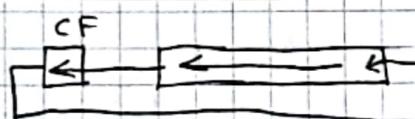
★ Etkilediği flagler: OF, CF

ROL Komutu



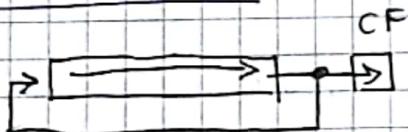
En yüksek değerlikli bit hem CF'ye hem en düşük değerlikli bite kayar.

RCL Komutu :



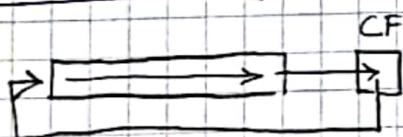
En yüksek değerli bit CF'ye
CF ise en düşük değerli bite kayar.
Tam bir cycle gibi.

RDR Komutu :



En düşük değerli bit hem CF'ye
hemde en yüksek değerli bite kayar.

RCR Komutu :



En düşük değerli bit CF'ye
CF ise en yüksek değerli bite kayar.
Tam bir cycle gibi

★ ROL op1, op2 (Hepsi iin genel)

op1 { op2

Register, Imm.
Memory, CL

Örnek:

MOV AL, 080h ; 1000 0000 CF=0
SAL AL, 1 ; 0000 0001 CF=1

Örnek:

MOV AL, 089h ; 1000 1001
 ROR AL, 1 ; 1100, 0100

8	9
C	4

CF=0
 CF=1

Örnek: BX' registerine yüklenen herhangi bir sayının içinde kah. bitinin 1 olduğunu say DX ile.

MOV BX, 00F0FH ; Sayı 0000 1111 0000 1111 → 8+2=10
 MOV CX, 16 ; Sayı 16 bitlik.

CLC ; CF=0

DONGU: ; Döngü başı

RCL BX, 1 ; 1 kere sola rotate yap son bit CF'ye yoktur.
 JNC DEVAM ; Eğer CF=0 ise devam etmeye git.
 INC DX ; Eğer CF=1 ise DX'yi 1 artır.

DEVAM:

LOOP DONGU ; Döngü başına dön.

RET ; Return

* Burada tüm bitleri tek tek CF içine atarak CF içinde bitleri kontrol ediyoruz.

Örnek: BX'te tutulan sayı negatif olma kadar rotate et.

Kah. rotate yapıldığını CX'te tut.

MOV BX, 00F0FH ; 0000 1111 0000 1111

DONGU: ; döngü başı

INC CX ; CX'yi 1 artır.

ROL BX, 1 ; BX'yi 1 defa sola rotate et.

JNC DONGU ; CF=0 ise döngü başına gel

DEC CX ; CX'yi azalt

ROR BX, 1 ; BX'yi sağa kaydır.

* Burada yine CF kontrolü yapılıyor. Eğer CF'ye 1 gelmişse bir önceki adında en yüksek değerli bit 1'di. Bu adımı bir kez geriye alarak en sol biti 1 olan (negatif) sayıyı elde ederiz.

PUSH Komutu

→ Görevi: Stack'e 16 bitlik veri gönderilir.

→ Algoritması: $SP = SP - 2$
 $SS:[SP]$ (top of the stack) = operand.

PUSH operand

operand

REG16

SREG

Memory

Immediate

* Etkilediği flagler: Hibritisi.

POP Komutu

→ Görevi: Stack'ten veri çıkarılır.

→ Algoritması: $operand = SS:[SP]$ (top of the stack)
 $SP = SP + 2$

POP operand

operand

* Etkilediği flagler: Hibritisi.

REG16

SREG

Memory

* LIFO (last in first out) prensibi ile çalışır stack.

Veriler $SS:[SP]$ 'ye yazılır veya buradan okunur.

$SS:[SP]$ stack'e eklenen son verinin yerini gösterir.

Veri koyduktan sonra SP azalır ve veri çıkardıkça SP artar.

Default olarak $SP : FFFE$ 'dir. Stack'in en yüksek değeri yerini gösterir.

01000

← SP önce burayı gösterir ve eklenerek azaltılarak
10FFF yukarı çıkar.

Örnek:

MOV AX, 01234h
 PUSH AX
 POP DX ; DX = 1234h

34	10FFC ← PUSH dallıdan sonradan 10FFC.
12	10FFD
C0	10FFE ← SP'nin ilk 95 tane değerler.
00	10FFF

Automatic olarak DAT JFFmpos

→ POP diyince 34 → DL'ye
 12 → BH'ye
 yarılır ve SP ilk yerine
 döner 2 artarak.

Örnek: Fibonacci ilk 24 terimin hesaplayıp, bunları büyükten

küçüğe doğru 2000h adresinden başlayarak yaz.

MOV AX, 00h ; AX = 0
 MOV BX, 01h ; BX = 1
 MOV CX, 23 ; CX = 23 (ilk fibonacci f0'dan başlıyor.)
 MOV DX, 00 ; DX = 0 Sonra bunda tutulacak

MOV DI, 2000h ; Verinin en son yazılacağı yer.

DONGU:

MOV DX, AX
 ADD DX, BX
 PUSH DX
 MOV BX, AX
 MOV AX, DX
 LOOP DONGU
 MOV CX, 23

; Dongu başı (Bu döngü fib. sıralını hesaplar.)
 ; DX'le AX değerini al
 ; DX ile BX'yi topla DX'ye yaz.
 ; Bu sonuc fib. serisinin bir elemdir. Stack'e yazar.
 ; BX'le AX'in önceki değerini al.
 ; AX'le önceki DX değerini al.
 ; Dongu basına git.

DONGU2:

POP AX
 STOSW
 LOOP DONGU2
 RET

; Dongu başı (Bu döngü verileri belirtilen adrese yazar,
 ; önce en büyük değeri alır ve AX'ye sıradan değer
 ; AY'lık değeri belirtilen adrese yazar. YES:DI) atar.
 ; Dongu basına git
 ; Return.

* Bu kodda önce fibonacci serisinin elemanları, tek tek hesaplanıp stack'e konur. SP, en son eklenen elemanın gösterdiğiinden önce stack'ten en büyük eleman pop edilir.

CALL Komutu:

CALL operand

→ Görevi: Fonksiyon çağrısı yapmak.

→ Algoritması: Belirtilen proc, label veya adressteki fonksiyonu çalıştırır.

Operand

procedure

label

4-byte address

* Etkilediği flagler: Hiaburisi

PROC Anahtar Kelimesi

Proc-ismi PROC → yeni bir prosedür oluşturulur.

ENDP Anahtar Kelimesi:

Proc-ismi ENDP → prosedürün bittiğini gösterir.

RET Komutu:

RET

→ Görevi: Prosedürün çağırıldığı yere geri dönmeyi sağlar.

→ Algoritması: IP'nin prosedür çağrısından önceki değerini geri iade ederek CALL keyword'under sonraki komut satırının -4'sini işaretlemeyi sağlar.

* Etkilediği flagler: Hiaburisi

örnek:

CALL deneme ; Deneme prosedüründe çağır. (1)

ADD AX,1 ; AX'e 1 ekle (4) AX:1235

RET ; İşletim sisteminde döñ (5)

deneme PROC ; Prosedür başlığı

MOV AX,01234h ; AX'ye 1234h atla (2)

RET ; Çağrılan yere döñ (3)

deneme ENDP ; Prosedür sonu

★ Prosedürler C'deki fonksiyonlara benzer.

★ Prosedür izlediği sırada registerler, flagler degisebilir.
Başta bunların degişmesini istemeyiz. (C'de void dönen
fonksiyonlar gibi) Bunun için bazı komutlarımız var.
PUSHA
Komutu

PUSHA

- Görevi: Genel amacılı registerleri stacke yükler.
→ Algoritması:

PUSH AX
PUSH CX
PUSH DX
PUSH BX
PUSH SP
PUSH BP
PUSH SI
PUSH DI

POPA Komutu

POPA

- Görevi: Stack'teki genel amacılı registerleri geri yükler.
→ Algoritması:

POP DI
POP SI
POP BP
POP xx (SP is ignored)
POP BX
POP DX
POP CX
POP AX

★ Etkiledikleri flagler: Hibritisi.

★ Böylece prosedürde bu değerler deguisse bile eski hallerine getirilebilirler.

PUSHF Komutu

PUSHF

- Görevi: Flagleri stacke yükler.

★ Etkilediği flagler: Hibritisi.

★ Böylece prosedür içinde flagler etkilense bile eski hallerine döndürülebilirler.

! Kullanım sırası önemli; önce PUSHF sonra PUSHA kullanıldığsa CALL'dan sonra önce POPA sonra POPF kullanılmalıdır.
(Stack mantığına göre)

POPF Komutu

POPF

- Stack'teki değeri flag'e yükler.

★ Etkilediği flagler: Hibritisi.

Önemli: Büyükkaz ve Küçük kaz prosedürleri oluştur ve Ekrana 'Hello world' yazdır.

PUSHA

PUSHF

LEA BX, hello

CALL BUYUKYAZ

LEA BX, world

CALL KUCUKYAZ

POPF

POPA

MOV AH, 04Ch

INT 21h

hello DB "Hello", 10, 13, 0

world DB "World", 0

BUYUKYAZ PROC

MOV AX, 00EH

DONGU_B:

MOV AL, [BX]

CMP AL, 0

JZ BITIR_B

CMP AL, 05Bh

JA BUYUT

INT 10h

INC BX

JMP DONGU_B

BUYUT:

SUB AL, 020h

INT 10h

INC BX

JMP DONGU_B

BITIR_B :

RET

BUYUKYAZ ENDP

KUCUKYAZ PROC

MOV AH, 00EH

DONGU_K:

MOV AL, [BX]

CMP AL, 0

JZ BITIR_K

CMP AL, 05Bh

JNA KUCULT

INT 10h

INC BX

JMP DONGULK

KUCULT:

ADD AL, 020h

INT 10h

INC BX

JUMP DONGUK

BITIR_K :

RET

KUCUKYAZ ENDP

; Registerleri stack'e at

; Flagleri stack'e at

; hello değişkeninin ilk adresini al

; BUYUKYAZ prosedürü çağır.

; world değişkeninin ilk adresini al

; KUCUKYAZ prosedürü çağır.

; Stackteki flagleri yerine koj

; stackteki registerleri yerine koj

; INT 21h/4Ch kesmesi için hazırlık

; 21h/4Ch kesmesi (Programı sonlandırır)

; Bundanlı 10,13 newline anlamına gelir.

; Değişkene değer ato.

; BUYUKYAZ prosedürü başlangıcı

; INT 10h/0EH kesmesi için hazırlık

; Döngü başı

; AL'ye BX adresindeki değeri al

; AL=0 mi diye bak

; 0 ise kelimenin sonuna gelmişiz demektir.

; AL 5Bh'dan (\rightarrow SAh) büyükse, büyük harftir.

; Eğer AL'de küçük harf varsa KUCULT git

; 10h/0EH kesmesi ekrana büyük karakteri yaz.

; BX bir sonraki elemanı göstermesi için artı.

; Döngü basına tən

; BUYUTmek için bu etikete gel

; Küçük harften 20h alarak büyük harf elde et.

; Büyük harfi ekrana bas

; BX'i artır.

; Döngü basına git.

; Kelimenin sonuna gelmişiz buraya atılır.

; CALL çağrısını geri döner.

; BUYUKYAZ prosedürü sonu.

; KUCUKYAZ prosedürü başlangıcı

; INT 10h/0EH kesmesi için hazırlık

; Döngü başı

; BX adresindeki değeri AL'ye aktar.

; AL=0 mi diye kontrol et

; Eğer sıfırsa kelime sonuna gelmişiz demektir.

; AL, 5Bh (\rightarrow SAh) büyük mi diye bak.

; AL küçükse, harf büyük demektir.

; Küçük harfi yazdır.

; BX'i artır.

; Döngü basına git.

; Harfleri küçültmek için buraya gel

; Büyük harf+20h ile küçük harfi elde et.

; Ekrana küçük harfi yaz.

; BX'i artır.

; Döngü basına git.

; Kelime sonuna geldiğimiizi belirten etiket

; CALL çağrısını dəsn.

; KUCUKYAZ prosedürü sonu.