Bilgisayar'ın Bileşenleri Nelerdir?

- CPU (İşlemci)
- Hafıza Birimi (RAM – ROM)
- I/O Portları

İki tür işletim sistemi vardır.

1. **Görsel Tabanlı** işletim sistemleri 32 bittir.
2. **Metin Tabanlı** işletim sistemleri komutla yazılır ve işletim sistemi 16 bite kurulabilir. 32 bitten aşağısına işletim sistemi kurulamaz.

İşletim sistemi kurulum dosyalarını harddiske yükler ancak birkaç dosya BIOS'a yüklenir. BIOS bir bilgisayardır, bünyesinde CPU - Hafıza Birim – I/O Portları barındırır ve bilgisayara bağlı birimleri kontrol eder. Harici olarak takılan driver'lar BIOS tablosuna kaydedilir. BIOS'da pil bulunur.

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

Bu program işlemci tarafından nasıl işlenir?

1- İşletim sistemi bu komutları RAM'e yükler.
2- RAM'e yüklenilebilmesi için değişkenlerin boyutlarına ihtiyaç duyulur.
3- Bu kodda atama ve toplama olmak üzere 2 tane komut bulunmaktadır.
4- c'nin içinde ne olduğunu bilmiyoruz ama RAM'de 4 byte yer kaplar.
5- İşlemci bu programı çalıştırmak istediği zaman c = a+b'den başlar.
6- İşlemci c = a+b komutunu almak için memory'ye adres gönderir ardından komutu alır.

int a = 10;
int b = 20;
int c = a+b;
3 adet fetch var.
1 adet execute var.
	int c = a+b+10;
	2 adet execute var.

İki program (kod) yazıp çalıştırdığımızı varsayalım bunların hızları aynı mıdır?

Hız olarak ikiside aynıdır çünkü herhangi bir program işlenmeden önce ASM diline dönüştürülmelidir ve ASM dilinde karmaşık kod bulunmaz. (**Karmaşık kod,** tek bir satırda birden fazla komutun (talimatın) yer almasıdır yani c = a+b-c; gibi.)

Boyut açısından bakıldığı zaman?

ASM'ye çevrildiği için ikisinin boyutuda aynıdır.

- Kullandığımız bilgisayarlarda **veri yolu – adres yolu – kontrol yolu** bulunmaktadır. Bunlar birimler arası iletişimi sağlar.
- Adres yolu tek bir durumda **çift taraflı** olur o da iki tarafta da memory bulunması halinde.
- **RAM'in** ve **önbelleğin** görevi veri erişimini hızlandırmak ve cihazların daha hızlı çalışmasını sağlamaktır. Önbellek işlemcinin dışında konumlandırılmaktadır.
- Komut, işlemcide yavaştır bu yüzden önbellek ortaya çıkmıştır.
- Bir birimde işlemci varsa **hafıza birimi (RAM-ROM)** de bulunmalıdır.
- Bir bilgisayarda ekran kartı dahili ise RAM yani hız etkilenir çünkü ekran kartı bilgisayarın işlemcisini ve hafıza birimini kullanır bu da ekstra iş gücü demektir.
- RAM'in ister 0. adresine ister 5. adresine ister de 10. adresine aynı sürede erişiriz ve bu adreslere decoder sayesinde erişiriz. Decoder verilerin anlamlı hale getirilmesidir.

Hafıza birimlerinin küçükten büyüğe sıralanışı:
Bit – Byte – Kilobyte – Megabyte – Gigabayt – Terabayt
Klasörün neden bir uzantısı yok?
Çalıştırması gereken bir program olmadığı için.

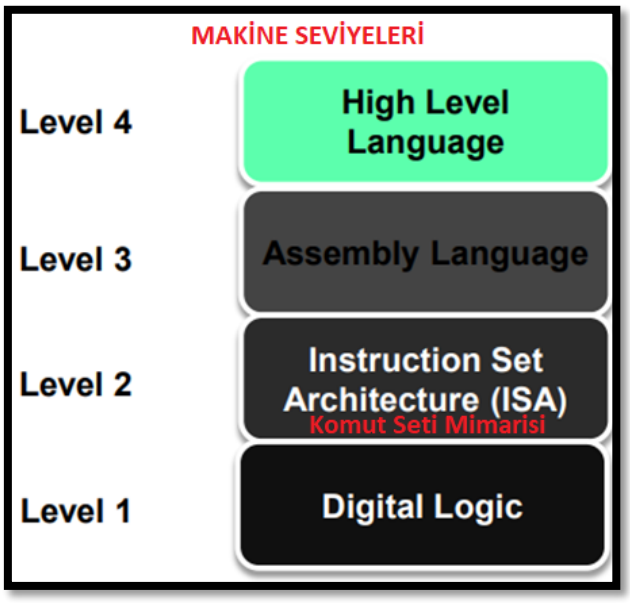
Bir Word dosyası açtığımızı varsayalım. Kaydedeceğimiz veri 300 byte olsun. Bu dosyayı iki farklı bilgisayara kaydedelim. Birisinde 400 byte RAM diğerinde de 100 byte RAM bulunsun. 400 byte RAM bulunan bilgisayar tek seferde Word dosyasını kaydeder ancak 100 byte RAM bulunan bilgisayar önce 100 byte'lık veriyi kaydeder daha sonra bu silinir ondan sonraki 100 byte'lık veri kaydedilir son olarak bu veri de silinip kalan 100 byte'lık veri kaydedilir.

Bir program ASM diline ne zaman dönüştürülür?

1- Çalıştırıldığı zaman
2- Kaydedildiği zaman

İşletim sistemi kaydettiğimiz programı ya ilerde çalıştırırsak diye ASM diline dönüştürür.

Kaydetmenin İçinde Ne Var?	**Çalıştırmanın İçinde Ne Var?**
1- Derlenir. 2- ASM diline dönüştürülür. 3- Makine diline dönüştürülür. 4- Harddisk'e kaydedilir.	1- Derlenir. 2- ASM diline dönüştürülür. 3- Makine diline dönüştürülür. 4- Harddisk'e kaydedilir. 5- Memory'ye yüklenir ardından işlemci tarafından işlenir.



- **İlk programlama dili** makine dilidir ardından Assembly dili son olarak yüksek seviyeli diller (C, C++, Python, Java gibi) gelir.
- ASM'ye çevrildikten sonra .exe (binary) dosyası oluşturulur. (Windows'a özel.)
- Yüksek seviyeli diller oluşturulduktan sonra **object dosyasıda** oluşturulur.
- Java **sınıf tabanlı** bir programdır. Sınıf tabanlı programlar **bağımsızdır**. (İstenilen bilgisayarda, istenilen işletim sisteminde çalıştırılabilir.) Dolayısıyla Java bağımsızdır ve bağımsızlığını **byte code** yapar.

Linking (Kütüphane)	Loading
Derleyici tarafından yapılır.	İşletim sistemi tarafından yapılır.
Kütüphane kullanırsak linking yapılır.	Harddisk ile RAM arasında gerçekleşir. (Hızlı olsun diye)
Hangi komutu kullandıysak kütüphaneden o komutla ilgili bilgileri getirir.	Loading, yazdığımız programın RAM'e nasıl kaydedileceğine karar verir.

Assembly yürütülebilir bir program oluşturmaz. Yürütülebilir program için aşağıdakiler gereklidir:

Program Çalıştırma Adımları

1- Assembly diline dönüştürülür. (Kod Binary formatına (makine diline) dönüştürülür.)
2- Linking gerçekleştirilir. (Kütüphaneye gidip komutları getirir.)
3- Loading gerçekleştirilir. (Program, RAMe taşınır.)
4- Execute gerçekleştirilir.

Uygulamaya her tıkladığımzda 'loading' olur çünkü uygulamanın komutları belleğe yüklenir.
Execution aşaması da gerçekleşip uygulama çalıştırılır.

SINAVDA PROGRAM ÇALIŞTIRMA ADIMLARI SORULURSA EXTRA OLARAK KOMUT ÇALIŞTIRMA ADIMLARINIDA YAZMAYI UNUTMA!

Komut Çalıştırma Adımları

1- Komut getirilir. **FETCH (İşlemci ile hafıza arasında ya da 2 tane memory arasında gerçekleşir.) (RAM'den kaydedicilere gider.)**
2- Kod çözülerek anlamlı hale getirilir. **DECODE (Kontrol birimi tarafından gerçekleştirilir.)**
3- Komut neyse o çalıştırılır. **EXECUTE (Execute, ALU veya çıkış portlarında gerçekleştirilir.)**

1 Byte = 8 Bits
1 Word = 2 Bytes = 16 Bits
1 Doubleword = 2 Words = 4 Bytes = 32 Bits
1 DWord = 2 Word = 4 Bytes = 32 Bits

- **Clock,** CPU işlemlerini senkronize etmekle görevlidir. **Clock,** tek bir komutun işlenme süresidir. İşlemcinin senkronize (sıralı) bir şekilde çalışmasını sağlamak için **clock** kullanılır. Clock süresi de **en uzun komutun** süresine göre ayarlanır.
- **Bir sonraki clock önbelleğe aktarılırken kimin yerine yazılır?**
  Önbellek tarafından en çok kullanılan değiştirilmez onun yerine başka bir değişken silinip yerine yazılır.
- **Clock neye göre ayarlanır?**
  İşlemcideki komutlardan maksimum olan değere göre ayarlanır.
- **Control Unit (Kontrol Birimi)** yürütme adımlarını koordine eder.
- **ALU,** aritmetiksel ve mantıksal işlemin yapıldığı alandır.
- **Program Counter,** çalıştırılmakta olan komutların adresi tutulur.

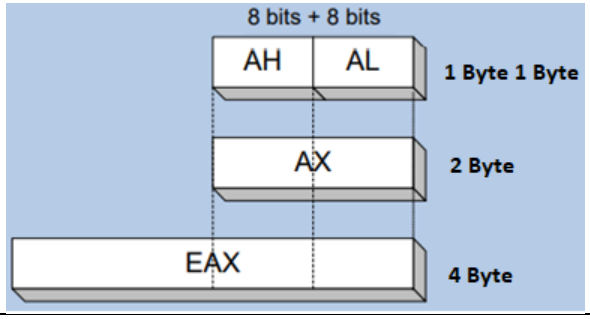
- İşlemci ile memory arasında clock farklıdır, işlemcide lojik kapıları olduğundan daha hızlıdır.
- **Wait Cycle** okuma ve yazma işlemi yapar. Wait Cycle'ın olma sebebi işlemcinin memory'den (RAM) daha hızlı olmasından kaynaklanır.
- Okuma işlemi komut ve veri üzerinde yapılırken yazma işlemi sadece veriler (değişkenler) üzerinde yapılır çünkü komut yazdırma şansımız yok.
- İlk programı çalıştırdığımız zaman önce program komutlarıyla değişkenler memory'ye kaydedilir (Loading). (Lakin birtakım değişkenler önbelleğe kaydedilir.)

KAYDEDİCİLER (REGISTERS)

Aritmetiksel işlemler ve veri aktarımı için kullanılır.

Kaydedicilerin boyutu en fazla 64 bittir.

Kaydediciler **genel amaçlı** ve **özel amaçlı** olmak üzere ikiye ayrılır. Genel amaçlı kaydedicilere istenildiği gibi erişilebilir.



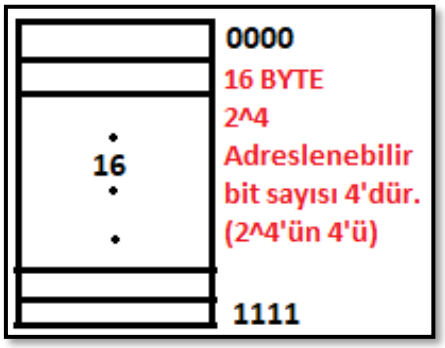
RAM 3 parçada kaydedilir.

1- **Kodlar (Code Segment)**
2- **Veriler (Data Segment)**
3- **Yığın (Fonksiyonlar kaydedilir.) (Stack Segment)**

İlk bilgisayarın özellikleri:

- İşlemcisi 8088
- 1 MB RAM = 2^{20} (20 tane adresleme biti var.)
- 16 bit register (kaydedici)
- 16 bit data bus

İşlemcinin 16 biti var RAM'in ise 20 tane adresleme biti var o yüzden kullanamaz. 2 tane kaydedici yan yana getirilerek problem çözülür.

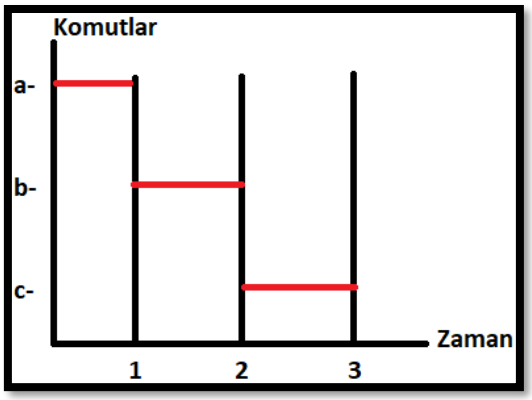


Single Cycle: Her clock'ta bir komut gerçekleşir. Yani bir programda 10 komut varsa her clock'un süresi 1 sn ise program 10 sn'de gerçekleşir.

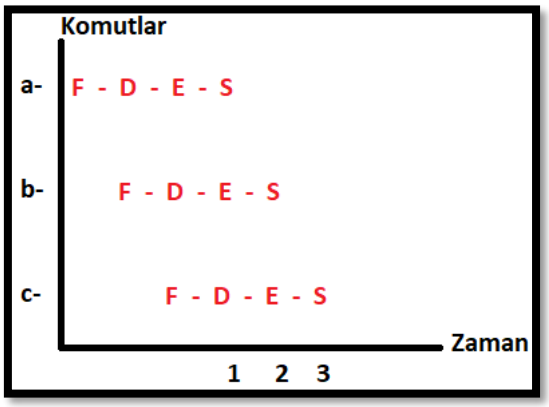
Multi Cycle: Komutları parçalara böler ve her clock'ta belli parçalarını getirir. Bu yüzden bir komutun kaç clock'ta biteceği değişkendir. Yani bir clock'ta birden fazla komut işleyebilir.

INTEL 80386 bilgisayarlarda **single cylce**, INTEL 80486'den sonraki bilgisayarlarda **multi cycle** vardır.

3 tane komutumuzun olduğunu varsayalım. (int a=5+3, b=2+3, c=4*2;)
Single cycle'da 2. komut 1. komut bittiğinde yani 1 sn sonra fetch yapılır.



Multi cycle'de 1. komut execute yapılırken,
2. komut decode yapılır.
2. komut decode yapılırken,
3. komut fetch yapılır.



Single cycle'da pipeline olmaz ancak multi cycle'da pipeline olur.

1- **Pipeline**
   - Pipeline, talimatların aynı anda yürütülmesidir. Pipeline biri işin süresini değiştirmiyor yapılan iş miktarını arttırıyor.
   - Execute'u uzun sürede yapılır çünkü komut karmaşıktır dolayısıyla uzun sürer.
2- **HyperThreading:** Bir işlemcide aynı anda iki işlemin yapılmasıdır. Bir core olmasına rağmen (yazılımsal olarak) sanki iki core varmış gibi davranmasıdır.
3- **SuperScalar,** aynı clockta birden fazla komut çalıştırabilen teknolojidir.
Pipeline'ın çalışma süresi uzun olduğu için işlemler gecikiyordu bu da maliyetin artmasına sebep oluyor bu yüzden SuperScalar teknolojisi geliştirildi.
4- **Compatibility Mode (Uyumluluk Modu)** kısaca, 64 bit bir bilgisayarda 32 bit ve 16 bit yazılımları kullanamaya izin veren moddur.

CISC KOMUT MİMARİSİ	RISC KOMUT MİMARİSİ
Çok fazla komut sistemine sahip olduğu için CISC tercih ediliyor.	RISC, CISC'e göre daha hızlıdır çünkü komut devreleri daha basittir.
Güncelleme işlemi zordur.	Güncelleme işlemi kolaydır.

Üç tür **hafıza yönetimi** vardır.

1- **Koruyucu (Protected) Mode** --> Segment
   Komutlar istediği gibi hafızadaki bölgelere erişemez.
   Ulaşılmak istenen hafıza bölgeleri descritpor'lerde tutulur.
   Programlar, **segment** bölgesinde çalışır.
2- **Real-Address Mode**
   Tüm komutların hafızaya erişimi serbesttir.
   Segment yok bu yüzden tek bir program çalıştırabiliriz.
3- **System Management Mode**
   Sadece işletim sistemi komutları hafızaya ulaşır ve işletim sistemi kullanır.

Bizim kullandığımız bilgisayarlarda üç tür hafızanın üçüde kullanılır.

Virtual Memory'nin (Sanal Hafıza) (Paging)

RAM'de yer kalmayınca sabit diskin bir bölümünü RAM olarak kullanmaktır.

- **LSB(Least Significant Bit)**, binary gösterimin en sağındaki yani en küçük değere sahip bittir. (**MSB(Most Significant Bit)**, binary gösterimin en solundaki yani en büyük değere sahip bittir.

- **Flat Segment Model:** Her program için bir tablo vardır.
- **Multi Segment Model:** Tüm programlar için bir tablo vardır.
- Yüksek seviyeli dil ASM diline dönüştürüldüğünde object dosyası oluşur. Ve sonra listing dosyası oluşur.
- Linking'i linkar yapar.
- Adres işlemciden memory'ye (RAM'e) gider.
- **stc:** Bayrak bitlerini 1 yapıyor. (0'da olabilir.)

Memory'nin Yapısı

1. **Data**
   Tüm programları çalıştırıp, ihtiyaç duyulduğu zaman RAM'e (memory'ye) yükler bu da bize RAM'de kullanım alanı açılmasını sağlar dolayısıyla verim elde edilmiş olunur.
2. **Code**
3. **Stack 100h**

Bilgisayardaki I/O Öncelik Seviyeleri

1. **BIOS**
2. **İşletim Sisteminin Güvenliği**
3. **Bizim Yazmış Olduğumuz Programlar** **---->** Word, powerpoint, C++

Basic Elements

1. **Label ----------> RAM'deki adres**
2. **Directives (Yönlendirme)** ----------> Komutlara directives denir
3. **Opicod (Mnemonics) and Operands** ------> Komutların yanında aldığı değişkenlerdir.
4. **Yorum Satırları**

Listing File'ın İçindekiler:

1. **Source Code**
2. **Addresses**
   Programı çalıştırabilmek için RAM'den adrese ulaşmamız lazım.
3. **Segment İsimleri**
4. **Object Code**
5. **Symbols** (Variables, Procedures and Constants)

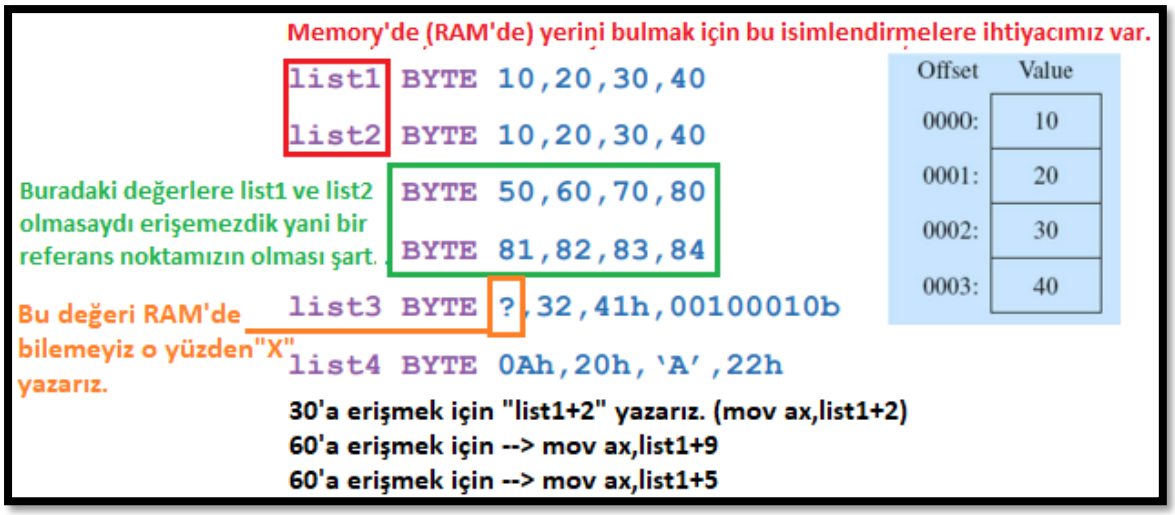
NOP (No Operation) Instructions

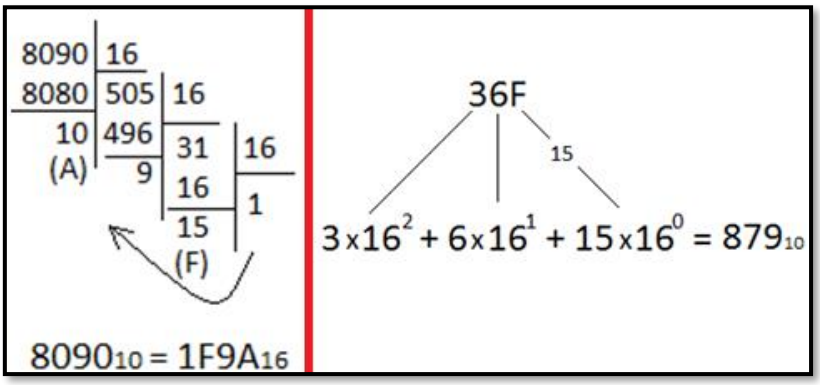
NOP, tek komutları çifte tamamlayarak okumanın daha hızlı olmasını sağlar.
BİLGİ: Memory'de (RAM'de) çift şeklinde okuma işlemi teke nazaran daha hızlıdır.
Rakamlar tek olursa okuma işlemini yavaştır sebebide devresinin karmaşık olmasıdır.
Okuma işlemi yapılırken 3 bayt işlemden sonra NOP eklenir ve 4 bayt'a ulaşılır bu sayede çift rakama ulaşmış olunup okuma işlemi daha hızlı olacaktır.

ALIGN Directive

ALIGN, tek komutları çifte tamamlayarak okumanın daha hızlı olmasını sağlar. ALIGN ve NOP arasındaki fark; ALIGN çift çift eklenerek artar.

```
bVal BYTE ?;          00404000
ALIGN 2 ;
// Eğer biz burada ALIGN yapmasak bir sonraki komutun adresi 00404001 olacaktı. Ancak RISC bu
// adrese gidip iş yapamayacağı için adresine 1 byte daha ekliyor.
wVal WORD ?;          00404002
bVal2 BYTE ?;         00404004
ALIGN 4;
// Burada ALIGN yapmasak bir sonraki komutun adresi 00404005 olacaktı. Yine tek adrese denk
// gelecekti. Ama biz dedik ki; 'ben 4 byte'lık yer ayırıcam. Ama üstteki komut zaten bunun 1
// byte'ını almış. Ben 3 byte ekleyeyim de sonraki adrese tam 4 eklenmiş olsun.'
dVal DWORD ?;         00404008
dVal2 DWORD ?;        0040400C
```

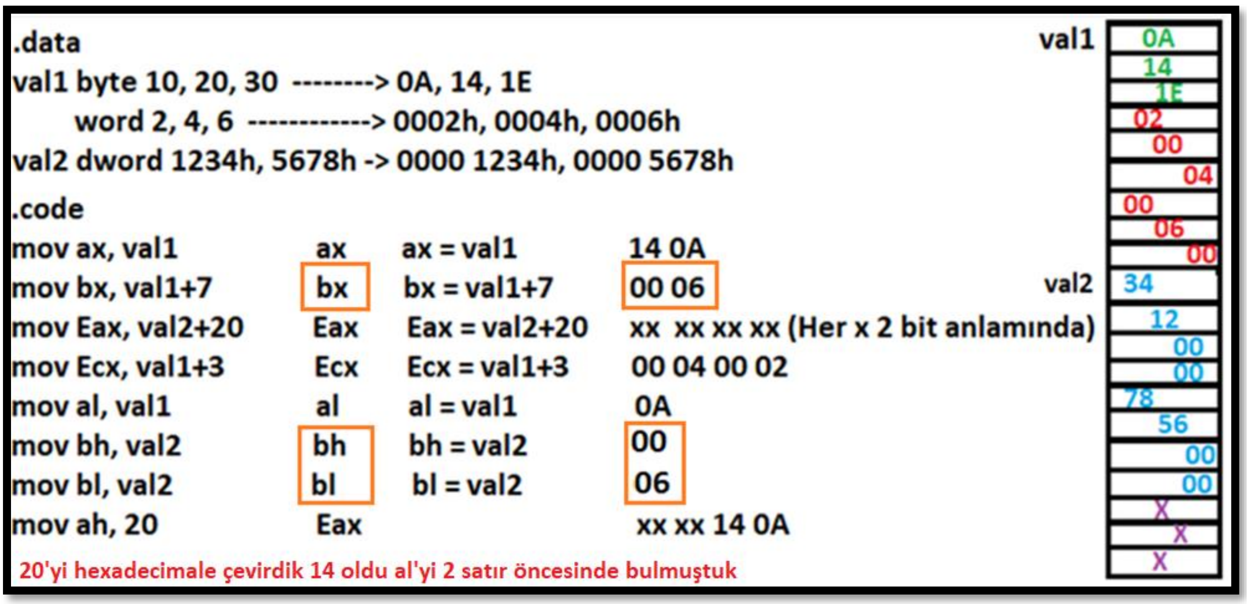




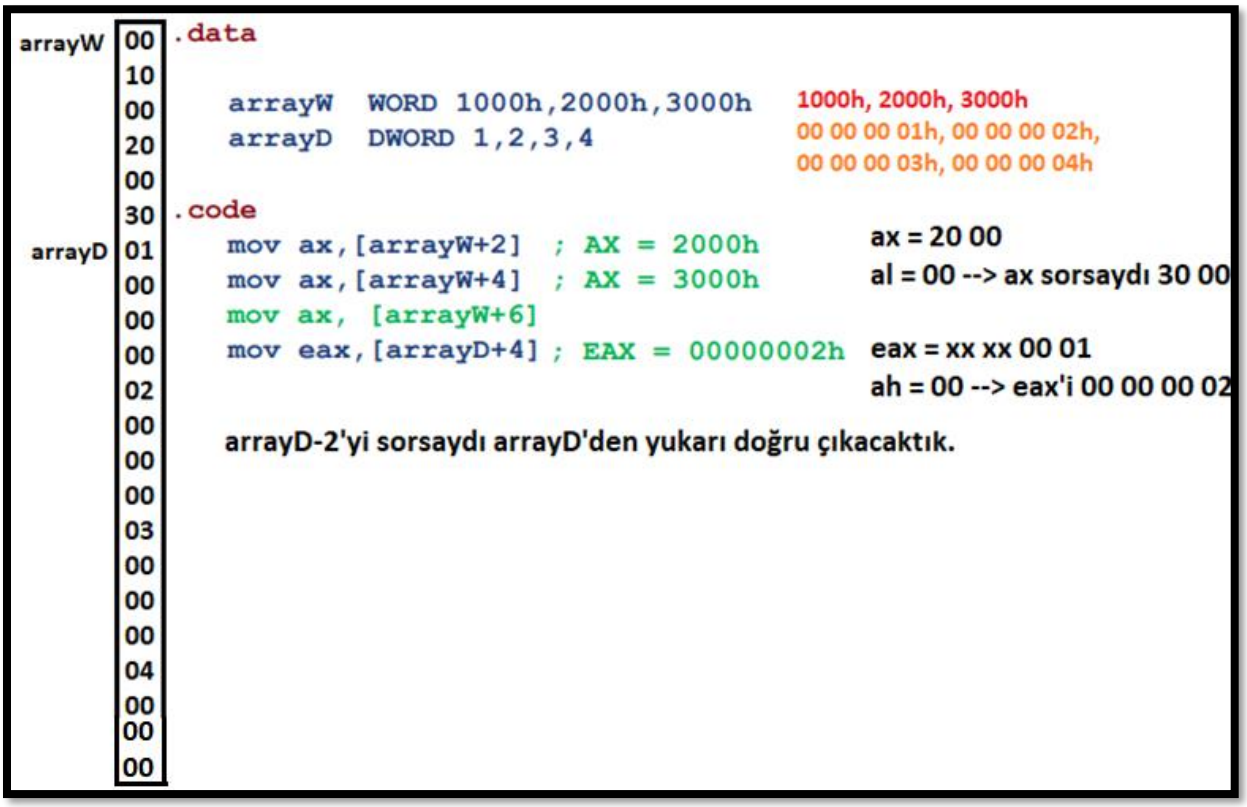
Little Endian, değerleri LSB'den başlayarak kaydeder.
Bir byte'dan daha büyük bütün veri tipleri kendi verilerini RAM'e ters sırayla yazar. Ancak bu, sadece sayıları için geçerlidir. Komutlar için bu geçerli değildir. Mesela 0AB2 komutu ters şekilde yazılmaz 2 byte olmasına rağmen.
Big Endian, değerleri MSB'den başlayarak kaydeder. İntel işlemcili bilgisayarlar **Little Endian** kullanır.
NOT: İster Little Endian ister Big Endian olsun register'a yazma işleminde memory'deki adres nereden başladıysa register'da LSB'ye yazılır.
NOT: Byte'ın dışında diğer boyutlar ters şeklinde yazılır!!
Ters: İlk (küçük) adrese değerin MSB biti yazılır. Yani MSB kısmını memory'ye yazarken önce yaz.

Verilen BYTE değeri tek haneliyse 2 haneliye tamamla ve hexadecimal'e dönüştür.
Verilen WORD değerini 4 haneliye tamamla ve hexadecimal'e dönüştür.
Verilen DWORD değerini 8 haneliye tamamla ve hexadecimal'e dönüştür.

DUP; içeriği, parantez içerisinde belirtilen, bellekte yer ayrılan sabit bir değerdir.
Aşağıdaki örnekte birinci elemanı 10, 2. , 3. ve 4. elemanları 0 olan (3 byte'lık yer ayrıldı), 5. elemanı ise 20 olan 5 byte'lık bir dizi oluşturuldu.



```
sayi1 dword 1, 2, 3, 4, 5 ---> 0000 0001, 0000 0002, 0000 0003, 0000 0004, 0000 0005
sayi2 byte    1, 2, 3, 4 -------> 0h1, 02h, 03h, 04h
sayi3 word 23h, 78h --------> 0023h, 0078h
      dword 23h, 48h --------> 0000 0023h, 0000 0048h
```



ASSEMBLY DİLİ HAKKINDA

- **Mov komutu,** kaydedicilere değer atar.
- Syntax'ı, **'mov hedef, kaynak'** yani kaynaktaki veri, hedefe aktarılır.
- Immediate, hedefte yer alamaz çünkü immediate değerlerin adresi olmaz.
- Hedefte adres bulunmalıdır dolayısıyla hedef, ya memory ya da kaydedici olmalıdır.

İmmediate; integer (sabit) değerleri (1, 3 gibi) ifade eder.
Register; ax, bx gibi değerleri ifade eder.
Memory (RAM); val1, sayi2 gibi değerler ifade eder yani adres içerir.

Constans

int a = 5;

5, immediate bir değerdir. 'a' ise memory'dir. Biz burada memory'e immediate değer atadık.

Strings

str3 BYTE 'A','E','I','O','U'

✓ Byte tipinde bir dizi oluşturur. Ancak direk String olarak değerlendirilmediği için satır sonuna 0 koymaya gerek yok.

menu BYTE "1. Oluştur", 0dh, 0ah,
"2. Kaydet", 0dh, 0ah,
"3. Sil", 0ah, 0ah,
"Seç", 0

✓ "1. Oluştur" yazar aşağıya iner, "2. Kaydet" yazar, aşağıya iner, "3. Sil" yazar ve aşağıya iner, "Seç" yazar ve 0 ile bitirir.
✓ 0Dh ---> Öncesinde gelen stringin bittiğini belirtir.
✓ 0Ah ---> Bir satır aşağıya iner.
✓ Her 0ah'den önce 0dh geliyor. Şöyle düşünülebilir. 0ah yazılırken yani bir satır aşağıya inileceğinde string'in bunu bilmesi gerekiyor. 0dh ise bunu bildiriyor.
✓ String'in bittiğini göstermek için string sonuna virgül ve 0 atılır.
✓ String'in tipi byte'dır. Her harf 1 byte'dir.

Array Size (Dizi boyutu)

$ -> şuanki satırın konumunu verir.

list BYTE 10, 20, 30, 40
listSize = ($ - list)

✓ byte tipinde list adında bir liste oluşturur.
✓ listSize ile listenin boyutunu öğrendik yani 4 byte boyutunda.

list2 DWORD 1,2,3,4
listSize2 = ($-list2)/4

DWORD olduğu için **4'e böldük**. WORD olsaydı 2'ye bölecektik.

EQU

✓ Sabit değerler için atama yapar.

var EQU 80

✓ var isimli bir sabit oluşturur ve 80 değerini atar. Bu sabit bir daha değiştirilemez. İçine herhangi bir değer atanabilir.

TEXTEQU

✓ String değerler için atama işlemi yapar.

msg TEXTEQU 12
msq TEXTEQU %(msg)

✓ değeri 12 olan msg adında bir sabit tanımlar.
✓ '%' ifadesi integer'i string'e dönüştürür.
✓ TEXTEQU ile EQU'nun farkı, TEXTEQU'nun sonradan değiştirilebilir olmasıdır.

Mov Komutu Hata Durumları

.data	***.code***
bVal BYTE 100	***mov al, wVal*** -----> wVal 2 byte, al ise 1 byte. Boyutları farklı olduğu için hata
bVal2 BYTE ?	***mov bVal2, bVal*** ----> Memory'den memory'e direk atama yapılamaz.
wVal WORD 2	***mov ds, 45*** --> ds, bir segment register'dir. Segment register'e atama olmaz.
dVal DWORD 5	***mov 25, bVal*** --------> immediate ifadelere atama yapılamaz.

Zero Extension (MOVZX Instruction)

Daha büyük boyuta sahip bir register'e (ax gibi), daha küçük boyuta sahip bir değer atandığında kalan kısım 0'lar ile doldurulur

mov BL, 10001111b

movzx ax, BL **En sol bit 0 ise 0 ile tamamla --> movzx**

Boyutu 1 byte olan BL register'ine, binary bir değer atanıyor. Daha sonra bu register'i, ax registerine atamaya çalışırsak ax'in boyutu 2 byte olduğundan kalan 1 byte 0'lar ile dolacaktır. AX yani destination operand (hedef) her zaman register olmalıdır.

Sign Extension (MOVSX Instruction)

Daha büyük boyuta sahip bir register'e, daha küçük boyuta sahip bir değer atandığında fazla kalan kısım, source'nin sign (işaret) biti ile doldurulur.

mov BL, 10001111b

movsx ax, BL **En sol bit 1 ise 1 ile tamamla --> movsx**

AX yani destination (hedef) operand'ı her zaman register olmalıdır.

XCHG Instruction (Swap = Exchange)

2 tane kaydedici kullanılarak swap işlemi yapılır. İki operand'dan en az biri register olmalıdır.

XCHG val1, val2 ----> Yanlış. İki operand'da memory olamaz çünkü adres içeriyor.

Operandlardan herhangi biri immediate olamaz. Çünkü immediate değer depolanmaz ki değeri değişebilsin.

.data

var1 WORD 1000h

var2 WORD 2000h

.code

xchg ax, bx;	exchange 16-bit regs
xchg ah, al;	exchange 8-bit regs
xchg var1, bx;	exchange mem, reg
xchg eax, ebx;	exchange 32-bit regs

LAHF & SAHF Instructions

LAHF, FLAG registerlerinin low byte'ını ah registerine kopyalar. Bu registerler arasında Sign, Carry, Zero flag'ları da vardır.

.data

saveFlag BYTE ?

.code

lahf

mov saveFlag ah; -------> lahf ile ah'ye kopyalanan flag değerlerini saveFlag'a aktardık.

SAHF, AH registerindek flag değerlerini flag'lara geri kopyalar.

.code

mov ah, saveflags; -----> saveflags değişkenindeki flag'ları ah ye geri atıyoruz.

INC & DEC Instructions (Bir Artırma & Bir Eksiltme)

INC bir artırır, DEC bir azaltır. Carry Flag'in durumunu etkilemezler.

```
.data
        myWord WORD 1000h
        myDword DWORD 10000000h
.code
        inc myWord ;          1001h
        dec myWord ;          1000h
        inc myDword ;         1000 0001h
        mov ax, 00FFh         (255=00FF)
        inc ax ;              ax = 0100h
```

ADD & SUB Instructions (Toplama & Çıkartma)

```
.data
        var1 DWORD 10000h
        var2 DWORD 20000h
.code
        mov eax, var1;  -------> eax registerine var1 değişkenini atar.   00010000h
        add eax, var2;  -------> eax'e var2'nin değerini ekler.            00030000h
        add ax, 0FFFFh; -----> ax'e  0FFFFh değerini ekler.                0003FFFFh
        add eax, 1;  ----------> eax registerine 1 değerini ekler.         00040000h
        sub ax, 1;  ----------> ax registerinden 1 çıkarır.                0004FFFFh
```

NEG Instruction

NEG'in amacı 2'ye tümlemektir yani önce tersini alıp sonra 1 ile toplamak.

```
.data
      valB BYTE -1
      valW WORD +32767
.code
      mov al, valB;          AL = -1
      neg al;                AL = +1
      neg valW;              valW = -32767
```

Eğer WORD'ün alabileceği boyuttan daha fazla bir atama yaparsak ve bunun negatifini almaya çalışırsak bize yanlış sonuç döndürecektir. Taşma olacağından dolayı overflow biti 1 olacaktır.

FLAGS

Her komutta tüm bayrak kaydedicilerinin bitleri duruma göre değişir. Peki neden?

Mov yani atama komutunda bayrak biti hiçbir zaman etkilenmez çünkü bayrak kaydedicisi işlemcinin içindedir mov komutu ise RAM'dedir.

Zero Flag, işlemin sonucu 0 ise flag 1 olur.

mov cx, 1
sub cx, 1; cx = 0 ZF = 1

Sign Flag, işlemin sonucu negatif ise flag 1 olur. Pozitife ise flag 0 olur.

sub cx, 1; cx = -1 SF = 1
add cx, 2; cx = 1 SF = 0

Sign flag, en büyük bitin aynısıdır. Yani MSB biti 1 ise 1, 0 ise sign flag'ı 0'dır.

mov al, 0
sub al, 1; al = 11111111b, SF = 1
add al, 2; al = 00000001b, SF = 0

Carry Flag, size (boyut) hatası çıkarsa flag 1 olur (unsigned olarak değerlendirilir).

mov al, 0FFh (255 = 0FF)
add al, 1; CF = 1 al = 00
// al register'inin boyutu 1 byte'dır. 0FF'de 1 byte yer kaplar. Ancak buna 1 eklendiği zaman al registerinin boyutu aşılacak ve carry flag 1 olacak.
mov ax, 0FFh (255 = 0FF)
add ax, 1
// ax registerinin boyutu 2 byte'tır. 0FF de 1 byte yer kaplar. Buna 1 eklendiği zaman al registerinin boyutu aşılmayacak ve carry flag 0 olacak.

Overflow Flag, aritmetiksel işlemlerde işaret hatası çıkarsa flag 1 olur.

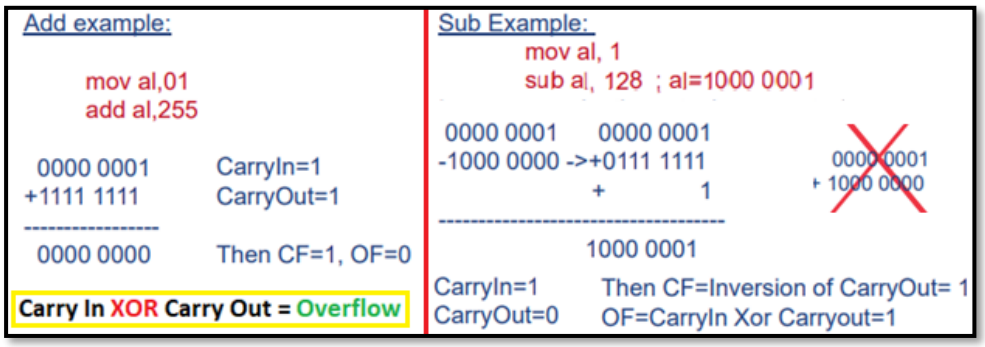
mov al, +127
add al, 1; OF=1 AL = -128
// 127 ile 1'i toplayınca cevap -128 çıkıyor. Bu yüzden işaret hatası olur yani OF=1

mov al, 7Fh; (+127=7F) OF=1 AL = 80h (-128)
add al, 1
// 127 ile 1'i toplayınca cevap -128 çıkıyor. Bu yüzden işaret hatası olur yani OF=1

Toplanan iki değer pozitif ise ya da çıkarılan değerlerin işareti farklı olduğunda overflow bit 1 olur. Aksi duruda overflow durumu oluşamayacaktır. Kontrol etmeye gerek yok.

mov ax, 00FFh (255 = 00FF)
add ax, 1; ax = 0100h SF=0 ZF=0 CF=0
// Boyutu 2 byte, değeri 00FFh olan ax registerine 1 eklemek boyut sorununa neden olmaz.
// Bu yüzden CF=0; Sayımız pozitif olduğu için SF=0; ax'ın değeri 0 olmadığı için ZF=0
add al, 1; AL = 00h SF=0 ZF=1 CF=1
// Boyutu 1 byte, değeri 00FFh olan al registerine 1 eklemek boyut sorununa neden olur.
// Çünkü 1 byte'lık registerin üstüne çıkmış oluyoruz. Yani CF=1; 1 ile toplayınca boyut sorunları ortaya
// çıksa da al'nin 8 biti 0 olmaktadır. Bu yüzden ZF=0; MSB biti 0 olduğu için SF=0

mov al, 2
sub al, 3; AL=FFh SF=1 ZF=0 CF=1
// 2 -3 = -1; -1'in karşılığı: 1111 1111'dir. MSB biti 1 olduğundan SF=1, sonuç 0 olmadığı için ZF=0, CF=1



PTR Operator
Örneğin sayi diye bir değişkenimiz olsun. Tipi de **WORD** olsun. Bu değişkene ileride **DWORD** tipinde bir değer atmak istersek sayinin tipini **PTR** ile varsayılan tipini DWORD olarak değiştirebiliriz.
.data
myDouble DWORD 12345678h
.code
mov ax, myDouble **ERROR**
// AX'ın boyutu 2 byte'dır. Ancak AX'e, 4 byte olan değişkenimizi atamaya çalışıyoruz.

mov ax, WORD PTR myDouble
// Diyoruz ki: 'ax registerine myDouble değişkenimi at. Ama atarken tipini WORD tipine (2 byte)
// çevir de at. Böylece tipleri aynı olur ve atama işlemi başarılı olur.'

Aşağıda PTR operatörü kullanılmış bir değişkenin adres yerleşimini görüyoruz.

NOT I: PTR, sadece boyutu büyük değişkenlerin boyutunu daraltmaz. Boyutu daha küçük olan bir değişkeni boyutu büyük olan bir değişkene atabilir.

```
.data
   myBytes BYTE 12h,34h,56h,78h
.code
   mov ax,WORD PTR [myBytes]   AX = 3412h
   mov ax,WORD PTR [myBytes+2  AX = 7856h
   mov eax,DWORD PTR myBytes   EAX = 78563412h
```

```
a BYTE 10h
b WORD 2040h
mov al, BYTE PTR b+3      al=XX
    'al, b' olmaz çünkü boyutları farklı
mov eax, DWORD PTR        eax=XX204010
```

10
40
20
X
X
X
X
X

```
.data
   varB BYTE 65h,31h,02h,05h
   varW WORD 6543h,1202h
   varD DWORD 12345678h
.code
    mov ax,WORD PTR [varB+2] ; ax=0502h
    mov bl,BYTE PTR varD     ; bl=78h
    mov bl,BYTE PTR [varW+2] ; bl=02h
    mov ax,WORD PTR [varD+2] ; ax=1234h
    mov eax,DWORD PTR varW   ; eax=12026543h
```

65
31
02
05
43
65
02
12
78
56
34
12

Type Operator (return gibi boyut neyse onu döndürür)

Return gibi düşünebiliriz dizinin boyutu ne ise bize onu geri değer olarak döndürür.

```
.data
        var1 BYTE ?
        var2 WORD ?
        var3 DWORD ?
.code
        mov eax, TYPE var1;       1
// var1'in tipini döndürür. var1'in tipi ise 1 byte'dır.
        mov eax, TYPE var2;       2
        mov eax, TYPE var3;       4
```

Lengthof Operator

Dizinin uzunluğunu yani eleman sayısını verir.

```
.data
        var1 BYTE 10, 20, 30
        array1 WORD 30 DUP(?), 0, 0
        digitStr BYTE "12345678", 0
.code
        mov ax, LENGTHOF var1;          3
        mov ecx, LENGTHOF array1;       32
        mov edx, LENGTHOF digitStr;     9
```

SIZEOF Operator (Sizeof = Lengthof * Type)

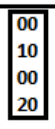
Bir değişkenin **eleman sayısı * veri tipini boyutunun** çarpımına eşittir.

```
.data
        digitStr BYTE "12345678", 0;      9
        array1 WORD 30 DUP(?), 0, 0       64 = 32*2
// Tipi WORD (yani boyutu 2 BYTE olan) 32 elemanlı bir dizi tanımladık.
.code
        mov ecx, SIZEOF array1;           ECX=64
// array1'in boyutunu alıp ecx registerine atar.
```

Label Operatoru

Var olan bir hafıza alanına isim takar. Memory'de label yazan komut için herhangi bir yer ayrılmaz. Pointer operatöründen kaçınmak için ihtiyaç duyulur.

```
.data
     dwList    LABEL DWORD
     wordList  LABEL WORD
     byteList  BYTE 00h,10h,00h,20h
.code
     mov eax,dwList    ; 20001000h
     mov cx,wordList   ; 1000h
```



eax ve cx registerlerinin değerleri aynıdır sebebi ise hem dwList hem de wordList, aynı bellek bölgesine takılmış isimlerdir.

OFFSET Operator (ESI)

Herhangi bir adresin (val1, sayi2 gibi) öncesine OFFSET yazdığımız zaman RAM'deki adresi getirmiş oluruz. İçindeki değeri vermez.

```
.data
        bVal BYTE ?          Varsayalım bVal, "00404000h" adresinde.
        wVal WORD ?
        cVal DWORD ?
        dVal DWORD ?
.code
        mov esi, OFFSET bVal;       ESI = 00404000
        mov esi, OFFSET wVal;       ESI = 00404001
// bVal+1 byte, wVal'ın adresini verir.

        mov esi, OFFSET cVal;       ESI = 00404003
// wVal, WORD yani 2 BYTE. wVal+2, cVal'ın adresini verir.

        mov esi, OFFSET dVal;       ESI = 00404007
```

INDIRECT Operands = Pointer = Dolaylı Adresleme

Değişkenlerin adreslerini (pointer) tutar. Adresler genelde ESI VE EDI register'larında tutulur. Başlarındaki E'de, 4 byte olduklarını gösterir. EAX, EBX vs AX, BX gibi

[degisken] ifadesi ile referans kırılır yani adresin içindeki değer alınabilir.

```
.data
        val1 BYTE 10h, 20h, 30h
.code
        mov esi, OFFSET val1;       esi, val1'in adresini depolar
        mov al, [esi]
// [esi] ifadesi ile esi'nin depoladığı adresi kırıp içindeki değeri aldıktan sonra al register'ine atar.

        inc esi
// esi registerini bir artırdık. ESI registeri şuanda val1 dizisinin başlangıç adresini tutuyor. Yani biz
// ESI'yi bir artırırsak bir sonraki bir byte artacak. O da val1 değişkeninin ikinci elamanına yani
// 20h'nin adresine denk geliyor. Ama dikkat şuan 20h'ye değil 20h'nin adresine geldik.

        mov al, [esi];              al = 20h
// ESI, şuanda 20h'nin adresinde. O adresi dereferans yaparak adresin içindeki elemana eriştik.

        inc esi
// ESI, en son 20h'nin adresini tutuyordu. Eğer bir daha artırısak bu sefer 30h'nin adresini tutacak.

        mov al, [esi];              al = 30h
// ESI, şuanda 30h'nin adresinde. O adresi dereferans yaparak adresin içindeki elemana eriştik.
```

INDIRECT OPERANDS & PTR

myCount WORD 0

// WORD tipinde bir değişken oluşturup içine 0 attık.

mov esi, OFFSET myCount

// myCount değişkenimizin adresini ESI registerine attık.

inc [esi];

// Bu satır hata verecektir. Tuttuğu adresin tipini bilmiyor ki artırma yapsın.

inc WORD PTR [esi];

// Hata vermez. PRT kullanarak ESI'nın tuttuğu adresin içindeki değeri, WORD tipine dönüştürüyoruz.

add [esi], 20;

// Bu satır hata verecektir. ESI'nin gösterdiği adresin değerine 20 eklemek istiyoruz ama biz ESI'nin değerini tipini bilmiyoruz. Diyebilirsiniz ki: "Biz zaten yukarıda PTR ile tipini vermiştik" PRT operatörü sadece bulunduğu komutta işlenmek üzere tipini değiştirir. Kalıcı bir değişiklik değildir.

add WORD PTR [esi], 20;

// Hata vermez. ESI'nin değerini tipini PRT sayesinde WORD olarak belirliyor ve belirledikten sonra esi'nin değerine 20 ekliyor.

Indexed Operands

C'de ve çoğu dildeki gibi dizinin elemanlarına index yardımıyla ulaşmaktır. İki çeşit gösterimi vardır:

[label + register] ya da label [register]

.data

arrayW WORD 1000h, 2000h, 3000h

.code

mov esi, 0; // esi register'inde şuan 0 var.

mov ax, [arrayW + esi]; // ax = 1000h

mov ax, arrayW[esi];

// "arrayW dizisinin ESI registerindeki değeri hangi değerse o dizinin o değerinci
// elemanını seç diyoruz. ESI'miz 0 olduğu için ilk elemanı seçer.

add esi, TYPE arrayW

// arrayW'nin tipi WORD (2 byte). O zaman TYPE 2 değerini dönecek ve bunu ESI'ye atayacak.

add ax, [arrayW + esi]

// arrayW'nin adresine ESI'yi yani 2 değerini ekler. O da dizini ikinci elemanının
// adresine denk gelir. O adresin değerini yani ikinci elemanını verir.

RAIDR

- Bir tabloda main memory'ye kaydedilen verilerin kaydedilme tarihi ve süreleri tutulur. Tüm memory'yi refresh etmek yerine bu tabloya göre refresh edilir. Bu teknolojiyede RAIDR denir.
- **Mimari Problemler:**
  1. Enerji kısıtları
  2. Tasarımın karmaşıklığı
  3. Daha önceden tasarlanmış olan mimarilere yeni teknolojiler ekleme

JMP Instruction

C'deki goto komutu ile aynı işlevi görür.

myLabel:

// burada kodlar var işte

// …

jmp myLabel

Şöyle çalışır; kod ***'jmp myLabel'e*** geldiğinde myLabel etiketinin olduğu satıra geri döner ve orayı tekrar çalıştırır. Eğer bir şart olmazsa orası sonsuza kadar çalışır.

LOOP Instruction

Loop (döngü) yazmaya yarar. Syntax'ı JMP'ninkine benzemektedir. Hatta aynısıdır.

mov ecx, 5;

myLabel:

// burada kodlar var işte

// …

loop myLabel

// ECX bir register türü ve döngülerdeki iterasyon sayısını tutar.

// loop myLabel'a gelindiğinde ECX registeri bir azaltılır. ECX 0 olduğunda döngü biter.

// Biz başta ecx'i 5 olarak tanımladık. ECX 0 olduğunda da döngümüz sonlandı.

NOT: ECX'in ilk değeri 0 olamaz. Yani "mov ecx, 0" kodu hata verecektir.

İç içe Döngü

- Eğer iç içe döngü yapılmak isteniyorsa **ECX** registeri önce bir değişkende saklanacak. ECX aynı anda iki loop'u çalıştıramaz. Bu yüzden dış döngünün kaç kere döneceğini kaydettik.
- Daha sonra içteki döngü kaç kez dönülmesi isteniyorsa ECX'e o sayı atanacak. Daha sonra dış döngü dönmek için diğer label'a (adrese) gidecek. Gitmeden hemen önce count, ECX'e atanacak. Yani loop'a girdiği anda ECX bir azaltılmış olacak.
- Label'e girdikten sonra ECX'i count'a atayacağız. Çünkü artık ecx, iç döngü için kullanılacak.
- Şimdi ise iç döngünün iterayon sayısı ECX'e atanacak ki iç döngü o kadar dönsün.

```
.data
   count DWORD ?          dış döngü (Loop L1) 100 kez,
.code                     iç döngü (Loop L2) ise 20 kez dönecek.
    mov ecx,100         ; dış döngü sayısını ayarla
L1:
      mov count,ecx     ; dış döngü sayısını kaydet
      mov ecx,20        ; iç döngü sayısını ayarla
L2:
       loop L2          ; iç döngüyü tekrarlayın
       mov ecx,count    ; dış döngü sayısını geri yükle
       loop L1          ; dış döngüyü tekrarlayın
```

- Register'ler büyürse ısı artar. Bu yüzden registerler daha fazla büyütülmüyor. Register'lar, lojik kapılar olduğu için hızlıdır. RAM'de lojik kapılar yoktur ve bu yüzden yavaştır.
- Erişim açısından bakınca ***main memory (RAM), cache'den daha hızlı*** ama genel olarak bakınca ***cache daha hızlı.***

- ***DRAM ve SRAM Hızlarını Karşılaştırma***
  Static RAM'de (SRAM) transistör kullanıldığı için refresh yapılmıyor. ***Dinamic RAM'de (DRAM)*** kapasitör kullanıldığı için refresh yapılır. Refresh olmadığı için ***SRAM hızlıdır*** ancak ***erişimde DRAM daha hızlıdır*** çünkü SRAM verileri satır satır arar.
  SRAM genellikle cache için, DRAM genellikle main memory (RAM) içindir.
  PC'lerimizde DRAM kullanırız

- **Cache Miss ve Cache Hit**
  Cache Miss, aranan verinin cache bellekte bulunmamasıdır.
  Cache Miss, Von Neumann Mimari'sinde daha yüksektir çünkü veri yolu ve komut yolu aynı olduğu için veriler hızlıca cache belleğe aktarılamamaktadır bu da cache miss oranının yüksek olmasına sebep oluyor.
  Cache Hit, aranan verinin cache bellekte bulunmasıdır.
  Cache Hit, Harvard Mimari'sinde yüksektir çünkü veri yolu ve komut yolu farklı olduğundan veri aktarımı daha hızlıdır.

- **Bilgisayar Mimarisinde Kaç Model Bulunmaktadır?**
  Control Flow Model (Von Neumann Model)
  1. Program counter olduğu için sıralı şekilde çalışmaktadır.
  2. Komutlar ve veriler için tek bir yol vardır.
  3. Bir cycle'da (zamanda) sadece bir instruction çalışır.
  4. Control Flow daha ucuzdur bu yüzden günümüz bilgisayarlarında control flow kullanılır.
  5. Debug işlemi daha kolaydır.

  Data Flow Model
  1. Program counter olmadığı için rastgele şekilde çalışır.
  2. Bir komutun çalıştırılmasının önceki komuta bağlı olmamasıdır. Bir komut önceki komuta bağlıysa bağlı olan komutla paralel olarak çalıştırılır.
  3. Potansiyel oalrak tüm komutlar aynı zamanda çalıştırılır.
  4. Daha hızlıdır. Program yazmak daha zordur.

Bizim bilgisayarlarımız ***Control Flow*** kullanır çünkü ***debug*** yapabilmek için.

Paralel programlama (paralellism), komutların aynı anda çalışmasıdır. Paralellism için birden fazla işlemci gerekir.
Programı ***thread***'lere de bölmekle paralel programlama yapmış olmayız, komutlar hala sıralı çalışır. Sadece aynı zamanda çalışıyor gibi gözükür.

mov ve inc/dec komutları hiçbir bayrak bitin etkilemez bu yüzden mov ve /inc/dec komutlarıyla ilgili herhangi bir işlem varsa oranın bütün bayrak bitleri 0 yapılır.

OFFSET'te sorduğu yere kadar olan adresi alırız.

NEG, WORD olarak tutulur yani 2 BYTE bu yüzden ax ile işlem yapıyormuşuz gibi düşünebiliriz çünkü ax'de 2 BYTE.

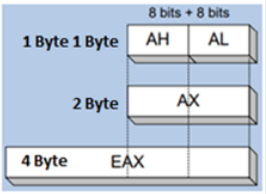
NEG'in amacı 2'ye tümlemektir yani önce tersini alıp sonra 1 ile toplamak.

```
.data
    bVar BYTE      12h, 34h, 56h, 78h
    wVar WORD   1234h, 5678h, 4321h, 8765h
    dVar DWORD 12345678h, 87654321h
.code
```

Başlangıç Adresi: FFFFF000

KOMUT	KAYDEDİCİ	Zero Flag	Sign Flag	Carry Flag	Overflow Flag
mov ax, [wVar+7]	ax = 78 87	0	0	0	0
mov eax, SIZEOF wVar	eax = 0000 0008	0	0	0	0
mov ebx, OFFSET [wVar+2]	ebx = FFFFF006	0	0	0	0
mov ax, [wVar+2]	ax = 5678h	0	0	0	0
Neg wVar	EDCC	0	1	0	0
mov bx, WORD PTR dVar	bx = 5678h	0	0	0	0
sub ax, bx	220F	0	1	1	0
mov al, 0FFh	FFh	0	0	0	0
add al, 1	00h	1	1	0	1
mov bl, 127	7Fh	0	0	0	0
add bl, 1	80h	0	0	0	0
mov al, [bVar+2]	56h	0	0	0	0
mov ebx, [dVar]	12 34 56 78	0	0	0	0
sub al, bl	00h	1	0	0	0

	RAM	
bVar	12	0000 = 0
	34	0001 = 1
	56	0010 = 2
	78	0011 = 3
wVar	34	0100 = 4
	12	0101 = 5
	78	0110 = 6
	56	0111 = 7
	21	1000 = 8
	43	1001 = 9
	65	1010 = 10
	87	1011 = 11
dVar	78	1100 = 12
	56	1101 = 13
	34	1110 = 14
	12	1111 = 15
	21	10000 = 16
	43	10001 = 17
	65	10010 = 18
	87	10011 = 19
	xx	10100 = 20
	xx	10101 = 21

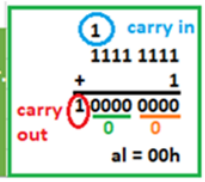


add bl, 1
127 = 0111 1111
+ 1
128 = 1000 0000 = 80h

127'yi hexadecimal'e çevirdik.

wVar = 3412h
0011 0100 0001 0010
1100 1011 1110 1101
\+ 1
1100 1100 1110 1101
12=C 12=C 14=E 13=D
En baştaki bit 1 old. İçin SF = 1 oldu.

5678h - 5678h = 0
Sonucumuz 0 olduğu için ZF=0 oldu.
SF= 0 oldu.
al, 1 BYTE olduğu için sağdaki 1 Byte'ı yani FF'i aldık.
al'de FF yani 1111 1111 var (bir önceki komut) 1 ile toplayınca 1 0000 0000 elde ederiz.
Toplama yaptığımızda oluşan elde Carry In, ortada kalan 0000 = 0, en sağda kalan 0000 = 0 olur.
al 1 BYTE old. İçin ortadaki ve sağkdai 0'ları aldık 00h elde ettik.
add için OF=1 ise CF=0 olur yani tam tersi olur eğer OF=0 ise CF=1 olur.
sub için "OF = CarryOut XOR CarrIn". Örneğin OF=1 XOR 1 = 0 olur.



Mov eax, SIZEOF wVar --> WORD'ün boyutu 2 BYTE ve 4 adet eleman var 2*4=8

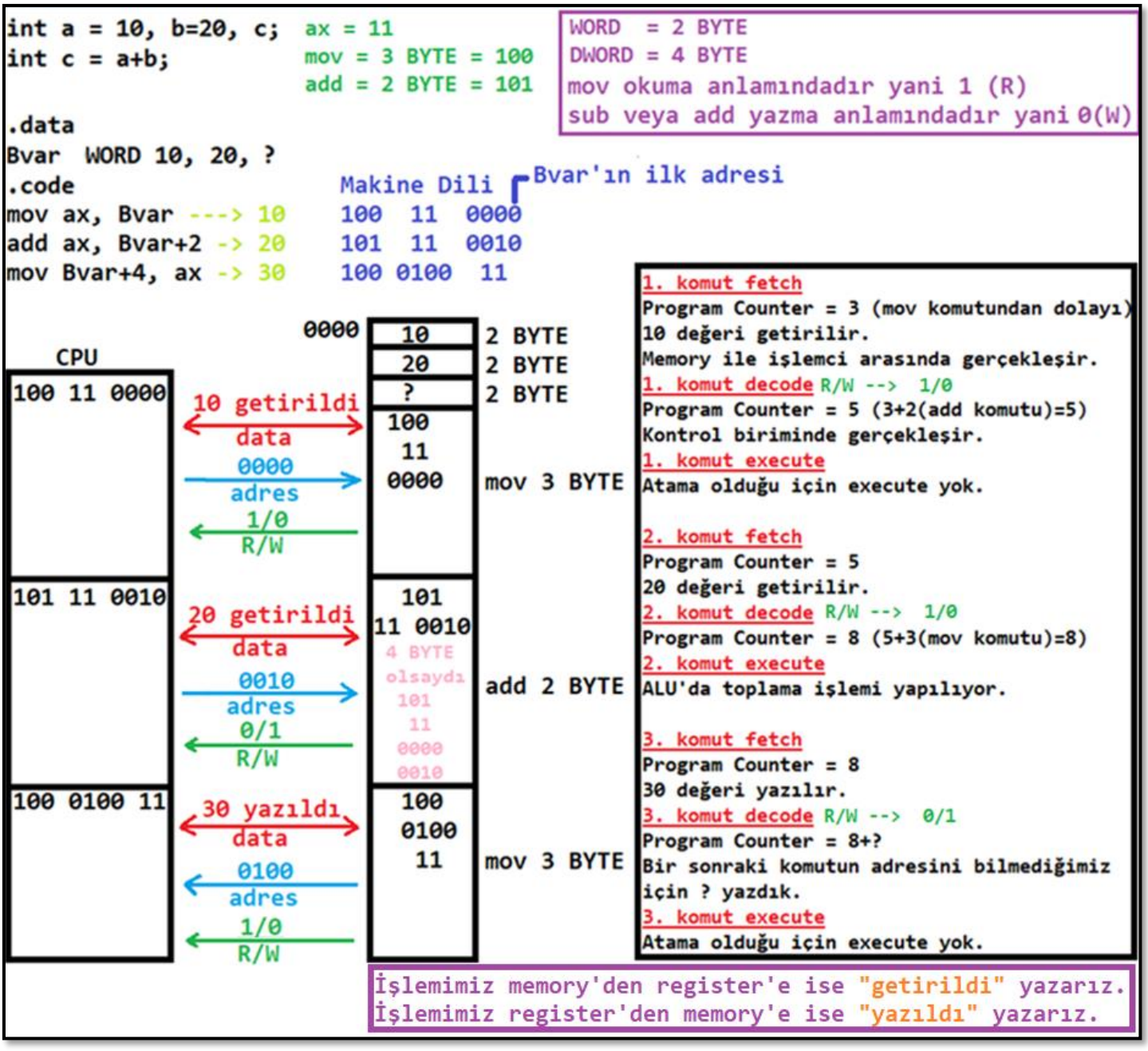
OFFSET'lerin tipi DWORD'dür (4 byte). Yani DWORD tipindeki bir değişken OFFSET tutabilir.

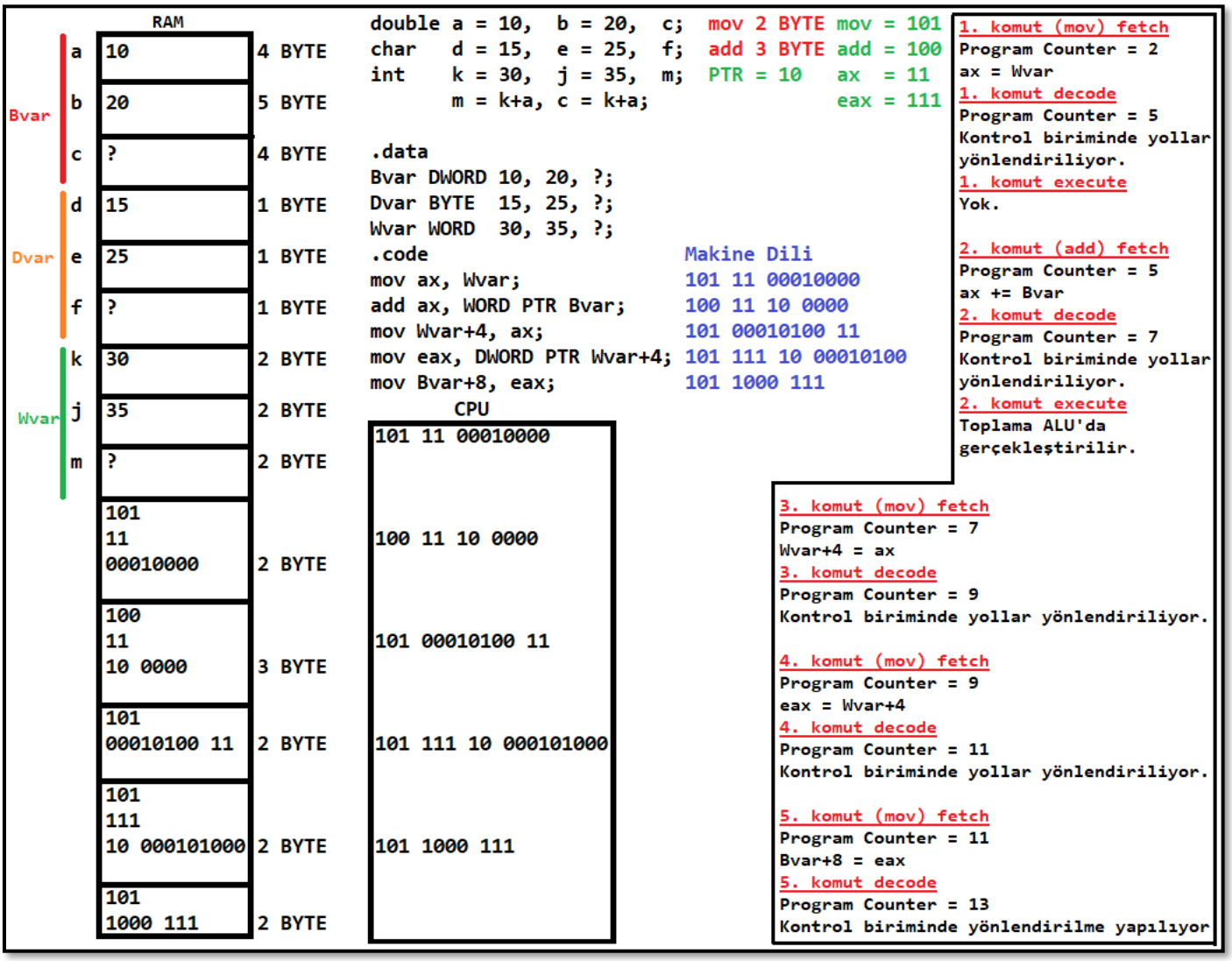
RISC (ARM)	CISC (X86)
Komut işleme stili bakımından, en fazla 3 operand içerir. 3 operand aldığı için bir önceki değeri kaybetmeyiz. ***operand:* add ax, val1** ***buradaki ax ve val1 operand, add opicode'dur*** ***add ax, bx, cx*** ***cx + bx = ax*** olur dolayısıyla kaydedicilerin bir önceki değeri kaybolmaz. **(İŞLEMCİ)**	Komut işleme stili bakımından, en fazla 2 operand içerir. 2 operand aldığı için işlemlerde bir değeri kaybederiz. ***add ax, bx* —> *ax = ax+bx*** Burada artık ax'in önceki değerine ulaşamayız ve bu da değer kaybına neden olur. **(İŞLEMCİ)**
Veri türü azdır. **(İŞLEMCİ)**	Veri türü fazladır. **(İŞLEMCİ)**
Register sayısı fazla ve devresi basit olduğu için RISC daha hızlıdır. Ayrıca update daha kolay çünkü devresi basit. **(İŞLEMCİ)**	Register sayısı az ve devresi karmaşık olduğu için CISC daha yavaştır. Ayrıca update zor çünkü devresi karmaşık. **(İŞLEMCİ)**
Semantic gap fazla yani fazla zaman harcar. **(İŞLEMCİ)**	Semantic gap az yani az zaman harcar. **(İŞLEMCİ)**
Memory organizasyonunda Big Endian kullanılır. Yani MSB bitinden başlayarak yazar. **(MEMORY)**	Memory organizasyonunda Little Endian kullanılır. Yani LSB bitinden başlayarak yazar. **(MEMORY)**
Komut boyutlarına göre **fixed length modeli** kullanılır yani her komut eşit boyuttadır. **(MEMORY)**	Komut boyutlarına göre **variable length modeli** kullanılır yani her komut kendi uzunluğu kadardır. **(MEMORY)**
Uniform yapıya sahiptir yani komutların konumları sabittir. **(MEMORY)**	***Non-Uniform*** yapıdadır yani komutların konumları sabit değildir. **(MEMORY)**
Align yapıdadır yani hizalıdır bu yüzden NOP komutunu kullanmaya gerek yok. **(MEMORY)**	***Unalign*** yapıdadır yani hizasızdır. İhtiyaç duyulursa NOP komutu kullanılarak align yapılabilir. **(MEMORY)**
Operandlar register olmak zorunda, buna ***load/store mimarisi*** denir. **(İŞLEMCİ)** ***Load/Store*** *problemi vardır.* Memory'deki bir değerin register'e kopyalanmasına ***LOAD,*** register'deki bir değerin hafızaya kopyalanmasına ise ***STORE*** denir. ***mov ax, var1*** ***mov bx, var2*** ***add cx, ax, bx*** RISC'de memory'ye erişemediği için böyle yaparız. (Load/Store problemi)	Operandlar hafızadaki bir yeri gösterebilir buna ***hafıza mimarisi*** denir. **(İŞLEMCİ)** Bir komutta sadece bir defa access olur buna **memory to memory** denir. CISC'de bir komutta sadece bir defa access olur. İki defa yazılırsa bu yanlış olur. ***add var1, var2*** Bu komut gerçekleşmesi için iki defa memorye erişim sağlayıp getirmesi gerekir fakat bunu gerçekleştiremez bu probleme ***memory to memory*** ismi verilmiştir. Çözüm: ***mov ax, var1*** ***add ax, var2***
Memory'ye gidip o adresin değerini değiştirmeye ***erişim (access)*** denir. RISC mimarisinde erişim olmadığı için kaydedici sayısı fazladır.	

Komutlar önbelleğe kaydedilmez çünkü önbellekte sadece değişkenler tutulur.
Fetch değişkenlerde daha çok yapılır.

ASM Çevirme Adımları

1- Değişken taraması yapılır, türleri ile birlikte (string, int) getirilir.
2- Değişkenler sınıflarına göre kümelenir.
3- Memory organizasyonu yapıldıktan sonra makine diline dönüştürülür.
4- Komut çalıştırma adımları (fetch-decode-execute) ve en az bir komut bir defa fetch yapılması lazım.
5- Program Counter her seferinde bir sonrakinin adresini içerecek şekilde otomatik olarak arttırılır.





Assembly yürütülebilir bir program oluşturmaz. Yürütülebilir program için aşağıdakiler gereklidir:

Program Çalıştırma Adımları

5- Assembly diline dönüştürülür. (Kod Binary formatına (makine diline) dönüştürülür.)
6- Linking gerçekleştirilir. (Kütüphaneye gidip komutları getirir.)
7- Loading gerçekleştirilir. (Program, RAMe taşınır.)
8- Execute gerçekleştirilir.

Uygulamaya her tıkladığımzda 'loading' olur çünkü uygulamanın komutları belleğe yüklenir. Execution aşaması da gerçekleşip uygulama çalıştırılır.

SINAVDA PROGRAM ÇALIŞTIRMA ADIMLARI SORULURSA EXTRA OLARAK KOMUT ÇALIŞTIRMA ADIMLARINIDA YAZMAYI UNUTMA!

Komut Çalıştırma Adımları

4- Komut getirilir. **FETCH (İşlemci ile hafıza arasında ya da 2 tane memory arasında gerçekleşir.) (RAM'den kaydedicilere gider.)**
5- Kod çözülerek anlamlı hale getirilir. **DECODE (Kontrol birimi tarafından gerçekleştirilir.)**
6- Komut neyse o çalıştırılır. **EXECUTE (Execute, ALU veya çıkış portlarında gerçekleştirilir.)**

Her komut ya ***single-cycle*** ya da ***multi-cycle*** olarak çalıştırılır.
Cycle: Bir işlemcinin hızını belirleyen osilatörün iki darbesi arasındaki süredir.

1- **Single Cycle**
   Her komut tek cycle'da gerçekleşir.
   Fetch-Decode-Execute tek komutta gerçekleştiği için frekans aralığı daha geniştir.
   En yüksek komut süresine göre clock ayarlanır.
   Single Cycle'da komutun işlendiğini gösteren flag tutulur. Komutun işlenmesi bittiyse flag 1 olur ve sonraki komuta geçilir.
2- **Multi-Cycle**
   Her komut birden fazla cycle'da gerçekleşir.
   Fetch-Decode-Execute farklı cycle'larda gerçekleşir.
   Komutların boyutları farklı olduğu için cevap süreleri de farklı olur.
   Multi Cycle'da birden çok flag vardır. Çünkü komutlar farklı zamanlarda işlenebilir.

Bir programın var 100 tane clock'dan oluştuğunu varsayalım. Singe Cycle'a göre programın çalışma süresi en büyük komutun süresine göre ayarlanır. Örneğin bu program için 2 saniye olsun o halde 100x2=200 saniyede gerçekleşir.
Multi Cycle'da 25'i 3 saniyede, 25'i 2 saniyede, 25'i 4 saniyede kalan 25'i de 5 saniyede gerçekleşsin.
25x2=50 saniye
25x3=75 saniye
25x4=100 saniye
25x5=125 saniye toplamda (50+75+100+125)/4=87,5

Bellekler
Location
Internal (Primary) location: Main memory, cache
External (Secondary) Location: Hard disk, cd...
Procecssor Location: Register

Access Methods
Sequential: Bilgi, diskte sıralı aranır.
Direct: Harddisk'de kullanılır.
Random: Main memory
Associative (Karşılaştırıcı): Cache
Access Method yani erişim olarak en hızlısı ***random'dur.***

Physical Characteristics
Volatile (Geçici) / Nonvolatile (Kalıcı)
Erasable (Silinebilir) / Nonerasable

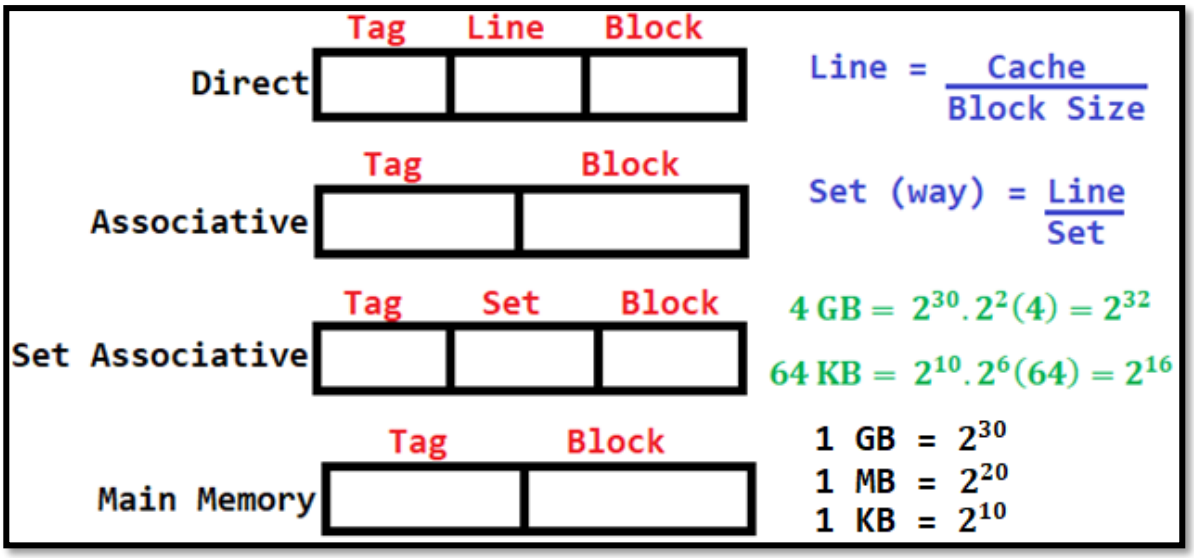
NOT: Hafıza birimi olarak en hızlısı cache.
NOT: Register'larda kullanılan erişim yöntemileri paralel bağlıdır bu yüzden erişim yöntemi yoktur.
NOT: Process'in içinde hangi hafıza birimi var: Register

Mapping

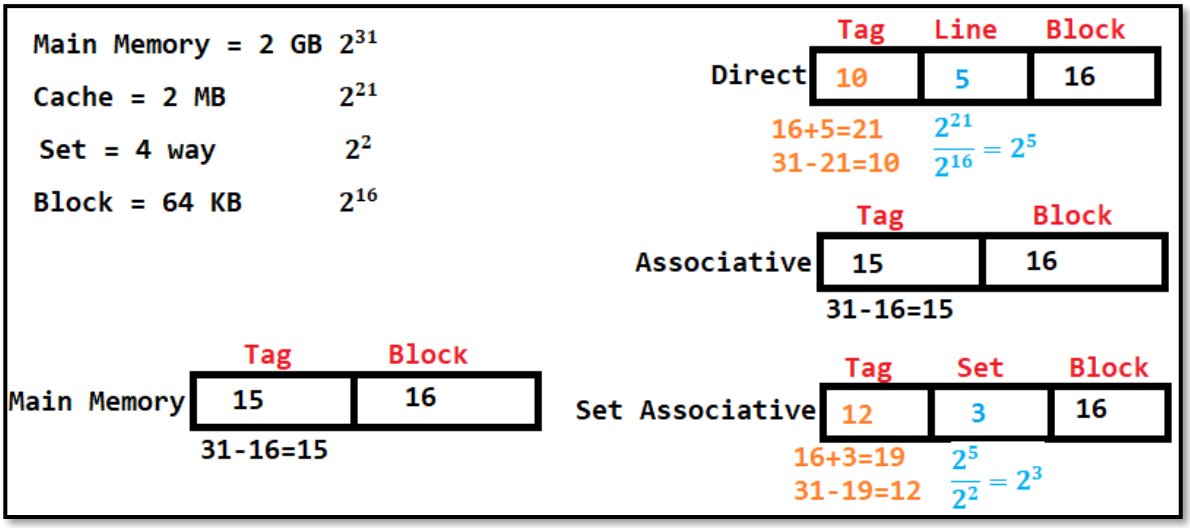
Mapping işlemide en genel tanımıyla CPU'dan gelen adresin cache boyutuna benzetilmesi işlemidir.
Main memory'deki "Blok"'un karşılığı cache'deki "Line" 'dır.

NOT: ***Set Associative*** daha karmaşıktır çünkü kendi içinde associative yapıdadır ve hafızadan alınan bloğun kümede hangi satıra yazılacağı replacement algoritması ile belirlenir.

Direct Mapping oluşturmak daha ucuz çünkü bir block bir satır yazılabilir.



ÖRNEK-1



ÖRNEK-2

Main Memory = 128 kbyte, cache = 16 kbyte, block size = 256 byte, line size = 256 byte, set = 2 way

1. Adım

Main Memory'nin adreslenebilir bit sayısını bulalım.

$2^7 . 2^{10} = 2^{17}$, adreslenebilir bit sayısı 17 bit.

2. Adım

Cache'in adreslenebilir bit sayısını bulalım.

$2^4 . 2^{10} = 2^{14}$, cache'in adreslenebilir bit sayısı 14 bit.

3. Adım

Block Size: 256 Byte = $2^8 . 2^1 = 2^8$

Line Size: 256 Byte = $2^8 . 2^1 = 2^8$

4. Adım

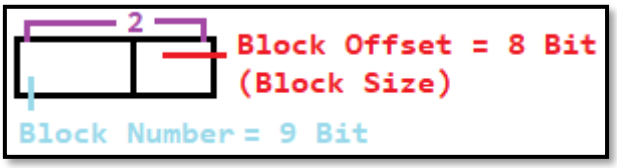
Set = 2^1

5. Adım

$$\text{Line} = \frac{\text{cache}}{\text{block size}} = \frac{2^{14}}{2^8} = 2^6$$

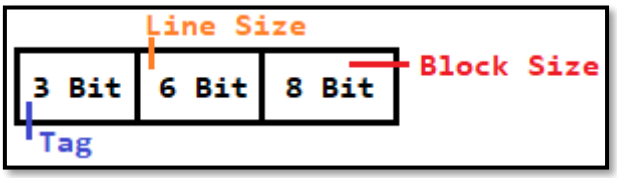
$$\text{Set} = \frac{\text{line}}{\text{set}} = \frac{2^6}{2^1} = 2^5$$

Physical Adress

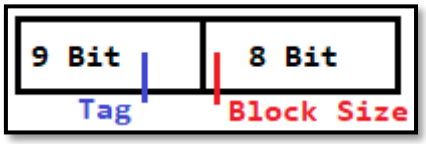


Cache Modlar

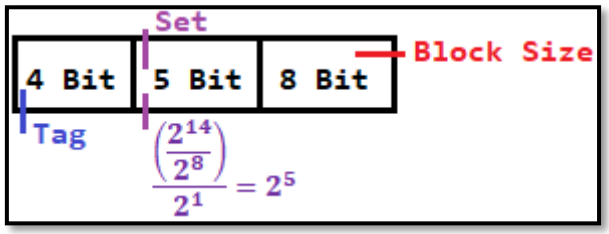
6. **Direct**



7. **Associative**



8. **Set Associative**



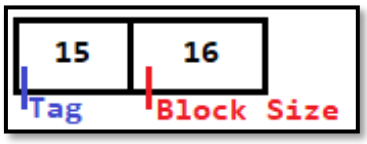
ÖRNEK-3

Bellek boyutu = 2 GB, ön bellek boyutu = 2 mb, block size = 64 kb, set = 8 way;

Main Memory (RAM) = 2 GB = 2^{31} Set = 8 = 2^3

Cache = 2 mb = 2^{21} Block Size = 64 kb = 2^{16} = Line Size

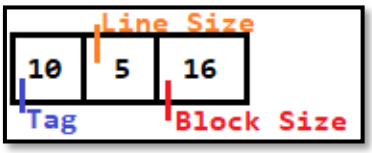
Main Memory



$$\frac{2^{31}}{2^{16}} = 2^{15}$$

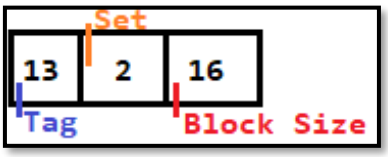
Cache

1. **Direct**



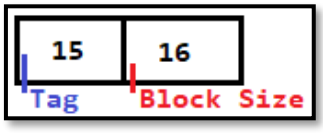
$$\frac{2^{21}}{2^{16}} = 2^5$$

2. **Set Associative**



$$\frac{2^5}{2^3} = 2^2$$

3. **Associative**



RAM Tasarımı

8 GB RAM'i 2 decoder ile tasarlayalım.

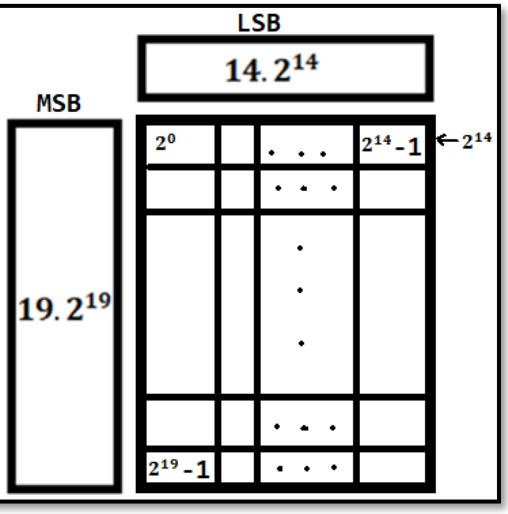
RAM = 8 GB = $2^{30}.2^3 = 2^{33}$

Block Size = 16 KB = $2^{10}.2^4 = 2^{14}$

$$\frac{2^{33}}{2^{14}} = 2^{19}$$

Yani 19 sütun, 14 satır.

Block Size'ı hoca vermezse 2^8 almalıyız.



Komutun İşlemcide İşlenme Süreci

Kodun program counter'ının tutulduğu register'dan ***adres yoluna*** bu adres yazılır ve RAM'e gider. RAM gelen adrese göre uygun komutu ***veri yolundan*** gönderir. Gelen komut ***kontrol biriminde*** decode edilir. Eğer komutun içinde bir değişken varsa bunların adresi farklı register'lere yazılır. Register'daki adres, adres yolundan memory'e giderek memory'den doğru değişken verisini veri yolu üzerinden getirir. Gelen değişkenler register'larda tutulur. Ancak komutun operandlarından biri immediate ise veriyi çekmek için doğal olarak memory'e gitmez. Immediate, işlemcide işlenmek üzere bekler.

Hafızadan gelip register'e atılan değişkenler ve (varsa) immediate, Multiplexer'e gider. Multiplexer, her iki operandın register olup olmadığına bakar. Eğer operandlardan birisi immediate ise ALU ona göre ayaklarından birisi ona göre ayarlanır. Çünkü ALU'ya sadece iki değer girebilir. Kısacası gelen değerler ALU'ya girer ve hesaplama yapılır. Daha sonra hesaplanan değer ALU'dan çıkp bir register'e atılır. Ve komut işlenmesi bitti. Ama program counter'in şuanda artırılması lazım diğer komutun adresini tutmak için. ALU, program counter'ın tutulduğu register'i alır ve az önce işlediği komutun boyutu kadar değerini artırır ve tekrar gidip aynı registere bunu yazar. Böylece program counter artmış olur.

Veri yolu ve adres yolu tek bir yol üzerinden gider. İkiside veri yolunu kullanır.
Bu yüzden bir komutun değişkenleri getirilirken diğer komut memory'den çekilemez çünkü veri yolu doldudur. Bu da Von Neumann mimarisinin dezavantajıdır. Bu bekleme süresine ***wait state*** denir.

İşlemcide ya register tipi ya da immediate tipi vardır. ALU'nun girişlerinden biri Register olmak zorundadır. Eğer gelen komutta bir register veya bir immediate varsa ALU'nun girişlerinden biri immediate olur. Memory diye bir ihtimal yok çünkü hesaplama için işlemciye gelen memory'ler de register'lere atılır.
Gelen komutun ikisi de register mi yoksa biri immediate mi diye ***multiplexer (seçici)*** bakar. Ona göre ALU'ya gönderir.

Control Flow'da unconditional jump ve conditional jump vardır.
JMP'nin operandı ya adrestir ya da registerdir. Yani registerin içindeki adrestir.