

# MIPS ve Temel Kavramlar

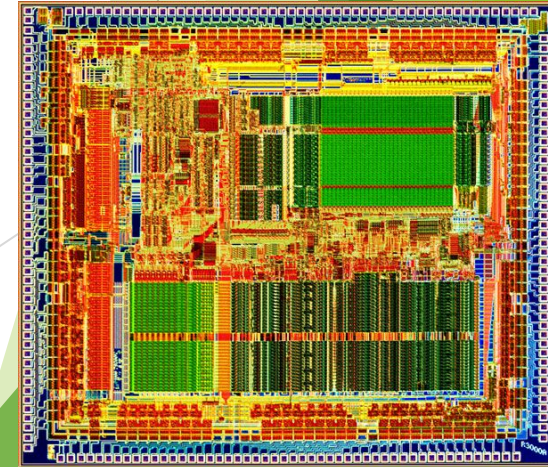
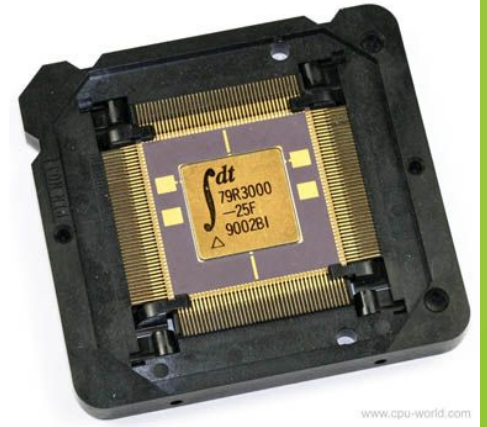
Ders2

# Neler öğreneceksiniz

- ▶ MIPS nedir
- ▶ Compiler, Interpreter ve Assembler kavramları
- ▶ CPU/MCU 'da «Word» kavramı
- ▶ Hafıza yapısına göre temel bilgisayar mimari tipleri
- ▶ Big ve Little endian kavramları
- ▶ Temel MIPS registerları
- ▶ MIPS assembly komutlarına giriş

# MIPS (Microprocessor without Interlocked Pipelined Stages)

- MIPS Technology firması tarafından geliştirilmiş bir RISC Instruction Set Architecture (ISA) 'dır. (Not: Million instructions per second (MIPS) ile karıştırmayın)
- 32 ve 64 bitlik mimari versiyonları bulunmaktadır
- Temel çıkış noktası 5 safhalı (5-stages) pipeline ve cache yapısının tek bir çipset içinde yüksek performans sağlayabileceği fikridir (Standford Üniversitesi 1981)
- MIPS I - V (32 bit), MIPS64 gibi versiyonları bulunmaktadır
- İlk MIPS mimarili işlemciler: R2000, R3000 (1-2mikron teknoloji, 8-40MHz, FPU desteği, 5-Stages), R4000(ilk 64 bit versiyon), R8000, R10000 ..
- Pek çok firma MIPS mimarili çipsetler kullanmaktadır



# Neden MIPS

- ▶ Basit ve yalın bir tasarımı vardır bu nedenle üniversitelerde genellikle MIPS mimarisi öğretilir
- ▶ Rahat anlaşılır bir Assembly diline sahiptir (*MIPS-I ISA temelli baz alınacaktır*)
- ▶ Eğitim için simülatör desteği (*R2000(ilk versiyon) ve R3000 işlemcilerini simüle edebilen SPIM (yeni versiyonu QTSPIM) derslerimizde kullanılacak*)
- ▶ Bazı örnek MIPS mimari implementasyonları
  - ▶ PlayStation2
  - ▶ Tesla S (otomobil otomasyon)
  - ▶ PIC32MZ MCU
  - ▶ Realtek's RTL8812BR 802.11ac Wi-Fi Router/Repeater
  - ▶ NEC VR4305



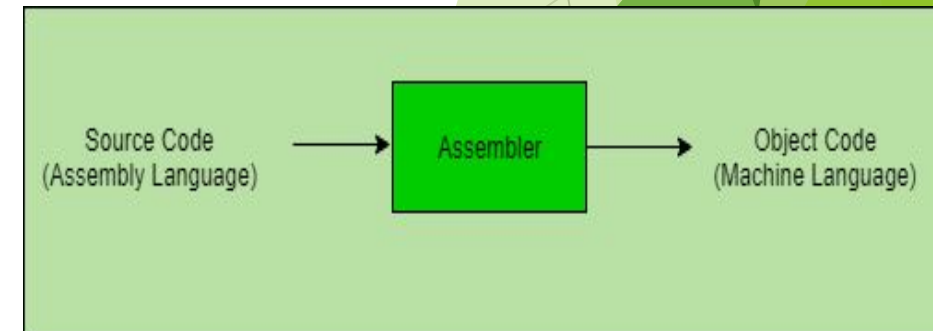
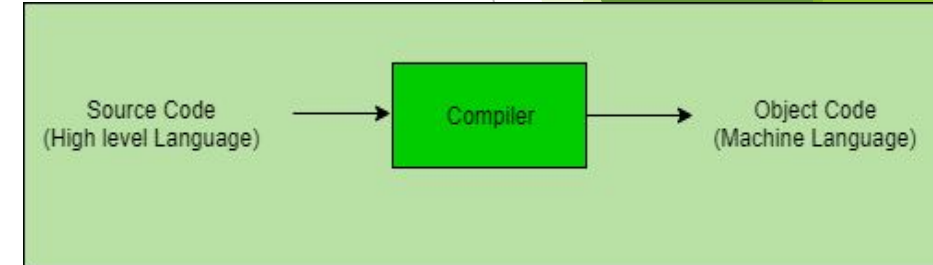
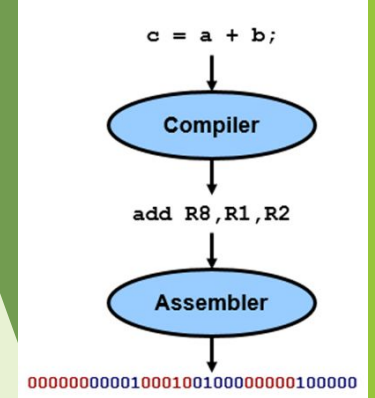
# MIPS Assembly Dilini tanımadan önce bazı kavramları hatırlamakta fayda var

- ▶ **Assembly Dili** : Assembly dili makine dili seviyesine en yakın programlama dilidir (Low-level programming language)
  - ▶ C/C++, Java, Python gibi üst seviyeli programlama dillerinin sağladığı pek çok avantaja Assembly programlamada sahip olamazsınız. Donanım bilginiz, programlama yeteneğiniz ve siz başbaşasınız
  - ▶ Kısaca üst seviye diller ile makine dili arasında kalan bir dil gibi değerlendirilebilir
  - ▶ Her mimarinin (ISA mimarisinin) kendine ait assembly dili olabilir ve her biri farklı komut setlerine sahip olabilir. Bu nedenle hangi mimari için assembly dili kullandığınız temel kriterdir.
  - ▶ Assembly programlama ile işlemci register 'ları, hafıza yönetimi, I/O işlemleri artık sizin ellerinizdedir
  - ▶ Eğer isterseniz bir çok compiler yazılımı, C/C++ gibi compiler bazlı dillerde yazmış olduğunuz üst seviye kodlara ait (onlara karşılık gelen) assembly kodlarını size üretebilir (hem de optimize edilmiş şeklini)

```
void sumarray(int a[], int b[], int c[]) {  
    int i;  
    for(i = 0; i < 100; i = i + 1)  
        c[i] = a[i] + b[i];  
}  
  
Loop:    addi    $t0,$a0,400    # beyond end of a[]  
        beq     $a0,$t0,Exit  
        lw      $t1, 0($a0)    # $t1=a[i]  
        lw      $t2, 0($a1)    # $t2=b[i]  
        add     $t1,$t1,$t2    # $t1=a[i] + b[i]  
        sw      $t1, 0($a2)    # c[i]=a[i] + b[i]  
        addi    $a0,$a0,4      # $a0++  
        addi    $a1,$a1,4      # $a1++  
        addi    $a2,$a2,4      # $a2++  
        j       Loop  
Exit:    jr      $ra
```

# Peki ya Assembler? O bir Compiler mı?

- ▶ **Assembler** bir yazılımdır ve yazılmış assembly kodlarını ilgili donanıma ait makine kodlarına dönüştürür (Object Code)
- ▶ Assembler ile compiler arasındaki farklara bakalım
  - ▶ Compiler da bir yazılımdır ancak o sizin yazmış olduğunuz üst seviye dil kodlarını (source codes, örnek C++) makine diline dönüştürür (Eğer isterseniz önce assembly diline sonra makine diline de dönüştürebilir)
  - ▶ Assembler ise assembly kodlarını makine diline çevirir
  - ▶ Compiler çevrim için daha fazla işlem safhası kullanır
- ▶ Derslerimizde biz MIPS assembly dili ile daha çok meşgul olacağız. Aslında bunu iki farklı şekilde deneyeceğiz
  - ▶ Üst seviyede yazılmış kodlarımızın bir derleyici tarafından ara assembly koduna dönüştürülmüş versiyonu gibi
  - ▶ Veya doğrudan biz MIPS assembly dili ile program yazıyormuşuz gibi

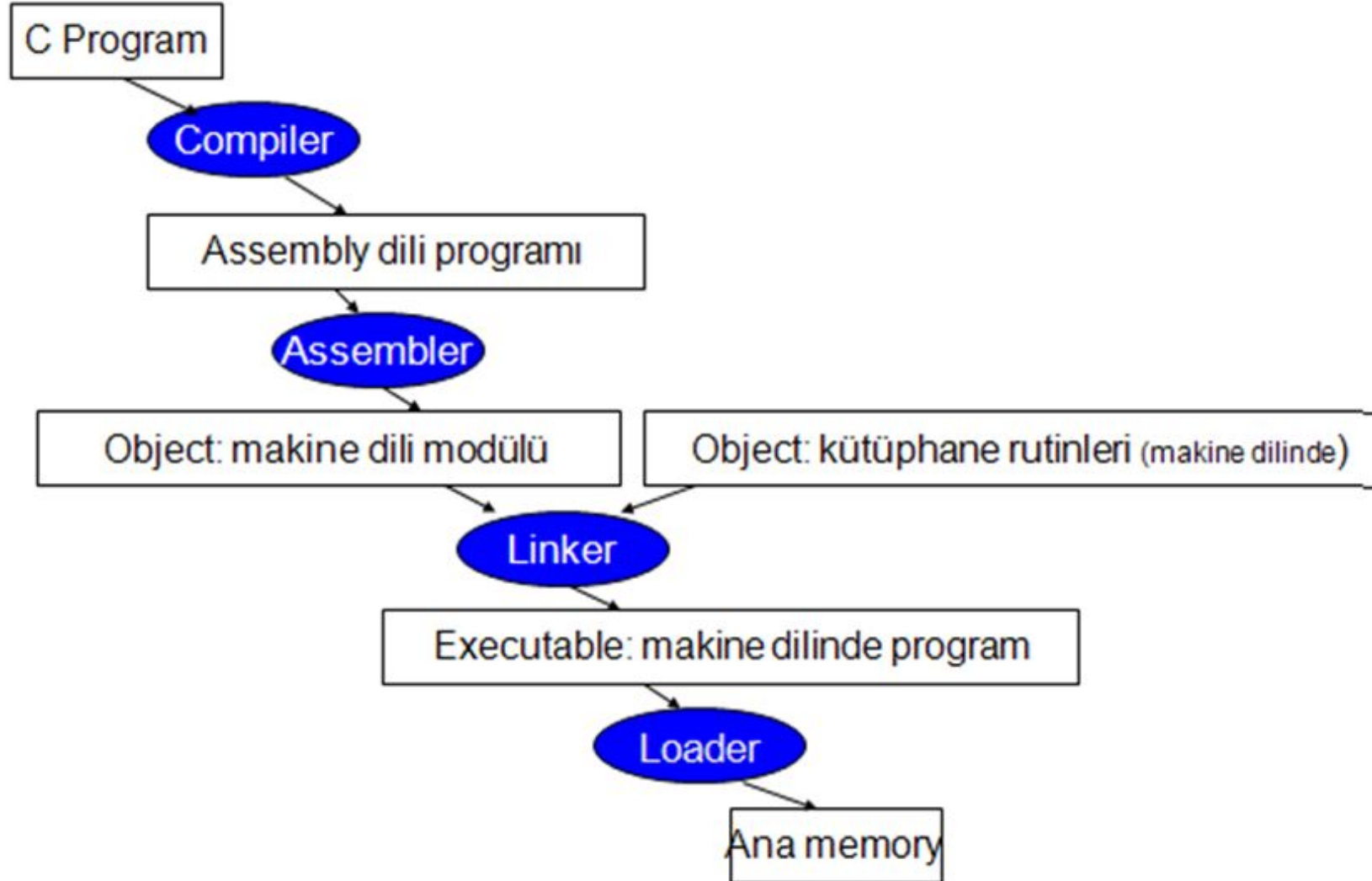


# Ha bir de Interpreter (Yorumlayıcı) vardı?

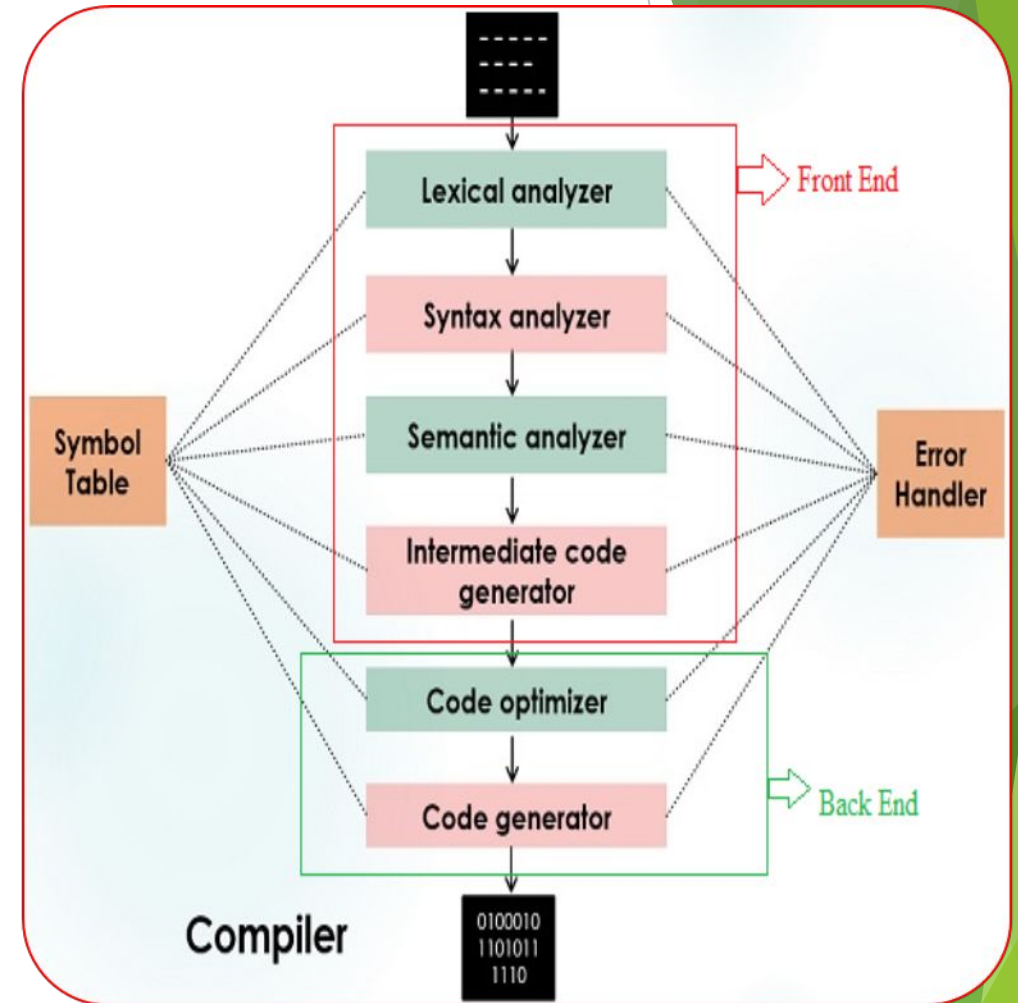
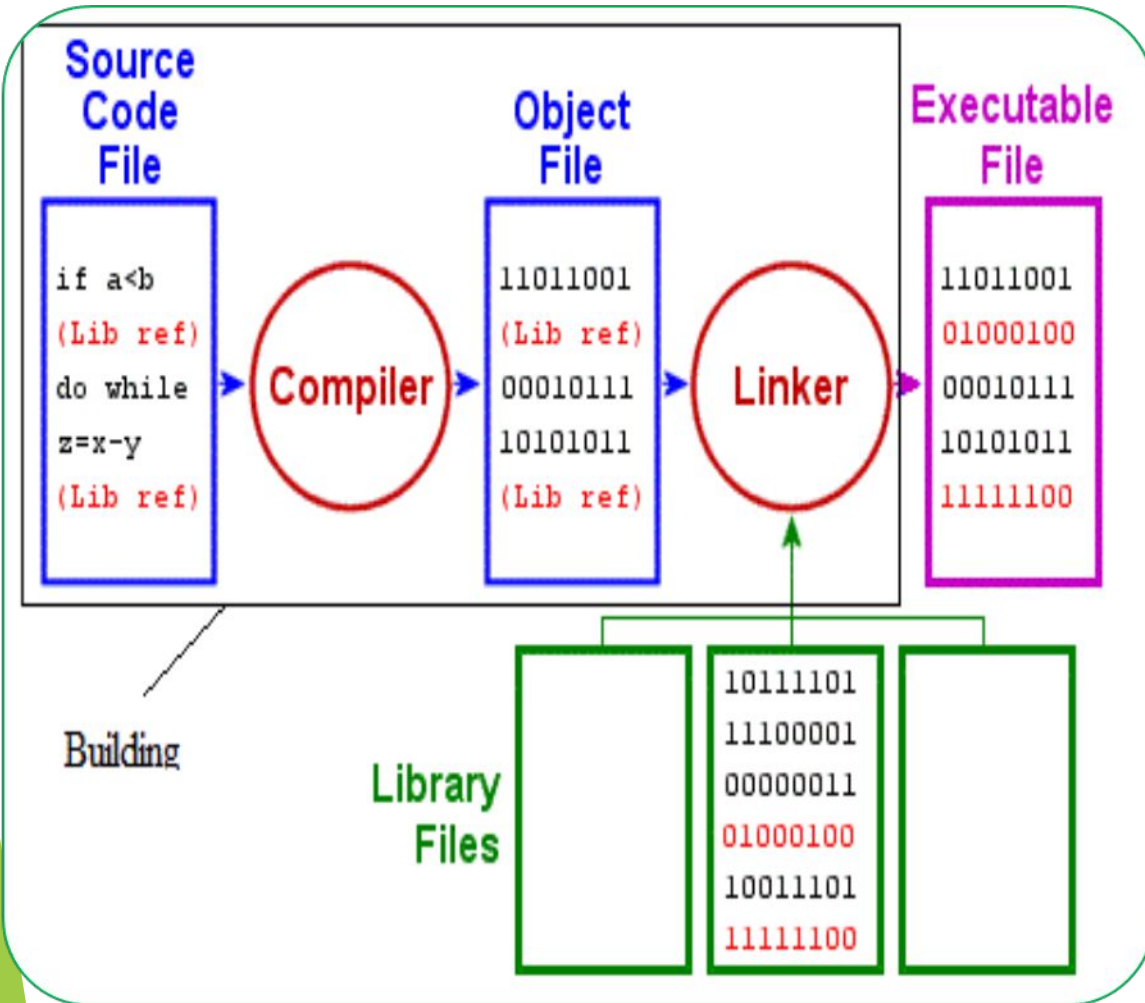
- ▶ Aslında bizim Interpreter kavramı ile bu derste işimiz yok. Biz compiler bazlı kod yazılımı ile ilgileneceğiz fakat hatırlatma olması açısından kısaca interpreter kavramından da bahsedelim;
  - ▶ Programlama dillerinin bir kısmı derleme temellidir. Örneğin C/C++ gibi. Bu tarz dillerde yazılan kodlar compiler yazılımları ile object code 'lara dönüştürülürler ve bu object code 'lar bir dosyada (örnek, .out dosyaları) tutulur. Program çalıştırılacağı zaman derlenmiş bu object dosya hafızaya yüklenir. Bir program bir kez doğru derlendiği zaman artık hatasız çalışması garanti edilir
  - ▶ Programlama dillerinin bir kısmı da kodları baştan aşağı derleyip bir dosyaya yüklemektense yazılan kodları sırasıyla çalıştırır. Yani bir object code üretilmez. Bir hata olduğunda çalışma kırılır. MATLAB, Python, PHP bu tarz dillerdir.



# Bizim esas aldığımız compiler temelli süreç adımları





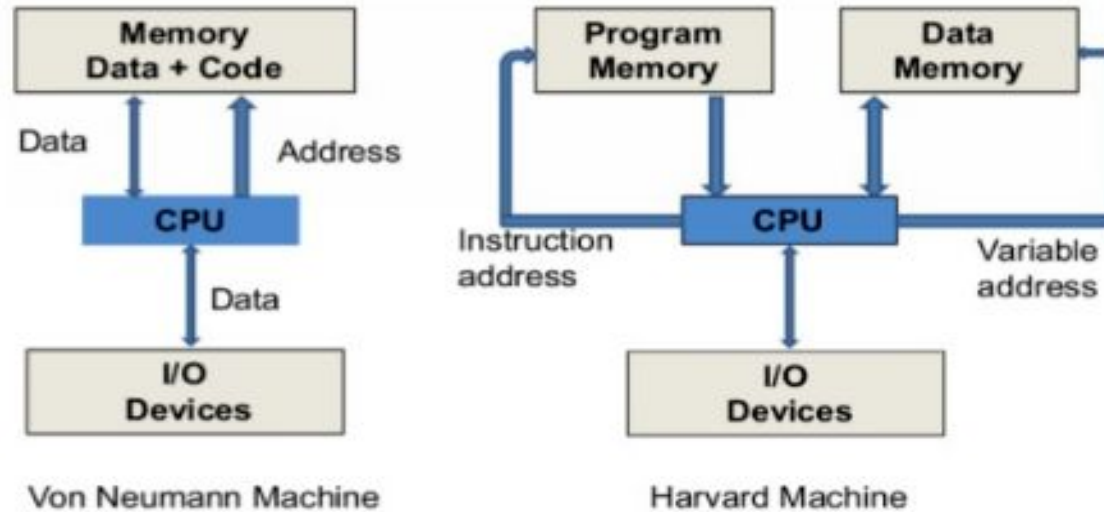


# İşlemcilerde bir önemli kavram : Word

- ▶ Bilgisayar mimarisinde işlemciler için kullanılan en önemli karakteristik parametrelerden biri «WORD» dur.
- ▶ Word, bir işlemci tarafından (bir ISA komutu yürütümünde) tek seferde işlenebilecek veri birimini ifade eder. Sabit boyutludur ve genellikle **Word length**, **Word size** veya **Word width** gibi ifadelerle tanımlanır. Bu sabit boyut bir seferde register 'lara yüklenebilecek en fazla veri boyutunu veya veri yolları üzerinde taşınabilecek maksimum veri genişliğini gösterir.
- ▶ Günümüzde işlemciler 8, 16, 24 (örn. PIC16), 32 (örn MIPS32) ve 64 bitlik Word 'ler kullanırlar. Ancak bazı özel tasarımlı işlemciler (özel DSP ler gibi) farklı boyutlu Word uzunlukları kullanabilir
- ▶ **MIPS-I 'de Word uzunluğu «32 bit» dir!! Bu rakam bu derste pek çok kez karşınıza çıkacaktır.**

# Bir önemli bilgi daha: Hafıza yapısına göre mimariler

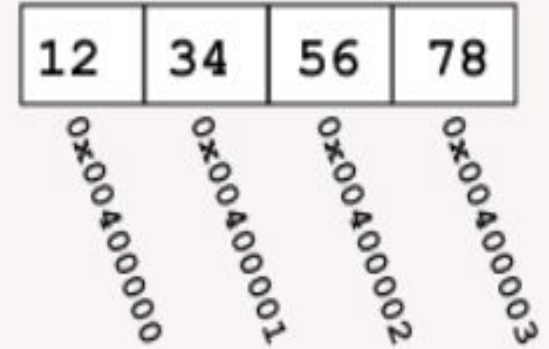
- Bilgisayar mimarileri hakkında konuşurken bir önemli kavramlardan biri de o mimarinin hafızayı kod ve veriler için nasıl kullandığıdır
- Günümüzde literatürde bunun için genellikle iki ana mimari sınıf vardır (Her ne kadar saf bir mimari yapısı bulmak zor olsa da ara sınıflandırmalara girmeyeceğiz)
  - Harvard Mimari (örn., PIC16 serisi)
  - Von-Neumann Mimari (örn, MIPS-I)



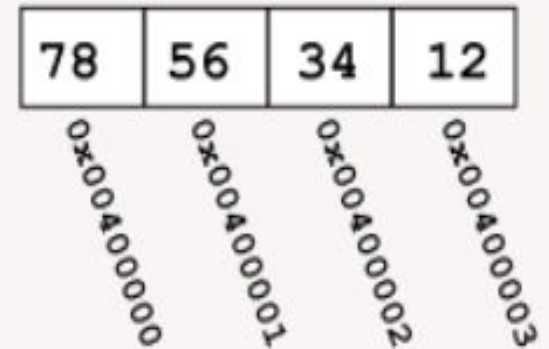
# «Big Endian» ve «Little Endian» kavramlarını bir kez daha hatırlayalım

- ▶ Verilerin hafızaya hangi şekilde yazıldığı veya hafızadan hangi adres sırsına göre okunacağı üzerine genel bir standart firmalar arasında belirlenmemiştir
- ▶ Hatırlanacağı üzere «byte bazlı» hafıza yapılarında veriler byte byte yazılır veya okunur. Her bir byte doğal olarak bir adrese(etikete) sahiptir ki verilerin nerede bulunduğu bilinsi. Eğer 32 bitlik bir veriniz varsa byte bazlı hafızada bu veri için 4 Byte 'lık alan kullanılacaktır. Hafızadaki bu dört byte 'ın her birinin bir adresi vardır ve bunlar sırayla artar. Örnek olarak 0x12345678 verisini hafızadaki 0x0040000 ile 0x0040001 nolu adreslere yazarken;
  - ▶ Bazı firmalar (MIPS, MIPS-I mimarisi gibi), verinin en önemli byte 'ını hafızada en düşük numaralı adresten başlayarak yazar. Bu tip yazmaya/okumaya «BIG ENDIAN» form denir
  - ▶ Bazı firmalar (Intel, x86 mimarisi gibi) ise değersiz byte 'ını en düşük adresten başlayarak yazmaya başlar. Bu forma ise «LITTLE ENDIAN» denir

## Big Endian



## Little Endian



# Bir hatırlatma daha

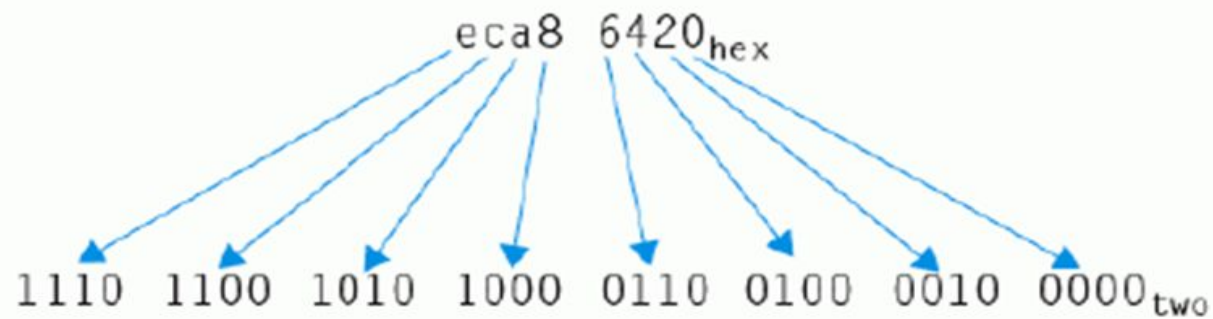
## Binary-to-Hexadecimal and Back

Convert the following hexadecimal and binary numbers into the other base:

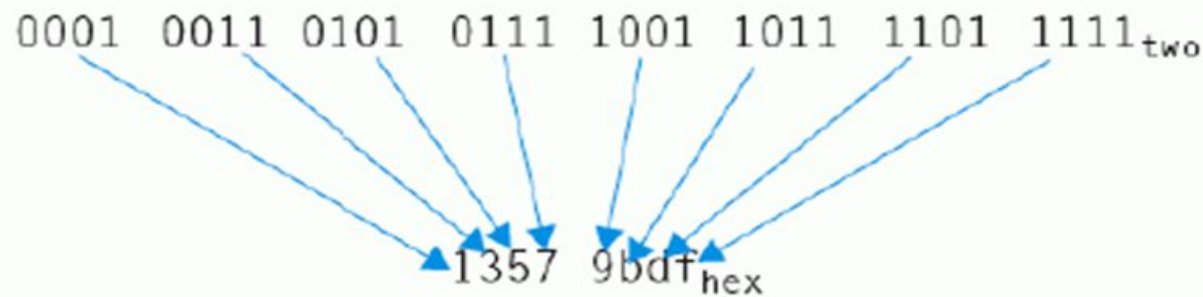
eca8 6420<sub>hex</sub>

0001 0011 0101 0111 1001 1011 1101 1111<sub>two</sub>

Just a table lookup one way:

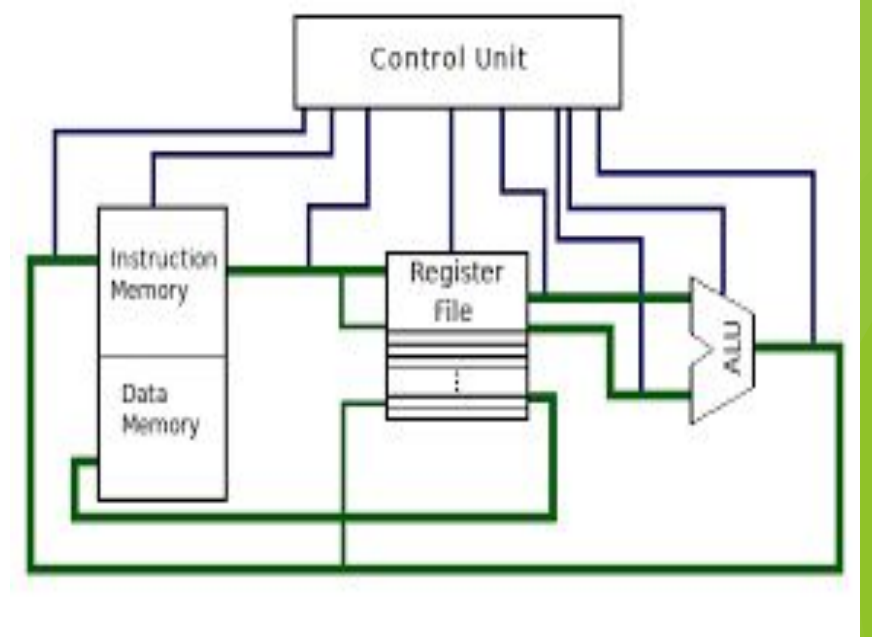


And then the other direction too:



# İşlemcilerde en önemli dahili bileşenlerden biri : REGISTER

- ▶ Register 'lar en hızlı erişilebilen hafıza konumlarıdır. CPU/MCU içerisinde yer alırlar
- ▶ Her ne kadar bazı bilgisayar organizasyonlarında farklı örnekleri olsa da, temel olarak bir register bir Word 'luk bilgi tutar. Bu bilgi, bir komut, adres veya her hangi bir data olabilir
- ▶ Erişimleri genellikle tek cycle alır. Öte yandan RAM için bu süre birkaç cycle 'dır
- ▶ Register 'lar FF 'dan oluşurlar
- ▶ CPU/MCU içindeki register gruplarına «REGISTER FILE» adı verilir





# MIPS-I Register 'ları

- ▶ MIPS 'de 32 adet «**genel amaçlı**» register vardır. Doğal olarak bu register 'lara bir etiket/ID gereklidir. Bu etiketler fiziksel olarak sayısal olmak zorundadır. Bu da demek oluyor ki 0 ile 31 arasında numaralar verilir. Bu numaralar içinde sizin 5 bitlik ( $2^5 = 32$ ) bir bilgi kullanmanız gerekecektir (bunu sonra tekrar hatırlayacağız)
- ▶ MIPS-I mimaride bu register 'ların bir kısmı «**özel amaçlı**» registerlardır (pc, sp gibi). Bunları yeri gelince daha detaylı göreceğiz.
- ▶ Ayrıca içinde her zaman sabit değer tutan «**constant register**» lar da MIPS-I bulunmaktadır. Bunlar özellikle bazı işlemleri daha kolaylaştırmaya yararlar. Bunlardaki veriler sadece okunabilir, değiştirilemez.
- ▶ Son tip register 'lar ise daha çok üretici ve donanım hakkında bilgi saklayan «**model-spesifik**» registerlardır. (read-only)
- ▶ MIPS assembly 'de en önemli kodlama manevraları register 'lar ve hafıza arasında gerçekleşecektir. Bu nedenle register 'lar ve hafıza yapısı dikkatle irdelenmelidir
- ▶ **ANA KURAL: MIPS-I, HAFIZA ÜZERİNDE DİREK İŞLEM YAPAMAZ. HAFİZADAKİ VERİLERİ ÖNCE KENDİ REGISTER 'LARINA GETİRİR. BURADA İŞLEM YAPAR EĞER SONUÇ TEKRAR HAFİZAYA GÖNDERİLECEKSE ONU AYRI BİR İŞLEM OLARAK HAFİZAYA YAZAR !!!**



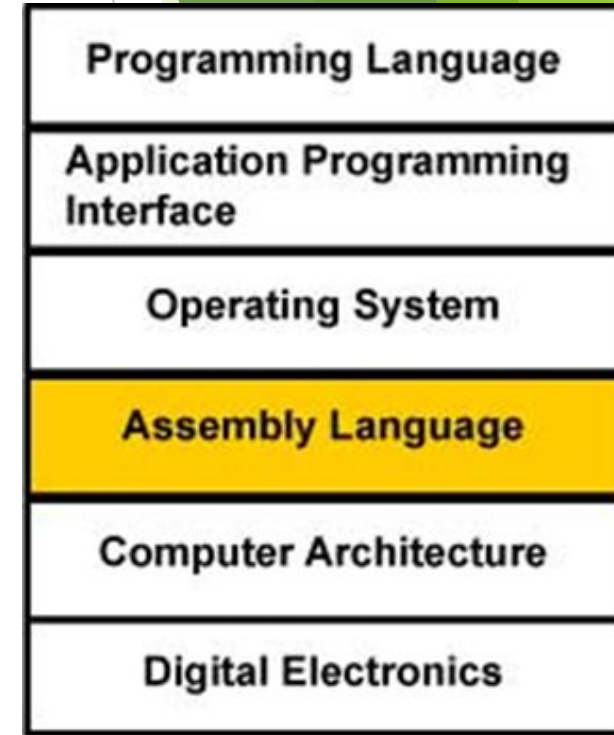
# Register Standart İsimleri (MIPS-32)

Name	Register Number	Usage	Preserved on call
\$zero	0	the constant value 0	n.a.
\$at	1	reserved for the assembler	n.a.
\$v0-\$v1	2-3	value for results and expressions	no
\$a0-\$a3	4-7	arguments (procedures/functions)	yes
\$t0-\$t7	8-15	temporaries	no
\$s0-\$s7	16-23	saved	yes
\$t8-\$t9	24-25	more temporaries	no
\$k0-\$k1	26-27	reserved for the operating system	n.a.
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

MIPS assembly  
'de register 'lar  
«\$» ile  
gösterilirler

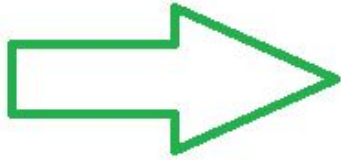
# MIPS-I Assembly ve ISA

- ▶ MIPS mimarisine geçişte en önemli abstraction «Assembly» katmanıdır
- ▶ Her işlemci mimarisi gibi MIPS-I 'de kendi assembly programlama diline sahiptir
- ▶ Bu demek oluyor ki kendi donanım mimarisine uygun komutları ve yazım kuralları vardır
- ▶ Assembly kodları yazarken register 'ları, hafıza lokasyonlarını, hangi aritmetik veya mantıksal işlemleri yapacağımızı söyleriz
- ▶ Yüksek seviyeli dillerde işimizi kolaylaştıran for, if gibi operasyonlar yoktur. Bunların assembly karşılıkları vardır. Bu da demek oluyor ki daha fazla kod, daha fazla dikkat ve doğal olarak daha fazla eziyet 😊



# MIPS Assembly

- ▶ Bu derste Assembly komutlarını sırası geldikçe öğreneceğiz
- ▶ İlk olarak «add» «addi» ve «sub» temel komutları ile başlayalım
- ▶ Tuzak: MIPS-I assembly 'de subi komutu YOKTUR. Çünkü addi ile negatif bir sabit sayı toplanabilir. (SORU: peki nasıl? Cevaplayınız..)

Komutlar			Anlamı
<b>add</b>	<b>\$s1, \$s2, \$s3</b>		
<b>addi</b>	<b>\$s1, \$s2, 100</b>		
<b>sub</b>	<b>\$s1, \$s2, \$s3</b>		
			<b><math>\\$s1 = \\$s2 + \\$s3</math></b>
			<b><math>\\$s1 = \\$s2 + 100</math></b>
			<b><math>\\$s1 = \\$s2 - \\$s3</math></b>

## Bir örnek

Diyelim ki C dilinde şöyle bir kod yazılmış olsun:

$$f = (g + h) - (i + j)$$

Ve burada **f**, **g**, **h**, **i**, **j** değişkenleri (bundan sonra **Operand** da diyebiliriz) sırasıyla MIPS 'in **\$s0**, **\$s1**, **\$s2**, **\$s3**, **\$s4** register 'larında tutulduğu kabul edilsin

Cevap:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1  # f gets $t0 - $t1, which is (g + h)-(i + j)
```

# addi ile Operand adreslerini belirleme tekniği

Çoğu zaman hafızada belirli adreslere ulaşmak gerekli olacaktır. Bu nedenle hafızanın hangi adresine ulaşacağınızı önceden belirli bir register 'da tanımlamak gereklidir. «addi» komutu burada önemli iş görecektir. Aşağıdaki örnekte \$zero sabit register 'ının işlevi nasıl kolaylaştırdığına dikkat ediniz



- Bir komut sabit bir giriş değerine ihtiyaç duyar.
- Bir immediate komut, girişlerden birini sabit bir değer olarak kullanır (Bir register operand'ı yerine).

```
addi $s0, $zero, 1000 # programın başlangıç adresinin 1000 olduğu, bunun $s0
                        # 'a' kayıtlı edilmesini, $zero ise devamlı 0'a eşit olan bir
                        # register olduğunun belirtilmesi.

addi $s1, $s0, 0      # Bu a operand'ının adresidir.
addi $s2, $s0, 4      # Bu b operand'ının adresidir.
addi $s3, $s0, 8      # Bu c operand'ının adresidir.
addi $s4, $s0, 12     # Bu dizi operandı d[0] in adresidir.
```



# Temel Hafıza Erişim Komutları

## «lw» komutu (Load Word)

- Amacı bir Word uzunluğundaki veriyi hafızadan istenilen register 'a getirmektir.
- Temel formu : lw hedefRegister, Offset(AdresTutanRegister)
- Örnekler

**lw \$t0, 0(\$s3)**

**lw \$t0, 32(\$s1)      #32 = 8x4**

# lw ile hafızada bir dizi elemanına ulaşma yöntemi

- ▶ Hatırlayacağınız üzere dizi elemanları hafızada arka arkaya tutulur. Bu demek oluyor ki eğer siz bir dizinin hafızadaki başlangıç adresini ve her bir elemanın boyutunu bilerseniz, dizi üzerinde istenilen elemana ulaşabilirsiniz. Örnek olarak integer değerler tutan bir `A[]` dizisi ele alalım ve bunun 6. elemanına (5 nolu index) ulaşmak isteyelim. Yine farz edelim ki `A[]` 'nın ilk elemanı (0 nolu index) `0x00200000` adresinde olsun. Integer değerlerin 32 bit olduğunu yani 4Byte uzunlukta olduğunu biliyoruz. Bu durumda 2 nolu index elemanın adresi
  - ▶  $0x00200000 + 5 * 4 = 0x00200014$  olacaktır (Hex olarak konuşuyoruz dikkat edin)
- ▶ Bu işlemi «lw» komutuyla iki şekilde kullanabiliriz.
  - ▶ Birincisi `A[0]` adresini istediğimiz bir register 'a atmak ve ofset (kayma) değerine 20 vermek
    - ▶ `addi $t0,$zero,0x00200014`
    - ▶ `lw $t1,20($t0)`
  - ▶ İkincisi istenilen adresi önceden hesaplayıp bir register 'a atıp 0 ofset değeri ile hafızadaki veriyi getirmek
    - ▶ `addi $t0,$zero,0x00200000`
    - ▶ `addi $t0,$t0,20`
    - ▶ `lw $t1,0($t1)`

#addi komutu ile ister decimal ister Hex sayı yazılabilir



## «sw» komutu (Store Word)

- ▶ Tahmin edeceğiniz üzere bir Word 'luk veriyi MIPS register 'ından alıp hafızada belirtilen adrese kaydeder
- ▶ Tıpkı «lw» de olduğu gibi bu hafıza adresi önceden bir register 'da tutulmalıdır
- ▶ Aşağıda iki farklı kullanım örneği gösterilmiştir

The diagram illustrates the components of the MIPS 'sw' instruction using two examples. Red lines connect labels to specific parts of the instructions:

- Gönderilecek veri** (Data to be sent) points to the register `$t0` in both instructions.
- Offset** points to the constant value (0 or 48) in both instructions.
- Hafıza adres bilgisi** (Memory address information) points to the register `$s1` in both instructions.

**Example 1:** `sw $t0, 0($s1)`

**Example 2:** `sw $t0, 48($s1)`

# Genel bir örnek

C kodu:  **$A[12] = h + A[8];$**

A[0] adresi \$s3 'de ve h değeri ise \$s2 'de olsun. Bu işleme ait MIPS assembly kodu

```
lw  $t0, 32($s3)  
add $t0, $s2, $t0  
sw  $t0, 48($s3)
```

## Örnek



Aşağıdaki C kodunu Assembler koduna dönüştür. Bir önceki hafıza yapısını ve register transferlerini kullan.

C kodu: **d[3] = d[2] + a;**

### Çözüm.

```
addi $s0, $zero, 1000
addi $s1, $s0, 0      # Bu a değişkeninin adresidir.
addi $s2, $s0, 4      # Bu b değişkeninin adresidir.
addi $s3, $s0, 8      # Bu c değişkeninin adresidir.
addi $s4, $s0, 12     # Bu d[0] değişkeninin adresidir.

lw   $t0, 8($s4)      # d[2], nin içeriği $t0 registerine getirilir.
lw   $t1, 0($s1)      # a'nın içeriği $t1 registerine getirilir.
add  $t0, $t0, $t1     # Toplam $t0'nin içindedir.
sw   $t0, 12($s4)     # $t0'ın içeriği d[3] nolu hafızadaki adrese yazılır.
```

# Ödev

Örnek:

- a) Aşağıdaki işlemin sonucunu elde ediniz.  
Data hafızasının başlangıç adresi 0'dır.  
A dizisinin başlangıç adresi 8'dir.
- a) Bu işlemlerin yüksek seviyeli dil kodu nedir.

```
addi    $s0, $zero, 0
addi    $s1, $s0, 8
lw      $s2, 0($s0)
lw      $s3, 20($s1)
add     $s4, $s2, $s3
sw      $s4, 28($s1)
```

```
addi    $s0, $zero, 4
addi    $s1, $s0, 8
lw      $s2, 0($s0)
lw      $s3, 16($s1)
add     $s4, $s2, $s3
sw      $s4, 20($s1)
```

**Not: hafıza yerleşimi Little Endian formundadır.**

Data Memory

19	0	1	0	1	0	1	0	1	39	1	1	1	0	1	0	1	1
18	1	1	0	0	1	0	1	0	38	1	0	0	0	1	1	0	0
17	0	0	1	1	1	1	0	1	37	0	0	0	1	1	0	1	0
16	1	0	0	1	1	1	0	1	36	1	0	0	0	0	1	1	0
15	0	0	1	1	0	0	0	1	35	1	0	1	1	0	1	0	1
14	0	1	0	0	1	1	0	1	34	1	1	0	0	1	0	1	1
13	1	1	1	1	0	1	1	0	33	0	1	0	0	1	1	0	0
12	0	1	0	0	1	1	0	1	32	0	0	0	1	1	1	0	0
11	1	0	1	1	1	0	1	0	31	0	1	1	0	1	0	0	0
10	0	1	0	1	0	1	0	1	30	0	0	1	1	0	0	1	1
9	0	1	0	1	1	0	1	1	29	0	0	1	1	0	0	0	1
8	1	1	1	0	0	1	0	0	28	0	1	1	1	0	1	0	1
7	1	0	1	0	1	1	0	1	27	1	1	0	0	1	0	0	0
6	0	1	0	1	1	1	0	1	26	1	0	0	1	1	0	1	0
5	0	0	1	0	1	0	1	0	25	0	1	1	0	0	0	0	1
4	0	1	0	1	0	1	0	1	24	0	1	0	0	1	1	0	1
3	1	0	0	0	0	0	1	1	23	1	1	1	1	0	1	0	1
2	0	1	1	1	1	1	1	0	22	0	1	0	1	1	0	1	0
1	1	0	0	0	0	0	1	1	21	1	0	0	0	1	1	0	1
0	0	1	0	0	0	1	1	1	20	0	1	1	0	1	0	1	0

# Son bir özet

## Özet tablo

**MIPS operands**

Name	Example	Comments
32 registers	$\$s0, \$s1, \dots$ $\$t0, \$t1, \dots$	Registerler DATA için hızlı yerleşimlidir. MIPS'de, aritmetiksel işlemler için Data registerlerde olmalıdır.
$2^{30}$ memory words	Memory(0), Memory(4), ..., Memory(4294967292)	MIPS'te hafızadaki DATA'ya sadece data transfer komutlarıyla erişilir. MIPS, Bayt adreslerini kullanır. Bu yüzden 32 bitlik bir kelimenin adresi ardışık 4 bayt adresidir. Hafıza, dizi ve "spilled register" şeklinde veri (data) yapılarını tutar.

**MIPS assembly language**

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	$\$s1 = \$s2 + 100$	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Data from memory to register
	store word	sw \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Data from register to memory