2.Bölüm
Komutlar: Bilgisayar dili
Instructions: Language of Computers

"Ben, Tanrıyla İspanyolca, kadınlarla İtalyanca, erkeklerle Fransızca, karımla Almanca konuşurum."

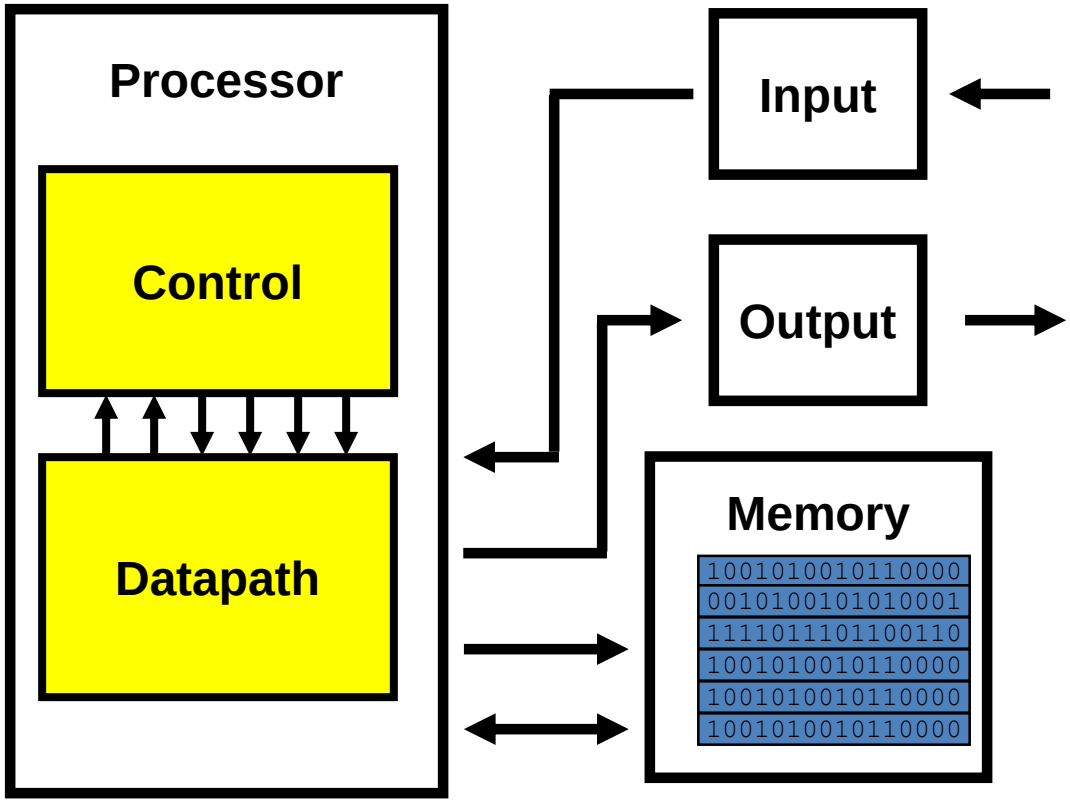
V.Charles – Fransa Kralı 1337-1380

"Geç bunları kralım; ben tek dil konuşurum anlayan geçer. Anlamayan seneye gelir"

Donanım Anabilim Dalı Hayrettin CAN – 2011

Bilgisayar Sistemi

- Bir bilgisayarın işlemcisi büyük bir lojik devredir.



İşlemci işlemi nasıl yapar?

- The “fetch/execute- (Alma(getirme) / yürütme)” süreci
  - İşlemci, komutu hafızadan getirir (fetches).
  - İşlemci, komutu “makine dilinde “yürütür (executes).

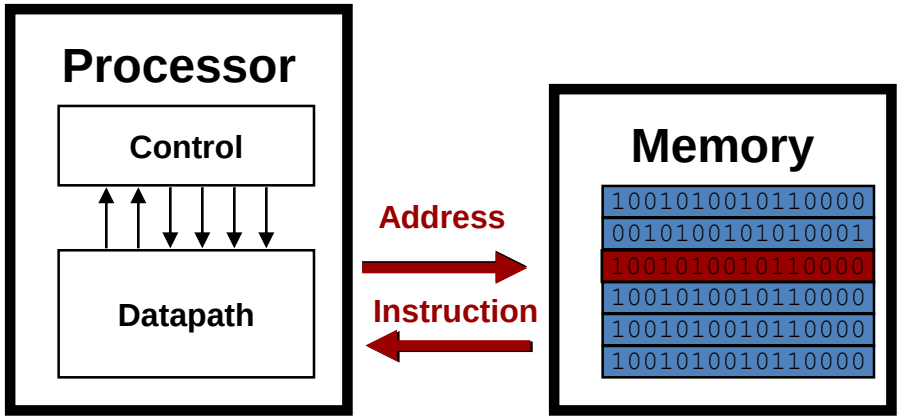
Yeni komut

Komutun yüklenmesi ve çözülmesi

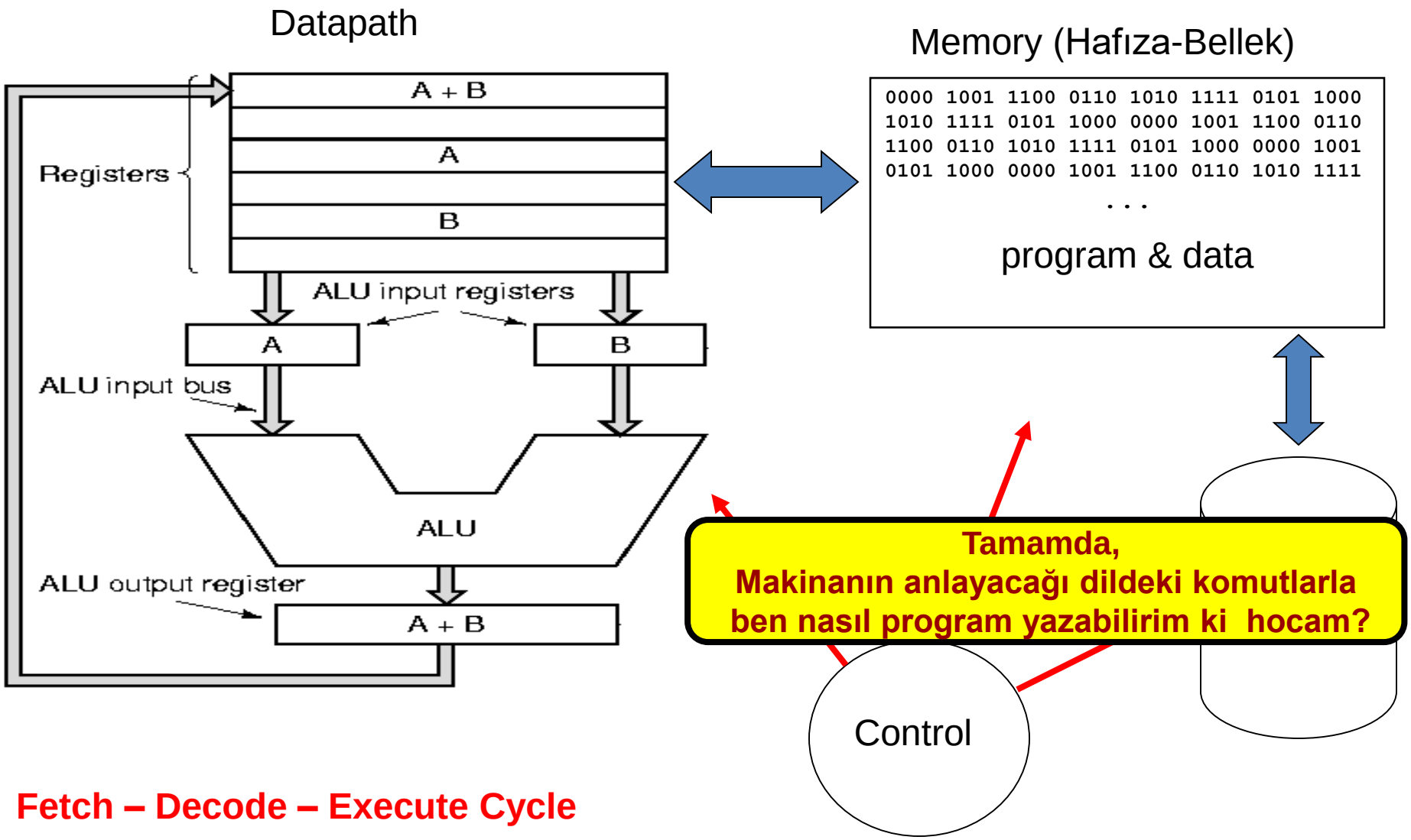
İşlenenin (Operand) getirilmesi

Operand’ın işlenmesi

Sonuçların kaydedilmesi



İşlemin yapılması -II



Fetch – Decode – Execute Cycle

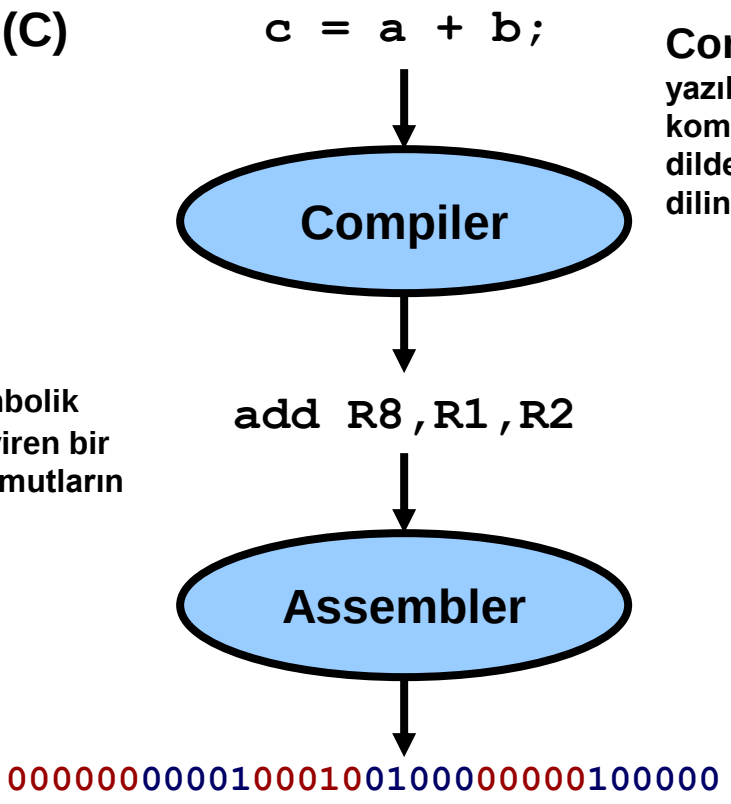
Bilgisayar sistemlerinde Soyutlama (biribirinden ayırma) (Abstractions in Computer Systems)

- Yüksek seviyeli bir dilde kodlanmış komutları makinanın işlemesi ve tekrar geri döndürmesi süreci oldukça karmaşık bir süreçtir.

- Tasarımcılar; karmaşıklığı yönetmek için soyutlama (ayrıştırma) tekniğini kullanırlar.
  - Sadece ilgili bilgilere odaklanılır.
  - Gereksiz ayrıntılara hiç bakılmaz.

Yazılımsal Ayrıştırma - Diller

Yüksek Seviyeli dil (C)



Compiler: **Yüksek seviyeli dilde yazılmış programı assemler sembolik dil komutlarına çevirir. Yüksek seviyeli dildeki birkaç satırlık program assemler dilinde çok daha fazla sayıda komut içerir.**

Assembler dili : **Bir sembolik komut kodunu binary forma çeviren bir programdır. (Binary koddaki komutların sembolik gösterimini yapan dil)**

Makine dili : Bilgisayar sistemiyle haberleşebilmek için , komutların binary formda oluşturulmuş şeklidir.

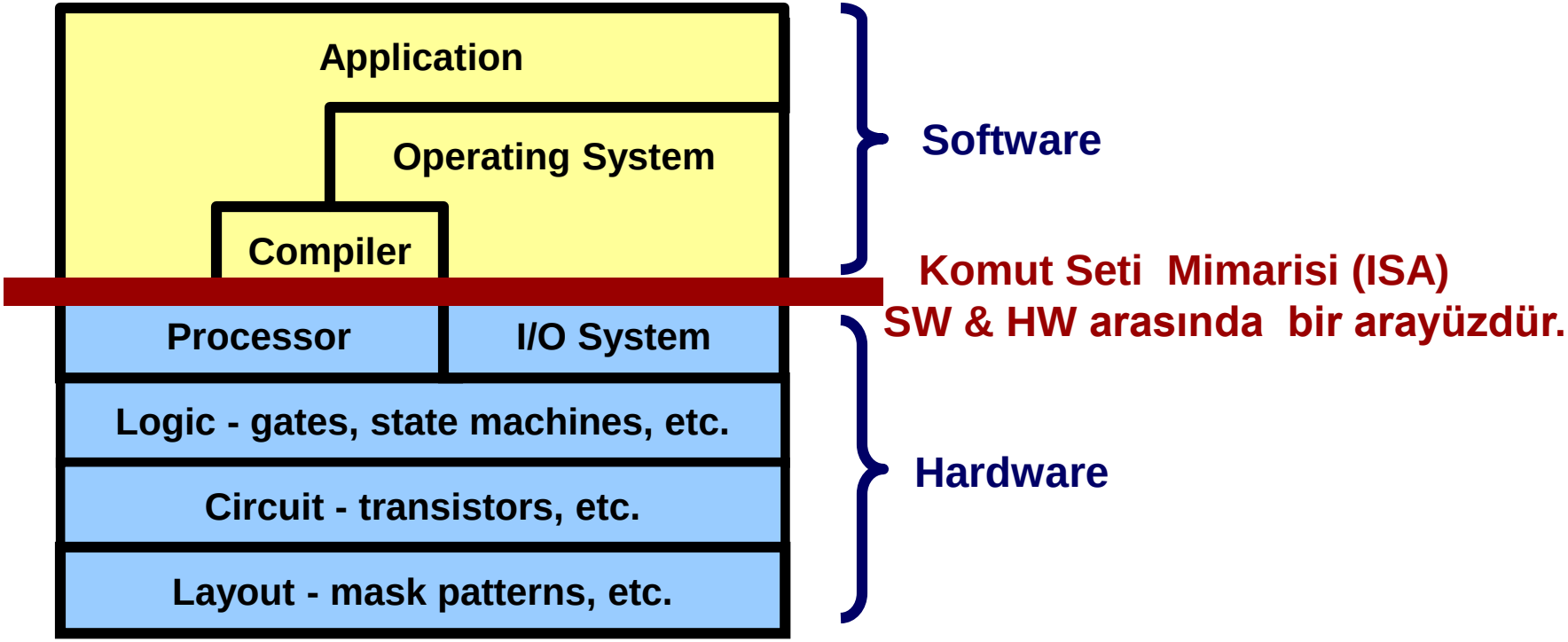
Yazılımsal ayrıştırma - Sistem Yazılımı

- İşletim Sistemi
  - Alt-seviye detayları programcıdan izole eder.
    - Sistem kaynaklarını yönetir.
    - Dosya sistemlerini yönetir.
  - Birden fazla programın işletilmesini koordine eder.
  - Kullanıcı programlarının sisteme zarar vermesini önler.
- **Kütüphaneler**
  - Üst düzey primitiv programlara , programcının erişimini sağlar.
  - İyi tanımlanmış arayüzler ile programlara erişim sağlanır(API).
- **Uygulamalar:** kullanıcı için özel fonksiyonları başarmak içindir.
  - Web Browser
  - Tablo
  - Word Processor

Komut Seti Mimarisi - Instruction Set Architecture (ISA)

(Donanım-Yazılım Arayüzü - The Hardware-Software Interface)

- Bilgisayarın en önemli ayrışımıdır



2.1. Giriş

- Bir bilgisayarı, donanımsal (hardware) olarak yorumlamak için onun dilini bilmelisiniz. Bir bilgisayar dilinin kelimelerine komut *(instruction*) denir. Ve genellikle bu kelimelerin bütününe *komut seti* diyeceğiz. Bu komutlar;

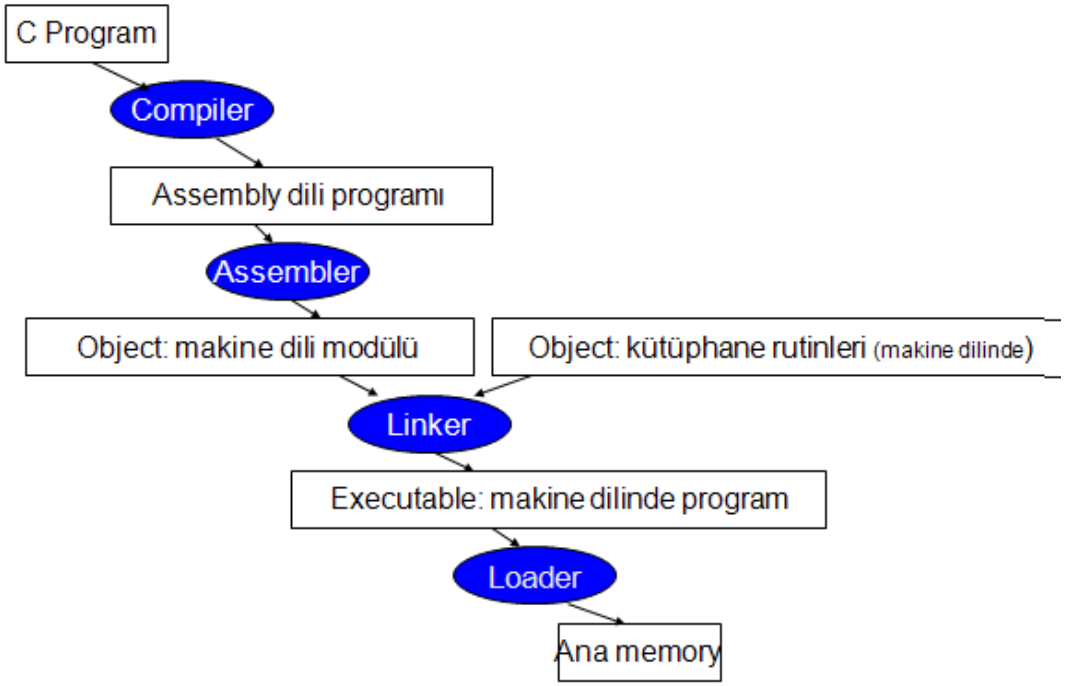
  - Kullanıcı tarafında: assembler dili
  - Bilgisayar tarafında : Makine dili

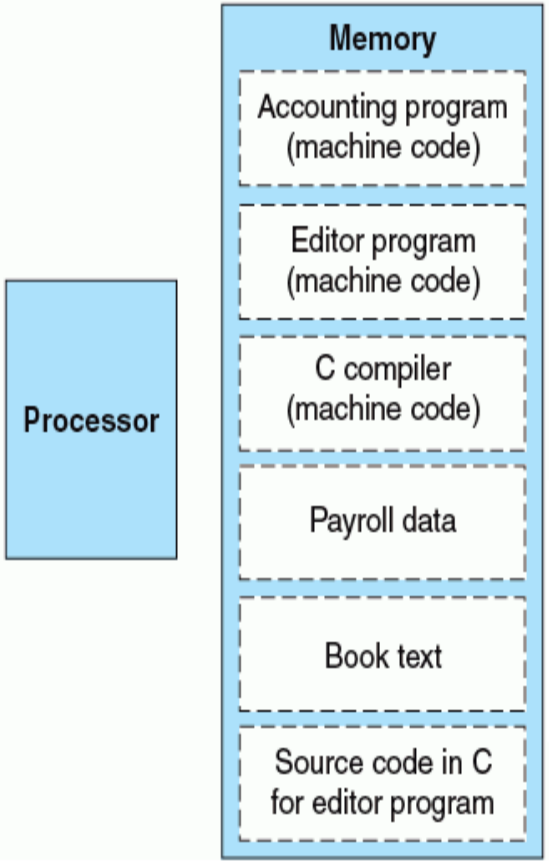
- Bir komut setinin öğrenilmesi demek;
  -Yüksek seviyeli programlama dilleri (C dili) ile ilişkisi nedir?
  -Yüksek seviyeli dille yazılmış komutu makine nasıl anlar?
  -Donanımsal olarak komutun gereği nasıl yapılır?
- Örneklerimizde yüksek seviyeli dil olarak C programlama dilini kullanacağız.

Komut seti mimarisi (ISA) SW/HW arayüzü olarak tanımlanabilir.

- 1980'lerden beri popular olarak kullanılan *MIPS (Mıllıon Instructıon of per Second)* komut seti mimarisini (Instruction Set Architecture – ISA) kullanacağız.
- MIPS komutlarını zamana yayarak adım adım öğreneceğiz.
- Komutların ve farklı tipten verilerin, hafıza biriminde sayı değerleri olarak saklandığını varsayan **"Stored program consept – Hafızalanmış (Depolanmış , saklı) Program konsept"** kullanacağız.

Stored program consept(Hafızalanmış Program Konsepti)





Günümüz bilgisayarları 2 önemli özelliği gözeterek inşa edilirler.

1-Komutlar, sayı sembolleri ile gösterilir.

2- Programlar ana hafızada kaydedilir. Ordan sayı sem

olarak okunur veya yazılır. Bu prensipler **"stored-program"** konseptidir.

Özel olarak; Hafıza, makine kodu oluşturan bir editör programının kaynak kodlarını ihtiva eder. Bu editör programı , ilgili yüksek seviye dilde yazılmış kodların makine kodu karşılığını text olarak oluşturur. Hatta bu editörde derlenmiş makine kodları doğrudan kullanılabilir.

ISA'nın temel prensipleri

- Komut seti mimarisinde (ISA) önemli dizayn prensipleri her zaman ön plandadır. Bunlara uyulunca donanımsal mimari optimize bir şekilde oluşur.

1-Basitlik düzenlilikten yanadır. (*simplicity favors regularity)*: Komutların belirli bir düzene göre işlenmesi Çözümü basitleştirir. Donanımın basitleşmesi için komutların tipini ve biçimini basitleştirtmek için kullanırız.

2- En küçük en hızlıdır (*smaller is faster):* İşlemciler sadece basit ilkellerin (primitive) uygulanmasını gözetmeli. Yani basit birimlerden oluşmalı. Register sayısı, ALU işlemlerinin sayısı, v.b donanım birimlerini doğru boyutta seçerken önemlidir.

3- *Make the common case fast : Çok kullanılanları hızlı yapın.*

4: Good design demands good compromises : İyi tasarım , iyi tavizler (uzlaşmalar) verilerek oluşur.

2.2 Bilgisayar donanımının İşlemesi
MIPS Komut Seti

- Herbir bilgisayar muhakkak olarak aritmetik işlemleri gerçekleştirebilmelidir.
  C kodu: a = b + c ;

- **Assembler kodu: (İnsanın anlayacağı makine komutları formu)**
- add a, b, c # b ve c'yi toplayıp a operand'ına koyar.

- **Makine kodu: (Donanımın anlayacağı makine komut formu)**
- 00000010001100100100000000100000

- ***MIPS'te her komut 3 operand'a (işlenen değer) sahiptir. Ve bir birim zamanda sadece 1 işlem başarılır.***

- ***Operandların sırası sabittir . (İlk operand hedef operand'dır.)***

- ***a, b, c aslında birer register (Kayıtçı)'dır. Registerlerin bit olarak kayıt kapasitesi, işlenenlere yazılacak değerler için çok önemlidir.***

- **Aşağıdaki C kodunu assembler koduna çevirelim.**

a = b + c + d + e

- Bu işlem 2'den fazla sayının toplanması işlemidir. MIPS'te aritmetik komutlar 3 operand'la işlendiğinden (En basit işleme şekli budur.) bu toplama işlemi ard arda toplamalar ile gerçekleşir.
- MIPS kodu :

```
add a, b, c     # a=b+c
add a, a, d     # a=b+c+d
add a, a, e     # a=b+c+d+e
```

- #-- yorum sembolüdür. Satır sonuna kadar etkilidir.
- Bir aritmetik işlem için, operand sayısı üçtür. Olması gereken budur ve daha az, daha fazla olmaması, "donanımın basitliği" felsefesine uyar.

Örnek:

Aşağıdaki 2 program parçası için C Compiler'ın çıkışı ne olur. (t geçici operand'ı ifade eder)

- C deyimi > C Compiler > MIPS instructions

a = b + c;

d = a –e;

```
add a, b, c
sub d, a, e
```

f = (g + h) – (i + j);

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

2.3 Donanımın İşlenenleri(Operands)

- Yüksek seviyeli programlama dillerinden farklı olarak MIPS'de herbir satır sadece bir komutun icra edilmesi içindir. Yani n+1 adet toplama için n tane komut icra edilmelidir (Simplicity favors regularitiy).
- Aritmetik komutlarda, İşlenenler (Operand) kaydedilmelidir. Bunun için MIPS'e yalnızca 32 register'in sağlanması yeterlidir. (Buna karşılık intel İşlemcilerde komutların hafızadaki verilerle direkt çalışılmasına izin verilir.)
- Register'lar ALU'ya en yakın küçük kapasiteli hafızalardır. İşlemcinin içindedirler.
- MIPS'te herbir register 32 bitliktir. Ve 32 register vardır.

  Bu ***MIPS-32***'dir. (*smaller is faster- 32 register'lı yapı 33 registerLI yapıdan daha hızlıdır.)*
- MIPS-64 ise 32 adet 64-bit register anlamındadır.

REGİSTER'ler Neye Var?

• C programlamada, her bir operand (değişken) ana hafızada tutulur.

• Donanımsal olarak, her seferinde ana hafızaya erişim zaman açısından pahalıdır.

•Eğer bir a operandına tekrar tekrar erişilecekse, bunu işlemci içerisindeki bir registere bir sefer getirip üzerinde işlem yapmak daha hızlı olur.

• Komutların icrasının kolaylaşması için, her bir komutu (add, sub v.b) yalnızca bir register'da işletmemiz gerekir.

• Not: C'de işlenenlerin sayısı çok fazla olabilir. Ancak assemler'da registerlerin sayısı sabittir. Fakat çok fazla değişken, geçici registerler ile işlenebilir.

MIPS'te Register isimleri için kural

- 32 adet registerleri numaralarını kodlayarak komutlarda belirginleştirebiliriz.
- *$s0, $s1, $s2, …Yüksek seviyeli programlama dillerindeki operand registerlerine karşılık gelir.*
- *$t0, $t1, $t2, … Yüksek-seviye dil kodlarını MIPS'te kodlarken kullanılan geçici değişkenler (temporary variables) için kullanılan registerlere karşılık gelir.*

Register Standart İsimleri (MIPS-32)

Name	Register Number	Usage	Preserved on call (Çağrılarda içeriğinkorunması)
$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

Örnek:

$$f = (g + h) – (i + j)$$

Örneğindeki **f, g, h, i, j** değişkenleri (Operand) **$s0, $s1, $s2, $s3,$s4** registerlerin'a atanmış olsun. Bu işlemin MIPS kodu karşılığı nedir?

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)
```

Hafıza İşlenenleri(Operands)

- *Şimdiye kadarki örneklerde birkaç değişken üzerinde işlem yapıyorduk. Bu durumda da registerlerde problem çıkmıyordu.*
- Fakat öyle işlemler vardır ki çok sayıda verinin kullanılması gerekir. Yani karmaşık ve kapasiteli bir veri bloğuyla çalışılması sözkonudur. Bu durumda işlenen sayısı register sayısından çok çok fazla olabilir.
- Bu durumda milyonlarca tane veriden oluşan veri yapılarını *ana hafızalar* üzerlerinde bulundururlar.
- **Ana hafıza;** yüksek seviye-programlama dili komutlarının,verilerin ve sonuçların saklandığı yerdir.
- **Registerler** ise, MIPS komutlarının çalıştırdığı aritmetiksel işlemlerin yapıldığı dataları bulundurur.

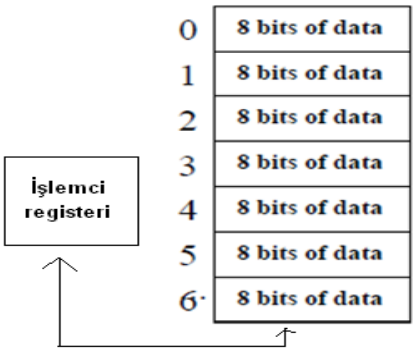
- Ana hafızadaki veriler, registerlare MIPS'ın içerdiği transfer komutlarıyla çağrılır. Bu komutlara *"veri transfer komutları (**lw, sw**)* denir.
- Hafızadaki bir veri kelimesine ulaşabilmek için; komut, ulaşılacak verinin hafıza adresini beslemelidir. Çünkü hafıza tek boyutlu çok büyük bir dizi şeklinde, adreslenmiş bir yapıdır.
- *Bir hafıza adresi, dizi içerisinde bir indextir (0,1,2…).*
- *"Adreslenmiş byte" anlamında, index hafızadaki 1 baytlık yeri işaret eder.*

Ana hafızadaki bir operand'ın atanmasının derlenmesi- Örnek (Compiling an assignment when an operand is in memory)

- Ana hafıza dizi şeklindeki operandlar'ı (İşlenenler - veri) registerler ile ilişkilendirmek için , derleyici bellekte bir yer ayırır. Ve data transfer komutu için bir başlangıç adresini de belirler.
- A , 100 baytlık bir dizi olsun. g için $s1 registeri, h için $s2 registeri , adres bloğunun başlangıç adresi için $s3 (başlangıç registeri) kullanılsın (Veriler 1 bayt'lık olsun).

g = h + A[8]; işlemi için

```
lw      $t0, 8($s3)     # temporary register $to'a A[8]'in get,rilmesi.
add     $s1, $s2, $t0
```



Başlangıç registeri *(Base register): Dizinin başlangıç adresi*

offset: *Data transferi için sabit(8).*

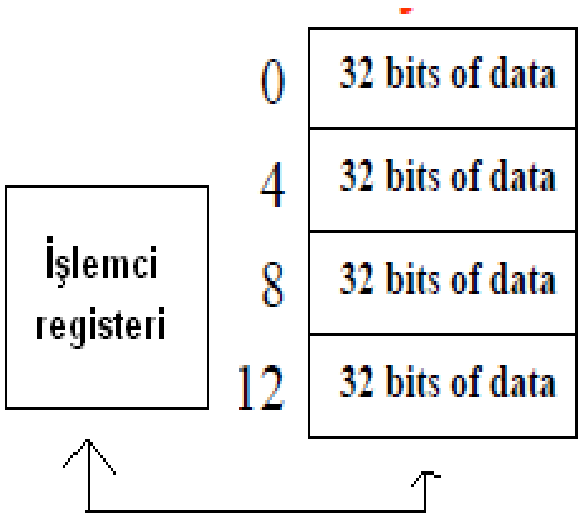
**** *Çoğu veri ögeleri 1 baytlık olabildiği gibi, 4 baytlık(32 bitlik) 64 baytLık veri ögesi uzunlukları da söz konusudur. 32bitlik veri elemanlarının transfer edilmesi ise aşağıdaki gibidir.*

- Bir önceki problemi 32bitlik kelime uzunluğu için kodlayalım.
  - Çoğu veri ögeleri 1 bayttan dah uzun kelimeler kullanır.
  - MIPS'te registerler 32bit (4 bayt) veri tutarlar.

0.kelimenin byte adresi 0'dan başlar

1.kelimenin byte adresi 4

2.kelimenin byte adrei 8…




```
lw      $t0, 32($s3)   #8x4=32
add     $s1, $s2, $t0
```

load ve Store (lw ve sw) kullanımının derlenmesi (Compiling Using Load and store)- örnek

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

Bu işlem C kodunda tek bir işlem olmasına rağmen ana hafızada 2 tane işlenen (değişken) vardır. Dolayısıyla 1'den fazla MIPS komutuna ihtiyaç vardır.

MIPS kodu :

```
lw    $t0, 32($s3)    #32 = 8x4
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # toplama sonucu nu ana hafızaya kaydet (store word) .
```

• Derleyicinin atamaları:

–$s3'ü başlangıç adresi için (A dizisinin başlangıç adresi)

–$s2' yi h değişkeni için.

• Hatırla!.Aritmetik işlemler registerlardaki verilerle yapılır. Hafızadakilerle değil.

Bu komut çalışmaz : **add 48($s3), $s2, 32($s3) niye?**

Sabit veya doğrudan işlenenler (değişkenler) (Constant or Immediate Operands)

- Çoğu zaman programlar, İşlemlerin bir çoğunda sabit değerler kullanır. Dolayısıyla MIPS aritmetik komutlarının yarıdan fazlası sabitlerle yapılan işlemlerdir.
- ***Açıklama Notu: Bu konuyla ilgili Donanım-Yazılım etkileşimi***

✓ ***Birçok programlarda çok sayıdaki değişken sayısı, bilgisayarın sahip olduğu sınırlı sayıdaki registerlardan fazladır. Dolayısıyla derleyici; en sık kullanılan değişkenleri registerlar'da tutmaya çalışır. Register'ler ve hafıza arasında yer değiştiren (Load ve store komutları ile) değişkenleri ise hafızanın belirli bir kısmına yerleştirir. Daha az kullanılan değişkenlerin konulduğu hafızanın belirli bölgesine " Sıçrama registerleri- spilling registers " denir.***

✓ ***Donanım prensibinden dolayı, hız ve kapasite ters orantılıdır. Dolayısıyla ana hafızanın hızı registerlere göre yavaştır.***

✓ ***Registerdaki veri daha kullanışlı, faydalı bir veridir. MIPS aritmetik komutları iki registeri okuyabilir, onlarla işlem yapabilir, onlara yazabilir. MIPS'in data transfer komutları ise sadece bir işleneni (operand) okuyabilir veya yazabilir . Onlarla işlem yapamaz.***

✓ ***Performanslı bir derleyici, registerleri etkili olark kullanan derleyicidir.***

- Biz şu ana kadar sadece komutları kullanmayı gördük. Bu komutları kullanmak için, ana hafızadan bir sabit yüklenmelidir. (Sabitler, program yüklenirken ana hafızaya yüklenirler).

• Örneğin ; C dilinde , *AddrConstant4 (programın başlangıcında* $s1 ile ilişkili hafıza yerinde olduğunu varsayalım*) adresindeki 4 sabitini ,* $s3 registerine ekleyelim:

```
lw      $t0,  AddrConstant4($s1)      # $t0 = constant 4
add     $s3, $s3, $t0                 # $s3 = $s3+ $t0
```

Veya;

```
addi    $s3, $s3, 4                   # $s3 = $s3+ 4
```

addi *"immediate instruction - Doğrudan, derhal işlenen komut"* olarak adlandırılır. Bu komut sabitlerle çalışmaya müsaade eder. Bu komutlar donanımsal 3.dizayn prensibini gösterir.

•*Dizayn prensibi 3:* ***"Make the common case fast"***

Özet

MIPS

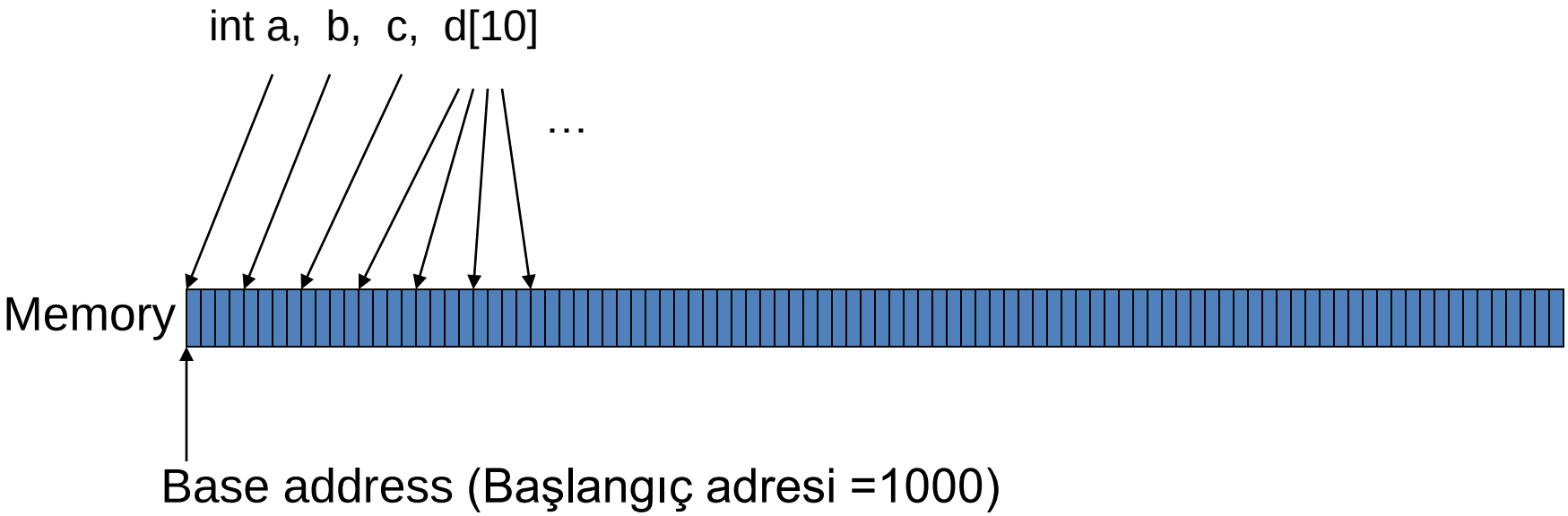
— Adreslenmiş baytları loading/storing yapar.

— Sadece registerlerda aritmetik işlemler yapar.

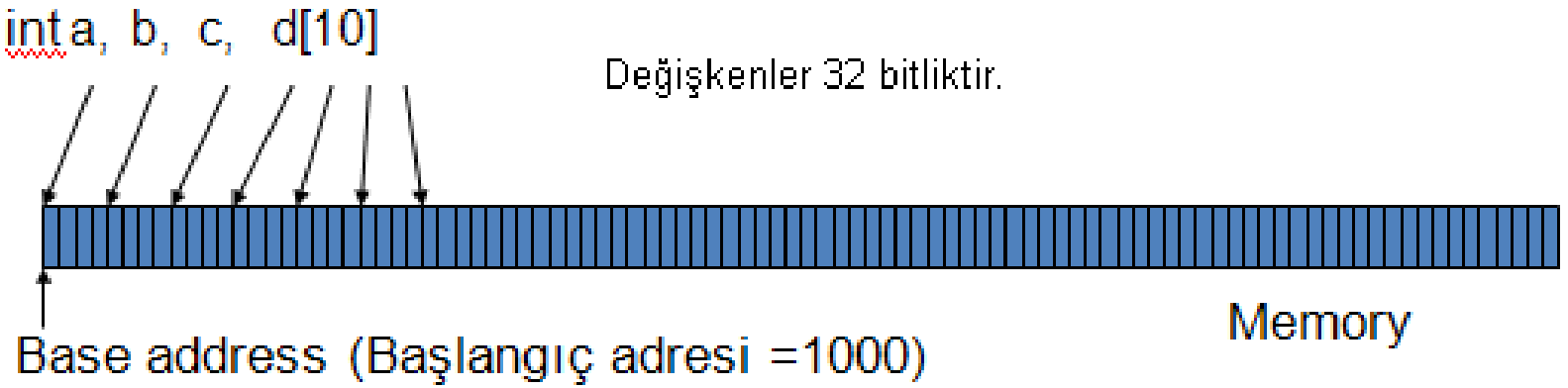
	Komutlar	Anlamı
add	**$s1, $s2, $s3**	**$s1 = $s2 + $s3**
addi	**$s1, $s2, 100**	**$s1 = $s2 + 100**
sub	**$s1, $s2, $s3**	**$s1 = $s2 –$s3**
lw	**$s1, 100($s2)**	**$s1 = Memory[$s2+100]**
sw	**$s1, 100($s2)**	**Memory[$s2+100] = $s1**

HAFIZA Adresleri (3.hafta ders başlangıcı)

- Compiler datayı(veriyi) hafızada organize eder. …
- O her operand'ın lokasyonunu (nereye kaydedildiğini) bilir.
- Compiler, load/store komutlarının gereği için hafızanın adresi belli bölgelerini ayırıp kullanır (*spilling registers*).



Immediate Operands - Örnek

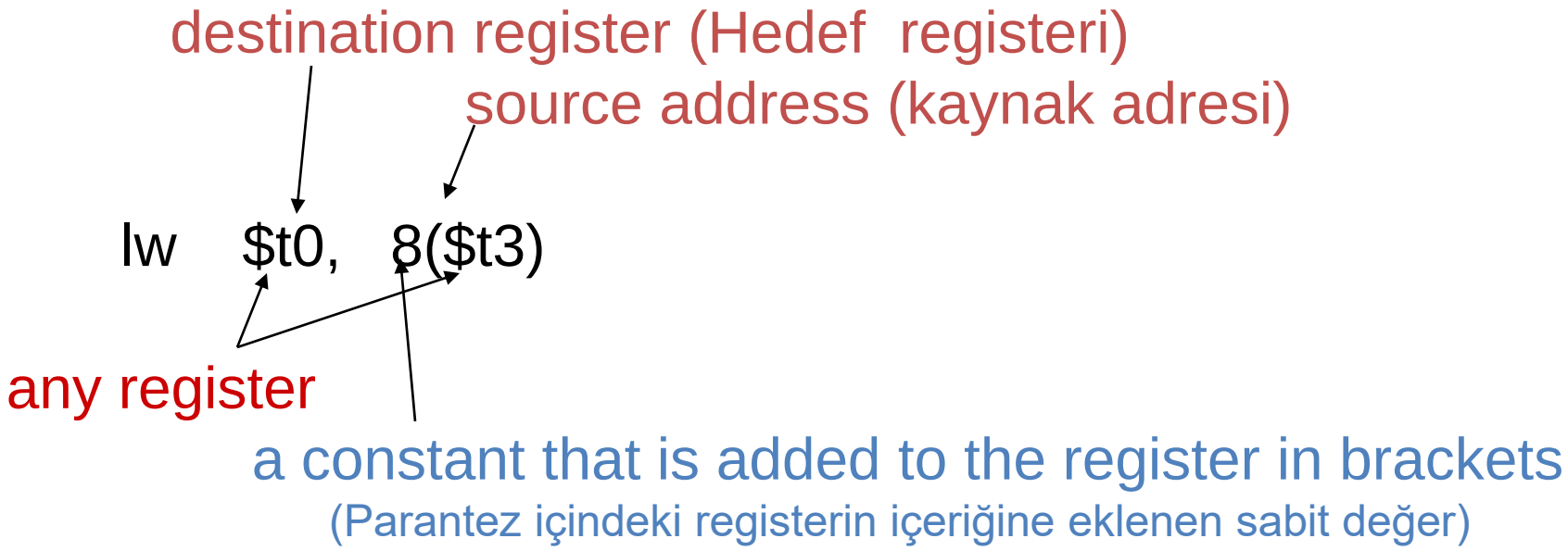


- Bir komut sabit bir giriş değerine ihtiyaç duyar.
- Bir immediate komut,girişlerden birini sabit bir değer olarak kullanır (Bir register operand'ı yerine).

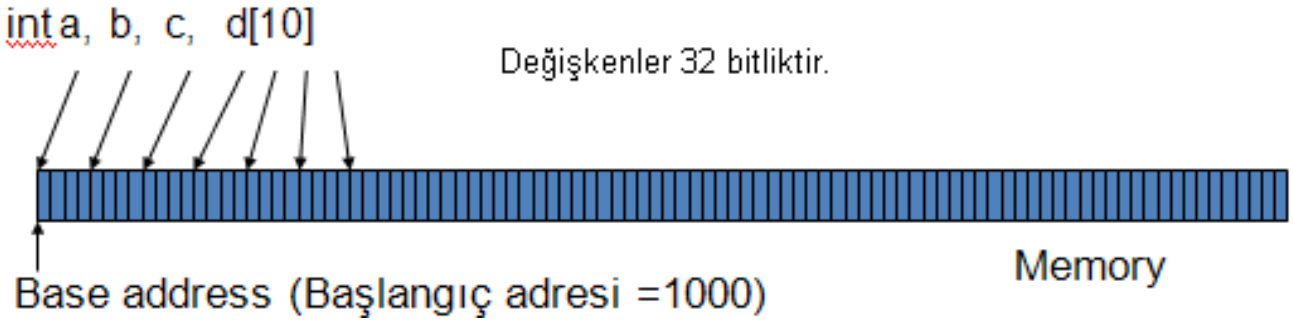
```
addi   $s0, $zero, 1000   # programın başlangıç adresinin 1000 olduğu, bunun $s0
                          # 'a kayıt edilmesini,  $zero ise devamlı 0'a eşit olan bir
                          #    register olduğunun belirtilmesi.
addi   $s1, $s0, 0        # Bu a operand'ının adresidir.
addi   $s2, $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] ın adresidir.
```

Memory Instruction Format
(Hafızadan transfer komutunun formatı)

- Hafızadan reg'e transfer komutunun formatı;



Örnek



Aşağıdaki C kodunu Assemler 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'In içeriği  d[3]  nolu hafızadaki adrese  yazılır.
```

Özet tablo

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $t0, $t1, . . .	Registerler DATA için hızlı yerleşimlerdir. 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 ardılış 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)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

HAFIZA YERLEŞİMİ

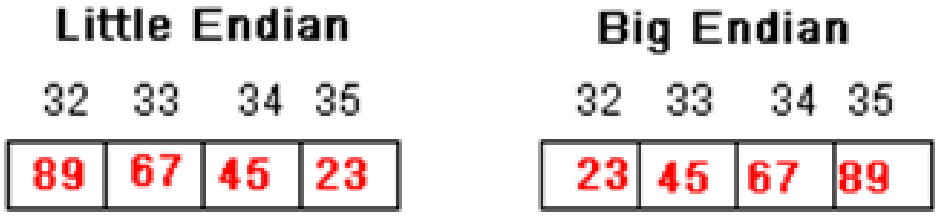
Büyük kapasiteli tek boyutlu dizi şeklinde gösterilen 32 bitlik veya 64 bitlik veriler, MIPS'te byte (8 bit skalyıcılar) veya kelime şeklinde ifadelerin saklandığı 8 bitlik veri saklayan benzersiz adreslerde hafızalanır. Dolayısıyla MIPS'te 32 bitlik (4 byt'lık) veri kelimeleri ardışıl 4 hafıza adresine ihtiyaç duyar.

Sonuç olarak MIPS'te ardarda 32 bitlik verilerin saklandığı hafıza gurubu adreslerinin sonuncusunun başlangıç adresi bir öncekinden 4 adres sonra olmalıdır.

Big Endian Yerleştirme şekli: 32 bitlik kelimenin en ağırlıklı 8 bit'ı(En soldaki byte - MSByte) bu kelime için ayrılmış ardışıl 4 hafıza adresinin en küçüğüne (ilkine)yazılır. Diğer byte'lar sırasıyla yazılır. Bu kullanım tarzı; MIPS, IBM 360, SUN Sparc mimarilerinde kullanılır.

Little Endian yerleştirme şekli: 32 bitlik sözcüğün en düşük 8 biti (LSByte) bu kelime için ayrılmış ardışıl 4 hafıza adresinin en küçüğüne (ilkine)yazılır. Big Endian formatının tersidir. Bu kullanım tarzı Intel 80X86, DEC Vax mimarilerinde kullanılır.

Örnek: 32 bitlik hexa decimal sayı olsun. **23456789**



Örnek:

a) Aşağıdaki işlemin sonucunu elde ediniz. Data hafızasının başlangıç adresi 0'dır. A dizisinin başlangıç adresi 8'dir.

b) 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)
```

Data Memory

19	0	1	0	1	0	1	0	1
18	1	1	0	0	1	0	1	0
17	0	0	1	1	1	1	0	1
16	1	0	0	1	1	1	0	1
15	0	0	1	1	0	0	0	1
14	0	1	0	0	1	1	0	1
13	1	1	1	1	0	1	1	0
12	0	1	0	0	1	1	0	1
11	1	0	1	1	1	0	1	0
10	0	1	0	1	0	1	0	1
9	0	1	0	1	1	0	1	1
8	1	1	1	0	0	1	0	0
7	1	0	1	0	1	1	0	1
6	0	1	0	1	1	1	0	1
5	0	0	1	0	1	0	1	0
4	0	1	0	1	0	1	0	1
3	1	0	0	0	0	0	1	1
2	0	1	1	1	1	1	1	0
1	1	0	0	0	0	0	1	1
0	0	1	0	0	0	1	1	1

39	1	1	1	0	1	0	1	1
38	1	0	0	0	1	1	0	0
37	0	0	0	1	1	0	1	0
36	1	0	0	0	0	1	1	0
35	1	0	1	1	0	1	0	1
34	1	1	0	0	1	0	1	1
33	0	1	0	0	1	1	0	0
32	0	0	0	1	1	1	0	0
31	0	1	1	0	1	0	0	0
30	0	0	1	1	0	0	1	1
29	0	0	1	1	0	0	0	1
28	0	1	1	1	0	1	0	1
27	1	1	0	0	1	0	0	0
26	1	0	0	1	1	0	1	0
25	0	1	1	0	0	0	0	1
24	0	1	0	0	1	1	0	1
23	1	1	1	1	0	1	0	1
22	0	1	0	1	1	0	1	0
21	1	0	0	0	1	1	0	1
20	0	1	1	0	1	0	1	0

2.4 Komutların Bilgisayarda temsil edilmesi (representing instructions the Computers)

- Bilgisayar sistemler binary sayı sisteminde çalışırlar. Bu sayı sisteminde 0 ve 1 olmak üzere 2 rakam vardır.
- Bu rakamların donanımsal olarak tanınması ve işlenmesi, sadece 2 durumda (düşük seviye = Lojik 0, ve yüksek seviye = lojik 1) bulunabilen elektriksel gerilim seviyeleri ile olur.
- Sözcükler 1 ve 0'ların yanyana dizilmesiyle oluşur. Sözcüğü oluşturan her rakama (0 veya 1) (binary digit) BİT denir.
- Komutlar ve sabitler ve değişkenler binary sözcükler şeklinde ifade edilir.
- MIPS assembler dilinde, *$s0* dan *$s7* ye kadar olan sembolleri 16 dan 23 'e kadarki registerleri, *$t0* dan *$t7* arasındaki semboller *8* den 15' e kadarki registerleri haritalar (MIPS-32 32 adet register'la işlem yapar). Burada $s0 16. registerdir. $t0 ise 8.registerdir.
- 32 tane registerin geri kalanları ise sonra açıklanacaktır.

Nostalji

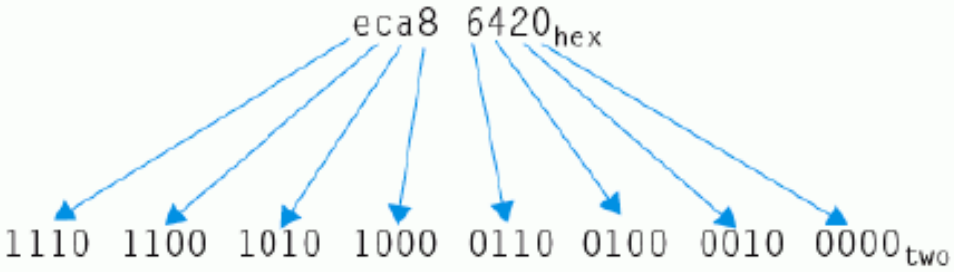
Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

Binary-to-Hexadecimal and Back

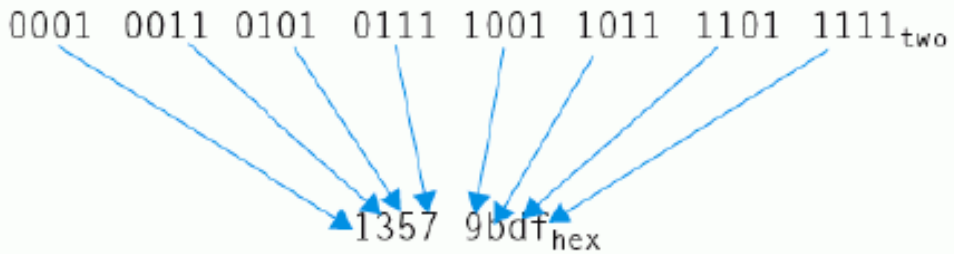
Convert the following hexadecimal and binary numbers into the other base:
$eca8\ 6420_{hex}$

$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{two}$

Just a table lookup one way:



And then the other direction too:



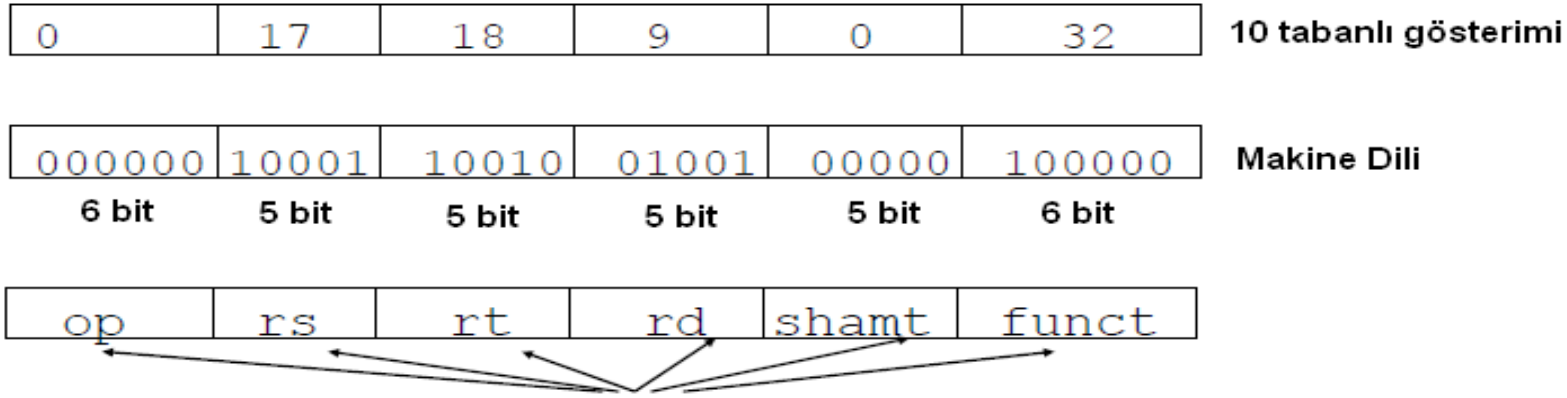
MIPS assembler komutunun Makine komutuna dönüşümü

• Komutlar, veri kelimelerinin uzunluğu ve registerler 32 bit uzunluğundadır.
– Örnek

add $t1, $s1, $s2

– kullanılan registerler; $t1=9, $s1=17, $s2=18

• **Bir komutun formatı**



- op (6 bits): Komutun temel işlevini belirtir (aritmetik, transfer, lojik, v.b), *opcode*
- rs (5 bits): 1.operand (Işlenen) için kaynak registeri belirtir.
- rt (5 bits): 2.operant için kaynak registerini belirtir.
- rd (5 bits): Sonucun yerleştirildiği hedef (varış yeri) registerini belirtir.
- shamt (5 bits): Kayma miktarı (Daha sonra açıklanacak) shift amount
- funct (6 bits): Başarılacak fonksiyon, fonksiyonun kodu (toplama, çıkrama v.b)

•Register gösterimi için niye 5 bitlik yer ayırıyoruz? Niçin
•Başarılabilecek fonksiyonların sayısı kaçtır? Neden

MIPS alanları

- Bir komut daha büyük bir alan ihtiyaç duyabilir. Örneğin ***lw komutu*** için , 2 tane registere ve bir tane sabite ihtiyaç vardır. Eğer sabiti ifade etmek için 5 bitlik adres alanı kullanılacaksa siz sabit bir değer olarak en fazla 32'yi ifade edebilirsiniz. Ancak, bir diziden veya veri yapısından eleman seçmek için kullanılan bu sabit değer çoğu zaman 32'den çok büyüktür. Bu nedenle, tüm komutları aynı uzunlukta tutmak ve tek bir komut formatı kullanmak oldukça zordur.. Bu durum bizi 4. donanım tasarım prensibine götürür.
- *Design principle 4: Good design demands good compromises.*

  (iyi tasarım , iyi tavizler (uzalaşmalar) verilerek oluşur)

  MIPS tasarımcıları, komut uzunluklarının benzer olması için, farklı gurublardaki komutlar için farklı komut formalrı oluşturmak yoluna gitmişlerdir.

Örnek;

R-Format'lı komutlar (registerler için)

I-Format'lı komutlar (sabitler veya veri transferi için)

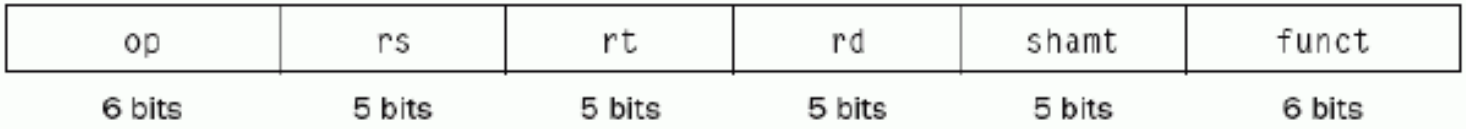
J-Formatlı komutlar (Şartsız Dallanmalar için)

R-tipi, I-Tipi komut formatları

I- tipi komut formatında, 16 bitlik alanın anlamı, başlangıç adres registerine 2**15 = 32.768 tane bayt adresi veya sabit değerin yüklenebileceğidir.

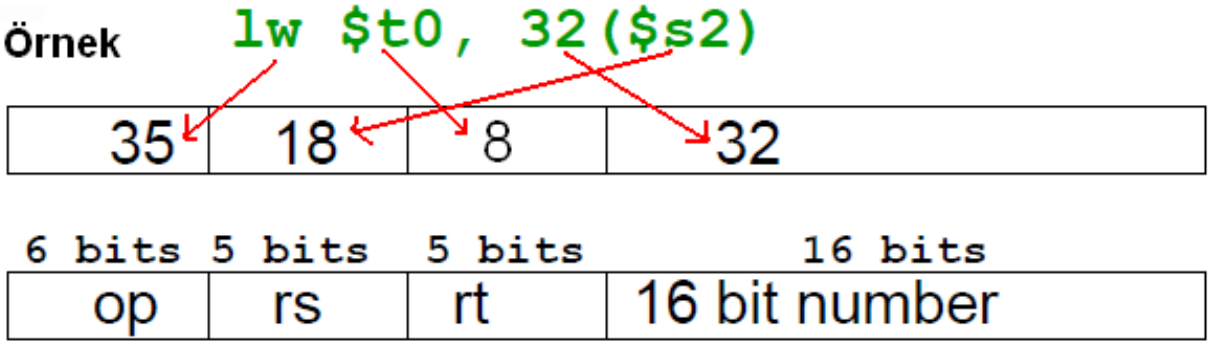
I- tipi komut formatında; ***rt alanı*** hedef register olarak değerlendirilir. Yani yüklemenin sonucunun yazıldığı registerdir.

R-Tipi (Registerlerdaki işlem komutları için)



I -Tipi (Sabitler veya veri transfer komutları için)

Örnek



UNUTMA !!!!!!!!

Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

** I-tipi format immediate(doğrudan) ve data transfer komutları içindir
** R-tipi formak Register komutları içindir.
** J-tipi format şartsız dallanma komutları içindir.

MIPS komutlarının kodlanması

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.6 MIPS instruction encoding.

- Dikkat !! ***subi*** komutu yoktur. Sadece ***addi*** komutu vardır. Sabit bir negatif değer olabilir.
- Formatın opcode kısmı komut kümesini belirtir (0: toplama veya çıkarma v.b aritmetik işlem komutlar, 35: lw komutu)
- Func kısmı ise komut gurubundaki özel işlem komutunu belirtir (toplama:32 v.b)

MIPS assemblerden, makine dili koduna çevrimine örnek (S.65):

$t1 A dizisinin başlangıç adresini, $s2 h operandını tutar.

```
A[300] = h + A[300];

lw   $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

Bu üç komutun MIPS makine dili kodu nedir?

Komutların desimal karşılığı

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Komutların binary karşılığı

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

8D2404B0
02484020
AD2804B0

ÖĞRETİLENLERİN ÖZETİ

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $s7 $t0, $t1, . . . , $t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16–23 and $t0-$t7 map to 8–15.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

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
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

LOJİK İşlemler, operatörler, (2.5. Logical Operations)

- İlk bilgisayarlardaki komutlar veri sözcüklerini bir bütün olarak ele alırdı. Ancak; kelime içindeki bit alanları ve hatta bireysel bitlerin üzerinde çalışmasınında yararlı olduğu ortaya çıktı. Her biri 8 bit olarak saklanır bir kelim içinde karakter incelenmesi böyle bir operasyona örnektir.
- Bu tip komutlar; paketler veya kelimeler şeklindeki verilerin içeriğini, bit açma, değerlendirme için kullanılır.
- Bu komutlar mantıksal(Lojik) işlemleri olarak bilinir.

Logical Operations	C operator	Java operator	MIPS instruction
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

- # Shift operations (kaydırma İşlemi):

sll $t2, $s0, 4 # $t2 = $s0 << 4

- $s0 registerinin içeriğini 4 bit sola kaydır. (Sağdan 0 doldurarak), ve sonucu $t2 registerine yaz.
- $s0'ın içeriği; 00000000 00000000 00000000 00001001 = 9

  Komut işlendikten sonraki değer.
- $t2'nin içeriği; 00000000 00000000 0000000 10010000 = 144
- 4 , "shift amount - shamt" olarak isimlendirilir ve *R-tipi komut formatında, kaydırma alanını oluşturur.*
- Hatırlayınızki bir bitlik sola kaydırma, sözcüğün 2 ile çarpılmasına eşdeğerdir.
- *Bu komut R-tipi bir komut'tur. Op ve func alanı 0'dır. Rs alanı kullanılmaz. 0 yazılır.*

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

$t2 0000 0000 0000 0000 0000 1101 0000 0000 ise

$t1 0000 0000 0000 0000 0011 1100 0000 0000 ise

and $t0, $t1, $t2 # $t0 = $t1 & $t2

$t0 0000 0000 0000 0000 0000 1100 0000 0000 olur.

or $t0, $t1, $t2 # $t0 = $t1 | $t2

$t0 0000 0000 0000 0000 00011 110010000 0000 olur.

<u>Önemli:</u> MIPS'te NOT işlemi, nor komutu ile başarılır. Çünkü;

A Nor 0 = NOT (A OR 0) = NOT (A)

nor $t0, $t1, $t3 # $t0 = ~($t1 | $t3)

$t3 0000 0000 0000 0000 0000 0000 0000 0000 ise

$t0 1111 1111 1111 1111 11000011 1111 1111 olur.

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, ..., $s7 $t0, $t1, ..., $t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16–23 and $t0-$t7 map to 8–15.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory

FIGURE 2.10 MIPS architecture revealed thus far.

- Tablodan da görüldüğü gibi MIPS'in lojik komutları sabit değerler (immediate) ile de işlem yapabilirler. Bu lojik komutlar *andi* ve *ori* dir.

```
andi    $s1, $s2, 100     # $s1 = $s2 & 100
ori     $s1, $s2, 100     # $s1 = $s2 | 100
```

2.6. instructions for making decisions (Karar Komutları)

- Bir bilgisayarı basit hesap makinesinden ayıran en önemli özellik karar verebilme yeteneğidir. Giriş verileri ve değişkenler arasına istenen işlemlerin yapılması sürecinde birçok farklı komutlar çalışır.
- Pororamlama dillerinde karar verme deyimi ***if*** dir. Bu deyim bazen de *go* deyimi ve bir etiket ile kombine edilir.

C 'de if deyiminin 2 tipi vardır.

- ***if (şart) deyim (clause)***
- **`if (`*şart*`) `*deyim1* `else` *deyim2***

2. `if` aşağıdaki gibi de tarif edilir .

```
if   (şart) goto L1;
        deyim2;
       go to L2;
   L1: deyim1;

   L2:
```

- **if – else kadar şık değil, fakat aynı anlamdadır.**

MIPS'teki karar komutları

- MIPS assembler dili iki tane karar verme komutu içerir. Bunlar *if……go* deyimlerine benzerdir (Şartlı dallanma komutları).

1-Branch if equal (beq - Eşitse dallan):

```
beq   register1, register2, L1   # register1'in içeriği register2'ni
                                 # içeriğine eşit ise L1'e dallan.
```

Bu deyimin C'de karşılığı;

```
if    (register1==register2) goto L1
```

2- Branch if not equal (bne - Eşit değilse dallan):

```
bne   register1, register2, L1   # register1'in içeriği register2'nin
                                 # içeriğine eşit değilse L1'e dallan.
```

Bu deyimin C'de karşılığı

```
if    (register1!=register2) goto L1
```

Örnek: Aşağıdaki C kodunun MIPS kodu karşılığını bulunuz.

if(i==j) f= g + h; else f = g - h;

- Buradaki f, g, h, i, j değişkenleri; MIPS' te sırasıyla f→$s0 , g → $s1, h → $s2, i → $s3, j → $s4 registerleri ile ilişkilendirilmişlerdir.

  MIPS kodu aşağıdadır.

```
        bne  $s3, $s4, Else   # eğer  i ≠ j ise Else etiketli yere git
        add  $s0, $s1, $s2    #  h = g + j , eğer  i ≠ j  ise atlanır (işlenmez)
        j        exit         # Exit etiketli yere atla. (Şartsız dallan).
Else:   sub   $s0, $s1, $s2   #  h=g-j , eğer i=j ise  atlanır(işlenmez)
Exit:
```

Not: 1-) Hafızadaki değişkenlerin registerlere transfer edilmesi gösterilmemiştir.

2-) Atlama yerlerinin adresleri (etiketleri) compiler ve assemler dili programlayıcısı tarafından belirlenir. Y.Seviyeli dil değil.

MIPS *şartsız dallanma komutu*

j etiket

```
goto label ; C'deki kod.
```

- Aynı örneği başka şekilde gerçekleştirelim

```
if (i!=j)                beq   $s4, $s5, Else
h=i+j;                   add   $s3, $s4, $s5
else                       j   Exit;
h=i-j;             Else: sub   $s3, $s4, $s5
                   Exit:..
```

Not: beq $s0, $s0, etiket

j etiket ile aynı işi görür mü?

Döngüler (LOOPS)

- Kararlar hem, iki alternatif arasında seçim yapmak için (if deyimi ile karar), hemde bir hesaplamadaki iterasyon için (döngü şeklinde bulunur).
- Her iki durum için benzer assemler komutları kullanılır.

Örnek: C'de kodlanmış aşağıdaki *while* döngüsünün MIPS assemler kodu nedir?

```
while ( save[i] == k )
{
        i += 1;
}
```

varsayılım ki; *i* ve *k* sırasıyla $s3 ve $s5 registerlerindedir ve dizinin başlangıç değeri ise $s6 registerine kayıt edilmiştir.

İlk adım, **save[i]** operandını bir geçici (temporary) registere yüklemektir.

- **save[i]** operandını geçici registere yükleyebilmek için önce onun adresini bilmeye ihtiyaç vardır.
- *i* değişkenini , save dizisinin başlangıç adresine eklemeden önce , i değişkenini 4 ile çarpmalıyız (her bir kelime 32 bit olduğu için). Bu işlemi i sözcüğünü 2 defa sola kaydıran komut (sll) ile yapabiliriz.
- Biz programa bir LOOP isimli döngü etiketi eklemeliyiz. Böylece döngünün sonunda, dallandığımız yere geri dönebiliriz.

```
Loop:   sll     $t1, $s3,2     # $t1 =  4*i yükle
```

Save[i] dizi indexsinin adresini elde edelim.(dizinin başlangıç adresine $t1'in içeriğini yükleyerek)

```
        add     $t1, $t1 ,$s6          # $t1 = save[i] adresi
```

Elde edilen bu adres değeri ile, save[i] içeriğinin $t0 register'ine e transferi.

```
        lw      $t0, 0 ($t1)   # $t0 = save[i]
```

Save[i] ≠k 'ya göre LOOP testi

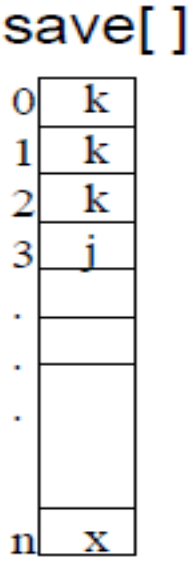
```
        bne     $t0, $s5,  Exit  # eğer Save[i] ≠k  ise,  Exit'e git
```

i'ye 1 eklenme işlemi

```
        addi    $s3, $s3,1       # i = i + 1
```

Loop dönüsünün başına şartsız dallanma işlemi

```
        j       Loop     # Loop döngüsüne git.
Exit:
```



Karşılaştırılanların eşit olması veya olmaması testi

- Değişik şekillerde başarılabilir. Görme testi (to see) de bunlardan birisidir.
- MIPS'te , bu işlem karşılaştırma komutu ile yapılır. Bu komut ***"set on less than"*** *veya* ***slt*** komutudur. Bu komut iki registerin içeriğini karşılaştırır; eğer 1. registerin içeriği 2.registerin içeriğinden küçük ise 3. bir registeri 1' set eder. Tersi ise 0'a set eder.

```
slt     $t0, $s3, $s4    #  $s3 < $s4 ise $t0 = 1
```

- Sabit operand'ların karşılaştırmasına çok sıkça rastlanır. ***$zero*** registerinde devamlı 0 yüklü olduğundan 0 ile karşılaştırmalar bununla yapılır. Diğer sabit değerleri karşılaştırmak için; ***slti*** komutu kullanılır. Örneğin, $s2 registerinin içeriğinin 10 sayısıyla karşılaştırmak için;

```
slti    $t0, $s2,10     # eğer $s2 < 10 ise, $t0 = 1
```

MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, . . . , $s7` `$t0, $t1, . . . , $t7,` `$zero`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers `$s0`–`$s7` map to 16–23 and `$t0`–`$t7` map to 8–15. MIPS register `$zero` always equals 0.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

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
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Data from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Data from register to memory
Logical	and	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1 = ~ ($s2 |$s3)`	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $$s1,$s2,10`	`$s1 = $s2 >> 10`	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,L`	`if ($s1 == $s2) go to L`	Equal test and branch
	branch on not equal	`bne $s1,$s2,L`	`if ($s1 != $s2) go to L`	Not equal test and branch
	set on less than	`slt $s1,$s2,$s3`	`if ($s2 < $s3) $s1 = 1; else $s1 = 0`	Compare less than; used with `beq, bne`
	set on less than immediate	`slt $s1,$s2,100`	`if ($s2 < 100) $s1 = 1; else $s1 = 0`	Compare less than immediate; used with `beq, bne`
Unconditional jump	jump	`j L`	go to L	Jump to target address

FIGURE 2.12 MIPS architecture revealed through Section 2.6. Highlighted portions show MIPS structures introduced in Section 2.6.

MIPS Machine Language (ex. in decimal)

Name	Format	Example						Comments
Add	R	0	2	3	1	0	32	add $s1, $s2, $s3
Sub	R	0	2	3	1	0	34	sub $s1, $s2, $s3
Addi	I	8	2	1	100			addi $s1, $s2, 100
Lw	I	35	2	1	100			lw $s1, 100($s2)
Sw	I	43	2	1	100			sw $s1, 100($s2)
Beq	I	4	1	2	100			beq $s1, $s2, 100
Bne	I	5	1	2	100			bne $s1, $s2, 100
Slt	R	0	2	3	1	0	42	slt $s1, $s2, $s3
Slti	I	10	2	1	100			slti $s1, $s2, 100
J	J	2	10000					j 10000
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
Format R	R	op	rs	rt	rd	Shamt	funct	Arithmetic Instruction format
Format I	I	op	rs	rt	Address / immediate			Transfer, branch, Imm. Format
Format J	J	op	Target address					Jump instruction format

Registerleri tekrar hatırlayalım !!!!!!!
Bunlar bi-li-ne-cek

- The 32 MIPS registerleri guruplara göre:

  - Register 0 : $zero devamlı olarak 0 değeri ile yüklüdür.
  - Regs 2-3 : $v0, $v1 bir prosedürün döndürdüğü değer yüklüdür.
  - Regs 4-7 : $a0-$a3 prosedüre giriş argümanları yüklüdür.
  - Regs 8-15 : $t0-$t7 geçici değerler
  - Regs 16-23: $s0-$s7 Değişkenler için
  - Regs 24-25: $t8-$t9 daha fazla geçici değerler
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra Dönüş adresi yüklüdür.

Soru Aşağıda; bir programın assembler kodları verilmiştir. Bu assemler programı şekildeki data hafızasıyla etkileşiyor. Kodların karşılıklarını karşılarına yazınız ve data hafızasındaki datalara göre işlemin sonucu elde ediniz.? Sonuç data hafızasındaki hangi adrese yazılır?

Data hafızası

adres	data
100	AB101091
104	0023FF97
108	1000345A
112	DC5678FA
116	111151CE
120	FFFFCCDE
124	6784FBCE
128	889966E1
132	0001F9C5

```
addi  $t0, $zero, 100
addi  $t1  $t0, 4
lw    $s0  0 ($t1 )
lw    $s1  8 ($t0)
add   $s1  $s1, $s0
sw    $s1  28 ($t0)
```