



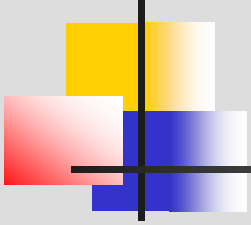
Registers ve MAL

- SAL gercek bir assembly dilinin cogu ozelligini icerir. Ancak gercek bir assembly dili degildir.
- SAL MAL in bir supersetidir (SAL MAL in tum instructionlarini icerir)
- MAL
 - MIPS RISC mimarisinin assembly dilidir.
- SAL, MAL tarafından desteklenmeyen bazi abstractionlari icerir.



MAL

- Instruction larin formatlari mimarinin performansina etkide bulunur
- Assembly dilleri memory e dogrudan erismeyi saglayan instructionlar icerirler
- Assembly dilinde data type kavrami yoktur. Bellek gozlerindeki bitlerin sabit bir anlami yoktur. Bu bitlerin anlami instructionlarin bunlari nasil kullandigina baglidir. Bellekteki bir word unsigned integer, floating point number veya character (bir kac) olarak yorumlanabilir. Hatta bu word bir instruction olarak da olabilir



Instructions

- Bir çok modern processor lerin (MIPS RISC mimarisi dahil) instruction lari fixed-size (tum instructionlarin boyu ayni) 32-bit dir.
- Instructionlarin fixed-size olmasi bir cok kolayliga yol acar
 - Instruction larin fetch edilmesi (memory den CPU ya getirilmesi)
 - Sonraki run olacak instruction in adresinin hesaplanmasi



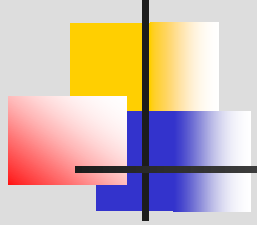
Load/store architecture

- Eger memory e sadece load/store instructionlariyle erisilebiliyorsa buna load/store architecture denir.
 - Modern RISC (Reduced Instruction Set Computer) processor ler load/store mimarisine sahiptirler
 - Komutlariin kısa ve sabit uzunlukta olmasina yol acar.



RISC Mimarisi

- 70 li yılların sonu ve seksenli yılların başında RISC projesi yürütülmüştür.
- Proje IBM, Stanford University ve UC-Berkeley tarafından ortaklaşa yürütülmüştür
- IBM 801 (1975), Stanford MIPS (Hennessy & Patterson), Berkeley RISC 1, 2, Alpha (DEC), Power PC (Apple+IBM+Motorola), ARM, SPARC
- Temel özellikleri
 - One cycle execution
 - Pipelining
 - Large number of registers



RISC vs. CISC



MIPS

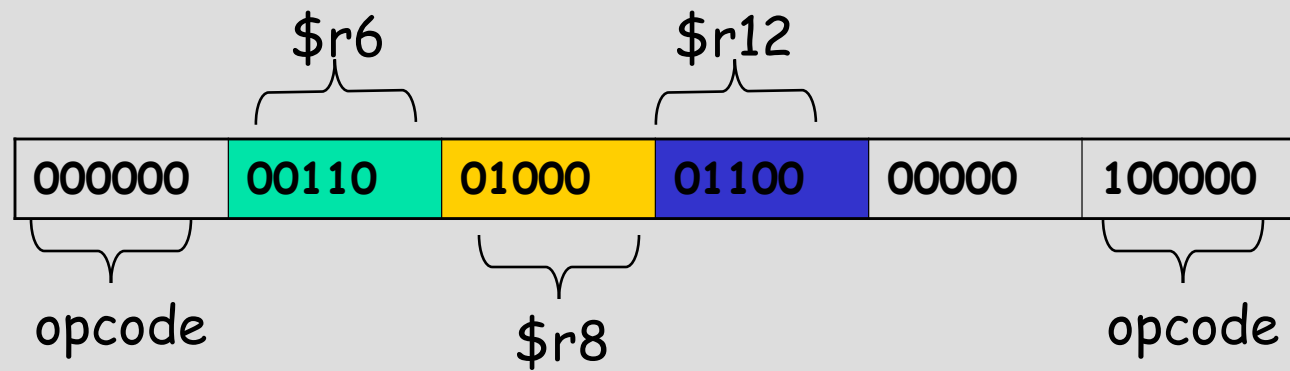
MIPS mimarisi yaklaşık 111 instructiondan oluşur

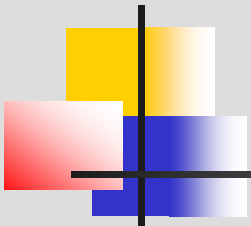
- 21 arithmetic instructions (+, -, *, /, %)
- 8 logic instructions (&, |, ~)
- 8 bit manipulation instructions
- 12 comparison instructions (>, <, =, >=, <=, ¬)
- 25 branch/jump instructions
- 15 load instructions
- 10 store instructions
- 8 move instructions
- 4 miscellaneous instructions



MIPS

add \$r12, \$r6, \$r8





ADD -- Add

Description:	Adds two registers and stores the result in a register
Operation:	$\$d = \$s + \$t$; advance_pc (4);
Syntax:	add \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0000

AND -- Bitwise and

Description:	Bitwise ands two registers and stores the result in a register
Operation:	$\$d = \$s \& \$t$; advance_pc (4);
Syntax:	and \$d, \$s, \$t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0100



Registers

- MAL architecture iki farkli register file i vardır.
 - 32 genel register (\$r0 \$r31)
 - Her biri 32-bit liktir.
 - 32 farkli registerin kodlanmasi icin 5 bit lik bir alan gerekir (instruction code)
 - \$r0 daima 0 (sifir) icerir
 - \$r1 assembler tarafından kullanilir (makine diline translation sirasinda, pseudo instructions and large constants)
 - \$26 ve \$27 operating system tarafından kullanilirlar
 - \$28, \$29 ve \$r31 procedure cagriminda kullanilirlar (parameter passing, holding return address vs.)



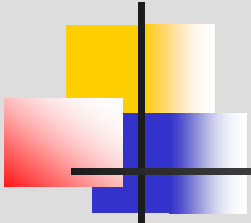
Registers

- 32 floating point registers
 - Floating point operand larini tutmak icin kullanilirler
 - \$f0,, \$f31
 - Floating point operands 32 (single precision)/64 (double precision) olabilir.
 - Double precision icin FP registerlar ciftler halinde kullanilabilirler
 - (\$f0,f1), (\$f16, \$f17)
 - Odd numbered (tek numarali) FP registerlar arithmetic ve branch instructionlarinda kullanilamazlar. Sadece data transferleri icin kullanilabilirler



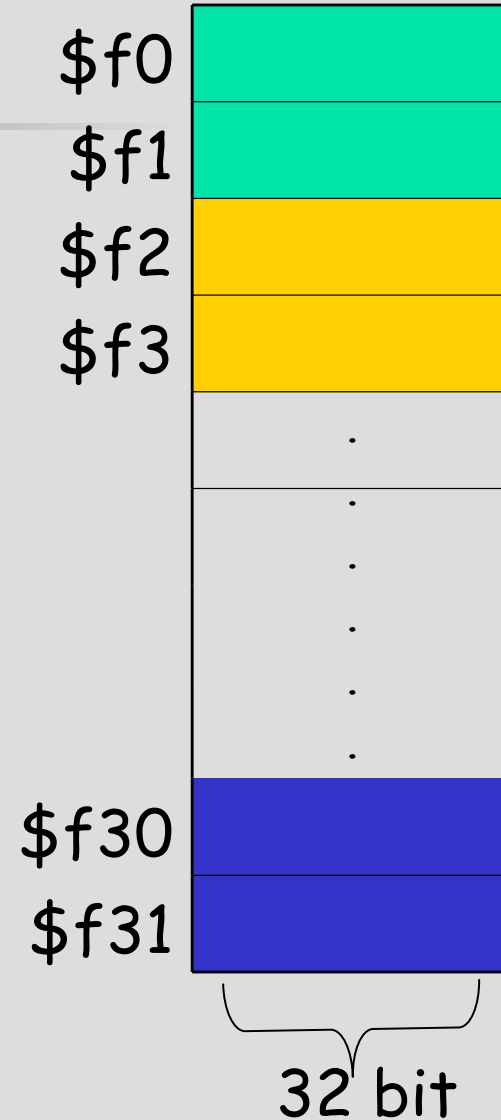
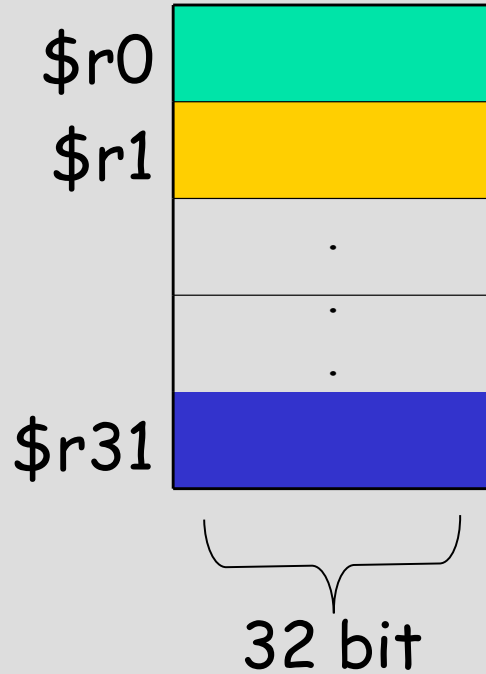
Registers

- Special purpose (özel amaçlı) registers
 - PC (Program Counter): Bir sonraki Execute edilecek instruction in adresini tutar.
 - SP (Stack Pointer)



Floating point register file

General purpose register file





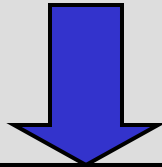
Load/Store Architecture

- Eger belleğe erişim sadece load/store instructionları ile sınırlı ise buna load/store architecture denir
- Bu mimarıda sadece iki grup instruction address belirlerler (load/store, control instructions)
- Veriler üzerinde işlemler yapmak için önce bu veri bellekten okunup (load) bir register'a yazılması gerekir. Registerlar üzerinde işlem yapıldıktan sonra belleğe tekrar yazılır (store)
- Butun RISC işlemciler load/store architecture'ine sahiptir.



SAL:

sub difference, subtrahend, minuend



MAL:

lw	\$8, subtrahend
lw	\$9, minuend
sub	\$10 , \$8, \$9
sw	\$10, difference



Instruction larda Adress Belirlenmesi

- Load/store architecture larda registerlar source (kaynak) operand lari temin eder (arithmetic ve logic instruction)
- Bu arithmetic ve logic instruction lari en fazla uc register kodlarlar (source1, source2, destination)
- Load/store instruction lari verinin bellekten okunacagi / yazilacagi adresi belirlemesi gerekir.
- Branch instruction lari da branch edilecek instruction in adresini belirlemek zorunda
- SPIM de address ler 32 bit oldugu icin load/store veya branch instruction in kendisinde kodlanamazlar (instructionlarin uzunluklari 32 bit)



Effective Address

- Load/store ve branch instructionlari nasil address belirler?
- Effective address hesabi: load/store ve branch instruction larinin ifade ettigi address in hesaplanmasi
- Bunun icin cesitli yontemler var



Addressing modes

- İşlemciler çeşitli yöntemlerle adres belirlerler. Bu yöntemlere addressing modes denir
- Addressing modes operand in address inin nasıl hesaplanması gerektiğini belirler



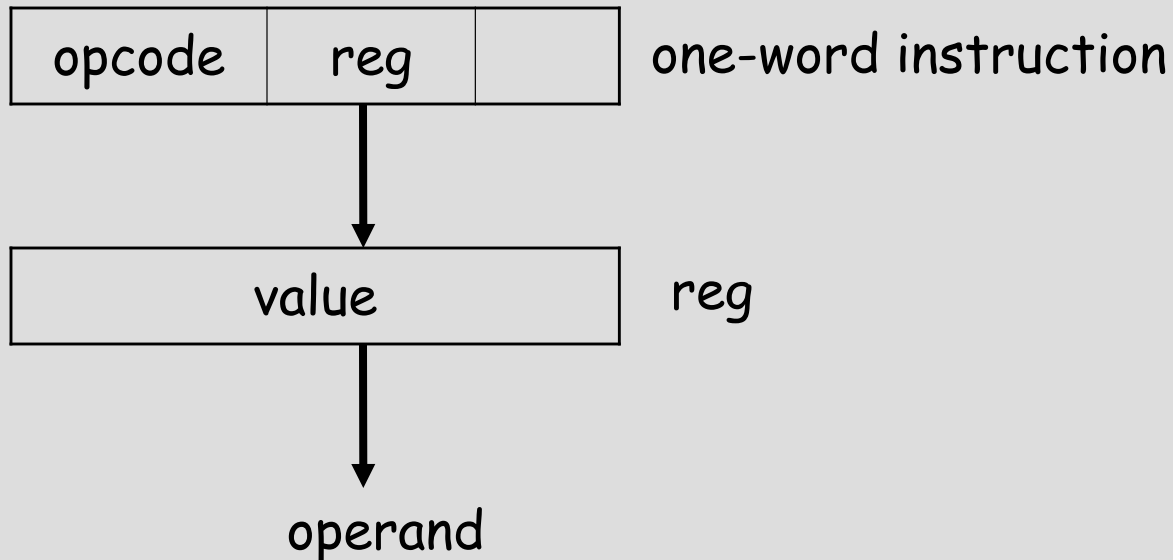
Immediate (İvedi) addressing

- Address direct olarak instructionin icine yerlestirilir.
- Eger instruction da address icin 8 bitlik bir alan ayrilmissa bu alan -128 .. +127 olarak yorumlanabilir.
- PC-relative



Register

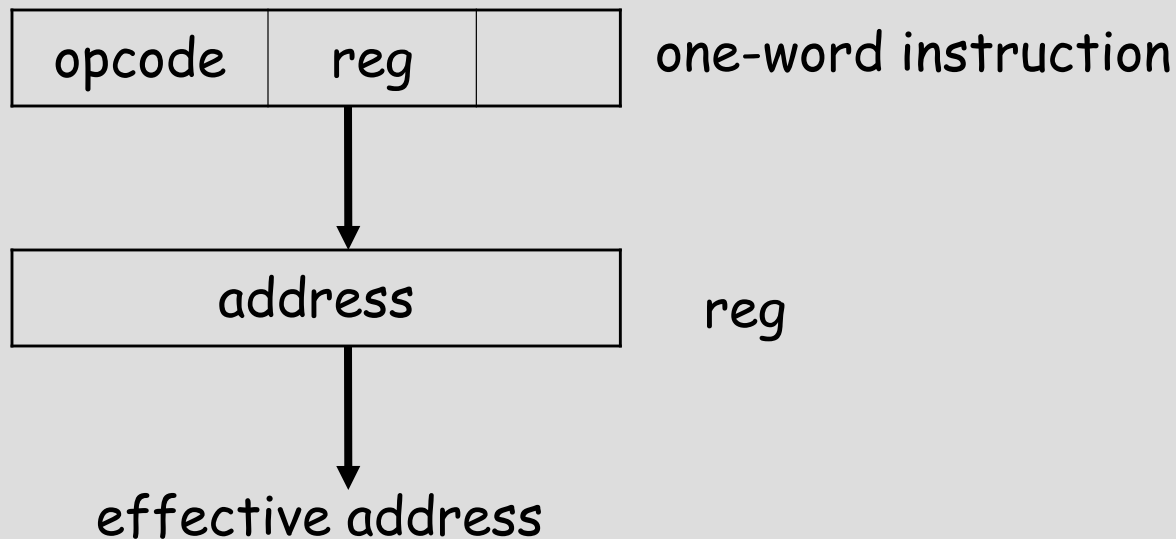
- Operand bir register da bulunur





Register Direct

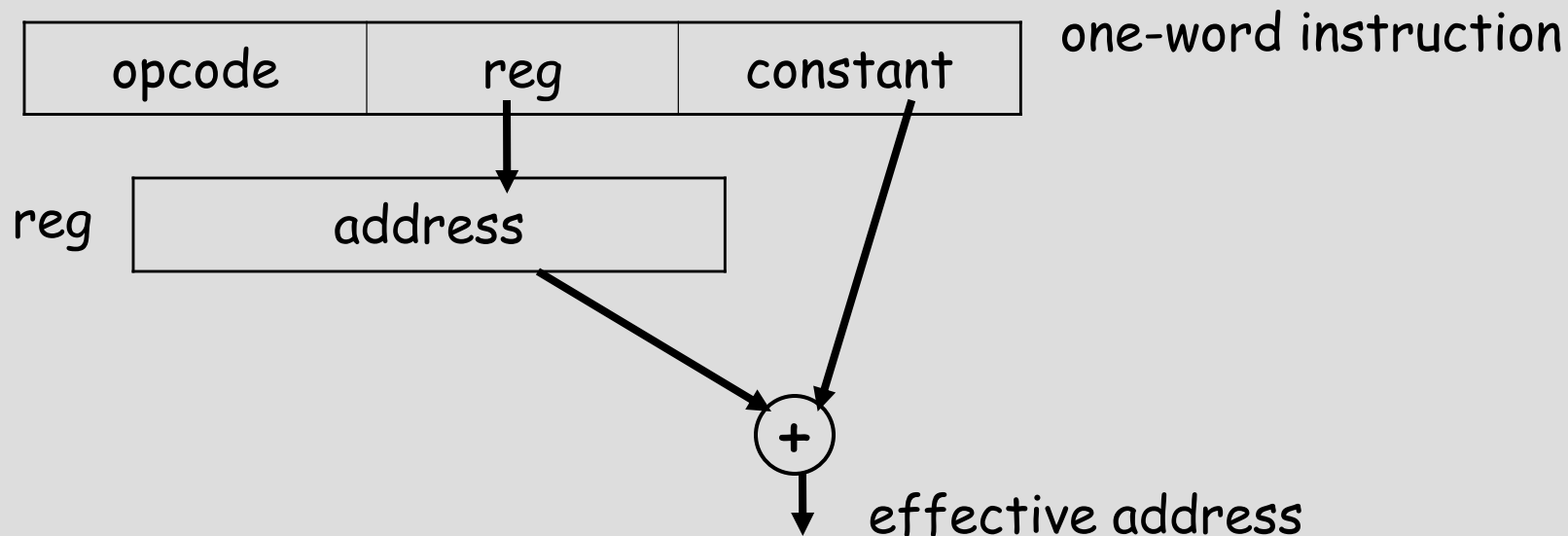
- Operand in address i bir register da bulunur



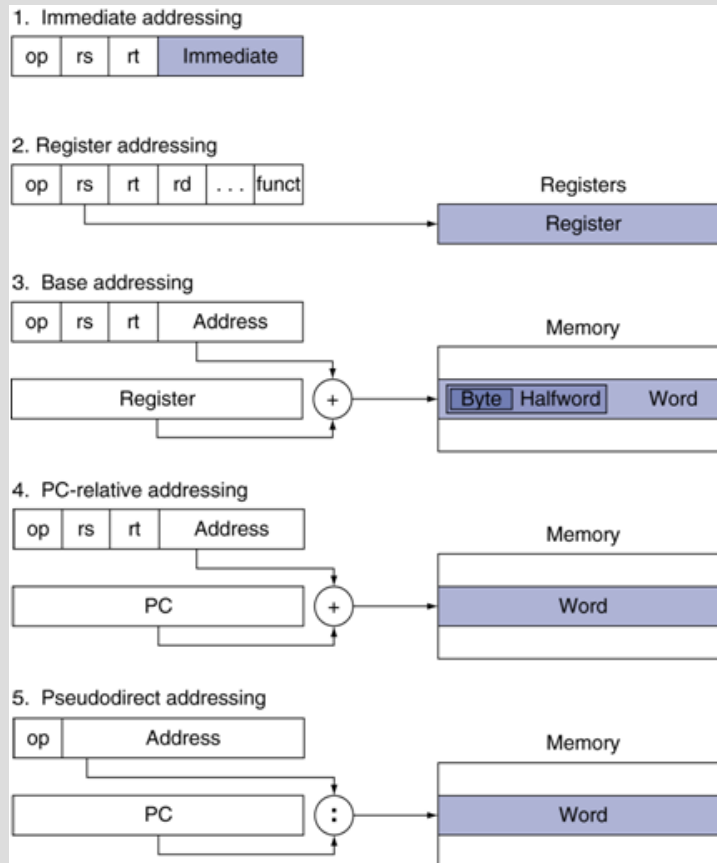


Base Displacement

- Effective address bir register in iceriginin (the base) instruction in icerisinde belirlenen bir constant (the displacement) a eklenmesiyle bulunur. Bu mode **indexed** veya **relative** olarak da bilinir.



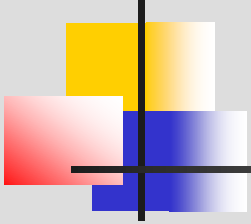
Adresleme Modları



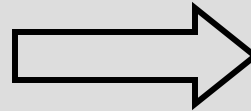


Control Instructions

- Load/store mimarilerde sadece load/store instructions ve control instructions address belirlerler
- Kontrol instructions kosul dogru oldugunda gidilecek instruction in address ini belirlemek zorunda
- Genelde target address ler bulunulan yerden kısa mesafeler olduğu için, cogu target address ler control instruction inin address ine küçük degerli bir offset eklenerek bulunur. (PC-relative addressing)



```
beq var1, var2, label  
.....  
.....  
.....  
label:
```



```
beq var1,var2,offset
```

Offset program assembled edilirken hesaplanır.



Load/Store Instructions

- `lw R, address`
 - En genel halde address bir constant, bir base register içerir.
 - Address, registerin belirlediği degere constant eklenmek suretiyle bulunur
 - Address 4 un kati olmak zorunda

`lw $22, 12($25)`

displacement base register

Target address: $12 + [\$25]$



Immediate addressing

li R, constant # $R \leftarrow \text{constant}$

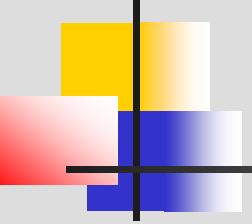
la R, label # label a bound olan variable addressini
R ye yazar

la \$8, var_name
lw \$8, (\$8)

lw \$8, var_name

la \$8, ar # load address of array ar
add \$8, \$8, 20 # get address of sixth element
lw \$8, (\$8) # load element ar[5]

.data
ar: .word 0:50
.text



```
# $8    -- a flag, 1 if the algorithm is done
# $9    -- an offset to the correct element of the array
# $10   -- address of the element to compare
# $11   -- the array element for comparison
# $12   -- the neighbor of the array element for comparison
# $14   -- base address of array ar

la      $14, ar
loop:   li      $8, 1                # flag = true
        li      $9, 0
for:    add     $10, $14, $9
        lw      $11, ($10)          # load element
        lw      $12, 4($10)         # load next element
        sub     $13, $11, $12
        blez    $13, noswap         # if they are in order, don't swap
        li      $8, 0
        sw      $11, 4($10)         # swap elements
        sw      $12, ($10)
noswap: add     $9, $9, 4
        sub     $13, $9, 196        # see if end of the array reached
        blz     $13, for
        beq     $8, $0, loop        # loop until done
done
```