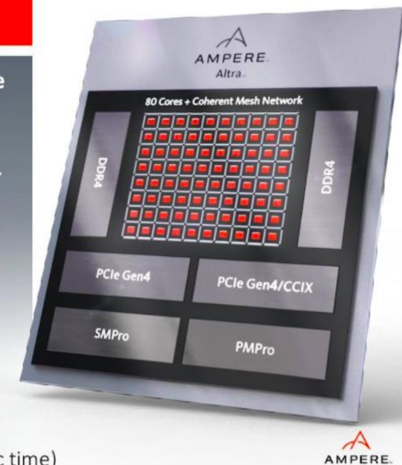
Εντολές
Γλώσσα Μηχανής
(Language of the Machine)

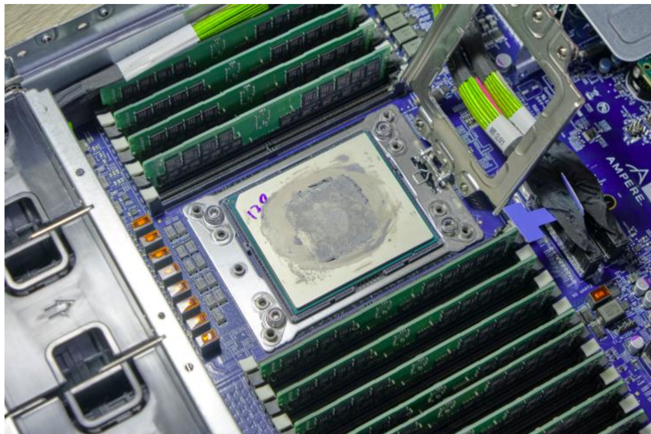
Σύγχρονα Υπολογιστικά Συστήματα

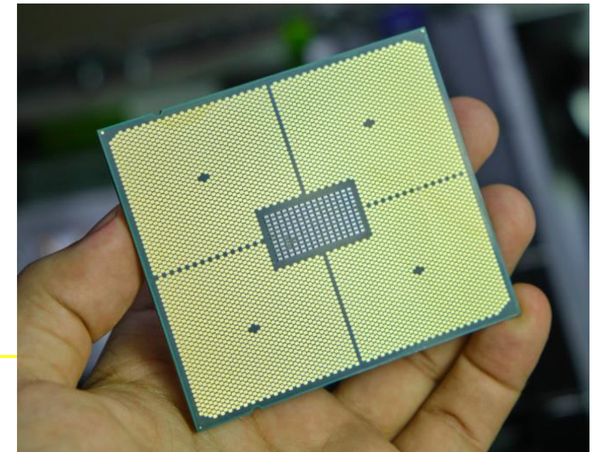
The Ampere™ Altra™ Processor

Predictable High Performance	High Scalability	Power Efficiency
• Up to 80 cores • Coherent mesh-based interconnect • High memory bandwidth and density	• Industry leading power/core • Cache-coherent multi-socket support • Flexible I/O connectivity	• Leading power/core • Advanced system, security, and power management • Monolithic die on leading 7nm process



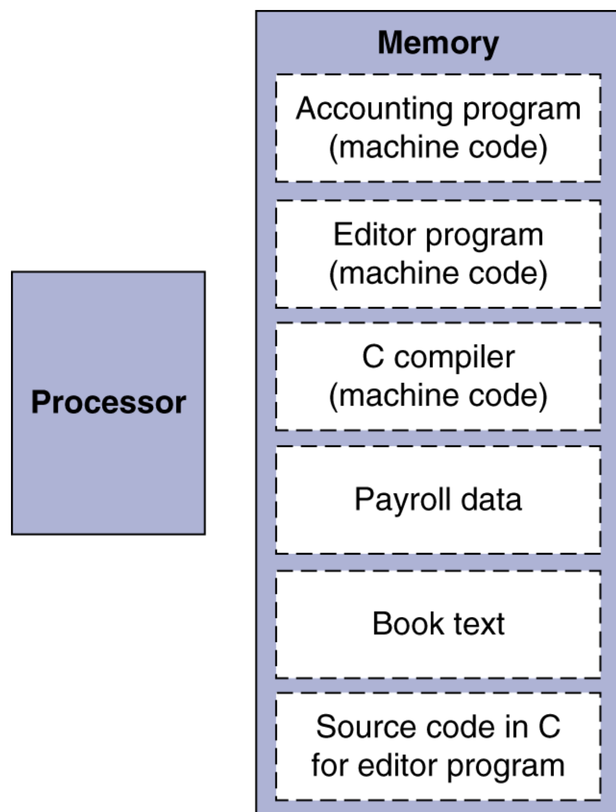
Embargo: March 3, 2020 (6:00 AM Pacific time)





Υπολογιστές με Πρόγραμμα

The BIG Picture



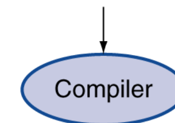
- Οι εντολές ενός προγράμματος αναπαριστάνονται στο δυαδικό, όπως και τα δεδομένα
- Εντολές και δεδομένα είναι αποθηκευμένα στην memory
- Προγράμματα λειτουργούν, ενεργούν σε προγράμματα
  - e.g., compilers, linkers, …
- Η συμβατότητα στο δυαδικό επιτρέπει στα compiled programs να λειτουργούν και σε άλλους υπολογιστές
  - Standardized ISAs

Επίπεδα Κώδικα Προγράμματος

- Γλώσσα υψηλού επιπέδου
  - Το επίπεδο αφαιρετικότητας είναι πιο κοντά στην περιοχή του προβλήματος
  - Υποστηρίζει παραγωγικότητα και μεταφερσιμότητα (portability)
- Γλώσσα Assembly
  - Αναπαράσταση με κείμενο των εντολών του επεξεργαστή
- Αναπαράσταση κώδικα για το Hardware
  - Ακολουθία δυαδικών ψηφίων (εντολές και δεδομένα)

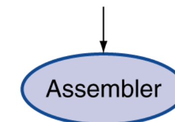
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

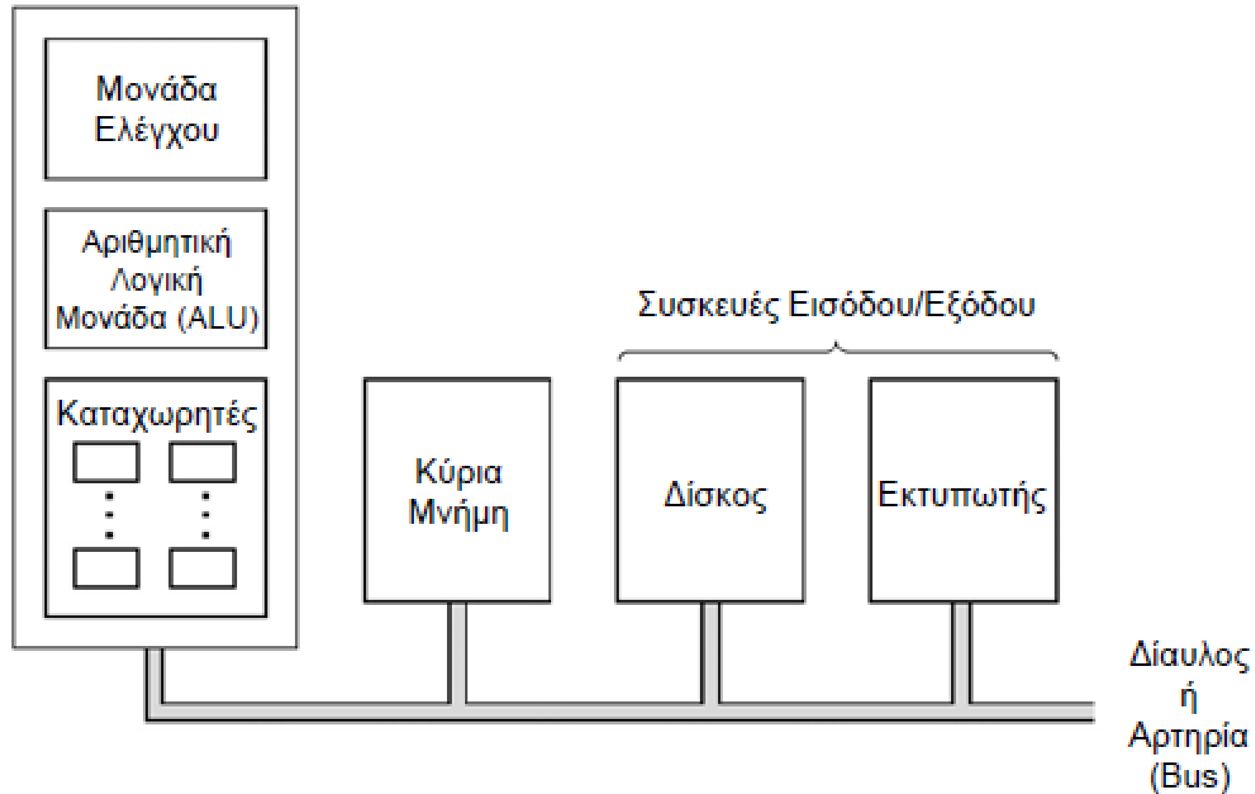
```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Οργάνωση Υπολογιστή



Τυπική οργάνωση υπολογιστή

Εκτέλεση προγραμμάτων σε υπολογιστή

1. Ο υπολογιστής εκτελεί προγράμματα που συνθέτονται από εντολές σε γλώσσα μηχανής.
2. Ένα πρόγραμμα προς εκτέλεση είναι αποθηκευμένο στην κύρια μνήμη σε συνεχόμενες θέσεις.
3. Η CPU έχει έναν καταχωρητή ειδικού σκοπού που μας λέει σε ποια θέση μνήμης βρίσκεται η επόμενη εντολή. Ονομάζεται *μετρητής προγράμματος* (*program counter – PC*).
4. Υπάρχει επίσης και ένας δεύτερος καταχωρητής ειδικού σκοπού στον οποίο κρατείται η τρέχουσα εντολή που εκτελείται. Ονομάζεται *καταχωρητής εντολής* (*instruction register – IR*).
5. Άλλοι πιθανοί καταχωρητές ειδικού σκοπού: α) *καταχωρητής δεδομένων* (*data register – DR*), β) *καταχωρητής διευθύνσεων* (*address register – AR*), γ) *συσσωρευτής* (*accumulator – AC*).

INSTRUCTION SET

- Οι λέξεις στη γλώσσα μηχανής ονομάζονται εντολές (instructions) και το λεξιλόγιο τους ονομάζεται ομάδα εντολών (instruction set).
- Στόχος: Η ομάδα εντολών του Η/Υ να κάνει πιο εύκολο το κτίσιμο υλικού και μεταγλωττιστή, ενώ ταυτόχρονα μεγιστοποιεί την απόδοση και ελαχιστοποιεί το κόστος.
- Στο μάθημα αυτό θα ασχοληθούμε με τη ομάδα εντολών του MIPS

MIPS INSTRUCTION SET

- Κάθε μηχανή πρέπει να είναι ικανή να κάνει αριθμητικές πράξεις.
  - π.χ.: **add a, b, c** a←b+c

- Κάθε αριθμητική εντολή στη γλώσσα MIPS πρέπει να έχει **τρεις μεταβλητές** (τελεσταίους).

Αρχή 1: Η ομοιότητα των λειτουργιών επιφέρει την απλότητα στο Hardware

MIPS INSTRUCTION SET

a = b + c + d + e

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

Μεταβλητές

Σχόλια

Εντολή

ΠΑΡΑΔΕΙΓΜΑΤΑ

C-like

a = b + c

d = a – e

Assembly

add a,b,c

sub d,a,e

C-like

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

Assembly

add t0,g,h

add t1,i,j

sub f,t0,t1

Καταχωρητές

- Υπάρχει περιορισμός στις μεταβλητές που υπάρχουν στις συμβολικές γλώσσες
- Οι μεταβλητές αντιστοιχούν σε καταχωρητές (registers)
- Ο MIPS έχει 32 καταχωρητές ($0,$1,..,$31)
- Η αποτελεσματική χρήση των καταχωρητών είναι το κλειδί στην απόδοση των προγραμμάτων.

Αρχή 2: Το μικρότερο είναι γρηγορότερο

Κύρια Μνήμη: εκατομμύρια θέσεις...

Καταχωρητές (2)

- Οι αριθμητικές πράξεις στο MIPS μπορούν να γίνουν μόνο σε καταχωρητές

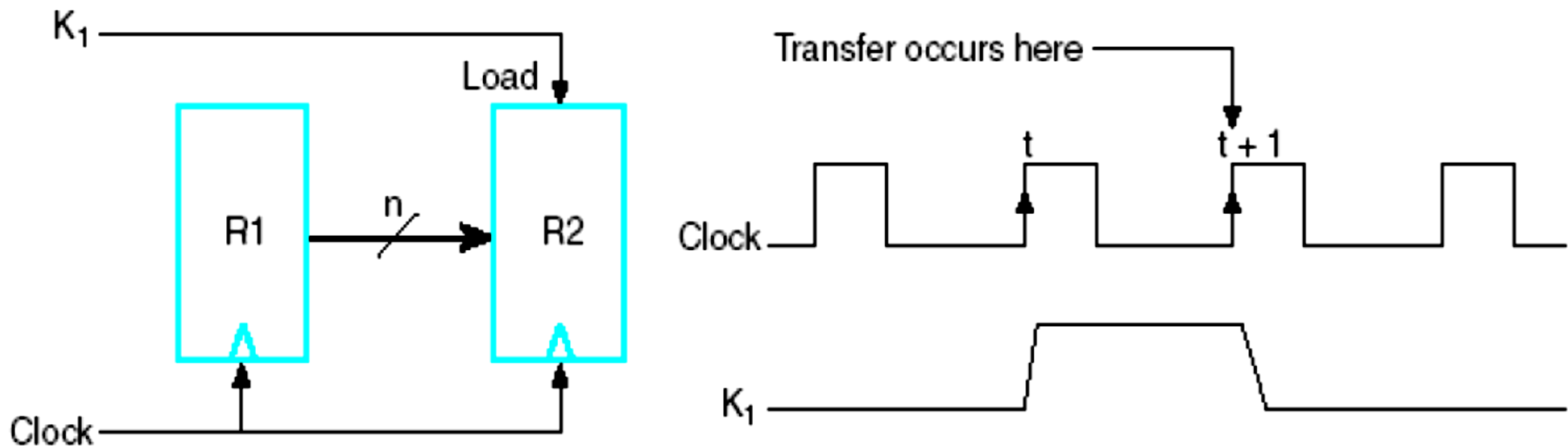
f=(g+h)-(i+j)

```
add $8,$17,$18
add $9,$19,$20
sub $16,$8,$9
```

Λειτουργίες σε Καταχωρητές

Μπλοκ διάγραμμα 8-bit καταχωρητή:

Διάβασμα, αποθήκευση σε καταχωρητή: R2←R1



Τελεστές στους Καταχωρητές

- Οι αριθμητικές και λογικές εντολές χρησιμοποιούν ως τελεστές τους καταχωρητές
- Ο MIPS έχει ένα αρχείο καταχωρητών: 32 × 32-bit register file
  - Χρησιμοποιείται για δεδομένα που προσπελαύνονται συχνά
  - Αριθμείται από 0 έως 31
  - Δεδομένο (περιεχόμενο) των 32-bit ονομάζεται λέξη (word)
- Ονόματα καταχωρητών για τον Assembler
  - $t0, $t1, …, $t9 για τιμές προσωρινές (temporary values)
  - $s0, $s1, …, $s7 για μεταβλητές αποθήκευσης (saved variables)

Αρχείο Καταχωρητών MIPS

- $a0 – $a3: ορίσματα (καταχωρητές: 4 – 7)
- $v0, $v1: αποτελέσματα (καταχωρητές: 2, 3)
- $t0 – $t9: προσωρινοί (μπορεί να αλλαχθούν από τον callee)
- $s0 – $s7: για αποθήκευση (πρέπει να αποθηκευτούν/ αποκαταστασταθούν από τον callee)
- $gp: global pointer για δεδομένα static (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Παράδειγμα με Τελεστές Καταχωρητές

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

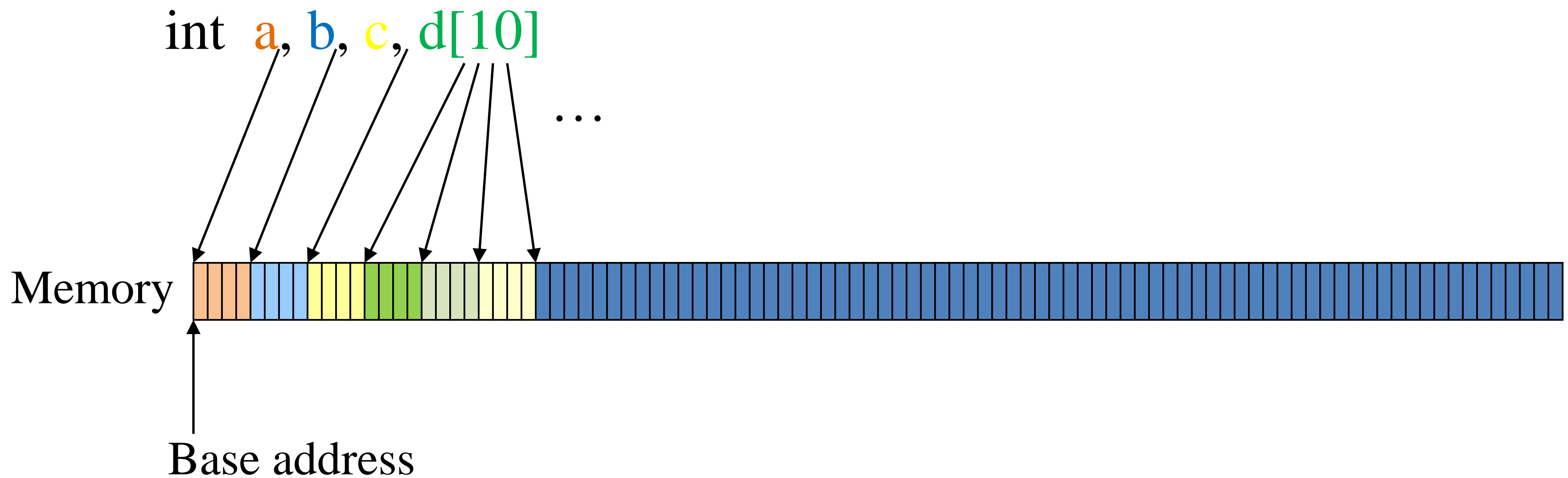
Περισσότερα Δεδομένα?

- Όλα τα δεδομένα ενός προγράμματος δεν μπορούν να αποθηκευτούν σε καταχωρητές – περιορισμένος αριθμός καταχωρητών
  - Οι καταχωρητές αποθηκεύουν τα τρέχοντα δεδομένα/μεταβλητές

- Οι μεταβλητές βρίσκονται πάντα αποθηκευμένες στην μνήμη και φορτώνονται σε καταχωρητές με ειδικές εντολές μεταφοράς δεδομένων:
  - Load word (lw),
  - Store word (sw)

Διεύθυνση μνήμης

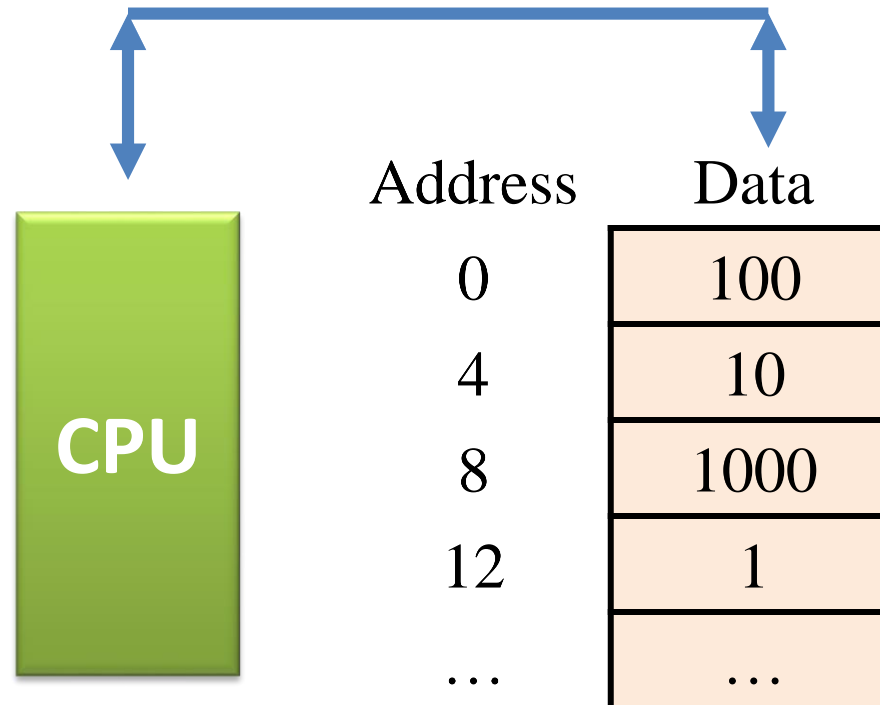
- Ο compiler οργανώνει τα δεδομένα στην μνήμη…
- Γνωρίζει την τοποθεσία κάθε μεταβλητής.



Δεδομένα στη Μνήμη

- Η κύρια μνήμη χρησιμοποιείται για συνήθεις τελεστές και σύνθετα δεδομένα:
  - Πίνακες, δομές, δυναμικά δεδομένα
- Για να εφαρμοστούν αριθμητικές λειτουργίες:
  - Φορτώνονται (Load) οι τιμές από τη μνήμη στους καταχωρητές
  - Αποθηκεύονται (Store) τα αποτελέσματα από τους καταχωρητές στη μνήμη
- Η κύρια μνήμη διευθυνσιοδοτείται ανά byte
  - Κάθε byte (8-bit) έχει μία διεύθυνση
- Οι λέξεις είναι ευθυγραμμισμένες στην κύρια μνήμη
  - Οι διευθύνσεις είναι πολλαπλάσιο του 4

Πρόσβαση στην Μνήμη

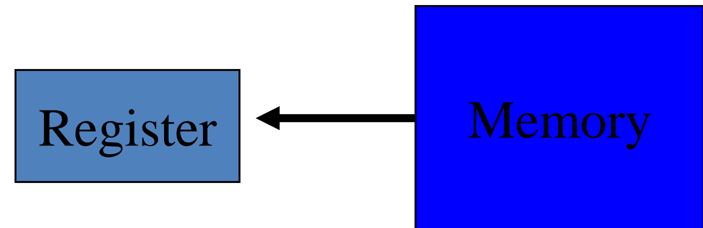


- Στον MIPS υπάρχουν εντολές για **load word** και **store word**, για 16 bit (half word) - lh/sh και 8bit - lb/sb δεδομένων
- sw: RAM <- Reg

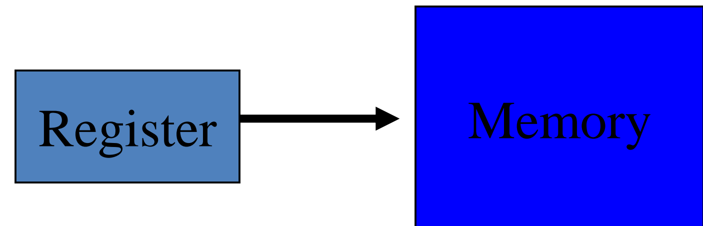
Τιμές στην Μνήμη

- Οι τιμές πρέπει να μεταφερθούν στους καταχωρητές πριν να εκτελεστούν οι εντολές (add and sub)

Load word
lw $t0, memory-address



Store word
sw $t0, memory-address



Πως υπολογίζεται η διεύθυνση μνήμης ??

Καταχωρητές vs. Μνήμη

- Οι καταχωρητές προσφέρουν ταχύτερη προσπέλαση σε σχέση με τη μνήμη
- Οι λειτουργίες στη μνήμη απαιτούν Loads και Stores
  - Πρέπει να εκτελεστούν περισσότερες εντολές
- Ο compiler πρέπει να χρησιμοποιεί τους καταχωρητές για τις μεταβλητές όσο περισσότερο γίνεται
  - Χρήση την μνήμης για τις μεταβλητές που χρησιμοποιούνται λιγότερο συχνά

Τελεστές Απ'ευθείας σε Εντολή

- Αριθμοί – σταθερές κατευθείαν στην εντολή (immediate: 0..2^16-1)

```
addi $s3, $s3, 4
```

- Δεν υπάρχει εντολή subtract immediate
  λύνεται ως εξής:

```
addi $s2, $s1, -1
```

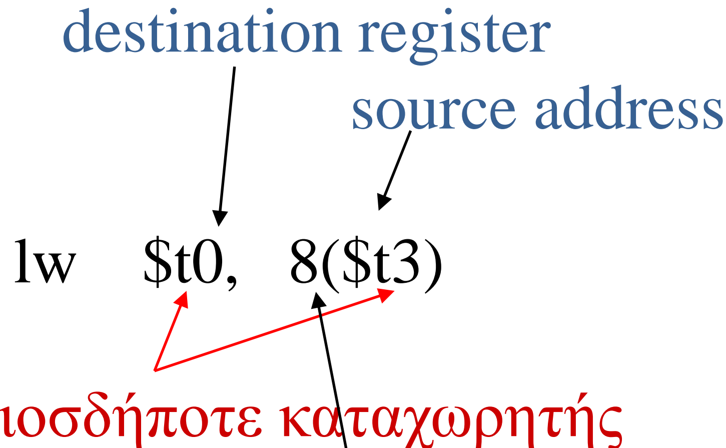
Design Principle 3:

Οι συνήθεις πράξεις πρέπει να είναι γρήγορες
Οι μικρές σταθερές (constants) είναι συνήθεις

Με τελεστές Immediate αποφεύγεται μια load instruction

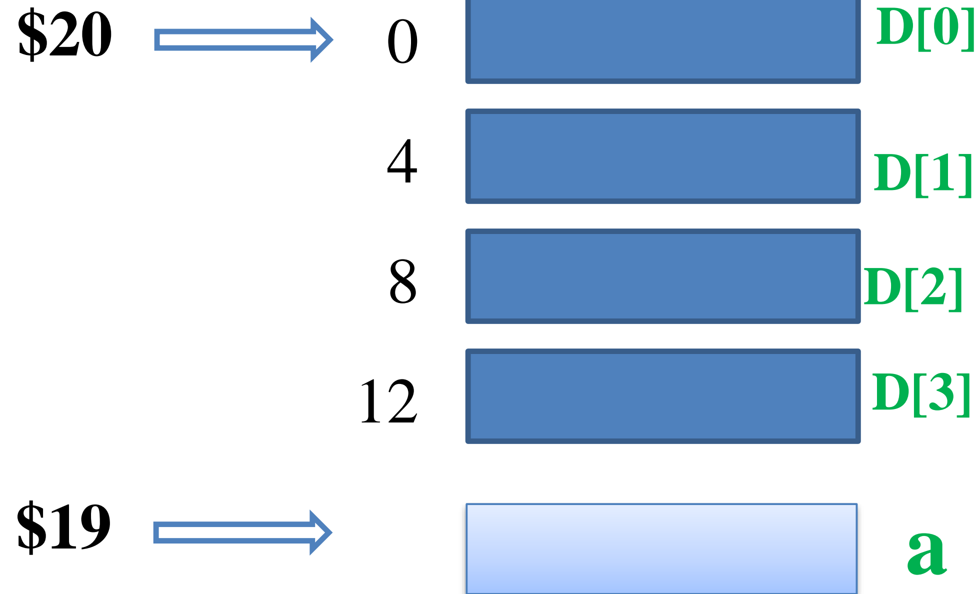
Μορφή των εντολών μεταφοράς δεδομένων από την μνήμη

- Η μορφή(format), συντακτικό μιας εντολής **load** :



Ένας σταθερός αριθμός που προστίθεται στην τιμή του καταχωρητή στην παρένθεση

Παράδειγμα



Μετατροπή σε assembly:

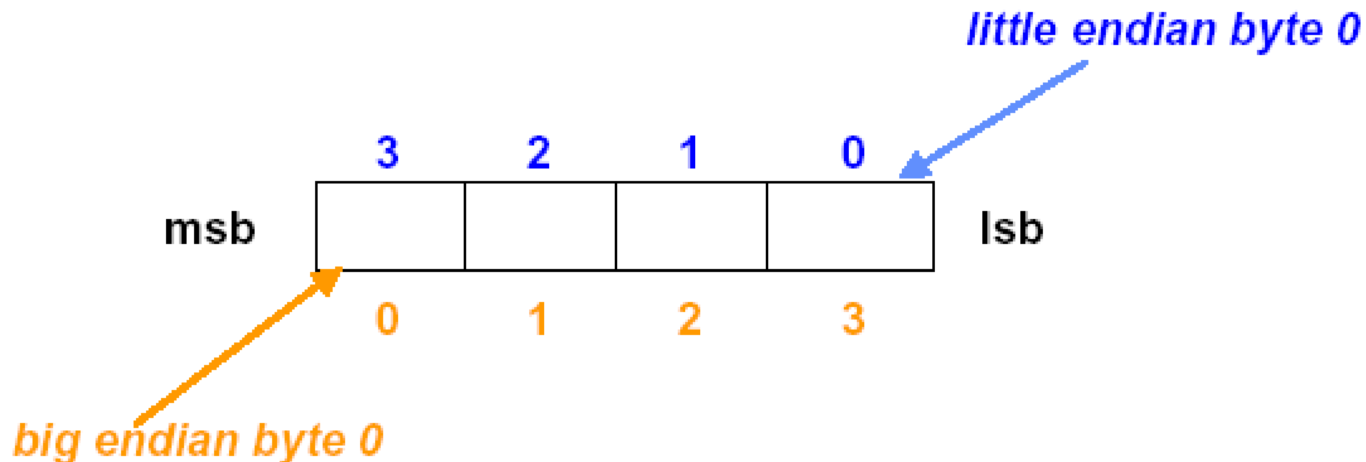
C code: d[3] = d[2] + a;

Assembly:

```
la $20, d # 3000
la $19, a
lw    $7, 8($20)    #  d[2] is brought into $7
lw    $9, 0($19)    #   a  is brought into $9
add   $7, $7, $9    #  the sum is in $7
sw    $7, 12($20)   #  $7 is stored into d[3]
```

ΔΙΑΤΑΞΗ ΔΕΔΟΜΕΝΩΝ

- Δεδομένα: Byte, Half word, word.
- Ο τρόπος που αποθηκεύονται τα δεδομένα στη μνήμη ονομάζεται **endianess**:
  - Big Endian
  - Litle Endian



ΠΑΡΑΔΕΙΓΜΑ ENDIANESS

1234567890 ή σε HEX: 499602D2

Little Endian

0100 1001	1001 0110	0000 0010	1101 0010
4 9	9 6	0 2	D 2
Byte 3	Byte 2	Byte 1	Byte 0

Big Endian

1101 0010	0000 0010	1001 0110	0100 1001
D 2	0 2	9 6	4 9
Byte 0	Byte 1	Byte 2	Byte 3

ΠΑΡΑΔΕΙΓΜΑ

C-like

```
A[i]=h+A[i]
```

Assembly

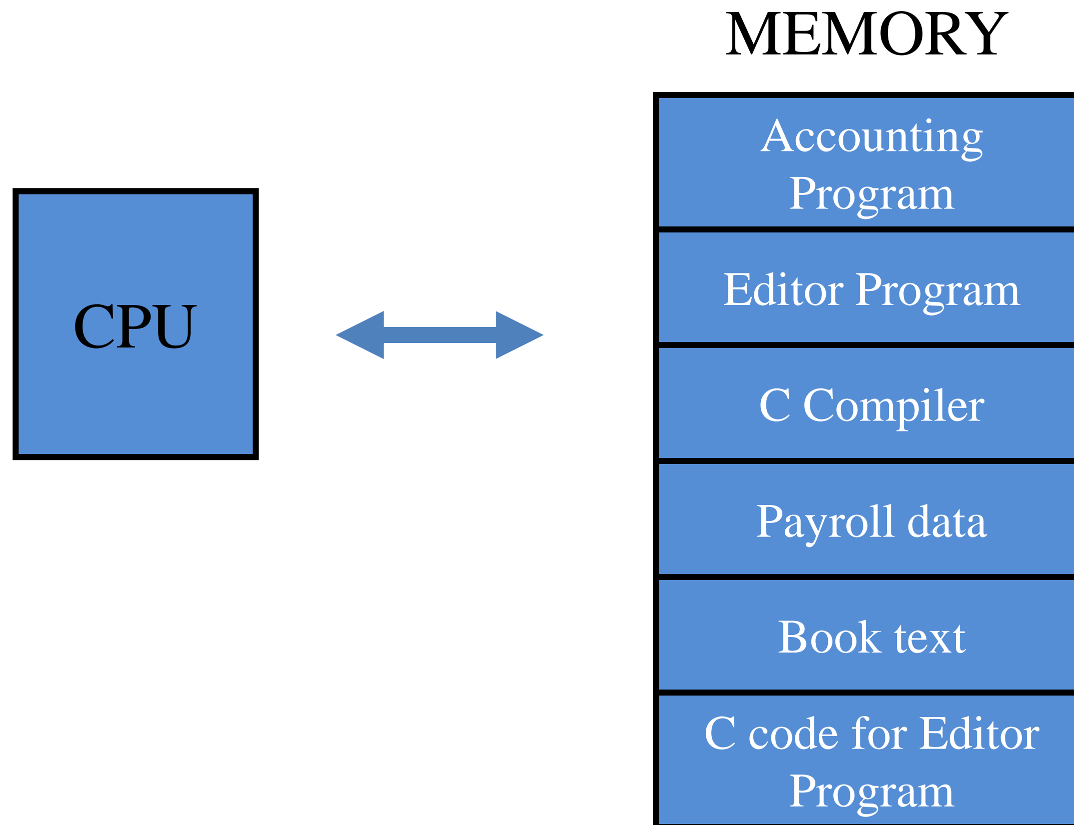
```
lw $9, i ($0)
lw $18, h ($0)
addi $14, $0, 4
mul $10, $9, $14
mflo $19
lw $8, A ($19)   # ο καταχωρητής $8 παίρνει προσωρινά την τιμή του A[i]
add $8, $18 , $8# ο καταχωρητής $8 παίρνει προσωρινά την τιμή του h+A[i]
sw $8, A ($19)   # Το h+A[i] φυλάγεται πίσω στη θέση A[i]
```

ΟΙ ΔΥΟ ΒΑΣΙΚΕΣ ΑΡΧΕΣ

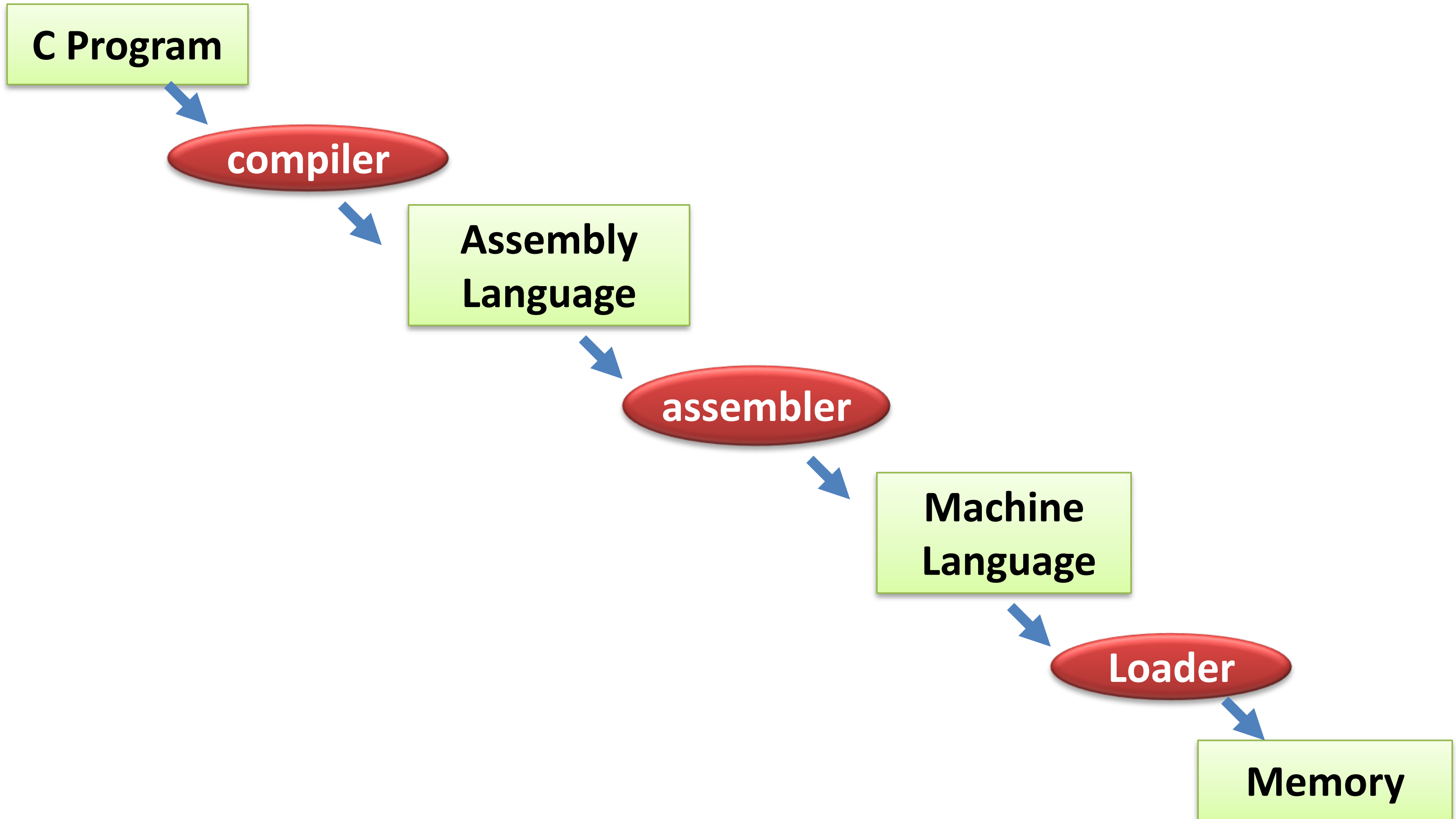
1. Οι εντολές απεικονίζονται ως αριθμοί.

2. Τα προγράμματα μπορούν να αποθηκευθούν στη μνήμη για να διαβαστούν, ή να γραφούν, όπως οι αριθμοί.
   Αυτή η αρχή ονομάζεται **“η έννοια του αποθηκευμένου προγράμματος”**
   (**Αποθηκευμένα Προγράμματα :** Φυλάγονται τα προγράμματα στη μνήμη σαν κώδικας σε γλώσσα μηχανής.

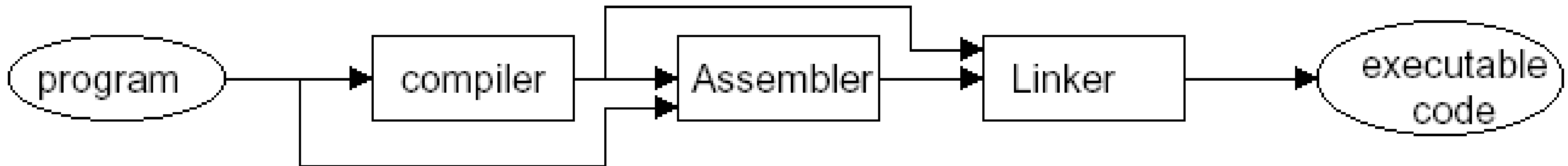
ΑΠΟΘΗΚΕΥΜΕΝΟ ΠΡΟΓΡΑΜΜΑ



ΙΕΡΑΡΧΙΑ ΜΕΤΑΦΡΑΣΗΣ



ΙΕΡΑΡΧΙΑ ΜΕΤΑΦΡΑΣΗΣ



ΙΔΙΑΙΤΕΡΟΤΗΤΕΣ MIPS

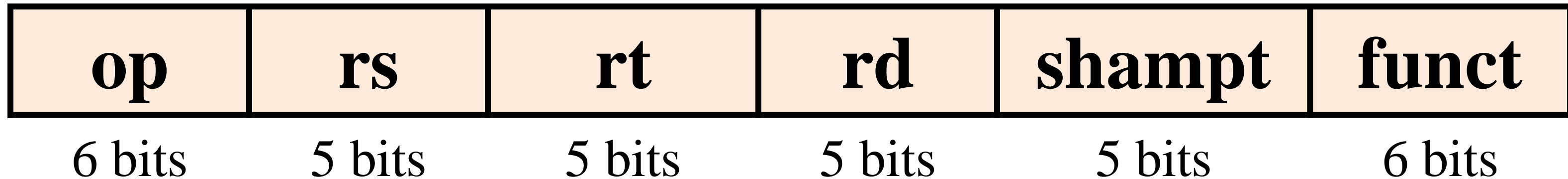
- Ο $0 στον MIPS έχει πάντα τιμή 0
- Ο χρήστης δεν μπορεί να αλλάξει την τιμή του $0
- Ο assembler «δημιουργεί» ψευδοεντολές

Π.χ.

```
Move $8,$18          #αντιγραφή του $18 στον $8
add $8,$0,$18
```

INSTRUCTION FORMAT

- R TYPE



- op: λειτουργία
- rs: πρώτος καταχωρητής εισόδου
- rt: δεύτερος καταχωρητής εισόδου
- rd: καταχωρητής αποτελέσματος
- shampt: ποσότητα μετακίνησης
- funct: συνάρτηση (function)

ΠΑΡΑΔΕΙΓΜΑ ΕΝΤΟΛΗΣ R-type

Add $8,$17,$18

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Αρχή 3: Η καλή σχεδίαση απαιτεί συμβιβασμούς.

Πράξεις Ολίσθησης (Shift)

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- shamt: πόσες θέσεις να γίνει ολίσθηση
- Shift lσeft logical
  - Αριστερή ολίσθηση και γέμισμα με 0 bits
  - `sll` by *i* bits: πολλαπλασιάζει by 2^i
- Shift right logical
  - Δεξιά ολίσθηση και γέμισμα με 0 bits
  - `srl` by *i* bits: διαίρεση by 2^i (unsigned only)

Αριστερή Ολίσθηση
Shift left logical (sll)

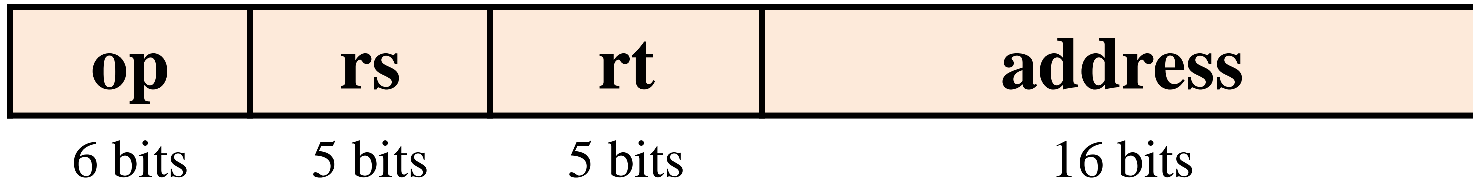
- sll $9,$8,5 μετακινεί τα bits αριστερά (lower to higher)
- Έστω ότι ο $8 περιέχει:

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010

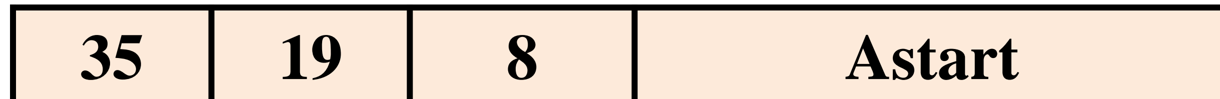
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0100 0000

INSTRUCTION FORMAT

- I TYPE



- Π.Χ. lw $8, Astart ($19)



ΠΑΡΑΔΕΙΓΜΑΤΑ ΕΝΤΟΛΩΝ

Instruction	Format	op	rs	rt	rd	Shamt	Funct	Address
ADD	R	0	REG	REG	REG	0	32	N.A.
SUB	R	0	REG	REG	REG	0	34	N.A.
LW	I	35	REG	REG	N.A.	N.A.	N.A.	Address
SW	I	43	REG	REG	N.A	N.A.	N.A.	Address

ΠΑΡΑΔΕΙΓΜΑ ΕΝΤΟΛΩΝ

```
lw $8, Astart ($19)
add $8, $18 , $8
sw $8, Astart ($19)
```

Έστω ότι η διεύθυνση Astart είναι η 1200

<table>
<tr><td>35</td><td>19</td><td>8</td><td colspan="3">1200</td></tr>
<tr><td>0</td><td>18</td><td>8</td><td>8</td><td>0</td><td>32</td></tr>
<tr><td>43</td><td>19</td><td>8</td><td colspan="3">1200</td></tr>
<tr><td>6bit</td><td>5bit</td><td>5bit</td><td>5bit</td><td>5bit</td><td>6bit</td></tr>
</table>

<table>
<tr><td>100011</td><td>10011</td><td>01000</td><td colspan="3">0000 0100 1011 0000</td></tr>
<tr><td>000000</td><td>10010</td><td>01000</td><td>01000</td><td>00000</td><td>100000</td></tr>
<tr><td>101011</td><td>10011</td><td>01000</td><td colspan="3">0000 0100 1011 0000</td></tr>
</table>

ΕΝΤΟΛΕΣ ΛΗΨΗΣ ΑΠΟΦΑΣΗΣ

Conditional Branches

- beq: Branch Equal

  beq $1,$2,L1

- bne: Branch not equal

  bne $1,$2,L1

- Γιατί δεν υπάρχουν `blt`, `bge`, κλπ?
- Hardware για <, ≥, … πιο αργό σε σχέση με =, ≠

ΕΝΤΟΛΕΣ ΛΗΨΗΣ ΑΠΟΦΑΣΗΣ

C language :

```
        if ( i = = j )  goto  L1;
        f = g + h;
    L1: f = f - i;
```

Αν f,g,h,i,j αποθηκεύομαι στους $16,S17,S18,S19,$20

```
     beq   $19, $20, L1          # go to L1  αν  i = j
     add  $16, $17, $18          # f = g + h (αγνοείται αν i = j)
L1:  sub $16, $16, $19           # f = f - i  εκτελείται πάντοτε
```

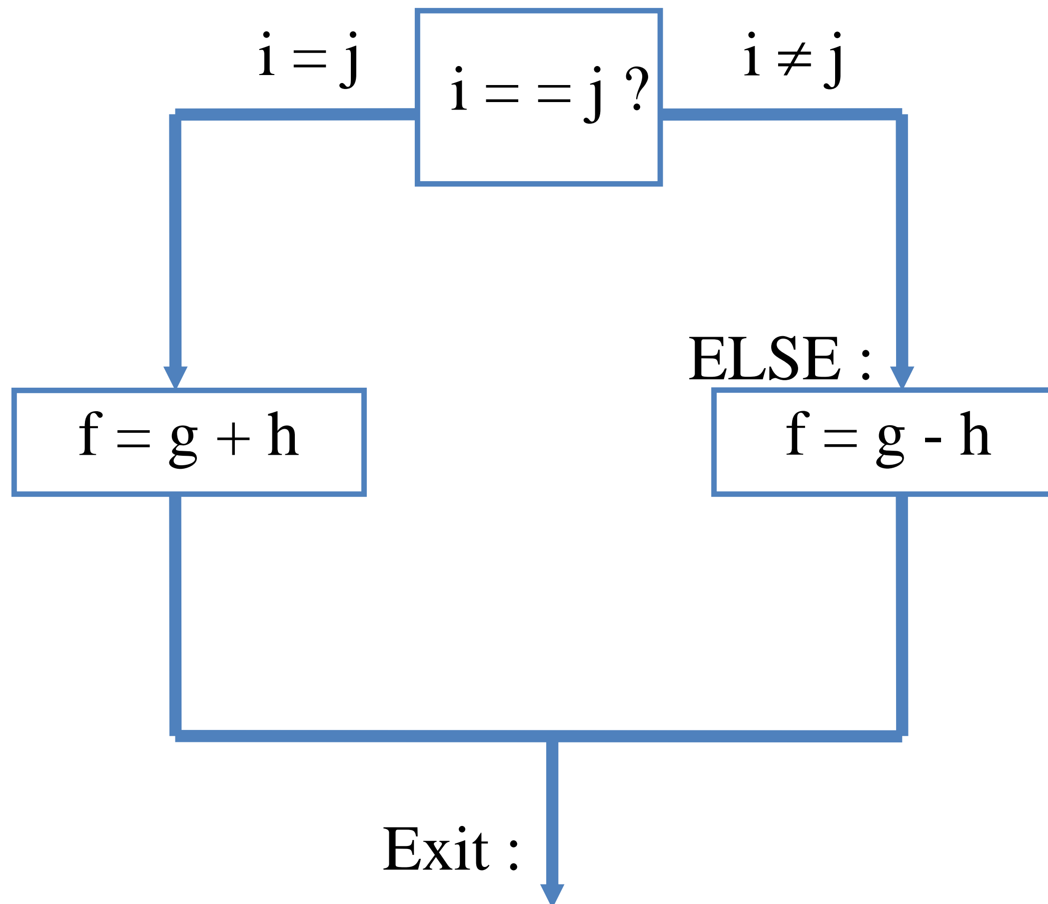
ΔΙΑΣΥΝΔΕΣΗ SW-HW

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

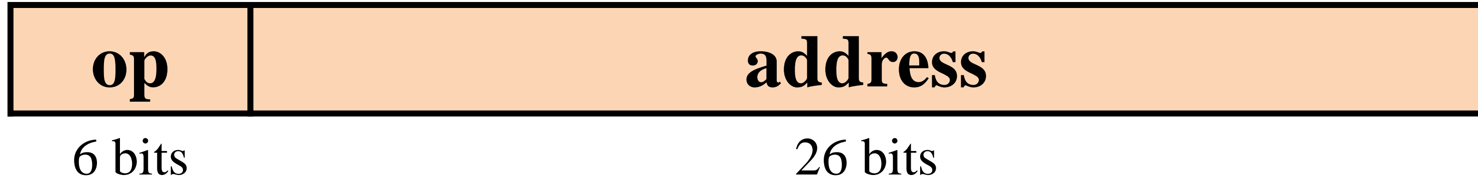
μεταγλωττίζεται στον πιο κάτω κώδικα MIPS:

```
        bne    $19, $20, Else          #  goto   Else  αν i ≠ j
        add   $16, $17, $18            #  f = g + h ( αγνοείται αν   i ≠ j )
        j  Exit                        #  go to Exit
Else :  sub $16, $17, $18              #   f = g - h  ( αγνοείται αν   i = j )
Exit :
```

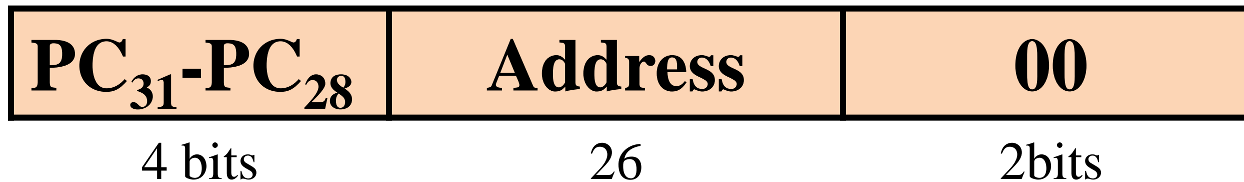
ΔΙΑΣΥΝΔΕΣΗ SW-HW



J Format



Ψευδο-απευθείας διευθυνσιοδότηση (pseudo direct)



PC: **0110**00000000000001100001001101000

J 100000

Bin: 00000000001100000110101000000

0110 00000000001100000110101000000 **00**

Παράδειγμα Εντολών j, b

`Loop: sll $t1, $s3, 2`	80000	0	0	19	9	4	0
`add $t1, $t1, $s6`	80004	0	9	22	9	0	32
`lw $t0, 0($t1)`	80008	35	9	8	0		
`bne $t0, $s5, Exit`	80012	5	8	21	2		
`addi $s3, $s3, 1`	80016	8	19	19	1		
`j Loop`	80020	2	20000				
`Exit: …`	80024						

ΠΑΡΑΔΕΙΓΜΑ

```
Loop :  g = g + A[i]
        i = i + j;
if (i != h) goto Loop;
```

Ο πίνακας A έχει 100 στοιχεία και ξεκινά από το Astart.
Οι μεταβλητές g, h, i, και j αποθηκεύονται στους $17, $18, $19 και $20.

```
Loop :    mult  $9, $19, $10       # Προσωρινός καταχωρητής $9 = i x 4,
                                   # υπολογισμός δείκτη
          lw  $8, Astart ($9)      # Προσωρινός καταχωρητής $8 = A[i]
          add  $17, $17, $8        # g=g+A[i]
          add  $19, $19, $20       # i = i+j
          bne  $19, $18, Loop      # Πήγαινε στο Loop αν   i ≠ h
```

Ο δείκτης του πίνακα i, πρέπει να πολλαπλασιάζεται επί 4 ($10).

ΠΑΡΑΔΕΙΓΜΑ

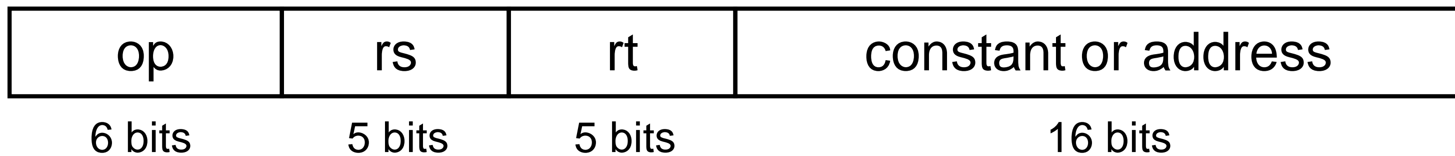
```
While (save[i]==k)
     i=i+1;
```

Οι μεταβλητές i, και k αποθηκεύονται στους \$19, \$21 και ο \$10=4, \$20=1

```
Loop: mul  $9,$19,$10        # Temporary reg $9 = i x 4
      lw   $8,Sstart($9)     # Temporary reg $8 = save[i]
      bne  $8,$21,Exit       # goto Exit if save[i] ≠  k
      add  $19,$19,$20       # i = i +1
      j Loop                 #goto Loop
Exit:
```

Εντολές Branch

- Οι περισσότες ετικέτες (branch targets) είναι «κοντινές» διακλαδώσεις
  - Forward or backward



- PC-relative addressing
  - Target address = PC + offset × 4
  - PC είναι ήδη αυξημένος κατά 4

Μακρινή Διακλάδωση

- Αν ο στόχος του branch είναι πολύ μακρυά (κωδικοποίηση σε 16-bit), assembler διαμορφώνει τον κώδικα...
- Παράδειγμα

```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
    L2: …
```

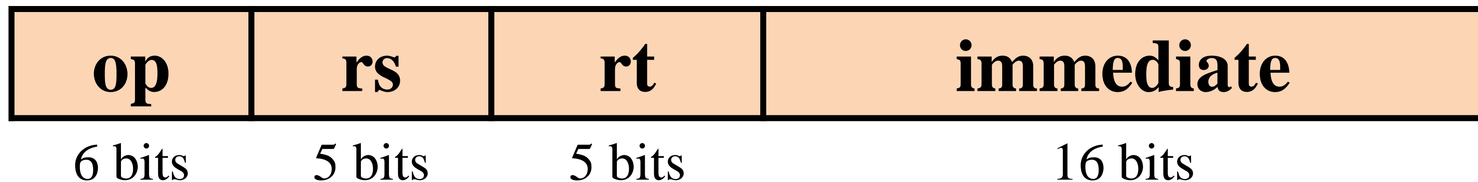
IMMEDIATE

- Στα προγράμματα υπάρχει μεγάλος αριθμός πράξεων με σταθερό τελεσταίο
- Η ύπαρξη I-type εντολών βελτιώνει την ταχύτητα

Αρχή 4: Τις πιο συχνές περιπτώσεις τις σχεδιάζουμε έτσι ώστε να εκτελούνται γρήγορα

IMMEDIATE

- $29+4
  - lw $24, AddrConstant4 ($0)
  - add $29, $29, $24
- Immediate type (I-Type)
  - addi $29, $29, 4



LUI

- lui reg,data

Load upper immediate

lui $8,255

001111	00000	01000	0000 0000 1111 1111

Αποτέλεσμα: περιεχόμενο $8

0000 0000 1111 1111	0000 0000 0000 0000

ΠΑΡΑΔΕΙΓΜΑ LUI

$16 ← 4000000

$16 ← 0000 0000 0011 1101 0000 1001 0000 0000

lui $16, 61 # 61 <d> = 0000 0000 0011 1101 <b>

0000 0000 0011 1101	0000 0000 0000 0000

addi $16, $16, 2304 # 2304 decimal = 0000 1001 0000 0000 in binary

0000 0000 0011 1101	0000 1001 0000 0000

$$4000000 = 61x2^{16}+2304$$

SET ON LESS THAN

- Slt reg1,reg2,reg3 # if reg2<reg3 then reg1=1
  # else reg1=0

  slt $1, $16, $17   # $1=1 if $16 < $17

  bne $1, $0, Less   # goto   Less if $1 ≠ $0

Οι δυο παραπάνω εντολές δημιουργούν την ψευδοεντολή **blt reg1,reg2**, label (**b**ranch **l**ess **t**han), η οποία μεταφράζεται όπως παραπάνω (δεν υποστηρίζεται από τη μηχανή)

Έλεγχος Ροής Εκτέλεσης προγράμματος

<u>Branches</u>

b target # unconditional branch to program label target
beq $t0,$t1,target # branch to target if $t0 = $t1
blt $t0,$t1,target # branch to target if $t0 < $t1
ble $t0,$t1,target # branch to target if $t0 <= $t1
bgt $t0,$t1,target # branch to target if $t0 > $t1
bge $t0,$t1,target # branch to target if $t0 >= $t1
bne $t0,$t1,target # branch to target if $t0 <> $t1

<u>Jumps</u>

j target # unconditional jump to program label target
jr $t3 # jump to address contained in $t3 ("jump register")

<u>Subroutine Calls</u>

subroutine call: "jump and link" instruction
jal sub_label # "jump and link"
copy program counter (return address) to register $ra (return address register)
jump to program statement at sub_label
subroutine return: "jump register" instruction
jr $ra # "jump register"
jump to return address in $ra (stored by jal instruction)

ΣΥΜΠΕΡΑΣΜΑ

Αρχή 2: Το μικρότερο είναι γρηγορότερο

- Η μηχανή RISC έχει μικρό αριθμό εντολών
- Σε επίπεδο assembly δημιουργούμε ψευδοεντολες.
- Οι ψευδοεντολές υλοποιούνται από τον Assembler με τη βοήθεια άλλων εντολών

RISC - Reduced Instruction Set Computer

Παράδειγμα (value[2]=value[0]+value[1];)

```
.text # text section
.globl main # call main by SPIM
main: la $10, value
  lw $8, 0($10)
  lw $9, 4($10)
  add $11, $8, $9
  sw $11, 8($10)
  j main
.data # data section
value: .word 10, 20, 0
```

```
Lw $8, value($0)
Addi $2, $0, 4
Lw $9, value($2)
add $11, $8, $9

Addi $2, $0, 8
sw $11, value($2)
J main
```

T (sec) = N * f * κύκλοιρολογιού_ανά_εντολή

Εντολές Ολίσθησης

- "Move" bits σε έναν καταχωρητή
- Left shift: μετακινεί bits από λιγότερη σημαντική θέση σε ψηλότερη
- Right shift: μετακινεί bits από ψηλότερη σημαντική θέση σε λιγότερη
- Δύο κατευθύνσεις: Left and Right
- Εντολές Shift
  - sll $t0,$t1,5 shift left logical
  - srl $t0,$t1,5 shift right logical
  - sra $t0,$t1,5 shift right arithmetic
  - sllv $t0,$t1,$t2 shift left logical value (register)
  - srlv $t0,$t1,$t2 shift right logical value (register)
  - srav $t0,$t1,$t2 shift right arithmetic value (register)

SLL

- sll $10,$11,5 μετακινεί bits από λιγότερη σημαντική θέση σε ψηλότερη (lower to higher) Έστω ότι ο $11 είναι:

- 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010
  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0100 0000

- logical shift: fill in 0s when value moved to the left by number of positions

SRL

- srl $10,$11,5 moves the bits right (higher to lower)

- Έστω οτι ο $11 είναι:
- 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
  0000 0101 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

$11	shamt	$10	
1000	1	0100	/2
1000	2	0010	/4
1000	3	0001	/8

shift right logical is division (integer) by 2^shamt

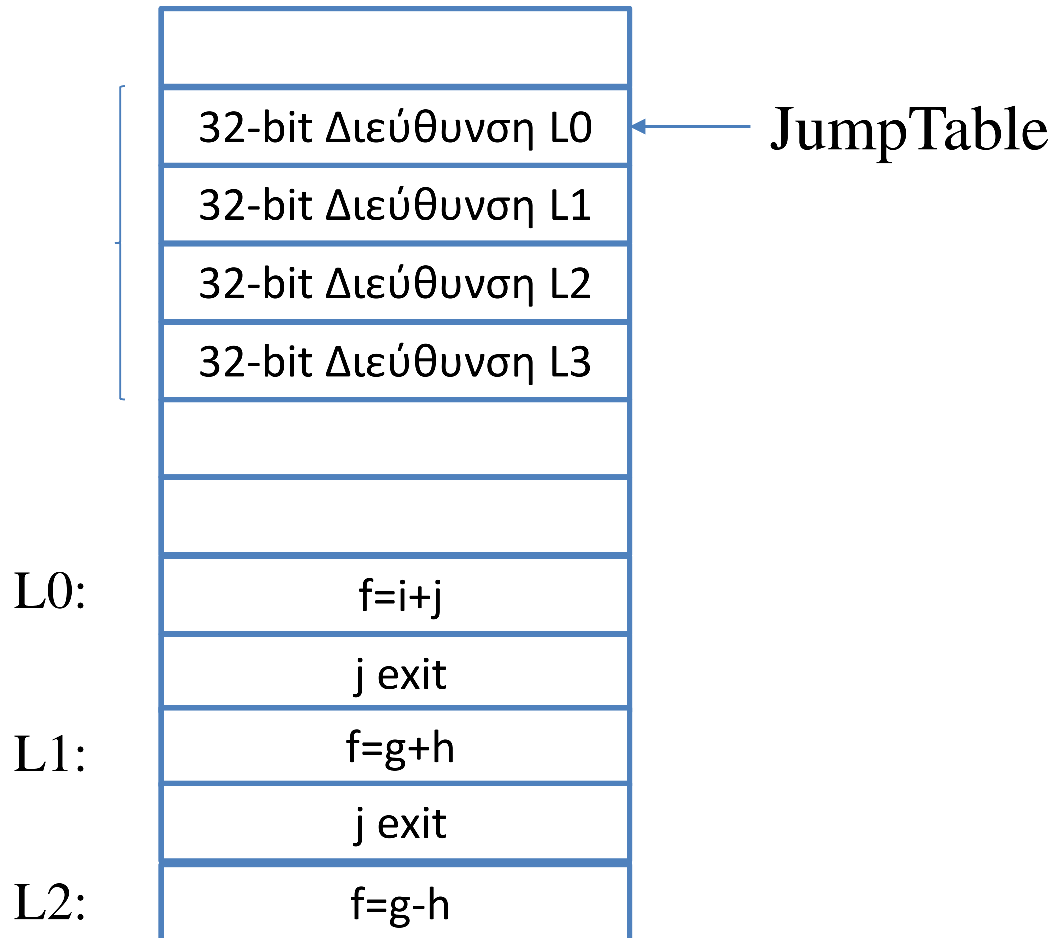
CASE

```
switch (k)
    {
        case 0:    f = i + j;    break;       /*  k = 0  */
        case 1:    f = g + h; break;          /*  k =1   */
        case 2:    f = g - h;  break;         /*   k =2  */
        case 3:    f = i - j;    break;       /*   k = 3 */
    }
```

f, g, h, i, j, k αποθηκεύονται στους καταχωρητές απο \$16 ως \$21

Ο καταχωρητής \$10 περιέχει τον αριθμό 4

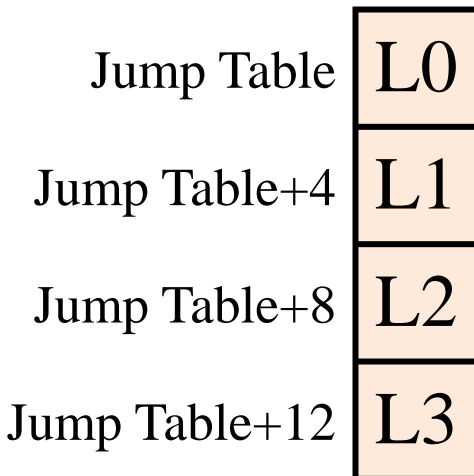
Οργάνωση Μνήμης



JumpTable

Κώδικας CASE

```
.data
JumpTable: .word L0, L1, L2, L3
```




```
Loop: mult $9, $10, $21          # $9 = k  x 4 ($10: 4, $21: k)
      lw  $8, JumpTable($9)      #  $8 = JumpTable[k]
      jr     $8                  #  Jump  on regr $8
L0:   add  $16, $19, $20         #  k = 0 so f gets i + j
      j      Exit                #  end of this case so goto Exit
L1:   add  $16, $17, $18         #  k = 1 so f gets g + h
      j      Exit                #  end of this case so goto Exit
L2:   sub $16, $17, $18          #  k = 2 so f gets g - h
      j      Exit                #  end of this case so goto Exit
L3:   sub $16, $19, $20          #  k = 3 so f gets i - j
Exit: add $1, $1, $0             # end of switch
    statement
```

CASE

7000	Loop:	mult $9, $10, $21	mult $9, $10, $21
7004		lw $8, 100($9)	lw $8, JumpTable($9)
7008		jr $8	jr $8
7012	L0:	add $16, $19, $20	add $16, $19, $20
7016		j 7040	j Exit
7020	L1:	add $16, $17, $18	add $16, $17, $18
7024		j 7040	j Exit
7028	L2:	sub $16, $17, $18	sub $16, $17, $18
7032		j 7040	j Exit
7036	L3:	sub $16, $19, $20	sub $16, $19, $20
7040	Exit:		

JumpTable

100+0x4	7012
100+1x4	7020
100+2x4	7028
100+3x4	7036

Special Purpose Registers

- PC: Program Counter
- RA: $31 Return address register
- Zero: $0 always zero
- SP: $29 Stack Pointer

Υπορουτίνες/Συναρτήσεις

- Ορισμοί
  - ***Caller:*** υπορουτίνα που κάνει την κλήση (main)
  - ***Callee:*** υπορουτίνα που καλείται (sum)

```
// High level code
void main() {
		int y;
		y = sum(42, 7);
		...
}

int sum(inta, intb) {
		return (a + b);
}
```

ΥΠΟΡΟΥΤΙΝΑ

jal address

Jump and Link:

$31 ← PC+4

PC ← address

Επιστροφή με jr $31

Αν έχω υπορουτίνα που καλεί υπορουτίνα υπάρχει η ανάγκη αποθήκευσης του $31. Η αποθήκευση αυτή γίνεται στη στοίβα.

Κλήση Υπορουτίνας σε Assembly

High-level code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS Assembly code

```
0x00400200 main:  jal simple
0x00400204        add $s0,$s1,$s2


...
0x00401020 simple: jr $ra
```

Ποιό είναι το περιεχόμενο του $31 ($ra) ? ….

Ορίσματα Εισόδου και Αποτελέσματα

- Ορίσματα εισόδου σε υπορουτίνα: $a0 -$a3
- Ορίσματα επιστροφής: $v0

```
// High-level code
int main()
{
  int y;
  ...
  // 4 arguments
  y = diffofsums(2, 3, 4, 5);
  ...
}

int diffofsums(int f, int g,
  int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result; // return value
}
```

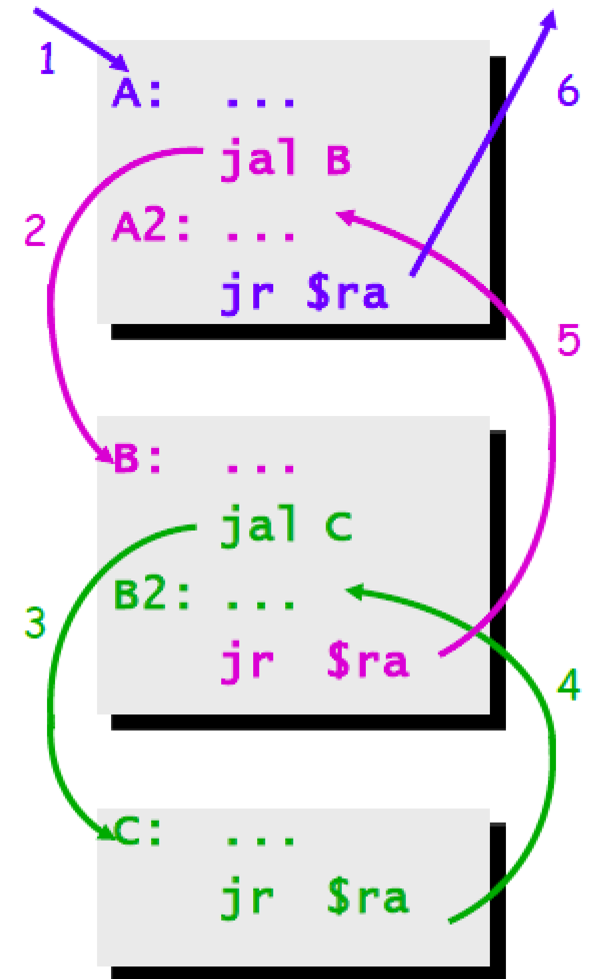
```
# MIPS assembly code
# $s0 = y

main:
  ...
  addi $a0, $0, 2    # argument 0 = 2
  addi $a1, $0, 3    # argument 1 = 3
  addi $a2, $0, 4    # argument 2 = 4
  addi $a3, $0, 5    # argument 3 = 5
  jal  diffofsums    # call procedure
  add  $s0, $v0, $0  # y = returned value
  ...

# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

Κλήσεις και Στοίβα

1. Someone calls A
2. A calls B
3. B calls C
4. C returns to B
5. B returns to A
6. A returns



ΣΤΟΙΒΑ (STACK)

- Είναι μια δομή LIFO (Last In First Out) στη μνήμη
- Υπάρχει ένας δείκτης (Stack Pointer) που δείχνει στην τελευταία εγγραφή (κορυφή στοίβας)
- Στο MIPS ο καταχωρητής $29 είναι ο Stack Pointer

ΛΕΙΤΟΥΡΓΙΕΣ ΣΤΟΙΒΑΣ

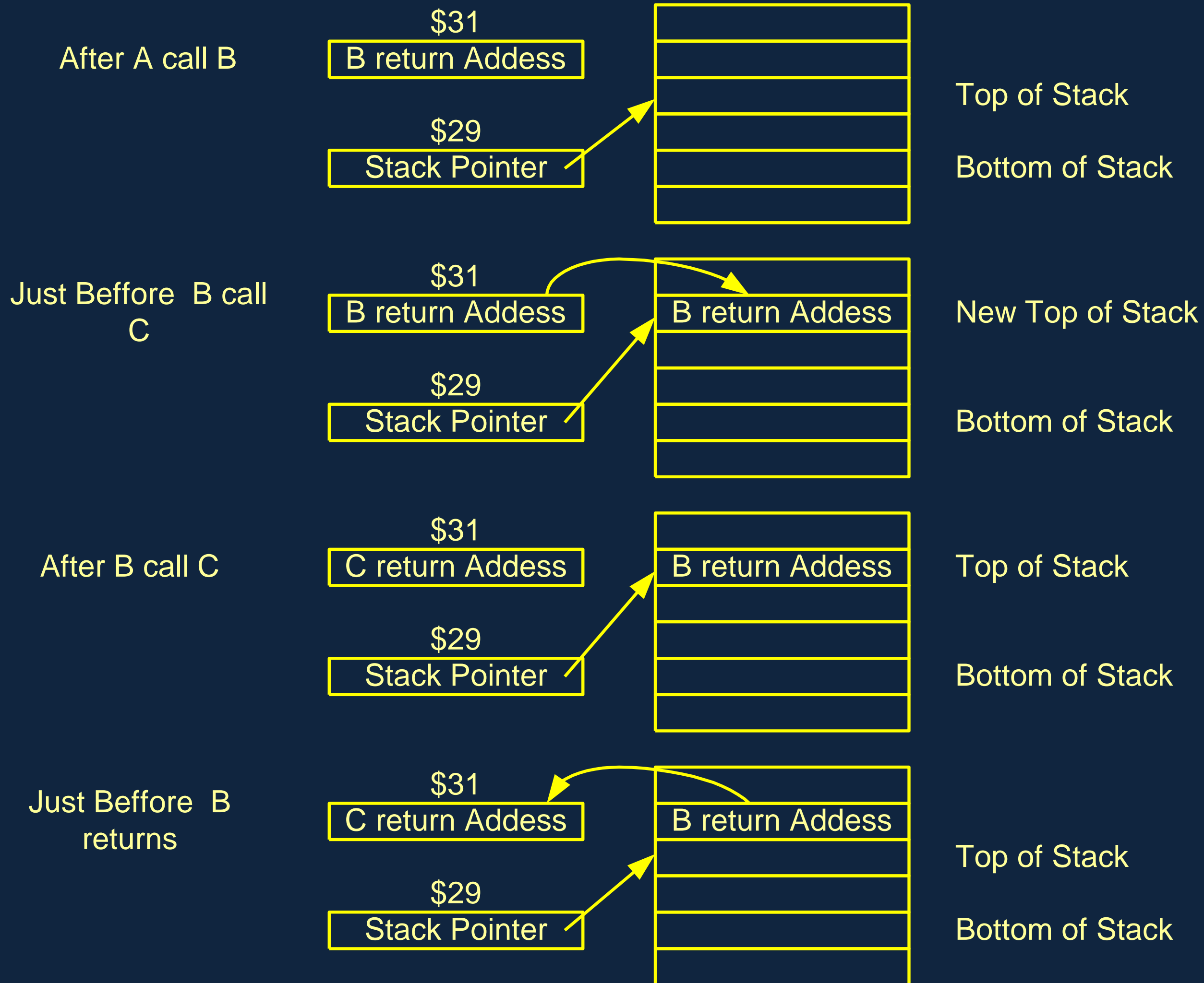
- Αποθήκευση
  - SP ← SP+1
  - M[SP] ← Register
- Ανάκληση
  - Register ←M[SP]
  - SP ← SP-1

Κλήση Εμφωλευμένης Υπορουτίνας

proc1:

```
addi $sp, $sp, -4     # make space on stack
sw $ra, 0($sp)        # save $ra on stack
jal proc2
...
lw $ra, 0($sp)        # restore $s0 from stack
addi $sp, $sp, 4      # deallocate stack space
jr $ra                # return to caller
```

ΣΤΟΙΒΑ (STACK)

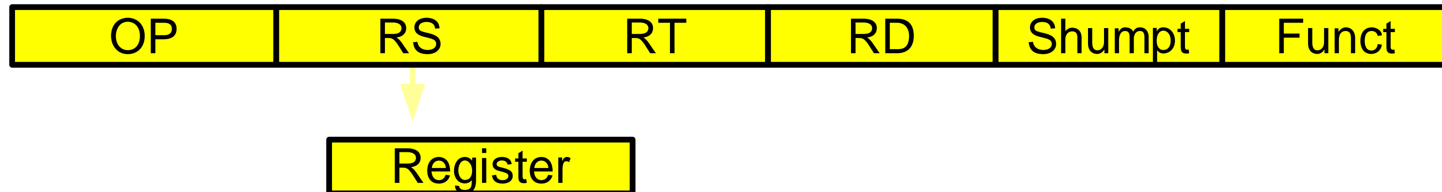


ADDRESSING

4 Modes of addressing

1. **Register addressing**

   R Type Format π.χ. Add $1,$2,$3



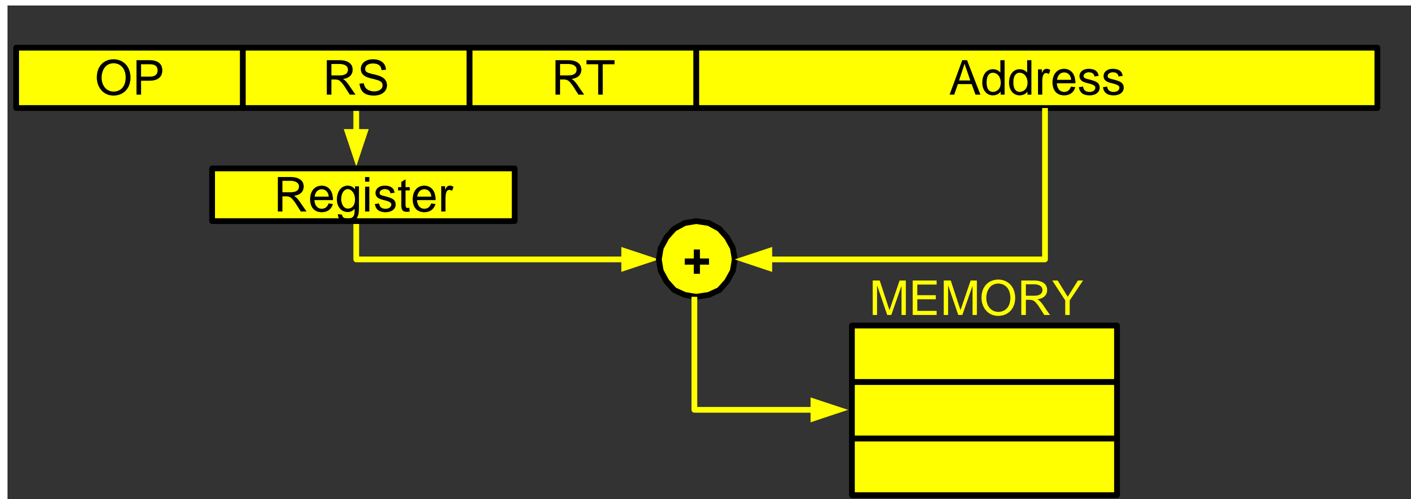
ADDRESSING

2. Base or displacement addressing

I-Type Format π.χ lw $1, 100($2)

θέση μνήμης reg+Address

πχ αν $2=52 θέση μνήμης =152



ADDRESSING

3. Immediate addressing

I-Type Format π.χ **addi $29, $29, 4**

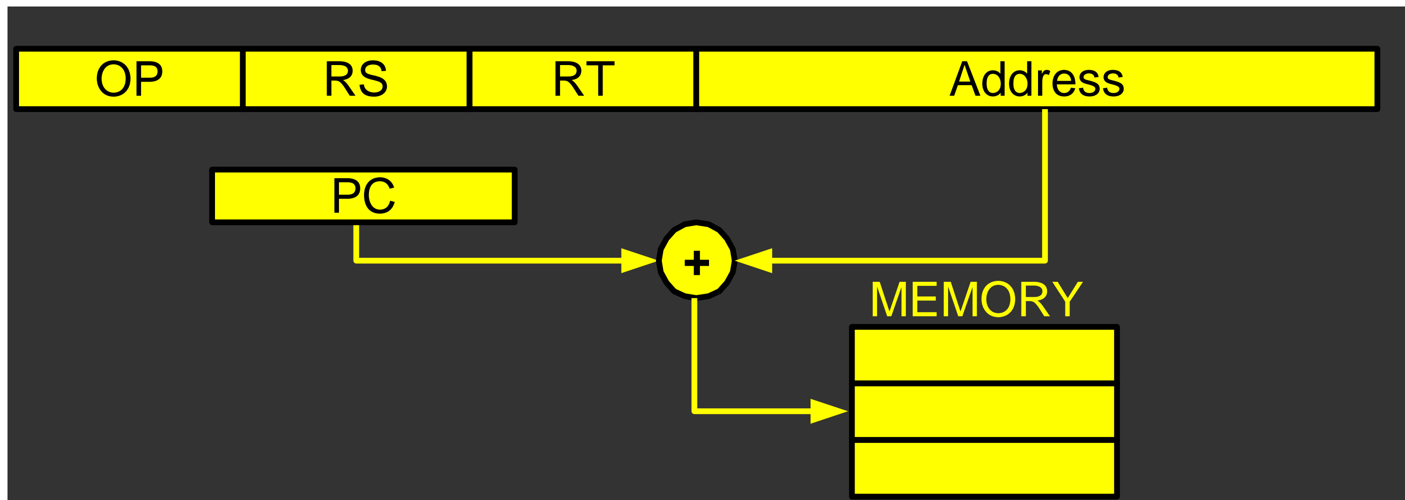
$29=$29+4

ADDRESSING

4. PC-relative addressing

I-Type Format π.χ beq $1,$2,100

αν $1=$2 μετάβαση στη θέση 100+PC



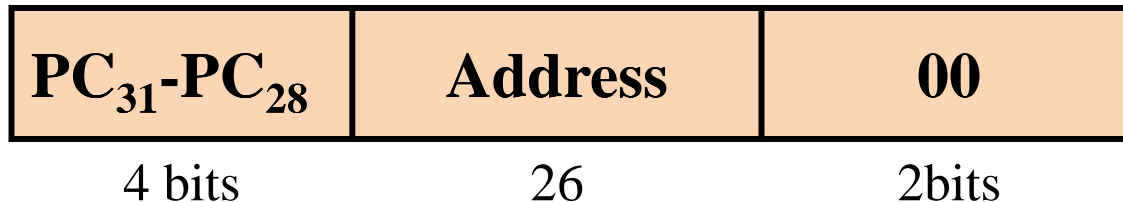
MIPS Memory Access

PC Relative

bne $1, $0, Exit

Pseudo Direct

J Address



Direct (Register Based)

Jr $31

ΠΑΡΑΔΕΙΓΜΑ RELATIVE ADDRESSING

```
Loop:   Mul  $9, $19, $10          # $9 = i x 4
        lw   $8, Sstart ($9)       # $8 = save [i]
        bne  $8, $21, Exit         # go to Exit if save [i] ≠k
        add  $19, $19, $20         # i=i+j
        j    Loop                  # goto Loop
Exit:
```

<table>
<tr><td>80000</td><td>0</td><td>19</td><td>10</td><td>9</td><td>0</td><td>24</td></tr>
<tr><td>80004</td><td>35</td><td>9</td><td>8</td><td colspan="3">1000</td></tr>
<tr><td>80008</td><td>5</td><td>8</td><td>21</td><td colspan="3">2</td></tr>
<tr><td>80012</td><td>0</td><td>19</td><td>20</td><td>19</td><td>0</td><td>32</td></tr>
<tr><td>80016</td><td>2</td><td colspan="5">80000</td></tr>
<tr><td>80020</td><td>Exit</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

ΕΝΤΟΛΕΣ MIPS

Category	Instructions	Example	Meaning	Comments
Arithmetic	Add	add $1, $2, $3	$1 = $2 + $3	3 operands; data in registers
	Subtract	sub $1, $2, $3	$1 = $2 - $3	3 operands; data in registers
	Add immediate	addi $1, $2, 100	$1 = $2 + 100	Used to add constants
Data Transfer	load word	lw $1, 100($2)	$1 = Memory [$2+100]	Data from memory to register
	store word	sw $1, 100($2)	Memory [$2+100] = $1	Data from register to memory
	load upper immediate	lui $1, 100	$1 = 100 * 2^16	Loads constants in upper 16 bits
Conditional Branch	branch on equal	beq $1, $2, 100	if ($1 == $2) go to PC + 4 + 100	Equal test; PC relative branch
	branch on not equal	bne $1, $2, 100	if ($1 != $2) go to PC + 4 + 100	Not equal test; PC relative
	set on less than	slt $1, $2, $3	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $1, $2, 100	if ($2 < 100) $1=1; else $1 = 0	Compare less than constant
Un-conditional Jump	Jump	j 10000	go to 10000	Jump to target address
	Jump register	jr $31	go to $31	For switch, procedure return
	jump and link	jal 10000	$31 = PC + 4; go to 10000	For procedure call