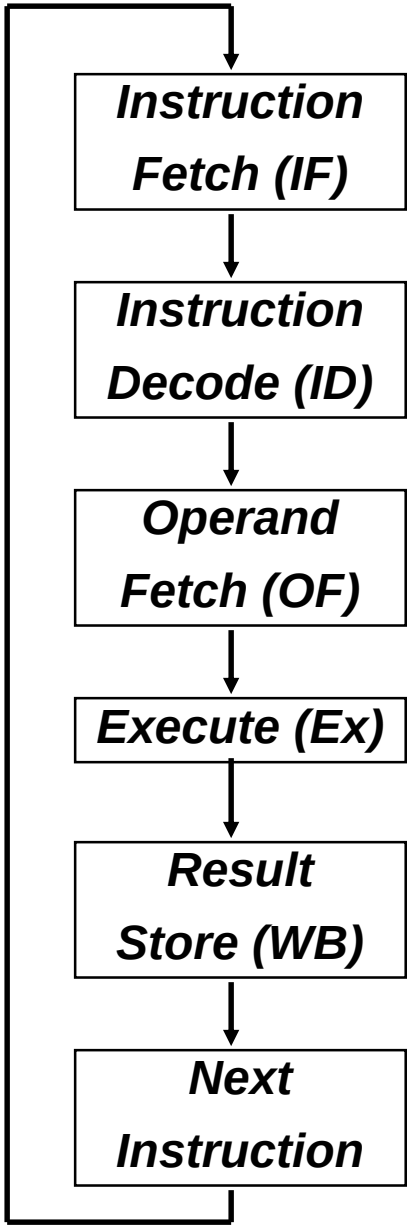
Αρχιτεκτονικές Συνόλου Εντολών



Αριθμός εντολών

Μορφή Εντολών:

μεταβλητό ή σταθερό μέγεθος bytes για κάθε εντολή; (8086 1-17 bytes, MIPS 4 bytes)

Πώς γίνεται η αποκωδικοποίηση (ID);

Που βρίσκονται τα ορίσματα (operands) και το αποτέλεσμα:

Μνήμη-καταχωρητές, *πόσα ορίσματα*, τι μεγέθους;

Ποια είναι στη μνήμη και ποια όχι;

Πόσοι κύκλοι για κάθε εντολή;

Κατηγορίες Αρχιτεκτονικών Συνόλου Εντολών

1. Αρχιτεκτονικές Συσσωρευτή (accumulator architectures)

   (μας θυμίζει κάτι?)

2. Αρχιτεκτονικές επεκταμένου συσσωρευτή ή καταχωρητών ειδικού σκοπού (extended accumulator ή special purpose register)

3. Αρχιτεκτονικές Καταχωρητών Γενικού Σκοπού

   3α. register-memory

   3b. register-register (RISC)

Αρχιτεκτονικές Συσσωρευτή (1)

1η γενιά υπολογιστών: h/w ακριβό, μεγάλο μέγεθος καταχωρητή.

Ένας καταχωρητής για όλες τις αριθμητικές εντολές (*συσσώρευε* όλες τις λειτουργίες → *Συσσωρευτής (Accum)*

Σύνηθες: 1ο όρισμα είναι ο Accum, 2ο η μνήμη, αποτέλεσμα στον Accum π.χ. add 200

Παράδειγμα: A = B + C

Accum = Memory(AddressB);	**Load AddressB**
Accum = Accum + Memory(AddressC);	**Add AddressC**
Memory(AddressA) = Accum;	**Store AddressA**

Όλες οι μεταβλητές αποθηκεύονται στη μνήμη. Δεν υπάρχουν βοηθητικοί καταχωρητές

Αρχιτεκτονικές Συσσωρευτή (2)

Κατά:

Χρειάζονται πολλές εντολές για ένα πρόγραμμα

Κάθε φορά πήγαινε-φέρε από τη μνήμη
(? Κακό είναι αυτό)
Bottleneck ο Accum!

Υπέρ:

Εύκολοι compilers, κατανοητός προγραμματισμός,
εύκολη σχεδίαση h/w

Λύση; Πρόσθεση καταχωρητών για συγκεκριμένες λειτουργίες
(ISAs καταχωρητών ειδικού σκοπού)

Αρχιτεκτονικές Επεκταμένου Συσσωρευτή

Καταχωρητές ειδικού σκοπού π.χ. δεικτοδότηση, αριθμητικές πράξεις

Υπάρχουν εντολές που τα ορίσματα είναι όλα σε καταχωρητές

Κατά βάση (π.χ. σε αριθμητικές εντολές) το ένα όρισμα στη μνήμη.

Αρχιτεκτονικές Καταχωρητών Γενικού Σκοπού

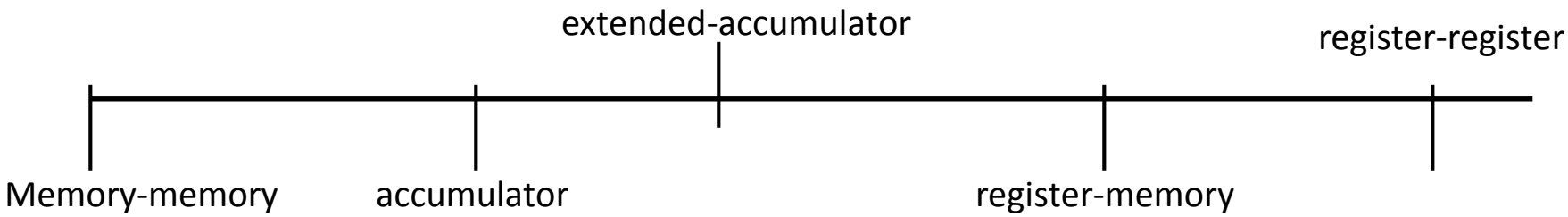
1. CISC

- Complex Instruction Set Computer
- Εντολές για πράξεις Register-Memory ή Memory-Memory
- Αφήνουν το ένα όρισμα να είναι στη μνήμη (πχ. 80386)

2. RISC

- Reduced Instruction Set Computer
- Πράξεις μόνο Register-Register (load store) (1980+)

Load R1, B Add R1, C Store A, R1	**A=B+C**	Load R1, B Load R2, C Add R3, R1, R2 Store A, R3



Αρχιτεκτονική Στοίβας

Καθόλου registers! Stack model ~ 1960!!!

Στοίβα που μεταφέρονται τα ορίσματα που αρχικά βρίσκονται στη μνήμη. Καθώς βγαίνουν γίνονται οι πράξεις και το αποτέλεσμα ξαναμπαίνει στη στοίβα.

Θυμάστε τα HP calculators με reverse polish notation

A=B+C →

push Address C
push AddressB
Add
pop AddressA

Αρχιτεκτονικές CISC

Εντολές μεταβλητού μήκους:

- 1-17 bytes 80x86
- 1-54 bytes VAX, IBM

Γιατί??

- Instruction Memory ακριβή, οικονομία χώρου!!!!

Compilers πιο δύσκολοι!!!

Εμείς στο μάθημα: register-register ISA! (load- store). Γιατί??

1. Οι καταχωρητές είναι γρηγορότεροι από τη μνήμη
2. Μειώνεται η κίνηση με μνήμη
3. Δυνατότητα να υποστηριχθεί σταθερό μήκος εντολών
4. (τα ορίσματα είναι καταχωρητές, άρα ό αριθμός τους (πχ. 1-32 καταχωρητές) όχι δ/νσεις μνήμης

Βασικές Αρχές Σχεδίασης (Patterson-Hennessy COD2e)

1. Η ομοιομορφία των λειτουργιών συμβάλλει στην απλότητα του υλικού (Simplicity favors Regularity)
2. Όσο μικρότερο τόσο ταχύτερο! (smaller is faster)
3. Η καλή σχεδίαση απαιτεί σημαντικούς συμβιβασμούς (Good design demands good compromises)

Γενικότητες? Θα τα δούμε στη συνέχεια.....

Σύνολο Εντολών MIPS

- Η MIPS Technologies έκανε εμπορικό τον Stanford MIPS
- Μεγάλο μερίδιο της αγοράς των πυρήνων ενσωματωμένων επεξεργαστών
- Εφαρμογές σε καταναλωτικά ηλεκτρονικά, εξοπλισμό δικτύων και αποθήκευσης, φωτογραφικές μηχανές, εκτυπωτές, …
- Τυπικό πολλών σύγχρονων ISA (Instruction Set Architecture)
- Πληροφορία στην αποσπώμενη κάρτα Αναφοράς Δεδομένων MIPS (πράσινη κάρτα), και τα Παραρτήματα Β και Ex

Σύνολο Εντολών MIPS

- Λέξεις των 32 bit
- Μνήμη οργανωμένη σε bytes
  - Κάθε byte είναι μια ξεχωριστή δνση
  - 2^{30} λέξεις μνήμης των 32 bits
  - Ακολουθεί το μοντέλο big Endian

Memory [0]	32 bits
Memory [4]	32 bits
Memory [8]	32 bits
Memory [12]	32 bits

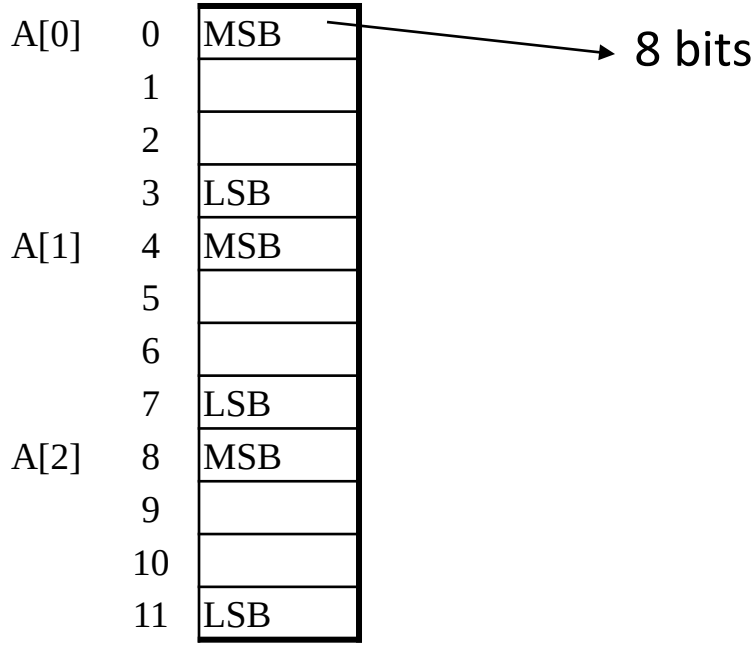
- Register File
  - 32 καταχωρητές γενικού σκοπού
- Εντολές :
  - αποθήκευσης στη μνήμη (lw, sw)
  - αριθμητικές (add, sub κλπ)
  - διακλάδωσης (branch instructions)

Big Endian vs Little Endian

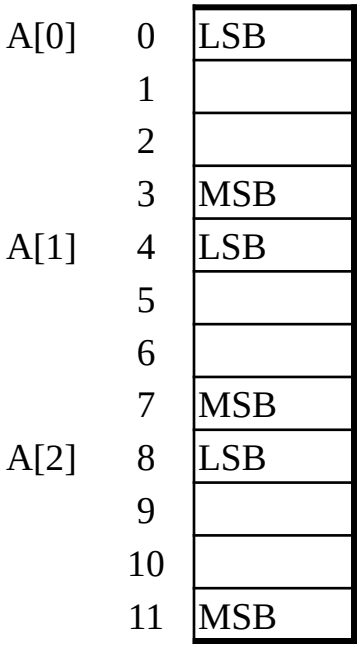
- **Big Endian:** Η δνση του **πιο σημαντικού** byte (MSB) είναι και **δνση** της λέξης
- **Little Endian:** Η δνση του **λιγότερο σημαντικού** byte (LSB) είναι και **δνση** της λέξης
- Η λέξη αποθηκεύεται πάντα σε συνεχόμενες θέσεις:

δνση, δνση+1, δνση+2,δνση+3

BIG_ENDIAN



LITTLE_ENDIAN



MIPS ISA : Αριθμητικές λειτουργίες

- Πρόσθεση και αφαίρεση (add, sub)
  - Πάντα 3 ορίσματα – ΠΟΤΕ δνση μνήμης
  - Δύο προελεύσεις και ένας προορισμός

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

- Όλες οι αριθμητικές λειτουργίες έχουν αυτή τη μορφή

- 1η αρχή σχεδίασης: *η απλότητα ευνοεί την κανονικότητα*
  - Η κανονικότητα κάνει την υλοποίηση απλούστερη
  - Η απλότητα επιτρέπει μεγαλύτερη απόδοση με χαμηλότερο κόστος

Τελεστές - Καταχωρητές

- Οι αριθμητικές εντολές χρησιμοποιούν καταχωρητές ως τελεστές (operands)
- Ο MIPS διαθέτει ένα αρχείο καταχωρητών (register file) με 32 καταχωρητές των 32-bit
  - Χρήση για τα δεδομένα που προσπελάζονται συχνά
  - Αρίθμηση καταχωρητών από 0 έως 31
- Ονόματα του συμβολομεταφραστή (assembler)
  - $t0, $t1, …, $t9 για προσωρινές τιμές
  - $s0, $s1, …, $s7 για αποθηκευμένες μεταβλητές
- $2^η$ αρχή σχεδίασης : *το μικρότερο είναι ταχύτερο*
  - παραβολή με κύρια μνήμη: εκατομμύρια θέσεων

Παράδειγμα

Κώδικας σε C

a = b + c;

d = a – e;

Μετάφραση σε κώδικα MIPS

add a, b, c

sub d, a, e

Παράδειγμα

Κώδικας σε C

f = (g + h) – (i + j);

Τι παράγει ο compiler?

Μετάφραση σε κώδικα MIPS

```
add $t0, $s1, $s2    # προσωρινή μεταβλητή t0
add $t1, $s3. $s4    # προσωρινή μεταβλητή t1
sub $s0, $t0, $t1
```

Τελεστές - Μνήμη

Οι γλώσσες προγραμματισμού έχουν:

- απλές μεταβλητές
- σύνθετες δομές (π.χ. arrays, structs)

Ο υπολογιστής τις αναπαριστά <u>ΠΑΝΤΑ ΣΤΗ ΜΝΗΜΗ.</u>

- Επομένως χρειαζόμαστε εντολές μεταφοράς δεδομένων από και προς τη μνήμη.

Τελεστές - Μνήμη

- Εντολή μεταφοράς δεδομένων από τη μνήμη

  load καταχωρητής, σταθερά(καταχωρητής)

  lw $t1, 4($s2)

- φορτώνουμε στον $t1 την τιμή M[$s2+4]

Παράδειγμα

- **Κώδικας C**

  g = h + A[8];

  - g στον $s1, h στον $s2 και η δνση βάσης του A στον $s3.

- **Μεταγλωττισμένος κώδικας MIPS**

  - Ο δείκτης 8 απαιτεί offset 32 (4 byte ανά λέξη).

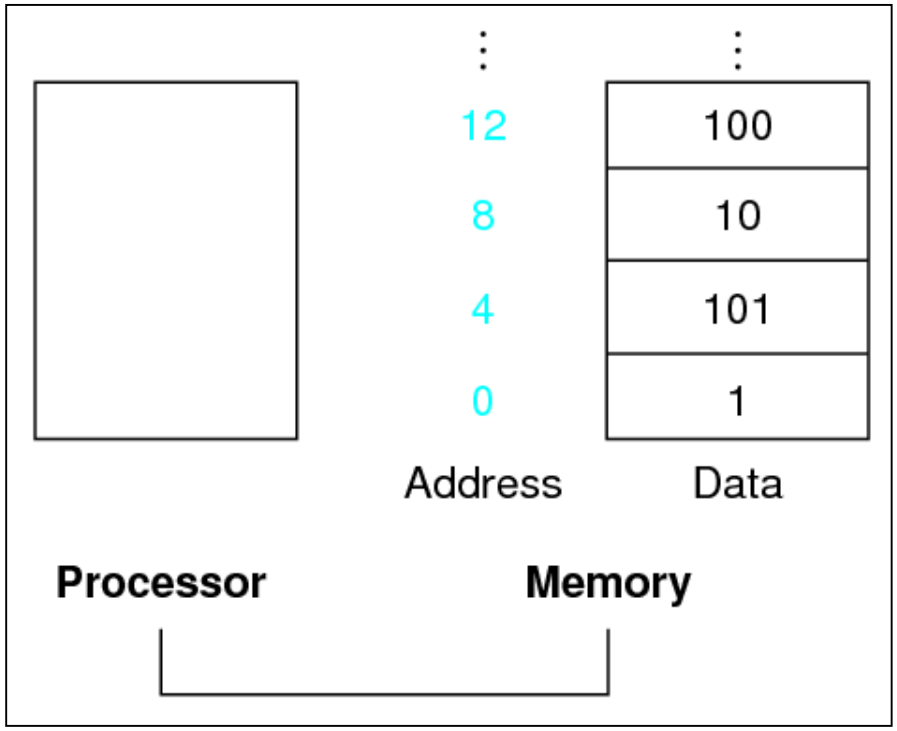
offset **base register**

lw $t0, 32($s3)

add $s1, $s2, $t0

Οργάνωση Μνήμης

- Μνήμη είναι byte addressable
- Δύο διαδοχικές λέξεις διαφέρουν κατά 4
- alignment restriction (ευθυγράμμιση)
  - λέξεις ξεκινάνε πάντα σε διεύθυνση πολ/σιο του 4



Παράδειγμα 2

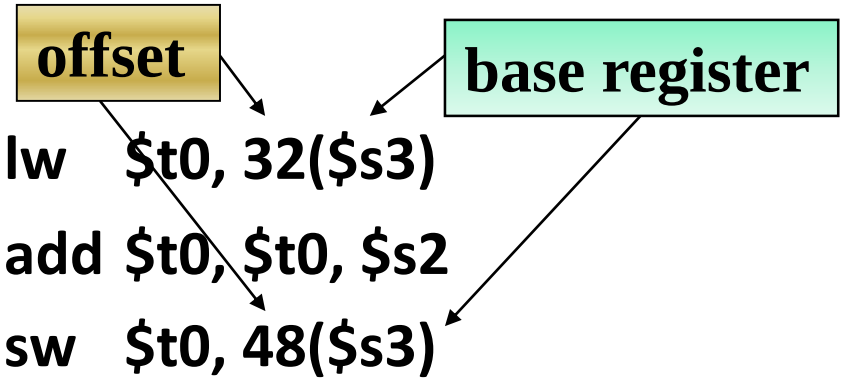
- **Κώδικας C**

  A[12] = h + A[8];

  - h στον $s2 και η δνση βάσης του A στον $s3.

- **Μεταγλωττισμένος κώδικας MIPS**

  - Ο δείκτης 8 απαιτεί offset 32 (4 byte ανά λέξη).



Άμεσοι Τελεστέοι (Immediate)

- Σταθερά δεδομένα καθορίζονται σε μια εντολή

addi $s3, $s3, 4

- Δεν υπάρχει εντολή άμεσης αφαίρεσης (sub immediate)
- Απλώς χρησιμοποιείται μια αρνητική σταθέρα

addi $s2, $s1, -1

- 3η αρχή σχεδίασης: *Κάνε τη συνηθισμένη περίπτωση γρήγορη*
  - Οι μικρές σταθερές είναι συνηθισμένες
  - Ο άμεσος τελεστέος αποφεύγει μια εντολή φόρτωσης (load)

Η σταθερά ΜΗΔΕΝ

- Ακολουθώντας πάλι την 3η αρχή σχεδίασης, ο MIPS έχει στον καταχωρητή $zero αποθηκευμένη τη σταθερά 0.
  - Δεν μπορεί να εγγραφεί άλλη τιμή

- Χρήσιμη σε πολλές λειτουργίες
  - Μετακίνηση δεδομένων μεταξύ καταχωρητών

  π.χ. add $t1, $t2, $zero

Σύνοψη – Τελεστέοι MIPS

Συνοπτικά, στον MIPS ο τελεστής κάποιας εντολής μπορεί να είναι :

1. Ένας από τους 32 καταχωρητές
2. Μία από τις 2^{30} λέξεις της μνήμης
3. Ένα από τα 2^{32} bytes της μνήμης

Κανόνες Ονοματοδοσίας και Χρήση των MIPS Registers

- Εκτός από το συνήθη συμβολισμό των καταχωρητών με $ ακολουθούμενο από τον αριθμό του καταχωρητή, μπορούν επίσης να παρασταθούν και ως εξής :

Αρ. Καταχωρητή	Όνομα	Χρήση	Preserved on call?
0	$zero	Constant value 0	n.a.
1	$at	Reserved for assembler	όχι
2-3	$v0-$v1	Values for result and expression evaluation	όχι
4-7	$a0-$a3	Arguments	ναι
8-15	$t0-$t7	Temporaries	όχι
16-23	$s0-$s7	Saved	ναι
24-25	$t8-$t9	More temporaries	όχι
26-27	$k0-$k1	Reserved for operating system	ναι
28	$gp	Global pointer	ναι
29	$sp	Stack pointer	ναι
30	$fp	Frame pointer	ναι
31	$ra	Return address	ναι

Αναπαράσταση Εντολών (1)

- Οι εντολές κωδικοποιούνται στο δυαδικό σύστημα
  - Κώδικας μηχανής (machine code)
  - Υλικό υπολογιστών → υψηλή-χαμηλή τάση, κλπ.

- Εντολές MIPS :
  - Κωδικοποιούνται ως λέξεις εντολής των 32 bit
  - Μικρός αριθμός μορφών (formats) για τον κωδικό λειτουργίας (opcode), τους αριθμούς καταχωρητών, κλπ. …
  - Κανονικότητα!

- Αριθμοί καταχωρητών
  - $t0 – $t7 είναι οι καταχωρητές 8 – 15
  - $t8 – $t9 είναι οι καταχωρητές 24 – 25
  - $s0 – $s7 είναι οι καταχωρητές 16 – 23

Αναπαράσταση Εντολών (2)

Συμβολική αναπαράσταση:

add $t0, $s1, $s2 → **Assembly**

Πώς την καταλαβαίνει ο MIPS?

	$s1	$s2	$t0	unused	
0	17	18	8	0	32
add					

Κώδικας μηχανής

000000	10001	10010	01000	00000	10000
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Μορφή Εντολής – Instruction Format

Θυμηθείτε την 1η αρχή σχεδίασης: *Η ομοιομορφία των λειτουργιών συμβάλλει στην απλότητα του υλικού*

R-Type
(register type)

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

Op: opcode

rs,rt: register source operands

Rd: register destination operand

Shamt: shift amount

Funct: op specific (function code)

add $rd, $rs, $rt

MIPS R-Type (ALU)

R-Type: Όλες οι εντολές της ALU που χρησιμοποιούν 3 καταχωρητές

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

- Παραδείγματα :
  - add $1,$2,$3 and $1,$2,$3
  - sub $1,$2,$3 or $1,$2,$3

Destination register in rd

Operand register in rs

Operand register in rt

Αναπαράσταση Εντολών στον Υπολογιστή (R-Type)

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

Τι γίνεται με τη load?

Πώς χωράνε οι τελεστές της στα παραπάνω πεδία? Π.χ. η σταθερά της lw.

lw $t1, 8000($s3)

σε ποιο πεδίο χωράει;

MIPS I-Type

- Δεν μας αρκεί το R-Type
  - Τι γίνεται με εντολές που θέλουν ορίσματα διευθύνσεις ή σταθερές?
  - Θυμηθείτε, θέλουμε σταθερό μέγεθος κάθε εντολής (32 bit)

- *Η καλή σχεδίαση απαιτεί σημαντικούς συμβιβασμούς (3η αρχή)*

I-Type:

op	rs	rt	address_offset
6 bits	*5 bits*	*5 bits*	*16 bits*

lw $rt, address_offset($rs)

Τα 3 πρώτα πεδία (op,rs, rt) έχουν το ίδιο όνομα και μέγεθος όπως και πριν

Αναπαράσταση Εντολών στον Υπολογιστή (I-Type)

Παράδειγμα:

lw $t0, 32($s3)

Καταχωρητές (σκονάκι ☺)
$s0, ..., $s7 αντιστοιχίζονται στους 16 - 23
$t0, ..., $t7 αντιστοιχίζονται στους 8 - 15

I-format

op	rs	rt	σταθερά ή διεύθυνση
6 bit	5 bit	5 bit	16 bit
xxxxxx	19	8	32

MIPS I-Type : Load/Store

OP	rs	rt	address
6 bits	5 bits	5 bits	16 bits

– *address: 16-bit memory address offset in bytes added to base register.*

- Παραδείγματα :

– Store word: sw $3, 500($4)

source register in rt

Offset

base register in rs

– Load word: lw $1, 30($2)

Destination register in rt

Offset

base register in rs

MIPS I-Type : ALU

Οι I-Type εντολές της ALU χρησιμοποιούν 2 καταχωρητές και μία σταθερή τιμή
I-Type είναι και οι εντολές Loads/stores, conditional branches.

OP	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

– *immediate: Constant second operand for ALU instruction.*

- Παραδείγματα :

  – add immediate: addi $1,$2,100

  – and immediate andi $1,$2,10

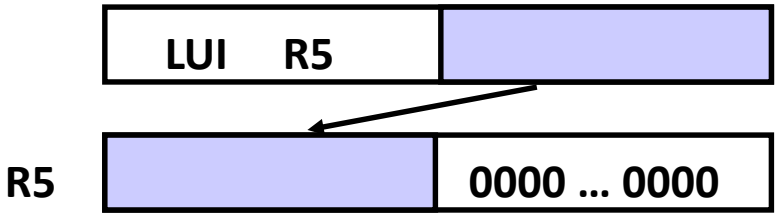
Constant operand in immediate

Result register in rt

Source operand register in rs

MIPS data transfer instructions : Παραδείγματα (1)

Instruction	*Σχόλια*
sw $3, 500($4)	Store word
sh $3, 502($2),	Store half
sb $2, 41($3)	Store byte
lw $1, 30($2)	Load word
lh $1, 40($3)	Load halfword
lhu $1, 40($3)	Load halfword unsigned
lb $1, 40($3)	Load byte
lbu $1, 40($3)	Load byte unsigned
lui $1, 40	Load Upper Immediate (16 bits shifted left by 16)



MIPS data transfer instructions : Παραδείγματα (2)

Τι γίνεται με τις μεγαλύτερες σταθερές;

- Έστω ότι θέλουμε να φορτώσουμε μια 32-bit σταθερά σε κάποιο καταχωρητή, π.χ. 1010101010101010 1010101010101010
- Θα χρησιμοποιήσουμε την "Load Upper Immediate" εντολή
  π.χ. lui $t0, 1010101010101010

Μηδενικά

$t0	**1010101010101010**	**0000000000000000**

- Στη συνέχεια πρέπει να θέσουμε σωστά τα lower order bits
  π.χ. ori $t0, 1010101010101010

	1010101010101010	**0000000000000000**
ori	**0000000000000000**	**1010101010101010**
	1010101010101010	**1010101010101010**

Αναπαράσταση Εντολών στον Υπολογιστή

εντολή	μορφή	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	δ.ε.
`sub`	R	0	reg	reg	reg	0	34_{ten}	δ.ε.
`addi`	I	8_{ten}	reg	reg	δ.ε.	δ.ε.	δ.ε.	σταθ.
`lw`	I	35_{ten}	reg	reg	δ.ε.	δ.ε.	δ.ε.	διευθ.
`sw`	I	43_{ten}	reg	reg	δ.ε.	δ.ε.	δ.ε.	διευθ.

Αναπαράσταση Εντολών στον Υπολογιστή

Παράδειγμα: **Μεταγλωττίστε το A[300] = h + A[300]**

$t1 δνση βάσης πίνακα A (32 bit/στοιχείο A[i]), $s2 μεταβλητή h

```
lw   $t0, 1200($t1)
add $t0, $s2, $t0
sw   $t0, 1200($t1)
```

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

op	rs	rt	rd	shamt	funct
10**0**011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	8	0	32
10**1**011	01001	01000	0000 0100 1011 0000		

Λογικές Λειτουργίες (Πράξεις) (1)

Λογικές Λειτουργίες	Τελεστές C	Εντολές MIPS
Shift left	<<	`Sll (shift left logical)`
Shift right	>>	`Srl (shift right logical)`
AND	&	`and, andi`
OR	|	`or, ori`
NOT	~	`nor`

Λογικές Λειτουργίες (Πράξεις) (2)

SHIFT

$s0: 0000 0000 0000 0000 0000 0000 0000 1001 = 9_{ten} ☺

sll $t2, $s0, 4

Κάνουμε shift αριστερά το περιεχόμενο του $s0 κατά 4 θέσεις

0000 0000 0000 0000 0000 0000 1001 0000 = 144_{ten}

και τοποθετούμε το αποτέλεσμα στον $t2.

!!Το περιεχόμενο του $s0 μένει αμετάβλητο!!

Λογικές Λειτουργίες (Πράξεις) (3)

SHIFT

```
sll $t2, $s0, 4
```

Καταχωρητές (σκονάκι ☺)
$s0, ..., $s7 αντιστοιχίζονται στους 16 - 23
$t0, ..., $t7 αντιστοιχίζονται στους 8 - 15

6 bit	5 bit	5 bit	5 bit	5 bit	6 bit
op	rs	rt	rd	shamt	funct
0	0	16	10	4	0
000000	00000	10000	01010	00100	000000

sll: opcode=0, funct=0

Λογικές Λειτουργίες (Πράξεις) (4)

AND, OR

$t2: 0000 0000 0000 0000 0000 1101 0000 0000

$t1: 0000 0000 0000 0000 0011 1100 0000 0000

and $t0, $t1, $t2 # Μάσκα

$t0: 0000 0000 0000 0000 0000 1100 0000 0000

or $t0, $t1, $t2

$t0: 0000 0000 0000 0000 0011 1101 0000 0000

Λογικές Λειτουργίες (Πράξεις) (5)

NOT, NOR

$t1: 0000 0000 0000 0000 0011 1100 0000 0000

$t3: 0000 0000 0000 0000 0000 0000 0000 0000

not $t0, $t1 **δεν υπάρχει** γιατί θέλουμε πάντα 2 καταχωρητές source. Άρα χρησιμοποιούμε τη **nor:**

A NOR 0 = NOT (A OR 0) = NOT A

nor $t0, $t1, $t3

$t0: 1111 1111 1111 1111 1100 0011 1111 1111

MIPS Arithmetic Instructions : Παραδείγματα

Instruction	*Παράδειγμα*	*Έννοια*	*Σχόλια*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

MIPS Logic/Shift Instructions : Παραδείγματα

Instruction	*Παράδειγμα*	*Έννοια*	*Σχόλια*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

Εντολές Λήψης Αποφάσεων (1)

<u>beq, bne</u>

beq reg1, reg2, L1 #branch if equal

Αν οι καταχωρητές reg1 και reg2 **είναι ίσοι**, πήγαινε στην ετικέτα L1

bne reg1, reg2, L1 #branch if not equal

Αν οι καταχωρητές reg1 και reg2 **δεν είναι ίσοι**, πήγαινε στην ετικέτα L1

Εντολές Λήψης Αποφάσεων (2)

Παράδειγμα:

if(i == j) f = g + h; else f = g – h;

με f, g, h, i, j αντιστοιχούνται σε $s0, ..., $s4

version 1

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j Exit
Else:   sub $s0, $s1, $s2
Exit:
```

version 2

```
        beq $s3, $s4, Then
        sub $s0, $s1, $s2
        j Exit
Then: add $s0, $s1, $s2
Exit:
```

Εντολές Λήψης Αποφάσεων (3)

Βρόχοι (Loops)

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

με i = $s3, k = $s5, save base addr = $s6

```
Loop:   sll   $t1, $s3, 2 #πολ/ζω i επί 4
        add   $t1, $t1, $s6
        lw    $t0, 0($t1)
        bne   $t0, $s5, Exit
        addi  $s3, $s3, 1
        j     Loop
Exit:
```

Εντολές Λήψης Αποφάσεων (4)

<u>Συγκρίσεις</u>

slt $t0, $s3, $s4 # set on less than

Ο καταχωρητής $t0 τίθεται με 1 αν η τιμή στον $s3 είναι μικρότερη από την τιμή στο $s4.

- Σταθερές ως τελεστέοι είναι δημοφιλείς στις συγκρίσεις

slti $t0, $s2, 10 # set on less than immediate

Ο καταχωρητής $t0 τίθεται με 1 αν η τιμή στον $s2 είναι μικρότερη από την τιμή 10.

Εντολές Λήψης Αποφάσεων (5)

- Γιατί όχι blt, bge κτλ;
- Το υλικό για τις <, ≥, … είναι πιο αργό από αυτό για τις =, ≠
    - Ο συνδυασμός συνθηκών για μια διακλάδωση περιλαμβάνει περισσότερη δουλειά ανά εντολή.
    - Πιο αργό ρολόι
    - Επιβαρύνονται όλες οι εντολές!

- Οι beq, bne είναι η συνήθης περίπτωση
- Καλός σχεδιαστικός συμβιβασμός.

MIPS Branch, Compare, Jump : Παραδείγματα

Instruction	*Παράδειγμα*	*Έννοια*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+10 *Equal test; PC relative branch*
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100 *Not equal test; PC relative branch*
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; 2's comp.*
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0 *Compare < constant; 2's comp.*
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; natural numbers*
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0 *Compare < constant; natural numbers*
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*

Εντολές διακλάδωσης – branching instructions

branch if equal

beq $s3, 4s4, L1 # goto L1 if $s3 equals $s4

branch if !equal

bne $s3, 4s4, L1 # goto L1 if $s3 not equals $s4

unconditional Jump

jr $t1 # goto $t1

..... είναι I –Type εντολές

slt $t0, $s3, $s4 #set $t0 to 1 if $s3 is less than $s4;else set $t0 to 0

Όμως: j L1 # goto L1

Πόσο μεγάλο είναι το μήκος του address L1;

Πόσο «μεγάλο» μπορεί να είναι το άλμα;

MIPS Branch I-Type

OP	rs	rt	address
6 bits	5 bits	5 bits	16 bits

– *address: 16-bit memory address branch target offset in words added to PC to form branch address.*

- Παραδείγματα :

Final offset is calculated in bytes, equals to {instruction field address} x 4, e.g. new PC = PC + 400

Register in rt

Register in rs

  - Branch on equal beq $1,$2,100
  - Branch on not equal bne $1,$2,100

MIPS J-Type

J-Type: jump j, jump and link jal

OP	jump target
6 bits	26 bits

– *jump target: jump memory address in words.*

final jump memory address in bytes is calculated from {jump target} x 4

- Παραδείγματα :
  - Jump — j 10000
  - Jump and Link — jal 10000

Εντολές Jump

- Jump (*J-type*):
  - j 10000 # jump to address 10000
- Jump Register (*R-type*):
  - jr rs # jump to 32 bit address in register rs
- Jump and Link (*J-type*):
  - jal 10000 # jump to 10000 and save PC in R31
  - Χρήση για κλήση διαδικασιών/μεθόδων.
  - Αποθηκεύει τη διεύθυνση επιστροφής (PC+4) στον καταχωρητή 31 ($ra)
  - Η επιστροφή από τη διαδικασία επιτυγχάνεται με χρήση “jr $ra”
  - Οι εμφωλιασμένες διαδικασίες θα πρέπει να αποθηκεύουν τον $ra στη στοίβα και να χρησιμοποιούν τους καταχωρητές $sp (stack pointer) και $fp (frame pointer) για να χειρίζονται τη στοίβα.

Σύνοψη – MIPS Instruction Formats

- R-type (add, sub, slt, jr)

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

- I-type (beq, bne + addi, lui + lw, sw)

op	rs	rt	immediate value / address offset
6 bits	5 bits	5 bits	16 bits

- J-type (j, jal)

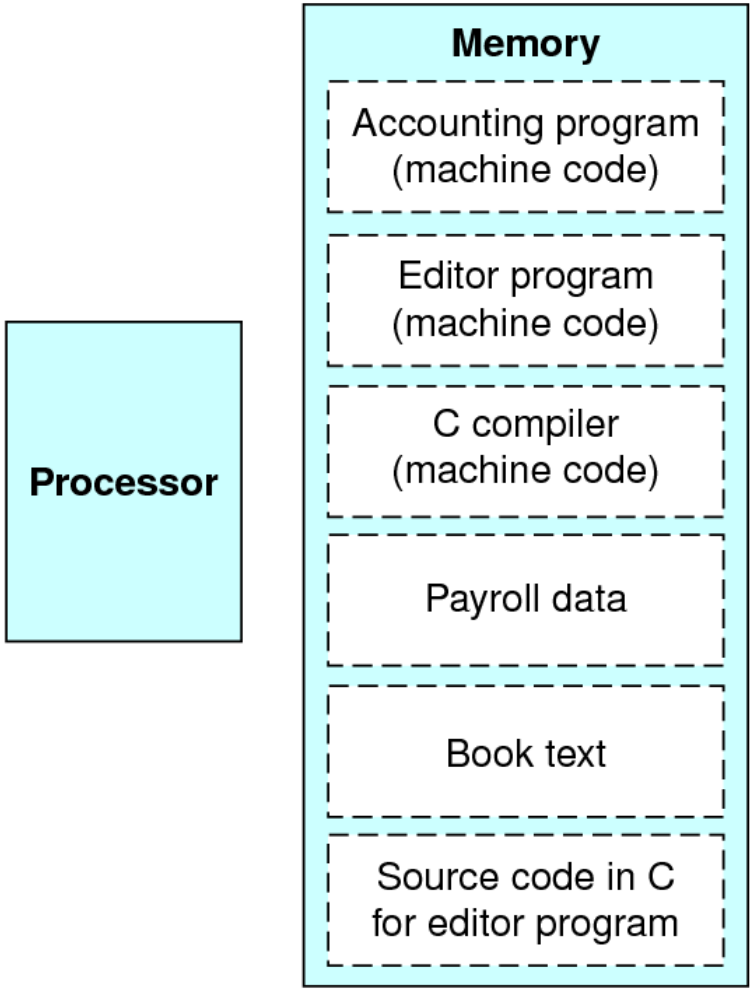
op	jump target address
6 bits	26 bits

Έννοια αποθηκευμένου προγράμματος

Ο υπολογιστής κάνει πολλές εργασίες φορτώνοντας δεδομένα στο μνήμη.

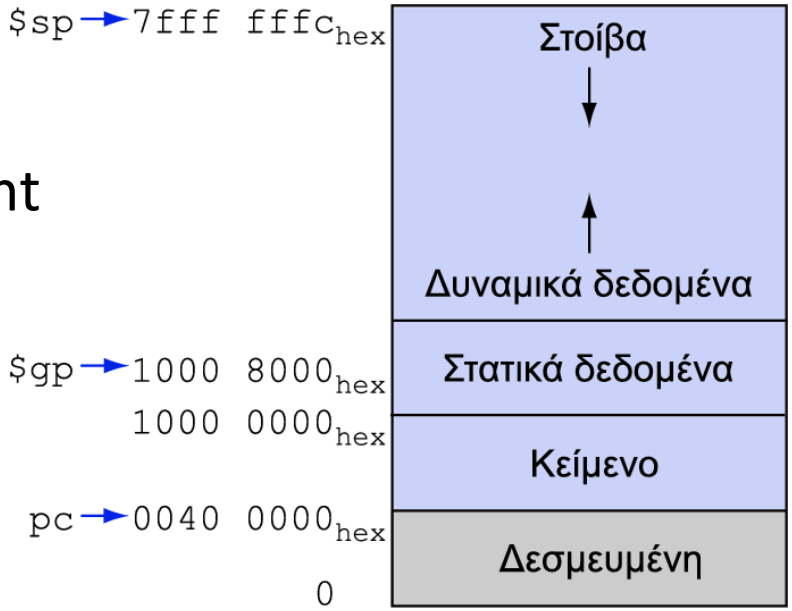
Δεδομένα και εντολές είναι στοιχεία στη μνήμη.

Π.χ. compilers μεταφράζουν στοιχεία σε κάποια άλλα στοιχεία.



Διάταξη της Μνήμης ενός προγράμματος (Memory layout of a program)

- Κείμενο (Text)
  - Κώδικας προγράμματος
- Στατικά Δεδομένα (Static data)
  - Καθολικές Μεταβλητές (π.χ. στατικές μεταβλητές της C, constant arrays και συμβολοσειρές (strings)
- Δυναμικά Δεδομένα
  - Σωρός (Heap)
  - π.χ. malloc στη C
- Στοίβα (stack)
  - Αυτόματη αποθήκευση



Memory layout of programs (συνέχεια)

1. Text segment (κώδικας προγράμματος)

2. Initialized data segment (or .data segment):

   contains global variables, static variables

   divided into read only area + read-write area

   e.g char s[]="hello world"

   int debug = 1

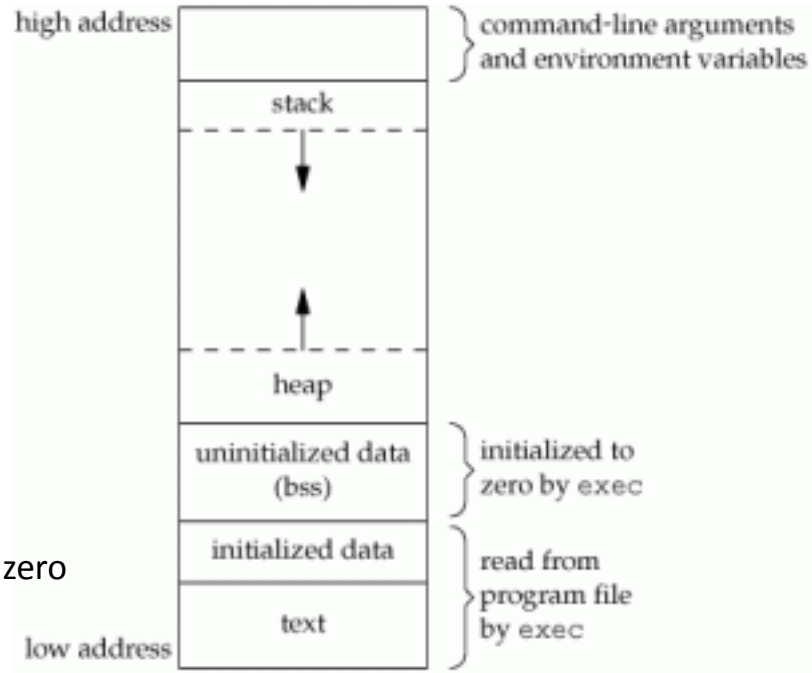
   const char * string ="hello world";

   "hello world" literal stored in ro area,

   string stored in rw area

   static int i = 10

   global int i = 10



3. Uninitialized data segment (or .bss segment)

   bss: block started by symbol

   all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

   int j ; static int i;

4. Stack

   stack pointer. Local variables from a function.

5. Heap

Heap pointer. Dynamic memory allocation. Malloc, realloc, free.

Τύποι Δεδομένων

- Applications / HLL
  - Integer
  - Floating point Character
  - String
  - Date
  - Currency
  - Text,
  - Objects (ADT)
  - Blob
  - double precision
  - Signed, unsigned

- Hardware support
  - Numeric data types
    - Integers
      - 8 / 16 / 32 / 64 bits
      - Signed or unsigned
      - Binary coded decimal (COBOL, Y2K!)
    - Floating point
      - 32 / 64 /128 bits
  - Nonnumeric data types
    - Characters
    - Strings
    - Boolean (bit maps)
    - Pointers

Τύποι Δεδομένων : MIPS (1)

- Βασικός τύπος δεδομένων: 32-bit word
  - 0100 0011 0100 1001 0101 0011 0100 0101
  - Integers (signed or unsigned)
    - 1,128,878,917
  - Floating point numbers
    - 201.32421875
  - 4 ASCII χαρακτήρες
    - C I S E
  - Διευθύνσεις μνήμης (pointers)
    - 0x43495345
  - Εντολές

Τύποι Δεδομένων : MIPS (2)

- 16-bit σταθερές (immediates)
  - addi $s0, $s1, 0x8020
  - lw $t0, 20($s0)
- Half word (16 bits)
  - **lh** (**lhu**): load half word lh $t0, 20($s0)
  - **sh**: save half word sh $t0, 20($s0)
- Byte (8 bits)
  - **lb** (**lbu**): load byte lb $t0, 20($s0)
  - **sb**: save byte sb $t0, 20($s0)

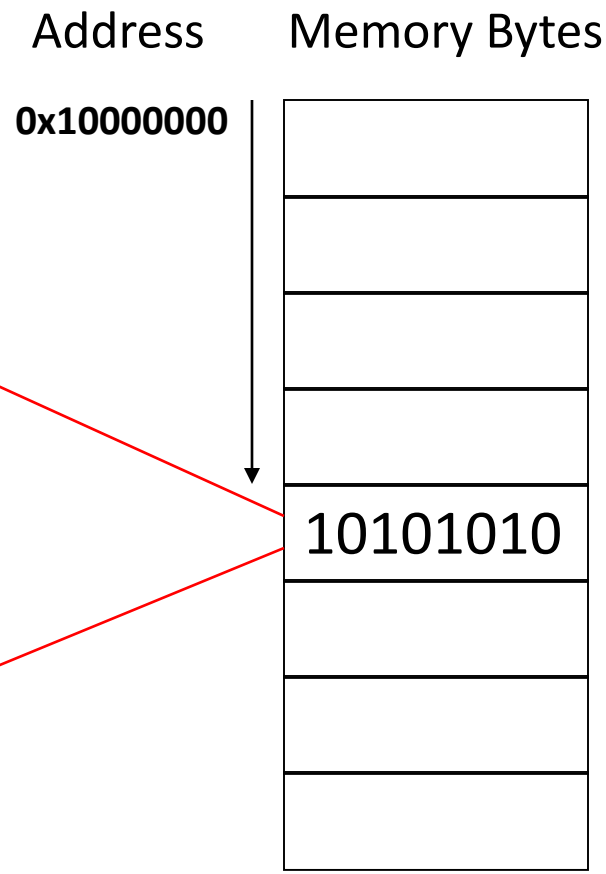
Εντολές λειτουργίας Byte

lb $s1, 4($s0)

$s0:	0x10000000
$s1:	0xFFFFFFAA

lbu $s1, 4($s0)

$s0:	0x10000000
$s1:	0x000000AA



Παράδειγμα : Αντιγραφή String

```
Void strcpy (char[], char y[]) {
    int i;
    i = 0;
    while ((x[i]=y[i]) != 0)
        i = i + 1;
}
```

```
strcpy:
    subi $sp, $sp, 4
    sw   $s0, 0($sp)
    add  $s0, $zero, $zero
L1: add $t1, $a1, $s0
    lb   $t2, 0($t1)
    add  $t3, $a0, $s0
    sb   $t2, 0($t3)
    beq  $t2, $zero, L2
    addi $s0, $s0, 1
    j    L1
L2: lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

C convention:

Null byte (00000000) represents end of the string

Importance of comments in MIPS!

Σταθερές

- Συχνά χρησιμοποιούνται μικρές σταθερές (50% των τελεστών)
  - e.g., A = A + 5;
- Λύση
  - Αποθήκευση 'τυπικών σταθερών' στη μνήμη και φόρτωση τους.
  - Δημιουγία hard-wired καταχωρητών (π.χ. $zero) για σταθερές όπως 0, 1 κτλ.
- MIPS Instructions:

```
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 0x4a
addi $29, $29, 4
```

8	29	29	4

Τρόποι Διευθυνσιοδότησης

- Διευθύνσεις για *δεδομένα* και *εντολές*
- Δεδομένα (τελεστές / αποτελέσματα)
  - Καταχωρητές
  - Θέσεις μνήμης
  - Σταθερές
- Αποδοτική κωδικοποίηση διευθύνσεων (χώρος: 32 bits)
  - Καταχωρητές (32) => 5 bits κωδικοποιούν 1 32-bit δνση
  - *Destructive* instructions: reg2 = reg2 + reg1
  - Accumulator
  - Stack
- Τα opcodes μπορούν να χρησιμοποιηθούν με διαφορετικούς τρόπους διευθυνσιοδότησης
  - ***Orthogonality*** of opcodes and addressing modes

Διευθυνσιοδότηση Δεδομένων

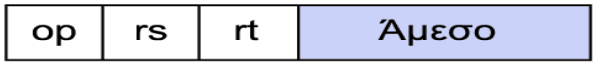
- Διευθυνσιοδότηση μέσω καταχωρητή (Register addressing)
  - Η πιο συνηθισμένη (σύντομη και ταχύτατη)
  - add $3, $2, $1
- Διευθυνσιοδότηση βάσης (Base addressing)
  - Ο τελεστέος είναι σε μια θέση μνήμης με κάποιο offset
  - lw $t0, 20 ($t1)
- Άμεση διευθυνσιοδότηση (Immediate addressing)
  - Ο τελεστέος είναι μια μικρή σταθερά και περιέχεται στην εντολή
  - addi $t0, $t1, 4 (signed 16-bit integer)

Διευθυνσιοδότηση Εντολών

- Οι διευθύνσεις έχουν μήκος 32 bits
- Καταχωρητής ειδικού σκοπού : PC (**program counter**)
  - Αποθηκεύει τη διεύθυνση της εντολής που εκτελείται εκείνη τη στιγμή
- Διευθυνσιοδότηση με χρήση PC (PC-relative addressing)
  - **Branches**
  - Νέα διεύθυνση: PC + (constant in the instruction) * 4
  - beq $t0, $t1, 20
- Ψεύδοαμεση διευθυνσιοδότηση (Pseudodirect addressing)
  - **Jumps**
  - Νέα διεύθυνση: PC[31:28] **:** (constant in the instruction) * 4

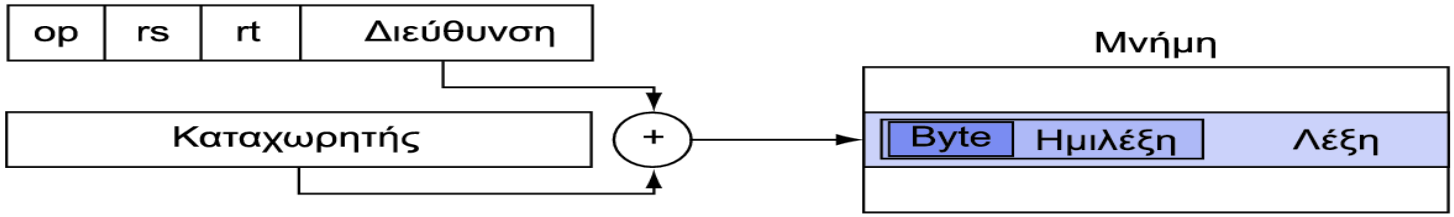
Περίληψη Τρόπων Διευθυνσιοδότησης

1. Άμεση διευθυνσιοδότηση

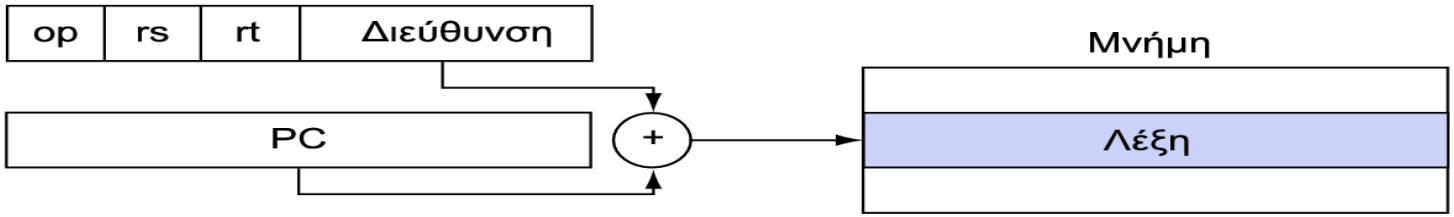


2. Διευθυνσιοδότηση μέσω καταχωρητή

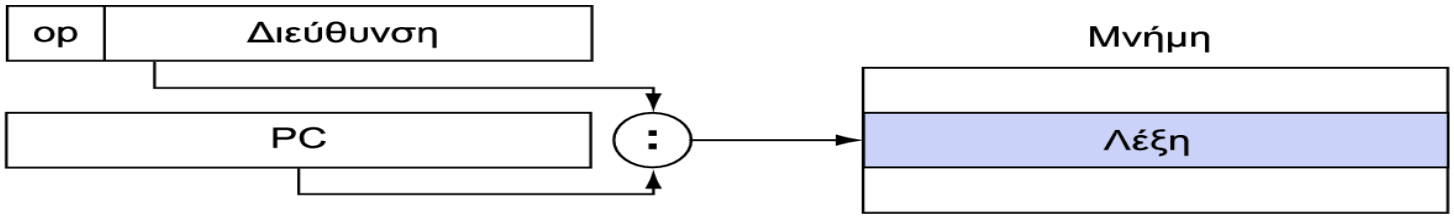
3. Διευθυνσιοδότηση βάσης



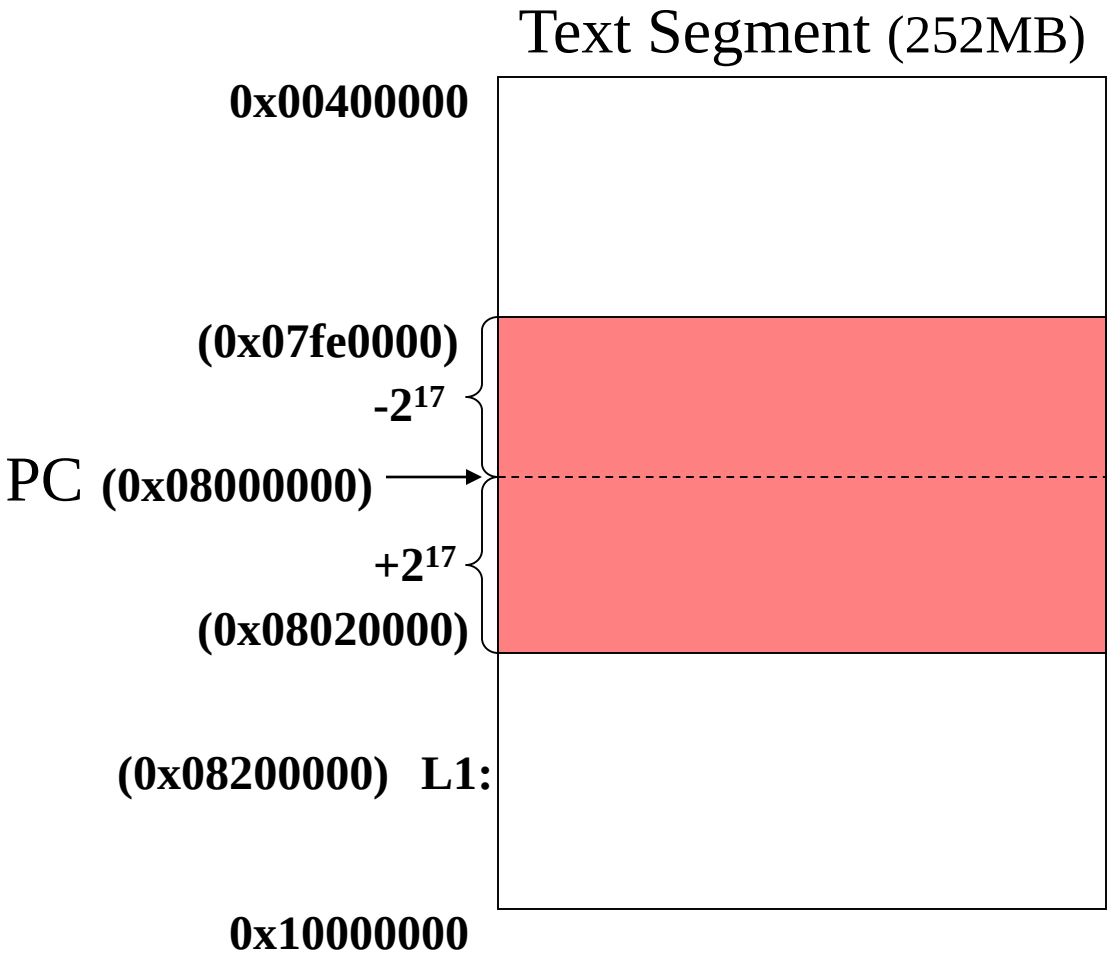
4. Σχετική διευθυνσιοδότηση ως προς PC



5. Ψευδο-απευθείας διευθυνσιοδότηση



Παράδειγμα : Απομακρυσμένες Διευθύνσεις




```
    beq  $s0, $s1, L1
```

↓

```
    bne  $s0, $s1, L2
    j    L1
L2:
```

Δείκτες (Pointers)

- Pointer: Μια μεταβλητή, η οποία περιέχει τη διεύθυνση μιας άλλης μεταβλητής
  - Αποτελεί τη HLL έκφραση της διεύθυνσης μνήμης σε γλώσσα μηχανής
- Γιατί χρησιμοποιούμε δείκτες;
  - Κάποιες φορές είναι ο μοναδικός τρόπος για να εκφράσουμε κάποιο υπολογισμό
  - Πιο αποδοτικός και συμπτυγμένος κώδικας
- Σημεία προσοχής όταν χρησιμοποιούμε δείκτες;
  - Πιθανώς η μεγαλύτερη πηγή bugs

  1) Dangling reference (λόγω πρώιμης απελευθέρωσης)

  2) Memory leaks (tardy free):
     - Αποτρέπουν την ύπαρξη διεργασιών που τρέχουν για μεγάλα χρονικά διαστήματα μιας και απαιτούν την επανέναρξη τους

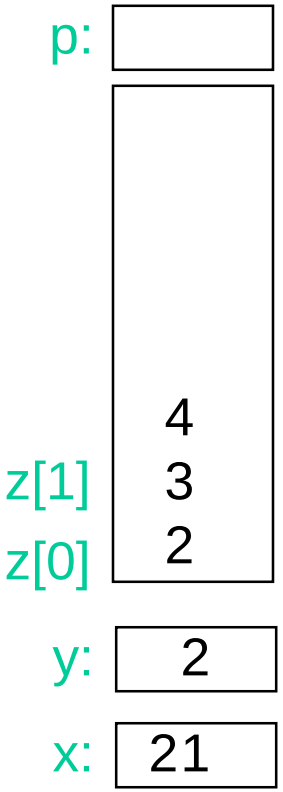
C Pointer Operators

- Έστω ότι η μεταβλητή c έχει την τιμή 100 και βρίσκεται στη θέση μνήμης 0x10000000
- Unary operator & → δίνει τη διεύθυνση:
  p = &c; gives address of c to p;
  - p "points to" c (p == 0x10000000)
- Unary operator * → δίνει την τιμή στην οποία δείχνει ο pointer
  - if p = &c => * p == 100 (Dereferencing a pointer)
- Dereferencing → data transfer in assembler
  - ... = ... *p ...; → load
    (get value from location pointed to by p)
  - *p = ...; → store
    (put value into location pointed to by p)

Pointer Arithmetic

```
int x = 1,  y = 2;  /* x and y are integer variables */
int z[10];          /* an array of 10 ints, z points to start */
int *p;             /* p is a pointer to an int */

x = 21;             /* assigns x the new value 21 */
z[0] = 2; z[1] = 3  /* assigns 2 to the first, 3 to the next */
p = &z[0];          /* p refers to the first element of z */
p = z;              /* same thing;  p[i] == z[i]*/
p = p+1;            /* now it points to the next element, z[1] */
p++;                /* now it points to the one after that, z[2] */
*p = 4;             /* assigns 4 to there, z[2] == 4*/
p = 3;              /* bad idea! Absolute address! Compiler gives a
                    warning*/
p = &x;             /* p points to x, *p == 21  */
z = &y;             /*illegal! array name is not a variable*/
z++;                /*illegal for the same reason*/
```



Constants – Constant reference

A reference to a variable (here int), which is constant. We pass the variable as a reference mainly, because references are smaller in size than the actual value, but there is a side effect and that is because it is like an alias to the actual variable. We may accidentally change the main variable through our full access to the alias, so we make it constant to prevent this side effect.

```
int var0 = 0;
const int * ptr1 = & var0;    //const int & ptr1 = var0; in c++
*ptr1 = 8; // Error          //ptr1=8; in c++
var0 = 6; // OK
```

Constants – Constant pointers

Once a constant pointer points to a variable then it cannot point to any other variable.

```
int var1 = 1;
int var2 = 0;

int *const ptr2 = &var1;
ptr2 = &var2; // Error
```

Constants – Pointer to constant

A pointer through which one cannot change the value of a variable it points is known as a pointer to constant.

```
int const * ptr3 = &var2;
 *ptr3 = 4; // Error
```

Constants – Constant pointer to a constant

A constant pointer to a constant is a pointer that can neither change the address it's pointing to and nor can it change the value kept at that address.

```
int var3 = 0;
int var4 = 0;
const int * const ptr4 = &var3;
*ptr4 = 1;      // Error
 ptr4 = &var4; // Error
```

What's the difference between `const int* p`, `int * const p` and `const int * const p`?

You have to read pointer declarations right-to-left.

`const int * p` means "p is a pointer to a constant integer" — that is, you can change the pointer, you cannot change the object where it points to.

`int * const p` means "**p is a constant pointer** to an integer" — that is, you can change the integer via p, but you can't change the pointer p itself.

`const int* const p` means "**p is a const pointer** to a **const int**" — that is, you can't change the pointer p itself, nor can you change the integer via p.

Let's.. play!

`int*` - pointer to int

`int const *` - pointer to const int

`int * const` - const pointer to int

`int const * const` - const pointer to const int

Now the first `const` can be on either side of the type so:

```
const int * == int const *
const int * const == int const * const
```

`int **` - pointer to pointer to int

`int ** const` - a const pointer to a pointer to an int

`int * const *` - a pointer to a const pointer to an int

`int const **` - a pointer to a pointer to a const int

`int * const * const` - a const pointer to a const pointer to an int

int const or const int ?

Η σειρά του τύπου και των qualifiers/specifiers στις C/C++ δεν έχει σημασία. Δηλαδή, όλα τα παρακάτω είναι ισοδύναμα:

- `const volatile unsigned long int`
- `volatile unsigned const int long`
- `unsigned int volatile long const`

Assembly Code : Παράδειγμα (1)

Έστω ακέραιος c με τιμή 100 που βρίσκεται στη θέση μνήμης 0x10000000, p στον $a0 και x στον $s0

1. **p = &c; /* p gets 0x10000000*/**
   lui $a0,0x1000 # p = 0x10000000
2. **x = *p; /* x gets 100 */**
   lw $s0, 0($a0) # dereferencing p
3. ***p = 200; /* c gets 200 */**
   addi $t0,$0,200
   sw $t0, 0($a0) # dereferencing p

Assembly Code : Παράδειγμα (2)

```
int strlen(char *s) {
    char *p = s;            /* p points to chars */
    while (*p != '\0')
        p++;                        /* points to next char */
    return p - s;           /* end - start */
}
```

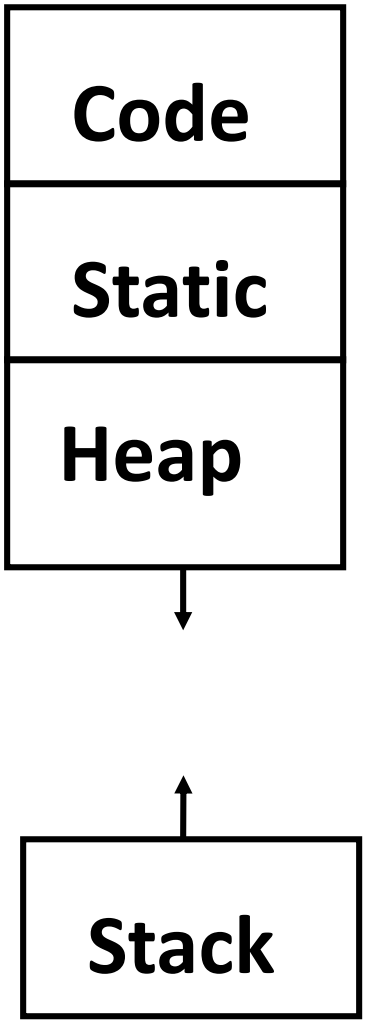
```
        mov  $t0,$a0
        lbu  $t1,0($t0) /* derefence p */
        beq  $t1,$zero, Exit
Loop:   addi $t0,$t0,1  /* p++ */
        lbu  $t1,0($t0) /* derefence p */
        bne  $t1,$zero, Loop
Exit:   sub  $v0,$t0,$a0
        jr   $ra
```

Επικοινωνία Ορισμάτων (Argument Passing Options)

- 2 μέθοδοι
  - Κλήση κατά τιμή (Call by Value): Ένα αντίγραφο του αντικειμένου στέλνεται στη μέθοδο/διαδικασία
  - Κλήση κατά αναφορά (Call by Reference): Ένας pointer στο αντικείμενο στέλνεται στη μέθοδο/διαδικασία
- Οι μεταβλητές μήκους 1 λέξης στέλνονται κατά τιμή
- Τι γίνεται στην περίπτωση ενός πίνακα; π.χ. a[100]
  - Pascal (call by value): Αντιγράφει 100 λέξεις του a[] στη στοίβα
  - C (call by reference) : Περνά ένα pointer (1 word) που δείχνει στο a[] σε ένα καταχωρητή

Lifetime of Storage and Scope

- Αυτόματα (stack allocated)
  - Τοπικές μεταβλητές μιας μεθόδου/διαδικασίας
  - Δημιουργούνται κατά την κλήση και απελευθερώνονται κατά την επιστροφή
  - Scope = η μέθοδος/διαδικασία
- Heap allocated
  - Δημιουργούνται με malloc
  - Πρέπει να απελευθερώνονται με free
  - Αναφορές μέσω pointers
- External / static
  - Επιζούν για ολόκληρη την εκτέλεση του προγράμματος



Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες

- 4 εκδόσεις μιας μεθόδου/διαδικασίας η οποία προσθέτει 2 πίνακες και αποθηκεύει το άθροισμα σε ένα 3ο πίνακα (*sumarray*)
  1. Ο 3ος πίνακας στέλνεται στη μέθοδο
  2. Χρήση ενός τοπικού πίνακα (στη στοίβα) για το αποτέλεσμα και πέρασμα ενός δείκτη σε αυτόν
  3. Ο 3ος πίνακας τοποθετείται στο heap
  4. Ο 3ος πίνακας ορίζεται ως static

Σκοπός του παραδείγματος είναι να δείξουμε τη χρήση των C statements, των pointers και της αντίστοιχης memory allocation.

Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες : Version 1

```
int x[100], y[100], z[100];
sumarray(x, y, z);
```

- C calling convention :

```
sumarray(&x[0], &y[0], &z[0]);
```

- Στην πραγματικότητα περνάμε pointers στους πίνακες

```
addi $a0,$gp,0      # x[0] starts at $gp
addi $a1,$gp,400    # y[0] above x[100]
addi $a2,$gp,800    # z[0] above y[100]
jal  sumarray
```

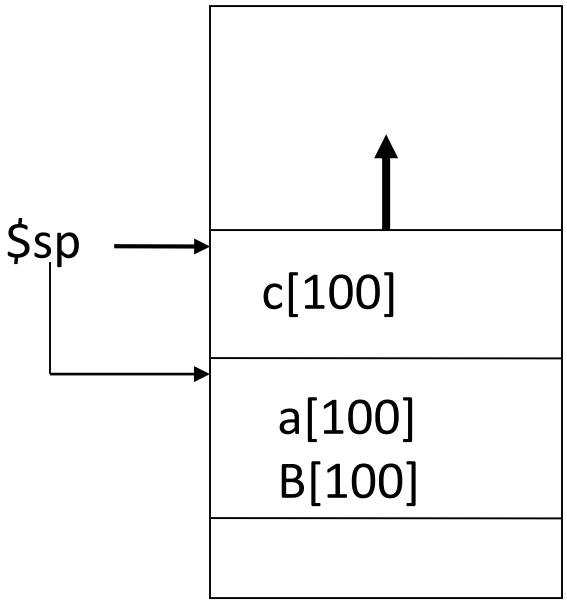
Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες : Version 1

```
void sumarray(int a[], int b[], int c[]) {
    int i;
    for(i = 0; i < 100; i = i + 1)
            c[i] = a[i] + b[i];
}
```

```
        addi    $t0,$a0,400     # beyond end of a[]
Loop:   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
```

Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες : Version 2

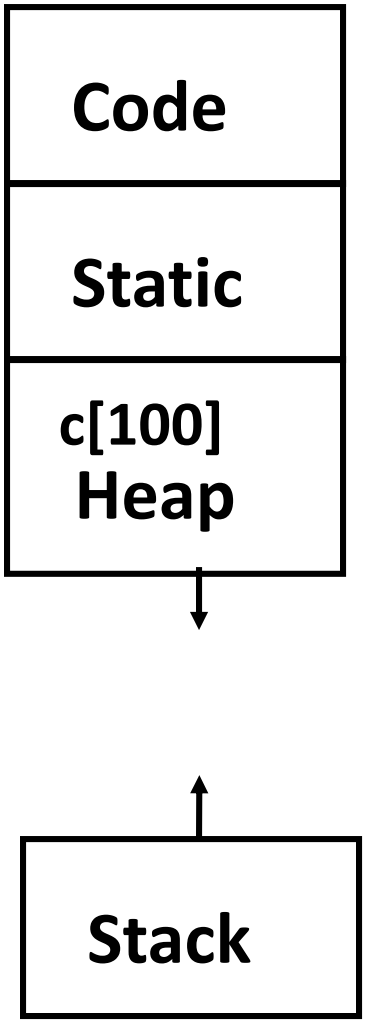
```
int *sumarray(int a[],int b[]) {
        int i, c[100];
        for(i=0;i<100;i=i+1)
            c[i] = a[i] + b[i];
        return c;
  }
```




```
        addi  $t0,$a0,400   # beyond end of a[]
        addi  $sp,$sp,-400  # space for c
        addi  $t3,$sp,0     # ptr for c
        addi  $v0,$t3,0     # $v0 = &c[0]
Loop:   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    $t1, 0($t3)   # c[i]=a[i] + b[i]
        addi  $a0,$a0,4     # $a0++
        addi  $a1,$a1,4     # $a1++
        addi  $t3,$t3,4     # $t3++
        j     Loop
Exit:   addi  $sp,$sp, 400  # pop stack
        jr    $ra
```

Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες : Version 3

```
int * sumarray(int a[],int b[]) {
        int i;
        int *c;
        c = (int *) malloc(100);
        for(i=0;i<100;i=i+1)
           c[i] = a[i] + b[i];
        return c;
}
```



- Ο χώρος που δεσμεύτηκε δεν επαναχρησιμοποιείται, εκτός αν απελευθερωθεί (freed)
  - Είναι πιθανό να οδηγήσει σε memory leaks
  - Java, Scheme διαθέτουν garbage collectors για να επανακτούν ελεύθερο χώρο

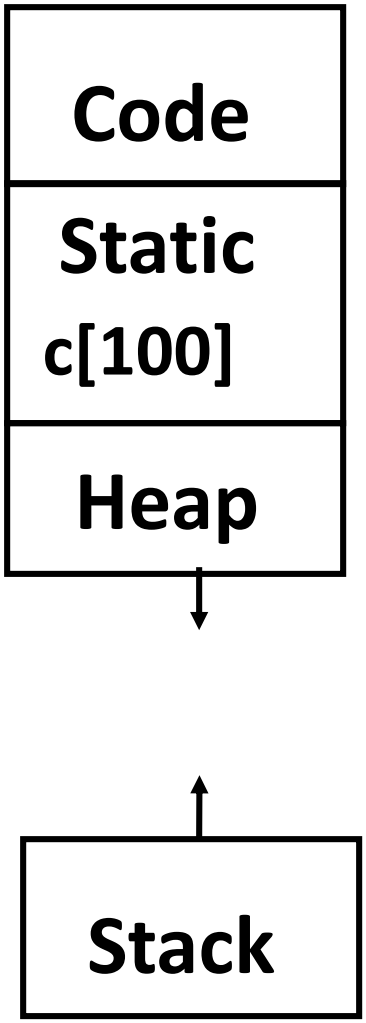
Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες : Version 3

```
            addi    $t0,$a0,400     # beyond end of a[]
            addi    $sp,$sp,-12     # space for regs
            sw      $ra, 0($sp)     # save $ra
            sw      $a0, 4($sp)     # save 1st arg.
            sw      $a1, 8($sp)     # save 2nd arg.
            addi    $a0,$zero,400
            jal     malloc
            addi    $t3,$v0,0       # ptr for c
            lw      $a0, 4($sp)     # restore 1st arg.
            lw      $a1, 8($sp)     # restore 2nd arg.
Loop:       beq     $a0,$t0,Exit
      ... (loop as before on prior slide )
            j       Loop
Exit:       lw      $ra, 0($sp)     # restore $ra
            addi    $sp, $sp, 12    # pop stack
            jr      $ra
```

Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες : Version 4

```
int * sumarray(int a[],int b[]) {
        int i;
        static int c[100];

        for(i=0;i<100;i=i+1)
                c[i] = a[i] + b[i];
        return c;
}
```



- Ο compiler δεσμεύει χώρο μια φορά για τη μέθοδο και ο χώρος επαναχρησιμοποιείται
  - Θα μεταβληθεί την επόμενη φορά που θα κληθεί η sumarray
  - Γιατί την αναφέρουμε; Χρησιμοποιείται στις C libraries!

Επανάληψη (1)

MIPS operands		
Name	**Example**	**Comments**
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	**Memory[0], Memory[4], ..., Memory[4294967292]**	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

Επανάληψη (2)

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]	Word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Συμπεράσματα

- Τα δεδομένα μπορεί να είναι οτιδήποτε
  - Datatyping περιορίζει τις μορφές των δεδομένων (data representations)
  - Οι εφαρμογές περιορίζουν το datatyping
- MIPS Datatypes: Number, String, Boolean
- Addressing: Pointers, Values
  - Πολλαπλοί τρόποι διευθυνσιοδότησης (direct, indirect,…)
  - Memory-based address storage (**jr** instruction)
- Πίνακες : *μεγάλα κομμάτια μνήμης*
  - Pointers vs stack storage
  - Προσοχή στα memory leaks!

Επανάληψη : Τρόποι Διευθυνσιοδότησης

Addr. mode	*Παράδειγμα*	*Έννοια*	*χρήση*
Register	add r4,r3	Regs[r4]← Regs[r4]+ Regs[r3]	a value is in register
Immediate	add r4,#3	Regs[r4]← Regs[r4]+3	for constants
Displacement	add r4,100(r1)	Regs[r4]← Regs[r4]+Mem[100+ Regs[r1]]	local variables
Reg. indirect	add r4,(r1)	Regs[r4]← Regs[r4]+Mem[Regs[r1]]	accessing using a pointer or comp. address
Indexed	add r4,(r1+r2)	Regs[r4]← Regs[r4]+Mem[Regs[r1]+ Regs[r2]]	array addressing (base +offset)
Direct	add r4,(1001)	Regs[r4]← Regs[r4]+Mem[1001]	addr. static data
Mem. Indirect	add r4,@(r3)	Regs[r4]← Regs[r4]+Mem[Mem[Regs[r3]]]	if R3 keeps the address of a pointer p, this yields *p
Autoincrement	add r4,(r3)+	Regs[r4]← Regs[r4]+Mem[Regs[r3]] Regs[r3]← Regs[r3]+d	stepping through arrays within a loop; d defines size of an element
Autodecrement	add r4,-(r3)	Regs[r3]← Regs[r3]-d Regs[r4]← Regs[r4]+Mem[Regs[r3]]	similar as previous
Scaled	add r4,100(r2)[r3]	Regs[r4]← Regs[r4]+ Mem[100+Regs[r2]+Regs[r3]*d]	to index arrays