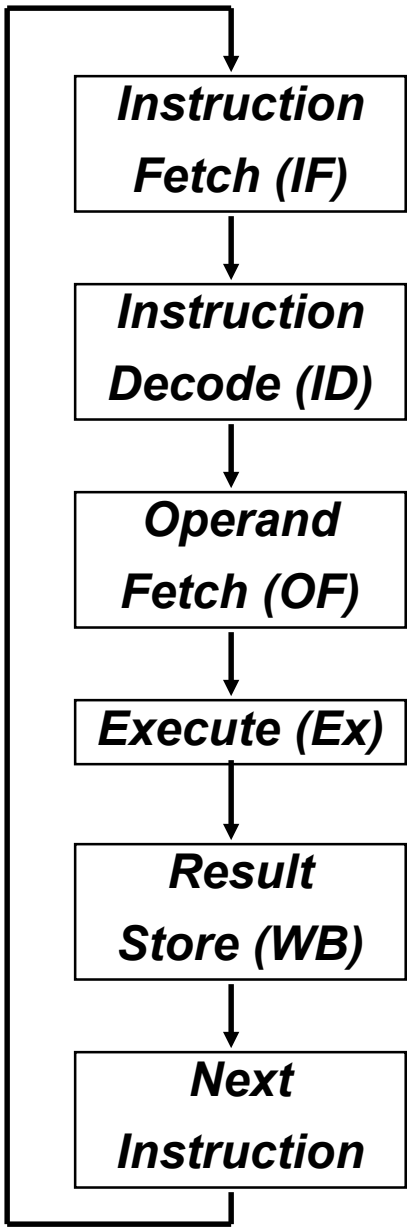
Αρχιτεκτονικές Συνόλου Εντολών



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

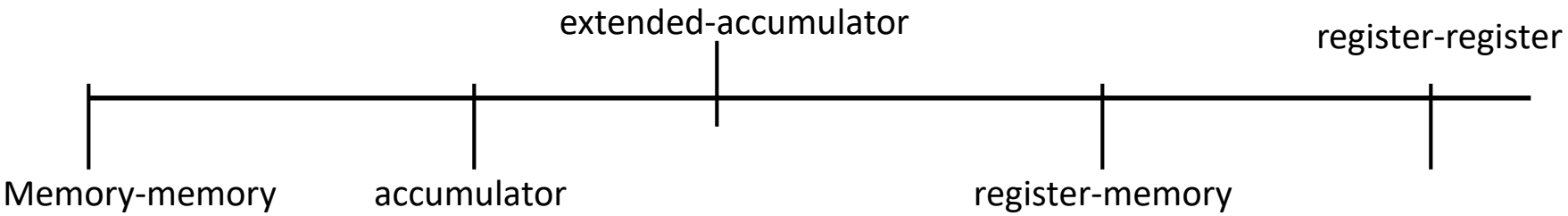
1. CISC

- 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

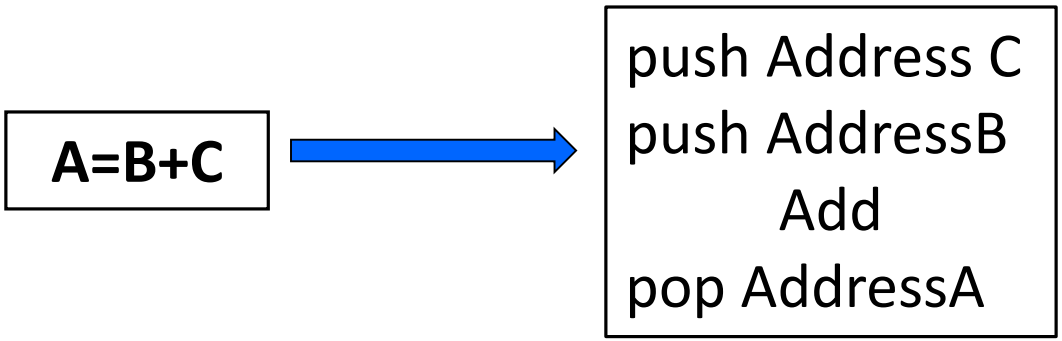


Αρχιτεκτονική Στοίβας

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



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)

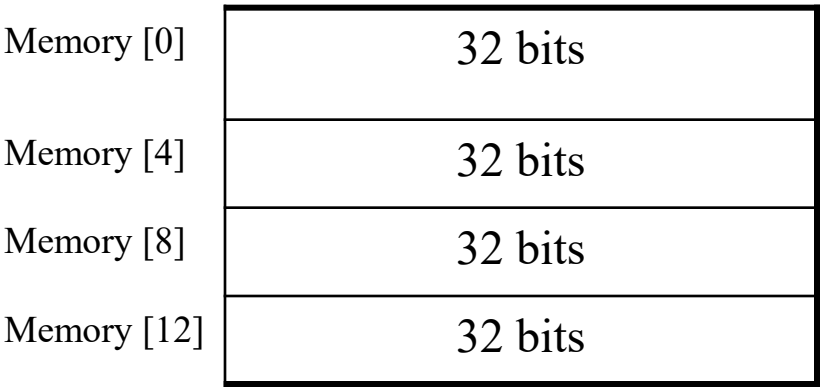
Γενικότητες? Θα τα δούμε στη συνέχεια.....

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

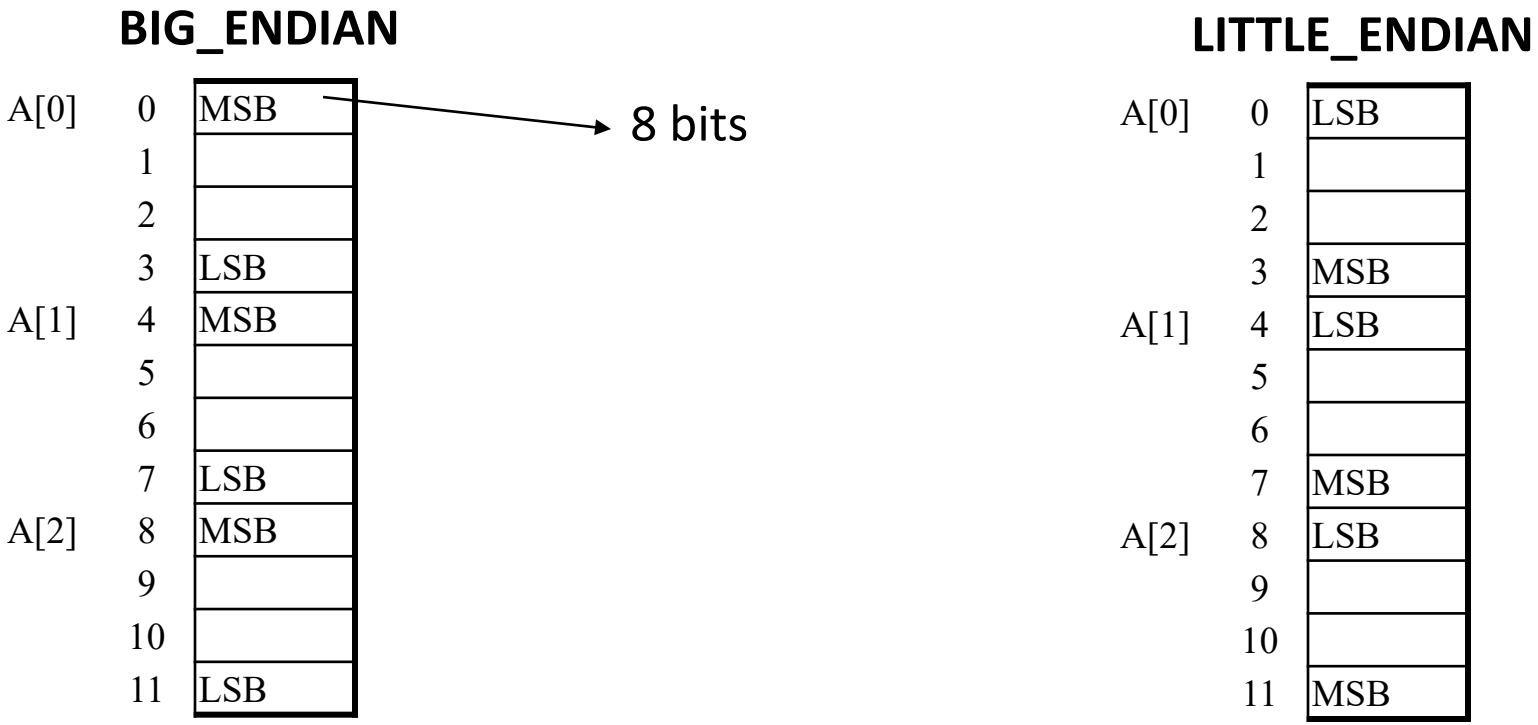
- Λέξεις των 32 bit
- Μνήμη οργανωμένη σε bytes
  - Κάθε byte είναι μια ξεχωριστή δνση
  - 2^{30} λέξεις μνήμης των 32 bits
  - Ακολουθεί το μοντέλο big Endian
- Register File
  - 32 καταχωρητές γενικού σκοπού
- Εντολές :
  - αποθήκευσης στη μνήμη (lw, sw)
  - αριθμητικές (add, sub κλπ)
  - διακλάδωσης (branch instructions)



Big Endian vs Little Endian

- **Big Endian:** Η δνση του **πιο σημαντικού** byte (MSB) είναι και **δ/νση** της λέξης – ***'big end' (leftmost) byte***
- **Little Endian:** Η δνση του **λιγότερο σημαντικού** byte (LSB) είναι και **δ/νση** της λέξης – ***'little end' (rightmost) byte***
- Η λέξη αποθηκεύεται πάντα σε συνεχόμενες θέσεις:

δνση, δνση+1, δνση+2,δνση+3



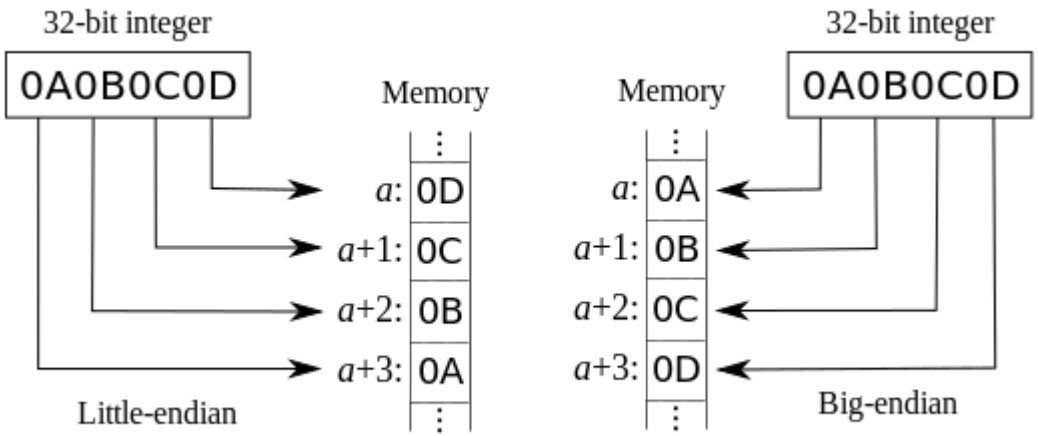
Big-endian (π.χ. IBM mainframes, RISC CPUs):

Το πιο αριστερά (leftmost) byte αποθηκεύεται πρώτο (στη μικρότερη διεύθυνση α), ακολουθούμενο από τα άλλα bytes, από αριστερά προς τα δεξιά (στις διευθύνσεις α+1, α+2, α+3). ("**Big end**" of number stored first).

Little-endian (π.χ. Intel processors):

Το πιο δεξιά (rightmost) byte αποθηκεύεται πρώτο (στη μικρότερη διεύθυνση α), ακολουθούμενο από τα άλλα bytes, από δεξιά προς τα αριστερά (στις διευθύνσεις α+1, α+2, α+3). ("**Little end**" of number stored first).

Έστω ότι έχουμε έναν 4-byte (32-bit) αριθμό: 0x0A0B0C0D (hex):



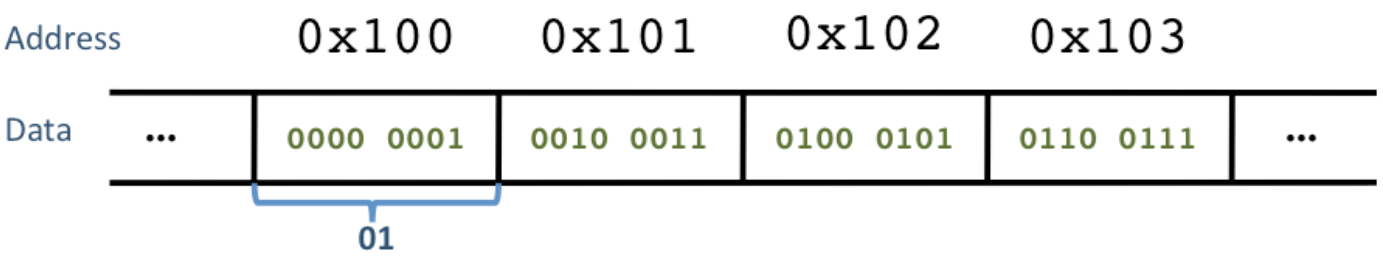
Suppose `x` is a 4-byte integer:

```
int x = 0x01234567;
```

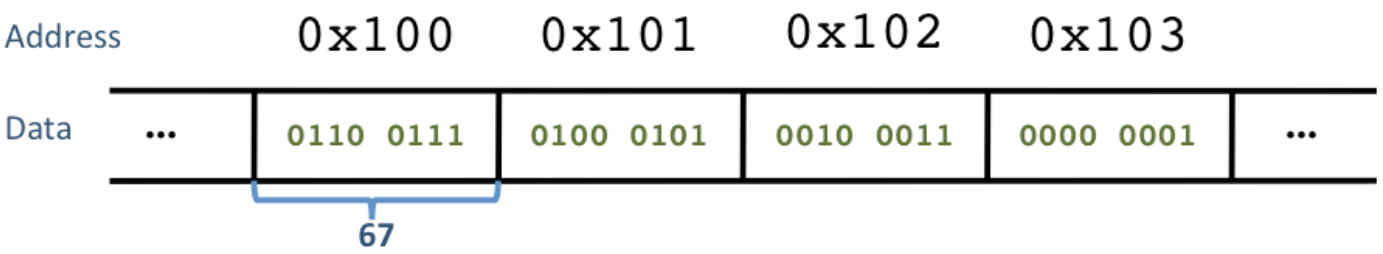
and the address of `x` is `0x100`.

Note: `x` in binary: `0000 0001 0010 0011 0100 0101 0110 0111`

Big endian byte order:



Little endian byte order:



MIPS ISA : Αριθμητικές λειτουργίες

- Πρόσθεση και αφαίρεση (add, sub)
  - Πάντα 3 ορίσματα – ΠΟΤΕ δ/νση 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 τελεστέους (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 ΠΑΝΤΑ ΣΤΗ ΜΝΗΜΗ.

- 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]

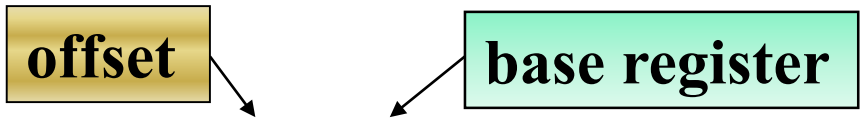
Παράδειγμα

- **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).

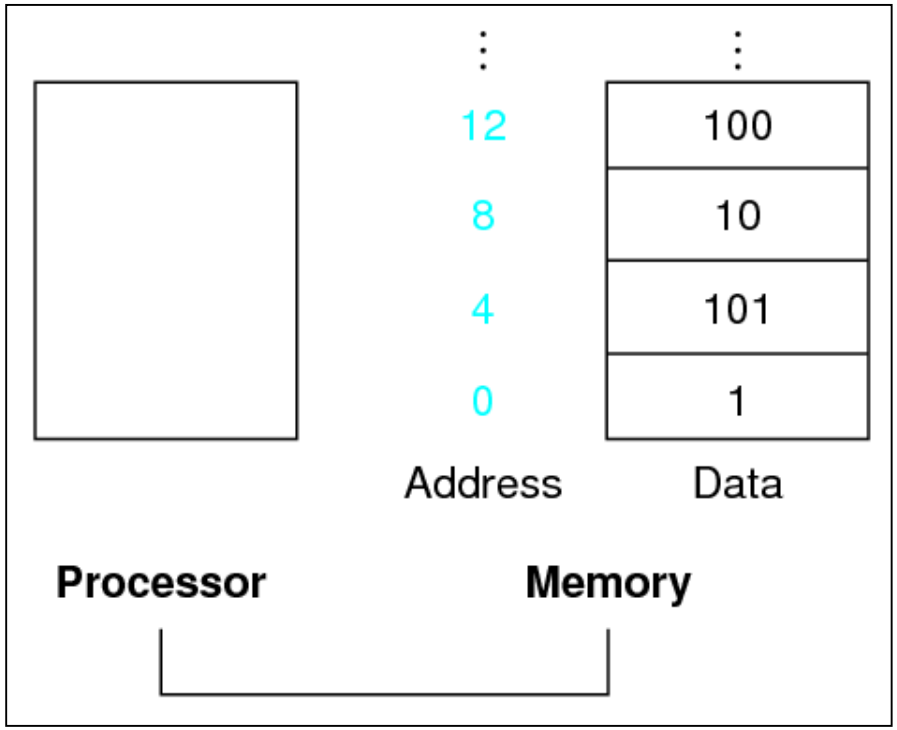


lw $t0, 32($s3)

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

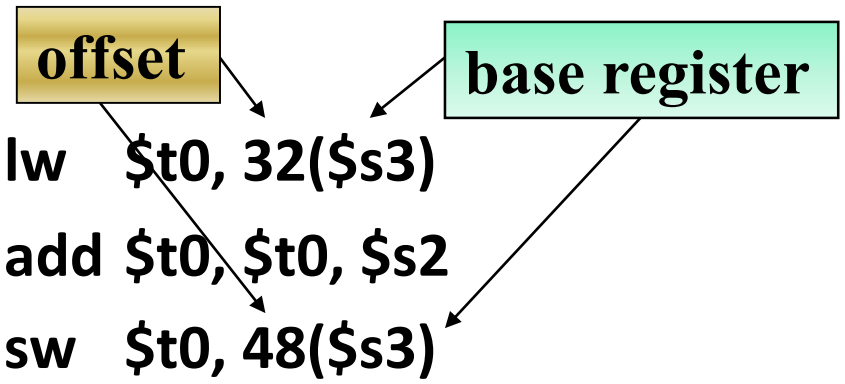


Παράδειγμα 2

- **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 (hardwired) 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!

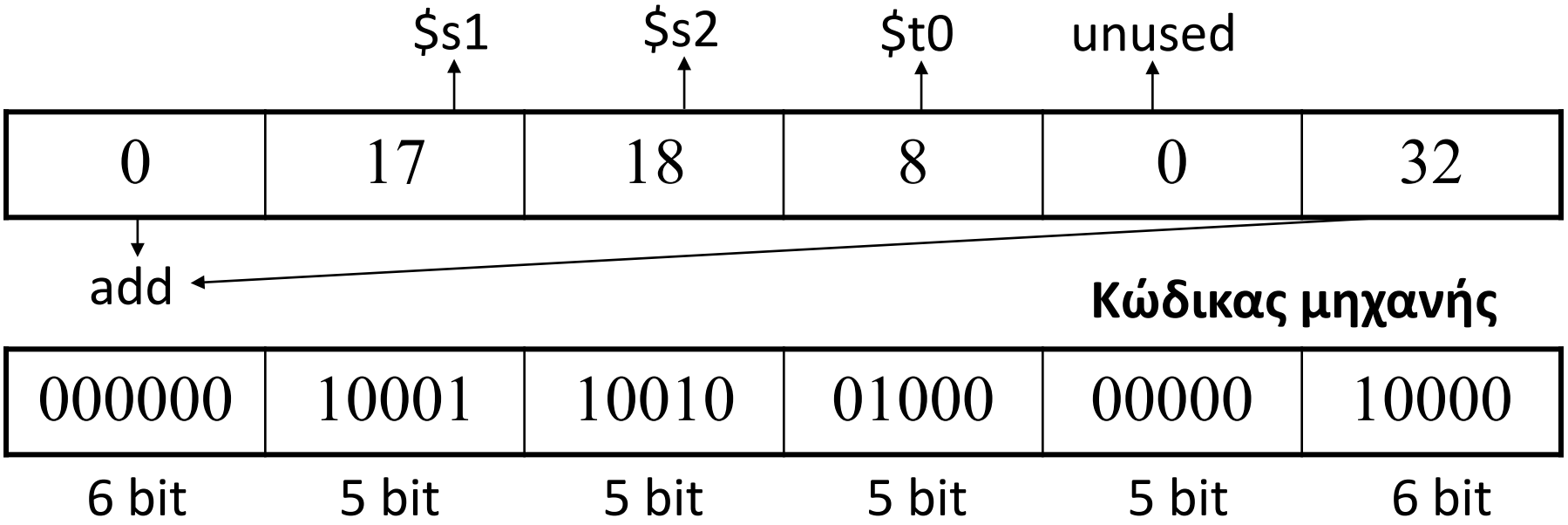
- Αριθμοί καταχωρητών
  - $t0 – $t7 είναι οι καταχωρητές 8 – 15
  - $t8 – $t9 είναι οι καταχωρητές 24 – 25
  - $s0 – $s7 είναι οι καταχωρητές 16 – 23

Αναπαράσταση Εντολών (2)

Συμβολική αναπαράσταση:

add $t0, $s1, $s2 → **Assembly**

Πώς την καταλαβαίνει ο MIPS?



Αναπαράσταση Εντολών (3)

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

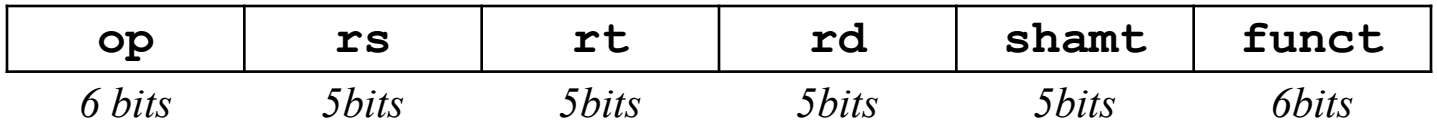
MIPS machine language

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 are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Μορφή Εντολής – Instruction Format

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

R-Type
(register type)



Op: opcode

rs,rt: register source operands

Rd: register destination operand

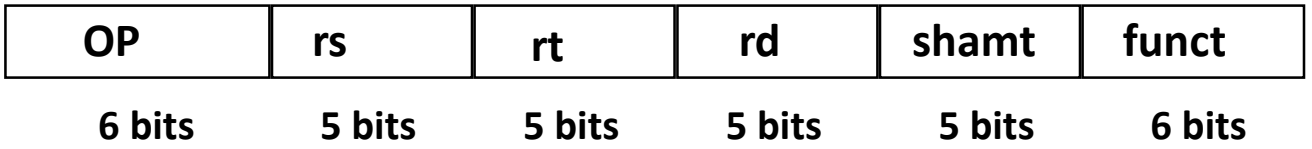
Shamt: shift amount

Funct: op specific (function code)

add $rd, $rs, $rt

MIPS R-Type (ALU)

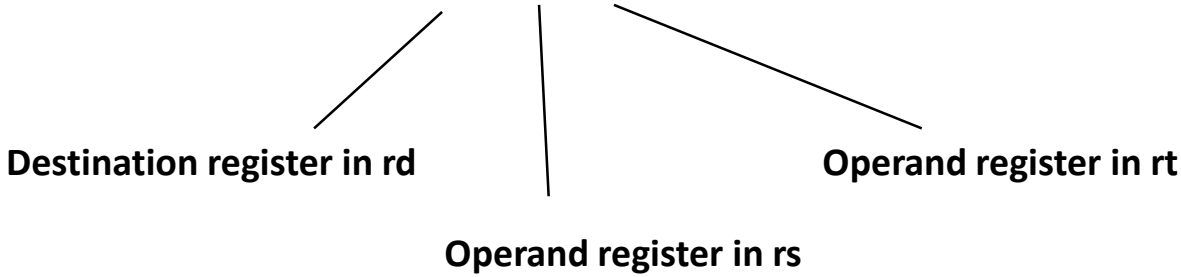
R-Type: Όλες οι εντολές της ALU που χρησιμοποιούν 3 καταχωρητές



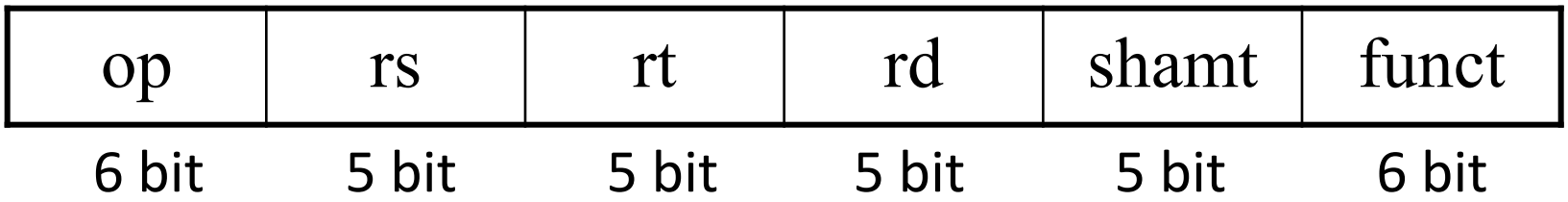
- Παραδείγματα :
  - add $1,$2,$3
  - sub $1,$2,$3

and $1,$2,$3

or $1,$2,$3



Αναπαράσταση Εντολών στον Υπολογιστή (R-Type)



Τι γίνεται με τη load?

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

lw $t1, 8000($s3)

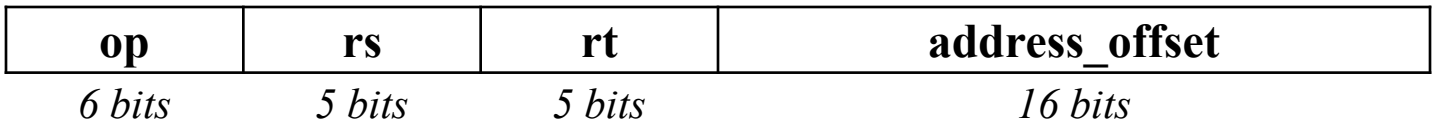
σε ποιο πεδίο χωράει;

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:



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

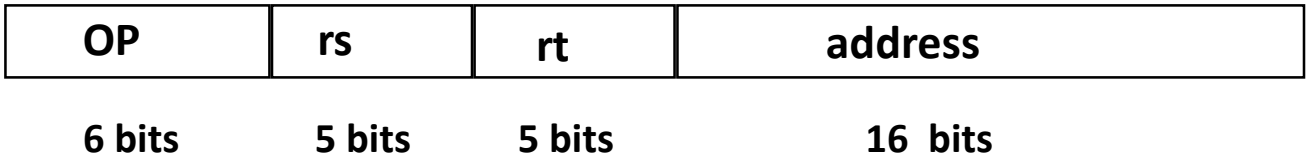
I-format

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

(υπενθύμιση: δεξαεξαδικό σύστημα-hexadecimal)

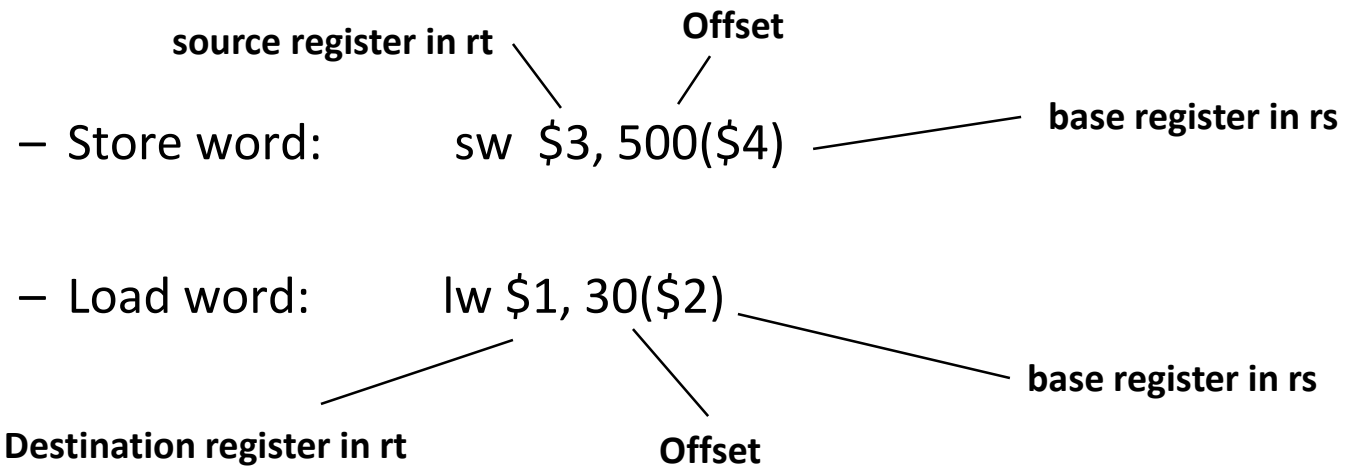
Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}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

MIPS I-Type : Load/Store



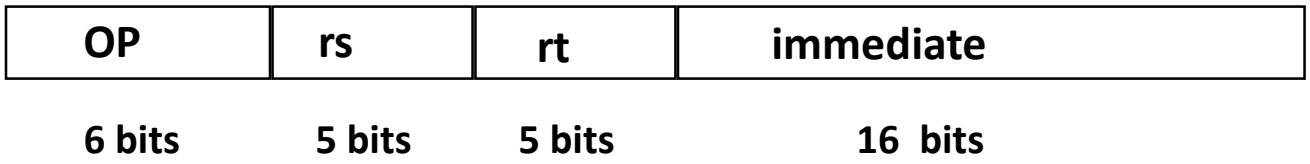
– *address: 16-bit memory address offset in bytes added to base register.*

- Παραδείγματα :



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.



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

- Παραδείγματα :
  - add immediate: addi $1,$2,100
  - and immediate andi $1,$2,10

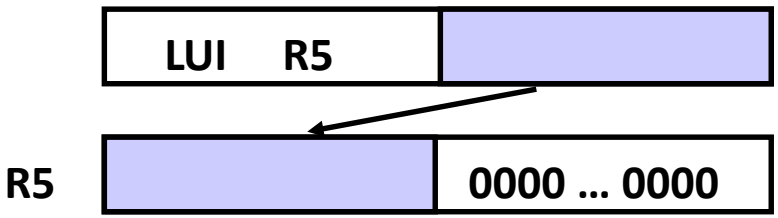
Result register in rt

Source operand register in rs

Constant operand in immediate

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)



The machine language version of `lui $t0, 255` `# $t0 is register 8:`

001111	00000	01000	0000 0000 1111 1111

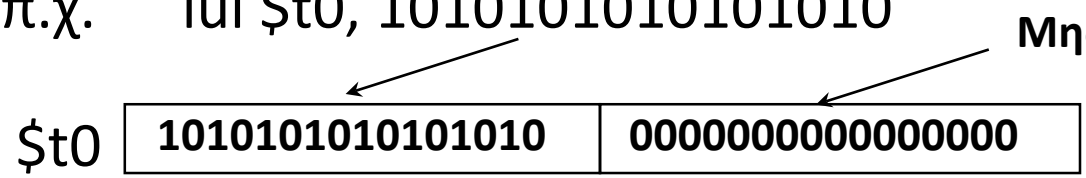
Contents of register `$t0` after executing `lui $t0, 255:`

0000 0000 1111 1111	0000 0000 0000 0000

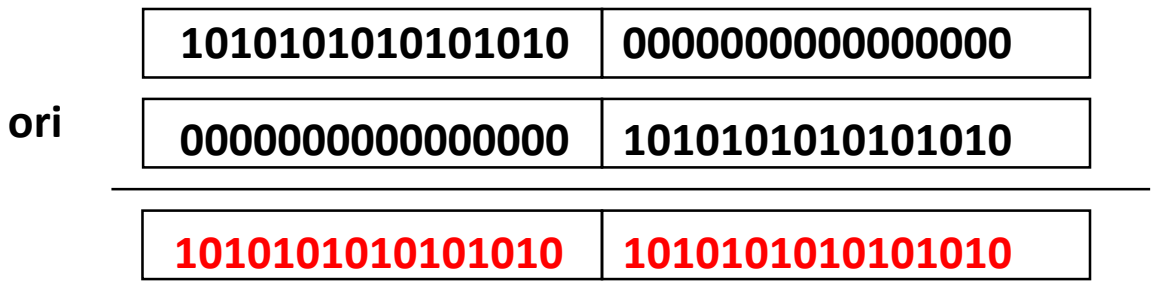
MIPS data transfer instructions : Παραδείγματα (2)

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



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

beq, bne

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;

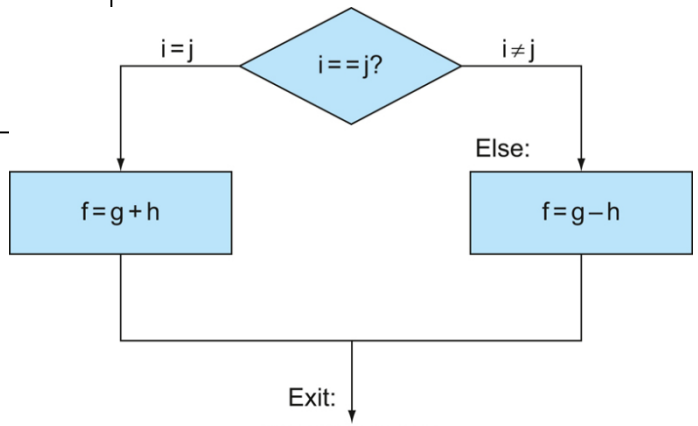
με f, g, h, i, j αντιστοιχούνται σε $s0, ..., $s4

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)

Συγκρίσεις

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*

Signed vs unsigned comparison

$s0: 1111 1111 1111 1111 1111 1111 1111 1111_{two}

$s1: 0000 0000 0000 0000 0000 0000 0000 0001_{two}

```
1. slt    $t0, $s0, $s1  #signed comparison
2. sltu   $t1, $s0, $s1  #unsigned comparison
```

Απάντηση:

1. $s0 = -1_{ten} (signed int)
   $t0 = 1 (since -1ten < 1 ten

2. $s0 = $4,294,967,295_{ten}$ (2^{32}-1)(unsigned int)
   $t1 = 0 (since $4,294,967,295_{ten} > 1_{ten}$)

Bounds check shortcut

Index-out-of-bounds check

Jump to `IndexoutofBounds` if `$s1` $\leq$ `$t2` or if `$s1` is negative:

```
sltu $t0, $s1, $t2 # $t0=0 if $s1≥length or $s1<0

Beq $t0, $zero, IndexoutofBounds #if bad, foto Error
```

Treating **signed** numbers as **unsigned** gives us a low cost way of checking if 0≤x<y, needed for Index-out-of-bounds check

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

– *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

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

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

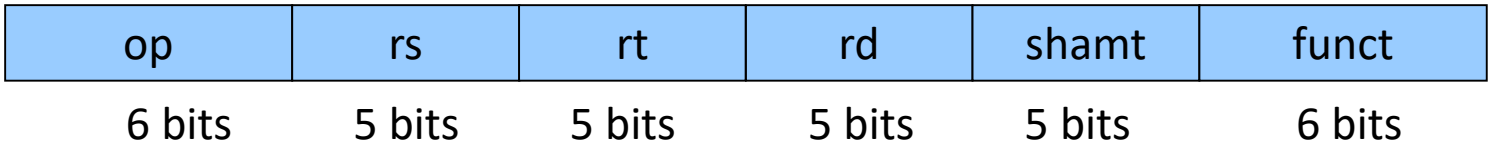
final jump memory address in bytes is calculated from {jump target} x 4

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 (e.g. jump address table for `case/switch` statements, jr $ra)
- Jump and Link (*J-type*):
  - jal 10000 # jump to 10000 and save PC in R31 ($ra)
  - 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.

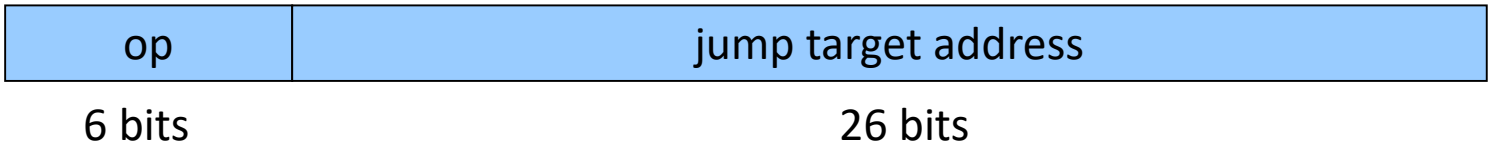
Σύνοψη – MIPS Instruction Formats

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



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

- J-type (j, jal)



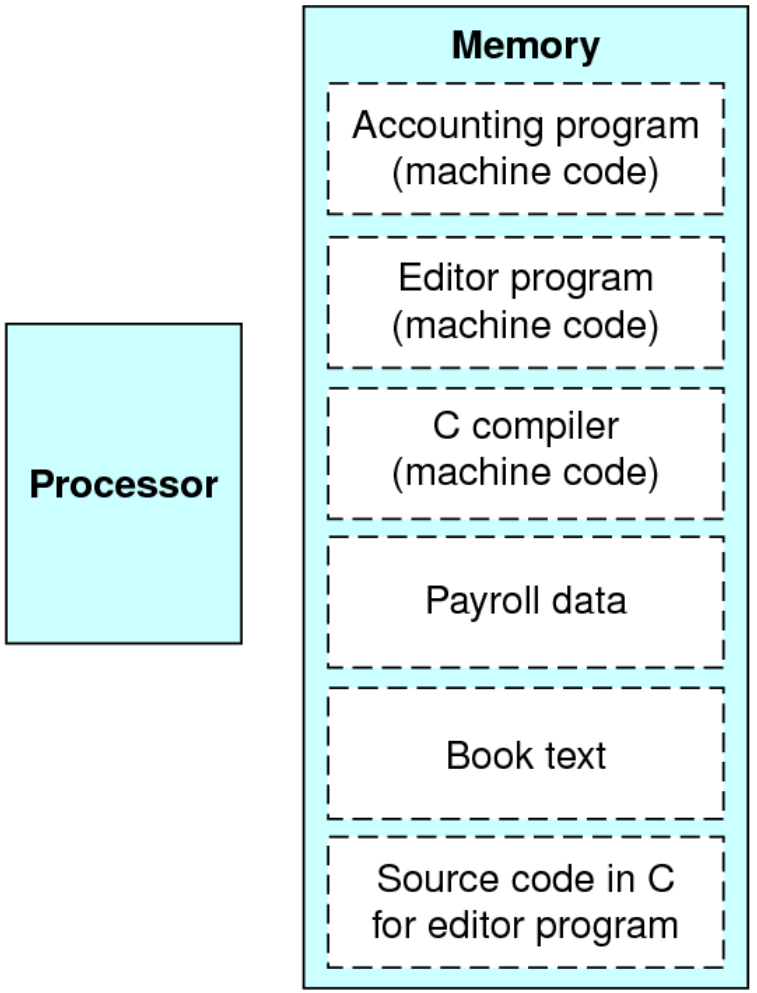
Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Έννοια αποθηκευμένου προγράμματος

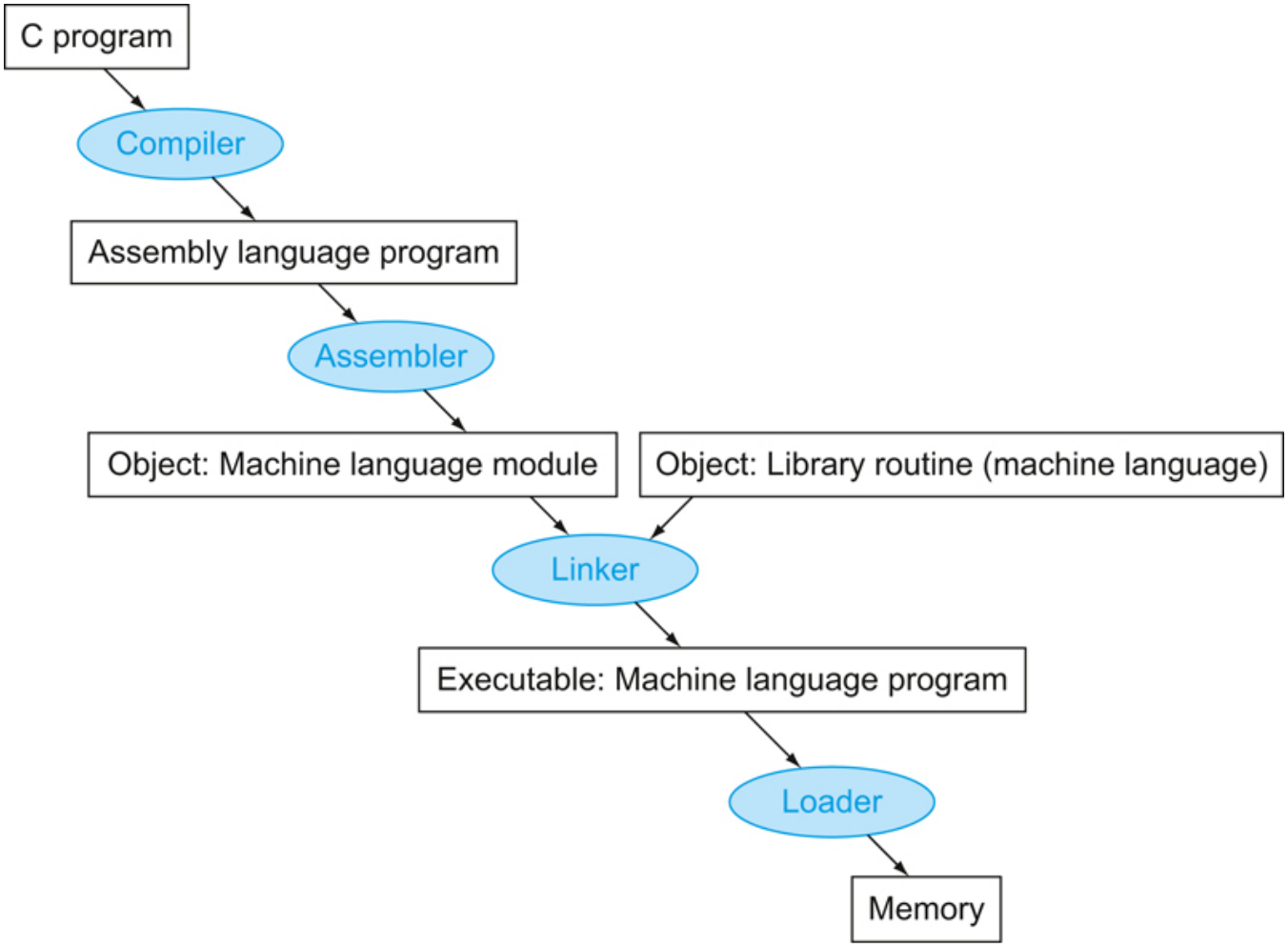
Ο υπολογιστής κάνει πολλές εργασίες φορτώνοντας δεδομένα στο μνήμη.

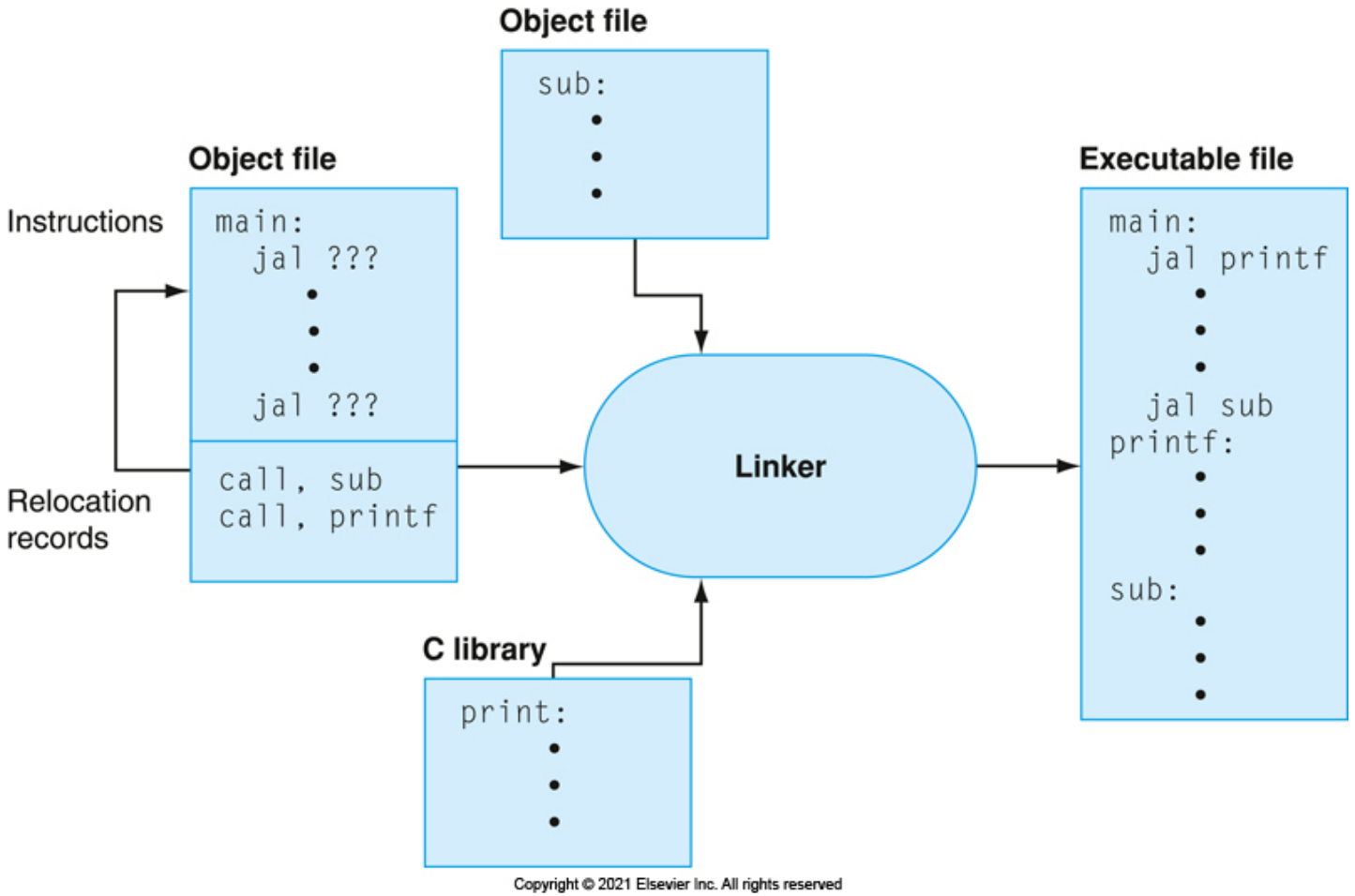
Δεδομένα και εντολές είναι στοιχεία στη μνήμη.

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

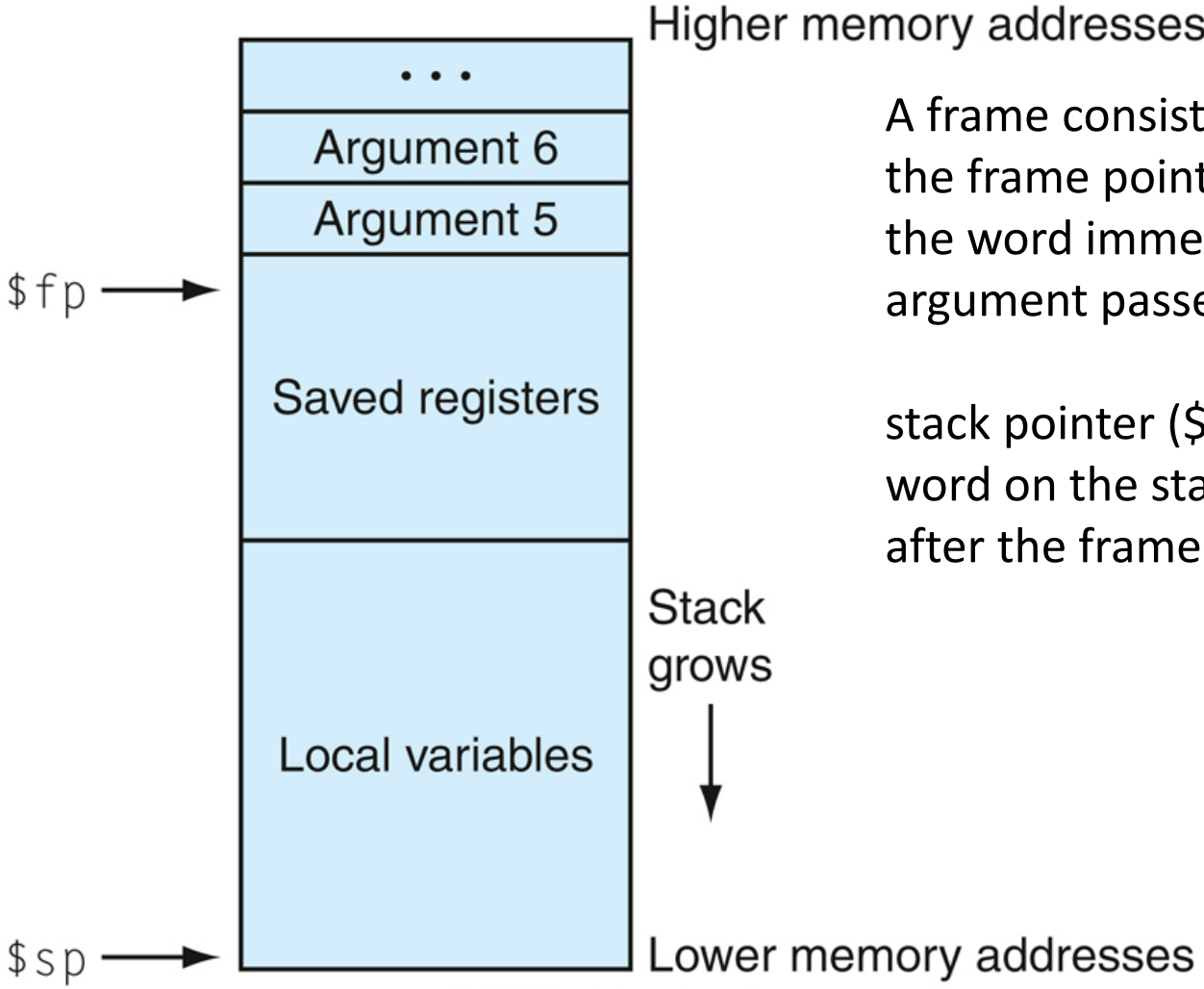


Translation hierarchy for C





Layout of a stack frame



A frame consists of the memory between the frame pointer ($fp), which points to the word immediately after the last argument passed on the stack.

stack pointer ($sp) points to the first free word on the stack-initially the first word after the frame pointer.

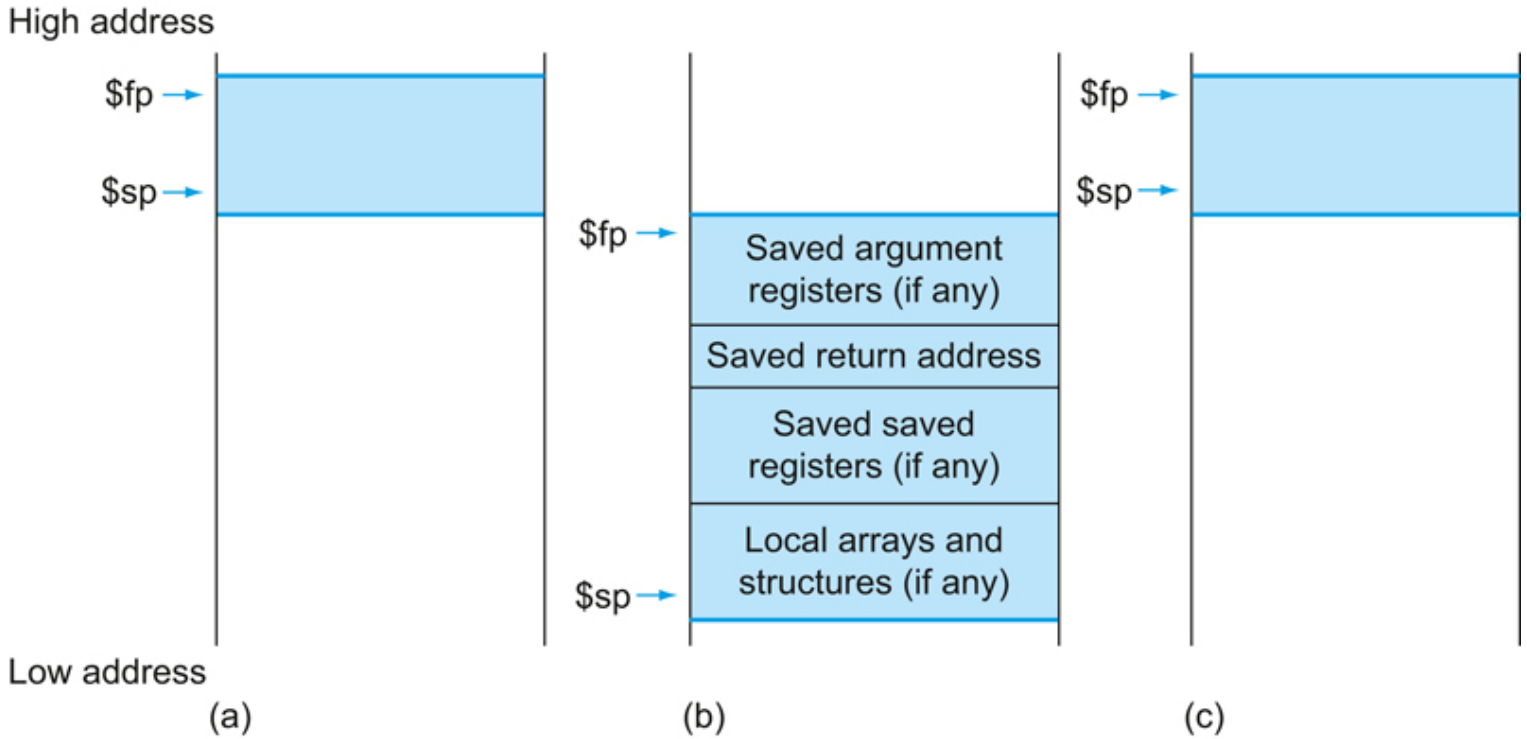
The first four arguments are passed in registers $a0-$a3 (though simpler compilers may choose to ignore this convention and pass all arguments via the stack).

The remaining arguments are pushed on the stack.

- The stack grows down from higher memory addresses
- The frame pointer is above stack pointer.

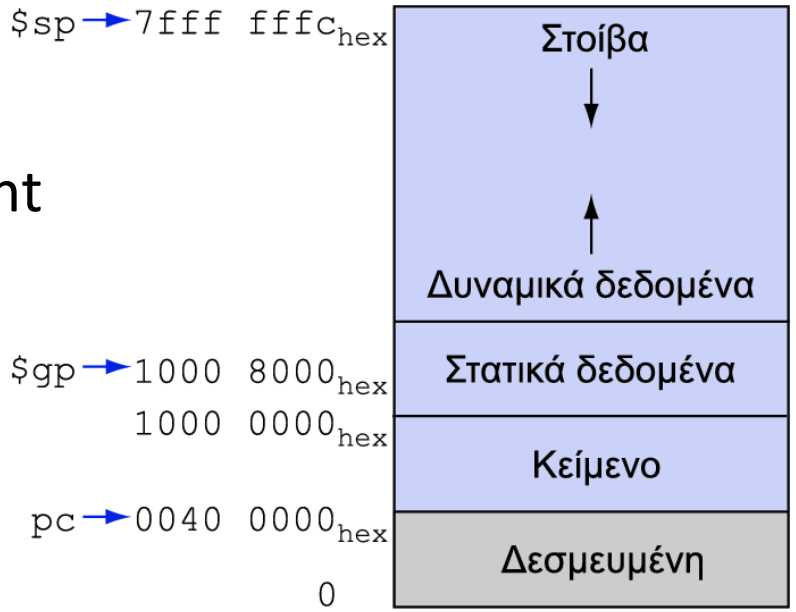
source: SPIM S20: A MIPS R2000 Simulator, *James R. Larus - larus@cs.wisc.edu*

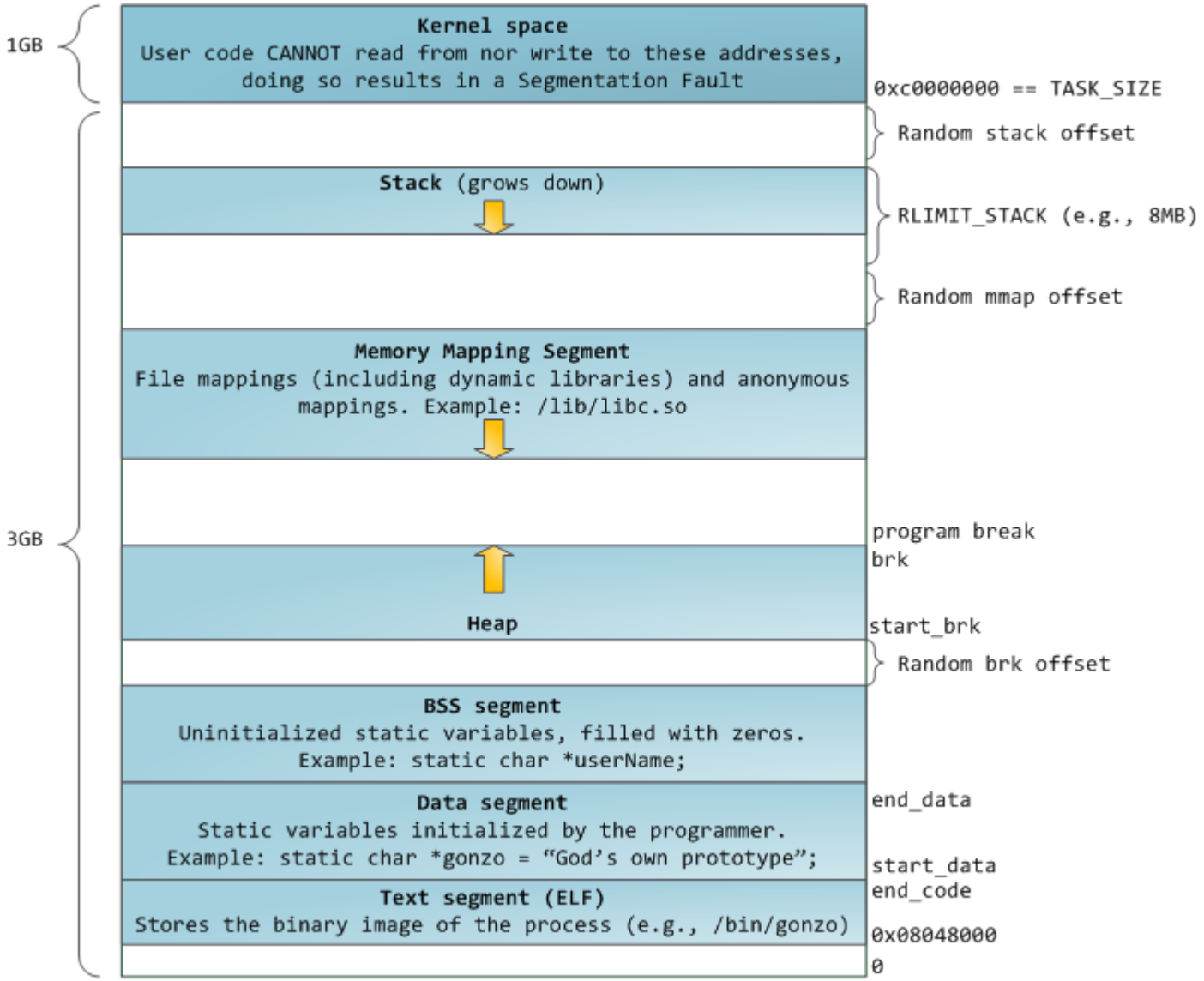
frame pointer vs stack pointer



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





Source: Gustavo's Duarte blog https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

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

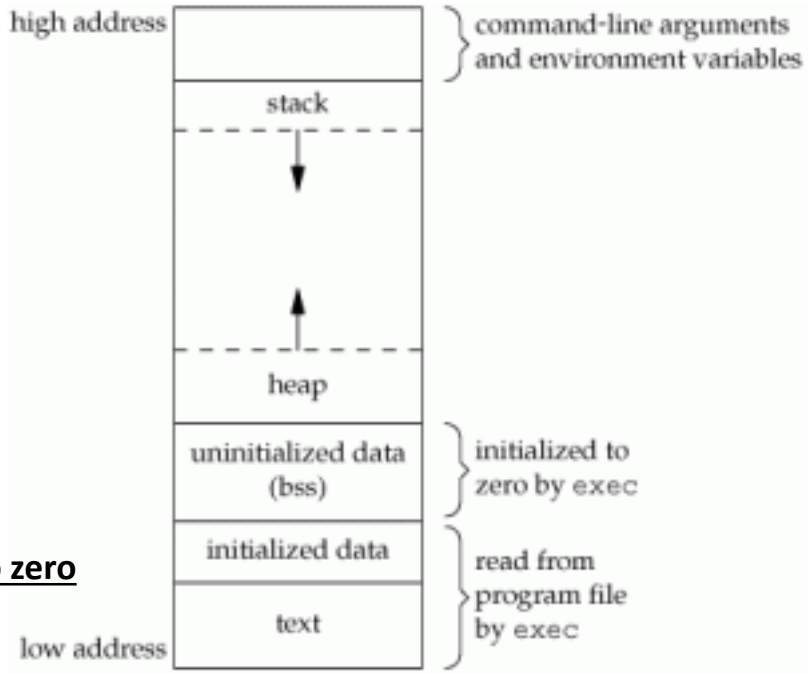
   const char * string ="hello world";

   "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**.

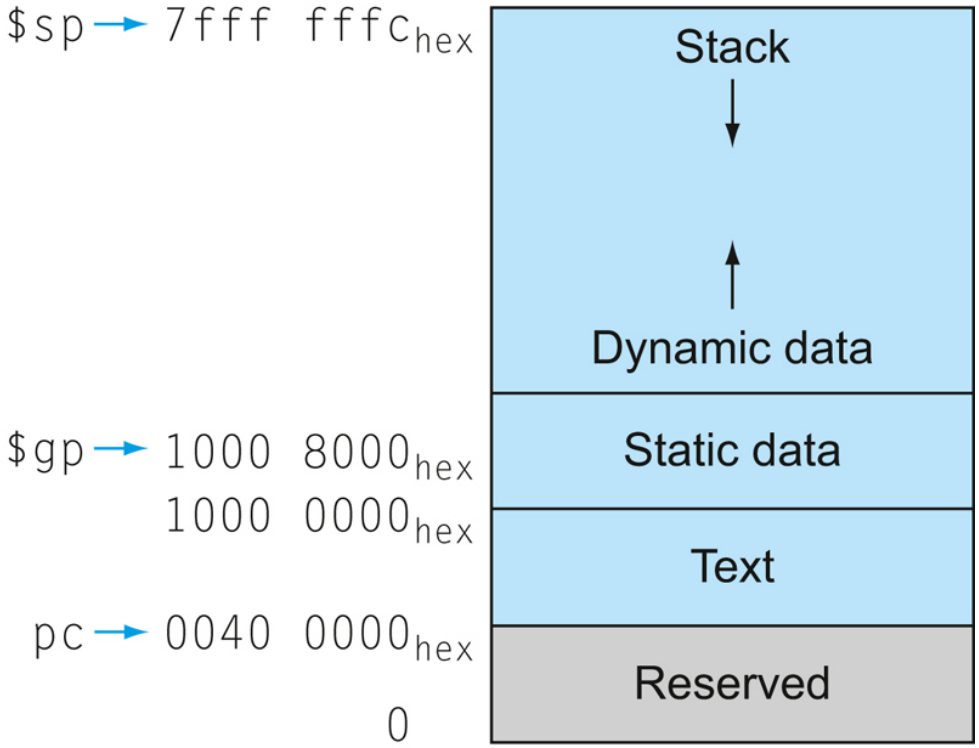
   int j ; static int i;

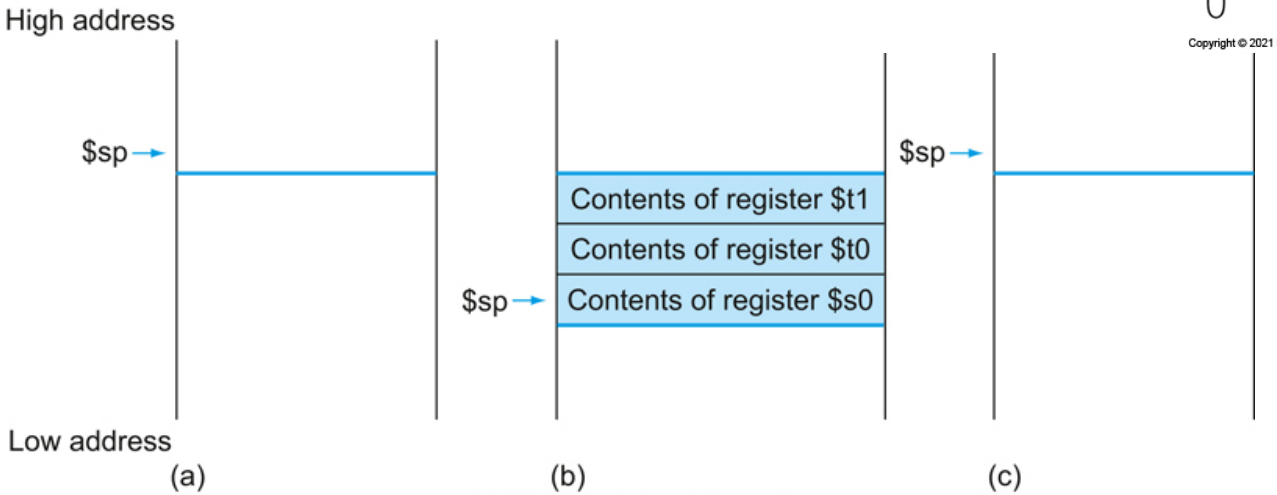
4. Stack

   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

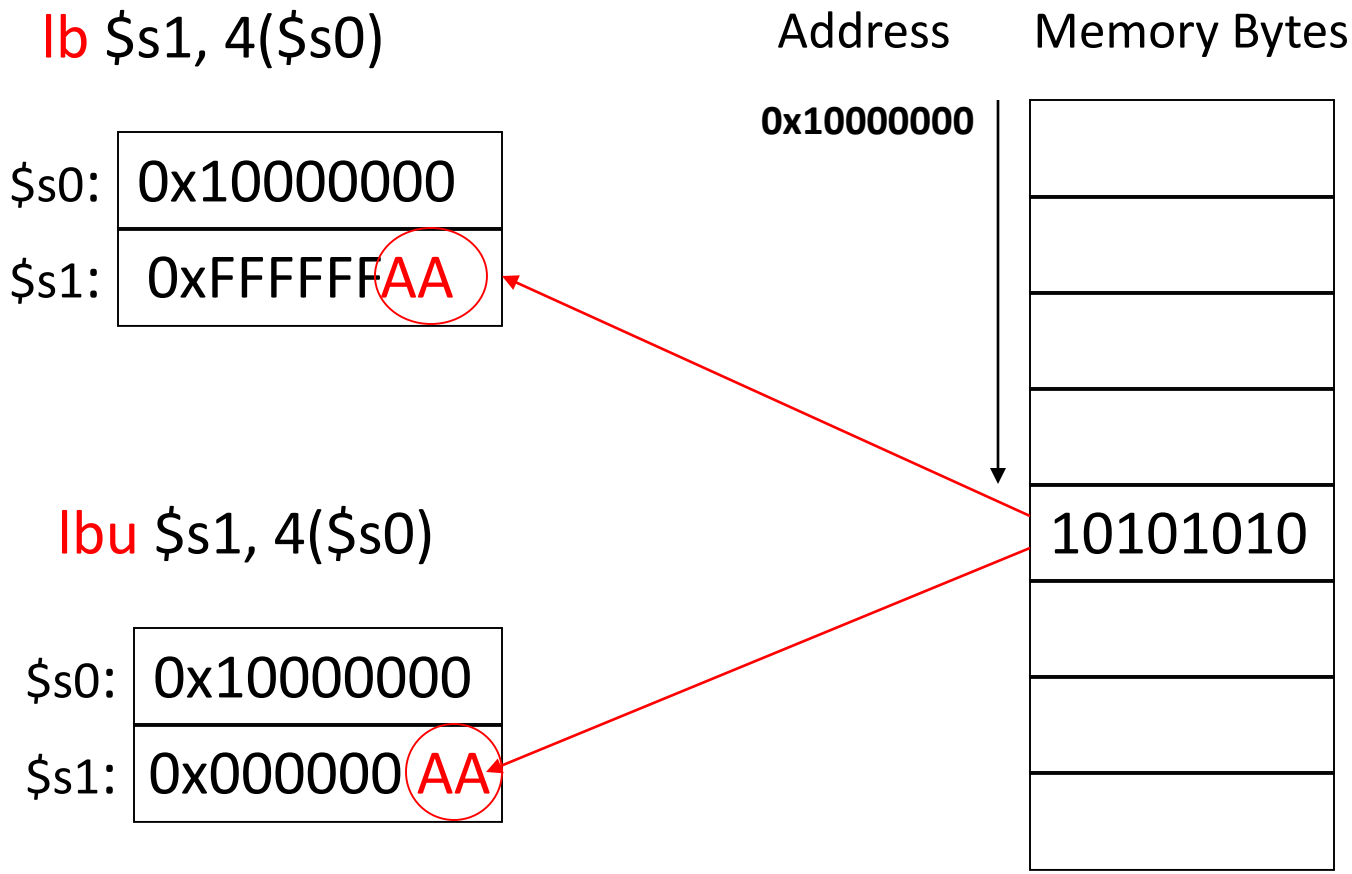
Τύποι Δεδομένων : MIPS (1)

- 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



Παράδειγμα : Αντιγραφή String

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

C convention:

Null byte (00000000)
represents end of the string

Importance of comments in MIPS!

```
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
```

Σταθερές

- Συχνά χρησιμοποιούνται μικρές σταθερές (50% των τελεστών)
  - e.g., A = A + 5;
- Λύση
  - Αποθήκευση ‘τυπικών σταθερών’ στη μνήμη και φόρτωση τους.
  - Δημιουργία 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

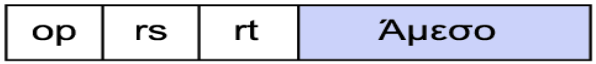
- Διευθυνσιοδότηση μέσω καταχωρητή (Register addressing)
  - Η πιο συνηθισμένη (σύντομη και ταχύτατη)
  - add $3, $2, $1
- Διευθυνσιοδότηση βάσης (Base addressing)
  - Ο τελεστέος είναι σε μια θέση μνήμης με κάποιο offset
  - lw $t0, 20 ($t1)
- Άμεση διευθυνσιοδότηση (Immediate addressing)
  - Ο τελεστέος 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

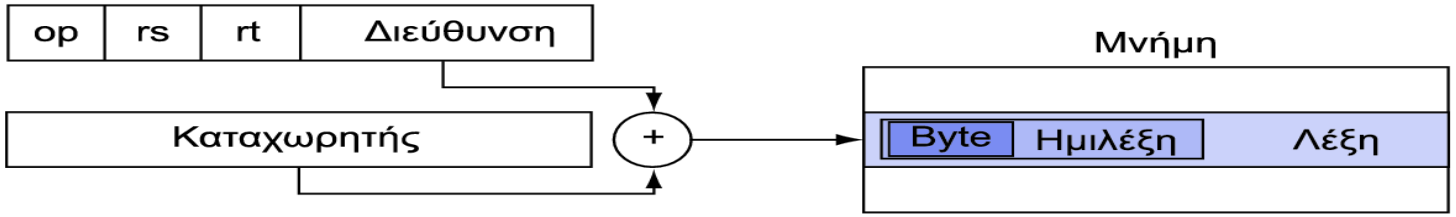
Περίληψη Τρόπων Διευθυνσιοδότησης

1. Άμεση διευθυνσιοδότηση

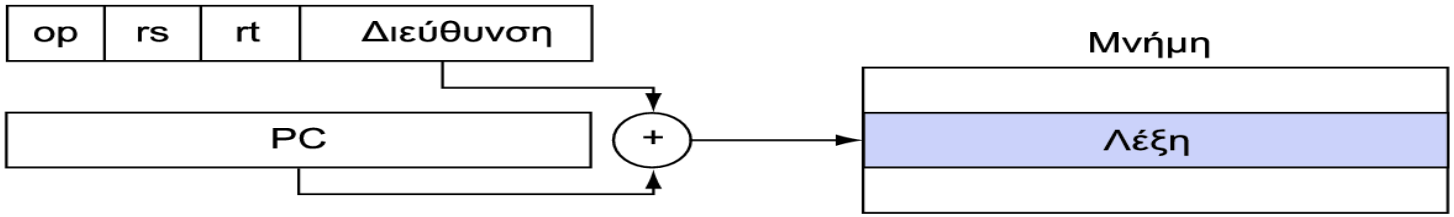


2. Διευθυνσιοδότηση μέσω καταχωρητή

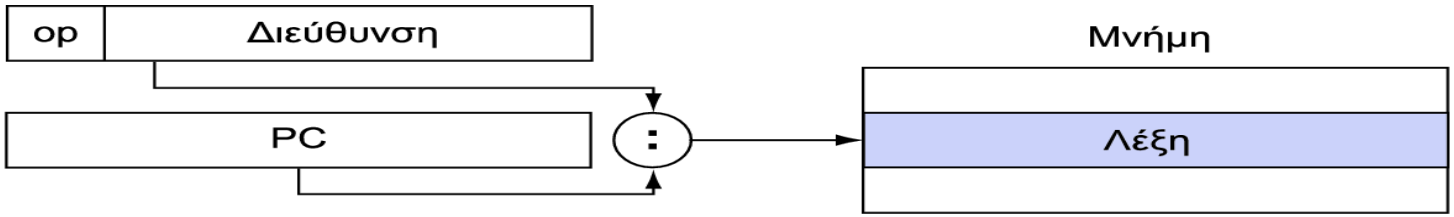
3. Διευθυνσιοδότηση βάσης



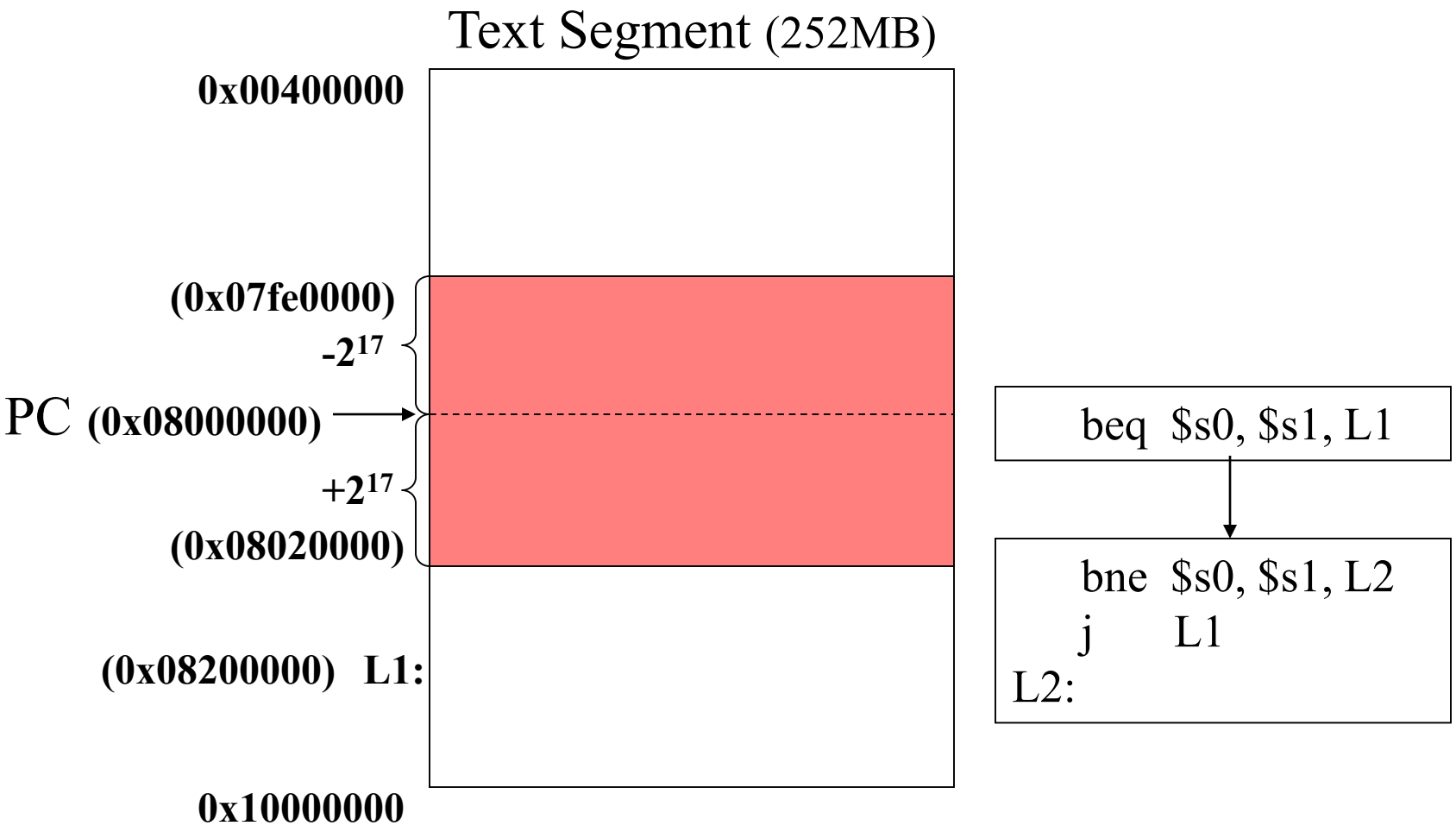
4. Σχετική διευθυνσιοδότηση ως προς PC



5. Ψευδο-απευθείας διευθυνσιοδότηση



Παράδειγμα : Απομακρυσμένες Διευθύνσεις



Δείκτες (Pointers)

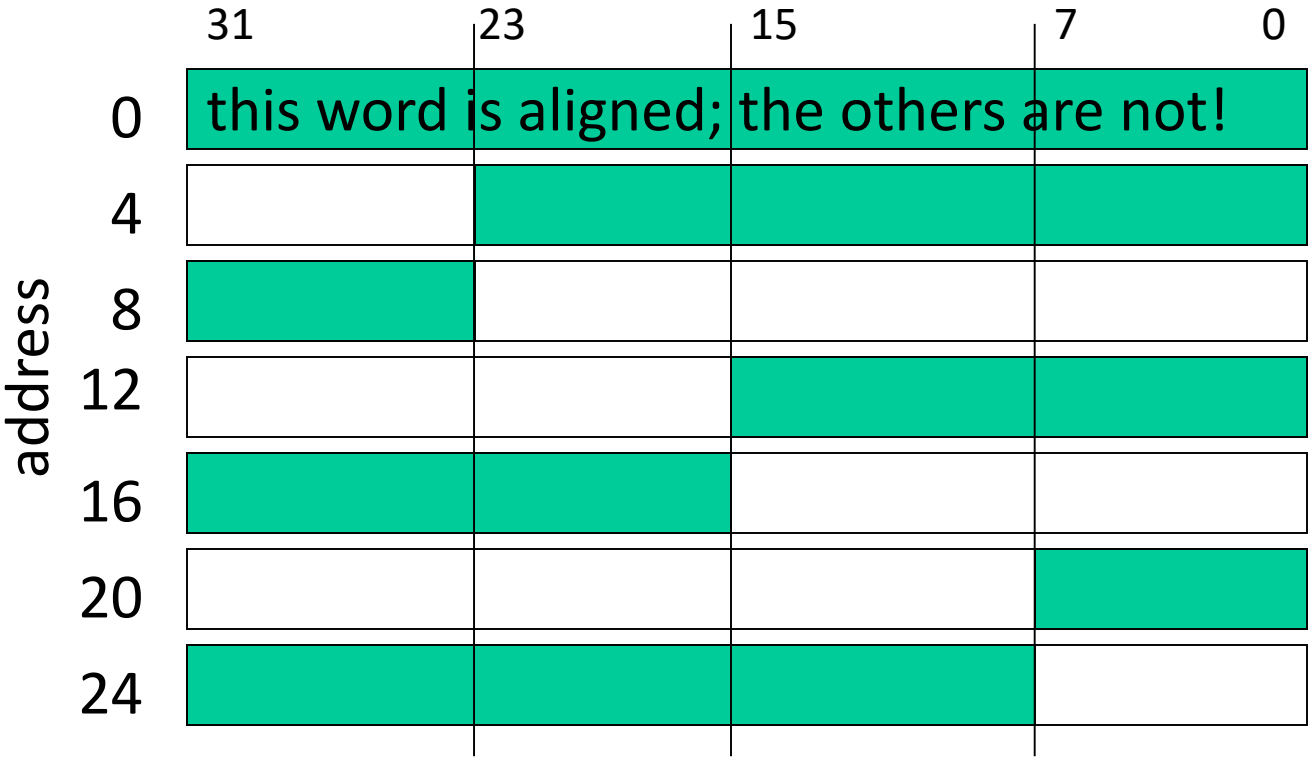
- Pointer: Μια μεταβλητή, η οποία περιέχει τη διεύθυνση μιας άλλης μεταβλητής
  - Αποτελεί τη HLL έκφραση της διεύθυνσης μνήμης σε γλώσσα μηχανής
- Γιατί χρησιμοποιούμε δείκτες;
  - Κάποιες φορές είναι ο μοναδικός τρόπος για να εκφράσουμε κάποιο υπολογισμό
  - Πιο αποδοτικός και συμπτυγμένος κώδικας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

Memory layout: Alignment



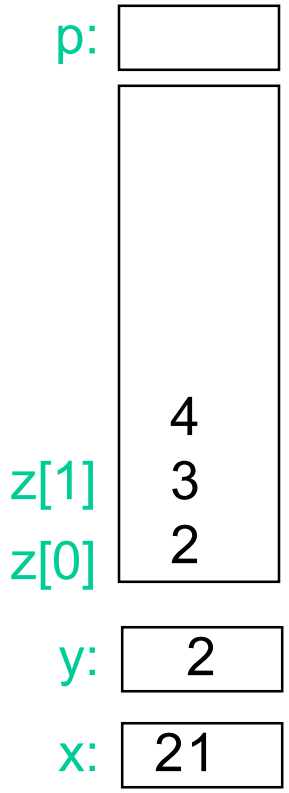
Τα (βασικά) δεδομένα στον MIPS ***πρέπει*** να είναι «ευθυγραμμισμένα» σε διεύθυνση που είναι πολλαπλάσιο του μήκους τους.

Λέξεις => πολ/σιο του 4, half => πολ/σιο του 2, char => πολ/σιο του 1

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.

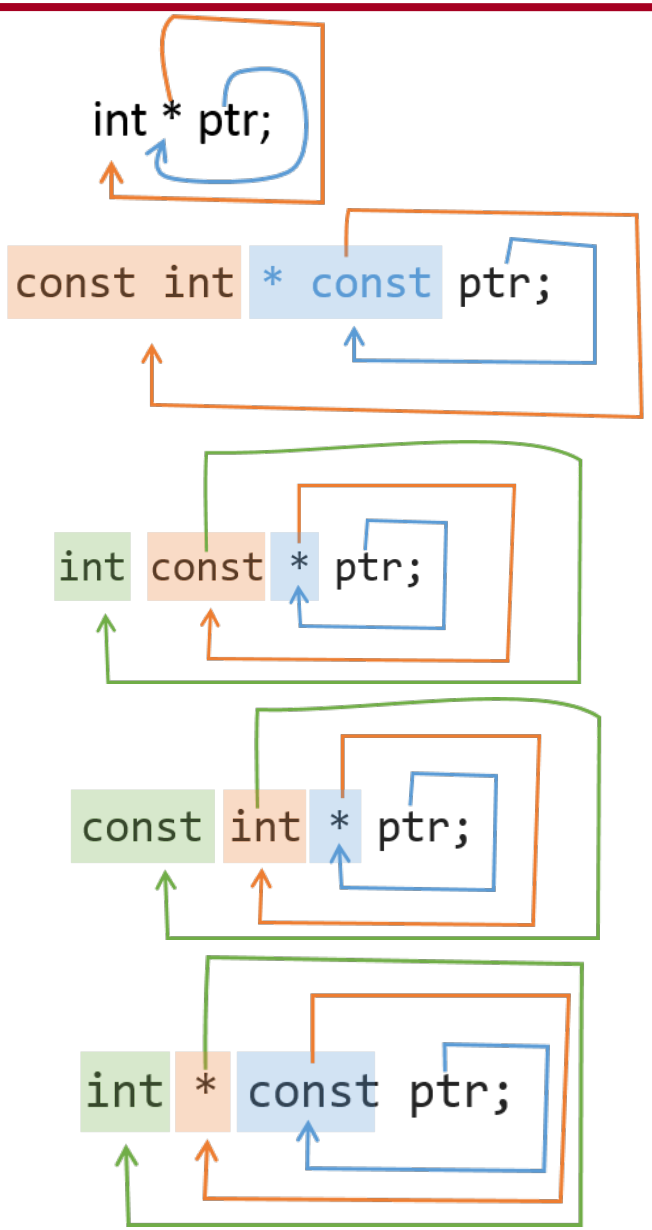
Let's.. play!

`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

Clockwise/Spiral Rule (http://c-faq.com/decl/spiral.anderson.html)



https://stackoverflow.com/questions/1143262/what-is-the-difference-between-const-int-const-int-const-and-int-const?rq=1

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`

Παράδειγμα: Arrays vs Pointers

Θέλουμε μια διαδικασία που να μηδενίζει τα στοιχεία ενός πίνακα `array` μεγέθους `size`

1. Array version of clear

```
clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}
clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

Procedure `clear1` (array index version)

```
clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}
```

```
        move  $t0,$zero       # i = 0
loop1:  sll   $t1,$t0,2       # $t1 = i * 4
        add   $t2,$a0,$t1     # $t2 = address of array[i]
        sw    $zero, 0($t2)   # array[i] = 0
        addi  $t0,$t0,1       # i = i + 1
        slt   $t3,$t0,$a1     # $t3 = (i < size)
        bne   $t3,$zero,loop1 # if (i < size) go to loop1
```

$a0: array (&array[0])

$a1: size

Procedure `clear2` (pointer arithmetic version)

```
clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```



```
        move  $t0,$a0         # p = address of array[0]
        sll   $t1,$a1,2       # $t1 = size * 4
        add   $t2,$a0,$t1     # $t2 = address of array[size]
loop2:  sw$zero,0($t0)        # Memory[p] = 0
        addi  $t0,$t0,4       # p = p + 4
        slt   $t3,$t0,$t2     # $t3 = (p<&array[size])
        bne   $t3,$zero,loop2 # if (p<&array[size]) go to loop2
```

$a0: array (&array[0])

$a1: size

Σύγκριση:

```
        move  $t0,$zero       # i = 0
loop1:  sll   $t1,$t0,2       # $t1 = i * 4
        add   $t2,$a0,$t1     # $t2 = address of array[i]
        sw    $zero, 0($t2)   # array[i] = 0
        addi  $t0,$t0,1       # i = i + 1
        slt   $t3,$t0,$a1     # $t3 = (i < size)
        bne   $t3,$zero,loop1 # if (i < size) go to loop1
```

Array index `i` έκδοση (clear1)

pointer `p` έκδοση (clear2)

```
        move $t0,$a0          # p = address of array[0]
loop2:  sw$zero,0($t0)        # Memory[p] = 0
        addi $t0,$t0,4        # p = p + 4
        sll  $t1,$a1,2        # $t1 = size * 4
        add  $t2,$a0,$t1      # $t2 = address of array[size]
        slt  $t3,$t0,$t2      # $t3 = (p<&array[size])
        bne  $t3,$zero,loop2 # if (p<&array[size]) go to loop2
```

```
        move  $t0,$a0         # p = address of array[0]
        sll   $t1,$a1,2       # $t1 = size * 4
        add   $t2,$a0,$t1     # $t2 = address of array[size]
loop2:  sw$zero,0($t0)        # Memory[p] = 0
        addi  $t0,$t0,4       # p = p + 4
        slt   $t3,$t0,$t2     # $t3 = (p<&array[size])
        bne   $t3,$zero,loop2 # if (p<&array[size]) go to loop2
```

Καλύτερη έκδοση, μια φορά ο υπολογισμός του array[size] εκτός loop

```
clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
         array[i] = 0;
}
clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
         *p = 0;
}
```

Σύγκριση:

```
      move  $t0,$zero      # i = 0
loop1:sll   $t1,$t0,2      # $t1 = i * 4
      add   $t2,$a0,$t1    # $t2 = &array[i]
      sw    $zero, 0($t2)  # array[i] = 0
      addi  $t0,$t0,1      # i = i + 1
      slt   $t3,$t0,$a1    # $t3 = (i < size)
      bne   $t3,$zero,loop1# if () go to loop1
```

Loop on index `i`.

```
      move  $t0,$a0        # p = & array[0]
      sll   $t1,$a1,2      # $t1 = size * 4
      add   $t2,$a0,$t1    # $t2 = &array[size]
loop2:sw    $zero,0($t0)   # Memory[p] = 0
      addi  $t0,$t0,4      # p = p + 4
      slt   $t3,$t0,$t2    # $t3=(p<&array[size])
      bne   $t3,$zero,loop2# if () go to loop2
```

Loop on pointer `p`. Optimized

Η array έκδοση με τον `index i` (αριστερά) πρέπει να έχει το `sll` (πολλαπλασιασμός x 4) και το `add` μέσα στον βρόχο, γιατί το `i` πρέπει να αυξάνεται σε κάθε επανάληψη και κάθε διεύθυνση στοιχείου πρέπει να υπολογίζεται από τον νέο δείκτη `i`.

Η έκδοση με τον `pointer p` (δεξιά) κάνει επανάληψη πάνω στον p, αυξάνοντας τον `p` κατευθείαν. Η pointer έκδοση μετακινεί το `sll` και τον υπολογισμό του array bound εκτός βρόχου, μειώνοντας τις εντολές που πρέπει να είναι εντός βρόχου από 6 σε 4.

Πώς προσθέτουμε επιπλέον έλεγχο ότι `size>0`;
Βάζουμε ένα `jmp` στην `slt` πριν το loop.

Bubblesort in C

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j+=1) {
            swap(v,j);
        }
    }
}
```

Swap procedure

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


Procedure body

```
swap: sll    $t1, $a1, 2        # reg $t1 = k * 4
      add    $t1, $a0, $t1      # reg $t1 = v + (k * 4)
                                # reg $t1 has the address of v[k]
      lw     $t0, 0($t1)        # reg $t0 (temp) = v[k]
      lw     $t2, 4($t1)        # reg $t2 = v[k + 1]
                                # refers to next element of v
      sw     $t2, 0($t1)        # v[k] = reg $t2
      sw     $t0, 4($t1)        # v[k+1] = reg $t0 (temp)
```

Procedure return

```
      jr     $ra                # return to calling routine
```

Saving registers

```
sort:   addi    $sp,$sp, -20        # make room on stack for 5 registers
        sw      $ra, 16($sp)        # save $ra on stack
        sw      $s3,12($sp)         # save $s3 on stack
        sw      $s2, 8($sp)         # save $s2 on stack
        sw      $s1, 4($sp)         # save $s1 on stack
        sw      $s0, 0($sp)         # save $s0 on stack
```

Procedure body

Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1	# copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero	# i = 0
	for1tst:	slt	$t0, $s0,$s3	# reg$t0=0if$s0Š$s3(iŠn)
		beq	$t0, $zero, exit1	# go to exit1 if $s0 Š $s3 (i Š n)
Inner loop		addi	$s1, $s0, -1	# j = i - 1
	for2tst:	slti	$t0, $s1,0	# reg$t0=1if$s1<0(j<0)
		bne	$t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)
		sll	$t1, $s1, 2	# reg $t1 = j * 4
		add	$t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)	# reg $t3 = v[j]
		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3	# reg $t0 = 0 if $t4 Š $t3
		beq	$t0, $zero, exit2	# go to exit2 if $t4 Š $t3
Pass parameters and call		move	$a0, $s2	# 1st parameter of swap is v (old $a0)
		move	$a1, $s1	# 2nd parameter of swap is j
		jal	swap	# swap code shown in Figure 2.25
Inner loop		addi	$s1, $s1, -1	# j -= 1
		j	for2tst	# jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1	# i += 1
		j	for1tst	# jump to test of outer loop

Restoring registers

```
exit1:  lw      $s0, 0($sp)         # restore $s0 from stack
        lw      $s1, 4($sp)         # restore $s1 from stack
        lw      $s2, 8($sp)         # restore $s2 from stack
        lw      $s3,12($sp)         # restore $s3 from stack
        lw      $ra,16($sp)         # restore $ra from stack
        addi    $sp,$sp, 20         # restore stack pointer
```

Procedure return

```
        jr      $ra                 # return to calling routine
```

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

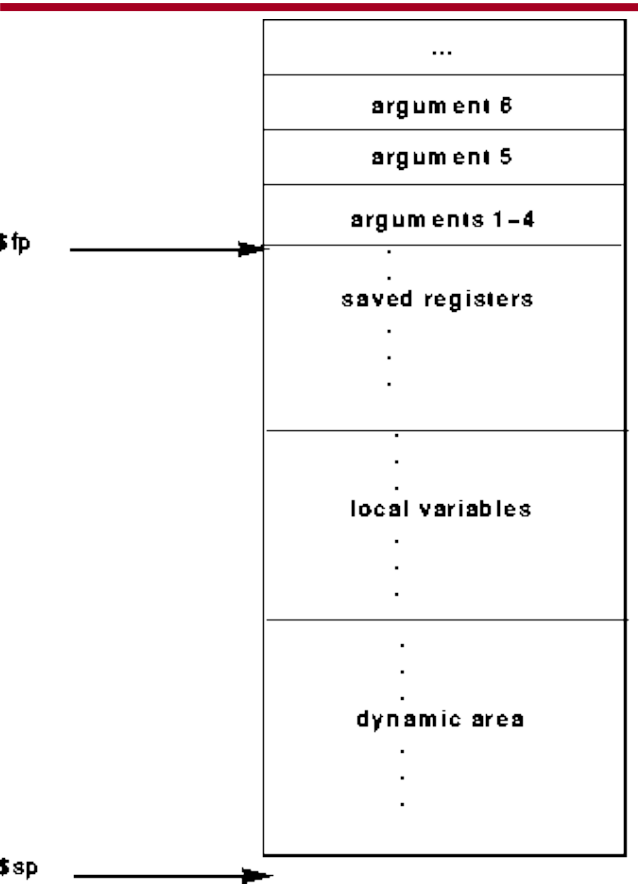
Assembly Code : Παράδειγμα (2)

```
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) /* dereference p */
        beq  $t1,$zero, Exit
Loop:   addi $t0,$t0,1  /* p++ */
        lbu  $t1,0($t0) /* dereference 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



The following steps are necessary to effect a call:

1. Pass the arguments. By convention, the first four arguments are passed in registers $a0-$a3 (though simpler compilers may choose to ignore this convention and pass all arguments via the stack). The remaining arguments are pushed on the stack.

2. Save the caller-saved registers. This includes registers $t0-$t9, if they contain live values at the call site.

3. Execute a jal instruction.

Within the called routine, the following steps are necessary:

1. Establish the stack frame by subtracting the frame size from the stack pointer.
2. Save the callee-saved registers in the frame. Register $fp is always saved. Register $ra needs to be saved if the routine itself makes calls. Any of the registers $s0- $s7 that are used by the callee need to be saved.
3. Establish the frame pointer by adding the stack frame size to the address in $sp.

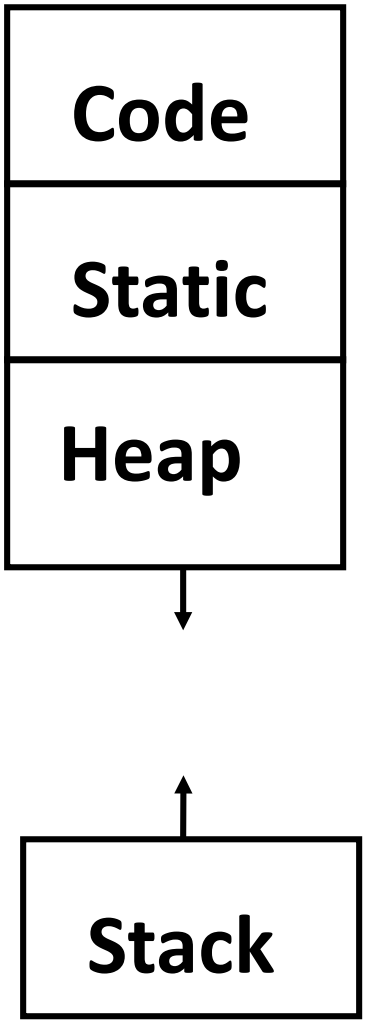
Finally, to return from a call, a function places the returned value into $v0 and executes the following steps:

1. Restore any callee-saved registers that were saved upon entry (including the frame pointer $fp).
2. Pop the stack frame by adding the frame size to $sp.
3. Return by jumping to the address in register $ra.

source: SPIM S20: A MIPS R2000 Simulator, *James R. Larus - larus@cs.wisc.edu*

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 τοπική μεταβλητή

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

```
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
```

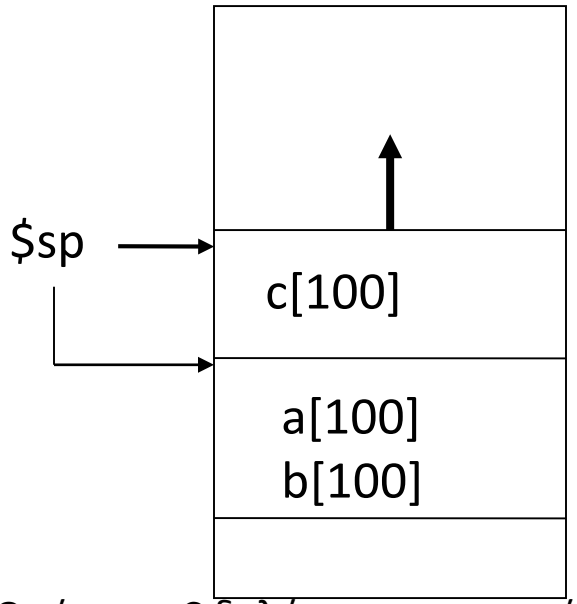
Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες : Version 1

```
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;
    }
```

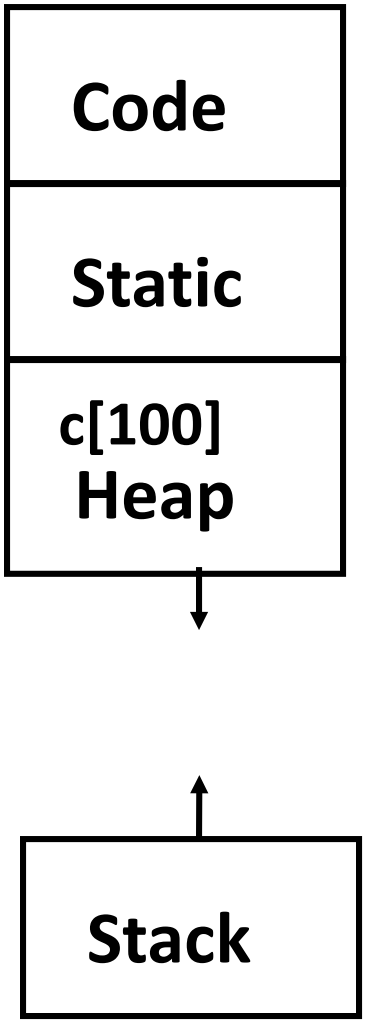



```
        addi  $t0,$a0,400   # beyond end of a[]
        addi  $sp,$sp,-400  # space for c
        addi  $t3,$sp,0     # ptr for c
        addi  $v0,$t3,0     # $v0 = &c[0]
Loop:   beq   $a0,$t0,Exit
        lw    $t1, 0($a0)   # $t1=a[i]
        lw    $t2, 0($a1)   # $t2=b[i]
        add   $t1,$t1,$t2   # $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
```

Ο πίνακας C δηλώνεται ως τοπική (automatic) μεταβλητή στην συνάρτηση, οπότε με την λήξη της απελευθερώνεται. Κατά συνέπεια δεν μπορεί να επιστραφεί ως αποτέλεσμα!

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

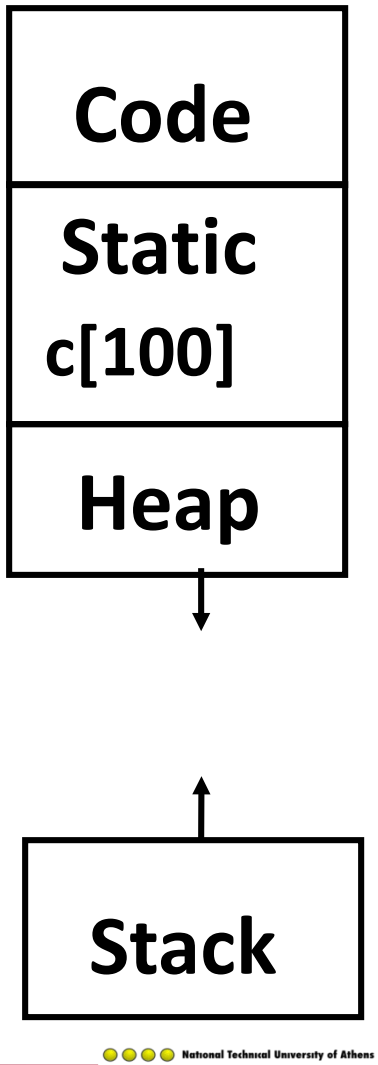
Πίνακες, Δείκτες και Μέθοδοι/Διαδικασίες : Version 3

```
                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 MIPS (1a)

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 MIPS (1b)

- Οι λέξεις (words) έχουν πλάτος 32 bits
- Υπάρχει ένα αρχείο καταχωρητών με 32 καταχωρητές των 32 bits
- Ονοματολογίες καταχωρητών: $0 - $31 ή με συμβολικά ονόματα: $zero, $at, $t0, … $sp, $ra
- Ο καταχωρητής $0 έχει *πάντα* την τιμή 0 (μηδέν)
- Ειδικός καταχωρητής: Program Counter (PC) με πλάτος 32 bits

$zero	$0	zero
$at	$1	assembler temporary register
$v0, $v1	$2, $3	expression evaluation and function result
$a0..$a3	$4..$7	procedure arguments
$t0..$t7	$8..$15	temporary
$s0..$s7	$16..$23	saved temporaries
$t8, $t9	$24, $25	temporary
$k0, $k1	$26, $27	used by OS Kernel
$gp $28		global pointer
$sp	$29	stack pointer
$fp	$30	frame pointer
$ra	$31	return address (from subroutine call)

Συμβάσεις Χρήσης Καταχωρητών

- Ο καταχωρητής $at χρησιμοποιείται από τον συμβολομεταφραστή για την σύνθεση ψευδοεντολών.
- Οι καταχωρητές $v0, $v1 χρησιμοποιούνται για την επιστροφή τιμών από συναρτήσεις (functions)
- Οι καταχωρητές $a0..$a3 χρησιμοποιούνται για το πέρασμα παραμέτρων σε διαδικασίες και συναρτήσεις (procedures & functions)
- Οι καταχωρητές $sp και $fp χρησιμοποιούνται ώς stack pointer και frame pointer αντίστοιχα.
- Ο καταχωρητής $ra (return address) χρησιμοποιείται για την αποθήκευση της διεύθυνσης επιστροφής από υπορουτίνα

Επανάληψη (3)

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
Uncondi-tional 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

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

Συμβάσεις Κλήσης Υπορουτινών

- Παράμετροι συνάρτησης/υπορουτίνας
  - Scalar τιμές (ακέραιοι, bytes, χαρακτήρες) στους καταχωρητές $a0-$a3.
  - Οι μή-scalar τιμές (συμβολοσειρές, πίνακες, structures, κ.λ.π), όπως και οι υπόλοιπες παράμετροι αν είναι πάνω από 4, περνιούνται στην στοίβα
- Τιμή επιστροφής συνάρτησης:
  - Στους καταχωρητές $v0, $v1 (οι γλώσσες προγραμματισμού συνήθως ορίζουν μόνο μία τιμή => $v0)

Επιπλέον Υλικό

- Οι καταχωρητές $t0..$t9 ονομάζονται «προσωρινοί» (temporary) και ενδείκνυται για την αποθήκευση προσωρινών τιμών οι οποίες δεν απαιτείται να διατηρηθούν και μετά από μια κλήση διαδικασίας ή συνάρτησης. Εάν η διαδικασία που καλείται χρησιμοποιήσει αυτούς τους καταχωρητές, οι παλαιές τους τιμές χάνονται.
- Οι καταχωρητές $s0..$s7 ονομάζονται saved και ενδείκνυται να χρησιμοποιούνται για αποθήκευση τιμών που διατηρούνται για περισσότερο χρόνο και μπορούν να διατηρούν την τιμή τους και μετά από μία κλήση διαδικασίας ή συνάρτησης.
- Οι καταχωρητές $t0..$t9 είναι τύπου «caller-save», ενώ οι $s0..$s7 είναι τύπου «callee-save».

Επιπλέον Υλικό

- Οι συναρτήσεις πρέπει να διατηρήσουν αναλλοίωτες τις τιμές των καταχωρητών τύπου callee-save ($s0-$s7).
  - Εάν μεταβάλλουν κάποιους καταχωρητές τύπου callee-save, αυτοί πρέπει να σωθούν στην στοίβα κατά την έναρξη της συνάρτησης (στον «πρόλογο»), και να επαναφερθούν πριν την επιστροφή (στον «επίλογο»), ώστε μετά την επιστροφή, οι καταχωρητές αυτοί να έχουν τις παλιές τιμές τους.
- Ακόμα, εάν μια συνάρτηση Α χρησιμοποιεί ένα καταχωρητή τύπου caller-save για να διατηρήσει μια τιμή πέρα από μια κλήση υπορουτίνας Β, τότε πρέπει να τον σώσει στην στοίβα πρίν την κλήση της Β και να τον επαναφέρει πριν τον ξαναχρησιμοποιήσει.
  - Αυτό γιατί η υπορουτίνα Β σύμφωνα με τις συμβάσεις χρήσης καταχωρητών του MIPS, έχει δικαίωμα να πανωγράψει τους καταχωρητές τύπου caller-save.
- Παρατήρηση: caller-save καταχωρητές εκτός των $t0-$t9 είναι ***και*** οι $v0, $v1, $a0-$a3, $ra (πανωγράφεται από την jal),

Επιπλέον Υλικό

- Η υπορουτίνα Α που καλεί (caller):
  - Σώζει στην στοίβα όσους καταχωρητές «caller-save» έχει χρησιμοποιήσει και χρειάζεται την τιμή τους μετά την κλήση
  - Περνάει τις παραμέτρους στους καταχωρητές $a0-$a3 ή/και στην στοίβα (ανάλογα με το πλήθος και το είδος των παραμέτρων)
  - Jal function
  - Επαναφέρει τους καταχωρητές «caller-save» πού είχε σώσει
- Η υπορουτίνα Β που καλείται αποτελείται από τρία κομμάτια:
  - **Πρόλογος:** κομμάτι κώδικα που κάνει διαδικαστικούς υπολογισμούς. Σώζει στην στοίβα όσους καταχωρητές «callee-save» θα χρησιμοποιηθούν στην υπορουτίνα αυτή.
  - **Κυρίως Σώμα:** ο κώδικας της συνάρτησης/υπορουτίνας
  - **Επίλογος:** Επαναφέρει τους καταχωρητές «callee-save» πού είχε σώσει
  - Ο καταχωρήτής $ra είναι τύπου caller-save, οπότε σώζεται στο πρόλογο και επαναφέρεται στον επίλογο μόνο αν η Β καλεί κάποια συνάρτηση

Επιπλέον Υλικό: Ποιόν καταχωρητή να χρησιμοποιήσω για τις μεταβλητές μου;

- Χωρίζουμε δύο είδη μεταβλητών: με *σύντομη* ζωή, και με *μακρόχρονη* ζωή. Η ζωή μιάς μεταβλητής ορίζεται ως η απόσταση από την εγγραφή στην μεταβλητή αυτή, μέχρι την τελευταία ανάγνωση αυτής της τιμής. Αν η τιμή δεν διαβάζεται (πριν πανωγραφτεί), η μεταβλητή είναι «νεκρή» και δεν χρειάζεται να καταναλώνει καταχωρητή.
- Σε μια συνάρτηση όμως μας ενδιαφέρει κυρίως αν η «ζωή» της μεταβλητής ***επεκτείνεται και πέρα από κλήση υπορουτίνας.***
  - Αν **ναι**, τότε ή ζωή της μεταβλητής είναι **«μεγάλη»**, και προσπαθούμε να την βάλουμε σε καταχωρητή τύπου callee-save.
  - Αν **όχι**, τότε ή ζωή της μεταβλητής είναι **«μικρή»**, και προσπαθούμε να την βάλουμε σε καταχωρητή τύπου caller-save.
- Ετσι, αν έχω πολλές συναρτήσεις με πολλές μεταβλητές μικρής ζωής που δεν επεκτείνονται πέρα από κλήση υπορουτίνας, τις τοποθετώ όλες (όσες χωράνε) σε καταχωρητές τύπου caller-save. Αφού η ζωές των μεταβλητών δεν εκτείνονται μετά από τις κλήσεις υπορουτινών, δεν χρειάζεται να σωθούν στην στοίβα!

Αναδρομική συνάρτηση factorial()

```
long factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
}
```

Αναδρομική συνάρτηση factorial() #1

```
factorial:
#  Prologos: apothikeysh dieythynshs epistrofhs, kai tou
orismatos
#  to opoio xreiazetai kai meta thn anadromikh klhsh ths
factorial
        addu $sp, $sp, -8
        sw $ra, 4($sp) # save return address
        sw $a0, 0($sp) # save argument (n)
# Telos prologou

        bgtz $a0, greater_than_zero
# If we reach here, the argument is zero, and fact(0) = 1.
        li $v0, 1
        j epilogue
```

Αναδρομική συνάρτηση factorial() #2

```
greater_than_zero:
        sub $a0, $a0, 1 # call factorial(n-1)
        jal factorial
        # factorial(n-1) is in $v0

        lw $v1, 0($sp) # restore n from stack
        mul $v0, $v0, $v1 # multiply n * factorial(n-1)

        # we are done, $v0 has the correct value
epilogue:
        lw $ra, 4($sp) # Epanafora ths dieythynshs
                       # epistrofhs
        addiu $sp, $sp, 8 # Epanafora tou $sp sthn timh
                          # prin thn klhsh
        jr $ra
```

Παράδειγμα χρήσης καταχωρητών

Έστω οι παρακάτω δύο συναρτήσεις A και B:

```
int A(int x) {
    int y;

    y = B(x);
    y = y + x;
    return y;
}

int B(int arg) {
    return arg + 5;
}
```

Οι συμβάσεις χρήσης καταχωρητών του MIPS ορίζουν ότι το X στην A (και στην B) θα πρέπει να περαστεί στον $a0.

Επίσης ορίζουν ότι η τιμή επιστροφής θα πρέπει να μπει στον $v0.

Ακόμα η διεύθυνση επιστροφής της συνάρτησης βρίσκεται στον καταχωρητή $ra

Προσπάθεια Πρώτη

Η Β είναι απλή: το arg υπάρχει ήδη στον καταχωρητή $a0, οπότε το αυξάνουμε κατά 1, και γράφουμε το αποτέλεσμα στον $v0

Στην Α, η παράμετρος εισόδου x βρίσκεται στον $a0, οπότε για να περαστεί στην Β βάλαμε την γραμμή που αντιγράφει το $a0 στο $a0, το οποίο προφανώς είναι περιττό

Κατόπιν, καλούμε την Β η οποία επιστρέφει το αποτέλεσμα της στον $v0

Ακολούθως προσθέτουμε το αποτέλεσμα με το $a0 και βάζουμε το αποτέλεσμα στον $v0

Η κλήση Α(10) καλεί την Β(10) η οποία επιστρέφει το 15, η Α προσθέτει το 10 και επιστρέφει 25

Ο κώδικας είναι φαινομενικά σωστός

```
A:
    mov $a0, $a0 // περιττό!
    jal B
    add $v0, $v0, $a0
    jr $ra

B:
    addi $v0, $a0, 5
    jr $ra
```

Τι θα γίνει αν αλλάξω λίγο την Β;

Προσπάθεια Δεύτερη

Η Β άλλαξε λίγο: πρώτα γίνεται η αύξηση στον $a0 και μετά η αντιγραφή του αποτελέσματος στον $v0.

Ο κώδικας της Β είναι σωστός!

Αλλά υπάρχουν προβλήματα:

1) Η κλήση Α(10) καλεί την Β(10) η οποία επιστρέφει το 15, η Α προσθέτει το 15 και υπολογίζει το 30 το οποίο και επιστρέφει η Α.

2) Το ίδιο συμβαίνει και για τον $ra! Το $ra στην Α δείχνει π.χ. στην main από όπου κλήθηκε. Μετά την κλήση Β αλλάζει ώστε να δείχνει στην 3η εντολή της Α. Η Β επιστρέφει σωστά, αλλά το jr $ra της Α δεν επιστρέφει στην main αλλά στην ίδια την Α.

```
A: mov $a0, $a0 // περιττό!
   jal B
   add $v0, $v0, $a0
   jr $ra

B: addi $a0, $a0, 5
   mov      $v0, $a0
   jr $ra
```

Προσπάθεια Τρίτη

Η Α δημιουργεί ένα αντίγραφο του $a0 σε ένα μπλε κουτί. Κατόπιν καλεί την Β η οποία αλλάζει την τιμή του $a0 και επιστρέφει το αποτέλεσμα της (15 στην περίπτωση της κλήσης Α(10)) στον $v0 και επιστρέφει στην Α.

Η Α επαναφέρει την παλιά τιμή του $a0 (10) από το μπλε κουτί, και τώρα η πρόσθεση είναι σωστή και υπολογίζει 15 + 10 = 25 το οποίο και γράφεται στον $v0.

Ο κώδικας της Α υπολογίζει πλέον το σωστό αποτέλεσμα ανεξάρτητα πως θα γραφτεί η Β!

Το SAVE γίνεται πριν την κλήση της συνάρτησης που φοβόμαστε

Το RESTORE γίνεται μετά την κλήση και πριν την χρήση του καταχωρητή που χρειαζόμαστε

```
A:  SAVE $a0, LOCATION
    jal B
    RESTORE $a0, LOCATION
    add $v0, $v0, $a0
    jr $ra

B:  addi $a0, $a0, 5
    mov     $v0, $a0
    jr $ra
```

Προσπάθεια Τρίτη++

Το ίδιο πρέπει να γίνει και για τον $ra!!!

Σε ένα άλλο πορτοκαλί κουτί η Α δημιουργεί ένα αντίγραφο του $ra

Η Α καλεί όποια συνάρτηση χρειάζεται (την Β στο συγκεκριμένο παράδειγμα)

Η Α επαναφέρει την παλιά τιμή του $ra από το πορτοκαλί κουτί, και τώρα η διεύθυνση επιστροφής είναι σωστή και δείχνει σε όποιον κάλεσε την Α.

Τώρα πια ο κώδικας της Α λειτουργεί σωστά υπολογίζει το σωστό αποτέλεσμα ανεξάρτητα πως θα γραφτεί η Β και επιστρέφει σωστά σε όποιον την κάλεσε!

```
A:  SAVE $ra, LOC-ra
    SAVE $a0, LOCATION
    jal B
    RESTORE $a0, LOCATION
    add $v0, $v0, $a0
    RESTORE $ra, LOC-ra
    jr $ra

B:  addi $a0, $a0, 5
    mov     $v0, $a0
    jr $ra
```

Εξήγηση 3ης Προσπάθειας

Η Α δεν γνωρίζει αν η Β όπως έχει γραφτεί θα χρησιμοποιήσει ή όχι κάποιους καταχωρητές. Δεν υπάρχει καμμία εγγύηση για κάτι τέτοιο. Όπως και να γράψω την Β είναι σωστό.

Συνεπώς, η Α θεωρεί ότι οτιδήποτε χρήσιμο για την ίδια (ο $a0) μπορεί να καταστραφεί από την Β.

Ο $ra είναι άλλη περίπτωση: ανεξάρτητα πως θα γράψω την Β (ακόμα και για κενή Β), το $ra πανωγράφεται από την ίδια την Α όταν εκτελείται η εντολή jal Β!

```
A:  SAVE $ra, LOC-ra
    SAVE $a0, LOCATION
    jal B
    RESTORE $a0, LOCATION
    add $v0, $v0, $a0
    RESTORE $ra, LOC-ra
    jr $ra

B:  addi $a0, $a0, 5
    mov     $v0, $a0
    jr $ra
```

Γενικός Τρόπος γραφής κώδικα

Πρόλογος

```
A:  SAVE $ra, LOCATION-ra
```

Σώμα

```
    SAVE $a0, LOCATION
    jal B
    RESTORE $a0, LOCATION
    add $v0, $v0, $a0
```

Επίλογος

```
    RESTORE $ra, LOCATION-ra
    jr $ra
```

```
B:                                  Πρόλογος
    addi $a0, $a0, 5
    mov    $v0, $a0                 Σώμα
    jr $ra                          Επίλογος
```

Κάθε συνάρτηση έχει Πρόλογο, Σώμα, Επίλογο. Ο πρόλογος μπορεί να είναι κενός (όπως στην περίπτωση της B).

Γράφουμε *πρώτα* το Σώμα, και *μετά* γεμίζουμε Πρόλογο και Επίλογο (όταν ξέρουμε τι χρειαζόμαστε όπως θα δούμε παρακάτω).

Πρέπει να συζητήσουμε:

1. γιατί το Μπλέ και το Πορτοκαλί κουτάκι είναι διαφορετικά;
2. που βρίσκονται τα κουτάκια αυτά;
3. Υπάρχουν άλλες επιλογές για να γράψω κώδικα;

Συζήτηση

Δυο τύποι SAVE/RESTORE

- Αυτά που αφορούν πρόλογο/επίλογο (μόνο το $ra στο προηγούμενο παράδειγμα)
- Τα άλλα που αφορούν καταχωρητές και τιμές που χρησιμοποιώ στο σώμα συνάρτησης

Που βρίσκονται τα κουτάκια αυτά; Πως γίνεται το SAVE/RESTORE;

- Ο χώρος θα πρέπει να είναι σε ιδιωτικό για την κάθε συνάρτηση μέρος
- Σε περίπτωση αναδρομικής συνάρτησης θα πρέπει να έχω πολλές τέτοιες θέσεις (μία για κάθε ενεργοποίηση της συνάρτησης)
- Συνεπώς θα πρέπει να είναι στην στοίβα. Οπότε SAVE = Push, Restore = Pop

Στοίβα συστήματος στον MIPS

- Υλοποιείται με τον $sp (η στοίβα μεγαλώνει προς τα κάτω)
- Push($a0) => ακολουθία: addui $sp, $sp, -4 ; sw $a0, 0($sp)
- Pop($a0) => ακολουθία: lw $a0, 0($sp) ; addui $sp, $sp, 4

Ο τελικός κώδικας της A

```
A:      addui   $sp, $sp, -8
Πρόλογος sw     $ra, 0($sp) //push

        sw      $a0, 4($sp) //push
        jal     B
Σώμα    lw      $a0, 4($sp) //pop
        add     $v0, $v0, $a0

        lw      $ra, 0($sp) //pop
Επίλογος addui  $sp, $sp, 8
        jr      $ra
```

Χρειαζόμαστε 2 λέξεις στην στοίβα

Στο offset 0 από τον $sp θα αποθηκευτεί ο $ra και σε offset 4 ο $a0

Η πρώτη εντολή του Προλόγου (addui) δημιουργεί τον χώρο κουνώντας τον $sp προς τα κάτω

Κατόπιν μπορούμε αν γράψουμε τον $ra υλοποιώντας το δεύτερο μισό της push

Ανάλογα αποθηκεύεται και ο $a0

Η επαναφορά του $a0 είναι απλά ένα lw

Η επαναφορά του $sp στην αρχική του θέση γίνεται στο τέλος του Επιλόγου αμέσως πριν το jr $ra.

Μια άλλη προσέγγιση;

Τι αν η B εγγυόταν ότι δεν θα άλλαζε κανένα καταχωρητή;

Η B σώζει και επαναφέρει τον $a0, οπότε η A *δεν βλέπει αλλαγή*!

Ομοίως και η A πρέπει να δώσει την ίδια εγγύηση σε όποιον την καλεί! Αλλά δεν χρησιμοποιεί άλλον καταχωρητή, οπότε είναι OK.

Η τεχνική αυτή λέγεται *callee-save*, διότι η καλούμενη συνάρτηση έχει την υποχρέωση να κάνει όλη την δουλειά.

Η προηγούμενη τεχνική λέγεται *caller-save* διότι η συνάρτηση που καλεί πρέπει να σώσει τις ενδιαφέρουσες τιμές

Τα πορτοκαλί τί είναι; (Callee-save!)

```
B:      addui   $sp, $sp, -4
        sw      $a0,  0($sp) //push

        addi $a0, $a0, 5
        mov     $v0, $a0

        lw      $a0, 0($sp) //pop
        addui   $sp, $sp, 4
        jr      $ra
```

```
A:      addui   $sp, $sp, -4
        sw      $ra, 0($sp) //push

        jal     B
        add     $v0, $v0, $a0

        lw      $ra, 0($sp) //pop
        addui   $sp, $sp, 4
        jr      $ra
```

Caller/Callee Save

Callee-Save

- Η κάθε συνάρτηση *εγγυάται ότι δεν θα αλλάξει κανένα καταχωρητή* (εκτός π.χ. του $v0)
- Απλούστερο:
  - Στον πρόλογο σώζω όλους τους καταχωρητές που χρησιμοποιώ
  - Στον επίλογο επαναφέρω την παλιά τιμή τους

Caller-Save

- Οι συναρτήσεις *δεν εγγυόνται τίποτα* σε αυτόν που τις καλεί
- Πλήρης ελευθερία χρήσης καταχωρητών
- Είναι πιο αποτελεσματικό (;)
- Πρέπει να σώζω μόνο ότι επηρεάζεται από κλήσεις συναρτήσεων => όχι όλους τους καταχωρητές.
  - Εντός του σώματος της συνάρτησης

Caller/Callee Save

Πότε συμφέρει το ένα και πότε το άλλο;

- Το να εγγυόμαι στους άλλους ότι δεν θα αλλάξω τίποτα είναι σημαντική ευθύνη
- Το να μην έχω εγγυήσεις κάνει την ζωή μου πιο δύσκολη.

Ιδανικά;

- Υπάρχουν περιπτώσεις που σε προσέγγιση caller-save δεν χρειάζεται να σώσω τους καταχωρητές που χρησιμοποιώ
- Όταν η «ζωή» της τιμής που βρίσκεται στον καταχωρητή δεν διασταυρώνεται με κλήση συνάρτησης:

```
addi   $t3, $a2, 4
xor    $a0, $t3, $a1
jal    B
…
addi   $t3, $v0, 5
```

Η τιμή στον $t3 δεν χρειάζεται μετά την κλήση της B, οπότε <u>δεν χρειάζεται save/restore</u>

Το καλύτερο και από τις δύο προσεγγίσεις;;;

Οι συμβάσεις του MIPS χωρίζουν τους καταχωρητές σε δύο ομάδες

- Callee-save ($s0-$s7)
- Caller-save ($t0-$t9)

Η κάθε συνάρτηση δίνει εγγύηση ότι δεν θα αλλάξει τους $s0-$s7

Η κάθε συνάρτηση μπορεί να αλλάξει τους υπόλοιπους καταχωρητές: $t0-$t7, $a0-$a3, $v0, $v1, $at, $ra

Απλή στρατηγική χρήσης:

- Για καταχωρητή που η ζωή του «διασταυρώνεται» με κλήση συνάρτησης ➔ χρήση $s_ (με save/restore σε πρόλογο/επίλογο)
- Για καταχωρητή που χρησιμοποιείται χωρίς να «διασταυρώνεται» με κλήση συνάρτησης ➔ χρήση $t_ (με save/restore σε πρόλογο/επίλογο)

Η Α με χρήση Calee-save καταχωρητών

A:
Πρόλογος

```
addui   $sp, $sp, -8
sw      $ra, 0($sp) //push
sw      $s0, 4($sp) //push
```

Σώμα

```
move    $s0, $a0
jal     B
add     $v0, $v0, $s0
```

Επίλογος

```
lw      $s0, 4($sp) //pop
lw      $ra, 0($sp) //pop
addui   $sp, $sp, 8
jr      $ra
```

Χρειαζόμαστε 2 λέξεις στην στοίβα

Στο offset 0 από τον $sp θα αποθηκευτεί ο $ra και σε offset 4 ο $s0

Στον Πρόλογο δημιουργώ χώρο και αποθηκεύω $ra και $s0

Αρχικά στο σώμα δημιουργώ αντίγραφο του $a0 στον $s0 και στην συνέχεια χρησιμοποιώ τον $s0 για να αναφερθώ στο όρισμα Χ της Α

Στον Επίλογο γίνεται η επαναφορά των$s0 και $ra και η επαναφορά του $sp στην αρχική του θέση

H Aloop

```
int Aloop(int X, int N) {
    int y = 0, i;

    for (i=N; i>0; i--) {
        y = B(X) + y;
    }
    return y;
}
```

Η Aloop καλεί Ν φορές την B(X) και αθροίζει το αποτέλεσμα.

--δεν εξετάζουμε αν η Aloop έχει έννοια ή μπορεί να γραφτεί αλλιώς—

Η «ζωή» της y *εκτείνεται* πέρα της κλήσης της B.

Η «ζωή» της X *εκτείνεται* πέρα της κλήσης της B.

Η «ζωή» της i *εκτείνεται* πέρα της κλήσης της B.

Η Aloop με Callee-save καταχωρητές

```
Aloop:addui   $sp, $sp, -16
      sw      $ra, 0($sp) //push
      sw      $s0, 4($sp) //push
      sw      $s1, 8($sp) //push
      sw      $s2, 12($sp) //push

      move    $s0, $a0    // X
      move    $s1, $a1    // I = N
      move    $s2, $zero // y
Loop: move    $a0, $s0 // B's arg
      jal     B
      add     $s2, $v0, $s2
      addi    $s1, $s1, -1
      bnez    $s1, Loop
      move    $v0, $s2

      lw      $s2, 12($sp) //pop
      lw      $s1,  8($sp) //pop
      lw      $s0,  4($sp) //pop
      lw      $ra,  0($sp) //pop
      addui   $sp, $sp, 16
      jr      $ra
```

Εδώ η Aloop υλοποιείται με χρήση saved καταχωρητών ($s0, $s1, $s2).

Τρεις τιμές πρέπει να «επιβιώσουν» μετά την κλήση της B, το X (στον $s0), το i (στον $s1) και το y (στον $s2).

Για να γίνει η κλήση B(X) πρέπει να αντιγράψουμε τον $s0 στον $a0

ΣΥΝΟΛΟ SAVE/RESTORE

= 4 / 4

Η Aloop με Caller-save καταχωρητές

```
Aloop:addui   $sp, $sp, -16
      sw      $ra, 0($sp) //push
```

```
Loop:  sw     $a0, 4($sp) //push
       sw     $a1, 8($sp) //push
       sw     $t0, 12($sp) //push
       jal    B
       lw     $a0, 4($sp) //pop
       lw     $t0, 12($sp) //pop
       add    $t0, $v0, $t0
       lw     $a1, 8($sp) //pop
       addi   $a1, $a1, -1
       bnez   $a1, Loop
       move   $v0, $t0
```

```
       lw     $ra,  0($sp) //pop
       addui  $sp, $sp, 16
       jr     $ra
```

Εδώ η Aloop υλοποιείται με χρήση temporary καταχωρητών ($t0, $a0, $a1)

Τρεις τιμές πρέπει να «επιβιώσουν» μετά την κλήση της B, το X (στον $a0), το i (στον $a1) και το y (στον $t0)

ΣΥΝΟΛΟ SAVE/RESTORE

= 3*N+1 / 3*N+1

Η Aloop2

```
int Aloop2(int X, int N) {
    int y;

    for (i=N, i>0; i--) {
        y = X + i;
        X = B(y);
    }
    return i;
}
```

Η Aloop2 καλεί N φορές την B(X) με παράμετρο το (X + i), και βάζει το αποτέλεσμα στην X

Η «ζωή» της y *δεν* εκτείνεται πέρα της κλήσης της B.

Η «ζωή» της X *δεν* εκτείνεται πέρα της κλήσης της B (ξεκινάει μετά την B και χρησιμοποιείται πριν την B!)

Η «ζωή» της i *εκτείνεται* πέρα της κλήσης της B.

Με Callee-save καταχωρητές

```
Aloop2:addui  $sp, $sp, -16
       sw     $ra, 0($sp) //push
       sw     $s0, 4($sp) //push
       sw     $s1, 8($sp) //push
       sw     $s2, 12($sp) //push
```

```
       move   $s0, $a0   // X
       move   $s1, $a1   // i = N
Loop:  add    $a0, $s0, $s1 //
       jal    B
       move   $s0, $v0
       addi   $s1, $s1, -1
       bnez   $s1, Loop
       move   $v0, $s1
```

```
       lw     $s2, 12($sp) //pop
       lw     $s1,  8($sp) //pop
       lw     $s0,  4($sp) //pop
       lw     $ra,  0($sp) //pop
       addui  $sp, $sp, 16
       jr     $ra
```

Εδώ η Aloop2 υλοποιείται με χρήση saved καταχωρητών ($s0, $s1, $s2)

Τρεις τιμές πρέπει να «επιβιώσουν» μετά την κλήση της B, το X (στον $s0), το N (στον $s1) και το y (στον $s2)

Για να γίνει η κλήση B(X) πρέπει να αντιγράψουμε τον $s0 στον $a0

ΣΥΝΟΛΟ SAVE/RESTORE

= 4 / 4

Με Caller-save καταχωρητές

```
Aloop2:addui  $sp, $sp, -8
       sw     $ra, 0($sp) //push
```

```
       move   $t0, $a1 // i=N
Loop:  sw     $t0, 4($sp) //push i
       add    $a0, $t1, $t0 // X+i
       jal    B
       lw     $t0, 4($sp) //pop
       move   $t1, $v0
       addi   $t0, $t0, -1
       bnez   $t0, Loop
       move   $v0, $t0
```

```
       lw     $ra,  0($sp) //pop
       addui  $sp, $sp, 8
       jr     $ra
```

Εδώ η Aloop2 υλοποιείται με χρήση temporary καταχωρητών ($t0, $a0, $a1)

Η μία τιμή που πρέπει να «επιβιώσει» μετά την κλήση της B, το i, βρίσκεται στον $t0 ο οποίος πρέπει να σωθεί και να επαναφερθεί. Το X αποθηκεύεται στον $t1 το οποίο *δεν χρειάζεται* να σωθεί.

ΣΥΝΟΛΟ SAVE/RESTORE

= N+1 / N+1

Με Caller- **και** Callee-save καταχωρητές

```
Aloop2:addui  $sp, $sp, -8
       sw     $ra, 0($sp) //push
       sw     $s0, 4($sp) //push
```

```
       move   $s0, $a1 // i=N
Loop:  add    $a0, $t1, $s0
       jal    B
       move   $t1, $v0
       addi   $s0, $s0, -1
       bnez   $s0, Loop
       move   $v0, $s0
```

```
       lw     $s0, 4($sp) //pop
       lw     $ra, 0($sp) //pop
       addui  $sp, $sp, 8
       jr     $ra
```

Εδώ η Aloop2 υλοποιείται με χρήση και temporary ($a0 για το y) και saved καταχωρητών ($s0 για το i)

Η μία τιμή που πρέπει να «επιβιώσει» μετά την κλήση της B, το i, βρίσκεται στον $s0 οπότε δεν χρειάζεται να κάνουμε κάτι. Το X αποθηκεύεται στον $t1 το οποίο *δεν χρειάζεται* να σωθεί

ΣΥΝΟΛΟ SAVE/RESTORE

= 2 / 2

Ακόμα καλύτερα!

```
Aloop2:addui  $sp, $sp, -8
       sw     $ra, 0($sp) //push
       sw     $s0, 4($sp) //push
```

```
       move   $s0, $a1 // i=N
Loop:  add    $a0, $v0, $s0
       jal    B
       addi   $s0, $s0, -1
       bnez   $s0, Loop
       move   $v0, $s0
```

```
       lw     $s0, 4($sp) //pop
       lw     $ra, 0($sp) //pop
       addui  $sp, $sp, 8
       jr     $ra
```

Το Χ αποθηκεύεται στον $v0 και δεν χρειάζεται η move

ΣΥΝΟΛΟ SAVE/RESTORE = 2 / 2

Σύνολο εντολών:

Στατικές (πρόγραμμα) 13

Δυναμικές: 4*N+9

Ο μικρότερος και αποδοτικότερος κώδικας!