

Κεφάλαιο 2

Εντολές: Η γλώσσα του υπολογιστή

'Simplicity is not a simple thing'
-- Charlie Chaplin

Σύνολο εντολών

- Σύνολο εντολών (Instruction set) – Το «ρεπερτόριο» των εντολών ενός υπολογιστή
- Διαφορετικοί υπολογιστές έχουν διαφορετικά σύνολα εντολών
 - αλλά, με πολλά κοινά χαρακτηριστικά
- Οι πρώτοι υπολογιστές είχαν πολύ απλά σύνολα εντολών
 - Απλοποιημένη υλοποίηση
- Πολλοί σύγχρονοι επεξεργαστές έχουν επίσης απλά σύνολα εντολών
- Προσομοιωτής SPIM: <http://spimsimulator.sourceforge.net/>

Το σύνολο εντολών του MIPS

- Χρησιμοποιείται ως παράδειγμα σε όλο το βιβλίο
- Η MIPS Technologies (www.mips.com) έκανε εμπορικό τον Stanford MIPS
- Μεγάλο μερίδιο της αγοράς των πυρήνων (cores) 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 B και E

Αριθμητικές λειτουργίες

- Πρόσθεση και αφαίρεση, τρεις τελεστές (operands)
 - Δύο προελεύσεις και ένας προορισμός

`add a, b, c # a gets b + c`
- Όλες οι αριθμητικές λειτουργίες έχουν αυτή τη μορφή
- *Σχεδιαστική αρχή 1:* η απλότητα ευνοεί την κανονικότητα
 - Η απλότητα επιτρέπει μεγαλύτερη απόδοση με χαμηλότερο κόστος

Αριθμητικό παράδειγμα

- Κώδικας C:

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

- Μεταγλωττισμένος κώδικας MIPS:

```
add t0, g, h    # t0 = g + h
add t1, i, j    # t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Τελεστές-καταχωρητές

- 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
 - Τα δεδομένα των 32-bit ονομάζονται «λέξη» (“word”)
- Ονόματα του συμβολομεταφραστή (assembler)
 - \$t0, \$t1, ..., \$t9 για **προσωρινές** τιμές
 - \$s0, \$s1, ..., \$s7 για αποθηκευμένες **μεταβλητές**
- *Σχεδιαστική αρχή 2:* το μικρότερο είναι ταχύτερο
 - παραβολή με κύρια μνήμη: εκατομμύρια θέσεων

Τελεστές καταχωρητές

- Κώδικας C:

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

- οι f, g, h, i, j στους \$s0, \$s1, \$s2, \$s3, \$s4

- Μεταγλωττισμένος κώδικας MIPS:

add \$t0, \$s1, \$s2

add \$t1, \$s3, \$s4

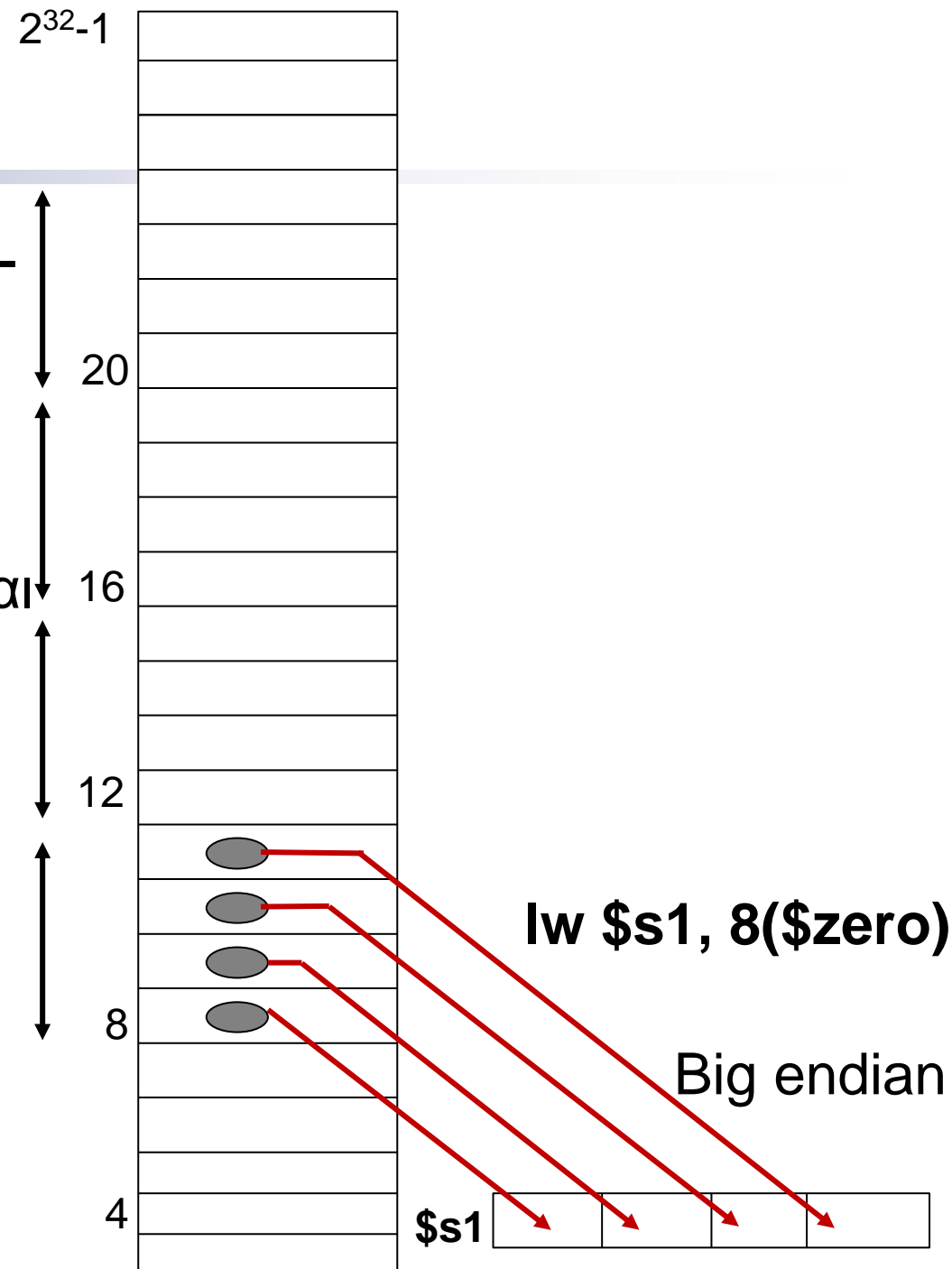
sub \$s0, \$t0, \$t1

Τελεστές μνήμης

- Η κύρια μνήμη χρησιμοποιείται για σύνθετα δεδομένα
 - Πίνακες (**arrays**), δομές (**structures**), δυναμικά δεδομένα
- Για να εφαρμοστούν αριθμητικές λειτουργίες
 - Φόρτωση (**Load**) τιμών από τη μνήμη σε καταχωρητές
 - Αποθήκευση (Store) αποτελέσματος από καταχωρητές στη μνήμη
- Η μνήμη διευθυνσιοδοτείται ανά byte (byte addressed)
 - Κάθε διεύθυνση (32-bit) προσδιορίζει **ένα** byte των 8 bit

Πρόσβαση Μνήμης

- Οι **λέξεις** (words) και οι **ημι-λέξεις** (half-words) είναι «ευθυγραμμισμένες» (“aligned”) στη μνήμη
 - Η διεύθυνση κάθε **λέξης** πρέπει να είναι **πολλαπλάσιο** του **4**
 - Η διεύθυνση κάθε **ημι-λέξης** πρέπει να είναι **πολλαπλάσιο** του **2**
- Ο MIPS είναι «**Μεγάλου άκρου**» (“Big Endian”)
 - Το περισσότερο σημαντικό byte βρίσκεται στη μικρότερη διεύθυνση μιας λέξης
 - Σύγκριση με «**Μικρού άκρου**» (“Little Endian”: το λιγότερο σημαντικό byte βρίσκεται στη μικρότερη διεύθυνση)



Παράδειγμα 1 με τελεστές μνήμης

- Κώδικας C:

`g = h + A[8];`

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

- Μεταγλωττισμένος κώδικας MIPS:

- Ο δείκτης 8 απαιτεί σχετική απόσταση (offset) ίση με 32

- 4 byte ανά λέξη

```
lw    $t0, 32($s3)    # load word
add   $s1, $s2, $t0
```

καταχωρητής βάσης

σχετική απόσταση

Παράδειγμα 2, με τελεστέους μνήμης

- Κώδικας C:

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

- `h` στον `$s2`, διεύθυνση τού `A` στον `$s3`

- Μεταγλωττισμένος κώδικας MIPS:

- Ο δείκτης 8 απαιτεί σχετική απόσταση 32

```
lw    $t0, 32($s3)    # load word
```

```
add   $t0, $s2, $t0
```

```
sw    $t0, 48($s3)    # store word
```

Καταχωρητές έναντι Μνήμης

- Οι καταχωρητές είναι ταχύτερα προσπελάσιμοι από τη μνήμη
- Οι λειτουργίες σε δεδομένα μνήμης απαιτούν φορτώσεις και αποθηκεύσεις
 - Εκτελούνται περισσότερες εντολές
- Ο μεταγλωττιστής πρέπει να χρησιμοποιεί τους καταχωρητές για μεταβλητές όσο περισσότερο γίνεται
 - Να «διασκορπίζει» (spill) στη μνήμη μόνο τις λιγότερο συχνά χρησιμοποιούμενες μεταβλητές

Άμεσοι τελεστές (immediate)

- Σταθερά δεδομένα καθορίζονται σε μια εντολή
`addi $s3, $s3, 4`
- Δεν υπάρχει εντολή άμεσης αφαίρεσης (subtract immediate)
 - Απλώς χρησιμοποιείται μια αρνητική σταθερά
`addi $s2, $s1, -1`
- *Σχεδιαστική αρχή 3:* Κάνε τη συνηθισμένη περίπτωση γρήγορη
 - Οι μικρές σταθερές είναι συνηθισμένες
 - Ο άμεσος τελεστής αποφεύγει μια εντολή φόρτωσης (load)

Η σταθερά Μηδέν

- Ο καταχωρητής 0 του MIPS (\$zero) είναι η σταθερά 0
 - Δεν μπορεί να γραφεί με άλλη τιμή
- Χρήσιμη για συνηθισμένες λειτουργίες
 - Π.χ., μετακίνηση (move) μεταξύ καταχωρητών
`add $t2, $s1, $zero`

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2+20]=\$s1; \$s1=0 \text{ or } 1$	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2 20$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
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 ($\$s1 \neq \$s2$) 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 on less than unsigned	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if ($\$s2 < 20$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if ($\$s2 < 20$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant unsigned
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	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

Απρόσημοι δυαδικοί ακέραιοι

- Με δεδομένο έναν αριθμό των n bit

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Εύρος: 0 έως $+2^n - 1$

- Παράδειγμα

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Με χρήση 32 bit

- 0 ως $+4,294,967,295$

Προσημασμένοι στο συμπλήρωμα του 2

- Με δεδομένο έναν αριθμό των n bit

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Εύρος: -2^{n-1} ως $+2^{n-1} - 1$

- Παράδειγμα

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Με χρήση 32 bit

- $-2,147,483,648$ ως $+2,147,483,647$

Προσημασμένοι στο συμπλήρωμα του 2

- Το bit 31 είναι το bit προσήμου
 - 1 για αρνητικούς αριθμούς
 - 0 για μη αρνητικούς αριθμούς (θετικούς και μηδέν)
- Μερικοί συγκεκριμένοι αριθμοί
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Ο πιο αρνητικός: 1000 0000 ... 0000
 - Ο πιο θετικός: 0111 1111 ... 1111

Προσημασμένη άρνηση

- Συμπλήρωμα και πρόσθεση του 1
 - «Συμπλήρωμα» σημαίνει $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111\dots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Παράδειγμα: βρείτε τον αντίθετο (άρνηση) του +2
 - $+2 = 0000\ 0000\dots0010_2$
 - $-2 = 1111\ 1111\dots1101_2 + 1$
 $= 1111\ 1111\dots1110_2$

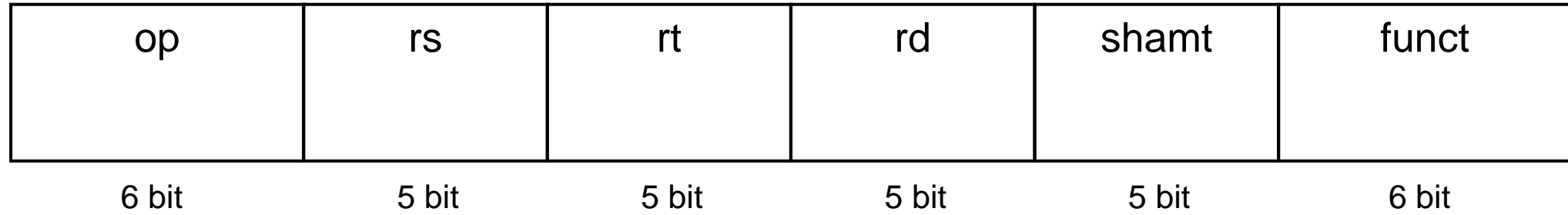
Επέκταση προσήμου

- Αναπαράσταση ενός αριθμού με περισσότερα bit
 - Διατήρηση της αριθμητικής τιμής
- Στο σύνολο εντολών του MIPS
 - **addi** : επέκταση προσήμου στη τιμή του άμεσου (immediate) ορίσματος
 - **l b, l h**: **επέκταση προσήμου** στο byte/ημιλέξη που φορτώνεται
 - **beq, bne**: επέκταση προσήμου στη μετατόπιση (displacement)
- Επανάληψη του bit προσήμου προς τα αριστερά
- Παραδείγματα: επέκταση 8-bit σε 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110

Αναπαράσταση εντολών

- Εντολές MIPS
 - Κωδικοποιούνται ως λέξεις εντολής των 32 bit
 - Μικρός αριθμός μορφών (formats) για τον κωδικό λειτουργίας (opcode), τους αριθμούς καταχωρητών, κλπ. ...
 - Απλότητα - κανονικότητα!
- Αριθμοί καταχωρητών
 - \$t0 – \$t7 είναι οι καταχωρητές 8 – 15
 - \$t8 – \$t9 είναι οι καταχωρητές 24 – 25
 - \$s0 – \$s7 είναι οι καταχωρητές 16 – 23

Εντολές μορφής R του MIPS



■ Πεδία εντολής

- **op**: κωδικός λειτουργίας (opcode)
- **rs**: αριθμός πρώτου καταχωρητή προέλευσης
- **rt**: αριθμός δεύτερου καταχωρητή προέλευσης
- **rd**: αριθμός καταχωρητή προορισμού
- **shamt**: ποσότητα ολίσθησης (00000 για τώρα)
- **funct**: κωδικός συνάρτησης (επεκτείνει τον κωδικό λειτουργίας)

Παράδειγμα μορφής R

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

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
---------	------	------	------	---	-----

0	17	18	8	0	32
---	----	----	---	---	----

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

$$00000010001100100100000000100000_2 = 02324020_{16}$$

Δεκαεξαδικό

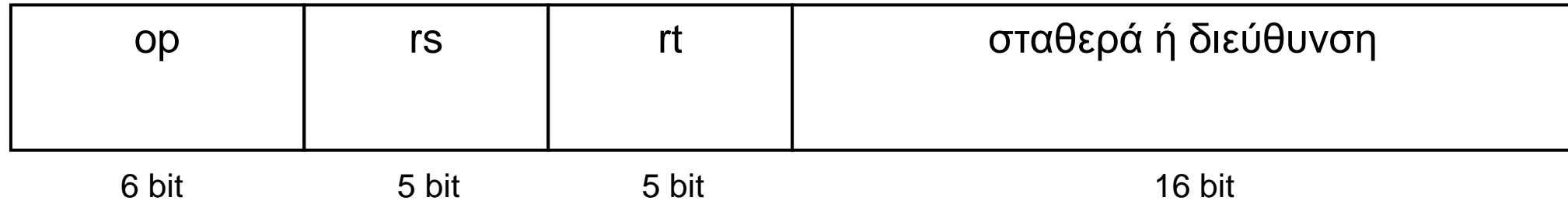
- Βάση 16
 - Συμπαγής αναπαράσταση σειρών bit
 - 4 bit ανά δεκαεξαδικό ψηφίο

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Παράδειγμα: **eca86420**

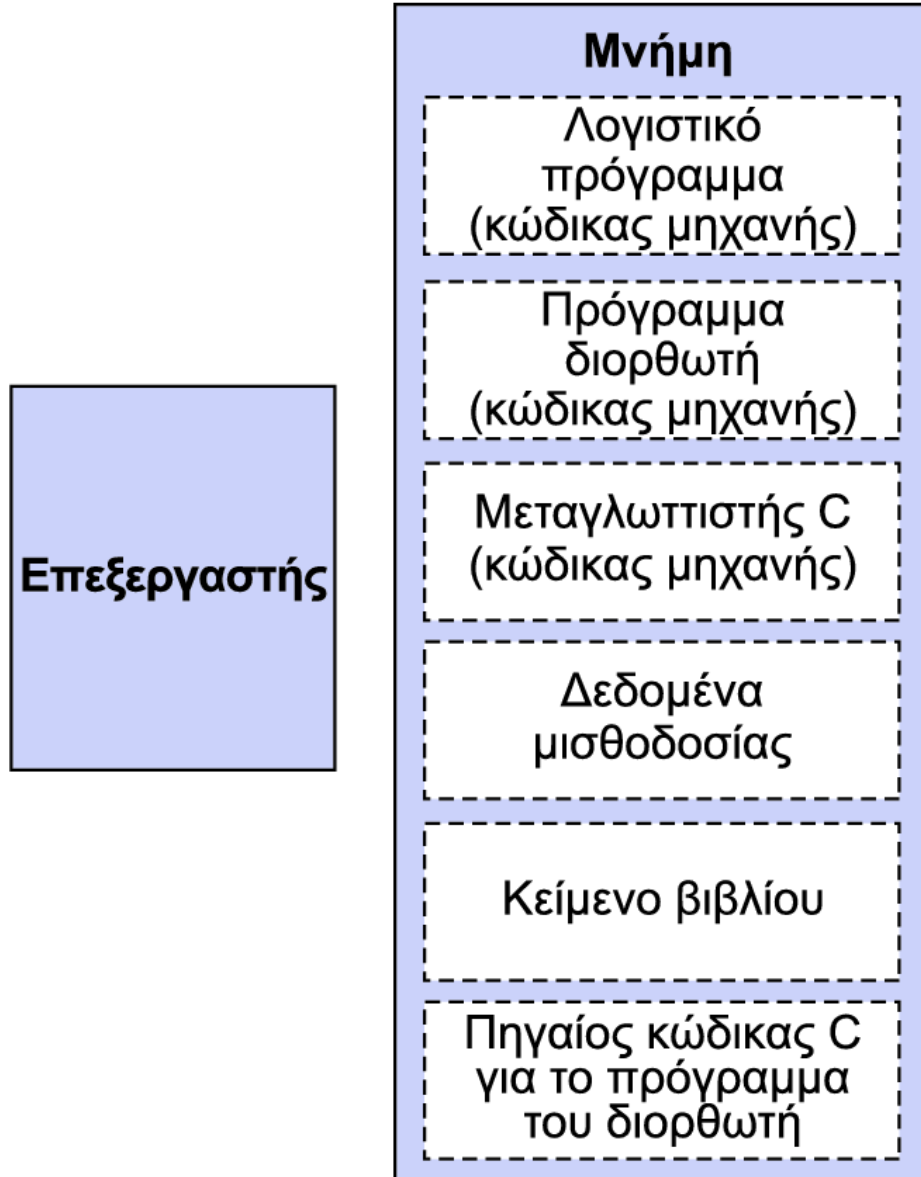
- | | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| e | c | a | 8 | 6 | 4 | 2 | 0 |
| 1110 | 1100 | 1010 | 1000 | 0110 | 0100 | 0010 | 0000 |

Εντολές μορφής I του MIPS



- Άμεσες αριθμητικές εντολές και εντολές load/store
 - rt: αριθμός καταχωρητή προορισμού ή προέλευσης
 - Σταθερά: -2^{15} έως $+2^{15} - 1$
 - Διεύθυνση: σχετική απόσταση (offset) που προστίθεται στη διεύθυνση βάσης που περιέχει ο rs

Υπολογιστές Αποθηκευμένου Προγράμματος



- Οι εντολές αναπαρίστανται σε δυαδικό, όπως τα δεδομένα
- Οι εντολές και τα δεδομένα αποθηκεύονται στη μνήμη
- Προγράμματα μπορούν να επενεργούν σε προγράμματα
 - π.χ, compilers, linkers, ...
- Η δυαδική συμβατότητα επιτρέπει τα μεταγλωττισμένα προγράμματα να εκτελούνται σε διαφορετικούς υπολογιστές
 - Καθιερωμένες ISA

Λογικές Λειτουργίες

- Εντολές για χειρισμούς ανά bit

Λειτουργία	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

- Χρήσιμες για εξαγωγή και εισαγωγή ομάδων bit σε μια λέξη

Λειτουργίες ολίσθησης

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

- shamt: αριθμός θέσεων ολίσθησης
- Shift left logical (αριστερή λογική ολίσθηση)
 - Αριστερή ολίσθηση και συμπλήρωση με bit 0
 - `sl l` κατά i bit πολλαπλασιάζει με 2^i
- Shift right logical (δεξιά λογική ολίσθηση)
 - Δεξιά ολίσθηση και συμπλήρωση με bit 0
 - `sr l` κατά i bit διαιρεί με 2^i (απρόσημοι μόνο)

Λειτουργίες AND

- Χρήσιμες για την «απόκρυψη» (masking) bit σε μια λέξη
 - Επιλογή κάποιων bit, μηδενισμών των άλλων

and \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

Λειτουργίες OR

- Χρήσιμες για να συμπεριληφθούν κάποια bit σε μια λέξη
 - Κάποια bit τίθενται στο 1, τα υπόλοιπα αμετάβλητα

or \$t0, \$t1, \$t2

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

Λειτουργίες NOT

- Χρήσιμες για την αντιστροφή bit σε μια λέξη
 - Αλλαγή του 0 σε 1, και του 1 σε 0
- Ο MIPS διαθέτει εντολή NOR των 3 τελεστών
 - $a \text{ NOR } b == \text{NOT } (a \text{ OR } b)$

`nor $t0, $t1, $zero`

Καταχωρητής 0:
πάντα ίσος με
μηδέν

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 1100 0011 1111 1111

Λειτουργίες διακλάδωσης

- Διακλάδωση (branch) σε μια εντολή αν μια συνθήκη είναι αληθής
 - Διαφορετικά, συνέχισε ακολουθιακά
- `beq rs, rt, L1`
 - αν (`rs == rt`) διακλάδωση στην εντολή με ετικέτα L1
- `bne rs, rt, L1`
 - αν (`rs != rt`) διακλάδωση στην εντολή με ετικέτα L1
- `j L1`
 - άλμα **χωρίς συνθήκη** στην εντολή με ετικέτα L1

Μεταγλώττιση εντολών If

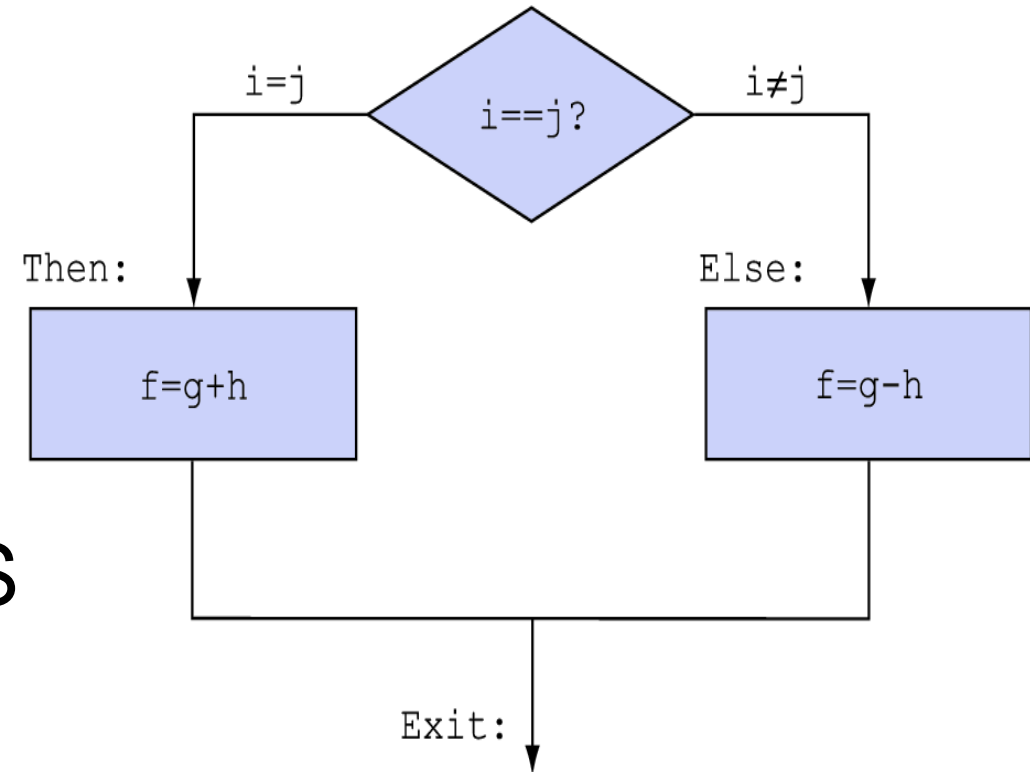
```
if (i==j) f = g+h;  
else f = g-h;
```

- f, g, h, i, j
- \$s0, \$s1, \$s2, \$s3, \$s4

■ Μεταγλωττισμένος κώδικας MIPS

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

ο συμβολομεταφραστής υπολογίζει τις διευθύνσεις



Μεταγλώττιση εντολών Loop

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

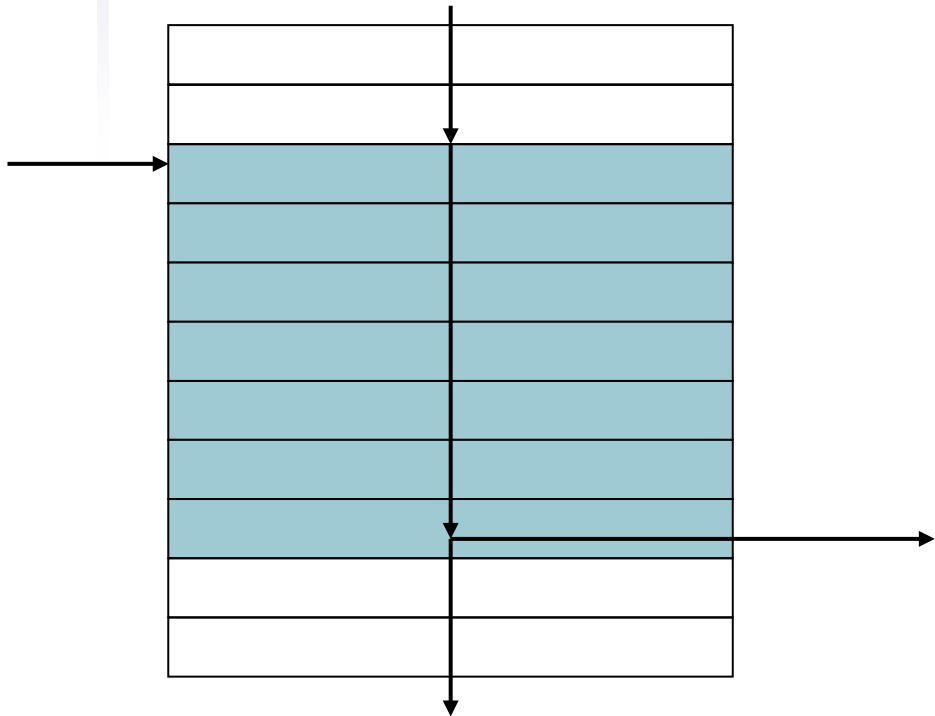
- i στον \$s3,
- k στον \$s5,
- η δ/νση του save στον \$s6

■ Μεταγλωττισμένος κώδικας MIPS:

```
Loop:  slt    $t1, $s3, 2
        add   $t1, $t1, $s6
        lw    $t0, 0($t1)
        bne   $t0, $s5, Exit
        addi   $s3, $s3, 1
        j     Loop
Exit:  ...
```

Βασικά μπλοκ

- Ένα **βασικό μπλοκ** (basic block) είναι μια ακολουθία εντολών **χωρίς**:
 - Διακλαδώσεις (εκτός από το τέλος)
 - Προορισμούς διακλάδωσης (εκτός από την αρχή)



- Ένας μεταγλωττιστής προσδιορίζει **βασικά μπλοκ** για βελτιστοποίηση
- Ένας προηγμένος επεξεργαστής μπορεί να επιταχύνει την εκτέλεση των **βασικών μπλοκ**

..άλλες λειτουργίες συνθήκης

- Το αποτέλεσμα παίρνει τη τιμή 1 αν μια συνθήκη είναι αληθής
 - Διαφορετικά, παίρνει τη τιμή 0
- `slt rd, rs, rt`
 - αν $(rs < rt)$ $rd = 1$ · διαφορετικά $rd = 0$
- `slti rt, rs, constant`
 - αν $(rs < constant)$ $rt = 1$ · διαφορετικά $rt = 0$
- Χρήση σε συνδυασμό με τις `beq`, `bne`

```
slt $t0, $s1, $s2 # if ($s1 < $s2)
bne $t0, $zero, L  # branch to L
```

Σχεδίαση εντολών διακλάδωσης

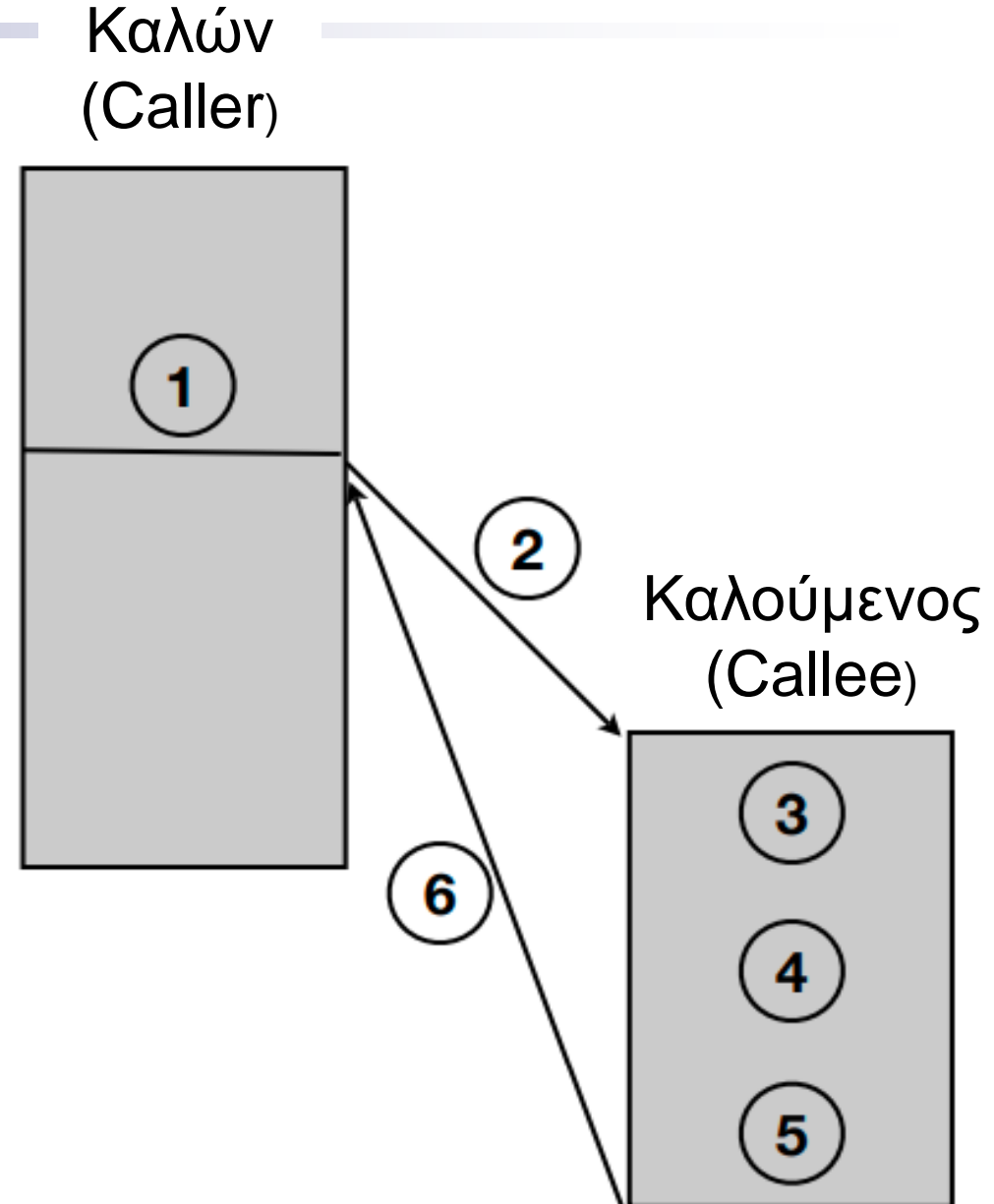
- Γιατί όχι `bl t`, `bge`, κλπ.;
- Το υλικό για τις `<`, `≥`, ... είναι πιο αργό από αυτό για τις `=`, `≠`
 - Ο συνδυασμός συνθηκών για μια διακλάδωση περιλαμβάνει περισσότερη δουλειά ανά εντολή, και απαιτεί πιο αργό ρολόι
 - Επιβαρύνονται όλες οι εντολές!
- Οι `beq` και `bne` είναι η συνήθης περίπτωση
- Καλός σχεδιαστικός συμβιβασμός

Προσημασμένες και απρόσημες πράξεις

- Προσημασμένη σύγκριση: `sl t, sl ti`
- Απρόσημη σύγκριση: `sl tu, sl tui`
- Παράδειγμα
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
 - `sl t $t0, $s0, $s1 # προσημασμένη`
 - $-1 < +1 \Rightarrow \$t0 = 1$
 - `sl tu $t0, $s0, $s1 # απρόσημη`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

Κλήση/ολοκλήρωση Διαδικασίας

1. Τοποθέτηση παραμέτρων σε καταχωρητές
2. Μεταφορά ελέγχου στη διαδικασία
3. Απόκτηση χώρου αποθήκευσης για τη διαδικασία
4. Εκτέλεση λειτουργιών της διαδικασίας
5. Τοποθέτηση αποτελέσματος σε καταχωρητή για τον καλούντα
6. Επιστροφή στη θέση της κλήσης



Χρήση καταχωρητών

- \$a0 – \$a3: ορίσματα (καταχωρητές 4 – 7)
- \$v0, \$v1: τιμές αποτελέσματος (καταχωρητές 2 και 3)
- \$t0 – \$t9: προσωρινοί (temporary)
 - Μπορούν να γραφούν με νέες τιμές από τον καλούμενο
- \$s0 – \$s7: αποθηκευμένοι (saved)
 - Πρέπει να αποθηκευτούν/επαναφερθούν από τον καλούμενο
- \$gp: καθολικός δείκτης (global pointer) για στατικά δεδομένα (καταχ. 28)
- \$sp: δείκτης στοίβας (stack pointer) (καταχ.29)
- \$fp: δείκτης πλαισίου (frame pointer) (καταχ.30)
- \$ra: δ/νση επιστροφής (return address) (καταχ. 31)

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

Εντολές κλήσης διαδικασίας

- Κλήση διαδικασίας: jump and link

j al ProcedureLabel

- Η δ/νση της επόμενης εντολής γράφεται στον \$ra
- Άλμα στη διεύθυνση προορισμού

- Επιστροφή από διαδικασία: jump on register

j r \$ra

- Αντιγράφει τον \$ra στον μετρητή προγράμματος (program counter)
- Μπορεί επίσης να χρησιμοποιηθεί για υπολογισμένα άλματα π.χ., για εντολές case/switch

Παράδειγμα διαδικασίας “φύλλου”

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Ορίσματα g, h, i, j στους \$a0, \$a1, \$a2, \$a3
- f στον \$s0 (συνεπώς πρέπει να αποθηκευθεί ο \$s0 στη στοίβα)
- Αποτέλεσμα στον \$v0

Παράδειγμα διαδικασίας “φύλλου”

leaf_example:

addi	\$sp,	\$sp,	- 4	Αποθήκευση \$s0 στη στοίβα
sw	\$s0,	0(\$sp)		
add	\$t0,	\$a0,	\$a1	Σώμα διαδικασίας
add	\$t1,	\$a2,	\$a3	
sub	\$s0,	\$t0,	\$t1	
add	\$v0,	\$s0,	\$zero	Αποτέλεσμα
lw	\$s0,	0(\$sp)		Επαναφορά του \$s0
addi	\$sp,	\$sp,	4	
jr	\$ra			Επιστροφή

Διαδικασίες “μη φύλλα”

- Διαδικασίες που καλούν άλλες διαδικασίες
- Για ένθετη (nested) κλήση, ο καλών πρέπει να αποθηκεύσει στη στοίβα:
 - Τη διεύθυνση επιστροφής του
 - Όποια ορίσματα και προσωρινές τιμές χρειάζονται μετά την κλήση
- Επαναφορά από τη στοίβα μετά την κλήση

Παράδειγμα διαδικασίας “μη φύλλου”

- Κώδικας C:

```
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

- Όρισμα n στον \$a0
- Αποτέλεσμα fact στον \$v0

fact:

```
addi    $sp, $sp, -8    # adjust stack for 2 items
sw      $ra, 4($sp)     # save the return address
sw      $a0, 0($sp)     # save the argument n
slti    $t0, $a0, 1     # test for n < 1
beq     $t0, $zero, L1  # if n >= 1, go to L1
addi    $v0, $zero, 1   # return 1
addi    $sp, $sp, 8     # pop 2 items off stack
jr      $ra             # return to caller
```

```
L1: addi $a0, $a0, -1    # n >= 1: argument gets (n - 1)
jal     fact            # call fact with (n - 1)
lw      $a0, 0($sp)     # return from jal: restore argument n
lw      $ra, 4($sp)     # restore the return address
addi    $sp, $sp, 8     # adjust stack pointer to pop 2 items
mul     $v0, $a0, $v0    # return n * fact (n - 1)
jr      $ra             # return to the caller
```

Παράδειγμα διαδικασίας μη-φύλλου

fact:	addi	\$sp, \$sp, - 8	# adjust stack for 2 items
	sw	\$ra, 4(\$sp)	# save return address
	sw	\$a0, 0(\$sp)	# save argument
	slti	\$t0, \$a0, 1	# test for $n < 1$
	beq	\$t0, \$zero, L1	
	addi	\$v0, \$zero, 1	# if so, result is 1
	addi	\$sp, \$sp, 8	# pop 2 items from stack
	jr	\$ra	# and return
L1:	addi	\$a0, \$a0, - 1	# else decrement n
	jal	fact	# recursive call
	lw	\$a0, 0(\$sp)	# restore original n
	lw	\$ra, 4(\$sp)	# and return address
	addi	\$sp, \$sp, 8	# pop 2 items from stack
	mul	\$v0, \$a0, \$v0	# multiply to get result
	jr	\$ra	# and return

2³²-1

n → \$a0
n! ← \$v0

492: ...
496: ...
500: **jal** fact
504: ...
508: ...

fact 800: addi \$sp,\$sp,-8
804: ...

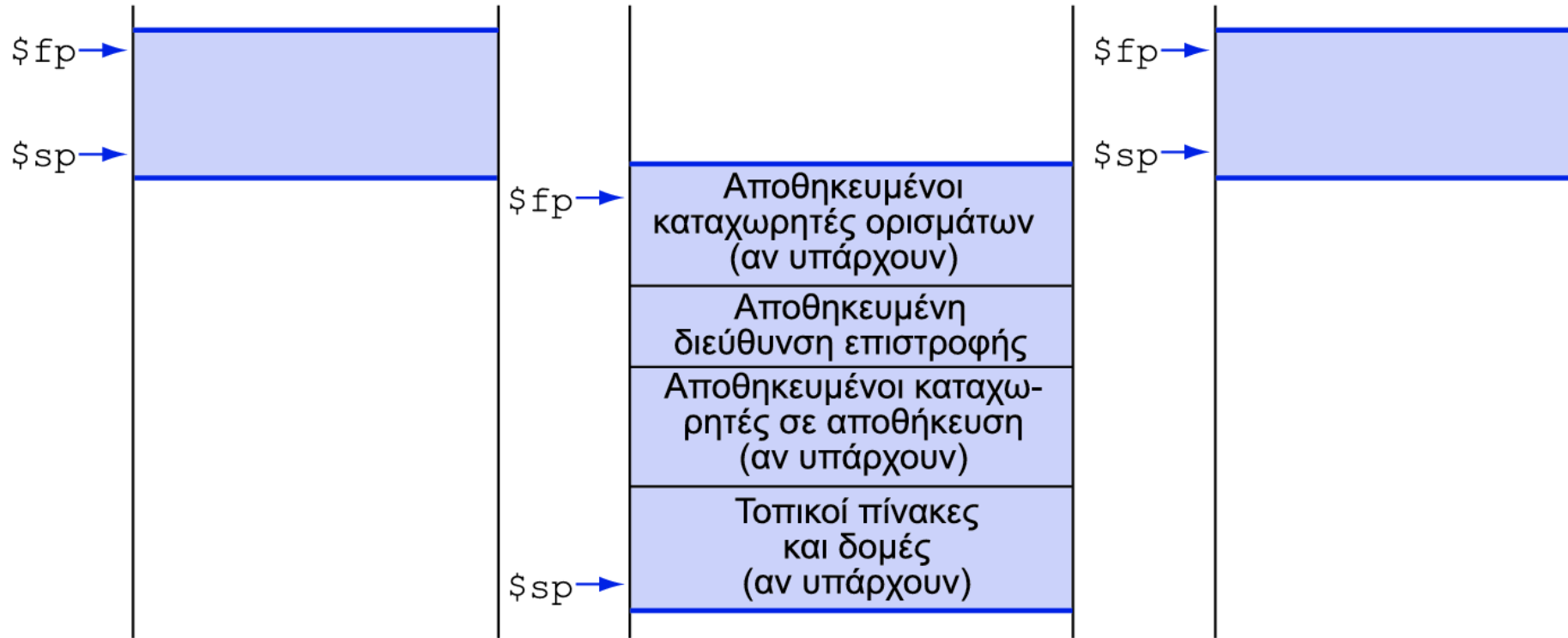
850: **jal** fact
...
892: ...
896: **jr** \$ra

\$sp

4
0

Τοπικά δεδομένα στη στοίβα

Υψηλή διεύθυνση



Χαμηλή διεύθυνση α.

β.

γ.

- Τοπικά δεδομένα δεσμεύονται από τον *καλούμενο*

- π.χ., οι αυτόματες μεταβλητές της C

- Πλαίσιο διαδικασίας (procedure frame) ή εγγραφή ενεργοποίησης (activation record)

- Χρησιμοποιείται από μερικούς μεταγλωττιστές για το χειρισμό της αποθήκευσης της στοίβας

Διάταξη της μνήμης

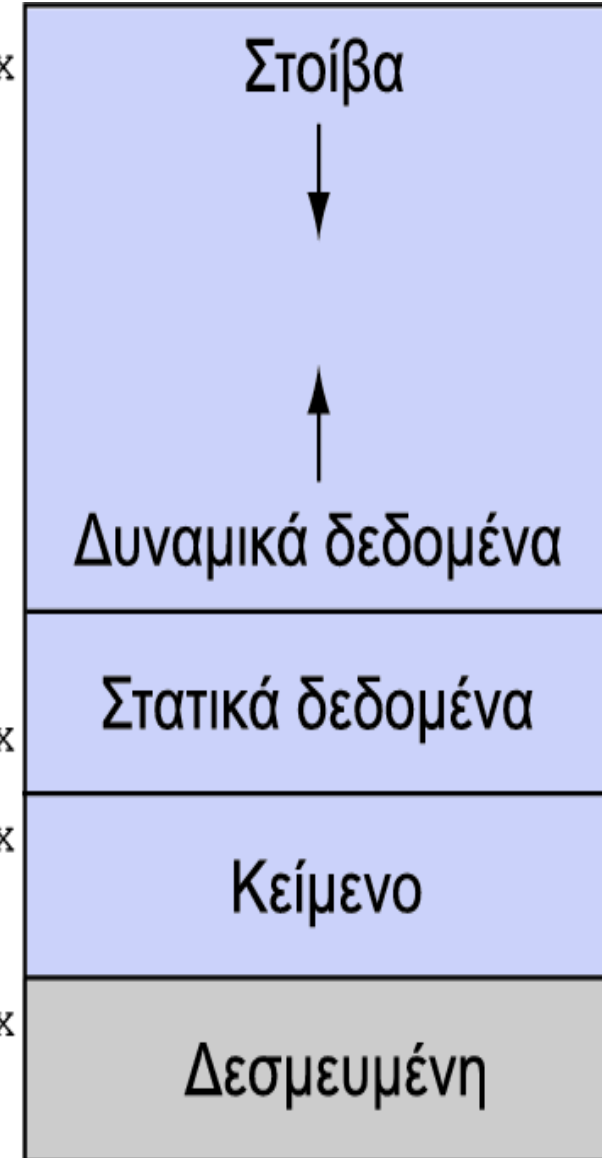
- Κείμενο (Text): κώδικας του προγράμματος
- Στατικά δεδομένα (Static data): καθολικές μεταβλητές
 - π.χ., στατικές μεταβλητές της C, πίνακες σταθερών (constant arrays) και συμβολοσειρές (strings)
 - Ο \$gp παίρνει αρχική τιμή που επιτρέπει ±σχετικές αποστάσεις μέσα στο τμήμα αυτό
- Δυναμικά δεδομένα: σωρός (heap)
 - π.χ., malloc στη C, new στη Java
- Στοίβα (stack): αυτόματη αποθήκευση

\$sp → 7fff fffc_{hex}

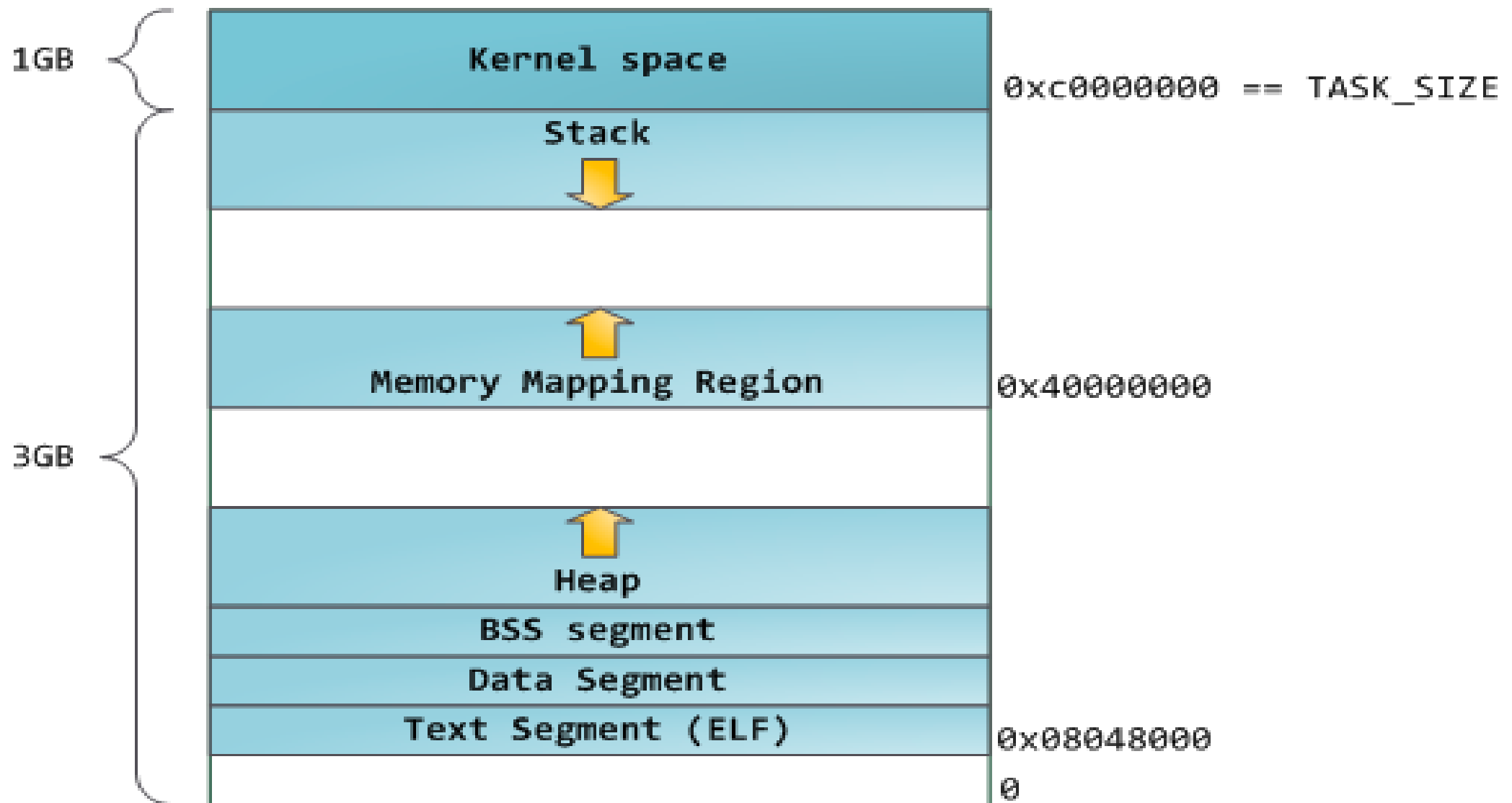
\$gp → 1000 8000_{hex}
1000 0000_{hex}

pc → 0040 0000_{hex}

0








Απεικόνιση μνήμης Linux



Δεδομένα χαρακτήρων

- Σύνολα χαρακτήρων σε κωδικοποίηση byte
 - ASCII: 128 χαρακτήρες
 - 95 γραφικής αναπαράστασης, 33 ελέγχου
 - Latin-1: 256 χαρακτήρες
 - ASCII, +96 επιπλέον χαρακτήρες γραφικής αναπαράστασης
- Unicode: σύνολο χαρακτήρων 32-bit
 - Χρήση σε Java, και σε wide characters της C++, ...
 - Τα περισσότερα αλφάβητα του κόσμου, και σύμβολα
 - UTF-8, UTF-16: κωδικοποιήσεις μεταβλητού μήκους

				
1D012	1D022	1D032	1D042	1D052

Λειτουργίες Byte/Ημιλέξης

`l b rt, offset(rs)` `l h rt, offset(rs)`

- Επέκταση προσήμου στα 32 bit στον `rt`

`l bu rt, offset(rs)` `l hu rt, offset(rs)`

- Επέκταση μηδενικού στα 32 bit στον `rt`

`sb rt, offset(rs)` `sh rt, offset(rs)`

- Αποθήκευση (store) μόνο του δεξιότερου byte/ημιλέξης

Εντολές byte

lb \$s1, 2(\$s2)

\$s2:

0x80000000

\$s1:

0xFFFFFFFFAA

lbu \$s1, 2(\$s2)

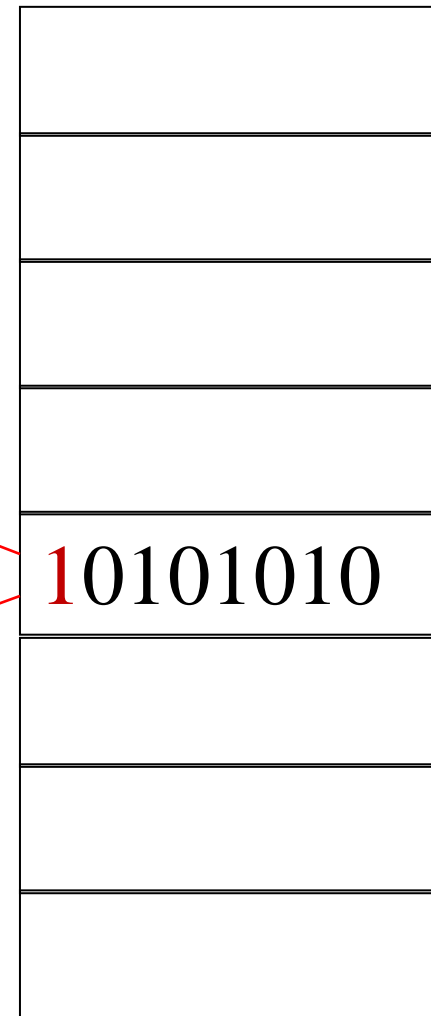
\$s2:

0x80000000

\$s1:

0x000000AA

MNHHMH



Παράδειγμα αντιγραφής string

- Κώδικας C (απλοϊκός):
 - Συμβολοσειρά (string) που τερματίζεται με μηδενικό χαρακτήρα (null char)

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

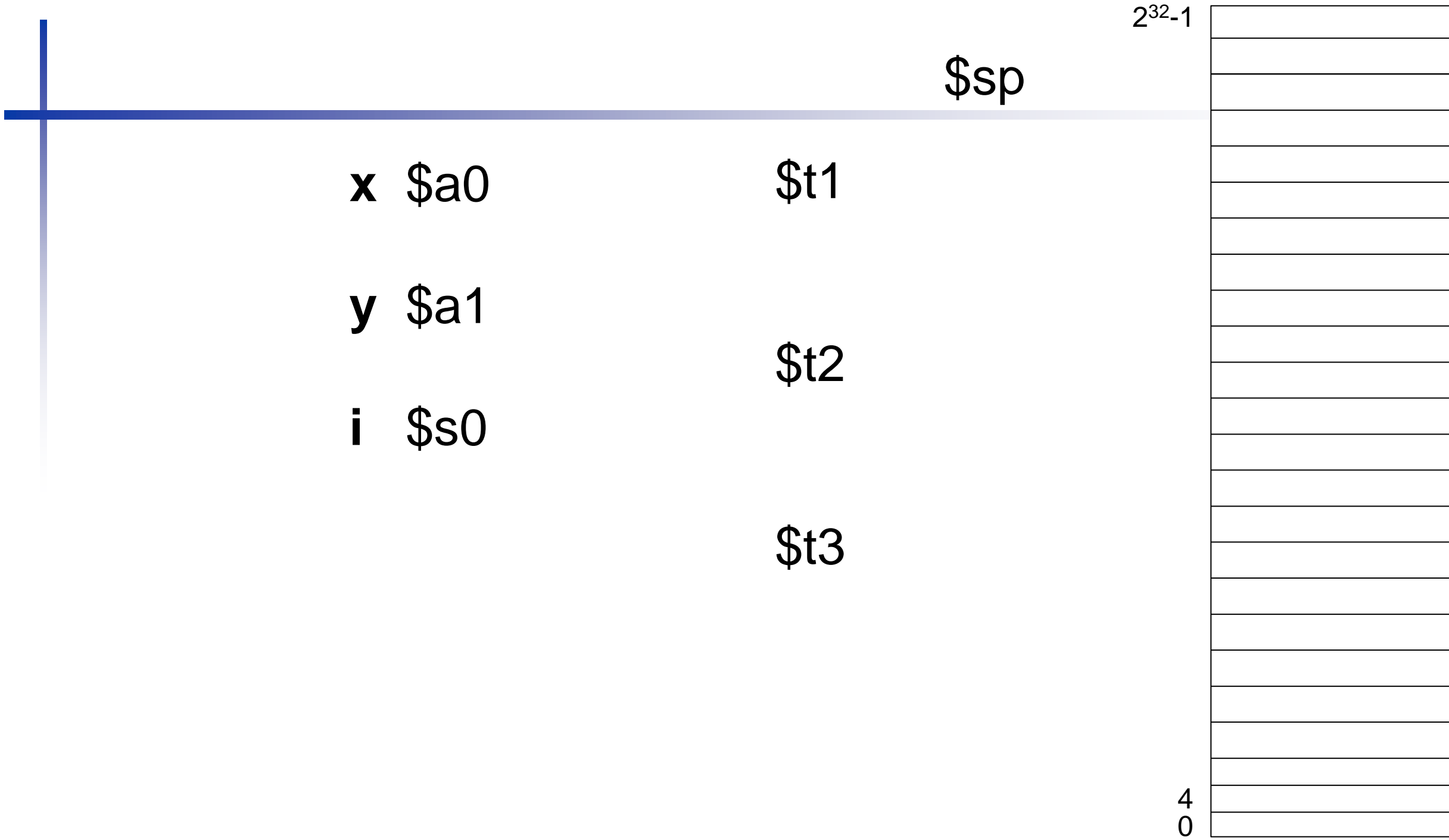
- Διευθύνσεις των x, y στον \$a0, \$a1
- Το i στον \$s0

Παράδειγμα αντιγραφής string

strcpy:

	addi	\$sp,	\$sp,	- 4	# adjust stack for 1 item
	sw	\$s0,	0(\$sp)		# save \$s0
<hr/>					
	add	\$s0,	\$zero,	\$zero	# i = 0
L1:	add	\$t1,	\$s0,	\$a1	# addr of y[i] in \$t1
	lbu	\$t2,	0(\$t1)		# \$t2 = y[i]
	add	\$t3,	\$s0,	\$a0	# addr of x[i] in \$t3
	sb	\$t2,	0(\$t3)		# x[i] = y[i]
	beq	\$t2,	\$zero,	L2	# exit loop if y[i] == 0
	addi	\$s0,	\$s0,	1	# i = i + 1
	j	L1			# next iteration of loop
L2:	lw	\$s0,	0(\$sp)		# restore saved \$s0
	addi	\$sp,	\$sp,	4	# pop 1 item from stack
	jr	\$ra			# and return

x, y στον \$a0, \$a1, το i στον \$s0



Σταθερές των 32 bit

- Οι περισσότερες σταθερές είναι μικρές
 - Ένα άμεσο πεδίο των 16 bit είναι αρκετό
- Για τις περιστρεφόμενες σταθερές των 32 bit

`lui rt, constant`

- Αντιγράφει τη σταθερά των 16 bit στα 16 αριστερά bit του rt
- Μηδενίζει τα δεξιά 16 bit του rt

\$s0

`lui $s0, 61`

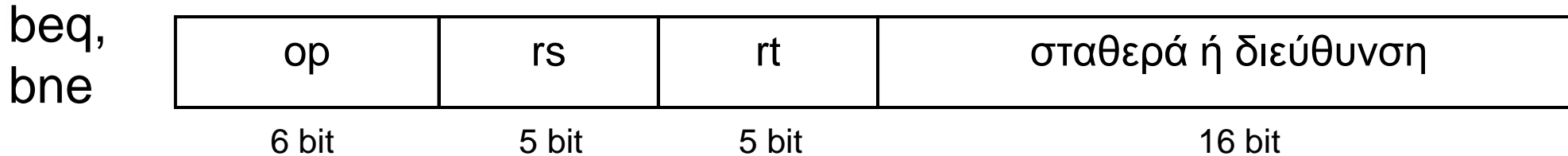
0000 0000 0011 1101	0000 0000 0000 0000
---------------------	---------------------

`ori $s0, $s0, 2304`

0000 0000 0011 1101	0000 1001 0000 0000
---------------------	---------------------

Διευθυνσιοδότηση διακλαδώσεων

- Οι εντολές διακλάδωσης (branch) καθορίζουν
 - Opcode, δύο καταχωρητές, δ/νση προορισμού
- Οι περισσότεροι προορισμοί διακλάδωσης είναι κοντά στην εντολή διακλάδωσης
 - Προς τα εμπρός και προς τα πίσω



- Διευθυνσιοδότηση σχετική ως προς PC (PC-relative addressing)
 - $\Delta/\nu\sigma\eta \text{ προορισμού} = PC + \text{offset} \times 4$
 - Ο PC είναι ήδη αυξημένος κατά 4

Παραδείγματα

beq \$s1,\$s2,25

if (\$s1 == \$s2) go to:

$PC \leftarrow PC + 4 + 25 \times 4$

$PC \leftarrow PC + 4 + 100$

bne \$s1,\$s2,15

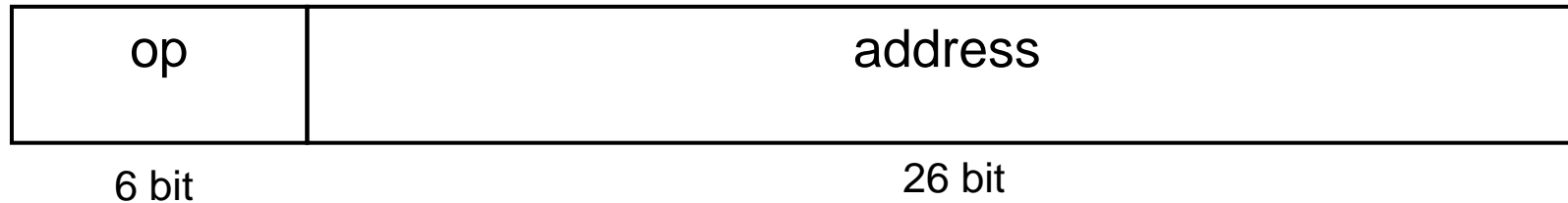
if (\$s1 != \$s2) go to

$PC \leftarrow PC + 4 + 15 \times 4$

$PC \leftarrow PC + 4 + 60$

Διευθυνσιοδότηση άλματος

- Οι προορισμοί άλματος (για τις εντολές j και $j\ al$) μπορεί να βρίσκονται οπουδήποτε στο τμήμα κειμένου (κώδικα)



- Ψευδο-άμεση (Pseudo-Direct) διευθυνσιοδότηση άλματος
 - Δ/νση προορισμού = $PC_{31...28} : (\text{address} \times 4)$

Παραδείγματα

j 2500

$$PC \leftarrow 2500 \times 4 = 10000$$

jal 1500

$$\$ra = PC + 4;$$

$$PC \leftarrow 1500 \times 4 = 6000$$

Παράδειγμα διεύθυνσης προορισμού

- Κώδικας βρόχου από προηγούμενο παράδειγμα
 - Υποθέτουμε ότι το **Loop** είναι στη θέση **80000**

```
Loop: sl1    $t1, $s3, 2
      add    $t1, $t1, $s6
      lw     $t0, 0($t1)
      bne    $t0, $s5, Exit
      addi   $s3, $s3, 1
      j      Loop
```

Exit: ...

80000

80004

80008

80012

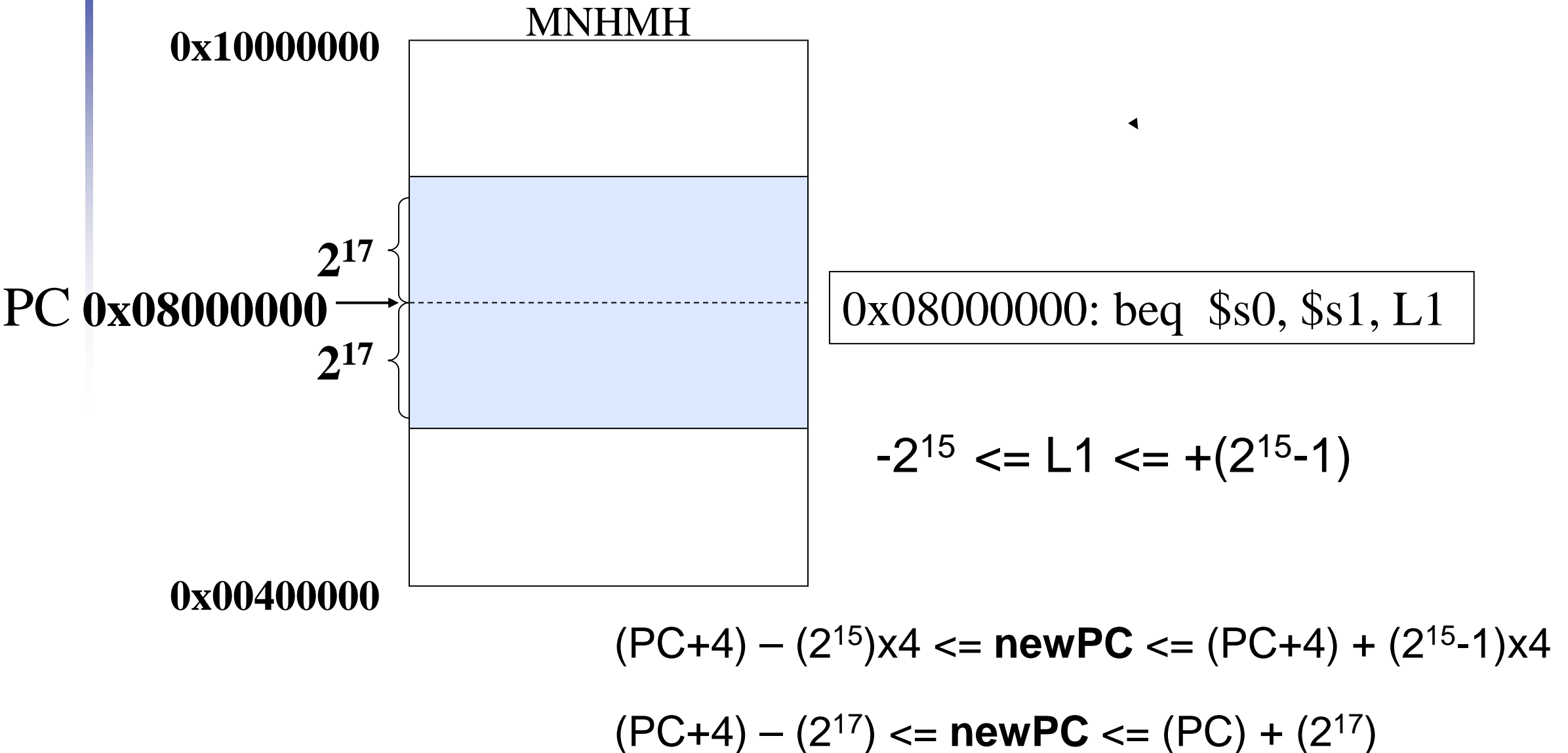
80016

80020

80024

0	0	19	9	4	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

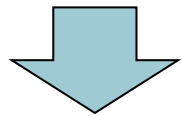
Όρια διακλάδωσης beq



Μακρινή Διακλάδωση

- Αν ο προορισμός της διακλάδωσης είναι πολύ μακριά για να κωδικοποιηθεί στα 16 bit του πεδίου σχετικής απόστασης (offset), ο συμβολομεταφραστής ξαναγράφει τον κώδικα
- Παράδειγμα

beq \$s0, \$s1, L1



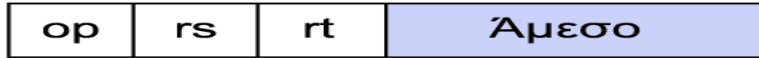
bne \$s0, \$s1, L2

j L1

L2: ...

Περίληψη τρόπων διευθ/σης

1. Άμεση διευθυνσιοδότηση



2. Διευθυνσιοδότηση μέσω καταχωρητή



Καταχωρητές
Καταχωρητής

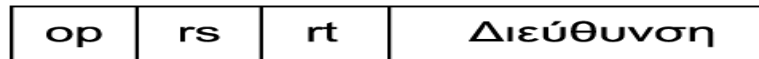
3. Διευθυνσιοδότηση βάσης



Μνήμη

Byte Ημιλέξη Λέξη

4. Σχετική διευθυνσιοδότηση ως προς PC



Μνήμη

Λέξη

5. Ψευδο-απευθείας διευθυνσιοδότηση



Μνήμη

Λέξη

Ψευδοεντολές συμβολομεταφραστή

- Οι περισσότερες εντολές του συμβολομεταφραστή αναπαριστούν εντολές μηχανής **μία προς μία**
- Οι **Ψευδοεντολές** (pseudoinstructions): δημιουργήματα τού συμβολομεταφραστή

move \$t0, \$t1 → add \$t0, \$zero, \$t1

```
blt $t0, $t1, L    → slt $at, $t0, $t1
                    bne $at, $zero, L
```

Παράδειγμα ταξινόμησης σε C

- Δείχνει τη χρήση των εντολών συμβολικής γλώσσας σε μια συνάρτηση ταξινόμησης φυσαλίδας (bubble sort) C
- Διαδικασία swap (φύλλο)

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

- Το v στον \$a0, το k στον \$a1, το temp στον \$t0

Η διαδικασία Swap

swap:	sll \$t1, \$a1, 2	# \$t1 = k * 4
	add \$t1, \$a0, \$t1	# \$t1 = v+(k*4)
		# (address of v[k])
	lw \$t0, 0(\$t1)	# \$t0 (temp) = v[k]
	lw \$t2, 4(\$t1)	# \$t2 = v[k+1]
	sw \$t2, 0(\$t1)	# v[k] = \$t2 (v[k+1])
	sw \$t0, 4(\$t1)	# v[k+1] = \$t0 (temp)
	jr \$ra	# return to calling routine

Η διαδικασία Sort σε C

- Μη φύλλο (καλεί τη swap)

```
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);
        }
    }
}
```

- Το v στον \$a0, το k στον \$a1, το i στον \$s0, το j στον \$s1

Το σώμα της διαδικασίας

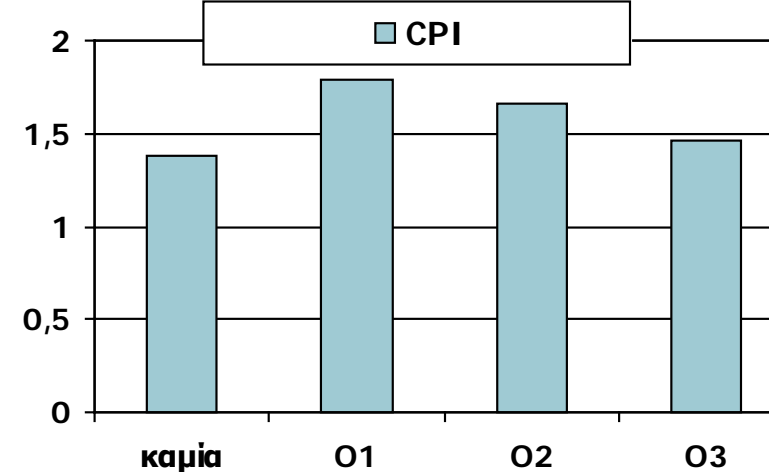
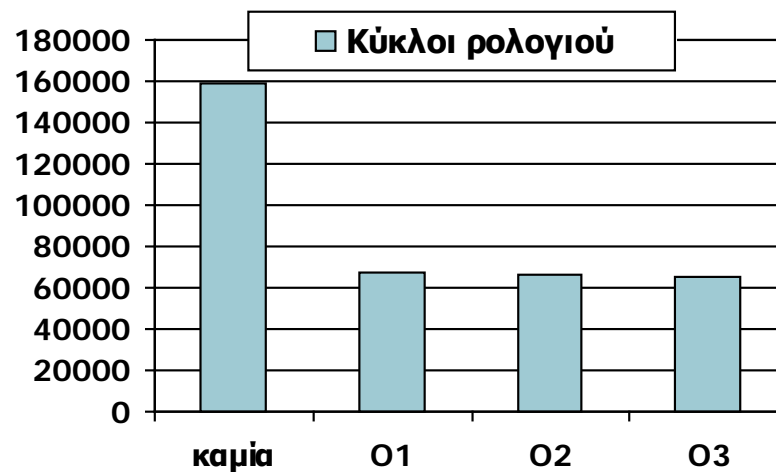
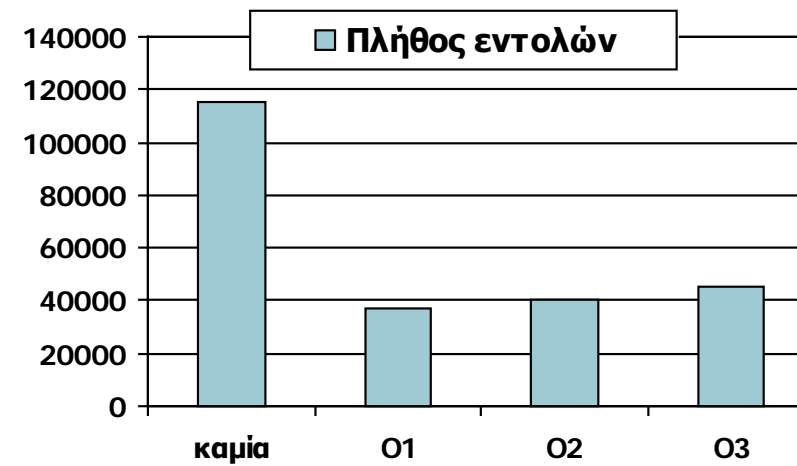
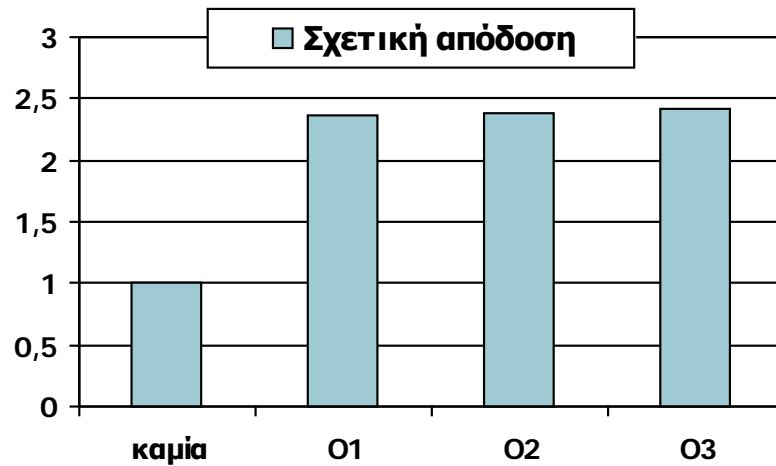
	move \$s2, \$a0	# save \$a0 into \$s2	Μεταφορά παραμέτρων
	move \$s3, \$a1	# save \$a1 into \$s3	
	move \$s0, \$zero	# i = 0	Εξωτερικός βρόχος
for1tst:	slt \$t0, \$s0, \$s3	# \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)	
	beq \$t0, \$zero, exit1	# go to exit1 if \$s0 ≥ \$s3 (i ≥ n)	
	addi \$s1, \$s0, -1	# j = i - 1	
for2tst:	slti \$t0, \$s1, 0	# \$t0 = 1 if \$s1 < 0 (j < 0)	Εσωτερικός βρόχος
	bne \$t0, \$zero, exit2	# go to exit2 if \$s1 < 0 (j < 0)	
	sll \$t1, \$s1, 2	# \$t1 = j * 4	
	add \$t2, \$s2, \$t1	# \$t2 = v + (j * 4)	
	lw \$t3, 0(\$t2)	# \$t3 = v[j]	
	lw \$t4, 4(\$t2)	# \$t4 = v[j + 1]	
	slt \$t0, \$t4, \$t3	# \$t0 = 0 if \$t4 ≥ \$t3	
	beq \$t0, \$zero, exit2	# go to exit2 if \$t4 ≥ \$t3	
	move \$a0, \$s2	# 1st param of swap is v (old \$a0)	Μεταβίβαση παραμέτρων και κλήση
	move \$a1, \$s1	# 2nd param of swap is j	
	jal swap	# call swap procedure	
	addi \$s1, \$s1, -1	# j -= 1	Εσωτερικός βρόχος
	j for2tst	# jump to test of inner loop	
exit2:	addi \$s0, \$s0, 1	# i += 1	Εξωτερικός βρόχος
	j for1tst	# jump to test of outer loop	

Η πλήρης διαδικασία

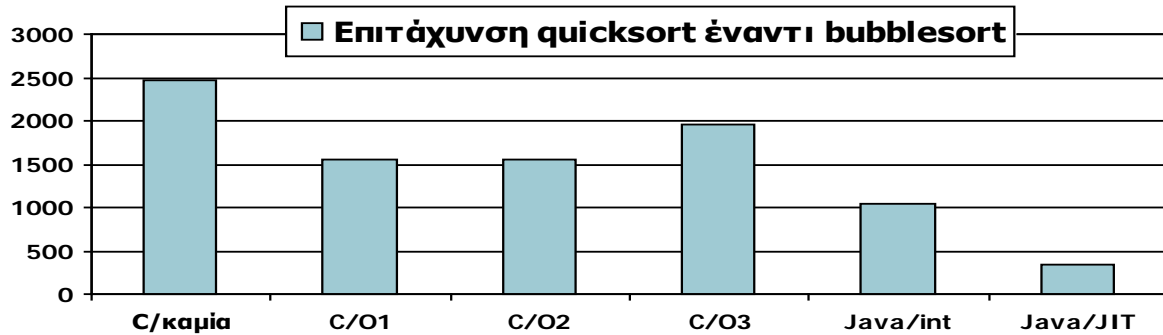
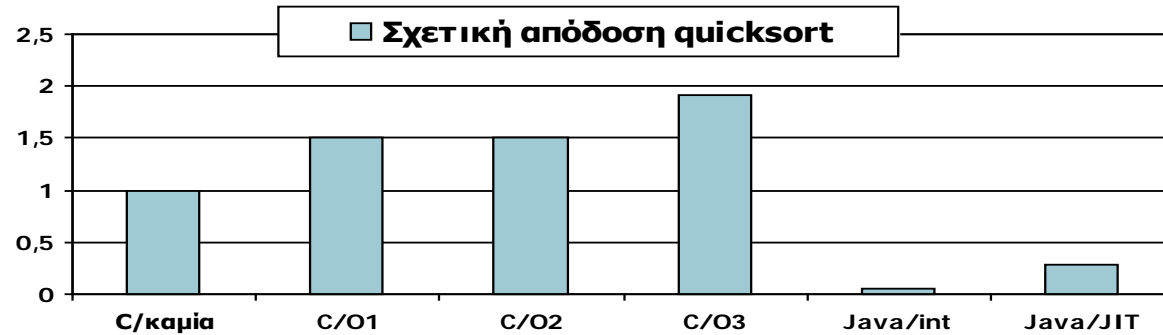
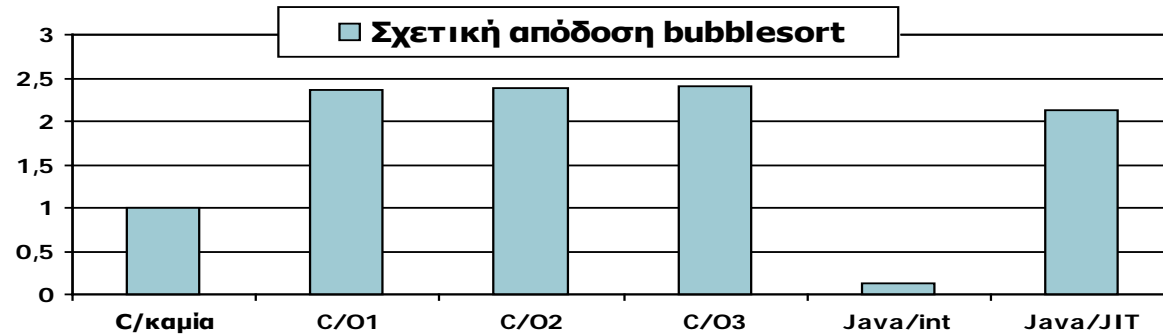
sort:	addi \$sp,\$sp, -20	# make room on stack for 5 registers
	sw \$ra, 16(\$sp)	# save \$ra on stack
	sw \$s3,12(\$sp)	# save \$s3 on stack
	sw \$s2, 8(\$sp)	# save \$s2 on stack
	sw \$s1, 4(\$sp)	# save \$s1 on stack
	sw \$s0, 0(\$sp)	# save \$s0 on stack
	...	# procedure body
	...	
exit1:	lw \$s0, 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
	jr \$ra	# return to calling routine

Επίδραση βελτιστοποιήσεων μεταγλωττιστή

Μεταγλώττιση με τον gcc σε Pentium 4 με Linux



Επίδραση της γλώσσας και του αλγορίθμου



Τι μάθαμε

- Το πλήθος εντολών και το CPI δεν είναι καλές ενδείξεις απόδοσης από μόνες τους
- Οι βελτιστοποιήσεις μεταγλώττιστή είναι ευαίσθητες στον αλγόριθμο
- Ο κώδικας Java/JIT είναι σημαντικά ταχύτερος από τη διερμηνεία της JVM
 - Συγκρίσιμος με τον βελτιστοποιημένο κώδικα C σε κάποιες περιπτώσεις
- Τίποτε δεν μπορεί να διορθώσει έναν ανόητο αλγόριθμο!

Πίνακες και δείκτες

- Η δεικτοδότηση πινάκων (array indexing) περιλαμβάνει
 - Πολλαπλασιασμό του αριθμοδείκτη με το μέγεθος του στοιχείου
 - Πρόσθεση στη διεύθυνση βάσης του πίνακα
- Οι δείκτες (pointers) αντιστοιχούν απευθείας σε διευθύνσεις μνήμης
 - Μπορούν να μας γλιτώσουν από τις δυσκολίες της δεικτοδότησης

Παράδειγμα: μηδενισμός πίνακα

```
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 =  
                           # &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 (...)  
                           # goto loop1
```

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

```
        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 (...)  
                           # goto loop2
```

Πλάνες

- Ισχυρή εντολή \Rightarrow υψηλότερη απόδοση
 - λιγότερες εντολές
 - Αλλά, οι σύνθετες εντολές είναι δύσκολο να υλοποιηθούν
 - Μπορεί να καθυστερήσουν όλες τις εντολές, ακόμη και τις πιο απλές
 - Οι μεταγλωττιστές είναι καλοί στο να παράγουν γρήγορο κώδικα με απλές εντολές
- Χρήση κώδικα συμβολικής γλώσσας για υψηλή απόδοση
 - Αλλά, οι μεταγλωττιστές είναι καλύτεροι στο χειρισμό των σύγχρονων επεξεργαστών
 - Περισσότερες γραμμές κώδικα \Rightarrow περισσότερα σφάλματα και μικρότερη παραγωγικότητα

Παγίδες

- Οι διαδοχικές λέξεις δεν βρίσκονται σε διαδοχικές διευθύνσεις
 - Αύξηση κατά 4, όχι κατά 1!
- Διατήρηση ενός δείκτη (pointer) προς μια αυτόματη μεταβλητή μετά την επιστροφή της διαδικασίας
 - π.χ., μεταβίβαση του δείκτη μέσω ενός ορίσματος
 - Ο δείκτης γίνεται άκυρος μετά το «άδειασμα» της στοίβας για τη διαδικασία

Συμπερασματικές παρατηρήσεις

- Σχεδιαστικές αρχές
 1. Η απλότητα ευνοεί την κανονικότητα
 2. Το μικρότερο είναι ταχύτερο
 3. Κάνε τη συνηθισμένη περίπτωση γρήγορη
 4. Η καλή σχεδίαση απαιτεί καλούς συμβιβασμούς
- Επίπεδα λογισμικού/υλικού
 - Μεταγλωττιστής, συμβολομεταφραστής, υλικό
- MIPS: τυπική αρχιτεκτονική συνόλου εντολών RISC
 - σύγκριση με x86

Συμπερασματικές παρατηρήσεις

- Μέτρηση εκτελέσεων εντολών MIPS σε μετροπρογράμματα
 - Κάνε τη συνηθισμένη περίπτωση γρήγορη
 - Κάνε συμβιβασμούς

Κατηγορία εντολής	Παραδείγματα MIPS	SPEC2006 Int	SPEC2006 FP
Αριθμητικές	add, sub, addi	16%	48%
Μεταφοράς δεδομένων	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Λογικές	and, or, nor, andi, ori, sll, srl	12%	4%
Διακλάδωσης υπό συνθήκη	beq, bne, slt, slti, sltiu	34%	8%
Άλματος	j, jr, jal	2%	0%