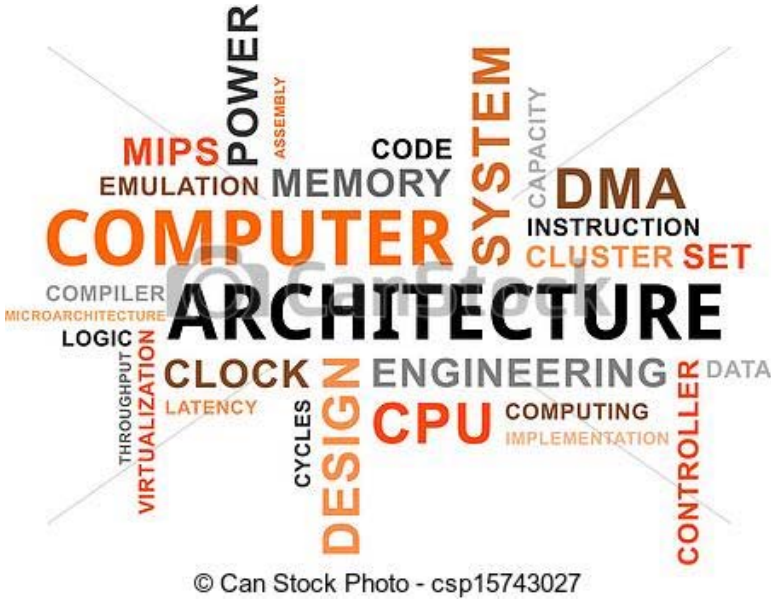
SCHOOL OF INFORMATICS



Αρχιτεκτονική Υπολογιστών

Διάλεξη 3 – Αρχιτεκτονική Συνόλου Εντολών (ISA)

Γεώργιος Κεραμίδας, Επίκουρος Καθηγητής

3ο Εξάμηνο, Τμήμα Πληροφορικής

ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

Αντιστοίχιση με ύλη Βιβλίου

- **Το συγκεκριμένο σετ διαφανειών καλύπτει τα εξής κεφάλαια/ενότητες:**
  - **Κεφάλαιο 2: 2.1 έως 2.14**

Εντολές Assembly

κώδικας_mnemonic [operands] // operand == τελεστής

Γενική μορφή εντολών

κώδικας_mnemonic

κώδικας_mnemonic [destination]

κώδικας_mnemonic [destination], [source]

κώδικας_mnemonic [destination], [source1], [source2]

Είδη Operands

Immediate, αριθμός (δεδομένο ή offset για θέση μνήμης)

Registers, αριθμός καταχωρητή

Memory, θέση μνήμης

Γενική Μορφή Εντολών

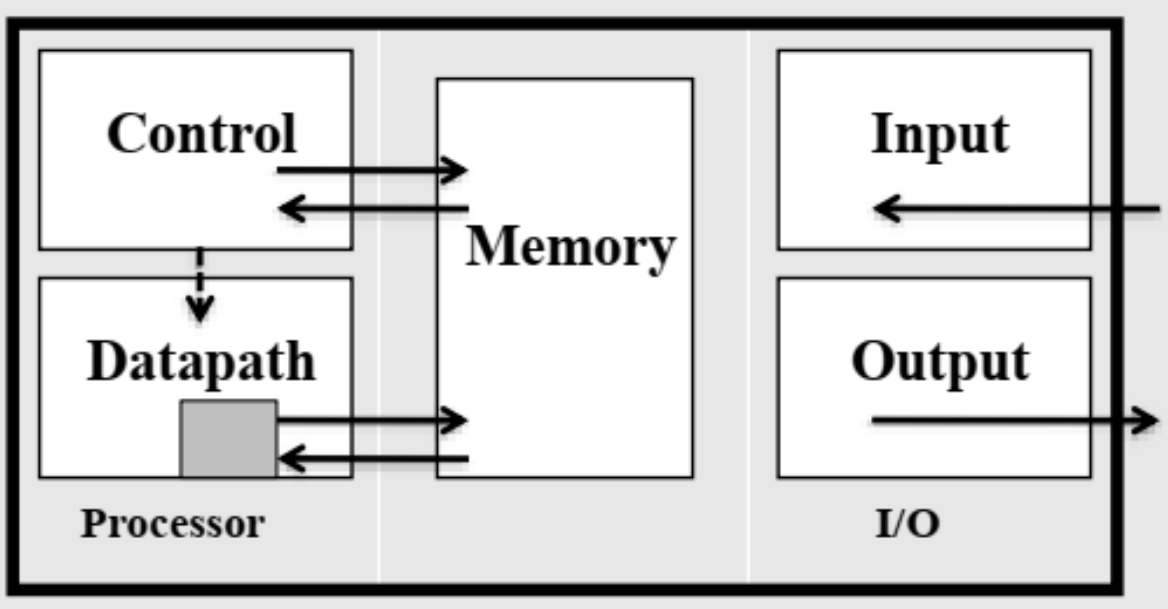
mov R1, 1000h // R1 ← 1000h

mov R1, R2 // R1 ← R2

Αν ο R2 = 5 → R1 = R2 = 5

mov R1, [1000h] ; R1 ← το περιεχόμενο της θέσης μνήμης 1000h

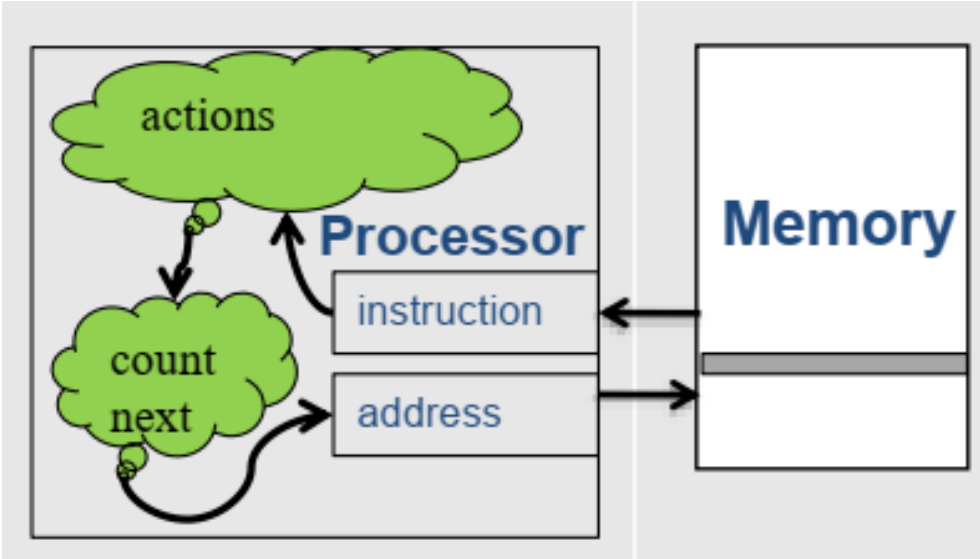
Καταχωρητές και Μνήμη



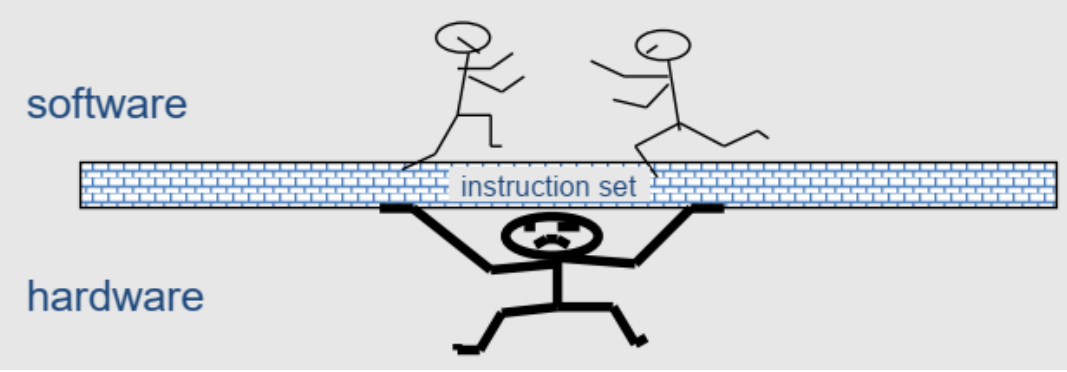
- Οι τελεστές των αριθμητικών εντολών πρέπει να είναι σε καταχωρητές
- Η μνήμη είναι μεγάλη αλλά αργή, οι καταχωρητές είναι γρήγοροι αλλά είναι πολύ λίγοι
- Ο μεταγλωττιστής συνδέει μεταβλητές με καταχωρητές (πχ. Η μεταβλητή count συνδέεται με τον καταχωρητή R1)
- Αν ένα πρόγραμμα έχει πολλές μεταβλητές τότε θα πρέπει να γίνουν πολλές μεταφορές ανάμεσα στους καταχωρητές και στην μνήμη

Η έννοια του αποθηκευμένου προγράμματος

- Οι εντολές είναι bits
- Τα προγράμματα αποθηκεύονται στην μνήμη
  - Ανάκληση (fetch) εντολής → ανάκληση (φόρτωση) εντολών από την μνήμη
  - Συγκεκριμένα bits της εντολής καθορίζουν ποια λειτουργία θα εκτελεστεί και που είναι τα δεδομένα
  - Εκτέλεση εντολής (+++)
  - Ανάκληση (fetch) επόμενης εντολής



Αρχιτεκτονική Συνόλου Εντολών



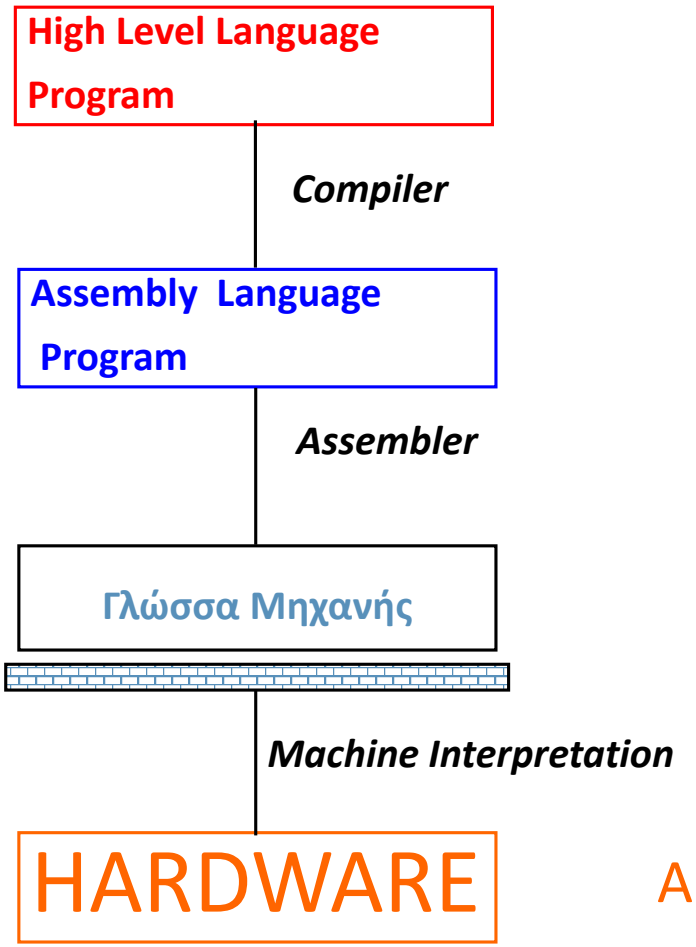
- Διεπαφή (interface) ανάμεσα στο υλικό (hardware) και το λογισμικό (low-level software)
- Instruction Set Architecture (ISA)
- Θα ασχοληθούμε με το ISA του επεξεργαστή MIPS
- ISA
  - → Όλες οι στοιχειώδης λειτουργίες που υποστηρίζει το υλικό.
  - → Όλα τα προγράμματα λογισμικού (κώδικες) μεταφράζονται σε αυτές τις στοιχειώδης λειτουργίες

- Το ISA μπορεί να είναι απλό (MIPS επεξεργαστής, εκπρόσωπος των αρχιτεκονικών τύπου RISC) ή πολύπλοκο (x86, x64/INTEL, εκπρόσωπος των αρχιτεκονικών τύπου CISC)
  - RISC: Reduced Instruction Set Computer
  - CISC: Complex Instruction Set Computer
- Παράδειγμα: Τι είναι καλύτερο και με βάση ποια μετρική?

MOVE Θέση_Μνήμης_1, Θέση_Μνήμης_2

MOVE R1, Θέση_Μνήμης_1
MOVE Θέση_Μνήμης_2, R1

Μορφή προγράμματος σε κάθε επίπεδο




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

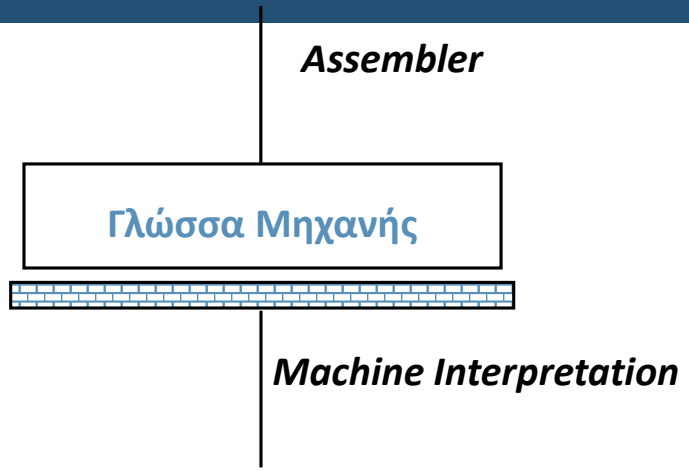
```
lw   $15, 0($2)
lw   $16, 4($2)
sw   $16, 0($2)
sw   $15, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:12] (παράδειγμα)

Αρχιτεκτονικές Συνόλου Εντολών

- **Ανεξάρτητα από τη δομή της CPU κάθε εντολή Assembly πρέπει να προσδιορίζει τα ακόλουθα:**
  - **Opcode: Ποια εντολή εκτελείται. Παράδειγμα: add, load και branch.**



0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

- **Πού βρίσκονται οι τελεστές: Οι τελεστές μπορεί να είναι αποθηκευμένοι σε καταχωρητές της CPU, στην κύρια μνήμη, ή σε θύρες εισόδου/εξόδου.**
- **Πού τοποθετείται το αποτέλεσμα: Μπορεί να αναφέρεται ρητά ή να υπονοείται από τον κωδικό της εντολής (opcode).**
- **Πού βρίσκεται η επόμενη εντολή: Αν δεν υπάρχουν ρητές διακλαδώσεις (branches), η προς εκτέλεση εντολή είναι η επόμενη. Σε περίπτωση εντολών jump ή branch η διεύθυνση προσδιορίζεται από αυτές.**

Αρχιτεκτονικές Συνόλου Εντολών

Γενικές Κατηγορίες

- Arithmetic and logical → Integer arithmetic & logical operations: add etc
- Data transfer → Loads-stores (from/to memory)
- Control → Branch, jump, procedure call, return, traps
- System → Operating system call, virtual memory management instructions
- Floating point → Floating point operations: add, multiply.
- Decimal → Decimal add, decimal multiply, decimal to character conversion
- String → String move, string compare, string search
- Graphics → Pixel operations, compression/ decompression operations

Παραδείγματα Εντολών μετακίνησης δεδομένων

Instruction	Meaning	Machine
MOV A,B	Move 16-bit data from memory loc. A to loc. B	VAX11
lwz R3,A	Move 32-bit data from memory loc. A to register R3	PPC601
li $3,455	Load the 32-bit integer 455 into register $3	MIPS R3000
MOV AX,BX	Move 16-bit data from register BX into register AX	Intel X86
LEA.L (A0),A2	Load the address pointed to by A0 into A2	MC68000

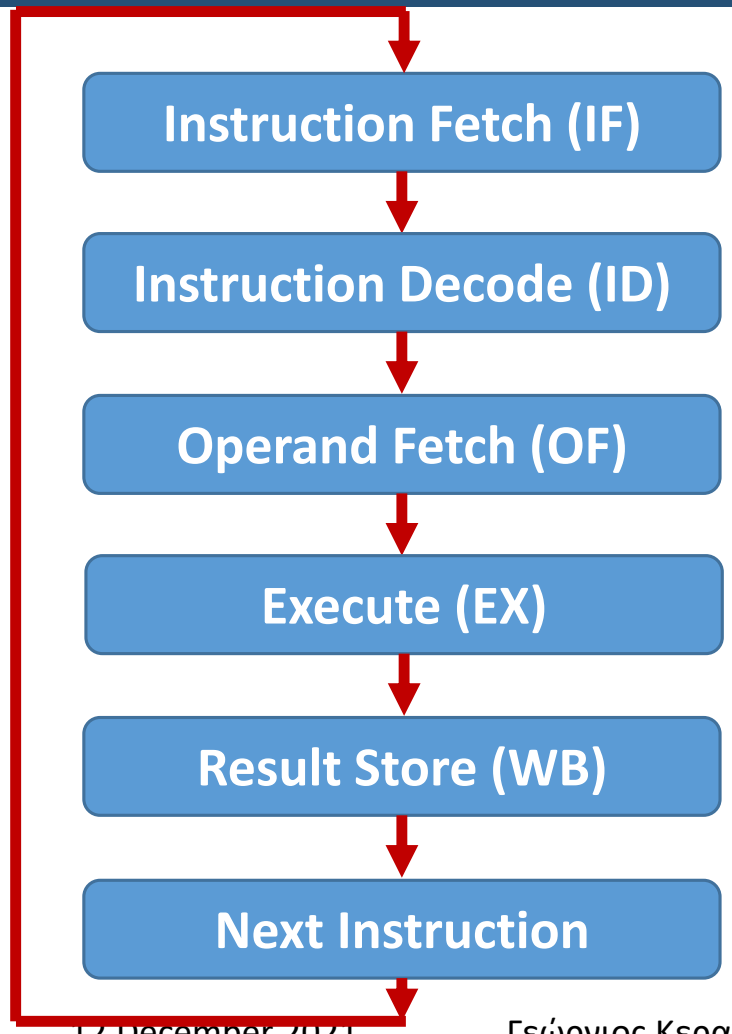
Παραδείγματα Εντολών της ALU

Instruction	Meaning	Machine
MULF A,B,C	Multiply the 32-bit floating point values at mem. locations A and B, and store result in loc. C	VAX11
nabs r3,r1	Store the negative absolute value of register r1 in r2	PPC601
ori $2,$1,255	Store the logical OR of register $1 with 255 into $2	MIPS R3000
SHL AX,4	Shift the 16-bit value in register AX left by 4 bits	Intel X86
ADD.L D0,D1	Add the 32-bit values in registers D0, D1 and store the result in register D0	MC68000

Παραδείγματα Εντολών Διακλάδωσης

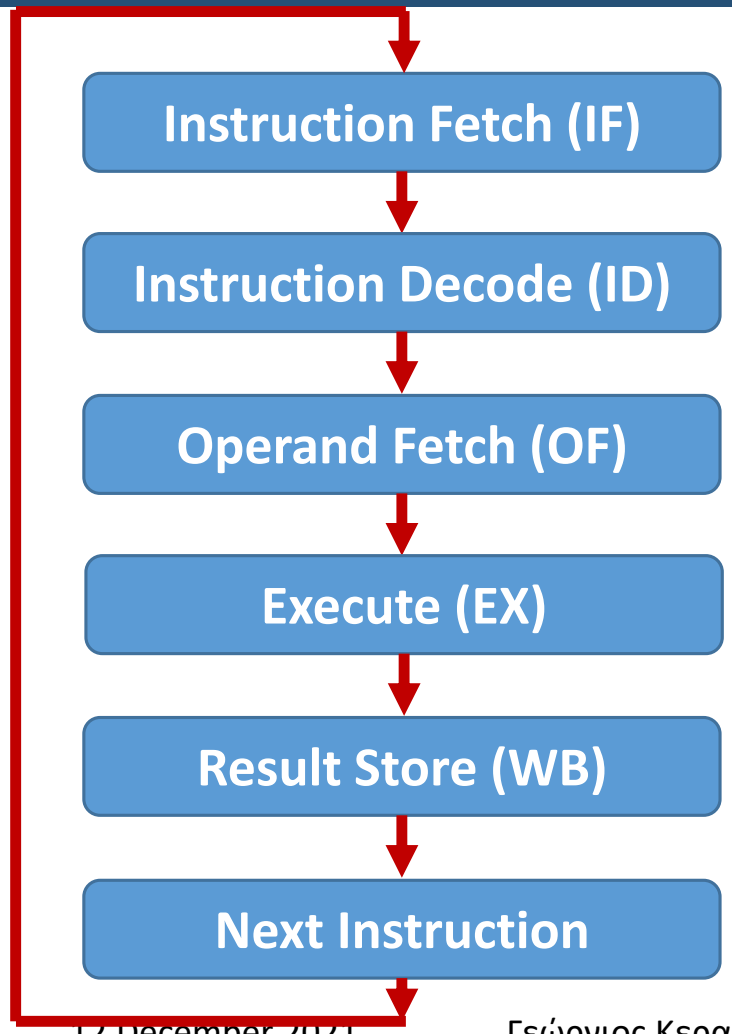
Instruction	Meaning	Machine
BLBS A, Tgt	Branch to address Tgt if the least significant bit at location A is set	VAX11
bun r2	Branch to location in r2 if the previous comparison signaled that one or more values was not a number	PPC601
Beq $2,$1,32	Branch to location PC+4+32 if contents of $1 and $2 are equal	MIPS R3000
JCXZ Addr	Jump to Addr if contents of register CX = 0.	Intel X86
BVS next	Branch to next if overflow flag in CC is set.	MC68000

Αρχιτεκτονικές Συνόλου Εντολών



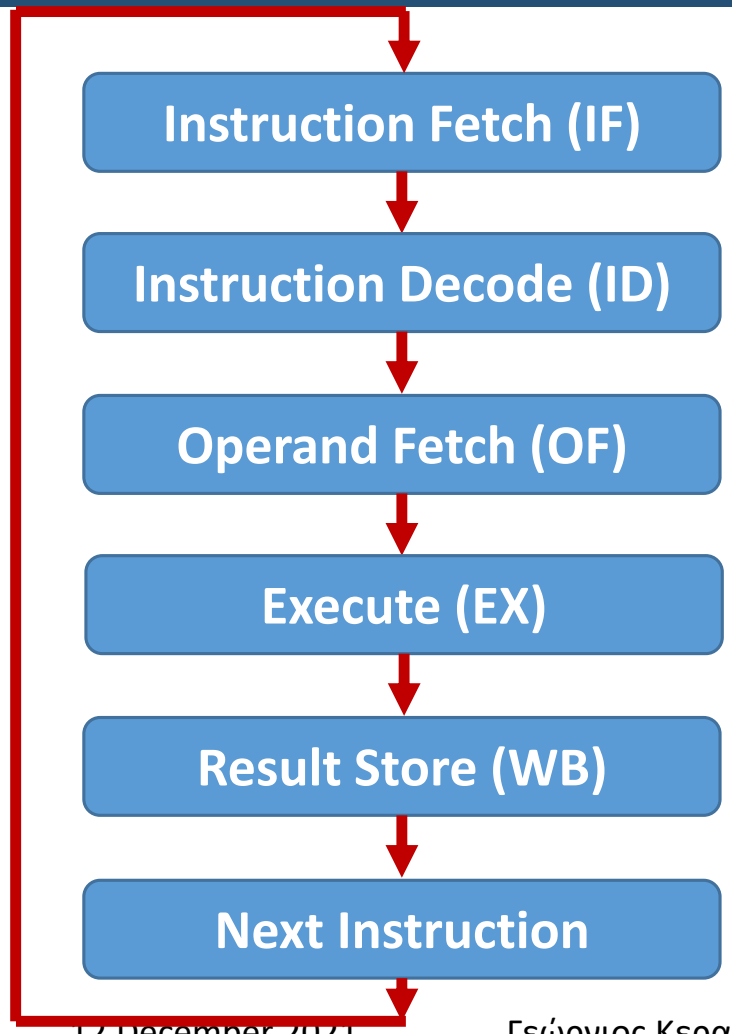
- Πάρε την εντολή από τη θέση αποθήκευσης του προγράμματος
- Καθόρισε τις απαιτούμενες ενέργειες και το μέγεθος της εντολής
- Εντόπισε και πάρε τα δεδομένα-τελεστές
- Υπολόγισε την τιμή του αποτελέσματος ή της κατάστασης
- Αποθήκευσε τα αποτελέσματα για μεταγενέστερη χρήση
- Καθόρισε την επόμενη εντολή

Αρχιτεκτονικές Συνόλου Εντολών



- MIPS – κάθε εντολή 1 κυκλο
- Intel – κάθε εντολή 1 – 10 κυκλους
- 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;

Αρχιτεκτονικές Συνόλου Εντολών



- **Μορφοποίηση ή Κωδικοποίηση Εντολών:**
  - **Πώς κωδικοποιείται;**
- **Θέση τελεστών και αποτελέσματος (addressing modes):**
  - **Πού αλλού εκτός μνήμης;**
  - **Πόσοι ρητοί τελεστές;**
  - **Πώς αντιστοιχίζονται (located) οι τελεστές μνήμης;**
  - **Ποιοι μπορούν να βρίσκονται στη μνήμη και ποιοι όχι;**
- **Τύποι και μέγεθος δεδομένων**
- **Πράξεις**
  - **Ποιες υποστηρίζονται**
- **Διαδοχή εντολών:**
  - **Jumps, conditions, branches**

Κατηγορίες Αρχιτεκτονικών Συνόλου Εντολών

- **Αρχιτεκτονικές Συσσωρευτή (accumulator architectures)**
- **Αρχιτεκτονικές εκταμένου συσσωρευτή ή καταχωρητών ειδικού σκοπού (extended accumulator ή special purpose register)**

- **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 (Accumulator)**
- **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

CISC: Complex Instruction Set Computer

- **Εντολές για πράξεις Register-Memory ή Memory-Memory**
- **Αφήνουν το ένα όρισμα να είναι στη μνήμη (πχ. PENTIUM)**

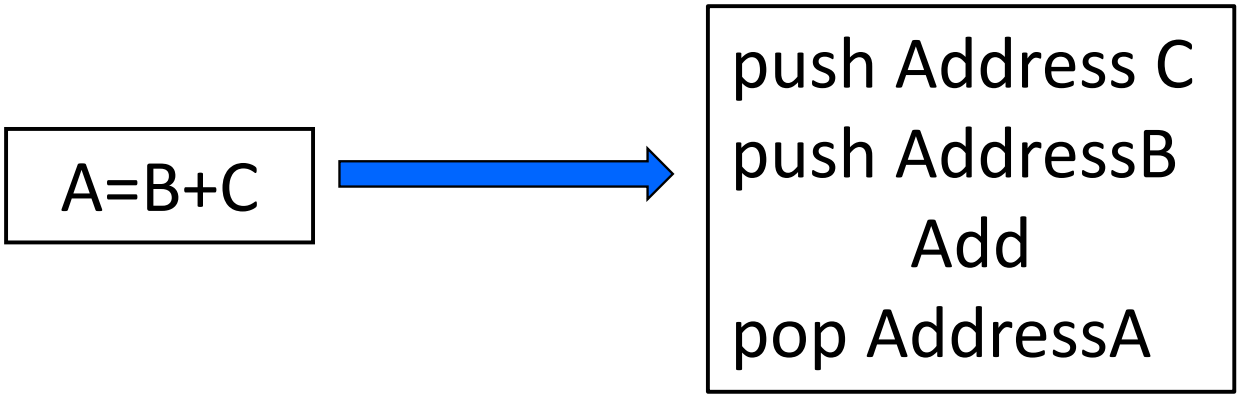
RISC: Reduced Instruction Set Computer

- **Πράξεις μόνο Register-Register (load store) (πχ. ARM)**

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**



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

Βασικές Αρχές Σχεδίασης

- **Απλές εντολές → μικρότερο hardware → ταχύτερο hardware (υψηλότερο ρολόι) (smaller is faster)**

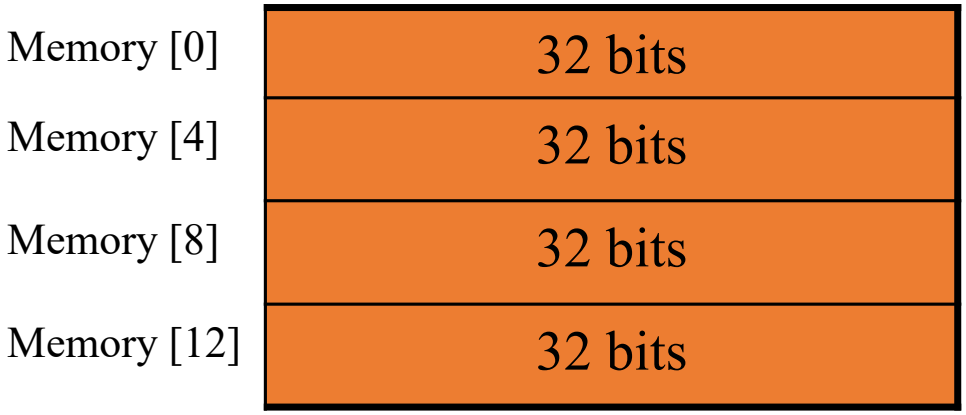
- **Ομοιόμορφες εντολές → απλούστερο hardware → ταχύτερο hardware (simpler is faster)**

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)**

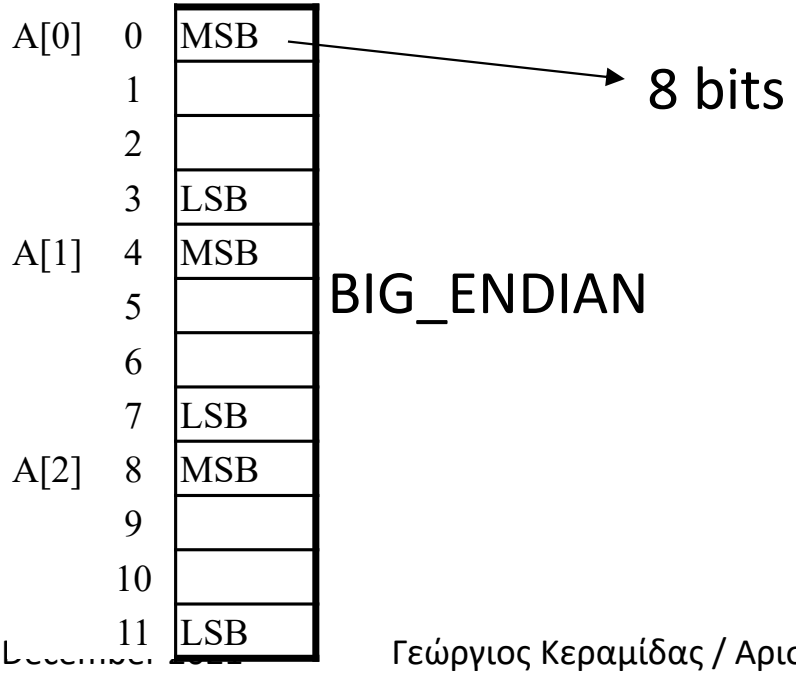
Σύνολο Εντολών MIPS

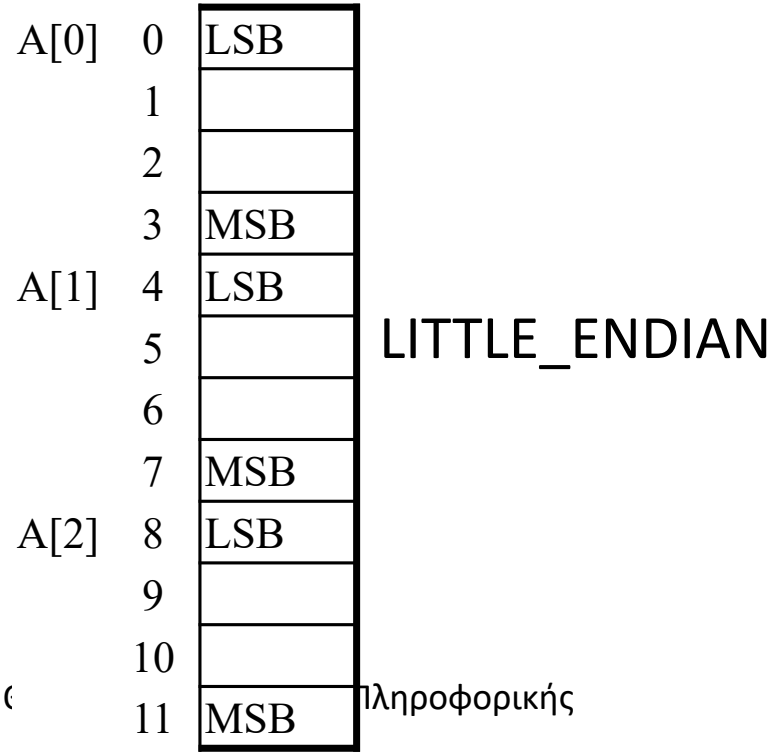
- Λέξεις των 32 bit
- Μνήμη οργανωμένη σε bytes
  - Κάθε byte είναι μια ξεχωριστή δνση
  - 2^{30} λέξεις μνήμης των 32 bits
  - Ακολουθεί το μοντέλο big Endian
- Register File
  - 32 καταχωρητές γενικού σκοπού
- Εντολές :
  - αποθήκευσης στη μνήμη (lw, sw)
  - αριθμητικές (add, sub κλπ)
  - διακλάδωσης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



Big Endian vs Little Endian

- **Big Endian:** Η δνση του πιο σημαντικού byte (MSB) είναι και δνση της λέξης
- **Little Endian:** Η δνση του λιγότερο σημαντικού byte (LSB) είναι και δνση της λέξης
- Η λέξη αποθηκεύεται πάντα σε συνεχόμενες θέσεις: δνση, δνση+1, δνση+2,δνση+3





Endianness

- **Πως τα bytes αποθηκεύονται στην μνήμη**
- **X = 12345678h**

Little Endian

Offset	Value
0000	78
0001	56
0002	34
0003	12

Big Endian

Offset	Value
0000	12
0001	34
0002	56
0003	78

MIPS ISA : Αριθμητικές λειτουργίες

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

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

- **Όλες οι αριθμητικές λειτουργίες έχουν αυτή τη μορφή**

- **Η κανονικότητα επιτρέπει μεγαλύτερη απόδοση με χαμηλότερο κόστος (simpler is faster)**

Τελεστές - Καταχωρητές

- Οι αριθμητικές εντολές χρησιμοποιούν καταχωρητές ως τελεστές (operands)
- Ο MIPS διαθέτει ένα αρχείο καταχωρητών (register file) με 32 καταχωρητές των 32-bit
  - Χρήση για τα δεδομένα που προσπελάζονται συχνά
  - Αρίθμηση καταχωρητών από 0 έως 31
- Ονόματα του συμβολομεταφραστή (assembler)
  - $t0, $t1, …, $t9 για προσωρινές τιμές
  - $s0, $s1, …, $s7 για αποθηκευμένες μεταβλητές

Παράδειγμα

Κώδικας σε C

a = b + c;

d = a – e;

Μετάφραση σε κώδικα MIPS -- υπάρχουν λάθη

add a, b, c

sub d, a, e

Παράδειγμα

Κώδικας σε C

f = (g + h) – (i + j);

Τι παράγει ο compiler?

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)

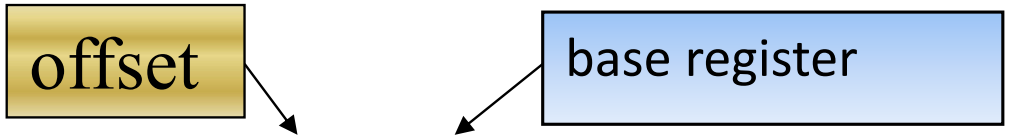
- **φορτώνουμε στον $t1 την τιμή M[$s2+4]**

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, ο A είναι τύπου int).**

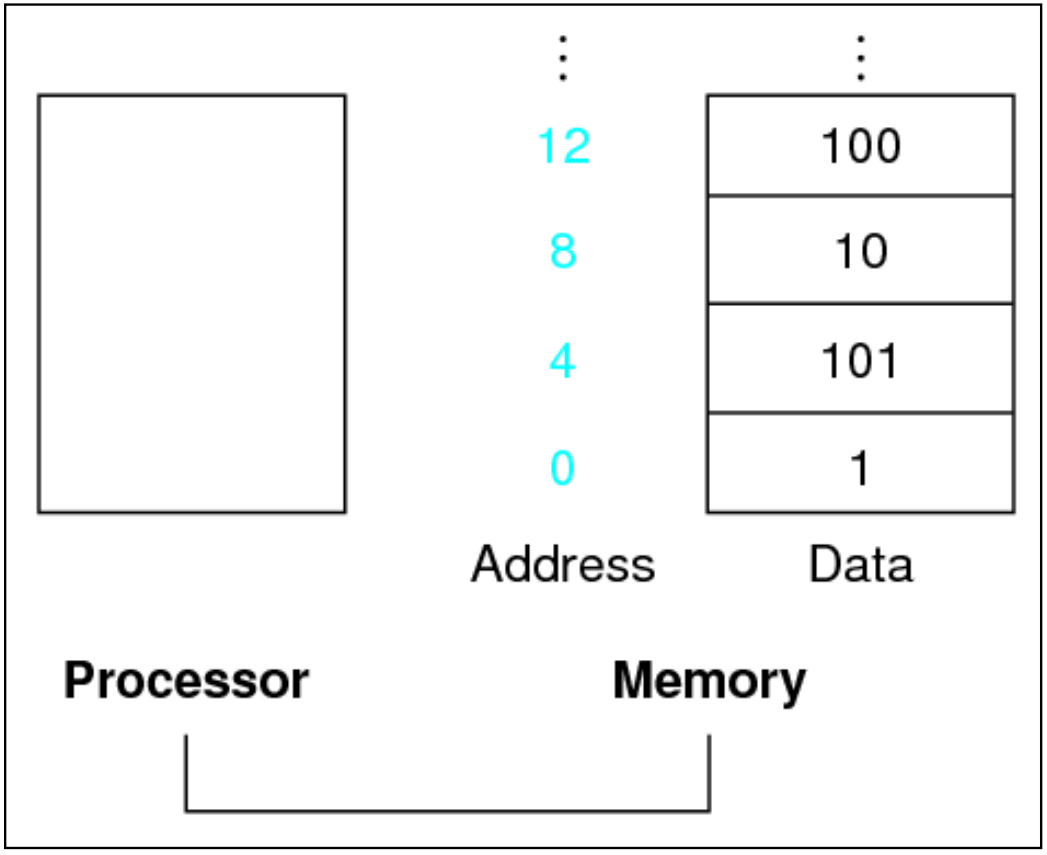


lw $t0, 32($s3)

add $s1, $s2, $t0

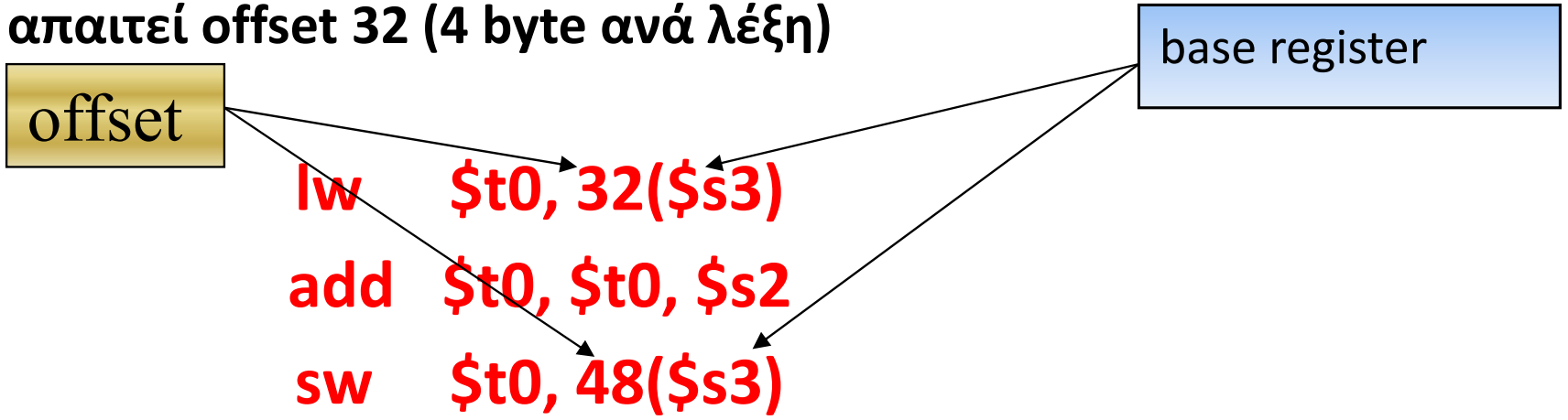
Οργάνωση Μνήμης

- **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

- Κάνε τη συνηθισμένη περίπτωση γρήγορη
  - Οι μικρές σταθερές είναι συνηθισμένες
  - Ο άμεσος τελεστέος αποφεύγει μια εντολή φόρτωσης (load)

Η σταθερά ΜΗΔΕΝ

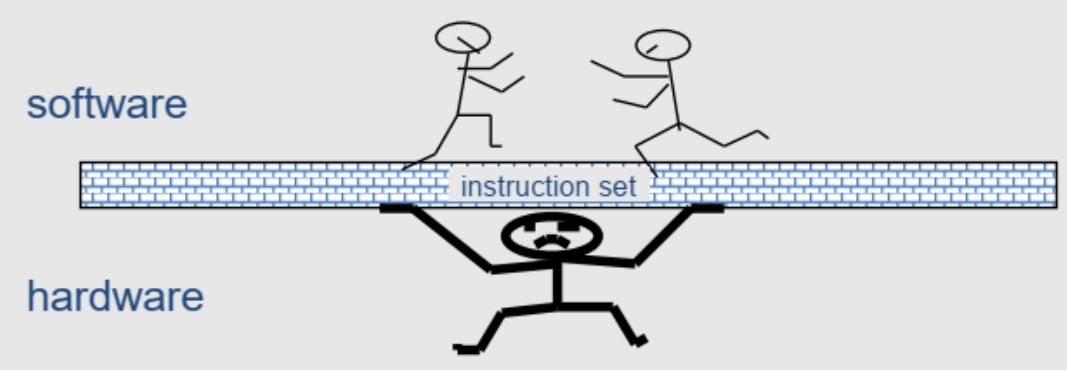
- **Make the common case fast, ο MIPS έχει στον καταχωρητή $zero αποθηκευμένη τη σταθερά 0.**
  - **Δεν μπορεί να εγγραφεί άλλη τιμή**

- **Χρήσιμη σε πολλές λειτουργίες**
  - **Μετακίνηση δεδομένων μεταξύ καταχωρητών π.χ. add $t1, $t2, $zero**

Σύνοψη – Τελεστέοι MIPS

- **Συνοπτικά, στον MIPS ο τελεστής κάποιας εντολής μπορεί να είναι :**
  - **Ένας από τους 32 καταχωρητές**
  - **Μία από τις 2^{30} λέξεις της μνήμης**
  - **Ένα από τα 2^{32} bytes της μνήμης**

Αρχιτεκτονική Συνόλου Εντολών



- Διεπαφή (interface) ανάμεσα στο υλικό (hardware) και το λογισμικό (low-level software)
- Instruction Set Architecture (ISA)
- Θα ασχοληθούμε με το ISA του επεξεργαστή MIPS
- ISA
  - → Όλες οι στοιχειώδης λειτουργίες που υποστηρίζει το υλικό.
  - → Όλα τα προγράμματα λογισμικού (κώδικες) μεταφράζονται σε αυτές τις στοιχειώδης λειτουργίες

- Το ISA μπορεί να είναι απλό (MIPS επεξεργαστής, εκπρόσωπος των αρχιτεκονικών τύπου RISC) ή πολύπλοκο (x86, x64/INTEL, εκπρόσωπος των αρχιτεκονικών τύπου CISC)
  - RISC: Reduced Instruction Set Computer
  - CISC: Complex Instruction Set Computer
- Παράδειγμα: Τι είναι καλύτερο και με βάση ποια μετρική?

MOVE Θέση_Μνήμης_1, Θέση_Μνήμης_2

MOVE R1, Θέση_Μνήμης_1
MOVE Θέση_Μνήμης_2, R1

Βασικές Αρχές Σχεδίασης

- **Απλές εντολές → μικρότερο hardware → ταχύτερο hardware (υψηλότερο ρολόι) (smaller is faster)**
- **Ομοιόμορφες εντολές → απλούστερο hardware → ταχύτερο hardware (simpler is faster)**
- **Προσπάθησε να βελτιστοποιήσεις την συνηθισμένη περίπτωση (make the common case fast)**

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. …**

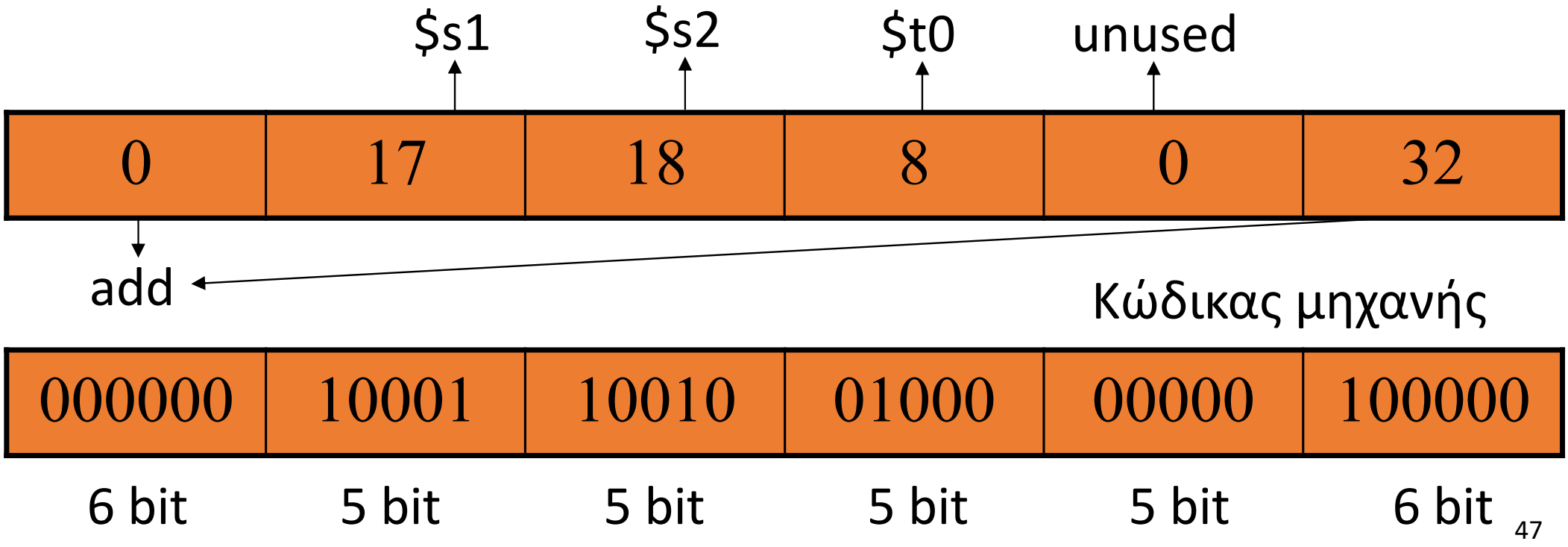
- **Αριθμοί καταχωρητών**
  - **$t0 – $t7 είναι οι καταχωρητές 8 – 15**
  - **$t8 – $t9 είναι οι καταχωρητές 24 – 25**
  - **$s0 – $s7 είναι οι καταχωρητές 16 – 23**

Αναπαράσταση Εντολών (2)

Συμβολική αναπαράσταση: add $t0, $s1, $s2

Assembly

Πώς την καταλαβαίνει ο MIPS?



Μορφή Εντολής – Instruction Format

R-Type (register type)

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

Op: opcode

rs,rt: register source operands

Rd: register destination operand

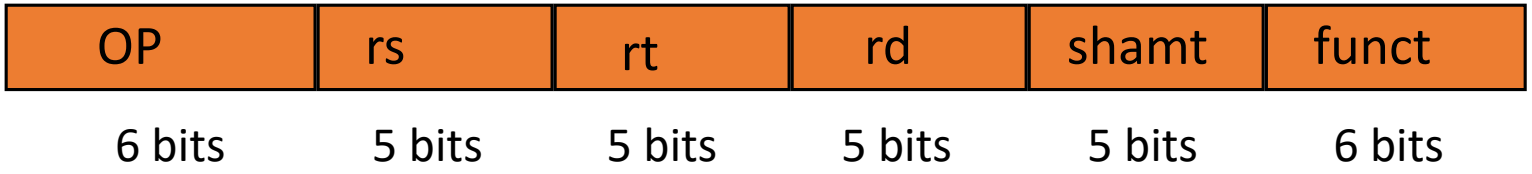
Shamt: shift amount

Funct: op specific (function code)

add $rd, $rs, $rt

MIPS R-Type (ALU)

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



- **Παραδείγματα :**

add $1,$2,$3 **and $1,$2,$3**

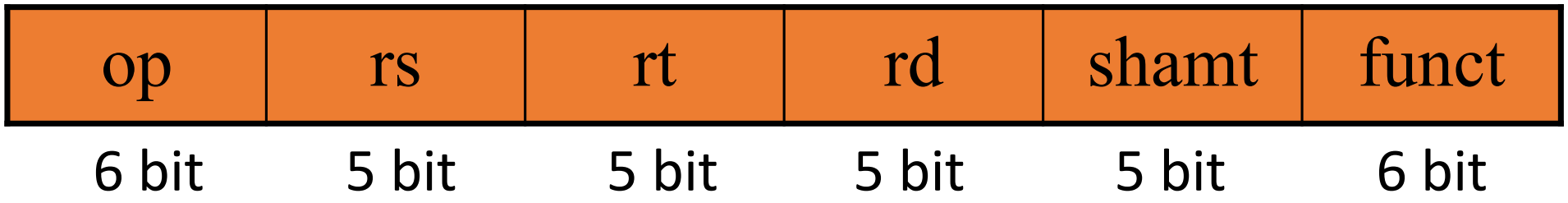
sub $1,$2,$3 **or $1,$2,$3**

Destination register in rd

Operand register in rs

Operand register in rt

Αναπαράσταση Εντολών στον Υπολογιστή (R-Type)



- **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)**

I-Type:

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

lw $rt, address_offset($rs)

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

Αναπαράσταση Εντολών στον Υπολογιστή (I-Type)

Παράδειγμα:

lw $t0, 32($s3)

Καταχωρητές
$s0, ..., $s7 αντιστοιχίζονται στους 16 - 23
$t0, ..., $t7 αντιστοιχίζονται στους 8 - 15

I-format

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

MIPS I-Type : Load/Store

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

- **Παραδείγματα :**
  - **Store word:** **sw $3, 500($4)**

source register in rt
Offset
base register in rs

  - **Load word:** **lw $1, 30($2)**

Destination register in rt
Offset
base register in rs

MIPS I-Type : ALU

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

6 bits 5 bits 5 bits 16 bits

- *immediate: Constant second operand for ALU instruction*

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

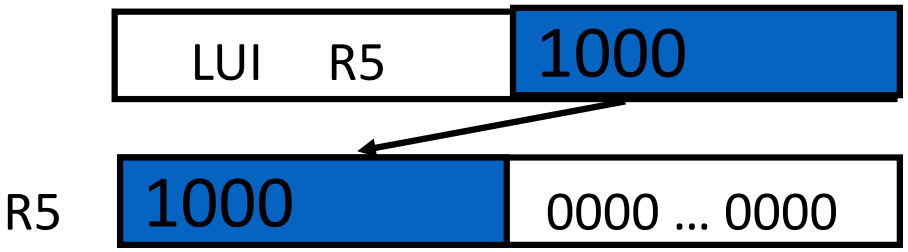
Result register in rt

Source operand register in rs

Constant operand in immediate

MIPS data transfer instructions : Παραδείγματα (1)

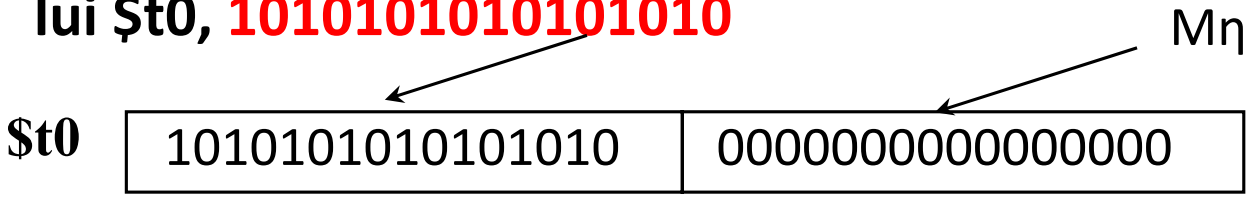
Instruction	Σχόλια
sw $3, 500($4)	Store word
sh $3, 502($2)	Store half
sb $2, 41($3)	Store byte
lw $1, 30($2)	Load word
lh $1, 40($3)	Load halfword
lhu $1, 40($3)	Load halfword unsigned
lb $1, 40($3)	Load byte
lbu $1, 40($3)	Load byte unsigned
lui $1, 40	Load Upper Immediate (16 bits shifted left by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



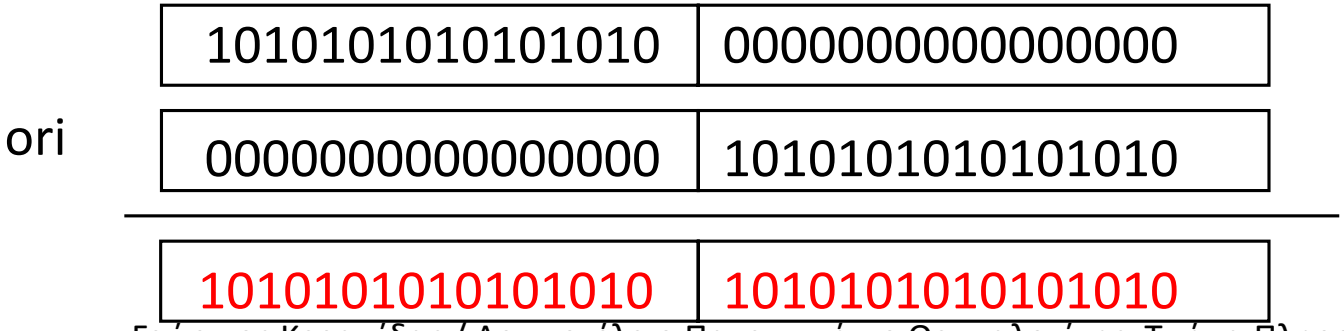
MIPS data transfer instructions : Παραδείγματα (2)

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
100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	0	100000
101011	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 (δεκαδικό)

sll $t2, $s0, 4

Κάνουμε shift αριστερά το περιεχόμενο του $s0 κατά 4 θέσεις

0000 0000 0000 0000 0000 0000 1001 0000 = 144 (δεκαδικό)

και τοποθετούμε το αποτέλεσμα στον $t2.

!!Το περιεχόμενο του $s0 μένει αμετάβλητο!!

Λογικές Λειτουργίες (Πράξεις) (3)

SHIFT

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

Καταχωρητές

$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 χρειάζεται γιατί μπορούμε να 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+100,** ***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.***

Είναι λίγο λάθος

MIPS Branch, Compare, Jump : Παραδείγματα

Instruction	*Παράδειγμα*	*Έννοια*
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 function/procedure call*

Εντολές διακλάδωσης – branching instruction

branch if equal

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

..... είναι I –Type εντολές

branch if !equal

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

unconditional Jump

jr $t1 # goto $t1

..... είναι R –Type εντολή

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

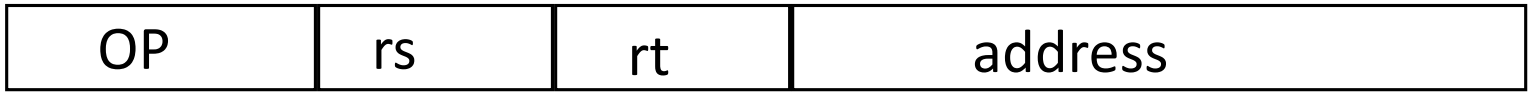
Εντολές διακλάδωσης – branching instruction

Όμως:

```
j L1 # goto L1
```

Πόσο μεγάλο είναι το μήκος του address L1;
Πόσο «μεγάλο» μπορεί να είναι το άλμα;

MIPS Branch I-Type



6 bits 5 bits 5 bits 16 bits

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

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

Παραδείγματα :

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

6 bits 26 bits

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

- **Παραδείγματα :**

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

  - **Jump** **j 10000**
  - **Jump and Link** **jal 10000**

Εντολές Jump (1)

- **Jump (J-type):**
  - **j 10000 # jump to address 10000**
- **Jump Register (R-type):**
  - **jr rs # jump to 32 bit address in register rs**

Εντολές Jump (2)

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

Σύνοψη – MIPS Instruction Formats

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

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

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

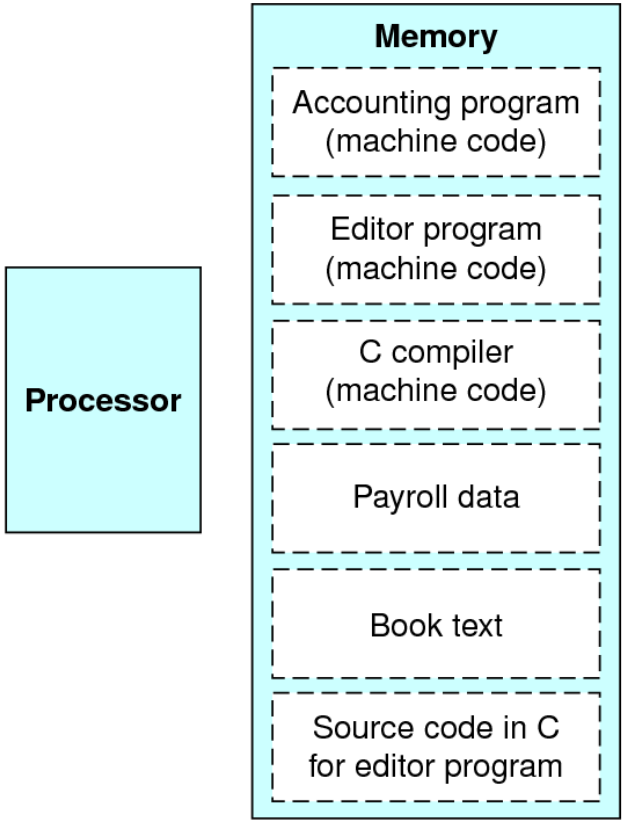
op	rs	rt	immediate value / address offset
6 bits	5 bits	5 bits	16 bits

J-type (j, jal)

op	jump target address
6 bits	26 bits

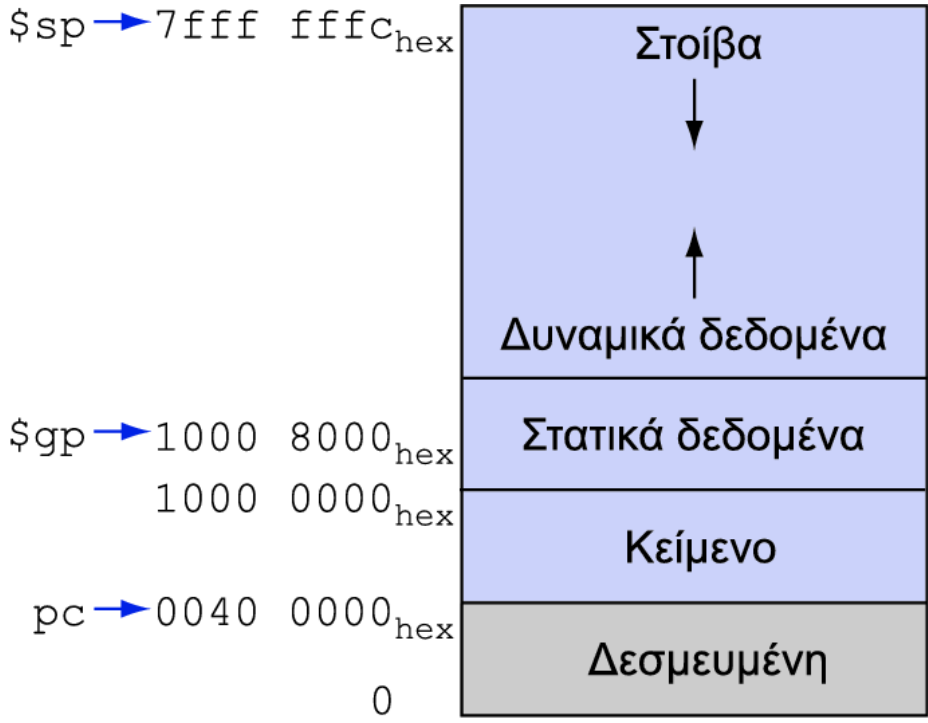
Έννοια αποθηκευμένου προγράμματος

- Ο υπολογιστής κάνει πολλές εργασίες φορτώνοντας δεδομένα στη μνήμη
- Δεδομένα και εντολές είναι στοιχεία στη μνήμη
- Π.χ. compilers μεταφράζουν στοιχεία σε κάποια άλλα στοιχεία
- Η μνήμη αποθηκεύει αριθμούς των x-bit (32, 64, κλπ)

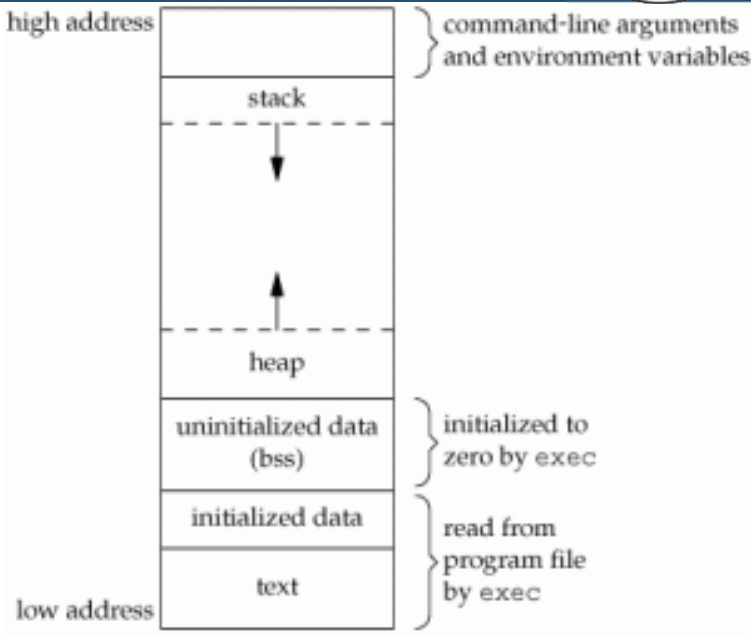


Διάταξη της Μνήμης ενός προγράμματος

- 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, τοπικές μεταβλητές, παράμετροι συνάρτησης



Διάταξη της Μνήμης ενός προγράμματος (2)



- **Text segment (κώδικας προγράμματος)**
- **Initialized data segment (or .data segment):**
  - **contains global variables, static variables,**
  - **divided into read only area + read-write area**

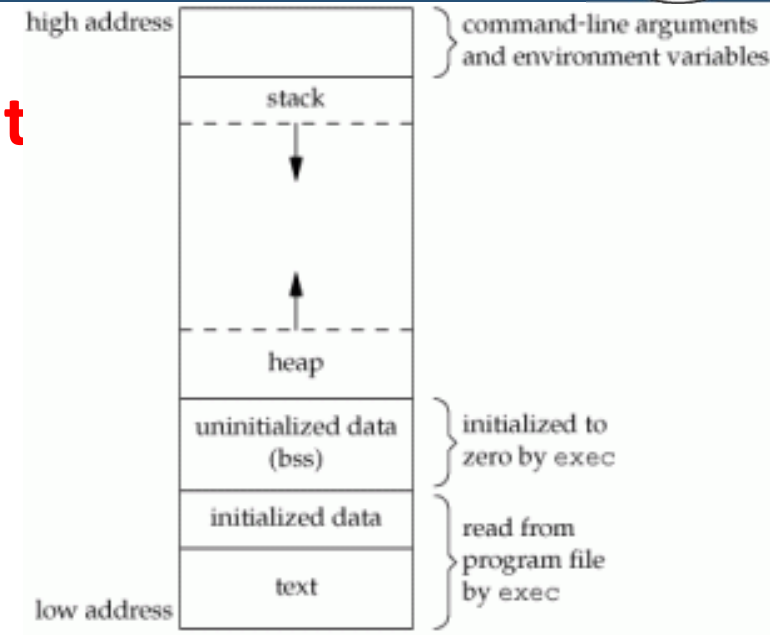
```
char s[]="hello world";
int debug = 1;
const char * string ="hello world";
```

"hello world" literal stored in ro area,
string stored in rw area

```
static int i = 10
global int i = 10
```

ro: read-only
rw: read-write

Διάταξη της Μνήμης ενός προγράμματος (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;**
- **Stack: stack pointer. Local variables from a function**
- **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 (en.wikipedia.org/wiki/ASCII)**
    - **C I S E**
  - **Διευθύνσεις μνήμης (pointers)**
    - **0x43495345**
  - **Εντολές (opcode = 010000, …)**

Τύποι Δεδομένων : MIPS (2)

- **16-bit σταθερές (immediates)**

  addi $s0, $s1, 0x8020

  lw $t0, 20($s0)

- **Half word (16 bits)**

  lh (lhu): load half word **lh $t0, 20($s0)**

  sh: save half word **sh $t0, 20($s0)**

- **Byte (8 bits)**

  lb (lbu): load byte **lb $t0, 20($s0)**

  sb: save byte **sb $t0, 20($s0)**

Εντολές λειτουργίας Byte

lb => Επέκταση προσήμου,

lbu => zero-fill

Παράδειγμα:
11111110 (8bits)=
1110 (4bits)=
-2 (δεκαδικό)

lb $s1, 4($s0)

$s0: 0x10000000

$s1: 0xFFFFFFAA

lbu $s1, 4($s0)

$s0: 0x10000000

$s1: 0x000000AA

Address

0x10000000

Memory Bytes

10101010

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

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

```
strcpy:
    subi $sp, $sp, 4
    sw   $s0, 0($sp)
    add  $s0, $zero, $zero # i=0
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 # i++
    j    L1
L2: lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Σταθερές

- **Συχνά χρησιμοποιούνται μικρές σταθερές (>50% των τελεστών)**
  - **π.χ. 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

6	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
    - reg2 = reg2 + reg1
- Τα opcodes μπορούν να χρησιμοποιηθούν με διαφορετικούς τρόπους διευθυνσιοδότησης

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**

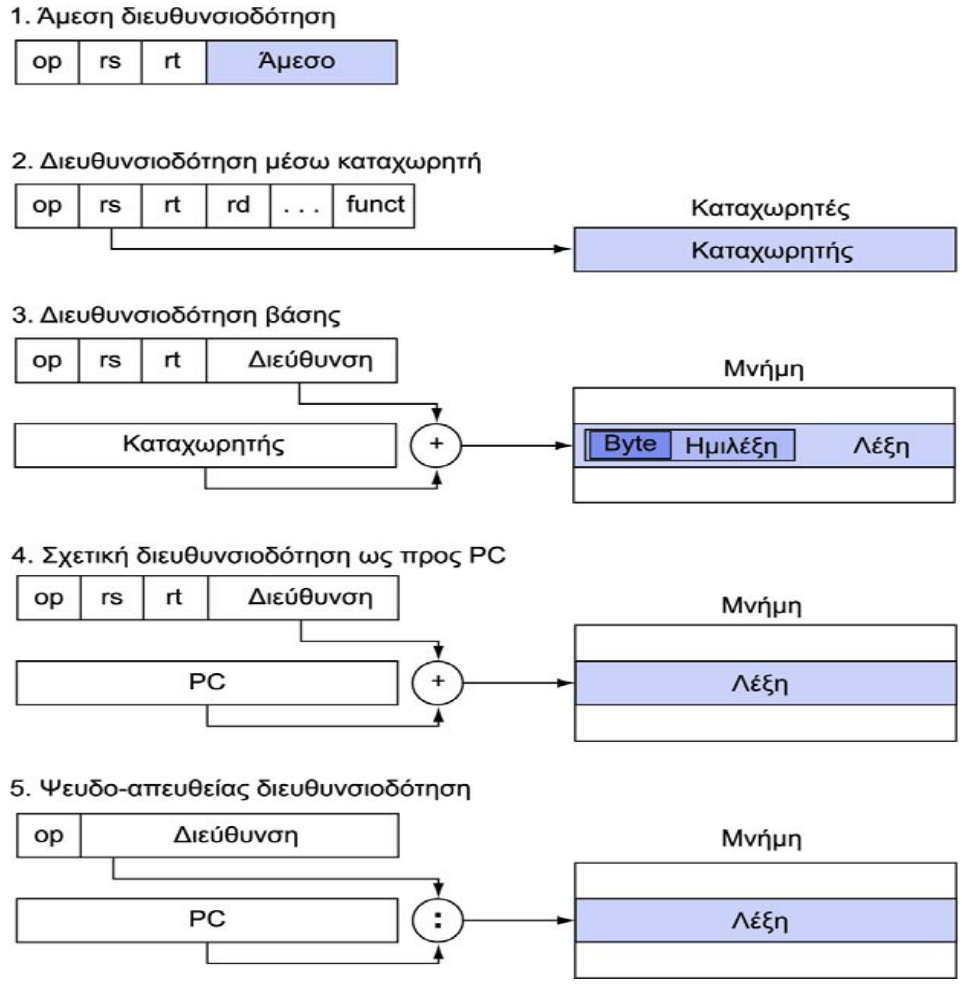
    lw $t0, 20 ($t1)
- **Άμεση διευθυνσιοδότηση (Immediate addressing)**
  - **Ο τελεστέος είναι μια μικρή σταθερά και περιέχεται στην εντολή**

    addi $t0, $t1, 4 (signed 16-bit integer)

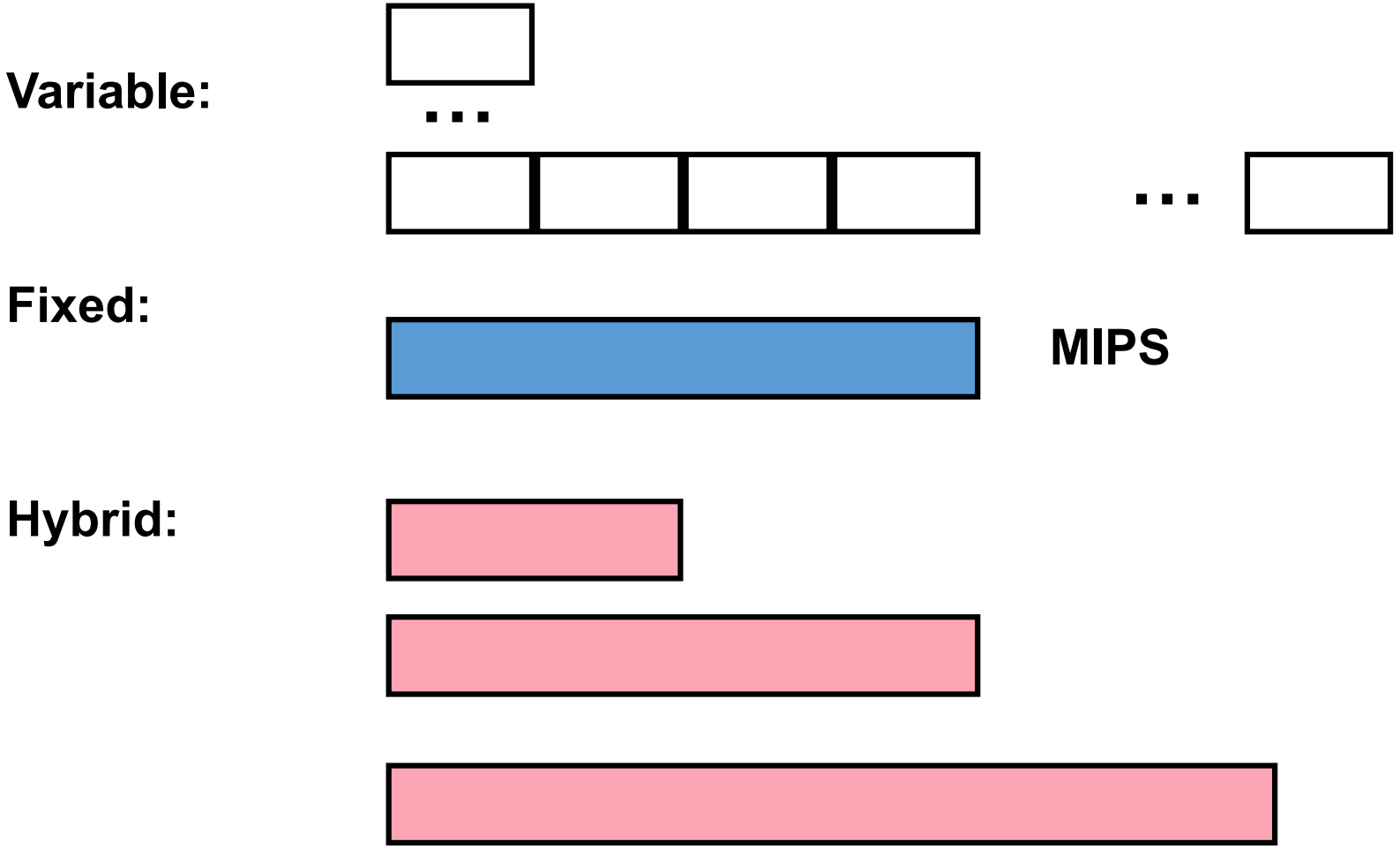
Διευθυνσιοδότηση Εντολών

- 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



Μορφές ISA



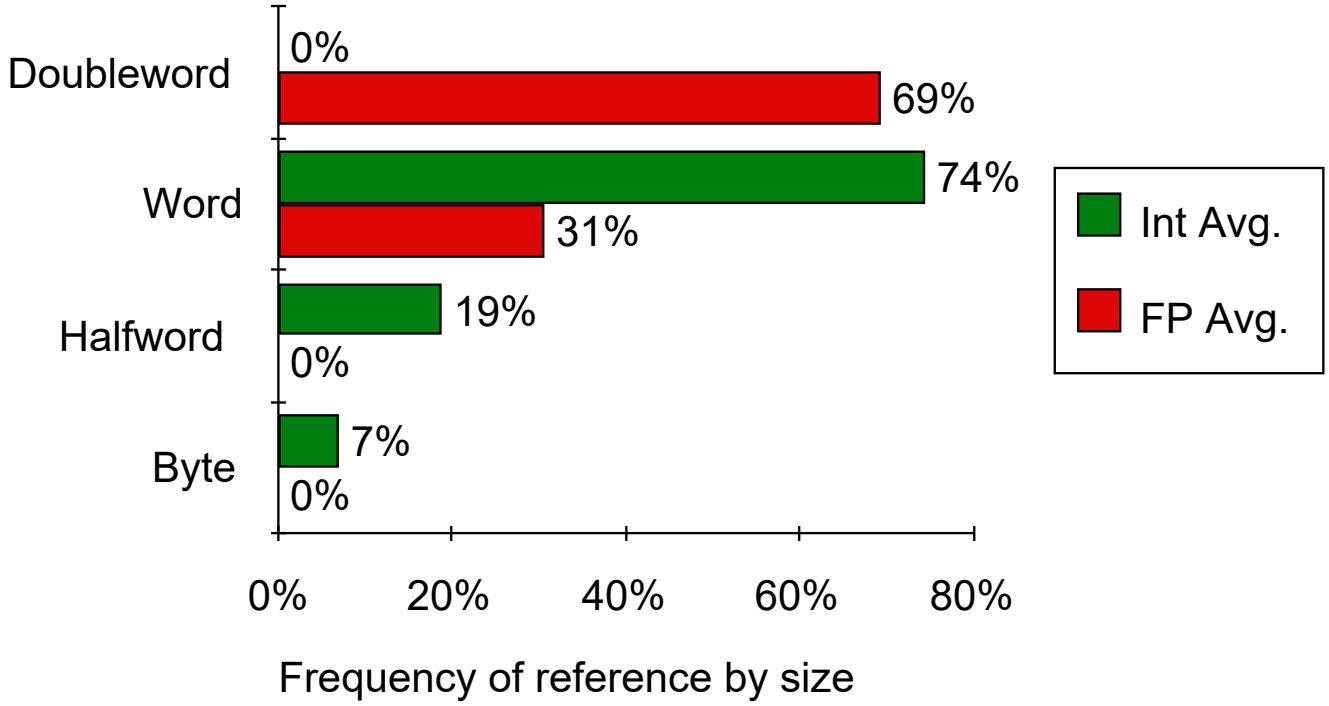
Top 10 80x86 Instructions

° Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

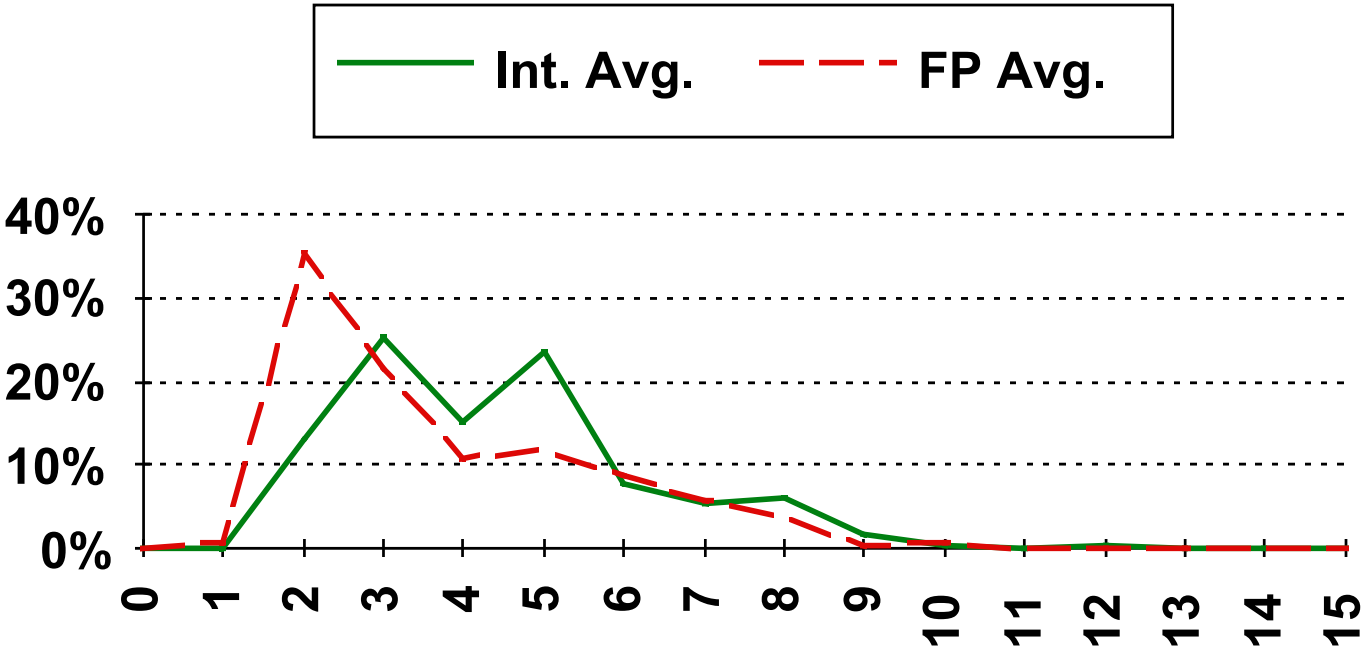
° **Simple instructions dominate instruction frequency**

Operand Size Usage

- **Υποστήριξη των ακόλουθων data sizes & types:**
  8-bit, 16-bit, 32-bit integers and
  32-bit and 64-bit IEEE 754 floating point numbers



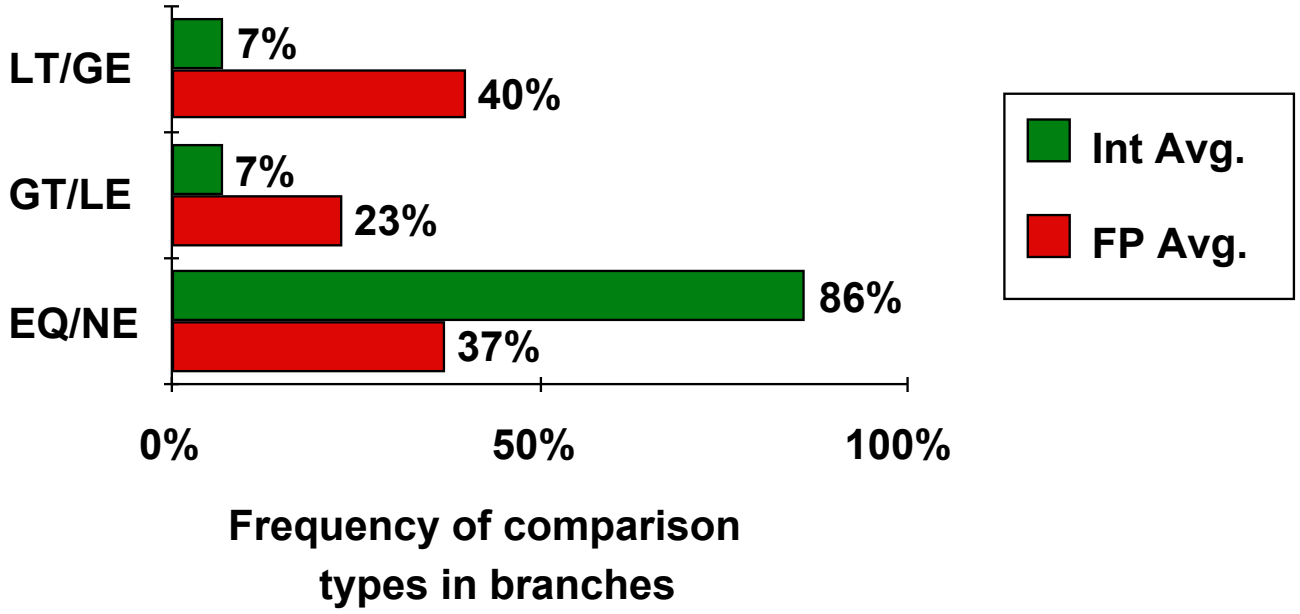
Conditional Branch Distance



75% των integer branches είναι 2 με 4 instructions

Conditional Branch Addressing

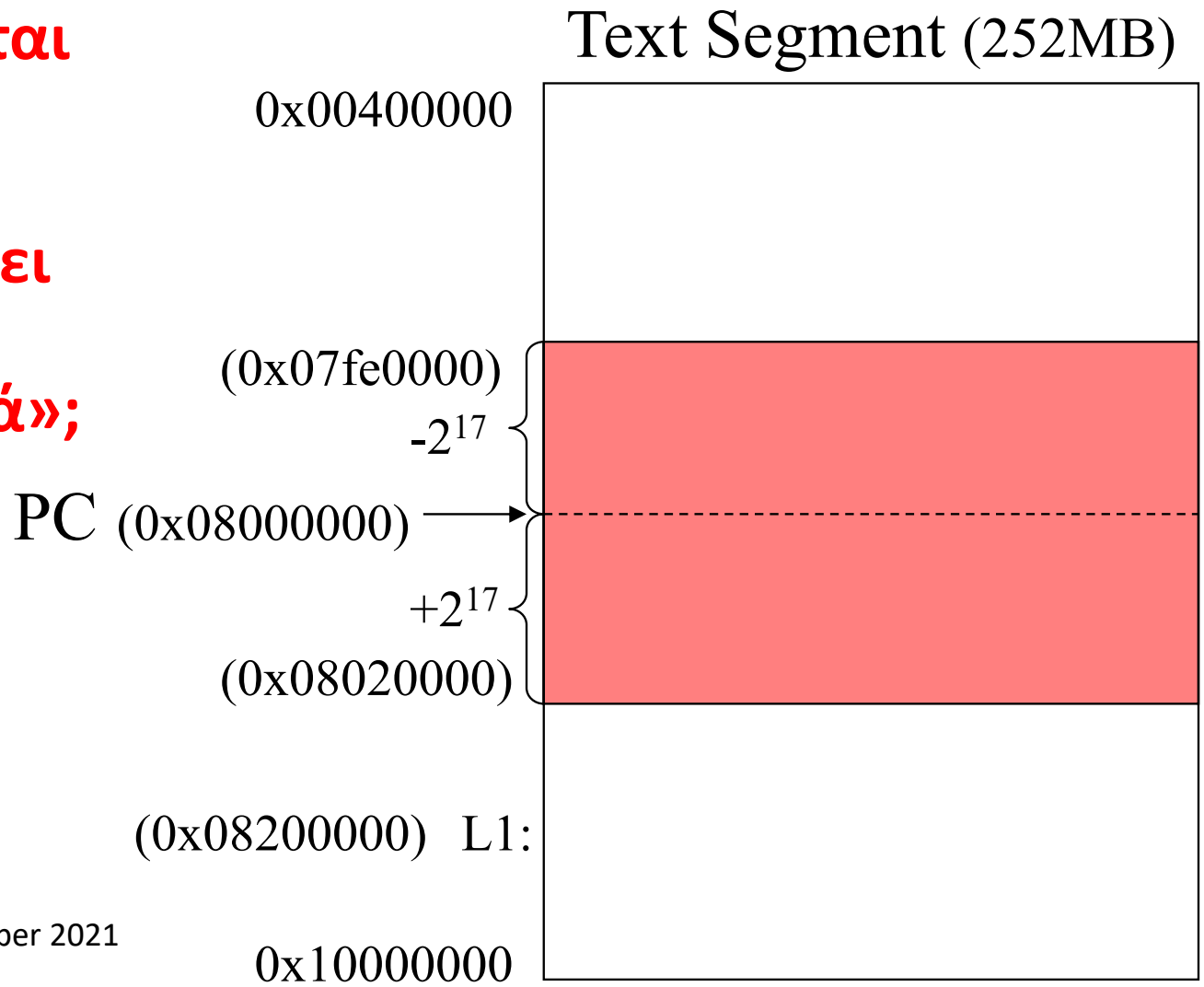
- Σχετική διεύθυνση ως προς το PC
- Τουλάχιστον 8 bits προτείνονται (+-128 instructions)
- Compare Equal/Not Equal σημαντικό για integer προγράμματα (86%)



Frequency of comparison types in branches

Παράδειγμα : Απομακρυσμένες Διευθύνσεις

Τι γίνεται αν ένα branch πηγαίνει «πολύ μακρυά»;




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

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

Δείκτες (Pointers)

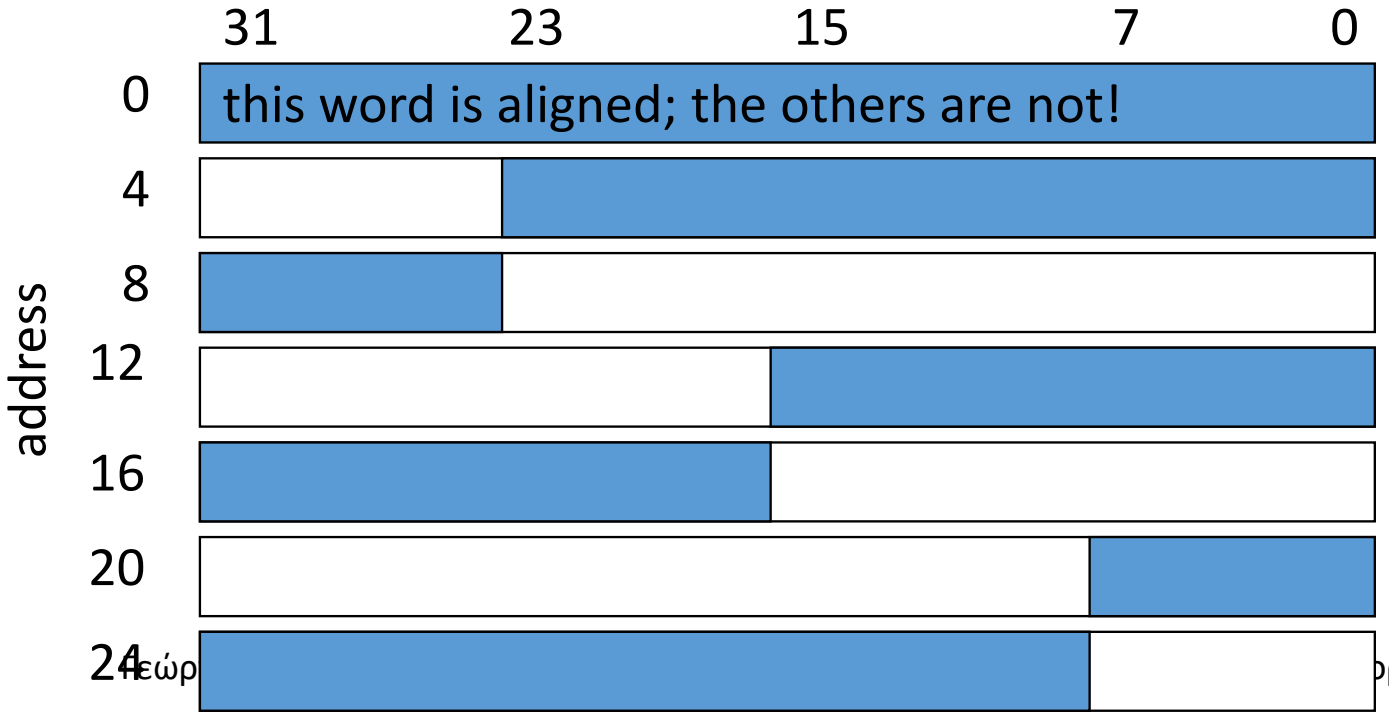
- **Pointer:** Μια μεταβλητή, η οποία περιέχει τη διεύθυνση μιας άλλης μεταβλητής
  - Αποτελεί τη HLL έκφραση της διεύθυνσης μνήμης σε γλώσσα μηχανής
- Γιατί χρησιμοποιούμε δείκτες;
  - Κάποιες φορές είναι ο μοναδικός τρόπος για να εκφράσουμε κάποιο υπολογισμό
  - Πιο αποδοτικός και συμπτυγμένος κώδικας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



Assembly Code : Παράδειγμα (1)

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

```
p = &c;  /* p gets 0x10000000*/
lui $a0,0x1000 # p = 0x10000000

 x = *p;  /* x gets 100 */
 lw  $s0, 0($a0) # dereferencing p

*p = 200; /* c gets 200 */
 addi $t0,$0,200
 sw   $t0, 0($a0) # dereferencing p
```

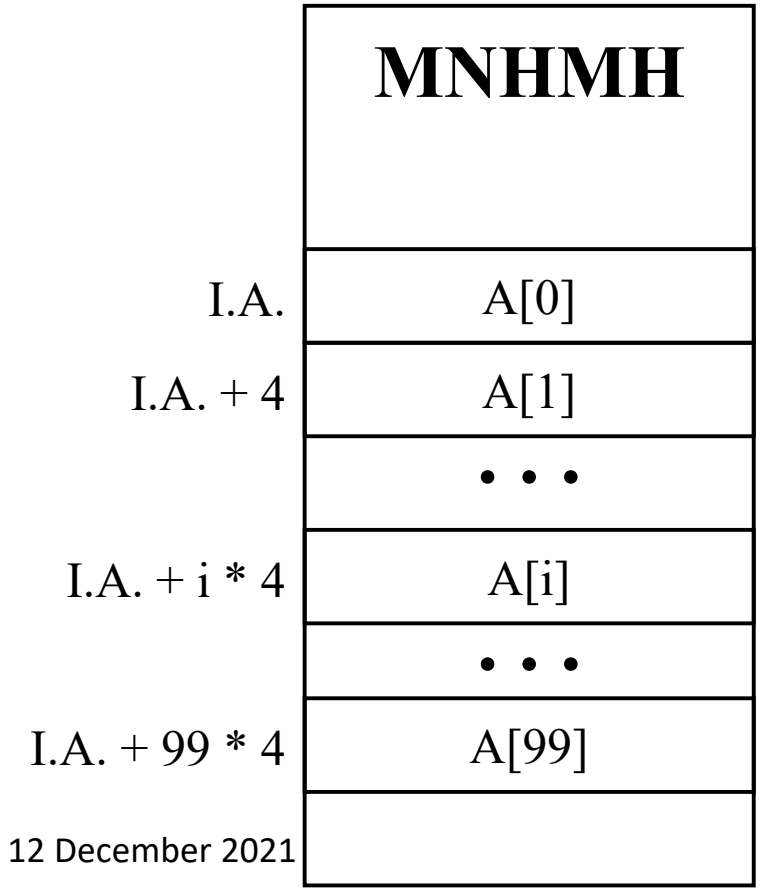
Assembly Code : Παράδειγμα (2)

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

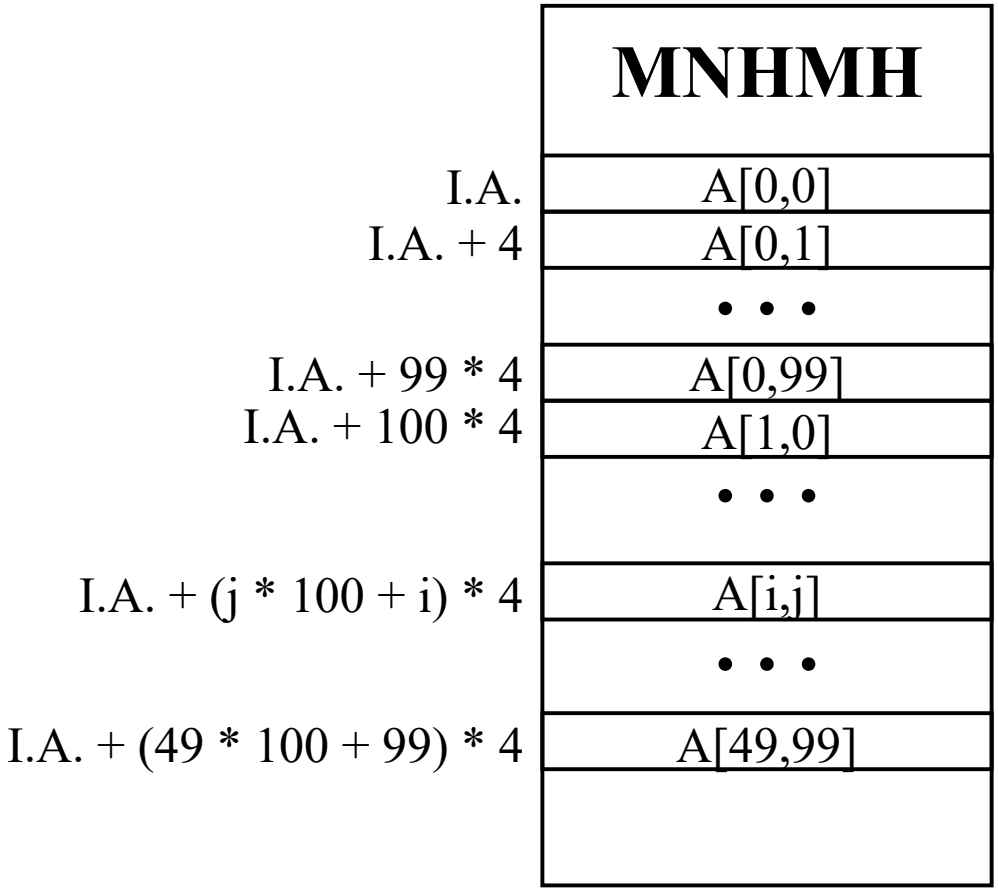
```
        mov  $t0,$a0
        lbu  $t1,0($t0)         /* derefence p */
        beq  $t1,$zero, Exit
Loop:   addi $t0,$t0,1          /* p++ */
        lbu  $t1,0($t0)         /* derefence p */
        bne  $t1,$zero, Loop
Exit:   sub  $v0,$t0,$a0
        jr   $ra
```

Πίνακες και Αποθήκευση στην Μνήμη

Μονοδιάστατος Πίνακας
int array[100]



Διδιάστατος Πίνακας
int array[50,100]



Μονοδιάστατοι Πίνακες

- **Υπολογισμός διεύθυνσης:**
  - **Δηλώνεται ένας πίνακας array[100]; Η αρχική διεύθυνση του πίνακα (δηλαδή η διεύθυνση του πρώτου στοιχείου) βρίσκεται στην διεύθυνση I.A. (initial address). Ποιά είναι η διεύθυνση του στοιχείου array[i];**
- **Απάντηση: address = I.A. + (i * sizeof(array element))**
  - **Για ένα πίνακα ακεραίων, η διεύθυνση του array[I] είναι:**
    - **address = I.A. + i * 4**
  - **Για ένα πίνακα από χαρακτήρες (char) είναι:**
    - **address = I.A. + i * 1**

Διδιάστατοι Πίνακες

- **Υπολογισμός διεύθυνσης:**
  - **Δηλώνεται ένας πίνακας array[nrows,ncolumns]; Η αρχική διεύθυνση του πίνακα (δηλαδή η διεύθυνση του στοιχείου array[0,0]) είναι I.A. (initial address). Ποιά είναι η διεύθυνση του στοιχείου array[j,i];**
- **Απάντηση:**

address = I.A. + (j * ncolumns * sizeof(array element)) + (i * sizeof(array element))

Ο πρώτος όρος (j * …) αντιστοιχεί στο μέγεθος σε bytes μιάς ολόκληρης γραμμής του πίνακα, ενώ ο δεύτερος (i * …) είναι ίδιος με την περίπτωση μονοδιάστατου πίνακα.

Για τον πίνακα int array[50, 100], η διεύθυνση του array[j, i] είναι:

address = I.A. + j * 100 * 4 + i * 4

Για ένα πίνακα από χαρακτήρες (char) με τις ίδιες διαστάσεις:

address = I.A. + j * 100 * 1 + i * 1

Επανάληψη MIPS (2)

- **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**

Συμβάσεις Χρήσης Καταχωρητών

- **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)**

Συμβάσεις Χρήσης Καταχωρητών

- **$t0..$t9 : «προσωρινοί» (temporary), χάνονται μετά από μια κλήση διαδικασίας ή συνάρτησης**
- **$s0..$s7 : saved, για αποθήκευση τιμών που διατηρούνται για περισσότερο χρόνο και μπορούν να διατηρούν την τιμή τους και μετά από μία κλήση διαδικασίας ή συνάρτησης**
- **Οι συναρτήσεις πρέπει να διατηρήσουν αναλλοίωτες τις τιμές των καταχωρητών $s0-$s7**
  - **Εάν μεταβάλλουν κάποιους από αυτούς τους καταχωρητές, αυτοί πρέπει να σωθούν στην στοίβα κατά την έναρξη της συνάρτησης**

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