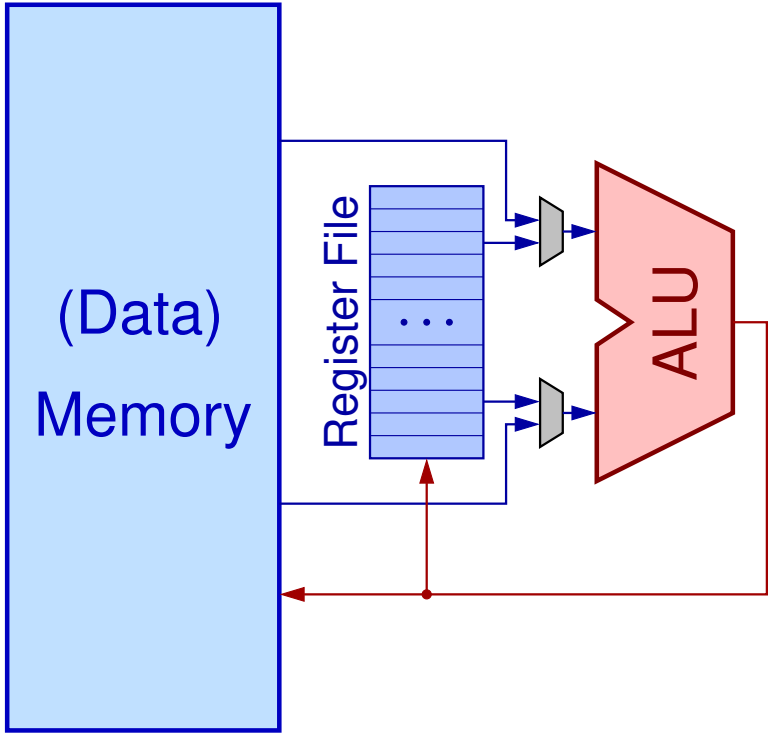
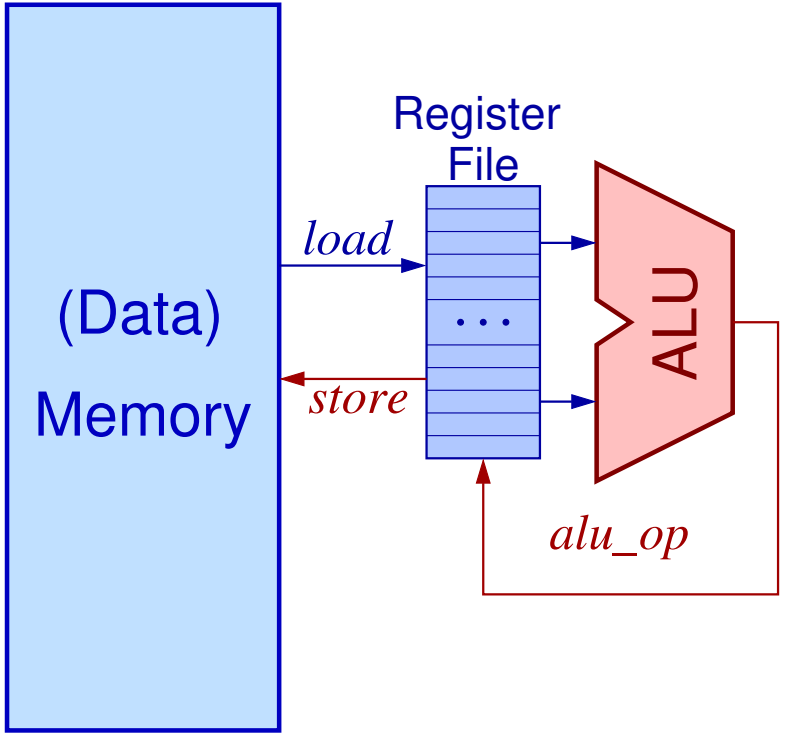
Επανάληψη 1: Προγραμματισμός σε Assembly – το παράδειγμα του Ρεπερτορίου Εντολών RISC-V

Άνοιξη 2024 – Μανόλης Κατεβαίνης

Τελεστέοι: Οπουδήποτε ή σε Καταχωρητές μόνον;

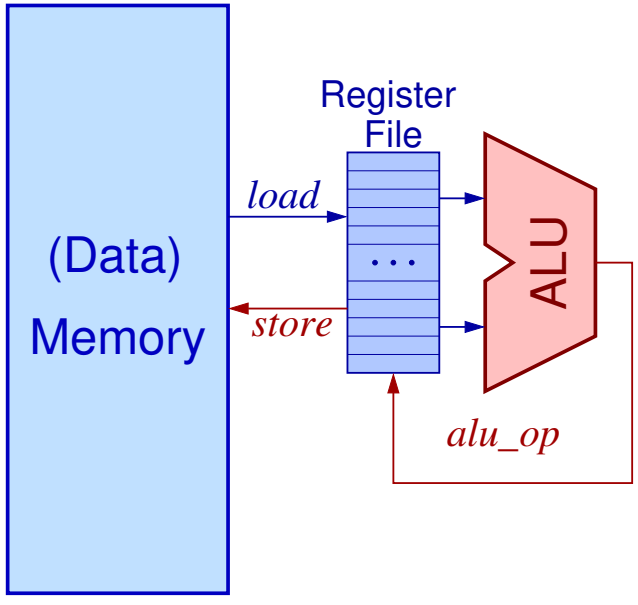


Οι τελεστέοι οπουδήποτε:
"**CISC**"
(Complex Instr. Set Computer)



Οι τελεστέοι μόνο σε καταχωρητές:
"**RISC**"
(Reduced Instr. Set Computer)

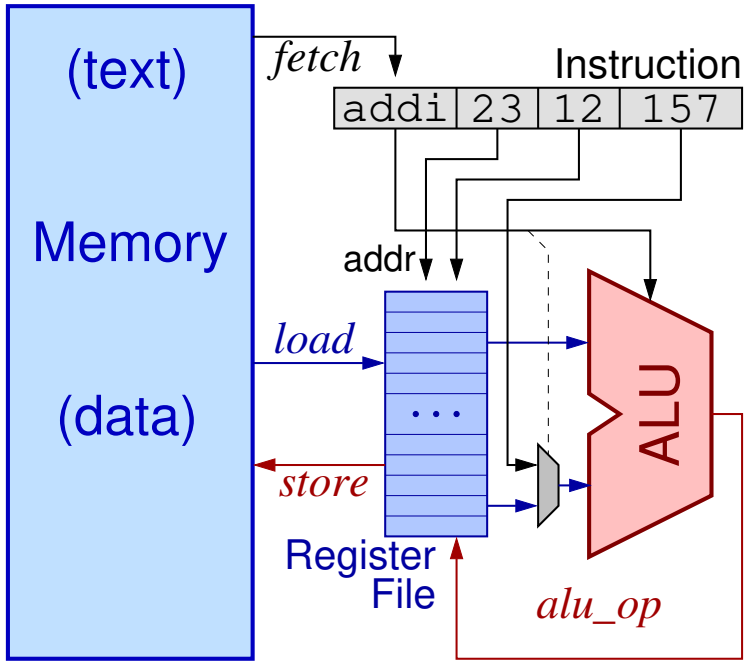
Εντολές reg2reg στον RISC-V



- `add x23, x12, x14`
  - σημαίνει: x23 ← x12 + x14
  - *xNN* σημαίνει «το περιεχόμενο του καταχωρητή υπ' αριθμ. *NN*»
  - καταχωρητές: x0, x1,… εώς και x31
  - ο x0 είναι «ειδικός»: x0 ≡ 0 πάντα!
  - επιτρέπονται εγγγραφές στον x0, αλλά αγνοούνται!
  - βλ. σχόλια και εξηγήσεις «γιατί» παρακάτω
- `sub x23, x12, x14` σημαίνει: x23 ← x12 - x14
- ομοίως: `and x23, x12, x14` `or` …, `xor` …, `mul` …

Σταθεροί αριθμοί στον RISC-V

- `addi x23, x12, 157`
  - "add immediate (constant)"
  - σημαίνει: x23 ← x12 + 157
  - `addi x23, x12, 14`
    προσθέτει τον αριθμό 14, *όχι* το περιεχόμενο του καταχωρητή x14, στο περιεχόμενο του καταχωρ. x12
  - οι σταθερές είναι πάντα προσημασμένες στον RISC-V



- *Δεν* υπάρχει "subi x23, x12, 14" ⇔ addi x23, x12, -14
- *Δεν* υπάρχει "subii x23, 157, x14": πολύ σπάνια, δεν αξίζει

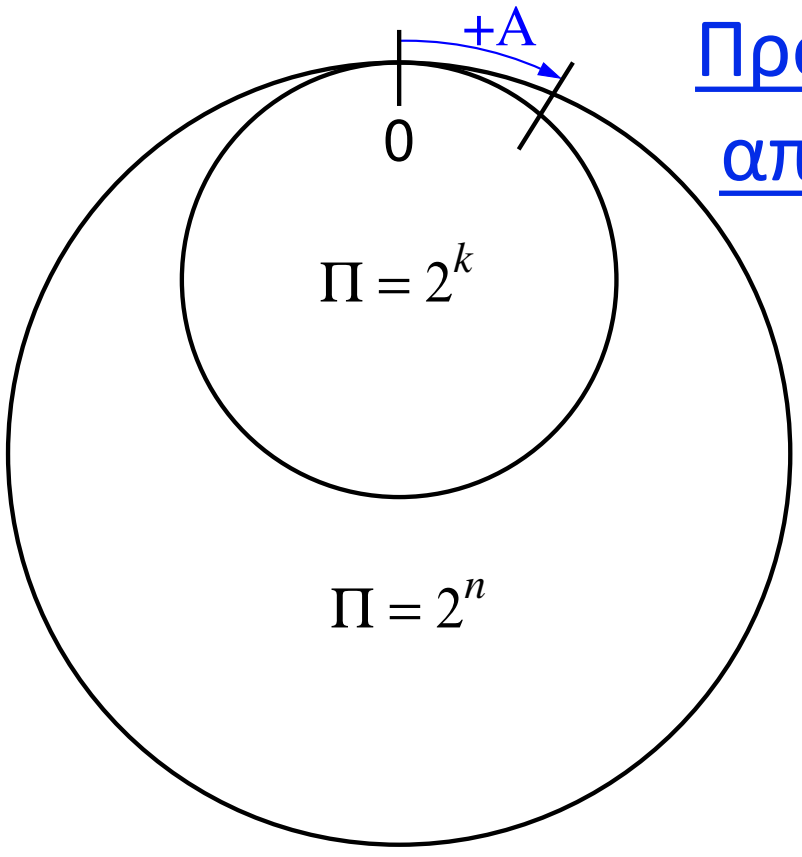
Σύνθεση άλλων τύπων εντολών σε RISC-style ISA's

- add x8, x0, x0 # i=0; αρχικοποίηση (μτβλ. i στον x8)
- addi x8, x0, 0 # i=0; εναλλακτική αρχικοποίηση
- addi x8, x0, 1 # i=1; αρχικοπ. σε μη μηδενική τιμή
- add x8, x9, x0 # i=j; αντιγραφή μεταβλητών (j στον x9)
- addi x8, x9, 0 # i=j; εναλλακτική αντιγραφή
- addi x8, x8, 1 # i=i+1;
- add x8, x8, x9 # i=i+j; "two-operand add"
- sub x8, x0, x8 # i=-i; αλγεβρικό αντίθετο

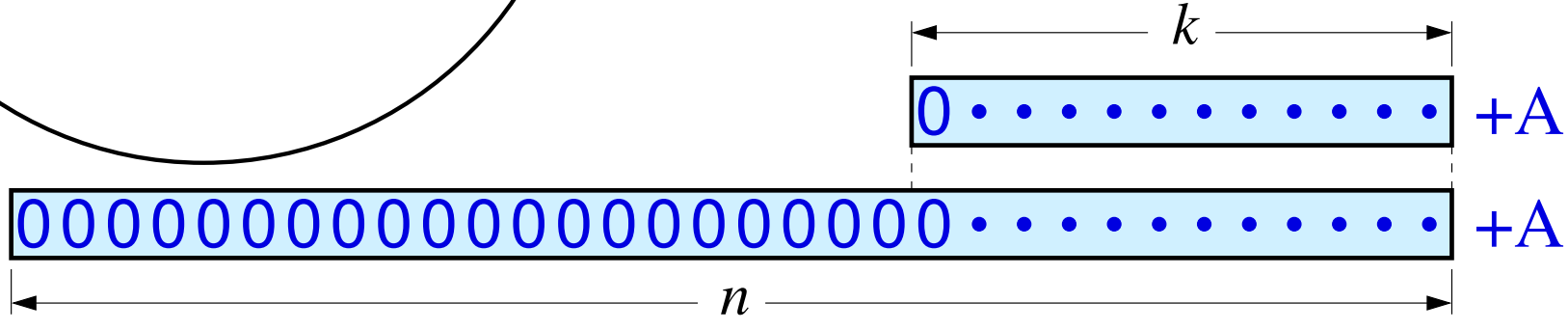
Μετατροπές Ακεραίων από λίγα σε πολλά bits

- Απρόσημοι (*Unsigned*) ακέραιοι (≥0) (ΗΥ-120 Εργ.5):
  - $b_{n-1}\times 2^{n-1} + \ldots + b_2\times 2^2 + b_1\times 2^1 + b_0\times 2^0$
  - πάντα ο ίδιος αριθμός, οσαδήποτε μηδενικά και αν προστεθούν αριστερά από τον MS άσσο
- Προσημασμένοι (*Signed*) ακέραιοι σε 2's Complement:
  - Όταν MS bit ==0 ⇒ ≥0, ίδιος όπως εάν unsigned
  - Όταν MS bit ==1 ⇒ <0, όσος εάν unsigned μείον 2^j όπου j το πλήθος των bits του αριθμού
  - Όταν το j αλλάζει??

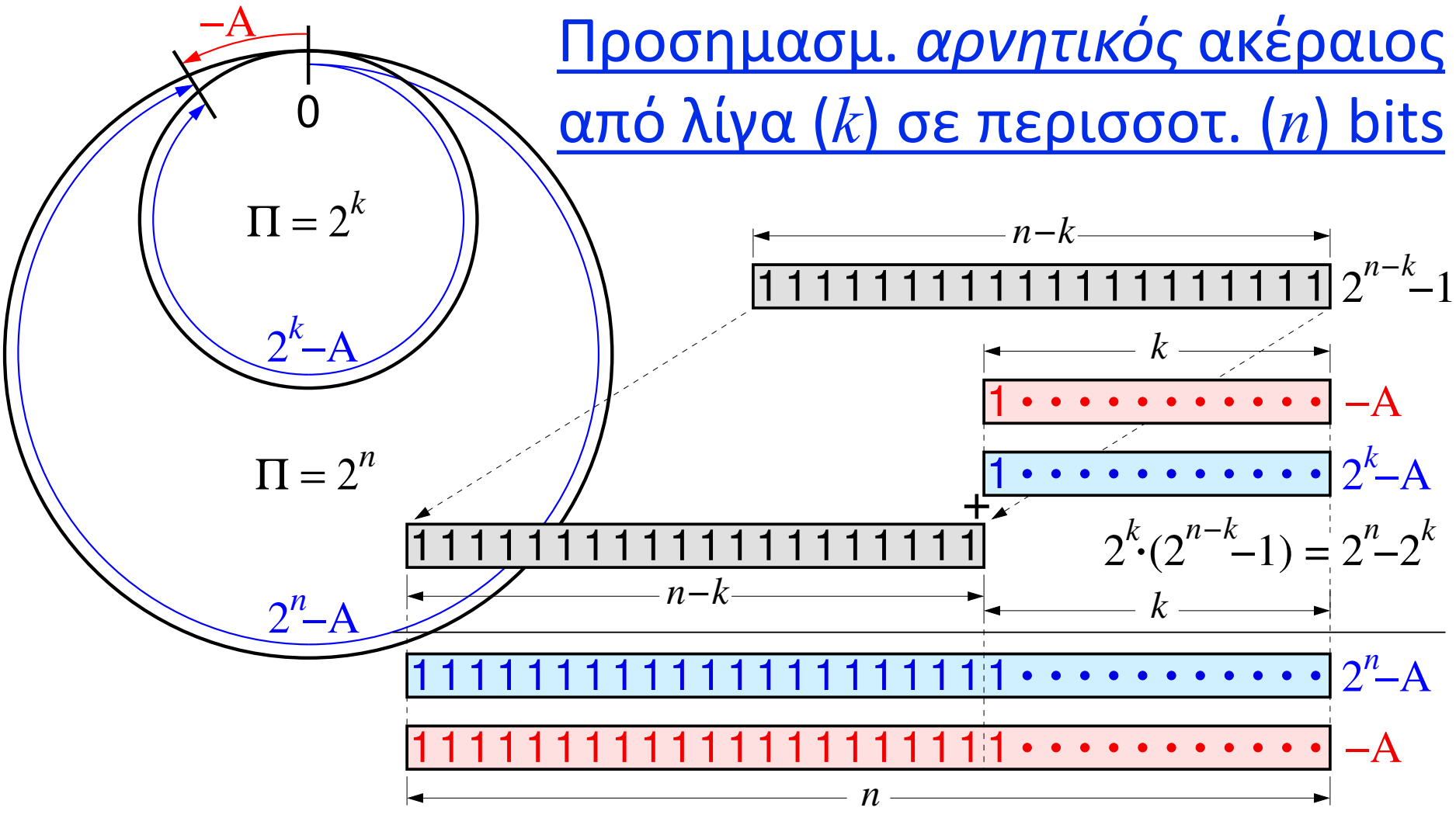
Προσημασμένος *θετικός* ακέραιος από λίγα (k) σε περισσοτ. (n) bits



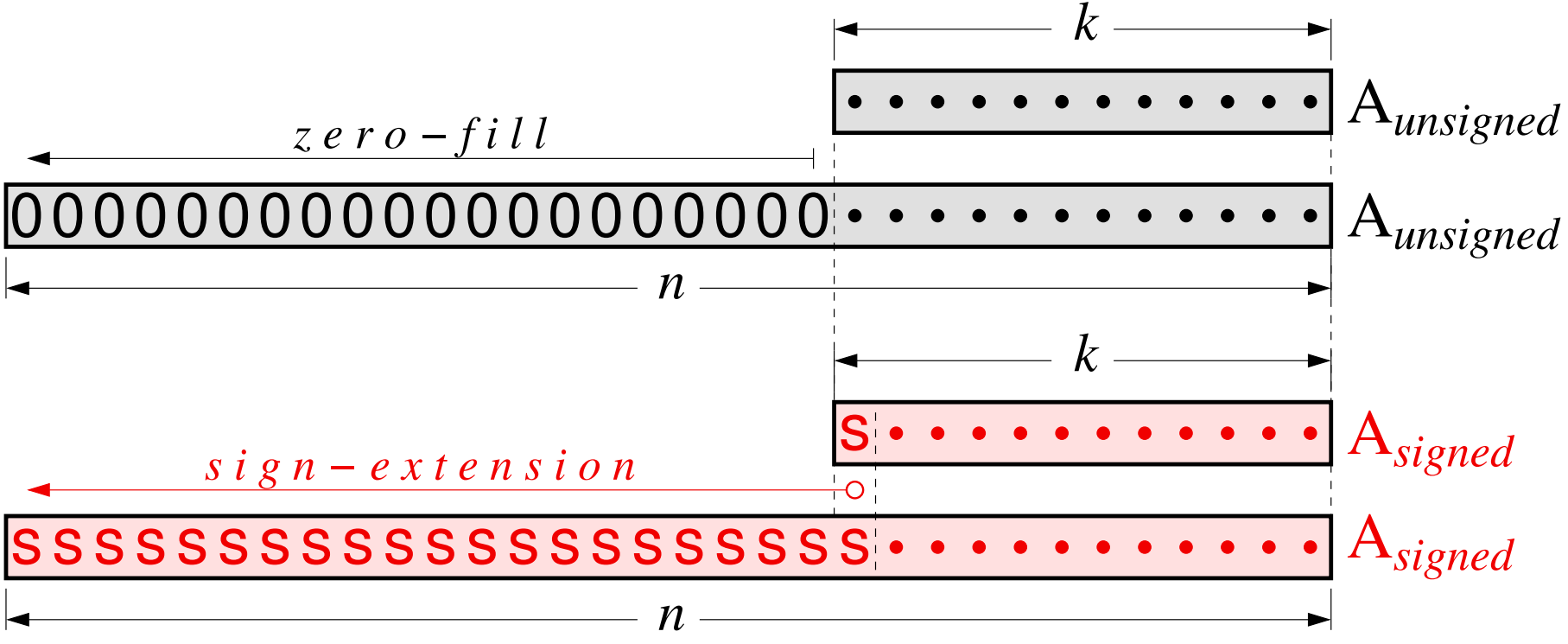
- MS bit == 0
- ίδιος αριθμός όπως και απρόσημος
- ίδιος οσαδήποτε μηδενικά αριστερά



Προσημασμ. *αρνητικός* ακέραιος από λίγα (k) σε περισσοτ. (n) bits



Προσημασμένοι σε περισ. bits: *Επέκταση Προσήμου*

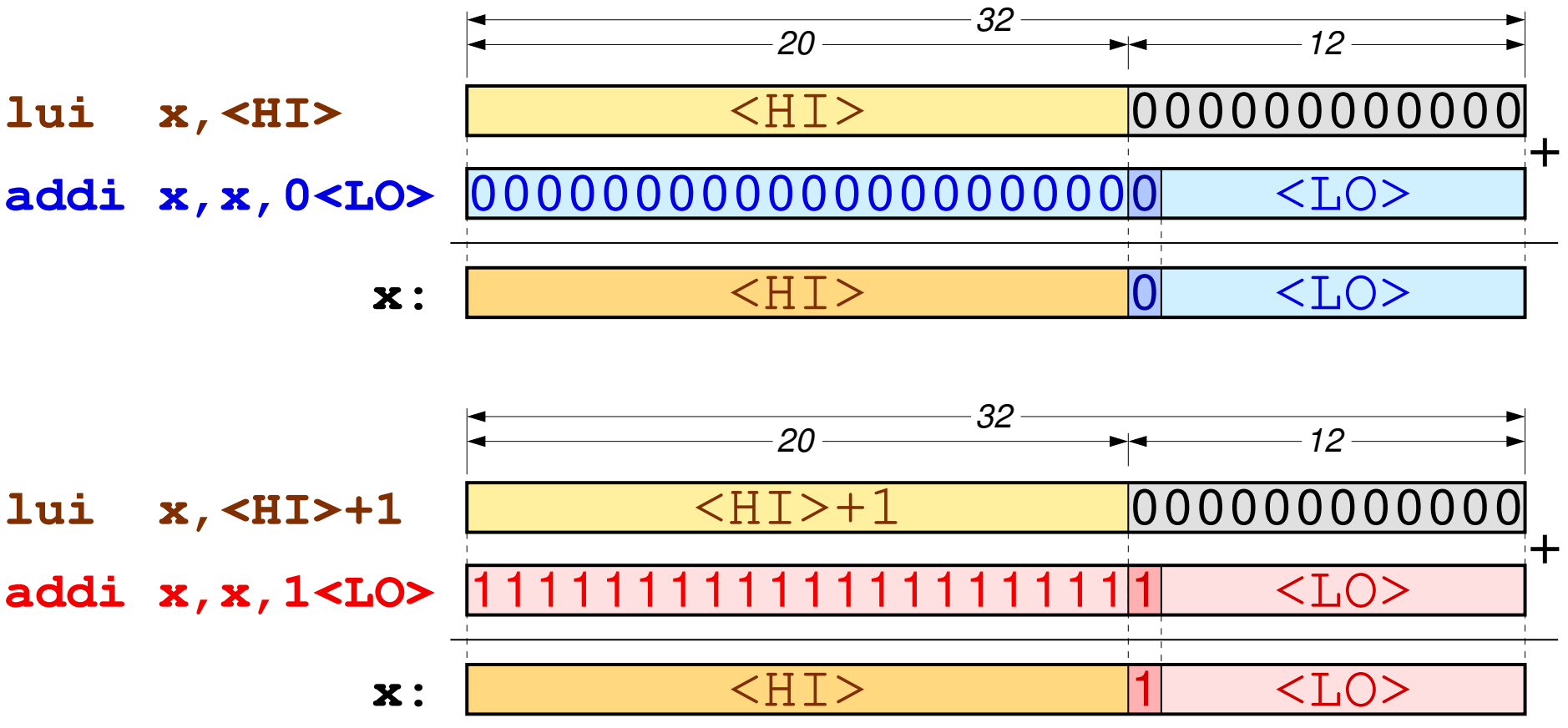


- Απρόσημοι: Συμπλήρωση Μηδενικών (ανεξαρτήτως MS bit)
- Προσημασμένοι *(όλα τα Immed. στον RV)*: Επέκταση Προσήμου

Load Upper Immediate (`lui`): σταθ. 20 bits αριστερά

- U-format: opcode (7 bits), rd (5 bits), Imm20
- Βάζει τη σταθ. Imm20 στα 20 «αριστερά» (MS) bits του rd, μηδενίζοντας τα 12 «δεξιά» (LS) bits του
- Χρήση: σύνθεση αυθαίρετων 32-μπιτων σταθερών
  - βλ. επόμενη διαφάνεια

Σύνθεση αυθαίρετης 32-μπιτης σταθεράς στον κατ. x



Διακλαδώσεις υπό Συνθήκη στον RISC-V

- `beq x26, x27, label` # if (x26==x27) goto label
- `bne x26, x27, label` # if (x26 ≠ x27) goto label
- `blt x26, x27, label` # if (x26 < x27) goto label
- `bge x26, x27, label` # if (x26 ≥ x27) goto label

- γιατί δεν χρειάζονται `ble` (≤), `bgt` (>) ? (άσκηση…)
- `bltu bgeu` → unsigned variants
- Δεν υπάρχουν συγκρίσεις καταχωρητή-σταθεράς
  – γιατί; (άσκηση…)

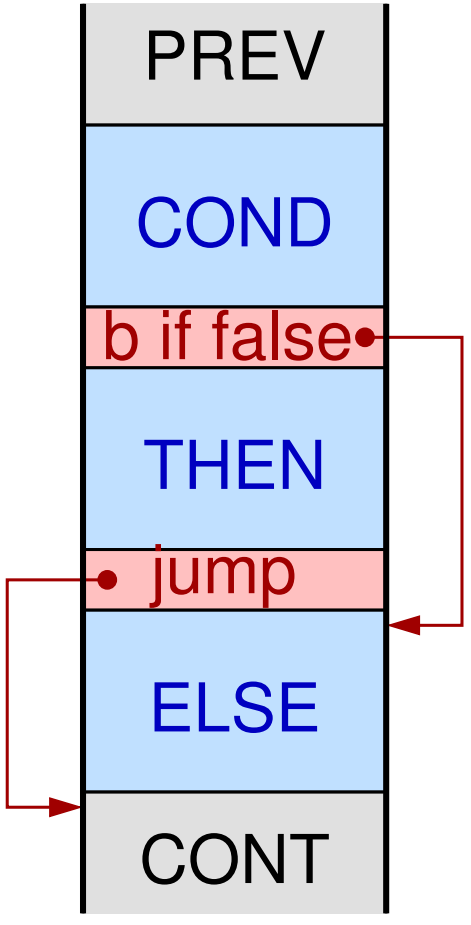
[πόσα bits?]

Μετάφραση του if-then-else σε Assembly

register int i, j, f, g, h;

if (i==j) { f = g+h; } else { f = g-h; }
h = 0;

```
        bne i, j, else1
        add f, g, h
        j  cont1          #jump
else1:  sub f, g, h
cont1:  add h, zero, zero
```

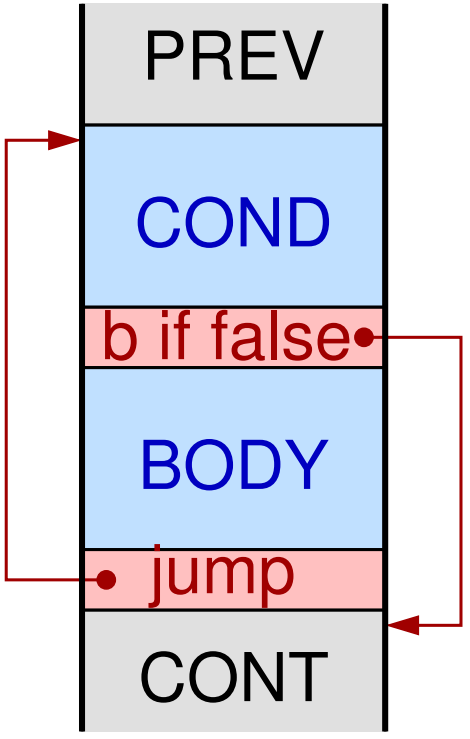


Μετάφραση Βρόχου while σε Assembly

```
long long int  i, key, table[N];
i = 0;
while ( table[i] != key ) { i++; }
      /* table[] must contain at least one element==key */
```

t1: i
t2: key
s0: table

```
        add  t1, x0, x0   # i=0;
loop1:  slli t0, t1, 3    # 8*i
        add  t0, s0, t0   # &table[i]
        ld   t0, 0(t0)    # table[i]
        beq  t0, t2, exit1
        addi t1, t1, 1    # i++;
        j    loop1
exit1:  ...
```



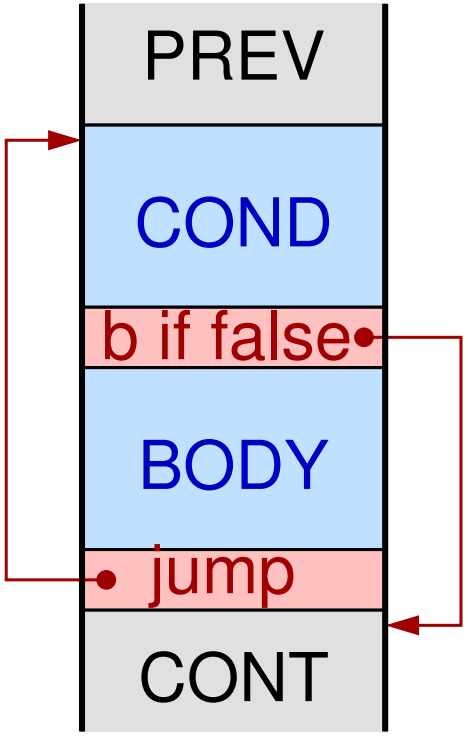
Βελτιστοποίηση Βρόχου με pointers

```
long long int  *p, key, table[N];
p = table;
while ( *p != key ) { p++; }
```

t1: p
t2: key
s0: table

/* table[] must contain at least one element==key */

```
        add   t1, s0, x0    # p= table;
loop1:  ld    t0, 0(t1)     # t0=*p
        beq   t0, t2, exit1
        addi  t1, t1, 8     # p++;
        j   loop1
exit1:  ...
```



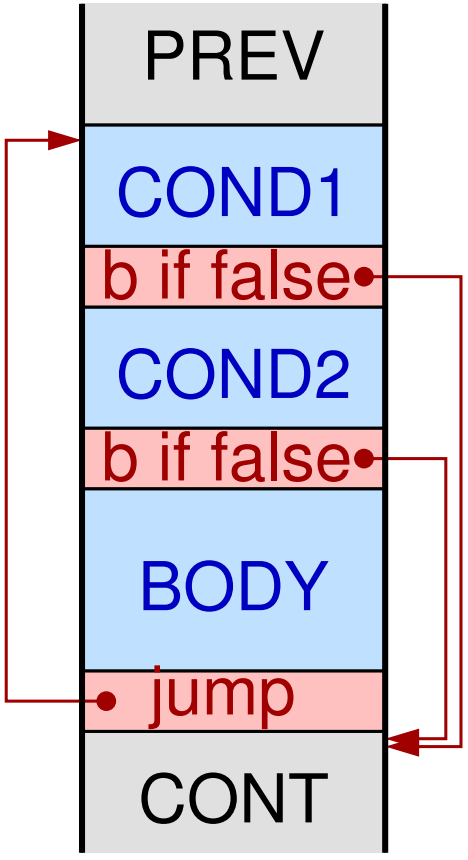
Άσκηση 5.5: αλλάξτε τη χωροθέτηση των στοιχείων του βρόχου ούτως ώστε οι ανακυκλώσεις μετά την είσοδο να εκτελούν μόνο μία εντολή CTI καθεμία

Σύνθετη Συνθήκη με υπολογισμό “Short-Circuit”

```
struct node {int value; struct node *next;} *p;
while ( p!=NULL && p->value != key )
        { p = p->next; }
    /* assume 32-bit RISC-V here */
```

t1: p
t2: key

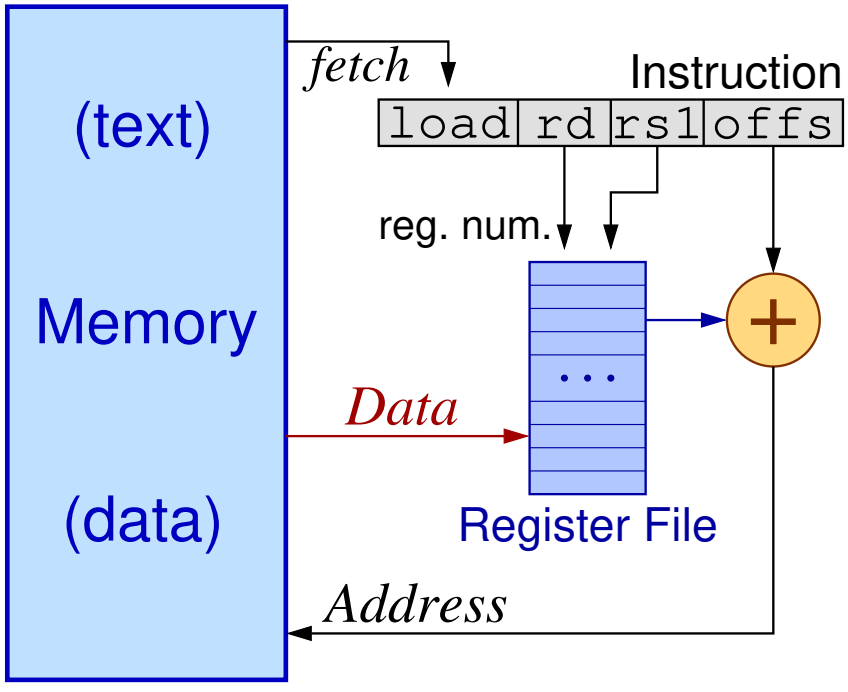
```
loop2: beq t1, x0, exit2
       lw  t0, 0(t1)    # p->value
       beq t0, t2, exit2
       lw  t1, 4(t1)    # p=p->next;
       j   loop2
exit2: ...
```



- Εάν ο p είναι NULL, *απαγορεύεται* να υπολογίσουμε (δηλ. προσπελάσουμε) το p->value

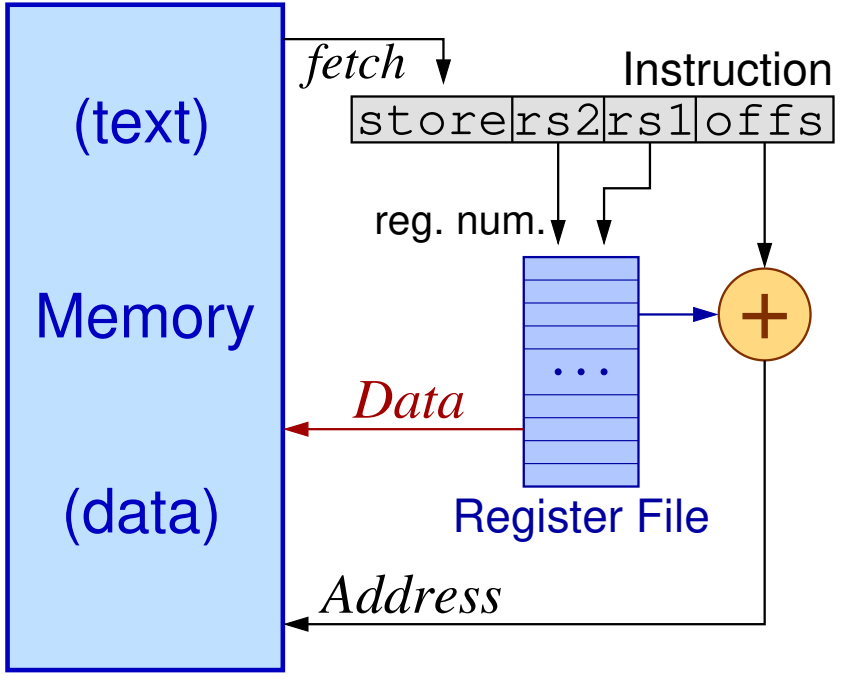
Οι εντολές Load και Store στον RISC-V

`lw rd, offset(rs1)`



$rd \leftarrow M[offset+(rs1)]$

`sw rs2, offset(rs1)`



$rs2 \rightarrow M[offset+(rs1)]$

- Μοναδικό Addressing Mode
- *Offset*: πάντοτε *signed* (12 bits)

Πλάτος Δεδομένων: πόσα Bytes ανάγν./εγγρ. Μνήμης;

- `lb` / `sb` → load/store ***Byte***: διαβάζει/γράφει 1 Byte
- `lh` / `sh` → load/store ***Half***: διαβάζει/γράφει 2 Bytes
- `lw` / `sw` → load/store ***Word***: διαβάζει/γράφει 4 Bytes

Ο βασικός RISC-V, "*RV32I*", είναι 32-μπιτος & έχει αυτές

– ο καταχ. rd ή rs2 που παίρνει/δίνει τα data: 32 bits – βλ. επόμ.

Η 64-μπιτη επέκταση "*RV64I*" έχει επίσης τις:

- `ld` / `sd` → load/store ***Double***: διαβάζει/γράφει 8 Bytes

  – στον *RV32I* δεν χωράει ο Double σε καταχωρητή
- Στην C: `int` → ***Word*** – `long long int` → ***Double***

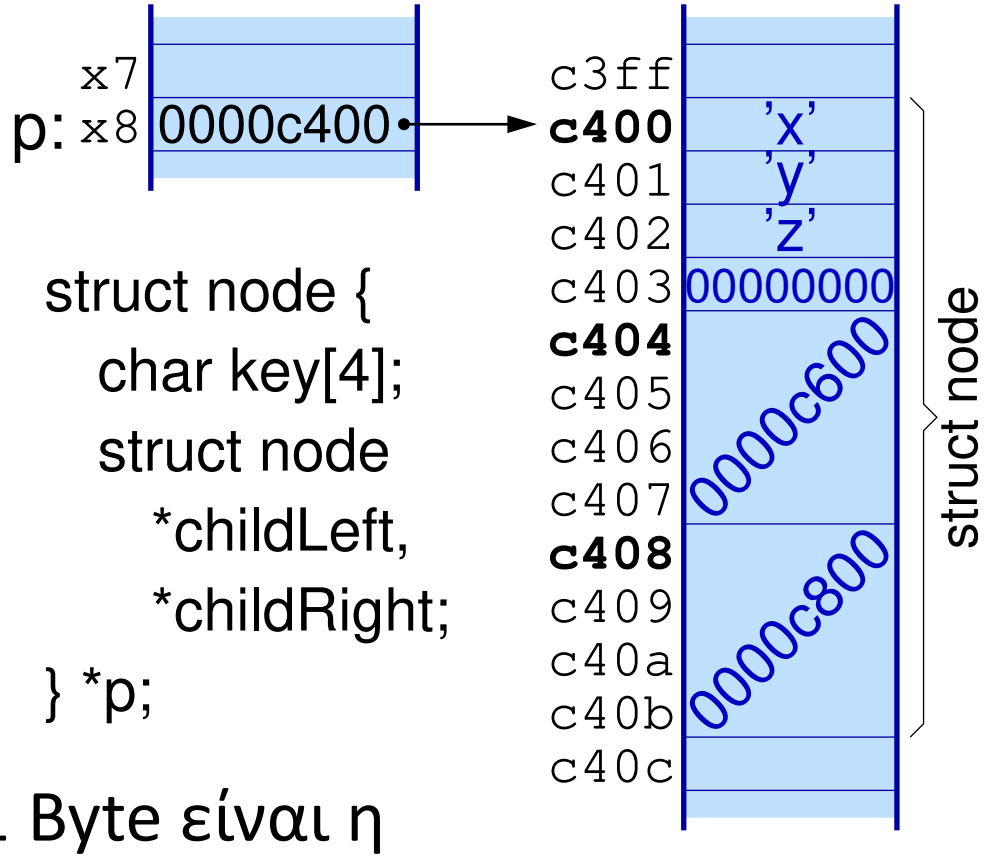
«Στενές» μεταφορές: τα υπόλοιπα bits του καταχωρητή;

Όταν τα μεταφερόμενα data έχουν λιγότερα bits απ' όσα ο καταχωρητής rs2 ή rd:

- Οι εντολές Store γράφουν στη μνήμη όσα bits τους λέμε, από την *λιγότ. σημαντ.* (*LS*) πλευρά του καταχ., *ως έχουν*
  - εάν ο αριθμός χωράει σε αυτά τα λιγότερα bits, τότε αυτός παραμένει αμετάβλητος, είτε είναι signed είτε είναι unsigned
- Οι εντολές Load διαβάζουν όσα bits τους λέμε από τη μνήμη, τα βάζουν στα *LS* bits του καταχ., και τα υπόλοιπα:
  - `lb`, `lh` (και `lw` σε *RV64I*) θεωρούν *signed* ⇒ «επεκτ. προσ.»
  - `lbu`, `lhu` (και `lwu` σε *RV64I*) θεωρούν *unsigned* ⇒ "zero fill"

Χρήση 1: Προσπέλαση Στοιχείων Δομής μέσω Pointer

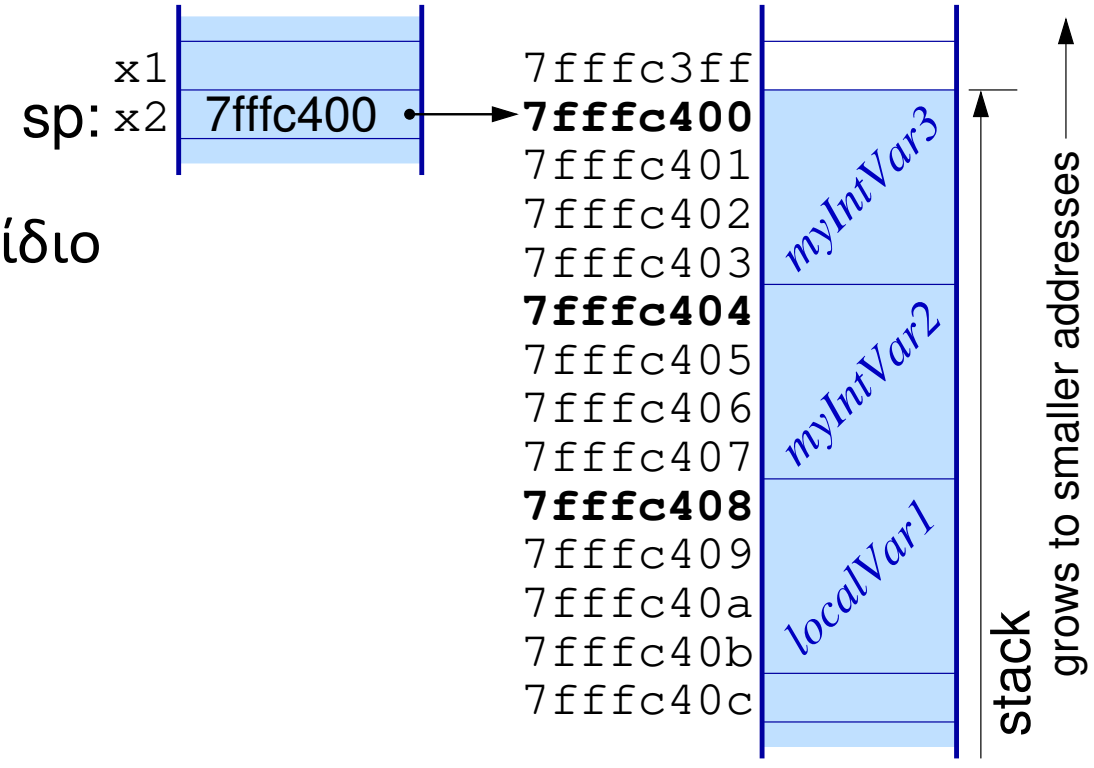
- `lbu x5, 0(x8)`
  - tmp0 = p->key[0];
- `lbu x6, 1(x8)`
  - tmp1 = p->key[1];
- `lw x7, 4(x8)`
  - tmp2 = p->childLeft;
- `lw x8, 8(x8)`
  - p = p->childRight;



- Διεύθυνση ποσοτήτας >1 Byte είναι η διεύθ. εκείνου του Byte της που έχει την *μικρότερη* διεύθ.

Χρήση 2: Προσπέλαση τοπικών μεταβλ. στη Στοίβα

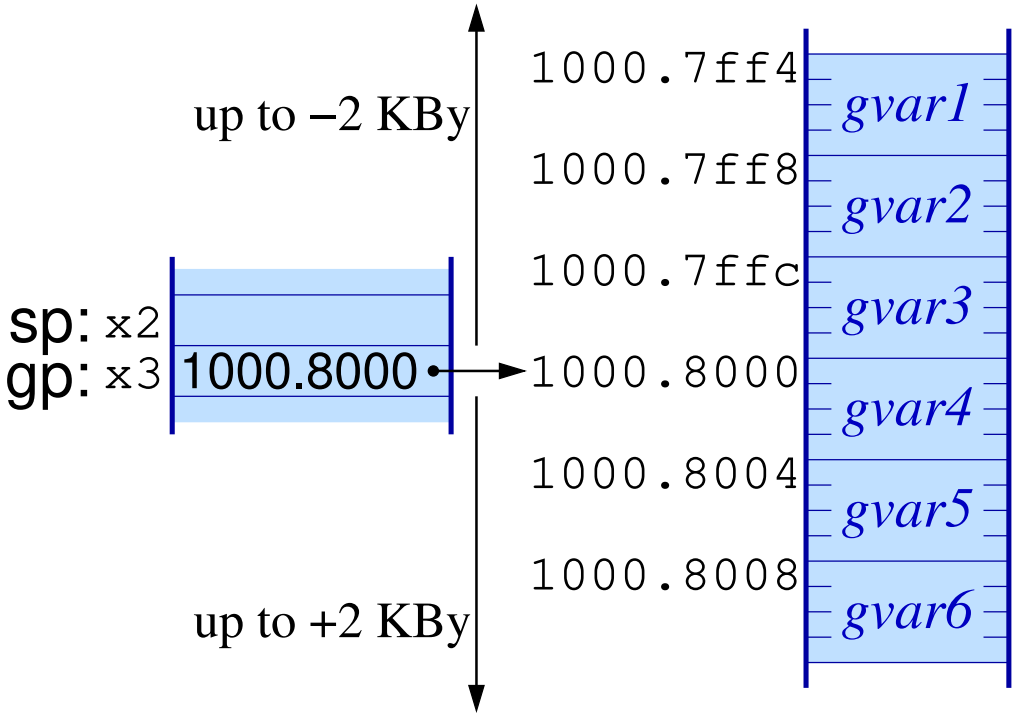
- `lw x5, 8(x2)`
- `lw x5, 8(sp)`
  - "x2" και "sp" είναι το ίδιο
  - tmp0 = localVar1;
- `sw x6, 4(sp)`
  - myIntVar2 = tmp1;
- `lw x7, 0(sp)`
  - tmp2 = myIntVar3;



"Local" variables in procedures: stack of activation frames

Χρήση 3: Καθολικές στατικές μεταβλητές μέσω **gp**

- Μεταβλητές ορισμένες εκτός διαδικασιών στη C: Στατικές, καθολικές (global)
- Χωριστά οι βαθμωτές (scalar – not arrays) εδώ, ώστε να χωρούν όλες, ει δυνατόν, σε 4 KBytes
- `x3` (**gp** – global pointer) δείχνει στη μέση του χώρου αυτού (12-bit signed offset)



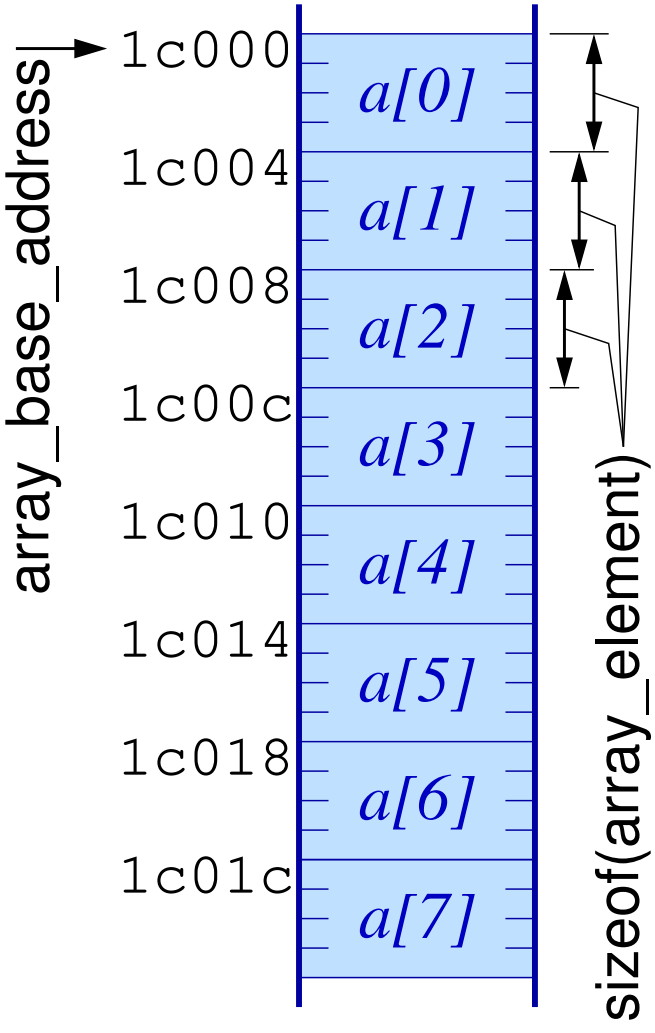
- `lw x5, -12(gp)` # tmp0 = gvar1;
- `sw x6, 4(gp)` # gvar5 = tmp1;

Πίνακες (arrays)

- Διεύθυνση Στοιχείου i =
  Διεύθυνση Βάσης +
  i × sizeof(*element*)
- Όταν βαθμωτά στοιχεία, συνήθ. sizeof(*element*) δύναμη του 2
  ⇒ πολλαπλασιασμ. = αρ. ολίσθηση
- Συνήθως οι Compilers βελτιστοπ.:

  for (i=0; i<N; i++) { … a[i] … } ⇒

  for (p=a; p<a+N; p++) { … *p … }

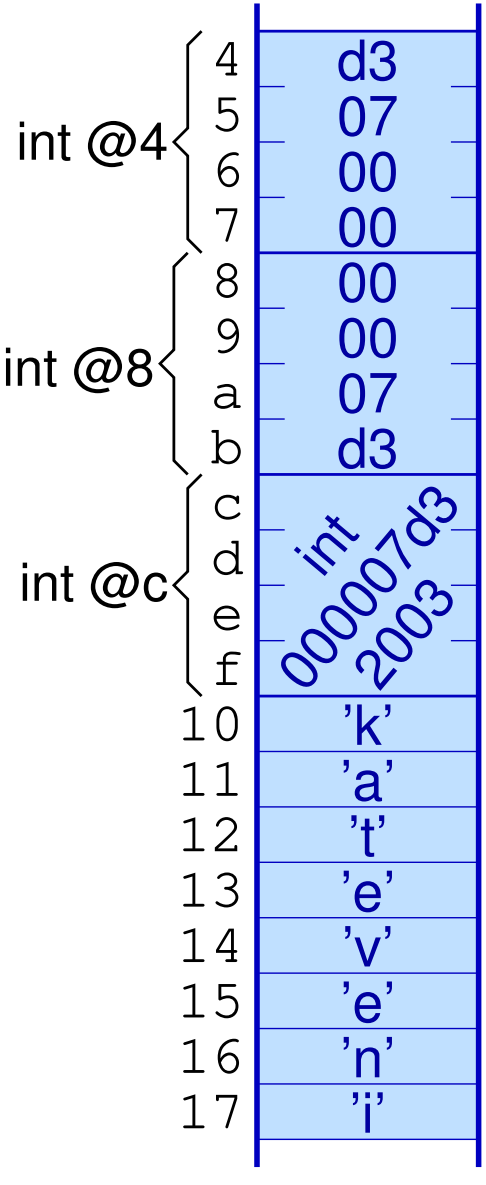


Μνήμη Byte-Adddressable

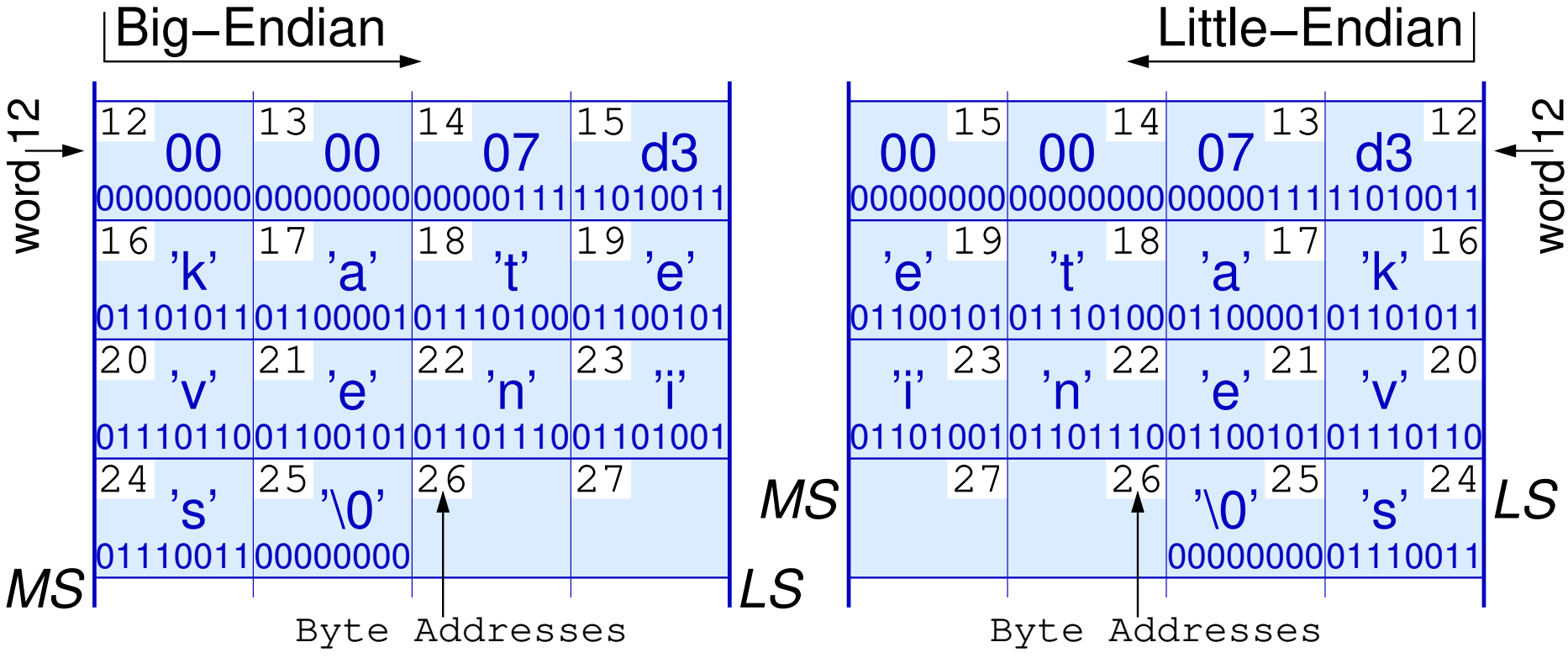
- Κάθε Byte έχει τη δική του Διεύθυνση
- Πίνακες κατά *αύξουσες* & συνεχόμενες διευθύνσεις, *πάντα*
  - π.χ. array of char @ base addr. `10`

Ποσότητες μεγαλύτερες από 1 Byte:

- Διεύθυνση = η διεύθυνση εκείνου από τα Bytes τους που έχει τη *μικρότερη* διεύθυνση, *πάντα*
- Τα Bytes μέσα στην ποσότητα: *πού;;*
  - π.χ. ακέραιος 2003 (δεκαεξαδικό 7d3)
  - πού πάνε τα Bytes 00, 00, 07, d3 ?



Αρίθμηση των Bytes μέσα σε μία Λέξη Ακεραίου;

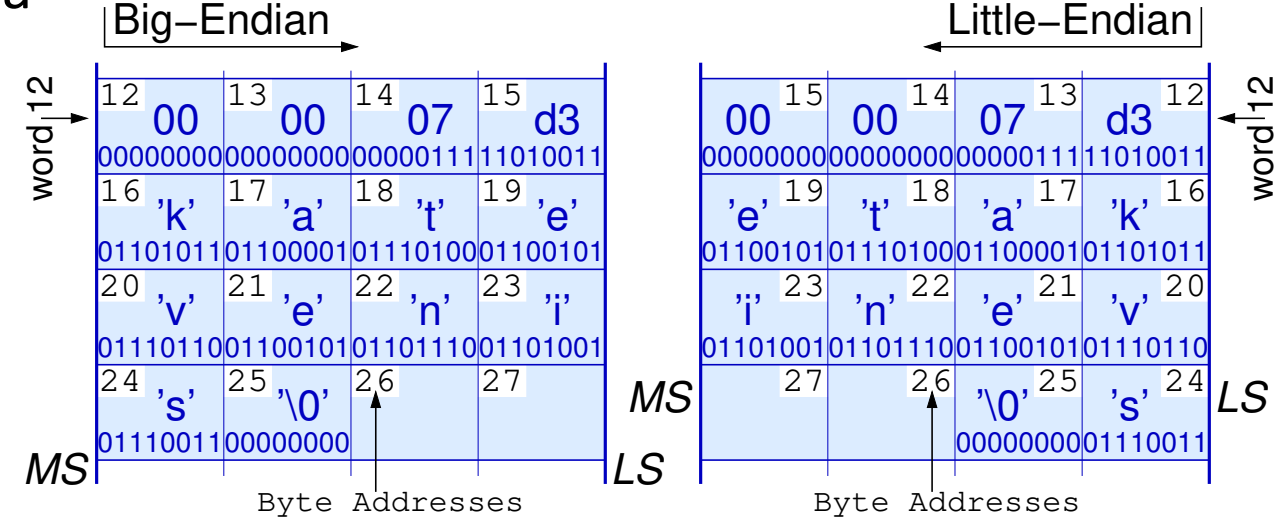


Δύο στρατόπεδα κατασκευαστών («Ιερός πόλεμος»)

Τι με νοιάζει εμένα; – (1) κανονικά προγράμματα

- char buf[10] με base addr. 16
- γράφω buf[0] = ‘k’
- γράφω buf[1] = ‘a’
- διαβάζω buf[0] ⟶ ‘k’
- διαβάζω buf[1] ⟶ ‘a’
- ίδια, σωστή συμπεριφορά σε ό,τι μηχανή και να τρέξω το πρόγραμμά μου

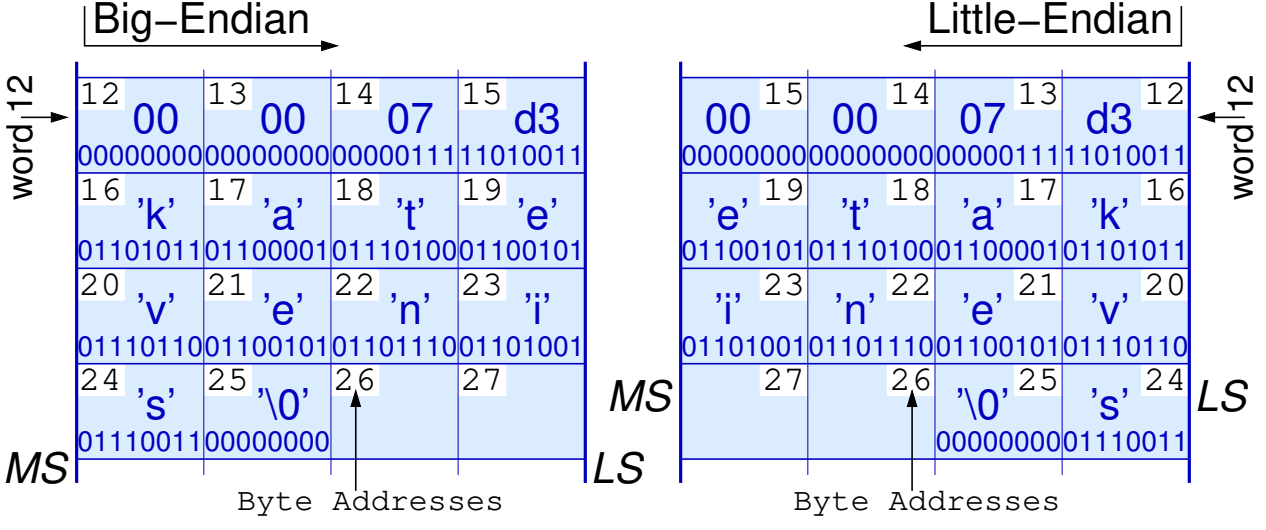
- int year=2003 στη διεύθυνση 12
- διαβάζω int από 12, βρίσκω 2003
- ίδια, σωστή συμπεριφορά σε ό,τι μηχανή και να τρέξω



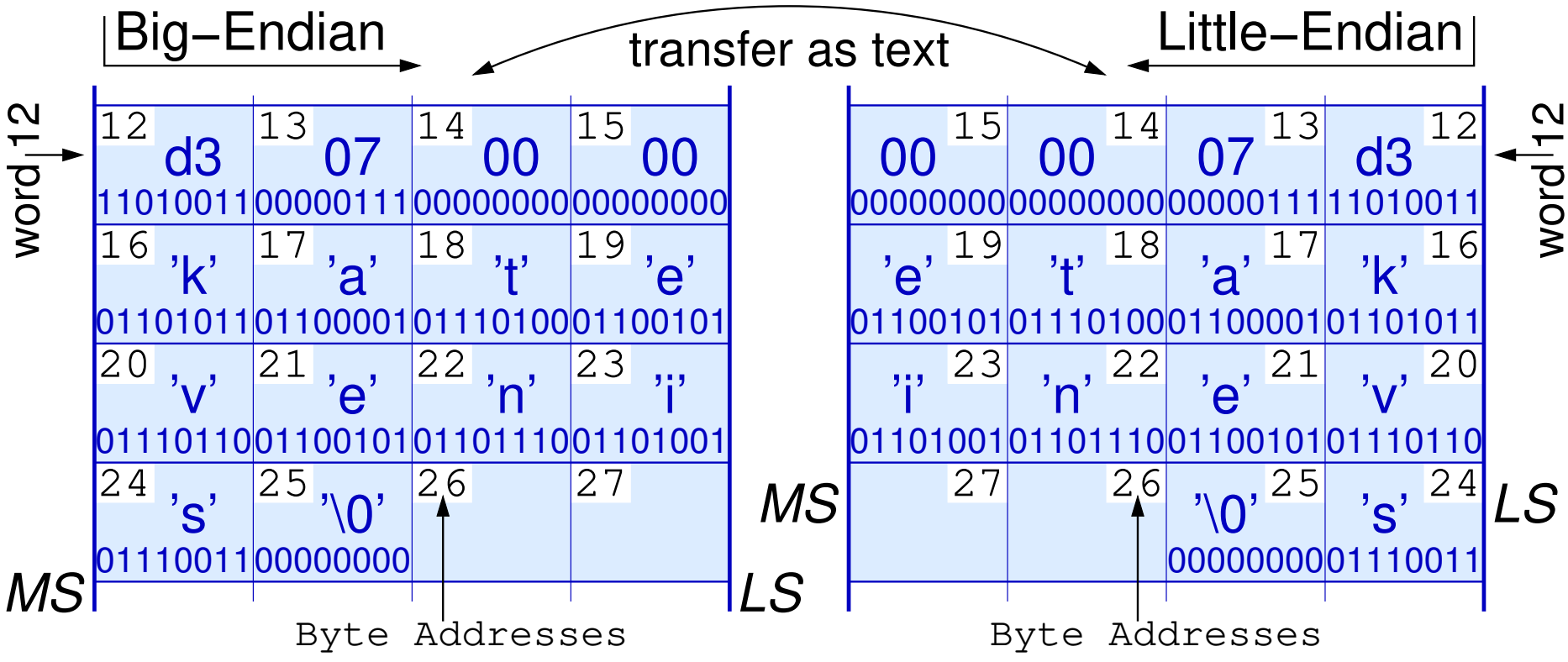
Τι με νοιάζει εμένα; – (2) «παράξενα» προγράμματα

- γράφω *string* “katevenis” ξεκινώντας από τη διευθ. 16
- διαβάζω *int* (!) από δ. 16:
- αλλού (big-e.) ⟶ 6b617465
- αλλού (little-e.) ⟶ 6574616b
- *non-portable* program !
- η συμπεριφορά αλλάζει από μηχανή σε μηχανή
- (και μάλλον *ανόητο* πρόγραμμα)

- γράφω *int* year=2003 στη δ. 12
- διαβάζω *char* (!) από διευθ. 12:
- αλλού (big-e.) βρίσκω: ‘\0’ (00)
- αλλού (little-e.) βρίσκω: ‘Σ’ (d3)

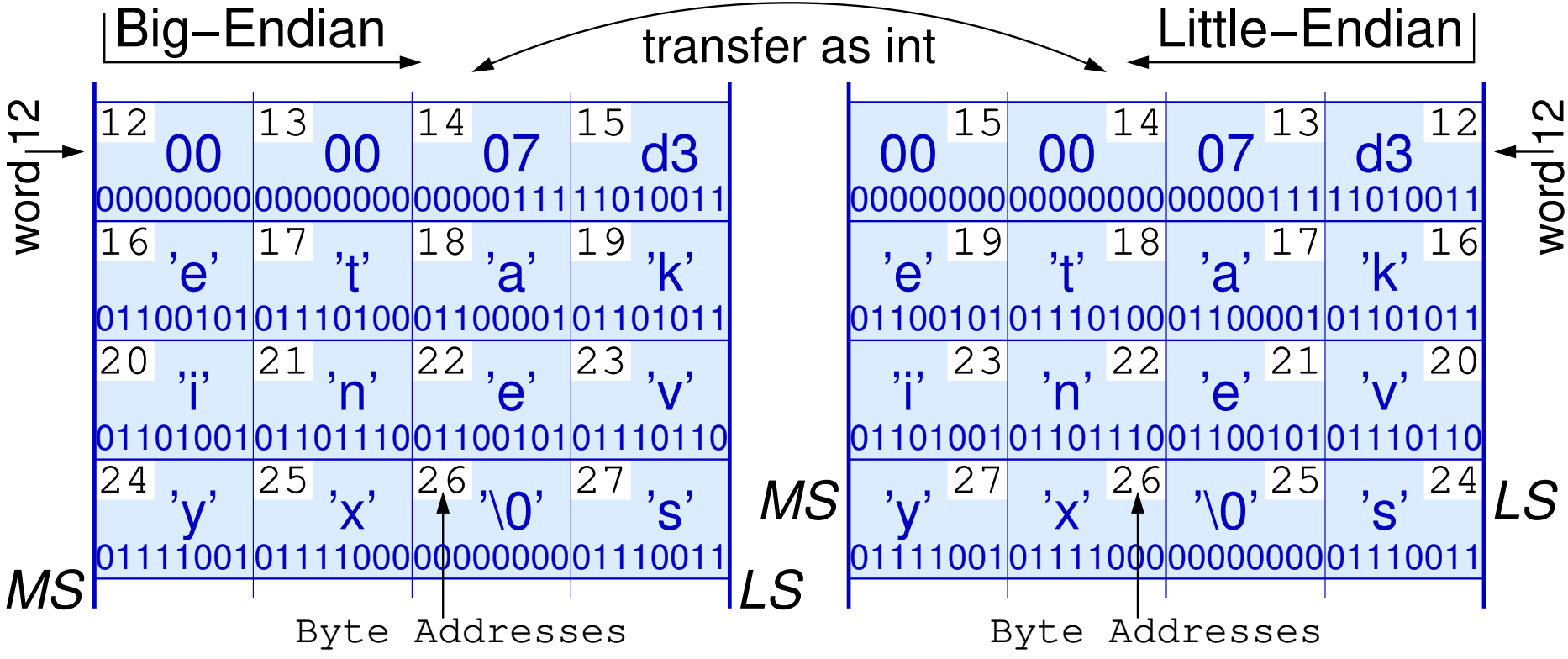


Τι με νοιάζει εμένα; – (3) μεταφορά αρχείου ως text



Ο ακέραιος 2003 γίνεται -754.515.968 στην άλλη μηχανή!

Τι με νοιάζει εμένα; – (4) μτφ. αρχείου ως binary (int)

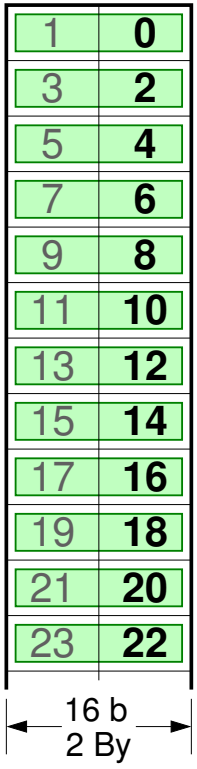


To string @16 γίνεται “etakinevyx” στην άλλη μηχανή!

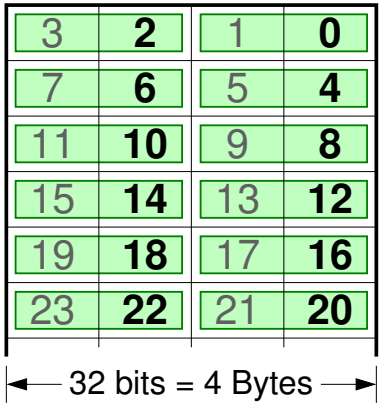
Προσπελάσεις σε Φαρδιές Μνήμες: Ευθυγράμμιση

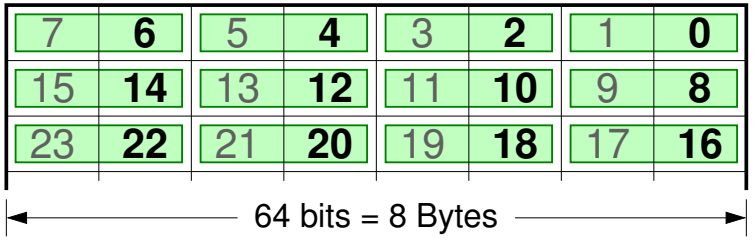
- Οι πραγματικές μνήμες συνήθως φαρδιές (32, 64,… bits)
  - γιά λόγους ταχύτητας: περισσότερα Bytes σε κάθε πρόσβαση
  - εάν ο επεξεργαστής χρειάζεται λιγότερα Bytes, η μνήμη του δίνει όλα τα Bytes στην «γραμμή» εκείνη, και ο επεξεργαστής, εσωτερικά, επιλέγει και κρατά εκείνα που θέλει
  - χωριστό σήμα *writeEnable* γιά κάθε Byte position: γιά να γράψουμε λιγότερα Bytes από μία ολόκληρη «γραμμή», ανάβουμε μόνον τα *wrEnab*'s εκεί που θέλουμε να γράψουμε
- Η πραγματική μνήμη προσπελάζει πάντα *ευθυγραμμισμένες* «γραμμές», στην κάθε πρόσβαση
- *Πόσες προσπελάσεις* χρειάζονται γιά ποσότητα n Bytes?

2–Byte "Half Words" Aligned on Addresses that are integer multiples of 2



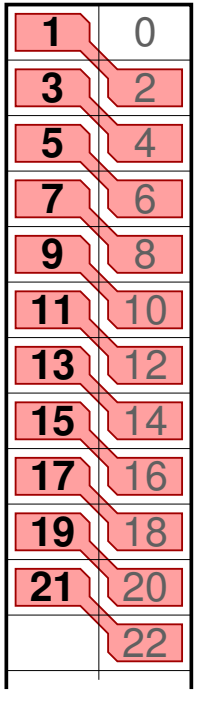
Addresses drawn assuming: *Little Endian layout*



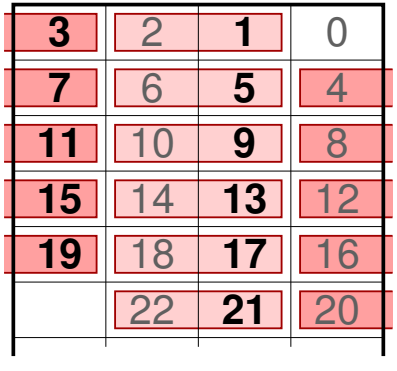


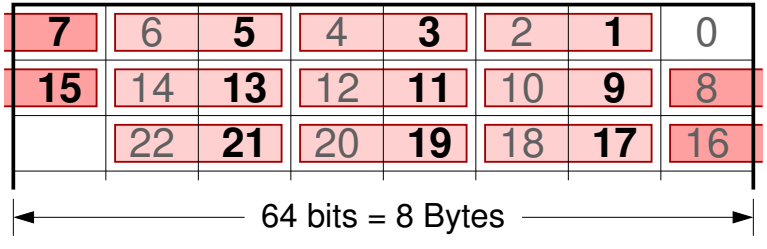
- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 2–Byte "half word" is shown in Bold

(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)



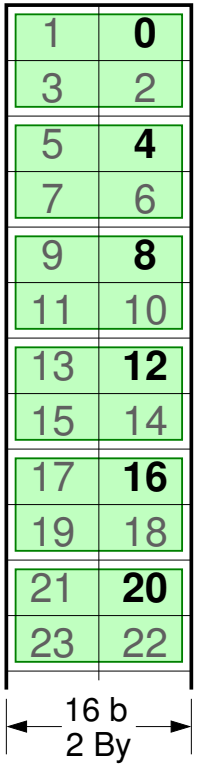
2–Byte "Half Words" at Addresses that are NOT integer multiples of 2



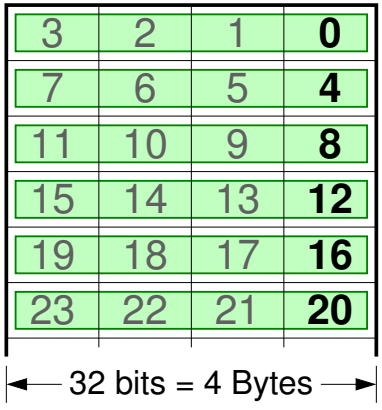


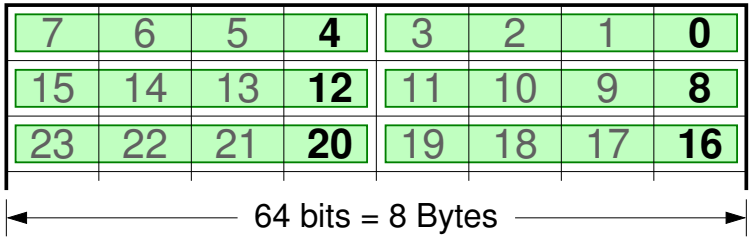
- Some (even if not all) of these 2–Byte half–w. incur a performance penalty when accessed

4–Byte "Words" Aligned on Addresses that are integer multiples of 4



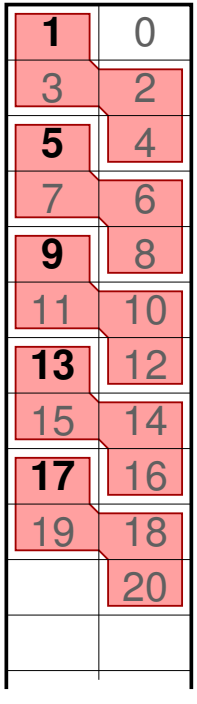
Addresses drawn assuming: *Little Endian layout*



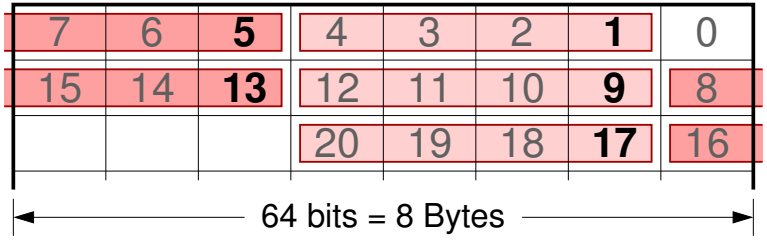


- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 4–Byte "word" is shown in Bold

(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)

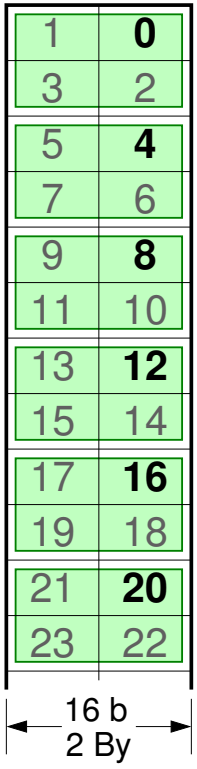


4–Byte "Words" at Addresses that are NOT multiples of 4, but are 1–off, i.e. Addr mod 4 == 1

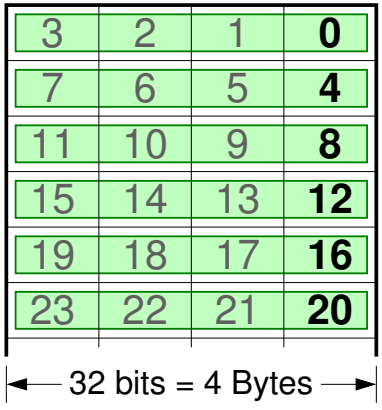


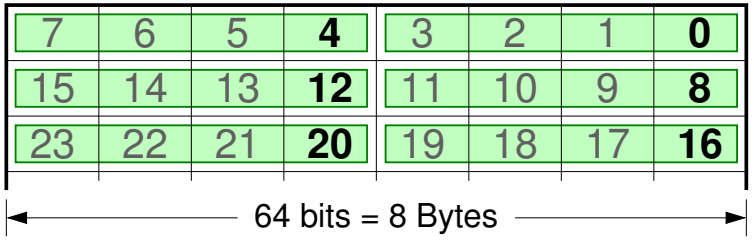
- Some (even if not all) of these 4–Byte words incur a performance penalty when accessed

4–Byte "Words" Aligned on Addresses that are integer multiples of 4



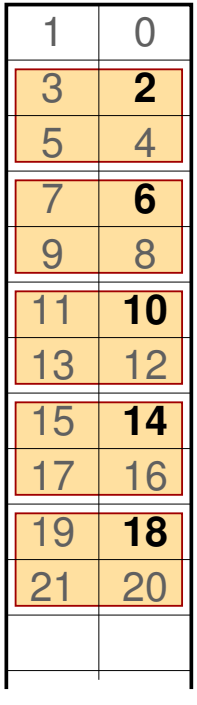
Addresses drawn assuming: *Little Endian layout*





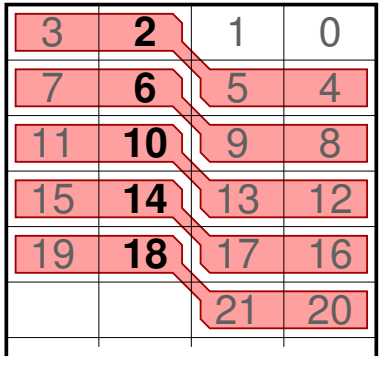
- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 4–Byte "word" is shown in Bold

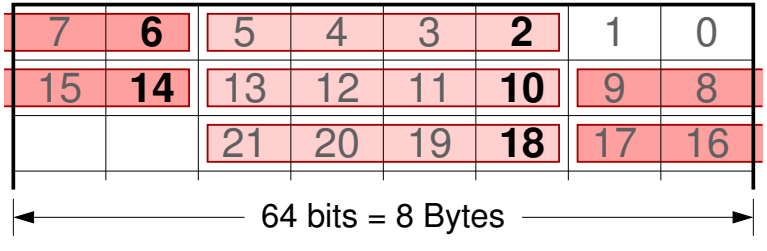
(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)



4–Byte words at multiples of 2 but not of 4 are OK in 2–Byte wide memories, but NOT in wider!

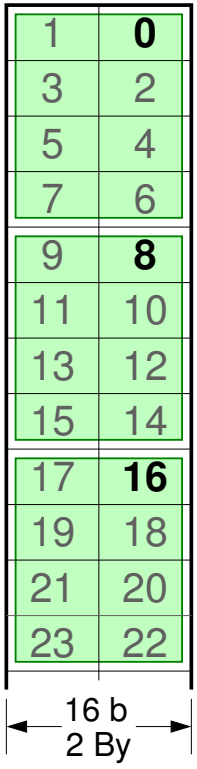
4–Byte "Words" at Addresses that are multiples of 2, but NOT multiples of 4 (i.e. Addr mod 4 == 2)



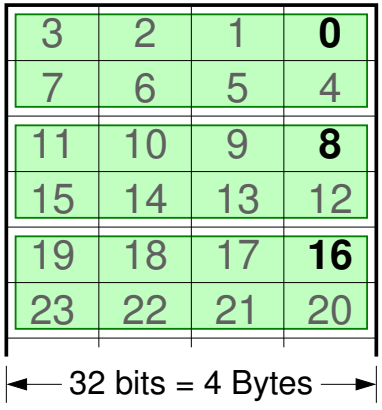


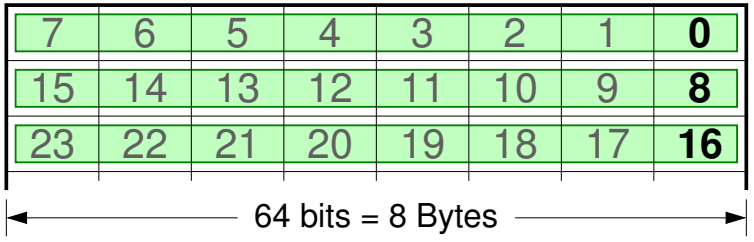
- Some (even if not all) of these 4–Byte words incur a performance penalty when accessed

8–Byte "Double Words" Aligned on Addresses that are integer multiples of 8



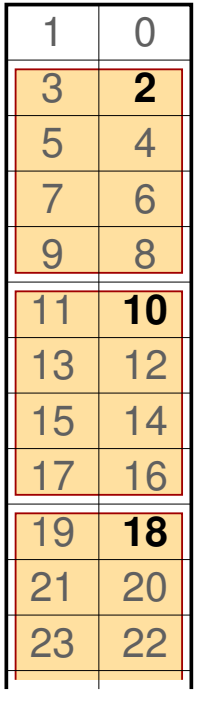
Addresses drawn assuming: *Little Endian layout*





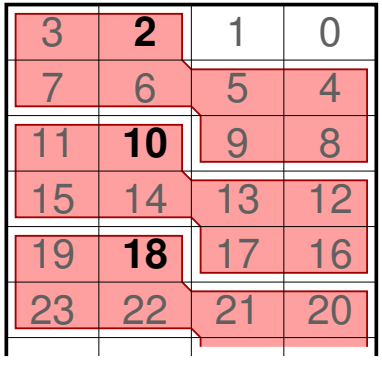
- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 8–Byte "double word" is shown in Bold

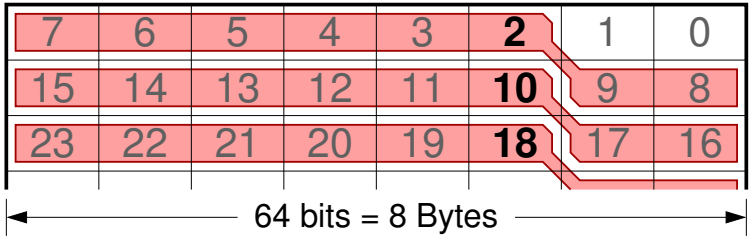
(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)



8–Byte doubles at multiples of 2 but not of 4 or 8 are OK in 2–Byte wide memories, but NOT in wider!

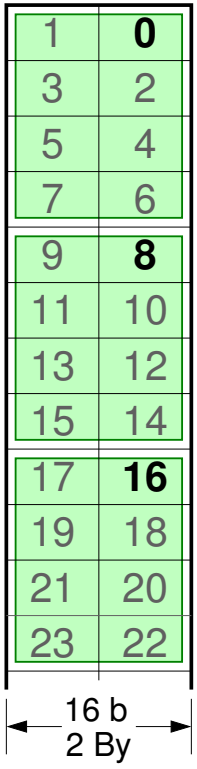
8–Byte "Doubles" at Addresses that are multiples of 2, but NOT multiples of 4 or 8 (i.e. Addr mod 8 == 2)



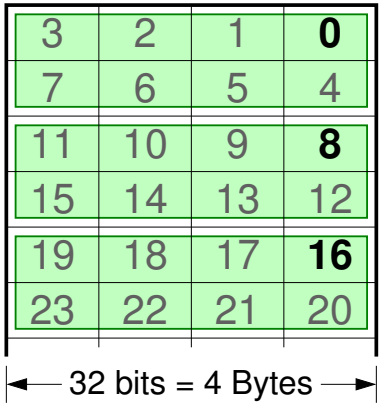


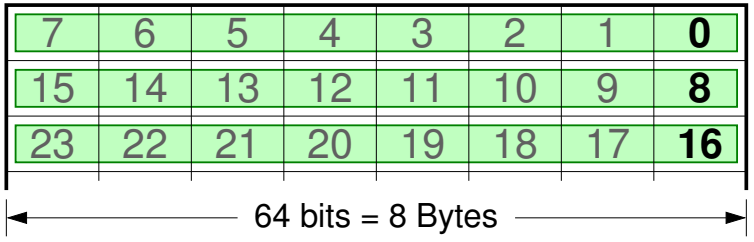
- In 4– and 8–wide memories, all of these doubles incur a performance penalty when accessed

8–Byte "Double Words" Aligned on Addresses that are integer multiples of 8



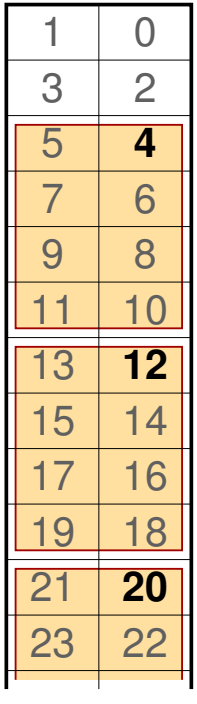
Addresses drawn assuming: *Little Endian layout*





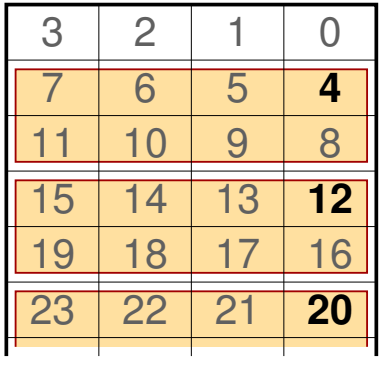
- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 8–Byte "Double word" is shown in Bold

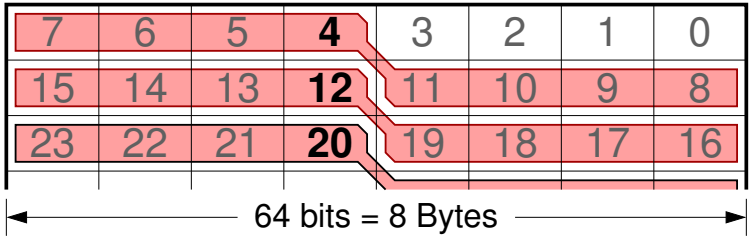
(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)



8–Byte Doubles at multiples of 4 but not of 8 are OK in 2– & 4– wide memories, but NOT in wider!

8–Byte "Doubles" at Addresses that are multiples of 4, but NOT multiples of 8 (i.e. Addr mod 8 == 4)





- In 8– Byte wide memories, these Doubles incur a performance penalty when accessed

Γιά ταχύτητα: Ευθυγράμμιση στα «φυσικά όρια»

Ηθικόν Δίδαγμα:

- Γιά τον *ελάχιστο* δυνατό αριθμό προσπελάσεων μνήμης,
- σε μνήμη *οιουδήποτε* πλάτους (πάντα δύναμη του 2)
  - δηλαδή γιά προγράμματα με "*portable* efficiency":
- Half-words (*2 Bytes*) σε διευθ. ακέραια πολλαπλ. του **2**
- Words (*4 Bytes*) σε διευθύνσεις ακέραια πολλαπλ. του **4**
- Double-words (*8 Bytes*) σε διευθ. ακέρ. πολλαπλ. του **8**
- Γενικά: *Ευθυγράμμιση* της κάθε ποσότητας (2^n By) στα «*φυσικά*» της όρια (ακερ. πολ. του 2^n)

Οδηγίες στον Assembler γιά Ευθυγράμμιση

- Σε μερικούς επεξ. υποχρεωτική η ευθυγράμ. (π.χ. MIPS)
- Σε άλλους *προαιρετική*: **RISC-V** (κ.α.)
  - παντού συμφέρει σε ταχύτητα, έστω και με κάποια κενά στη χρησιμοπ. του χώρου μνήμης (λιγότερα εάν βελτιστοποιηθούν)
- Οδηγίες Assembler να ευθυγραμμίσει το επόμενο item:

  `.align` **2** ⇒ σε ακέραιο πολλαπλ. του $2^{\mathbf{2}} = 4$

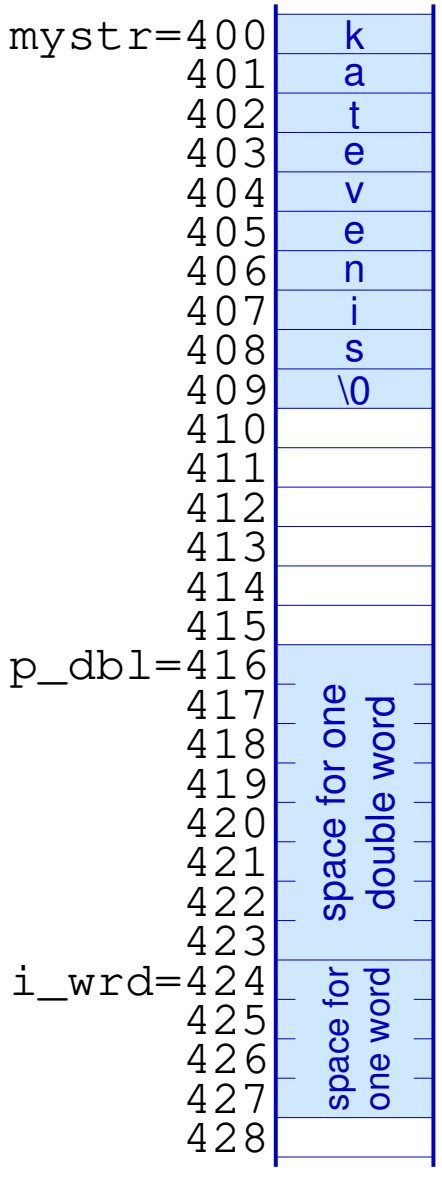
  `.align` **3** ⇒ σε ακέραιο πολλαπλ. του $2^{\mathbf{3}} = 8$

  κάνε τη διεύθ. όπου θα αρχίσει το επόμενο πράγμα στο data segment ακερ. πολ. του 2^n (αυξάνοντας την λίγο αν δεν είναι ήδη)

  `.space 32` ⇒ κράτησε εδώ χώρο 32 Bytes (άφησε κενό)

Παράδειγμα: .align . space

```
mystr:  .asciz "katevenis"
        .align 3  # πήδα στο επόμενο
                    ακέρ. πολ. του 2^3=8
p_dbl:  .space 8  # ονόμασε τη θέση
                    "p_dbl" και κράτα
                    χώρο 8 Bytes εκεί
i_wrd:  .space 4  # επόμ. θέση ονόματι
                    "i_wrd" και κράτα
                    χώρο 4 Bytes εκεί
```



Ευθυγραμμισμένες Ποσότητες & τα bits Διεύθυνσης

000001	000000
000011	000010
000101	000100

000011	000010	000001	000000
000111	000110	000101	000100
001011	001010	001001	001000

000111	000110	000101	000100	000011	000010	000001	000000
001111	001110	001101	001100	001011	001010	001001	001000
010111	010110	010101	010100	010011	010010	010001	010000

- Στις *ευθυγραμμισμένες* ποσότητες μεγέθους 2^n Bytes, τα n LS bits της διεύθυνσης του κάθε Byte τους δείχνουν τη θέση του Byte μέσα στην ποσότητα, και τα bits αυτά είναι όλα 0 στη διεύθυνση της ποσότητας, ενώ τα υπόλοιπα bits της διεύθ. είναι όλα ίδια σε όλα τα Bytes της ποσότητας
- Ομοίως: «Γραμμές» Κρυφών Μνημών, «Σελίδες» Εικονικής Μνήμης

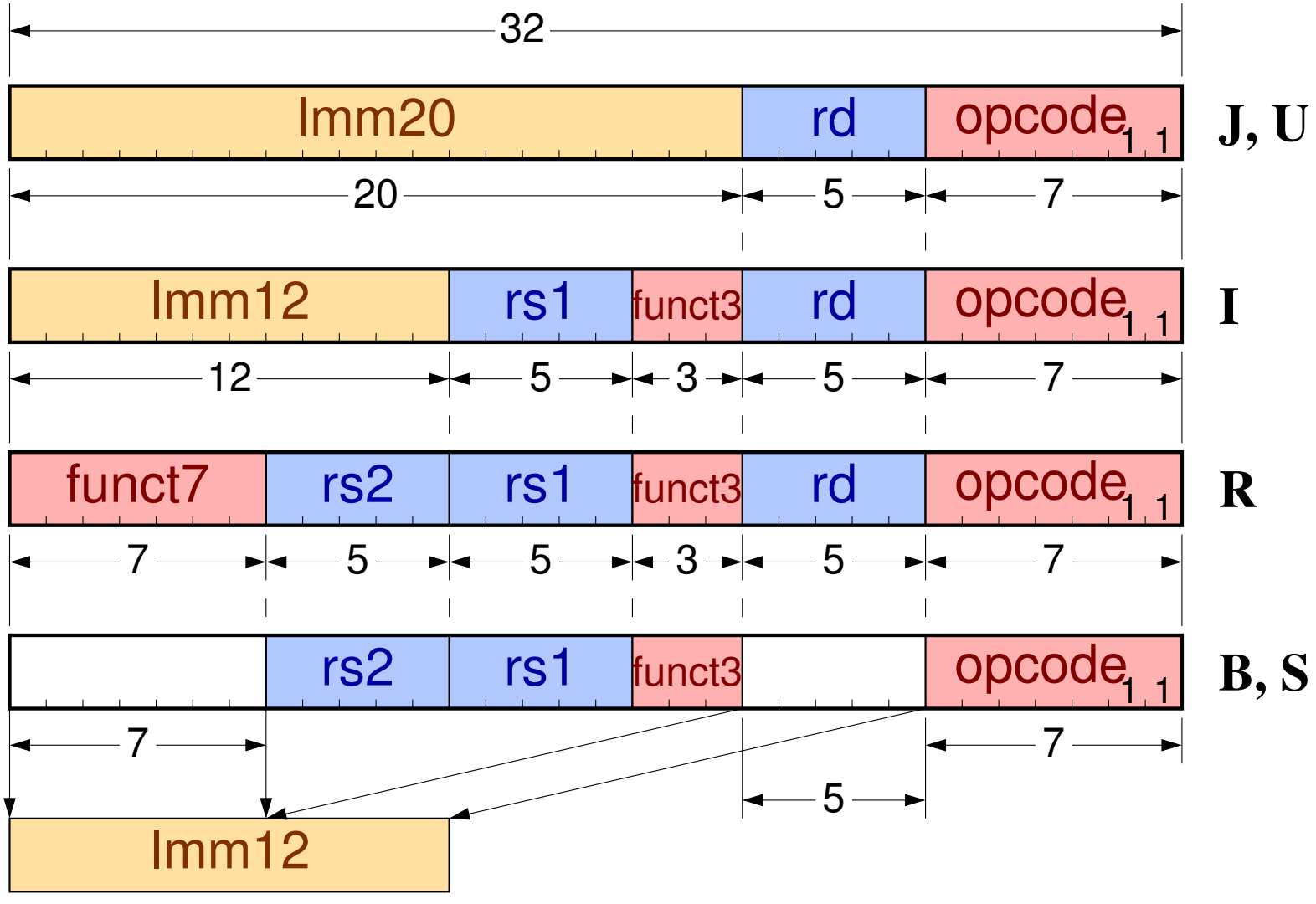
Τα Formats των εντολών του RISC-V

- Ο RV έχει *και* 16-μπιτες εντολές – *εμείς μόνον 32-μπιτες*

⇒ Διευθύνσεις εντολών *πάντοτε ακέραια πολλαπλ.* του **2**

- Όλα τα πεδία *καταχωρητών πάντοτε σε σταθερή θέση*
  - Καταχωρητές πηγής σε ίδια θέση γιά όλες τις εντολές: έτσι αρχίζει η ανάγνωσή τους *πριν* την αποκωδικοποίηση του opcode
  - Καταχωρητής προορισμού σε ίδια θέση γιά όλες τις εντολές: έτσι συγκρίνεται με τους καταχωρητές πηγής επόμενων εντολών, γιά ανίχνευση εξαρτήσεων, *πριν* την αποκωδ. opcodes
    - σε επεξεργαστές multiple-issue (superscalar), που διαβάζουν ≥ 2 εντολές, εξετάζουν πόσες ανεξάρτητες, τις εκτελούν εν παραλλήλω

Τα format Εντολών του RISC-V



Διακλαδώσεις RISC-V: PC-relative Addressing

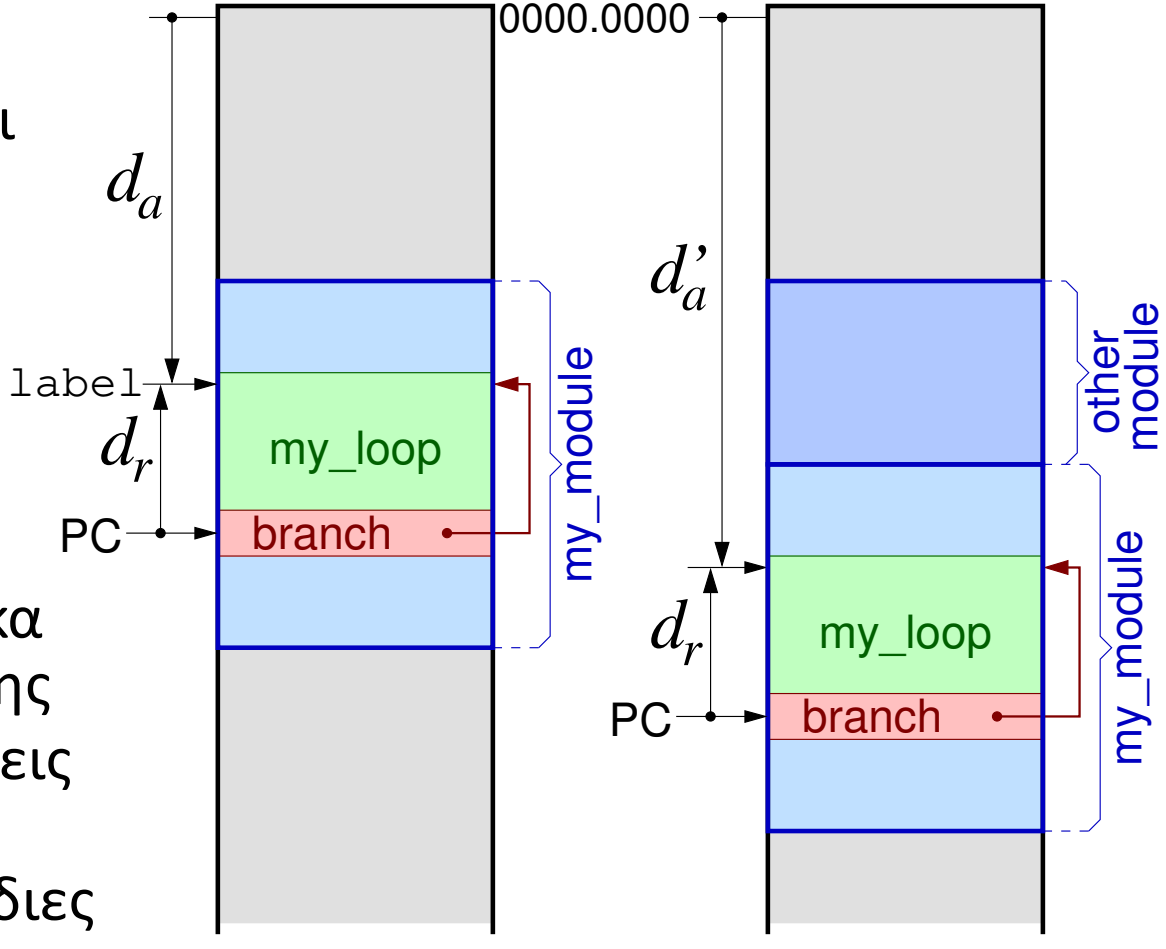
`beq rs1, rs2, Imm12` ⇒

if (rs1==rs2) {PC ← PC+2×Imm12} else {PC ← PC+4}

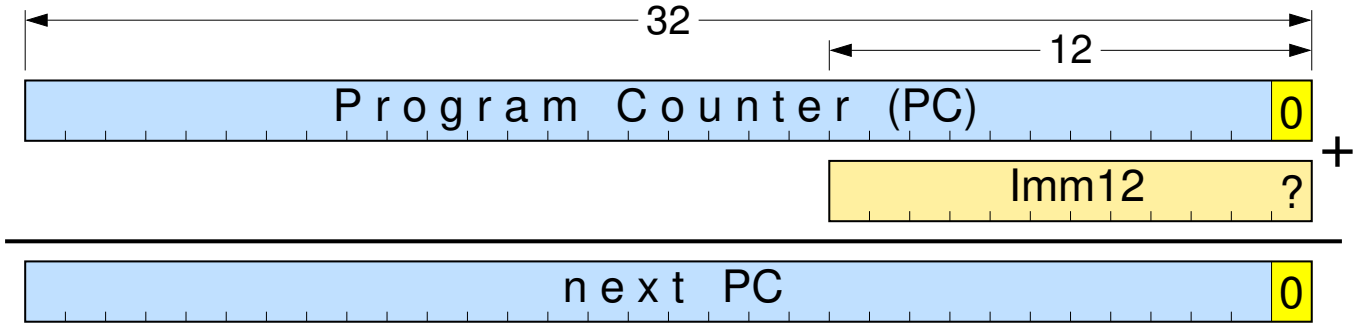
- Η σταθερά Imm12 (offset) πάντα προσημασμένη
- Θετικά offsets: συνήθως if-then-else
- Αρνητικά offsets: συνήθως βρόχοι

PC-relative Addressing Mode

- $d_r \ll d_a$
  - συνήθως βρόχοι και if-then-else είναι μικρά σε μέγεθος
  - συνήθως το text είναι σε σχετικά μεγάλες διευθυν.
- Relocation
  - φόρτωση του κώδικα σε άλλη θέση μνήμης
  - απόλυτες διευθύνσεις d_a αλλάζουν σε d_a'
  - σχετικές διευθ. d_r ίδιες

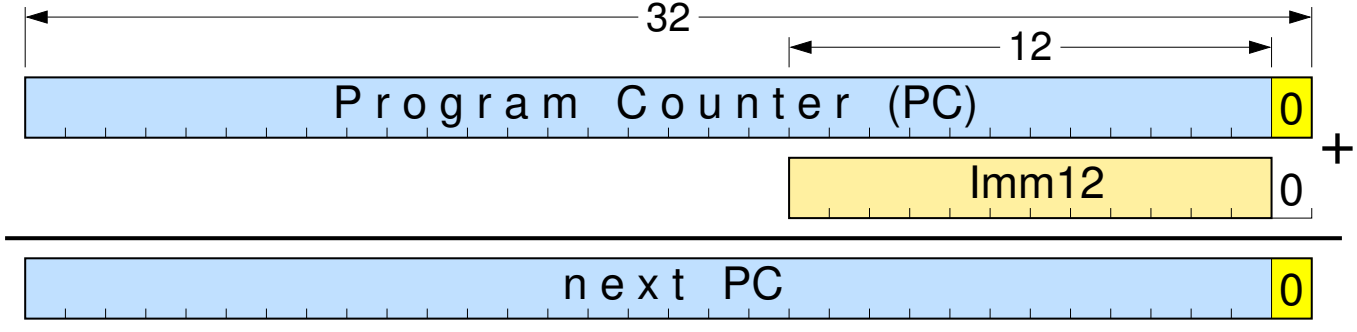


PC-relative addr.: Ευθυγράμμιση εντολών & Βεληνεκές



Απλοϊκή πρόσθεση ⇒ Βεληνεκές = ±2 KBytes

Ευθυγράμμιση εντ. RISC-V πάντα σε όρια half-words ⇒ δεξιό bit PC πάντα 0 ⇒ δεξιό bit σταθεράς υποχρεωτικά 0 ⟶ σπατάλη πληροφ.



Πρόσθεση διπλασίου ⇒ Βεληνεκές = ±2 KHalfWrd = ±4 KBytes

Το Imm12 μετρά σε μονάδες Half-Words

Άλμα με αποθ. Διεύθ. Επιστροφής: Κλήση Διαδικασίας

- Άλματα σε μεγαλύτερη απόσταση: Κλήση Διαδικασίας
- "Jump-and-Link": απλό άλμα + σωσ. Διεύθ. Επιστροφής:

`jal rd, Imm20` ⇒ $rd \leftarrow PC_{old} + 4$,

$PC_{new} \leftarrow PC_{old} + 2 \times Imm20$

- Η «από κάτω» εντολή στο $PC_{old}+4$ = Διεύθυνση Επιστροφής
- Η σταθερά Imm20 (offset) – πάντα *προσημασμένη*:
  - 20 bits: όσο χωρούσε στο J-format
  - επαρκής γιά κάλεσμα έως ±0.5 MHalfWords = ±1 Mbyte
  - υπερκαλύπτει και τις συνηθισμένες ανάγκες της «σκέτης» jump

Πού αποθηκεύουμε τη Διεύθυνση Επιστροφής;

- *CISC*: στη μνήμη, στη στοίβα (πιθανόν μαζί και με σωζόμενους καταχωρητές, «γιά υποστήριξη αναδρομής»)
- *RISC*: Διεύθυνση Επιστροφής *σε Καταχωρητή*:
  1. Μόνον οι εντολές store γράφουν στη μνήμη
  2. Ταχύτερο!
     - μόνον όσες διαδικασίες καλούν άλλην χρειαζ. στη στοίβα
     - μεγάλο ποσοστό των «δυναμικά» εκτελούμενων (≠ «στατικά» στο προγρ.) διαδικασιών δεν καλούν άλλη διαδικασία
  3. Σώσιμο άλλων μεταβλητών; ⟶ βλ. §6

Jump (Ψευδοεντολή): ειδική περίπτωση Κλήσης Διαδ.

- `x1` (αλλιώς: `ra`): συνήθης καταχ. γιά διευθ. επιστροφής
- Συνήθης κλήση διαδικασίας:

  `jal x1, Imm20` ⇒ $x1 \leftarrow PC_{old} + 4$,

  $PC_{new} \leftarrow PC_{old} + 2 \times Imm20$

- Ψευδοεντολή: **`j Imm20`** =

  `jal x0, Imm20` ⇒ $PC_{new} \leftarrow PC_{old} + 2 \times Imm20$

- Εγγραφή στον x0 αγνοείται: «Δεν με ενδιαφέρει η διεύθυνση επιστροφής – μην την κρατήσεις»

Άλματα σε μεταβλητές Διευθύνσεις: *Jump Register*

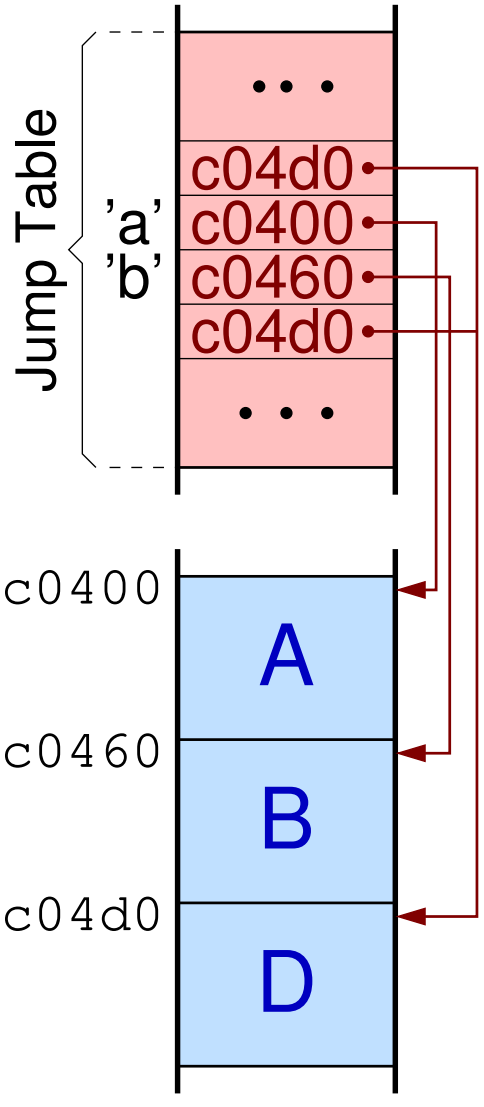
`jr rs1` ⇒ PC ← rs1

- Γιά Επιστροφή από Διαδικασία: `jr x1` ή `jr ra`
  - Μεταβλητή διεύθυνση διότι με καλούν από διάφορα μέρη
- Switch statement – multi-way "computed" branch
- Γιά άλμα σε αυθαίρετη διεύθυνση, οσοδήποτε μακριά
  - σύνθεση αυθαίρετης σταθ. σε καταχ. από προηγ. εντ. – βλ. παρακ.
- Περισσεύουν πολλά bits μέσα στο format της εντολής:
  - *Ψευδοεντολή*, ειδική περίπτωση άλλης γενικότερης…

Switch statement μέσω `jr`

```
char in_c;
switch (in_c) {
    case ’a’:  { A; }
    case ’b’:  { B; }
    default:   { D; }
}
```

```
x5 ← JumpTable[in_c]
jr  x5
```



jr: Ειδική περίπτωση της *Jump-and-Link-Register*

- Μιάς και περισσεύουν bits στην `jr`, γενικεύουμε σε άλλη εντολή:

  `jalr rd, Imm12(rs1)` ⇒

  $rd \leftarrow PC_{old} + 4$; $PC_{new} \leftarrow Imm12 + rs1$

- Και Κλήση Διαδικασιών σε αυθαίρετη/μεταβλητή διεύθυνση – όχι μόνο άλματα σε τέτοιες διευθύνσεις
  - Object-Oriented: type-dependent procedure @variable address
- I-format: διαθέσιμη και η σταθερά Imm12, εάν χρειαστεί
  - Ίδιο addressing mode με *load*: signed Imm12, όχι διπλασιασμός
  - Μαζί με μιά προηγ. εντολή Upper Imm20 συνθέτει αυθαιρ. σταθ.

Γιά Relocatable: Add Upper Immediate to PC (`auipc`)

- Κατασκευάζει την ίδια σταθερά όπως και η `lui`,
  προσθέτει τον PC σε αυτήν,
  και γράφει το αποτέλεσμα στον rd

- Κάλεσμα PC-relative σε αυθαίρετη απόστ.: **`auipc t0, HI`** (ή `HI+1`); **`jalr ra, LO(t0)`**
- Άλμα PC-relative σε αυθαίρετη απόσταση: **`auipc t0, HI`** (ή `HI+1`); **`jalr x0, LO(t0)`**
- Load PC-relative σε αυθαίρετη απόσταση: **`auipc t0, HI`** (ή `HI+1`); **`ld rd, LO(t0)`**
- Store PC-relative σε αυθαίρετη απόσταση: **`auipc t0, HI`** (ή `HI+1`); **`sd rs2, LO(t0)`**

Διακλαδώσεις οσοδήποτε μακριά (εάν χρειαστεί)

- if (i ≠ j) goto *farAway*; /* σπάνιο, αλλά εάν χρειαστεί */
  continue…

```
      beq i, j, cnt          # "else" continue…
      auipc t0, Imm20        # upper 20 bits of offset
      jalr x0, Imm12(t0)     # low 12 bits of farAway
cnt:  continue…
```

Πράξεις Σύγκρισης με αποτέλεσμα Boolean

- `slt  rd, rs1, rs2` # rd ← (rs1<rs2) – sign'd
- `sltu  rd, rs1, rs2` # rd ← (rs1<rs2) – uns.
- `slti  rd, rs1, Imm12` # rd ← (rs1<Imm12) s.
- `sltiu rd, rs1, Imm12` # rd ← (rs1<Imm12) uns.
- "Set-if-less-than": πράξη ALU, όχι διακλάδωση
- Το αποτέλεσμα είναι Boolean:
  - True = 000…0001
  - False = 000…0000
- Γιατί μόνον αυτές; – βλ. άσκηση 5.9
  - Ρεπερτόριο εντολών: Δομικοί λίθοι γιά σύνθεση πιό πολύπλ.

Το `Imm12` είναι πάντα signed, ακόμα και στην `sltiu`

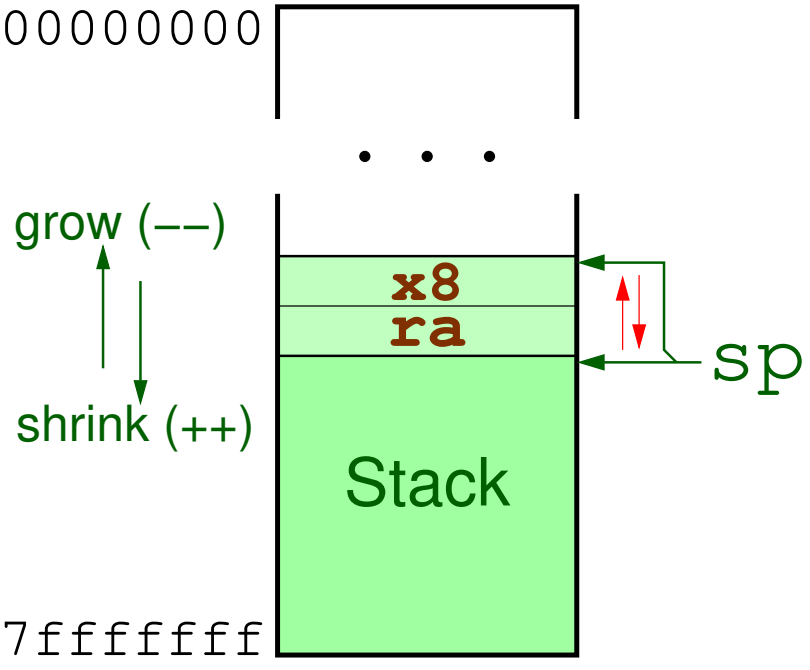
Σώσιμο Καταχωρητών στη Στοίβα και Επαναφορά

- Σώσιμο (save) των **ra**, **x8**:

```
addi sp, sp, -8
sw   ra, 4(sp)
sw   x8, 0(sp)
```

- Επαναφορά (restore):

```
lw   ra, 4(sp)
lw   x8, 0(sp)
addi sp, sp, +8
```



- Ο stack pointer (sp) πρέπει να δείχνει πάντα στο τελευταίο (minimum address) κατειλημμένο Byte στη στοίβα
  ⇒ στοίβα αυξάνει *πριν* το πρώτο save, μικραίνει *μετά* το τελευταίο

Χωριστή Μετάφραση: *δεν* ξέρω γονέα/παιδιά μου!

- Θέλουμε οι διαδικασίες σε χωριστά αρχεία

⇒ Όταν μεταφράζουμε (Compile) μιά διαδικασία, δεν ξέρουμε τη χρήση καταχωρητών του γονέα μου (της διαδικασίας που με κάλεσε), ούτε τη χρήση καταχωρητών των ενδεχομένων παιδιών μου (των διαδικασιών που εγώ ενδεχομένως θα καλέσω)

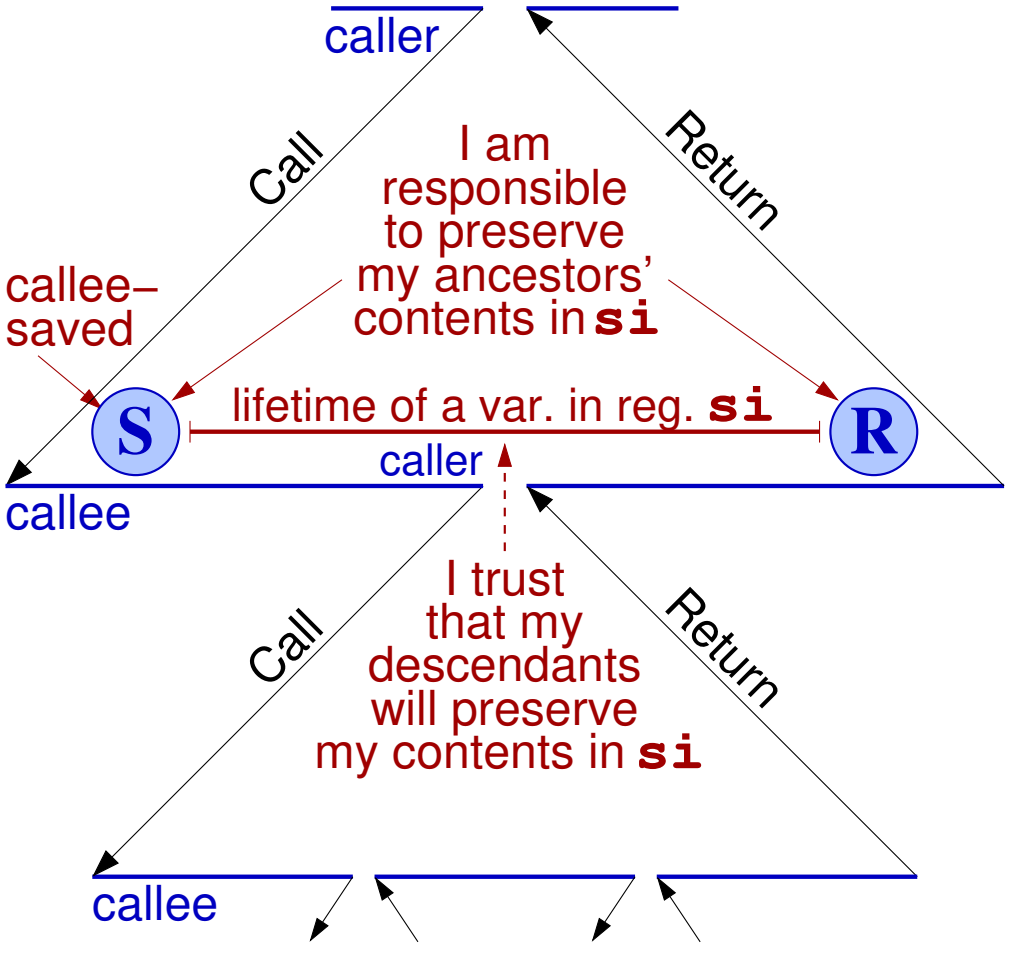
⇒ Ανάγκη Σύμβασης Κλήσης Διαδικασιών

- έχει οριστεί ενιαία, γιά όλους τους Compilers του Ρεπ. RISC-V
- εγγυάται την αρμονική συνένωση (linking) αρχείων δυαδικού κώδικα (π.χ. βιβλιοθηκών) παλαιών/νέων & ανά τον κόσμο

«*Διάρκεια Ζωής*» (*Lifetime*) Μεταβλητής/Καταχωρητή

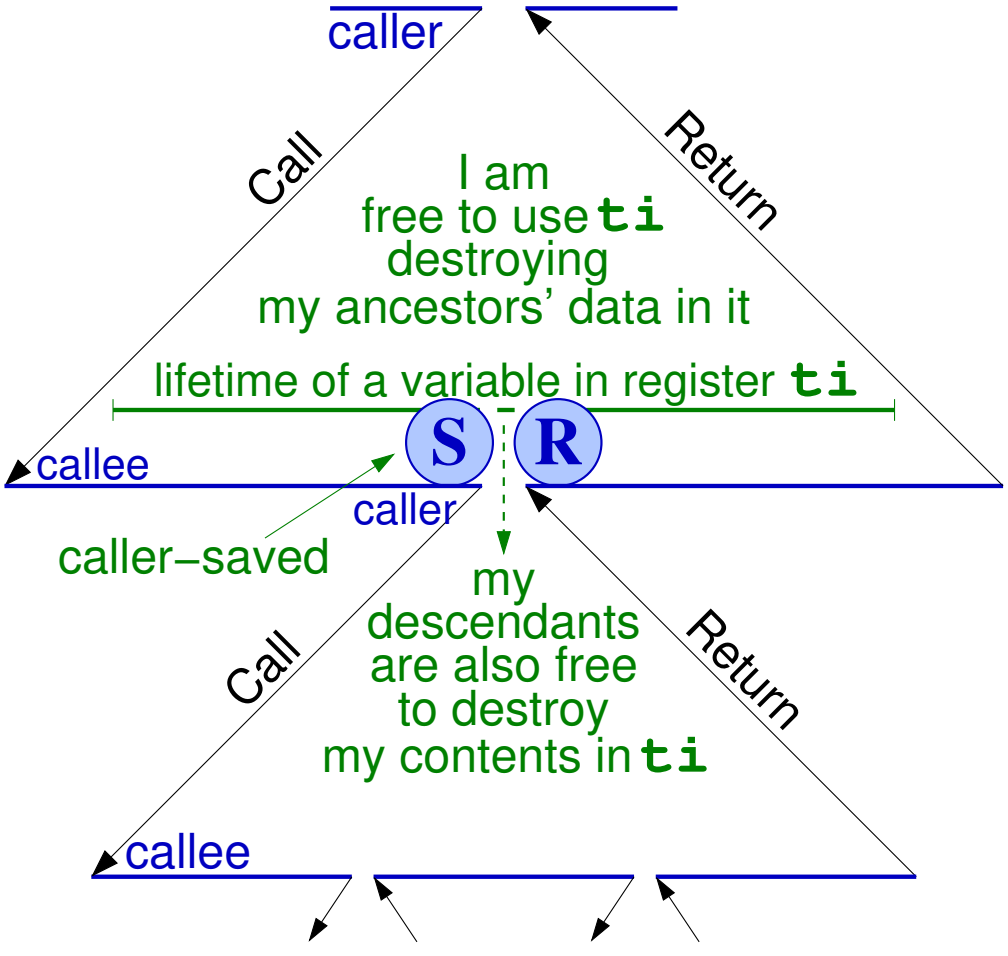
- Από την *εκχώρηση νέας* τιμής σε αυτήν/αυτόν
  - η οποία νέα τιμή *δεν* υπολογίζεται βάσει της παλαιάς του τιμής από την ίδια την εντολή που του εκχωρεί τη νέα τιμή
- Μέχρι την *τελευταία ανάγνωση* της τιμής του πριν την επόμενη εκχώρηση (ανεξάρτητης) νέας τιμής
  - εντολές που διαβάζουν και γράφουν τον ίδιο καταχωρητή (π.χ. addi t0, t0, 1) ούτε ξεκινούν ούτε τερματίζουν μιά «ζωή»
- Στη διάρκεια της ζωής «τον χρειαζόμαστε» (την τιμή του)
- Όταν «νεκρός», δεν τον χρειαζόμαστε (τιμή = σκουπίδια)

Το σώσιμο ευθύνη του καλουμένου (*Saved* registers)



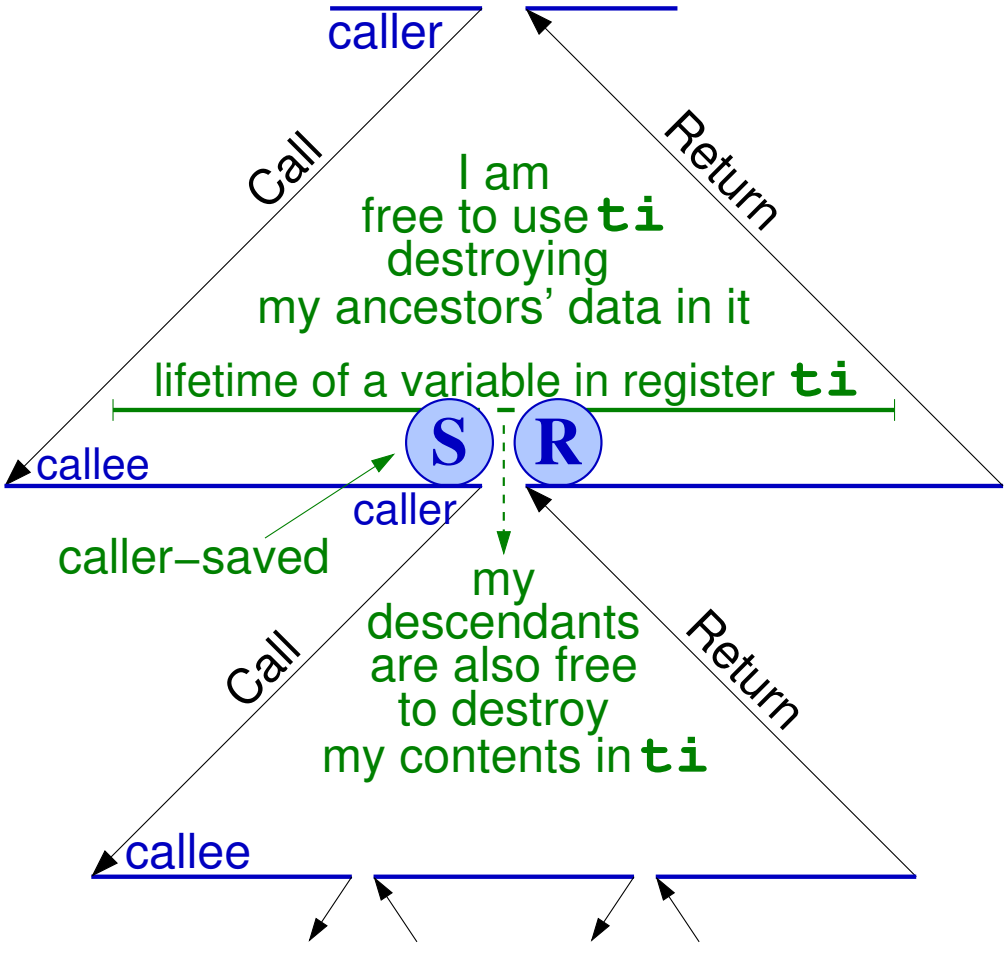
- Σύμβαση "Callee Saved"
- Ευθύνη καλουμένου να διατηρήσει ό,τι περιεχόμενα είχαν οι κατάχωρητές που αυτός θα χρησιμοποιήσει
- Όταν εγώ καλώ, ξέρω ότι οι απόγονοί μου θα διατηρήσουν τα δικά μου περιεχόμενα
- Register contents *saved* across procedure calls

Το σώσιμο ευθύνη του καλούντα (*Temporary* reg's)



- Σύμβαση "Caller Saved"
- Ευθύνη καλούντα να διατηρήσει ό,τι έχουν οι κατάχωρητές όταν αυτός καλεί παιδιά
- Εγώ είμαι ελεύθερος να χρησιμοποιήσω τέτοιους καταχωρητές
- Registers for *temporary* values, *not* preserved across procedure calls

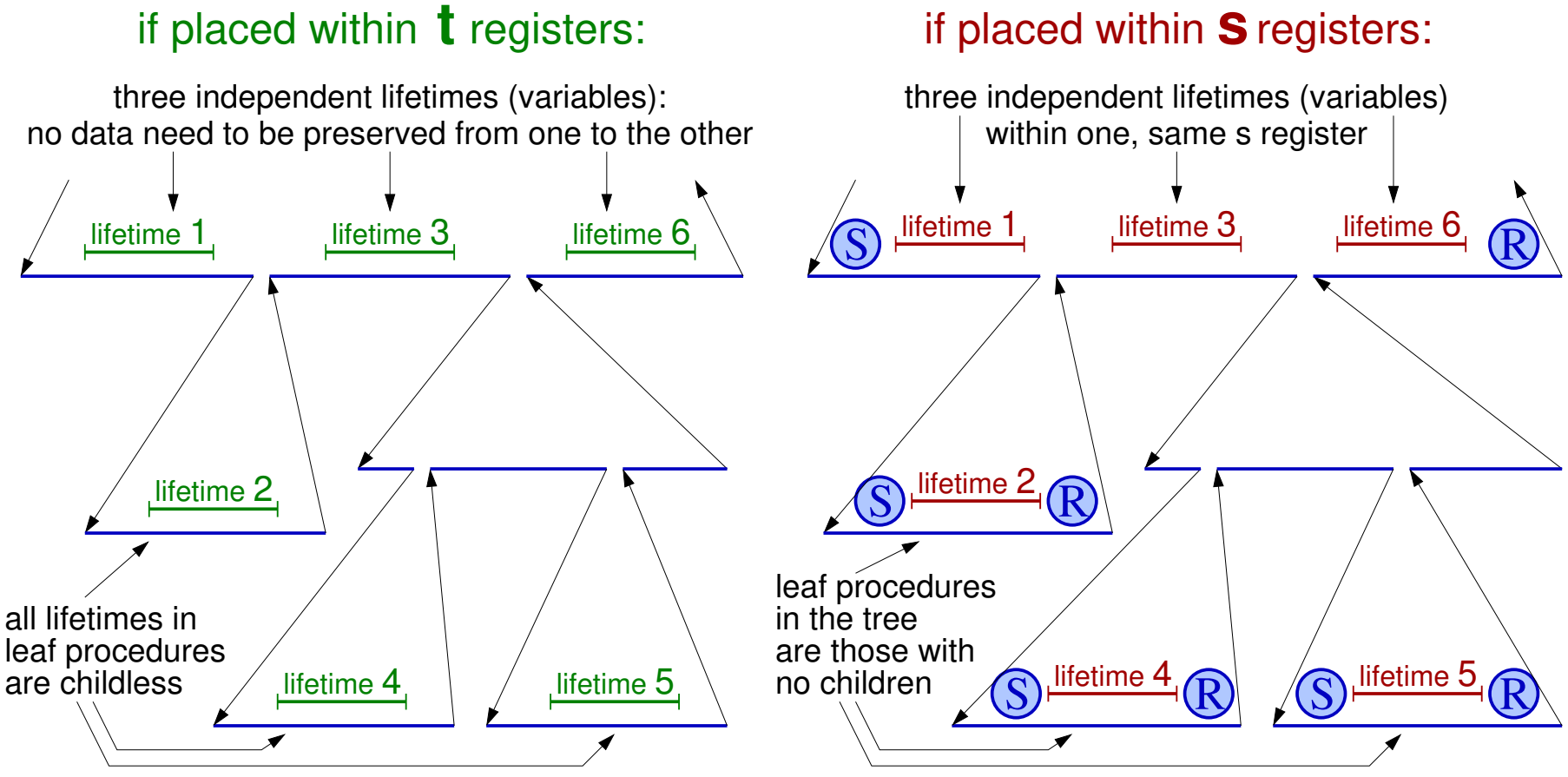
Temporaries συμφέρουν για ζωές χωρίς κλήσεις



Εάν δεν καλώ παιδιά ή όταν καλώ δεν έχω τίποτα χρήσιμο σε καταχωρ.:

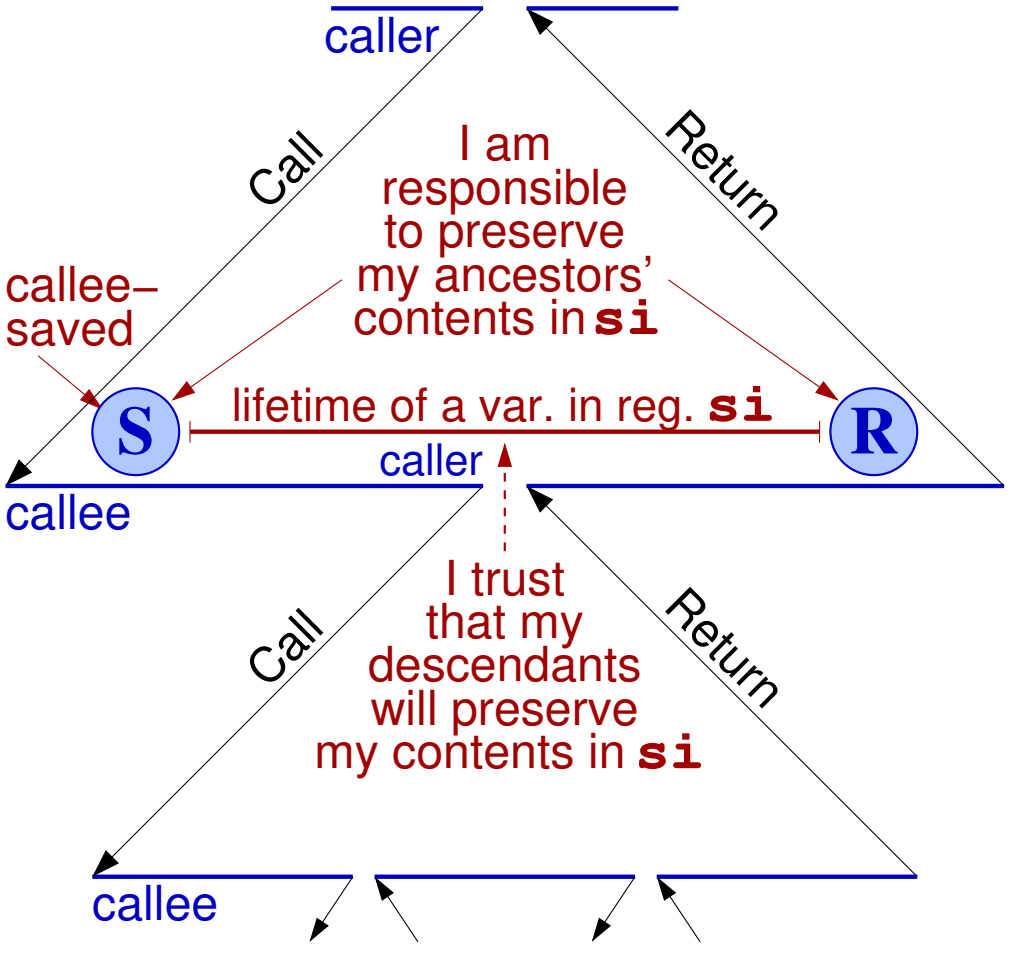
- Συμφέρει Caller Saved
- Ευθύνη καλούντα να διατηρήσει ό,τι έχουν οι κατάχωρητές όταν αυτός καλεί παιδιά
- Αλλά ο καλών ξέρει ότι δεν έχει τίποτα χρήσιμο όταν καλεί παιδιά

Lifetimes of variables that contain no procedure Calls



"t" registers are preferable for childless lifetimes: no save–restores to stack

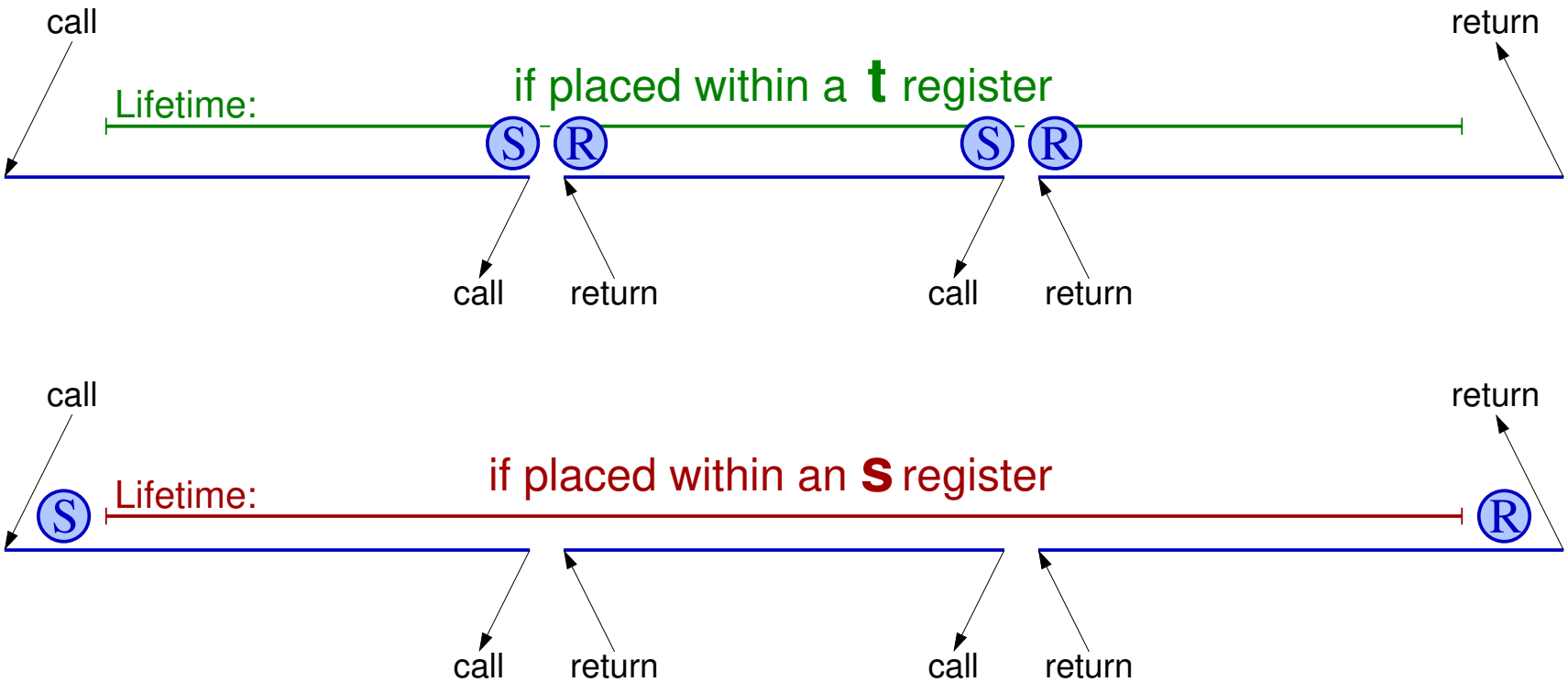
Όμως *Saved* συμφέρουν για ζωές με πολλές κλήσεις



Όταν όμως καλώ πολλά παιδιά ενόσω έχω κάτι χρήσιμο σε καταχωρητή:

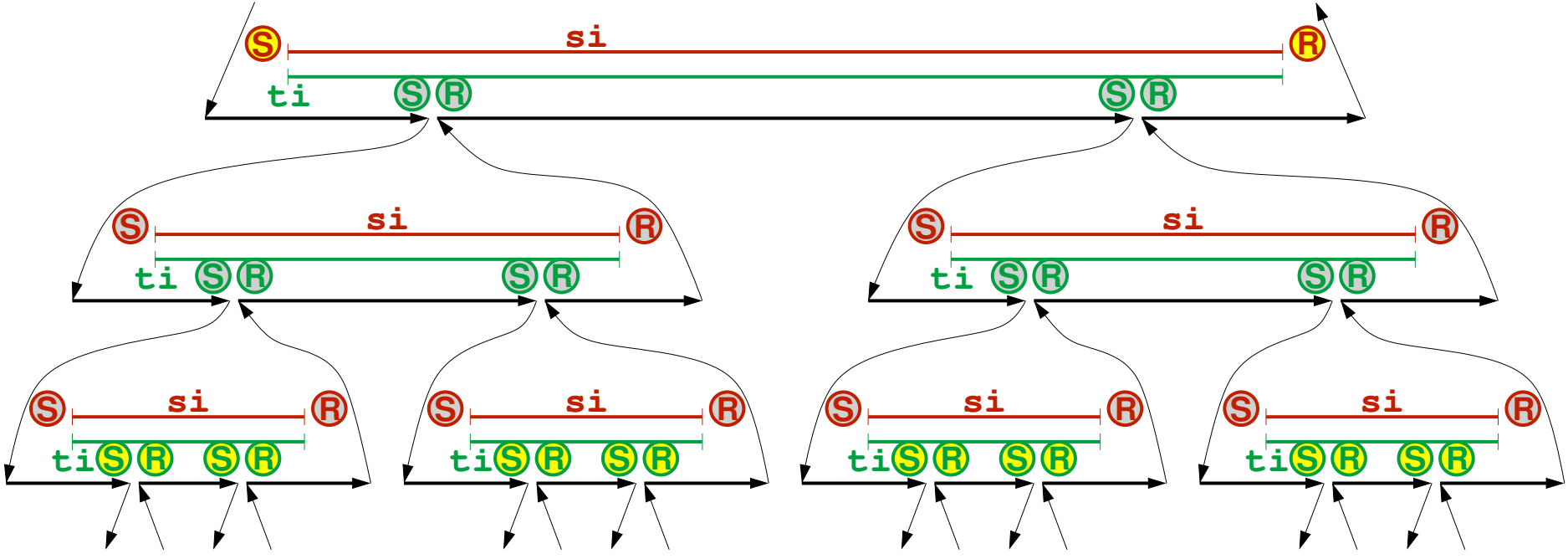
- Συμφέρει Callee Saved
- Ελπίζω οι απόγονοί μου να μην χρησιμοποιούν αυτόν τον καταχωρητή (εγώ δεν το ξέρω)
- Το ξέρουν όμως οι απόγονοί μου, και άρα τους μεταθέτω την ευθύνη για το σώσιμο

Lifetimes of variables that span 2 or more procedure Calls



"s" (saved) register is preferable: fewer save–restores to stack

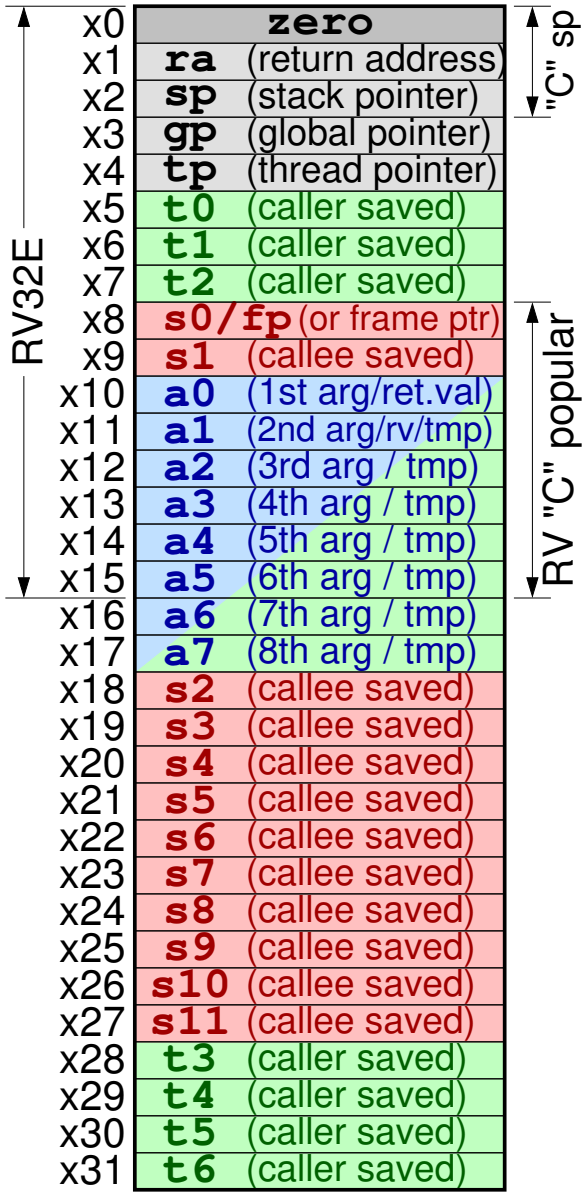
Τι διαφορά κάνει αν θα σώσει ο ένας ή ο άλλος;



- Γιά ό,τι δεδομένα πρέπει να διατηρηθούν «ζωντανά» διαμέσου κλήσεων, είναι το ίδιο αν θα τα σώσει ο πρόγονος (**ti**) ή ο απόγονος (**si**) στα ενδιάμεσα επίπεδα κλήσεων (γκρί S-R), *εκτός* στο πάνω-πάνω και στο κάτω-κάτω επίπεδο (κίτρινα S-R)
- Στο κατώτατο επίπεδο, δεν ξέρω εάν οι απόγονοί μου θα τον χρειαστούν τον καταχ., άρα αναβάλω το σώσιμο (μέσω **si**) ώστε εκείνοι να κάνουν ό,τι ξέρουν & συμφέρει

Δύο σύνολα reg's – ένα γιά φύλλα, άλλο γιά μακρόζωες

- Σαν να έχω χωρίσει τους καταχωρητές σε δύο υποσύνολα: Οι μεν **`ti`** γιά όλες τις διαδικασίες-φύλλα και τις ζωές χωρίς παιδιά, οι δε **`si`** στις «εσωτερικές» διαδικασίες του δένδρου γιά τις «μακρόβιες» μεταβλητές –εκείνες με πολλαπλές κλήσεις παιδιών στη διάρκεια της ζωής τους
- Εάν η κάθε διαδικασία που καλεί παιδιά καλεί κατά μέσον όρο π.χ. 10 παιδιά, είναι σαν το δένδρο καλεσμάτων να είναι 10-δικό δένδρο, οπότε το 90% των κόμβων του είναι φύλλα ⇒ no save/restore's στο 90% των περιπτώσεων!



Συμβάσεις Χρήσης Καταχωρητών

- Σύμβαση Κλήσης Διαδικασιών
  - μεταξύ Compilers – *δεν* αφορά το hardware, *δεν* την επιβάλει το hardware
- Τα Ορίσματα ίδια όπως Temporaries
  - Ορίσματα στους **a0**, **a1**, **a2**,…
  - Επιστρεφόμενη τιμή στον **a0** (& **a1** ?)
  - Όσοι καταχωρητές "**a**" (arguments) δεν απασχολούνται από ορίσματα είναι διαθέσιμοι γιά χρήση ως Temporaries
- Ο *RV32E* (*embedded*) έχει μόνον 16 καταχ.
- Ο *RVC* (*compressed*) συμπιέζει όσες εντολές χρησιμοποιούν μόνο τους `x0…2, x8…15`

```
long long int fact( long long int n )
{ if ( n<2 ) { return(1); } else { return( n * fact(n-1) ); } }
```

```
fact:  addi  t0, zero, 2   # immediate 2 needed for "if(n<2)"
       bge   a0, t0, elseF # if n<2 false, i.e. if n≥2 goto ELSE
       addi  a0, zero, 1   # THEN: create return-value 1, place in reg. a0
       jr    ra            # return --this is the end of the "then" clause
elseF: addi  sp, sp, -16   # PUSH1: allocate 16 Bytes on the stack
       sd    ra, 8(sp)     # PUSH2: save ra into first allocated word
       sd    a0, 0(sp)     # PUSH3: save my argument (n) into second word
       addi  a0, a0, -1    # create argument (n-1) into a0 for my child
       jal   ra, fact      # call my child procedure
       add   t0, a0, zero  # copy return value from my child into t0
                           # (because I need to restore my own argument into a0)
       ld    ra, 8(sp)     # POP1: restore ra from stack
       ld    a0, 0(sp)     # POP2: restore a0 from stack
       addi  sp, sp, 16    # POP3: dealloc the 16 B that I had allocated
       mul   a0, a0, t0    # multiply my own arg a0==n times the return
                           # value from my child that I had copied into t0, and
                           # place the result into a0, as my own return value
       jr    ra            # return
```