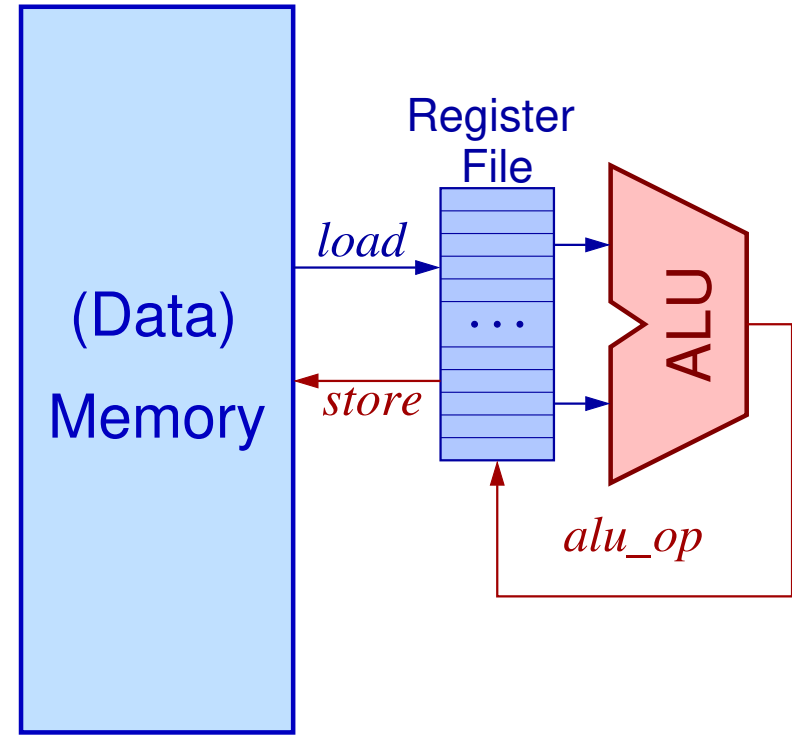
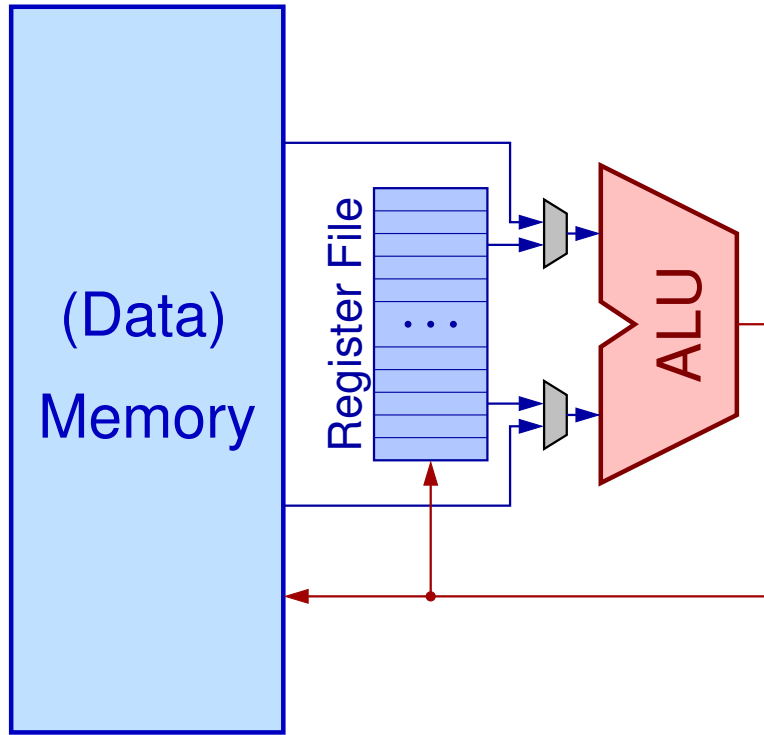


Επανάληψη 1:

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

Άνοιξη 2021 – Μανόλης Κατεβαίνης

Τελεστές: Οπουδήποτε ή σε Καταχωρητές μόνον;

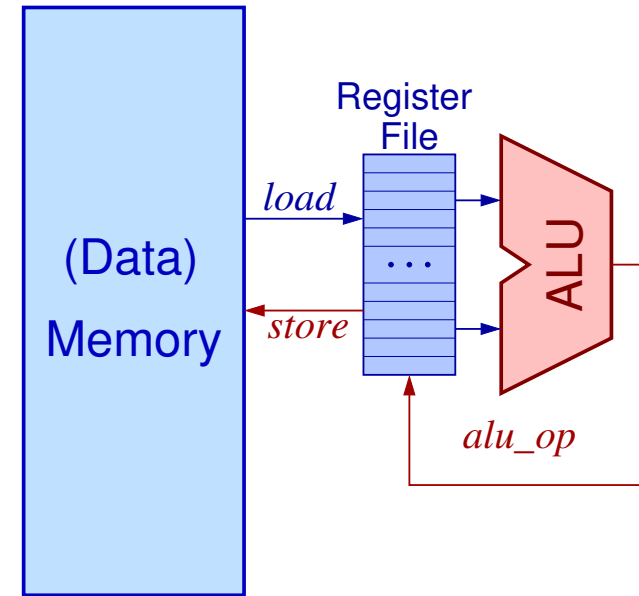


Οι τελεστές οπουδήποτε:
"CISC"
(Complex Instr. Set Computer)

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

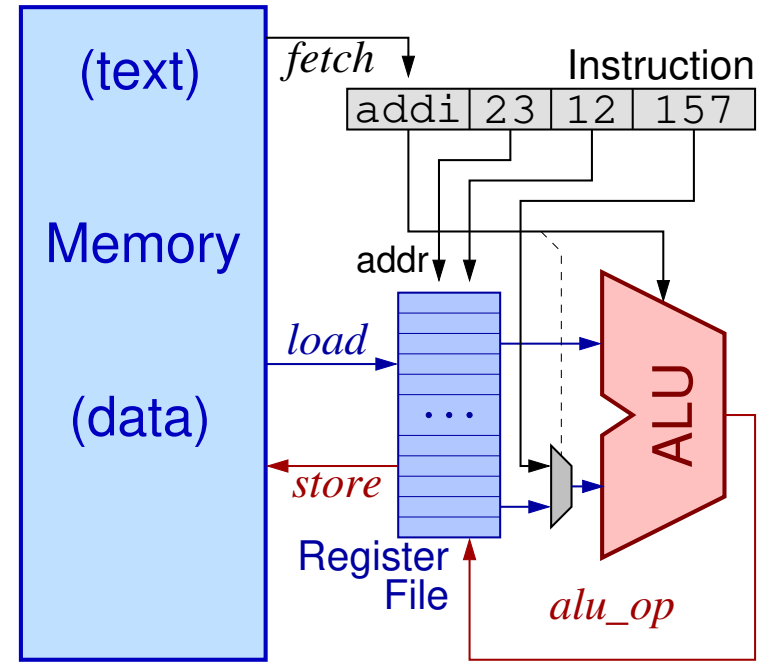
Εντολές reg2reg στον RISC-V

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



Σταθεροί αριθμοί στον RISC-V

- `addi x23, x12, 157`
 - “add immediate (constant)”
 - σημαίνει: $x23 \leftarrow x12 + 157$
 - `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`” \Leftrightarrow `addi x23, x12, -14`
- Δεν υπάρχει “`subii x23, 157, x14`”: πολύ σπάνια, δεν αξίζει



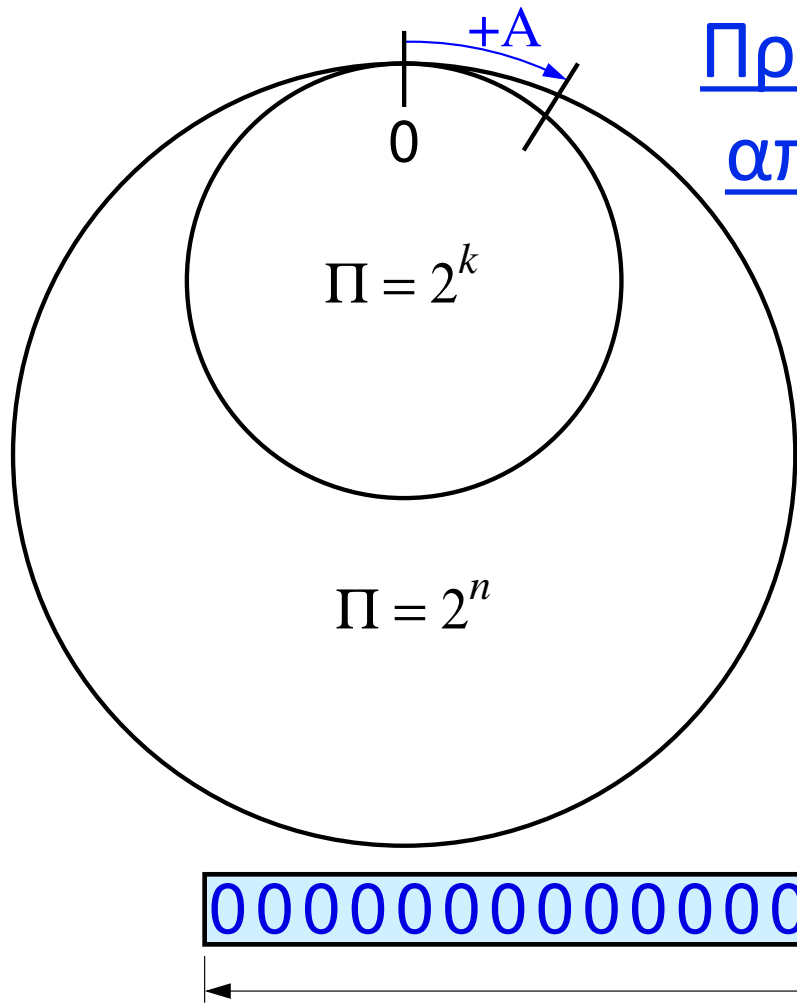
Σύνθεση άλλων τύπων εντολών σε RISC-style ISA's

- `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

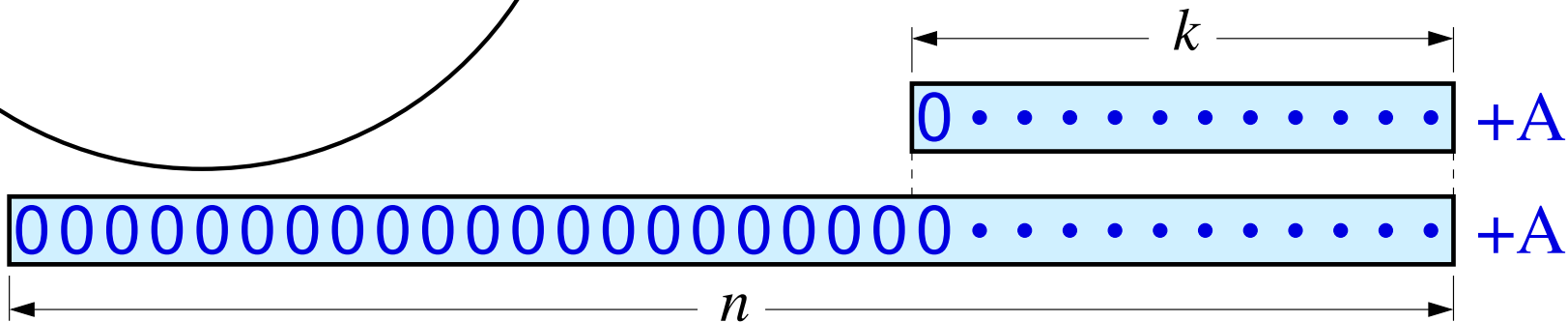
Μετατροπές Ακεραίων από λίγα σε πολλά bits

- Απρόσημοι (Unsigned) ακέραιοι (≥ 0) (HY-120 Εργ.5):
 - $b_{n-1} \times 2^{n-1} + \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$
 - 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 $\Rightarrow \geq 0$, ίδιος όπως εάν unsigned
 - Όταν MS bit == 1 $\Rightarrow < 0$, όσος εάν unsigned μείον 2^j
όπου j το πλήθος των bits του αριθμού
 - Όταν το j αλλάζει??

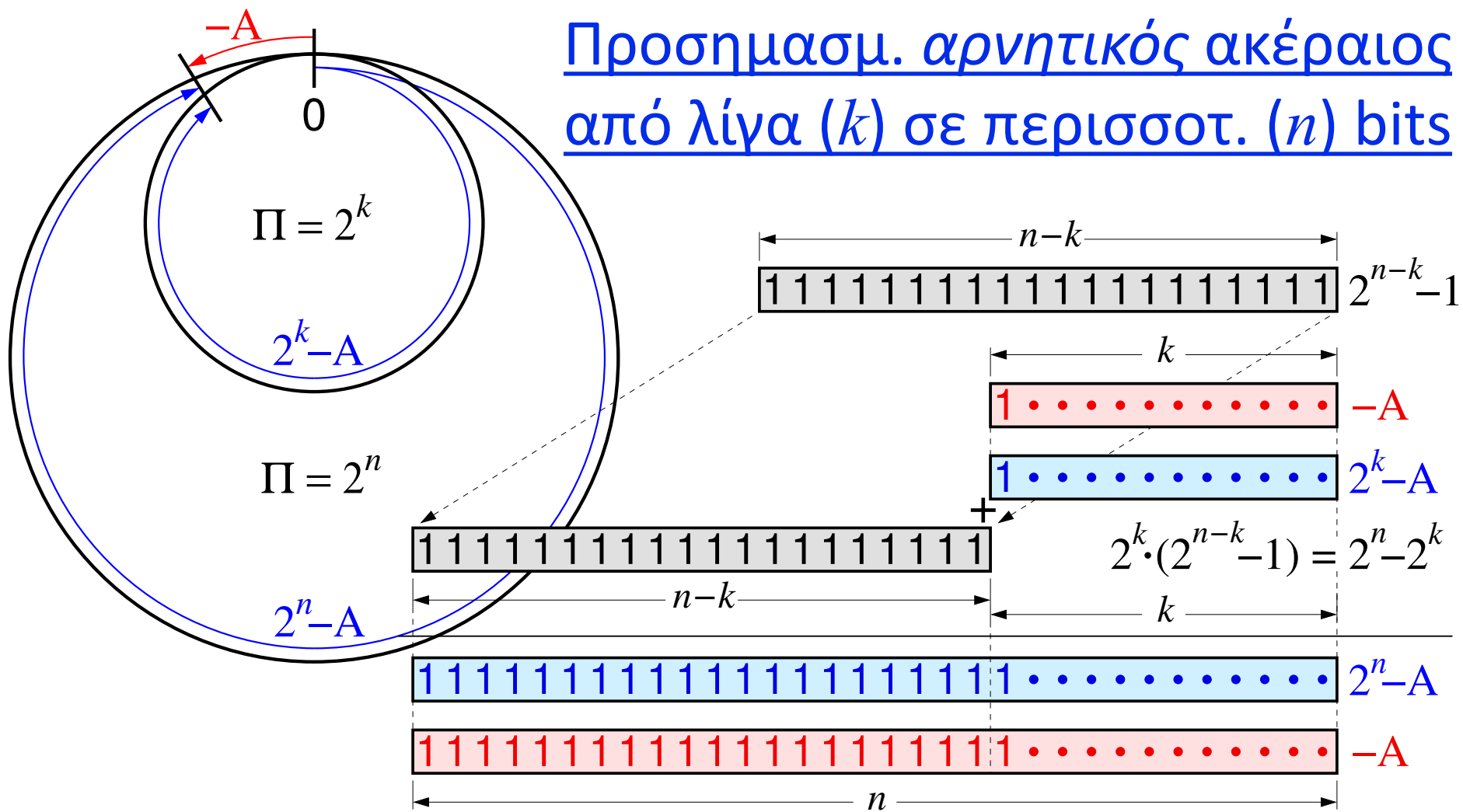
Προσημασμένος θετικός ακέραιος από λίγα (k) σε περισσοτ. (n) bits



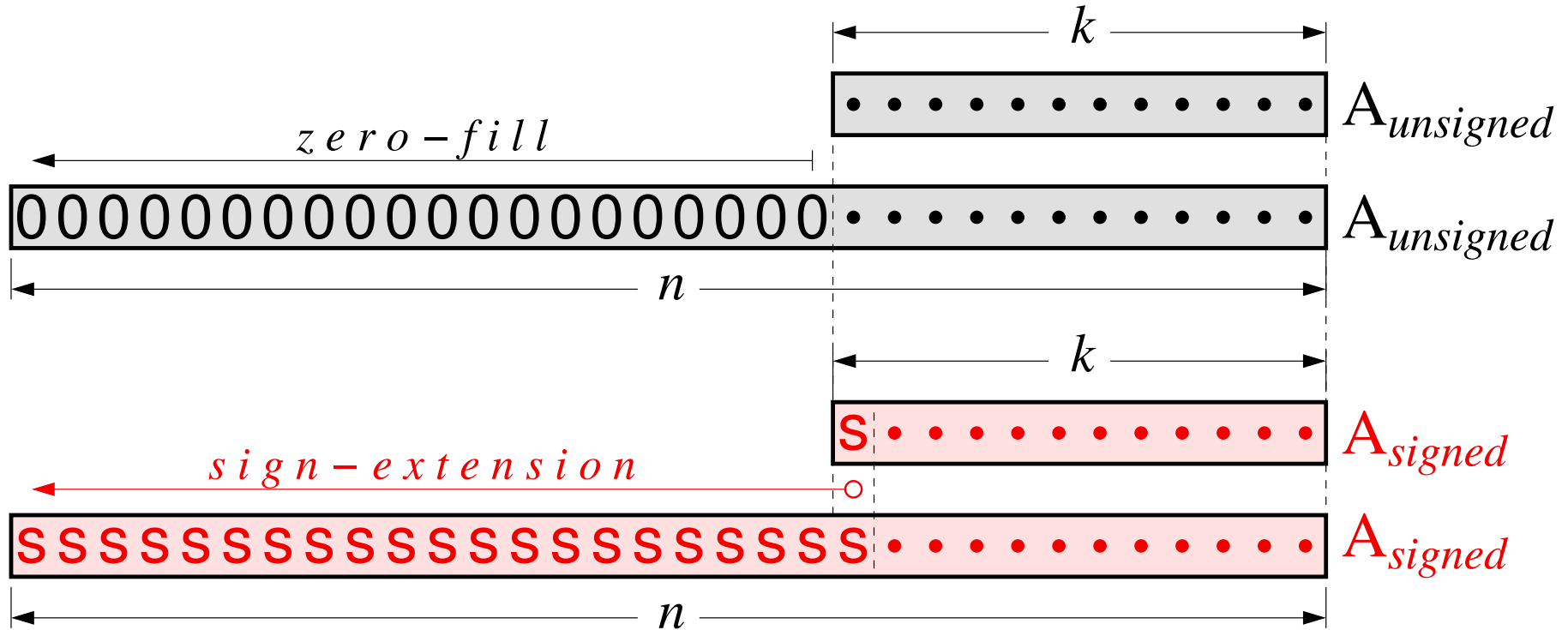
- MS bit == 0
- ίδιος αριθμός όπως και απρόσημος
- ίδιος οσαδήποτε μηδενικά αριστερά



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



Προσημασμένοι σε περισ. bits: Επέκταση Προσήμου

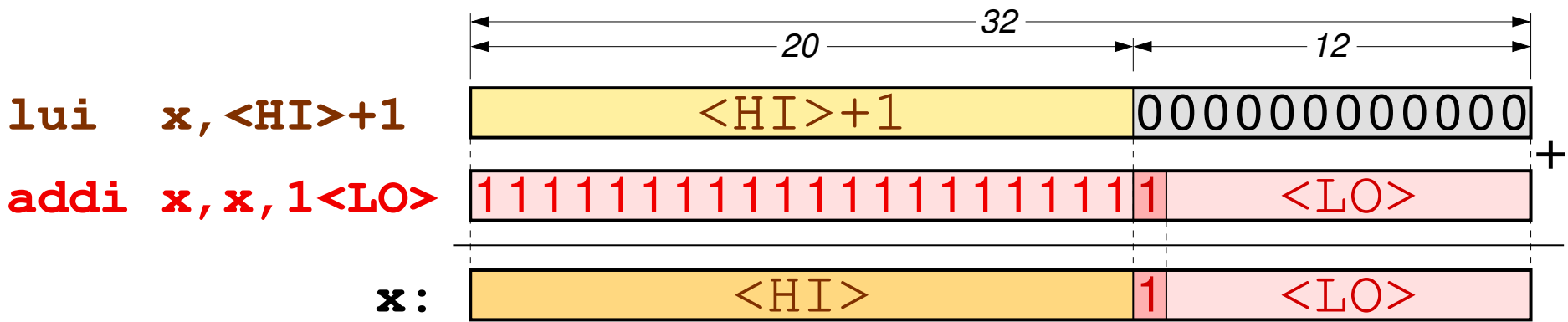
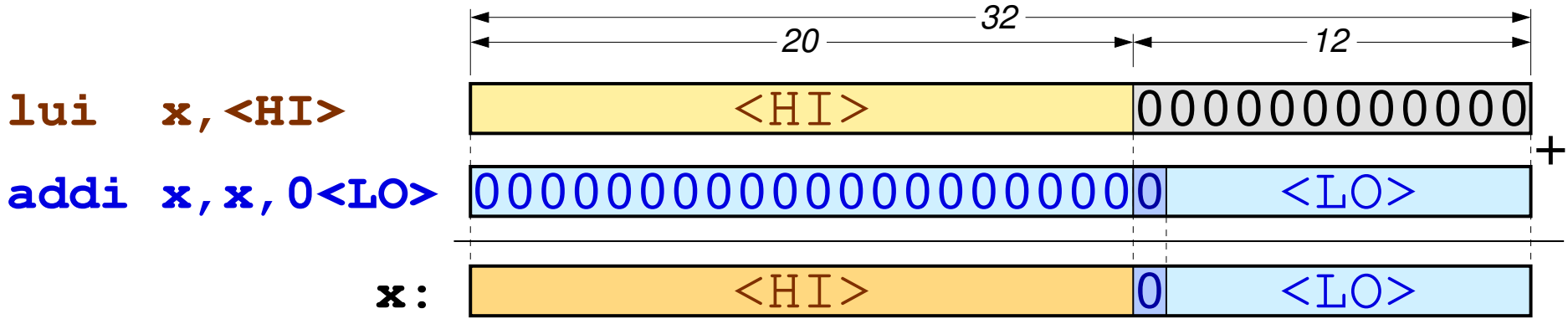


- Απρόσημοι: Συμπλήρωση Μηδενικών (ανεξαρτήτως MS bit)
- Προσημασμένοι (όλα τα *Immed.* στον *RV*): Επέκταση Προσήμου

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 ($x26 \neq x27$) goto label
- `blt x26, x27, label` # if ($x26 < x27$) goto label
- `bge x26, x27, label` # if ($x26 \geq x27$) goto label

- γιατί δεν χρειάζονται `ble` (\leq), `bgt` ($>$) ? (άσκηση...)
- `bltu` `bgeu` \rightarrow unsigned variants
- Δεν υπάρχουν συγκρίσεις καταχωρητή-σταθεράς
– γιατί; (άσκηση...)

[?]

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;
```

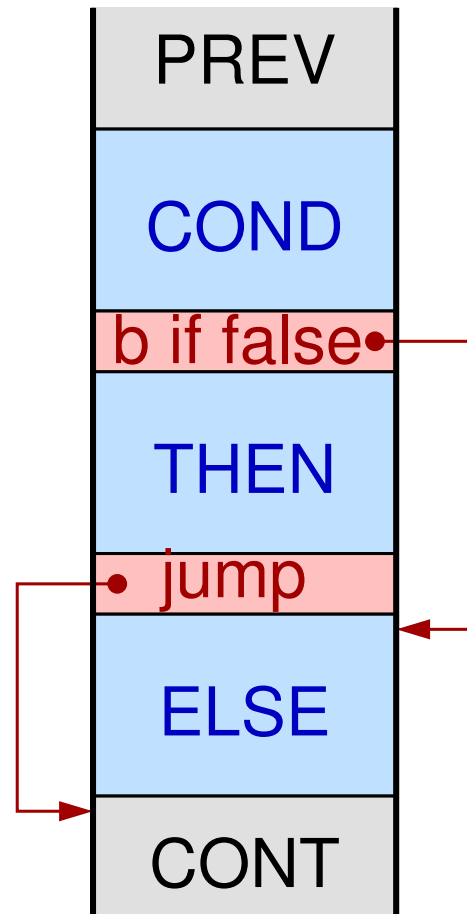
```
    bne i, j, else1
```

```
    add f, g, h
```

```
    j    cont1      #jump
```

```
else1: sub f, g, h
```

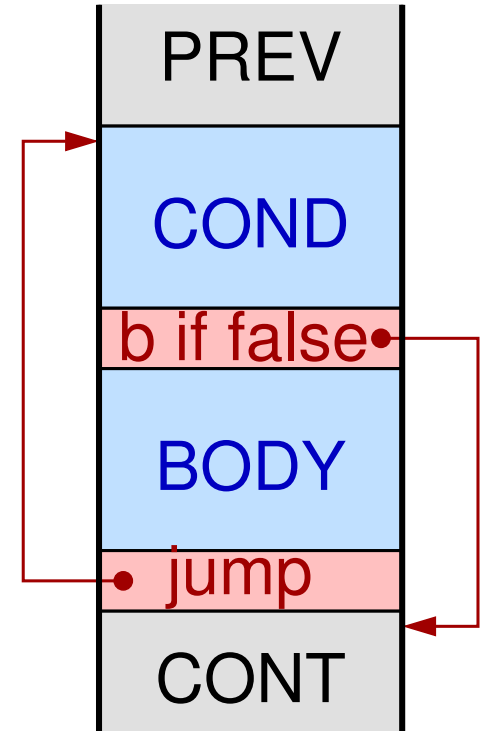
```
cont1: add h, zero, zero
```



Μετάφραση Βρόχου while σε Assembly

```
long long int i, key, table[N];      t1: i
i = 0;                               t2: key
while ( table[i] != key ) { i++; }   s0: table
/* table[] must contain at least one element==key */
```

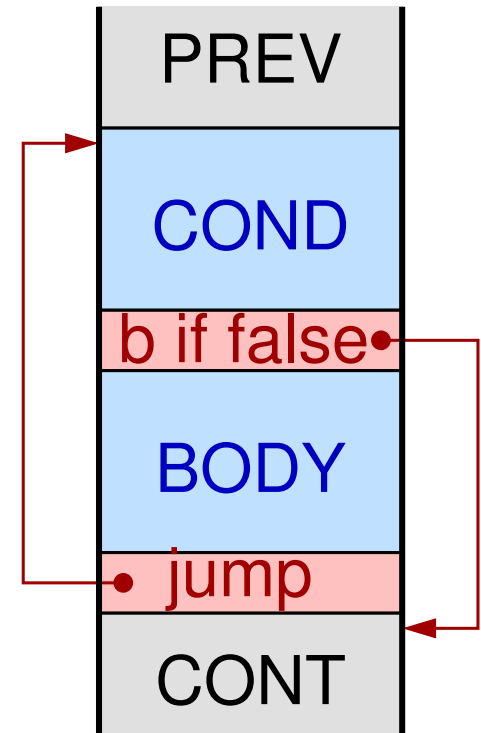
```
        add    t1, x0, x0    # i=0;
loop1:  slli   t0, t1, 3     # 8*i
        add    t0, s0, t0    # &table[i]
        ld     t0, 0(t0)    # table[i]
        beq   t0, t2, exit1
        addi   t1, t1, 1    # i++;
        j     loop1
exit1:  ...
```



Βελτιστοποίηση Βρόχου με pointers

```
long long int *p, key, table[N];      t1: p
p = table;                            t2: key
while ( *p != key ) { p++; }         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
```

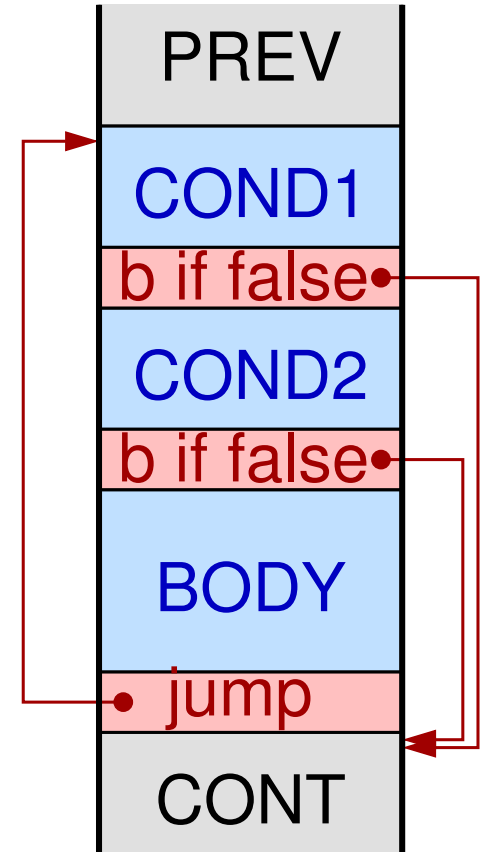
```
    beq t0, t2, exit2
```

```
    lw  t1, 4(t1)   # p=p->next;
```

```
    j   loop2
```

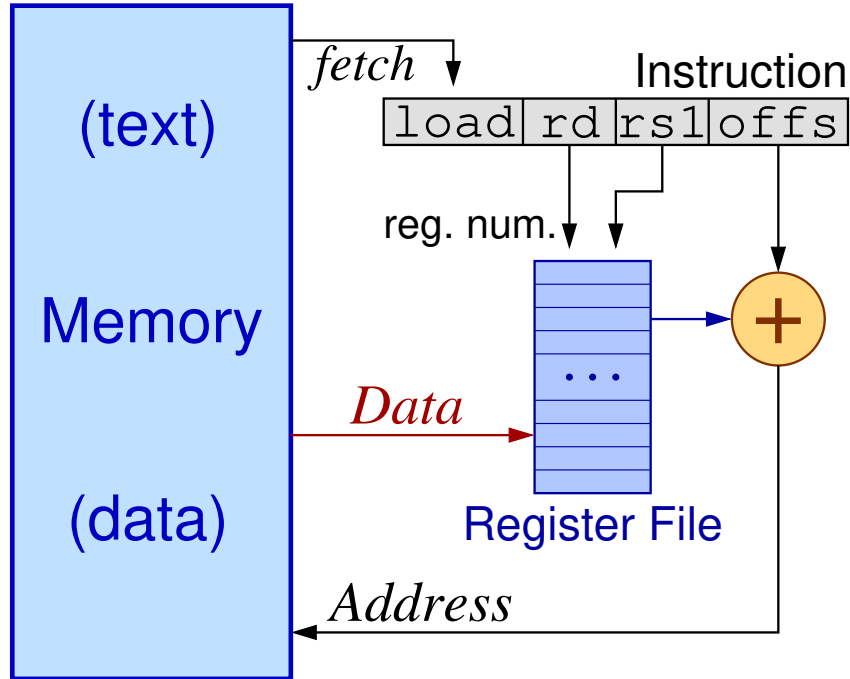
```
exit2: ...
```

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



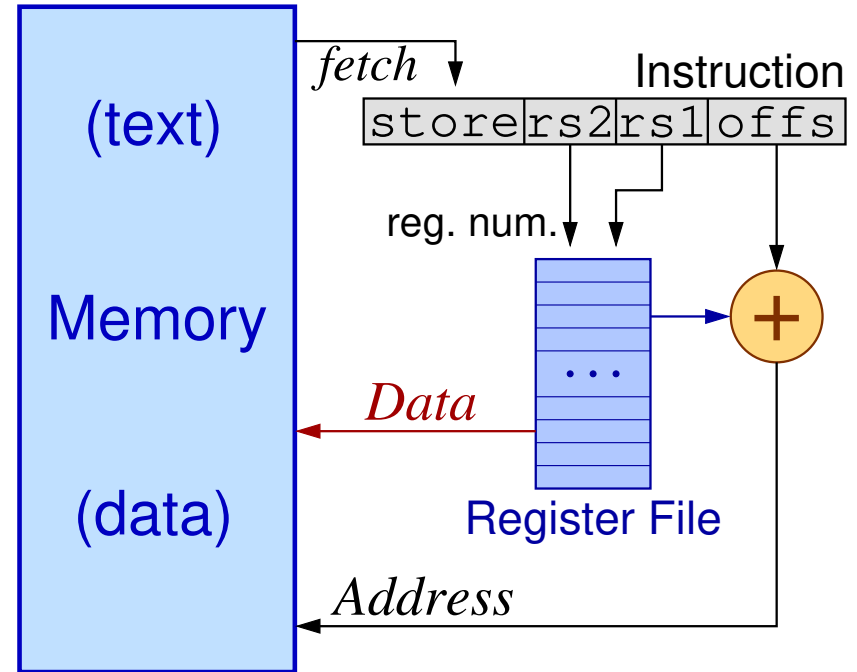
Οι εντολές Load και Store στον RISC-V

`lw rd, offset(rs1)`



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

`sw rs2, offset(rs1)`



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

- 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 *λιγότερη σημαντική (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 \Rightarrow «επεκτ. προσ.»
 - lbu, lhu (και lwu σε RV64I) θεωρούν unsigned \Rightarrow “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];`

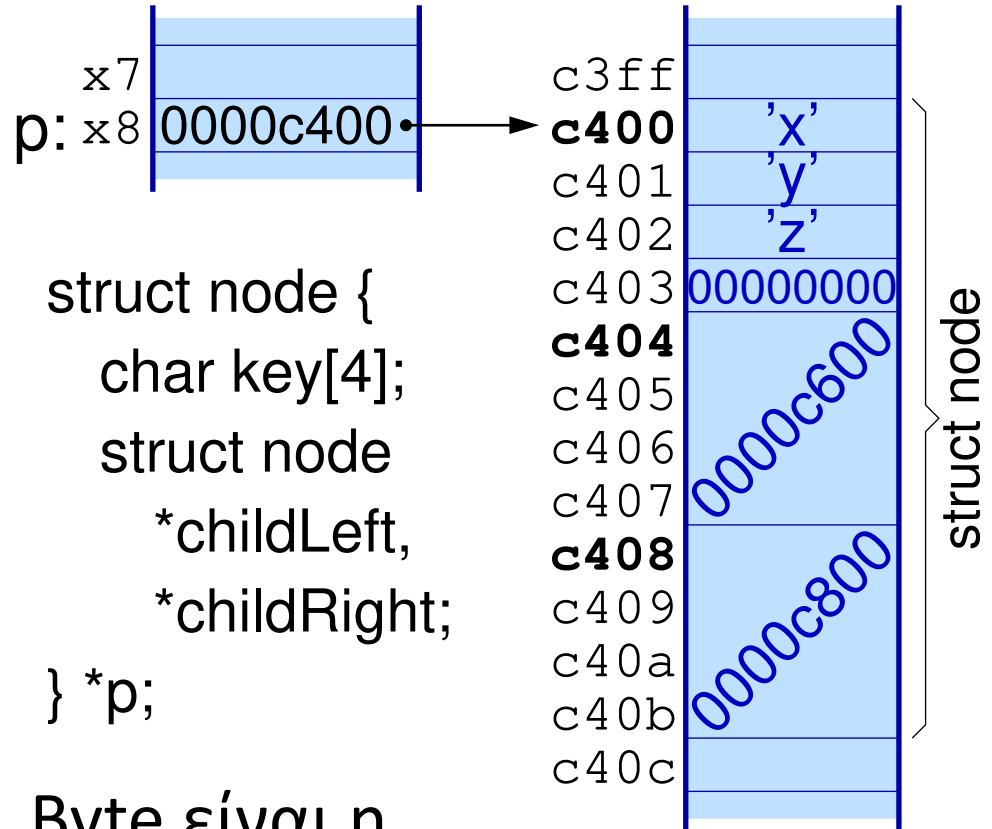
- `lw x7, 4(x8)`

 - `tmp2 = p->childLeft;`

- `lw x8, 8(x8)`

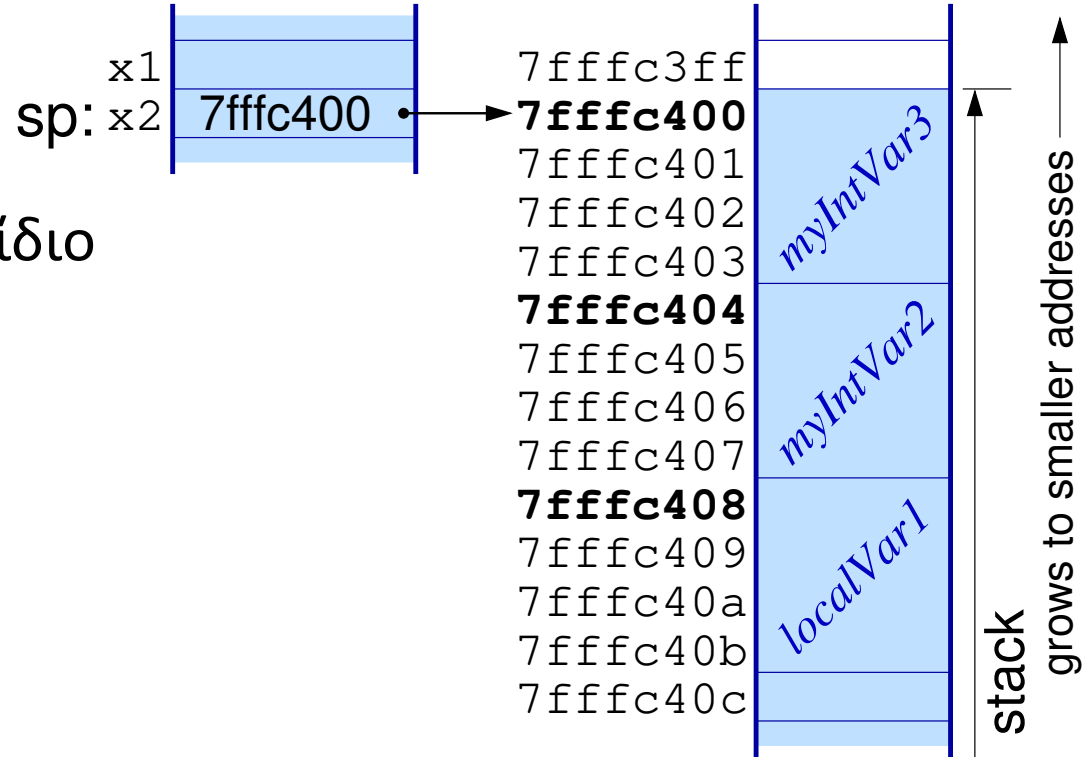
 - `p = p->childRight;`

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



Χρήση 2: Προσπέλαση τοπικών μεταβλ. στη Στοίβα

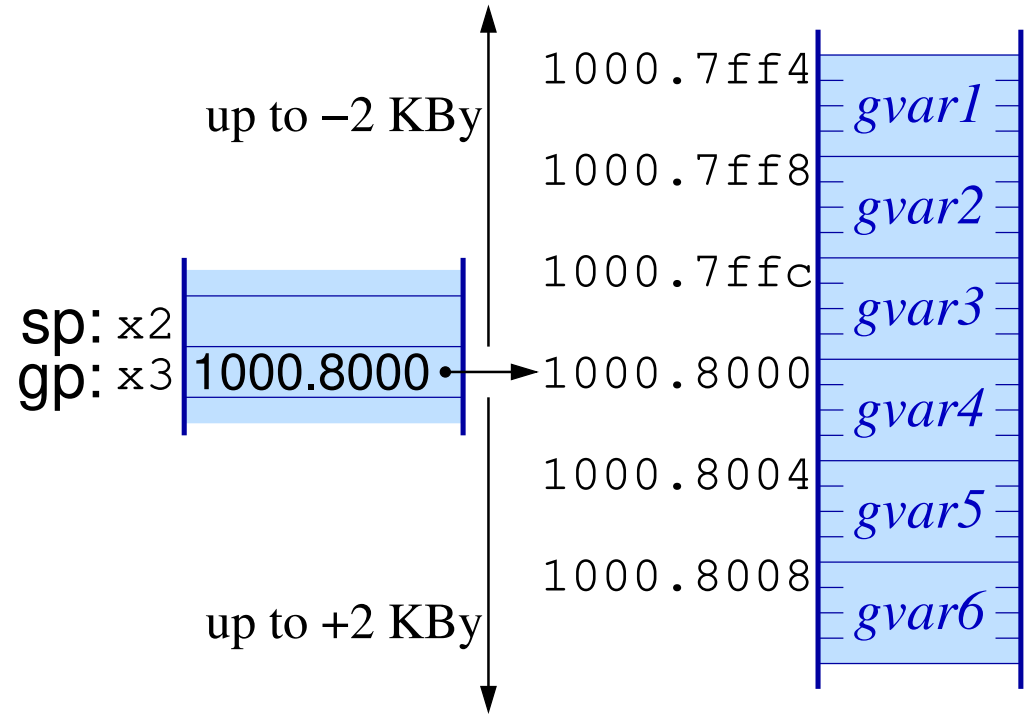
- `lw x5, 8(x2)`
- `lw x5, 8(sp)`
 - “x2” και “sp” είναι το ίδιο
 - `tmp0 = localVar1;`
- `sw x6, 4(sp)`
 - `myIntVar2 = tmp1;`
- `lw x7, 0(sp)`
 - `tmp2 = myIntVar3;`



“Local” variables in procedures: stack of activation frames

Χρήση 3: Καθολικές στατικές μεταβλητές μέσω gp

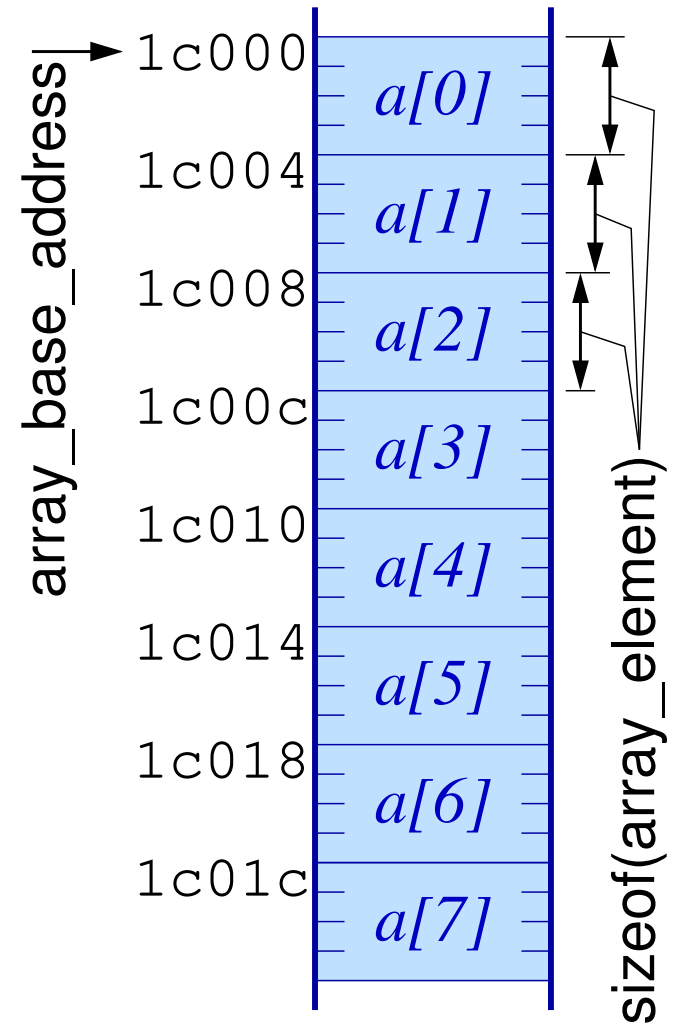
- Μεταβλητές ορισμένες εκτός διαδικασιών στη C: Στατικές, καθολικές (global)
- Χωριστά οι βαθμωτές (scalar – not arrays) εδώ, ώστε να χωρούν όλες, ει δυνατόν, σε 4 KBytes
- **x3 (gp – global pointer)** δείχνει στη μέση του χώρου αυτού (12-bit signed offset)



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

Πίνακες (arrays)

- Διεύθυνση Στοιχείου $i =$
Διεύθυνση Βάσης +
 $i \times \text{sizeof}(element)$
- Όταν βαθμωτά στοιχεία, συνήθ.
 $\text{sizeof}(element)$ δύναμη του 2
 \Rightarrow πολλαπλασιασμ. = αρ. ολίσθηση
- Συνήθως οι Compilers βελτιστοπ.:
for (i=0; i<N; i++) { ... a[i] ... } \Rightarrow
for (p=a; p<a+N; p++) { ... *p ... }

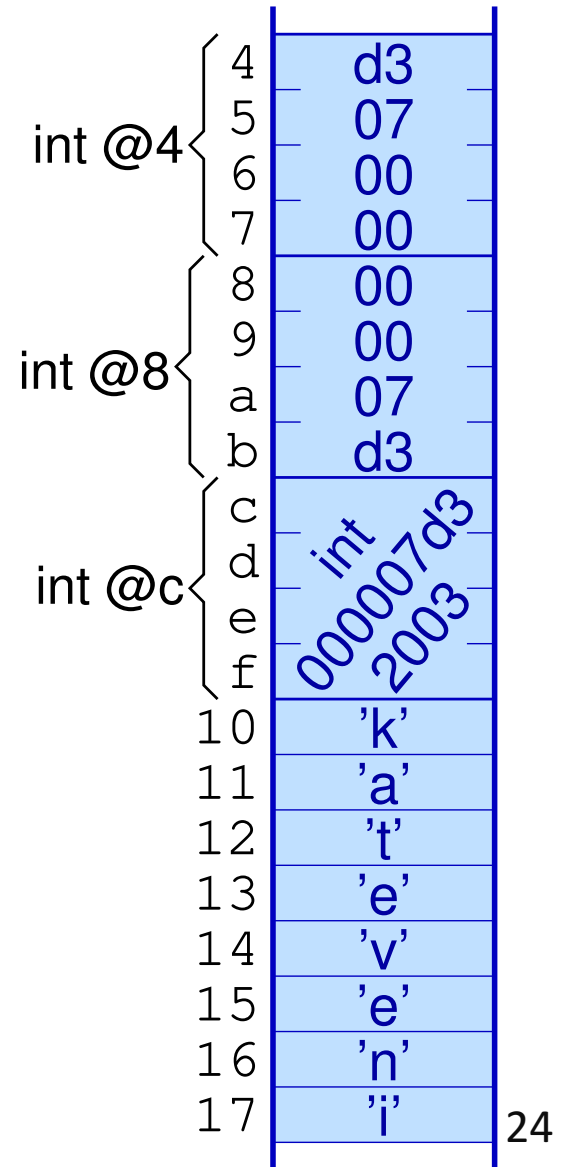


Μνήμη Byte-Addressable

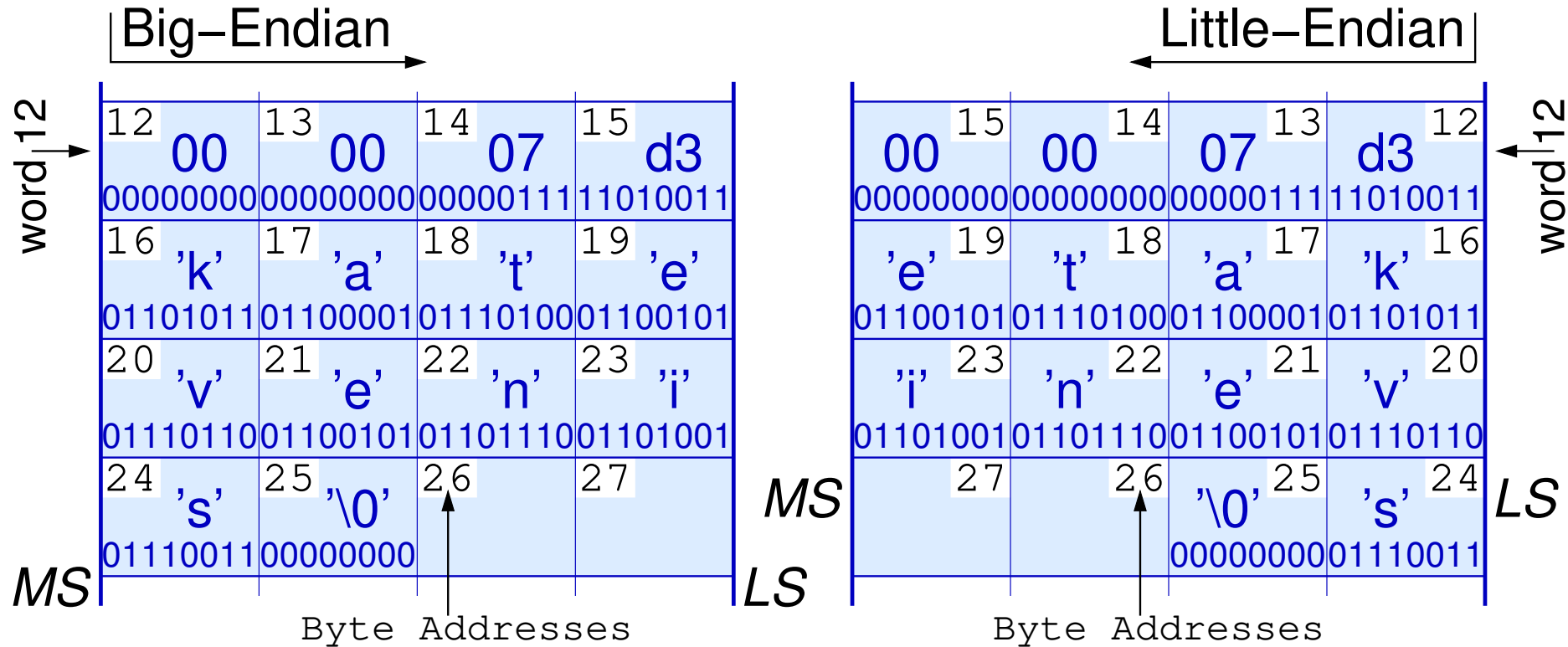
- Κάθε Byte έχει τη δική του Διεύθυνση
- Πίνακες κατά αύξουσες & συνεχόμενες διευθύνσεις, πάντα
 - π.χ. array of char @ base addr. 10

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 ?



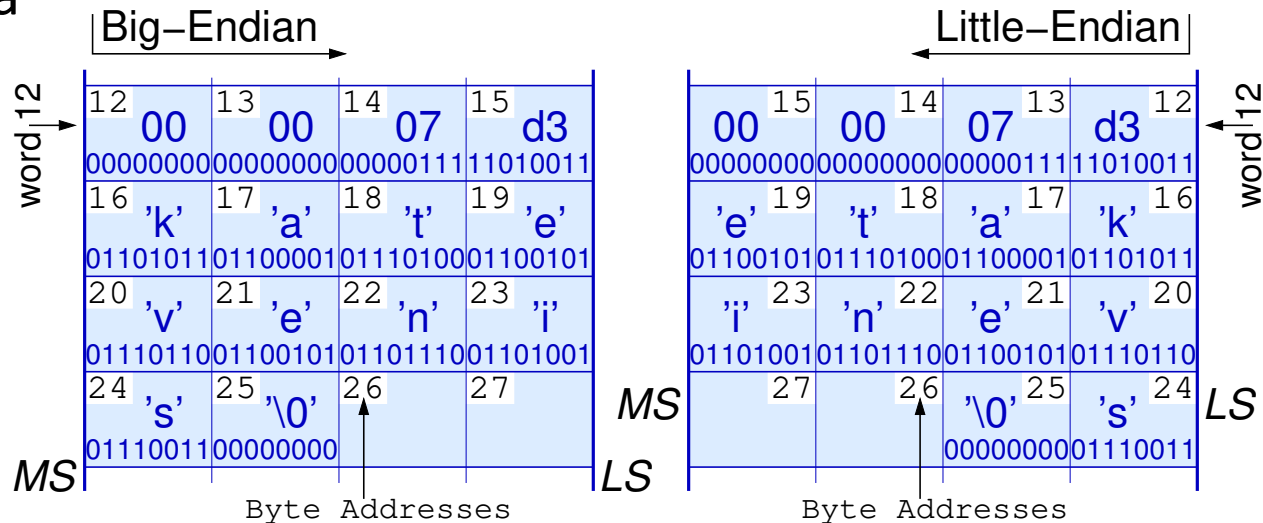
Αρίθμηση των Bytes μέσα σε μία Λέξη Ακεραίου;



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

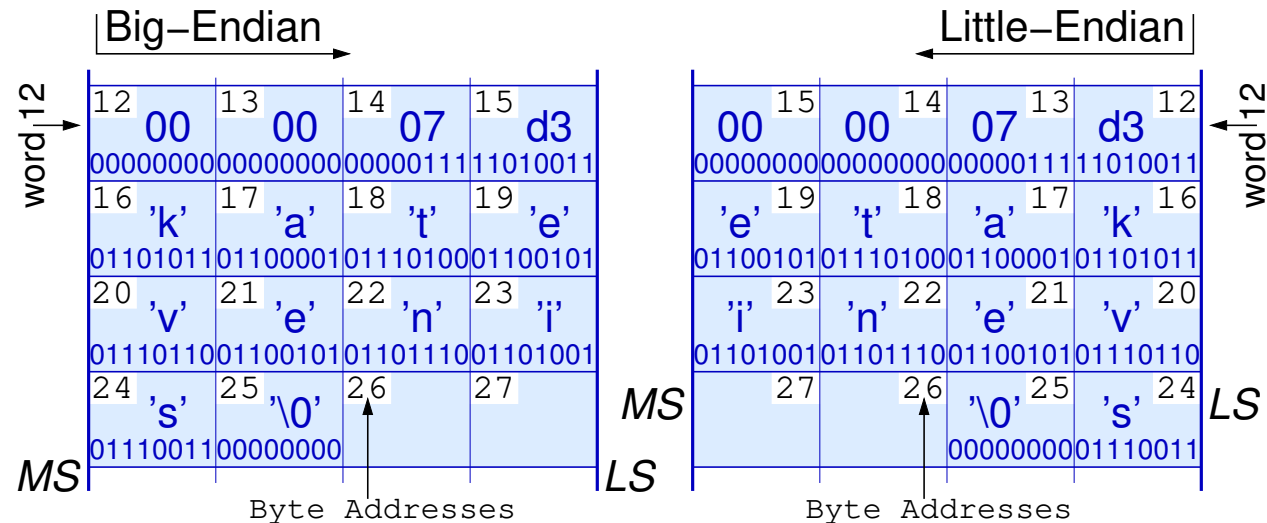


Τι με νοιάζει εμένα; – (2) «παράξενα» προγράμματα

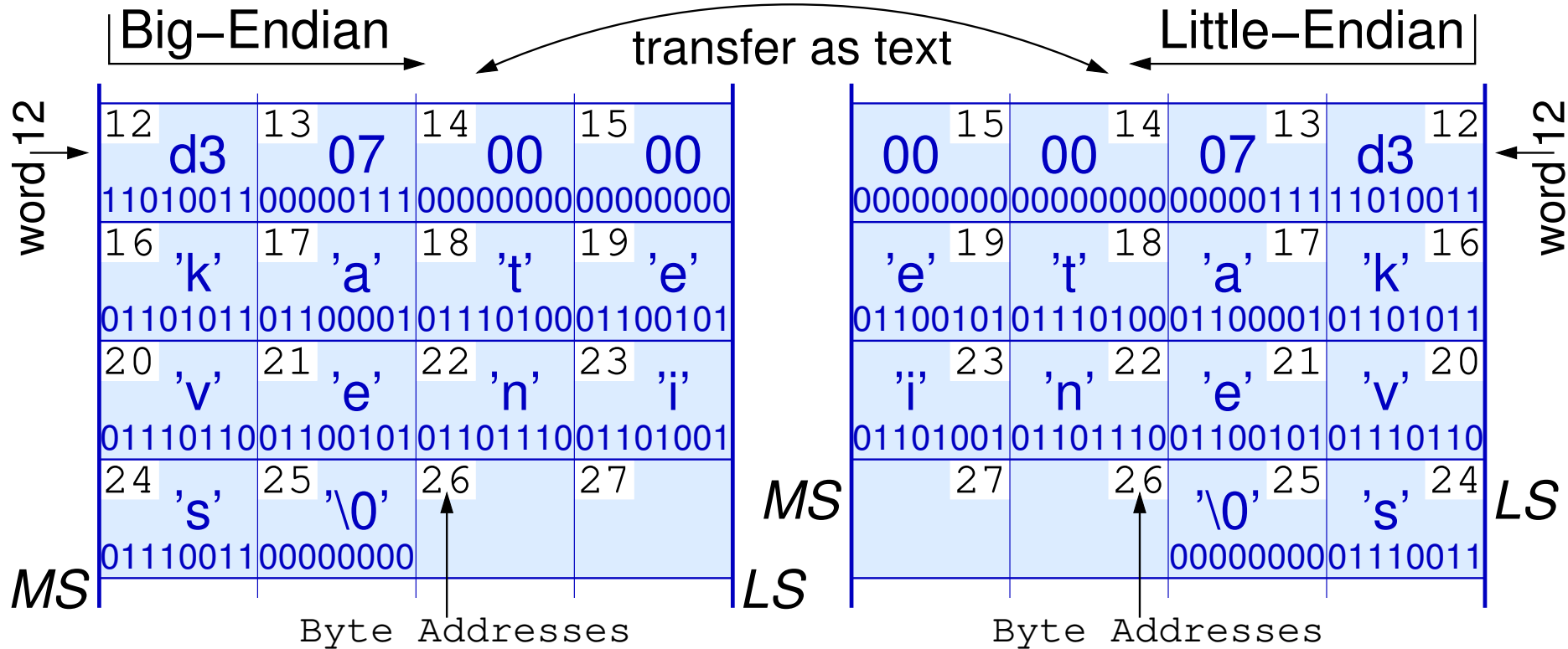
- γράφω string “katevenis” ξεκινώντας από τη διευθ. 16
 - διαβάζω int (!) από δ. 16:
 - αλλού (big-e.) → 6b617465
 - αλλού (little-e.) → 6574616b
- γράφω int year=2003 στη δ. 12
 - διαβάζω char (!) από διευθ. 12:
 - αλλού (big-e.) βρίσκω: ‘\0’ (00)
 - αλλού (little-e.) βρίσκω: ‘Σ’ (d3)

• non-portable program !

- η συμπεριφορά αλλάζει από μηχανή σε μηχανή
- (και μάλλον ανόητο πρόγραμμα)

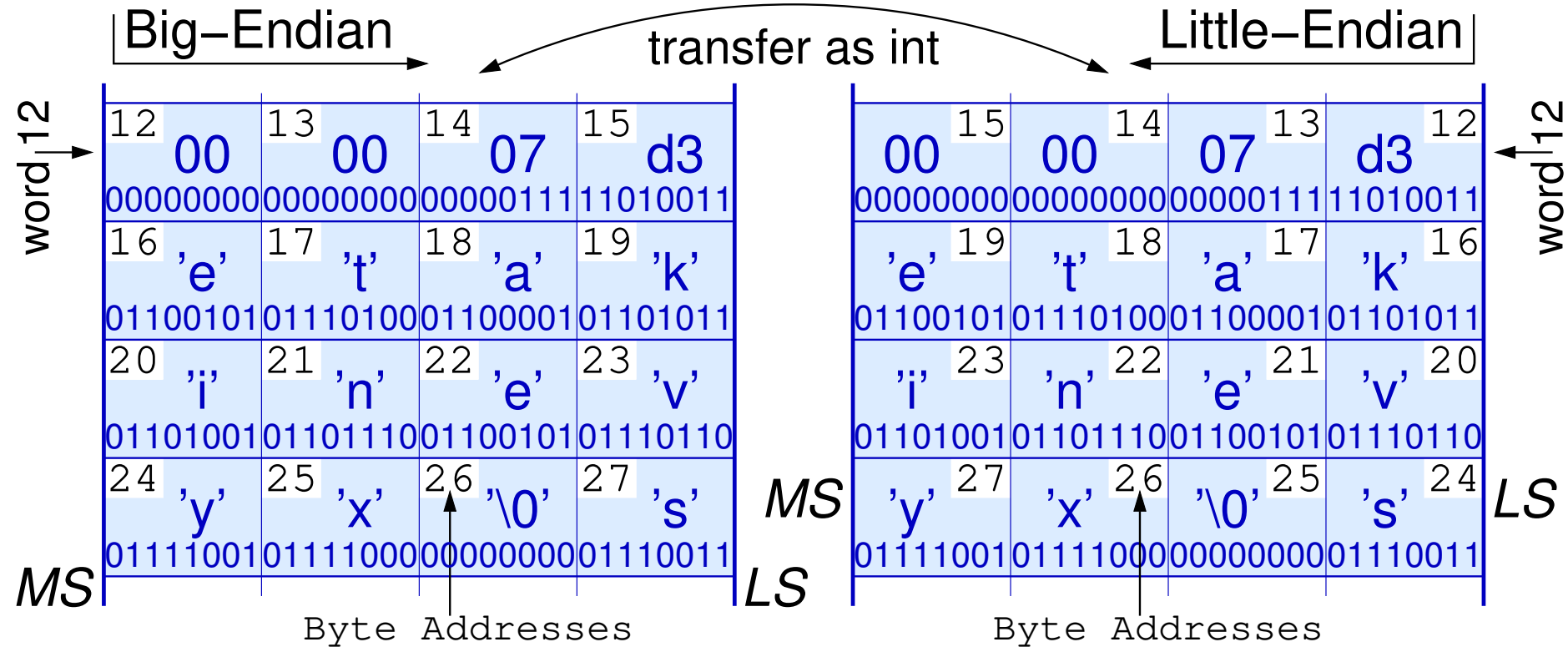


Τι με νοιάζει εμένα; – (3) μεταφορά αρχείου ως text



Ο ακέραιος 2003 γίνεται -754.515.968 στην άλλη μηχανή!

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



To string @16 γίνεται "etakinevnx" στην άλλη μηχανή!

Προσπελάσεις σε Φαρδιές Μνήμες: Ευθυγράμμιση

- Οι πραγματικές μνήμες συνήθως φαρδιές (32, 64,... bits)
 - για 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* για κάθε Byte position: για 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 για ποσότητα *n* Bytes?

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

1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14
17	16
19	18
21	20
23	22

Addresses drawn assuming: *Little Endian layout*

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20

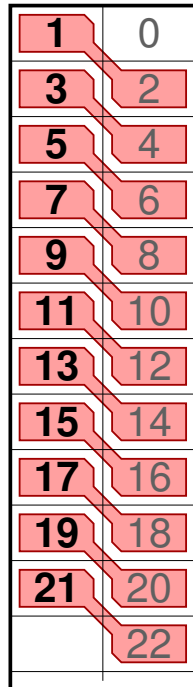
7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16

64 bits = 8 Bytes

- 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)

16 b
2 By



32 bits = 4 Bytes

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

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
	22	21	20

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
	22	21	20	19	18	17	16

64 bits = 8 Bytes

- 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

1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14
17	16
19	18
21	20
23	22

← 16 b
2 By →

Addresses drawn assuming: *Little Endian layout*

1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14
17	16
19	18
	20

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20

← 32 bits = 4 Bytes →

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16

← 64 bits = 8 Bytes →

- 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. $\text{Addr mod } 4 == 1$

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
			20

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
			20	19	18	17	16

← 64 bits = 8 Bytes →

- 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

1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14
17	16
19	18
21	20
23	22

← 16 b
2 By →

Addresses drawn assuming: *Little Endian layout*

1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14
17	16
19	18
21	20

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

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20

← 32 bits = 4 Bytes →

(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 multiples of 2, but NOT multiples of 4 (i.e. $\text{Addr mod } 4 == 2$)

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
		21	20

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16

← 64 bits = 8 Bytes →

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

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
		21	20	19	18	17	16

← 64 bits = 8 Bytes →

- 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

1	0
3	2
5	4
7	6

Addresses drawn assuming: *Little Endian layout*

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20

← 32 bits = 4 Bytes →

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16

← 64 bits = 8 Bytes →

- 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)

9	8
11	10
13	12
15	14
17	16
19	18
21	20
23	22

← 16 b
2 By →

1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14
17	16
19	18
21	20
23	22

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)

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16

← 64 bits = 8 Bytes →

- 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

1	0
3	2
5	4
7	6

Addresses drawn assuming: *Little Endian layout*

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20

← 32 bits = 4 Bytes →

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16

← 64 bits = 8 Bytes →

- 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)

9	8
11	10
13	12
15	14
17	16
19	18
21	20
23	22

← 16 b
2 By →

1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14
17	16
19	18
21	20
23	22

8-Byte "Doubles" at Addresses that are multiples of 4, but NOT multiples of 8 (i.e. $\text{Addr mod } 8 == 4$)

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20

7	6	5	4	3	2	1	0
15	14	13	12	11	10	9	8
23	22	21	20	19	18	17	16

← 64 bits = 8 Bytes →

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

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

Γιά ταχύτητα: Ευθυγράμμιση στα «φυσικά όρια»

Ηθικόν Δίδαγμα:

- Για τον ελάχιστο δυνατό αριθμό προσπελάσεων μνήμης,
- σε μνήμη οιουδήποτε πλάτους (πάντα δύναμη του 2)
 - δηλαδή για 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. ακερ. πολλαπλ. του 8
- Γενικά: Ευθυγράμμιση της κάθε ποσότητας (2^n By)
στα «φυσικά» της όρια (ακερ. πολ. του 2^n)

Οδηγίες στον Assembler για 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 \Rightarrow σε ακέραιο πολλαπλ. του $2^2 = 4$
 - .align 3 \Rightarrow σε ακέραιο πολλαπλ. του $2^3 = 8$
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 \Rightarrow κράτησε εδώ χώρο 32 Bytes (άφησε κενό)

Παράδειγμα: .align .space

```
mystr: .asciz "katevenis"  
      .align 3 # πήδα στο επόμενο  
              ακέρ. πολ. του 23=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
```

mystr=	400	k
	401	a
	402	t
	403	e
	404	v
	405	e
	406	n
	407	i
	408	s
	409	\0
	410	
	411	
	412	
	413	
	414	
	415	
p_dbl=	416	
	417	
	418	
	419	
	420	
	421	
	422	
	423	
i_wrd=	424	
	425	
	426	
	427	
	428	

Ευθυγραμμισμένες Ποσότητες & τα bits Διεύθυνσης

000001	000000
000011	000010
000101	000100

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

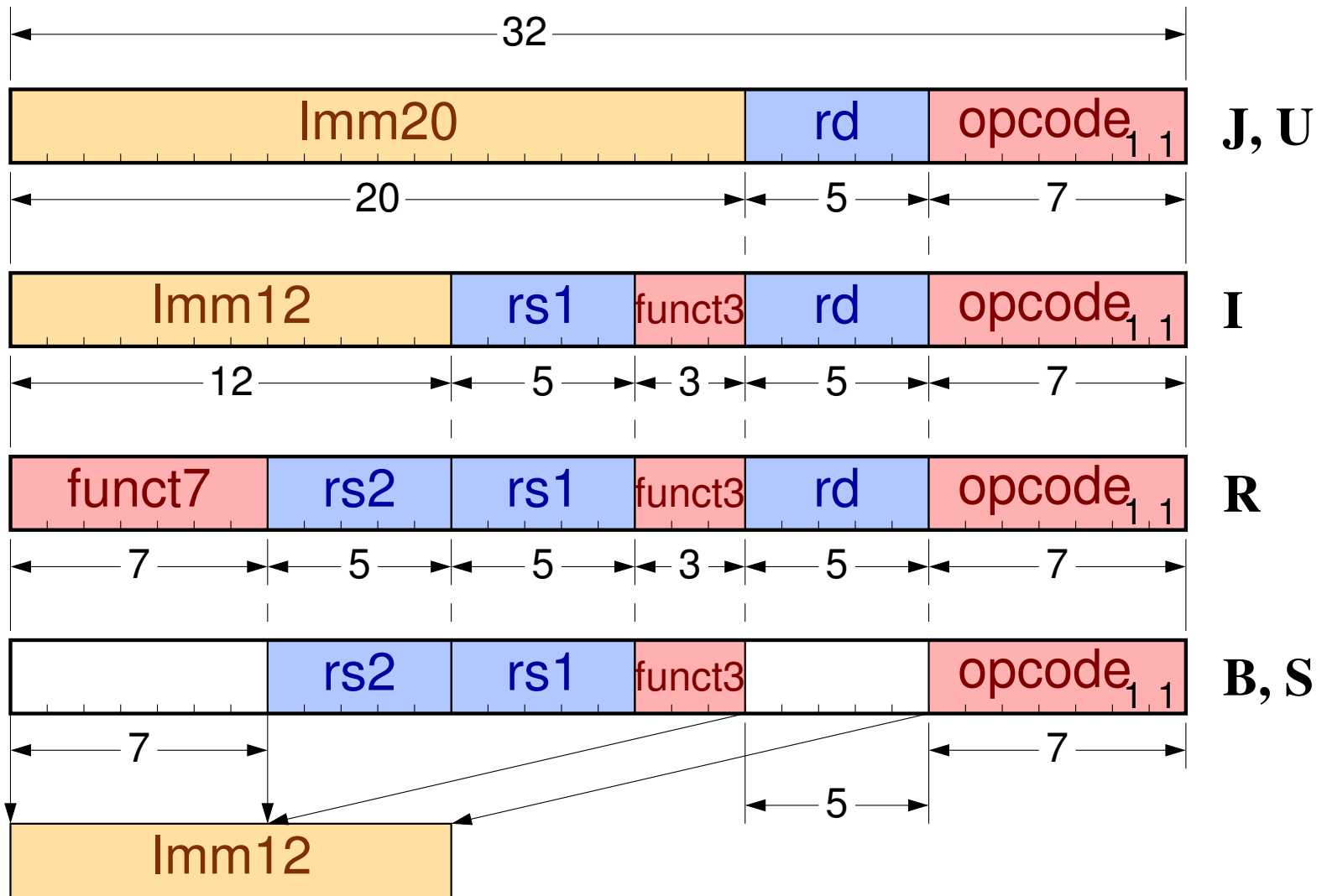
000111	000110	000101	000100	000011	000010	000001	000000
001111	001110	001101	001100	001011	001010	001001	001000
010111	010110	010101	010100	010011	010010	010001	010000

- 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 για όλες τις εντολές: έτσι αρχίζει η ανάγνωσή τους πριν την αποκωδικοποίηση του opcode
 - Καταχωρητής προορισμού σε ίδια θέση για όλες τις εντολές: έτσι συγκρίνεται με τους καταχωρητές πηγής επόμενων εντολών, για 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

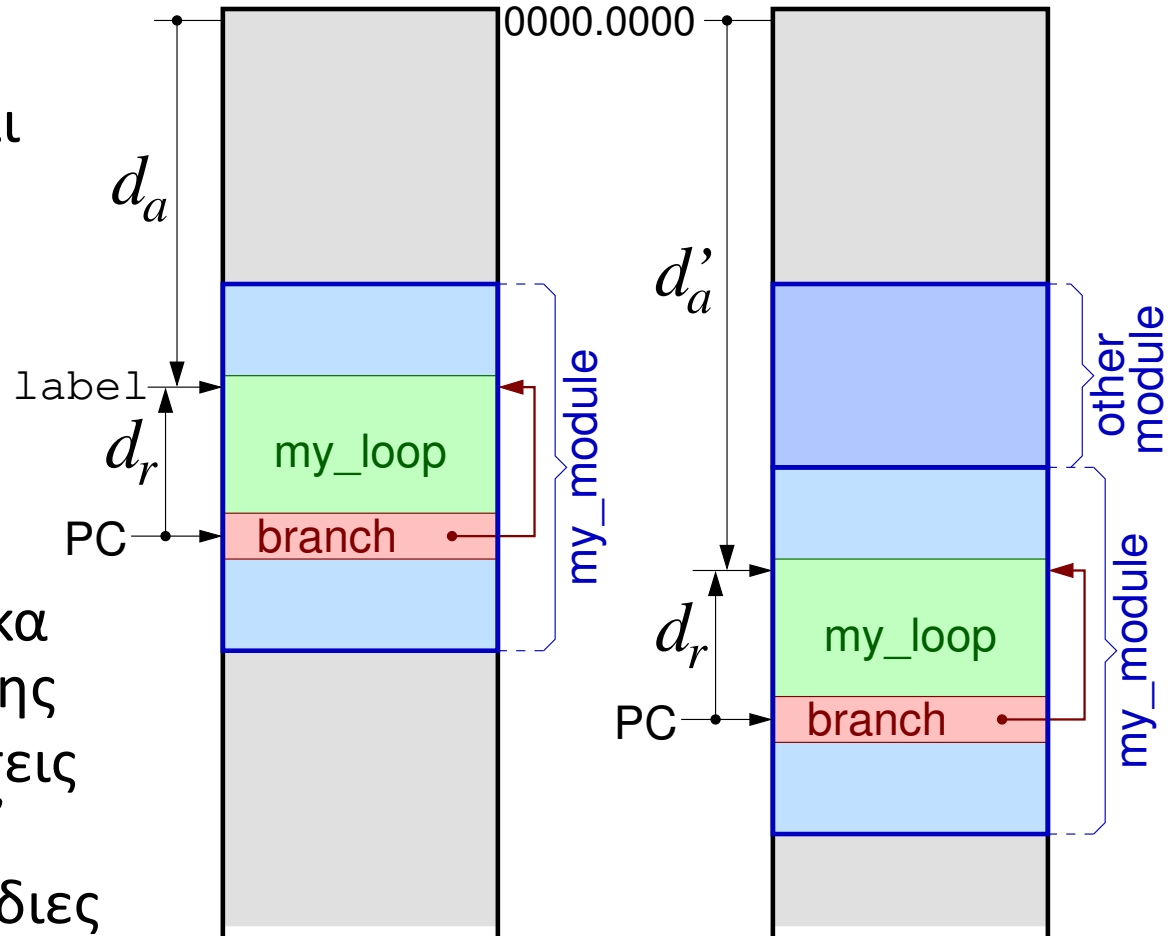
`beq rs1, rs2, Imm12` \Rightarrow

if (`rs1==rs2`) {PC \leftarrow PC+2×Imm12} else {PC \leftarrow PC+4}

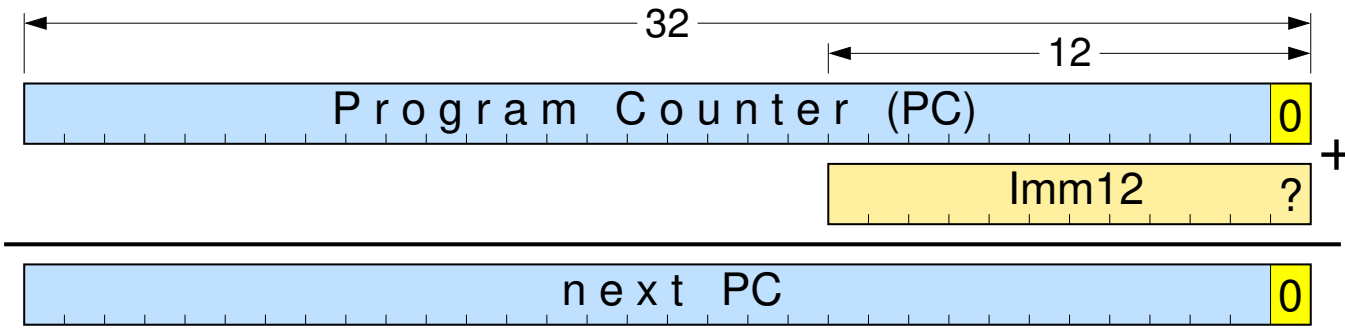
- Η σταθερά Imm12 (offset) πάντα προσημασμένη
- Θετικά offsets: συνήθως if-then-else
- Αρνητικά offsets: συνήθως βρόχοι

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 διευθύν.
- Relocation
 - φόρτωση του κώδικα σε άλλη θέση μνήμης
 - απόλυτες διευθύνσεις d_a αλλάζουν σε d_a'
 - σχετικές διευθ. d_r ίδιες

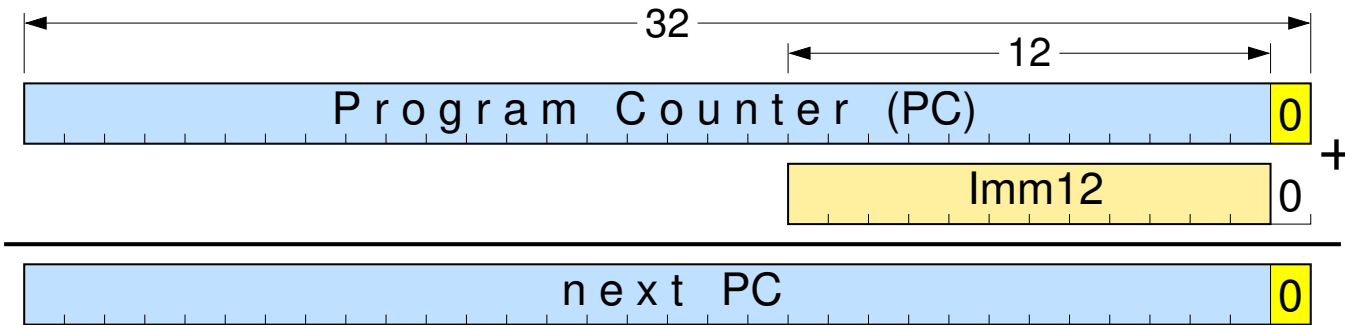


PC-relative addr.: Ευθυγράμμιση εντολών & Βεληνεκές



Απλοϊκή
πρόσθεση \Rightarrow
Βεληνεκές
 $= \pm 2$ KBytes

Ευθυγράμμιση εντ. RISC-V πάντα σε όρια half-words \Rightarrow δεξιό bit PC πάντα 0 \Rightarrow δεξιό bit σταθεράς υποχρεωτικά 0 \rightarrow σπατάλη πληροφο.



Πρόσθεση
διπλασίου \Rightarrow
Βεληνεκές
 $= \pm 2$ KHalfWrd
 $= \pm 4$ 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:
$$\text{j al rd, Imm20} \Rightarrow \begin{aligned} \text{rd} &\leftarrow \text{PC}_{\text{old}} + 4, \\ \text{PC}_{\text{new}} &\leftarrow \text{PC}_{\text{old}} + 2 \times \text{Imm20} \end{aligned}$$
- Η «από κάτω» εντολή στο $\text{PC}_{\text{old}} + 4 =$ Διεύθυνση Επιστροφής
- Η σταθερά Imm20 (offset) – πάντα προσημασμένη:
 - 20 bits: όσο χωρούσε στο J-format
 - επαρκής για κάλεσμα έως $\pm 0.5 \text{ MHalfWords} = \pm 1 \text{ Mbyte}$
 - 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 άλληνη χρειαζ. στη στοίβα
 - μεγάλο ποσοστό των «δυναμικά» εκτελούμενων (\neq «στατικά» στο προγρ.) διαδικασιών δεν καλούν άλλη διαδικασία
 3. Σώσιμο άλλων μεταβλητών; \rightarrow βλ. §6

Jump (Ψευδοεντολή): ειδική περίπτωση Κλήσης Διαδ.

- $x1$ (αλλιώς: ra): συνήθης καταχ. για διευθ. επιστροφής
- Συνήθης κλήση διαδικασίας:

$$\text{j al } x1, \text{ Imm20} \Rightarrow \begin{aligned} x1 &\leftarrow PC_{\text{old}} + 4, \\ PC_{\text{new}} &\leftarrow PC_{\text{old}} + 2 \times \text{Imm20} \end{aligned}$$

- Ψευδοεντολή: **j Imm20** =

$$\text{j al } x0, \text{ Imm20} \Rightarrow PC_{\text{new}} \leftarrow PC_{\text{old}} + 2 \times \text{Imm20}$$

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

Άλματα σε μεταβλητές Διευθύνσεις: *Jump Register*

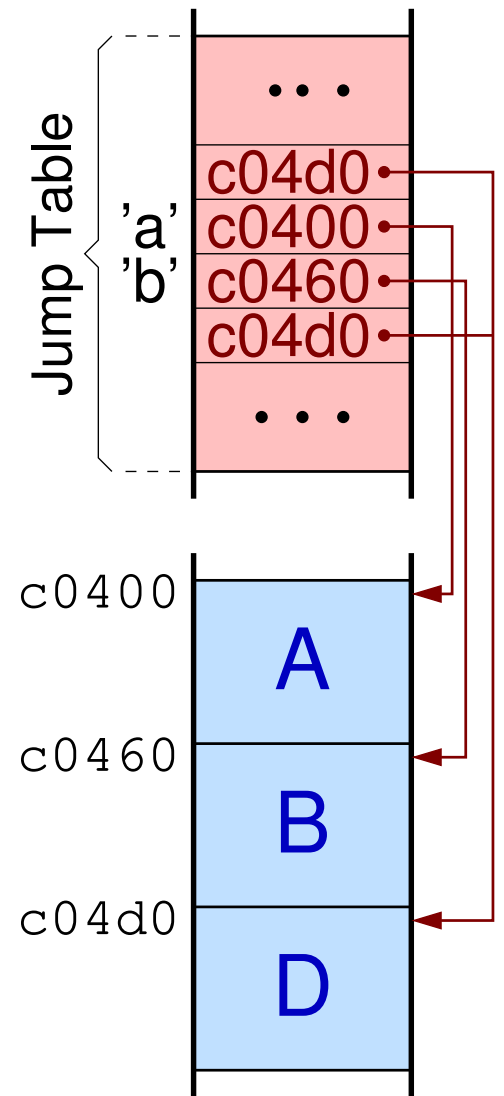
`jr rs1` \Rightarrow $PC \leftarrow rs1$

- Για Επιστροφή από Διαδικασία: `jr x1` ή `jr ra`
 - Μεταβλητή διεύθυνση διότι με καλούν από διάφορα μέρη
- Switch statement – multi-way “computed” branch
- Για 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 ← PCold + 4; PCnew ← 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 μία 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` # $rd \leftarrow (rs1 < rs2)$ – sign'd
- `sltu rd, rs1, rs2` # $rd \leftarrow (rs1 < rs2)$ – uns.
- `slti rd, rs1, Imm12` # $rd \leftarrow (rs1 < Imm12)$ s.
- `sltiu rd, rs1, Imm12` # $rd \leftarrow (rs1 < Imm12)$ 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 για σύνθεση πιο πολύπλ.

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

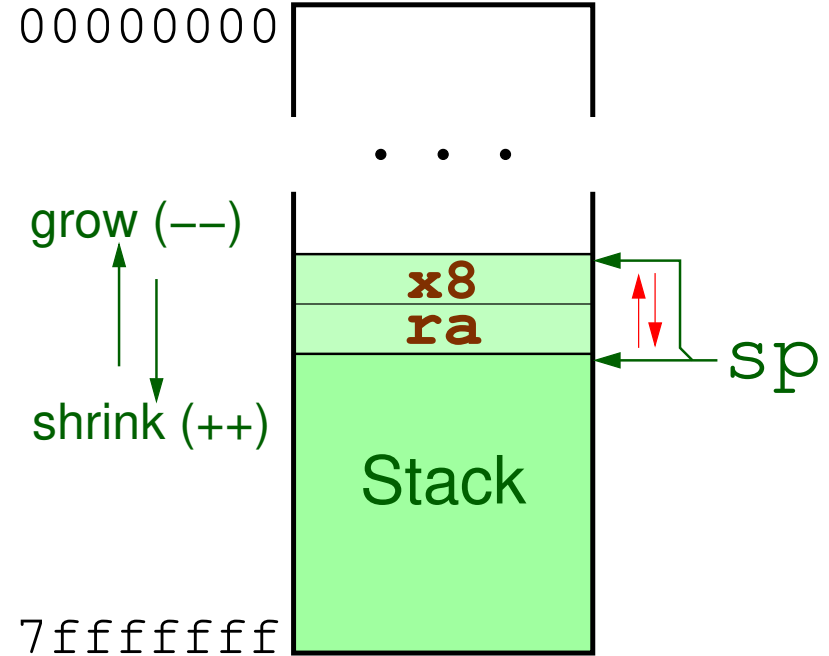
Σώσιμο Καταχωρητών στη Στοίβα και Επαναφορά

- 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

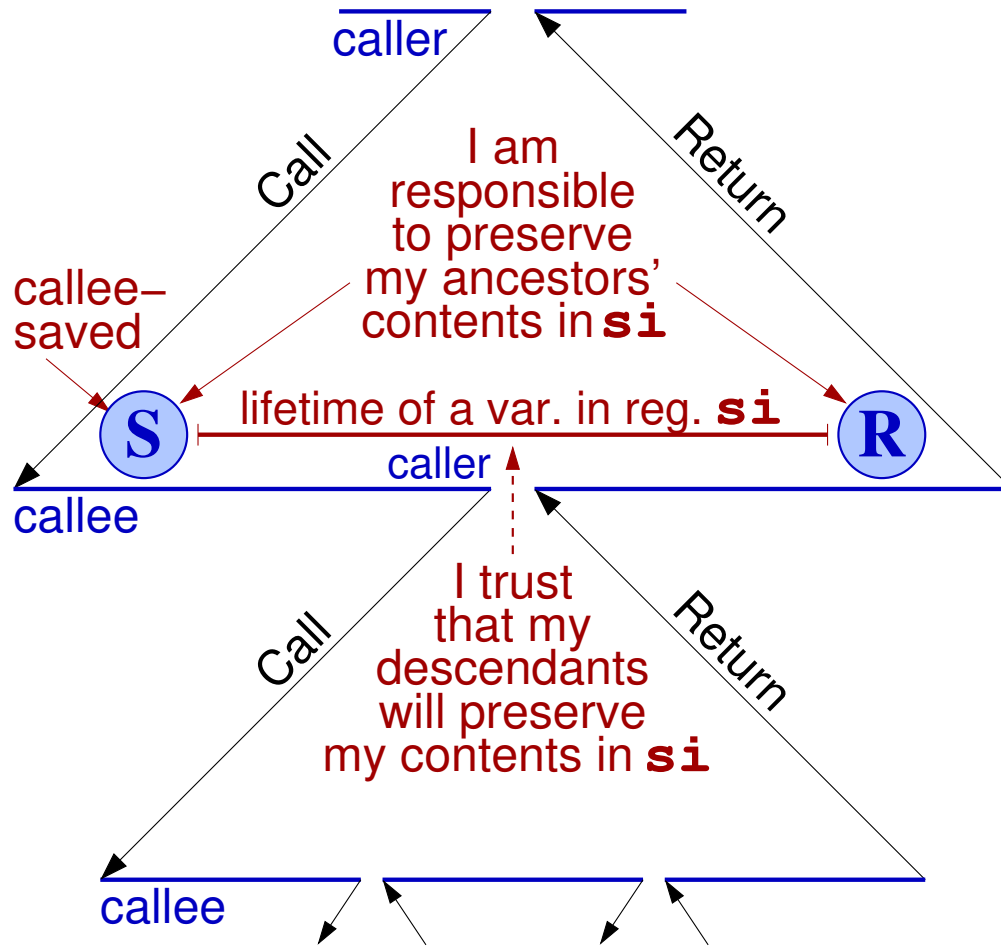
Χωριστή Μετάφραση: δεν ξέρω γονέα/παιδιά μου!

- 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, για 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

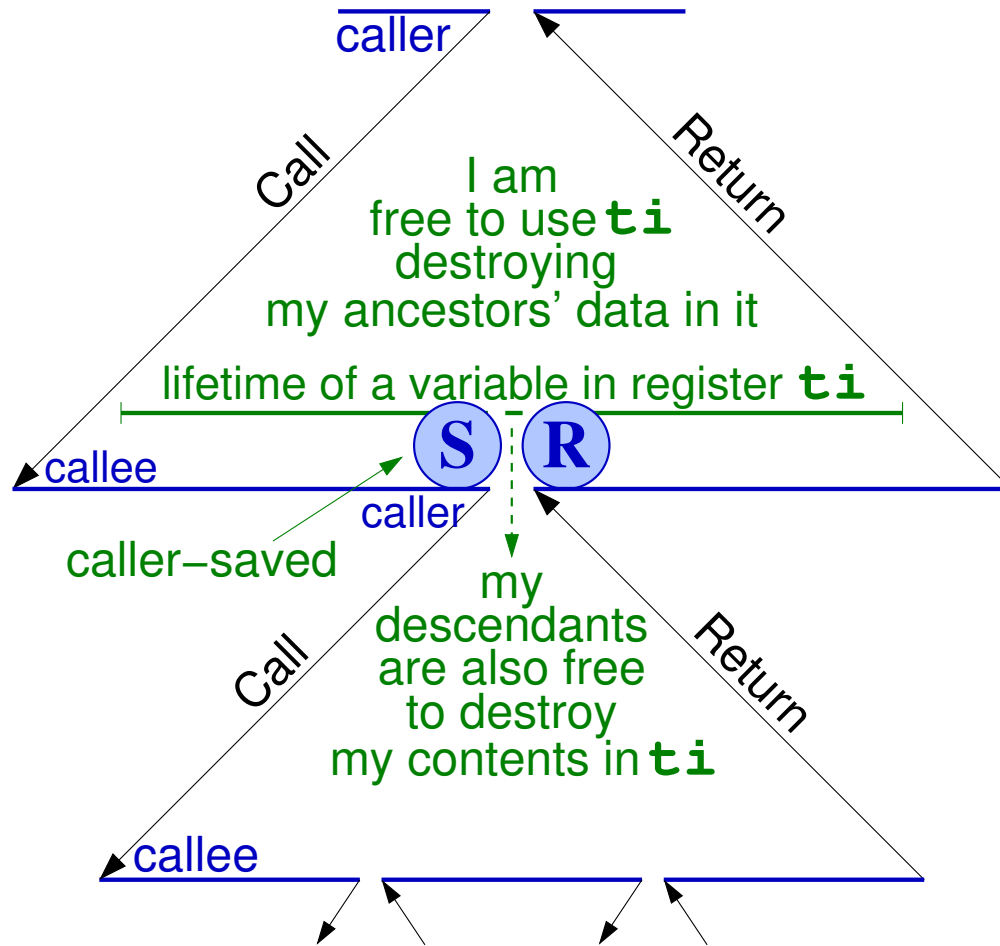
- Από την εκχώρηση νέας τιμής σε αυτήν/αυτόν
 - η οποία νέα τιμή δεν υπολογίζεται βάσει της παλαιάς του τιμής από την ίδια την εντολή που του εκχωρεί τη νέα τιμή
- Μέχρι την τελευταία ανάγνωση της τιμής του πριν την επόμενη εκχώρηση (ανεξάρτητης) νέας τιμής
 - εντολές που διαβάζουν και γράφουν τον ίδιο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

Το σώσιμο ευθύνη του καλουμένου (*Saved registers*)



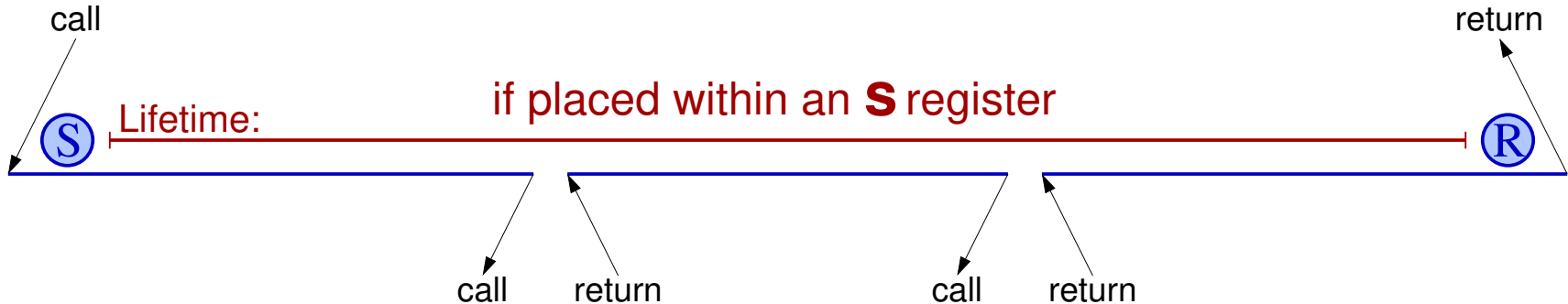
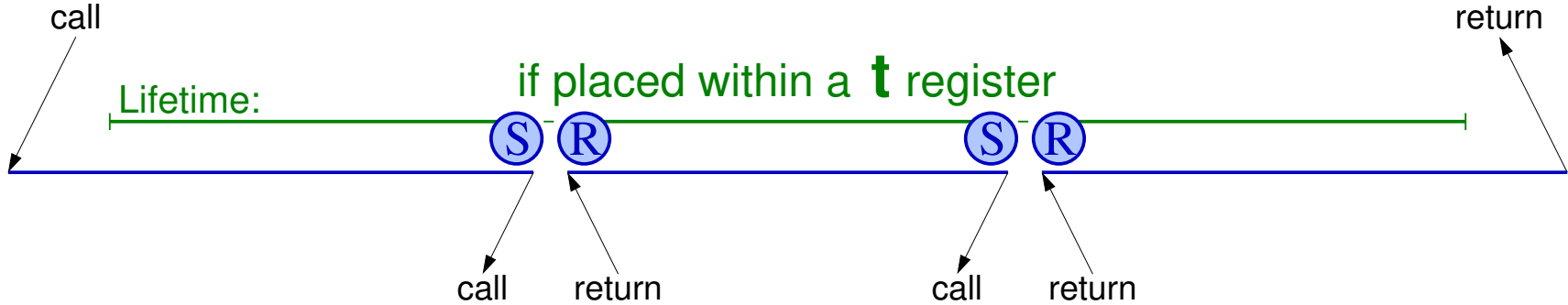
- Σύμβαση “Callee Saved”
- Ευθύνη καλουμένου να διατηρήσει ό,τι περιεχόμενα είχαν οι κατάρχωρητές που αυτός θα χρησιμοποιήσει
- Όταν εγώ καλώ, ξέρω ότι οι απόγονοί μου θα διατηρήσουν τα δικά μου περιεχόμενα
- Register contents *saved* across procedure calls

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

Lifetimes of variables that span 2 or more procedure Calls

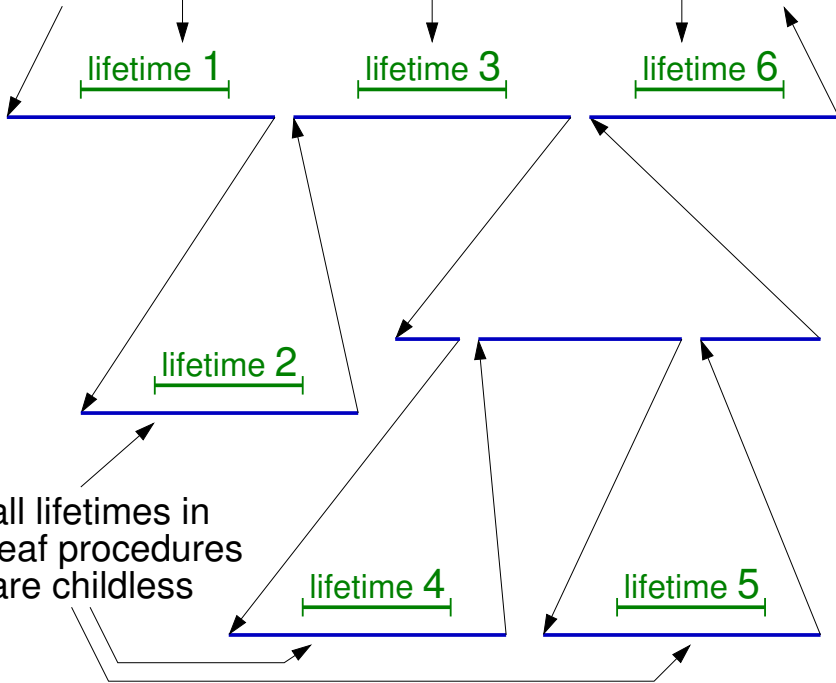


"s" (saved) register is preferable: fewer save–restores to stack

Lifetimes of variables that contain no procedure Calls

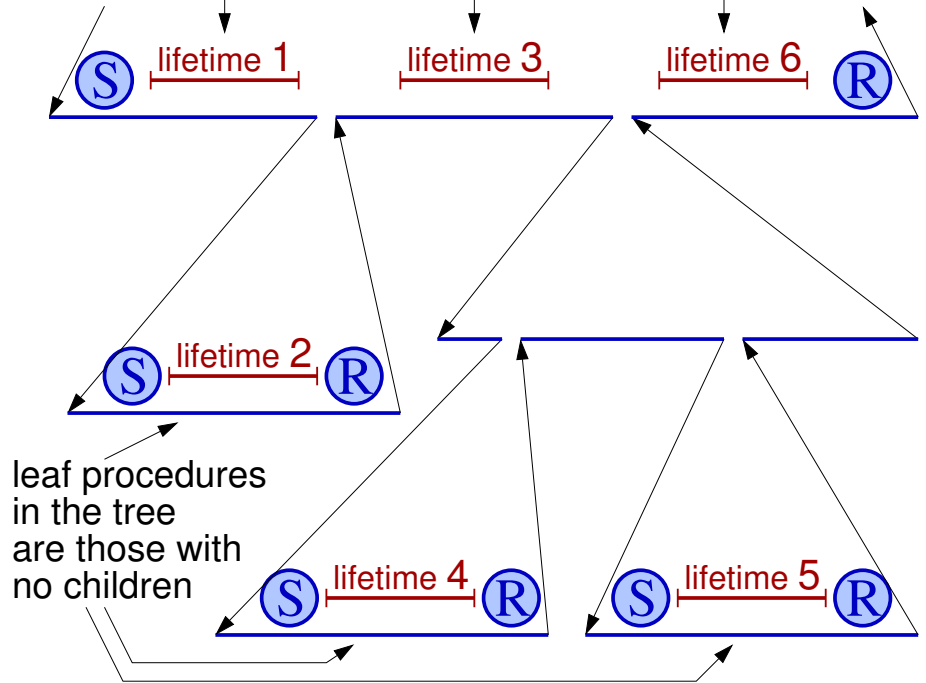
if placed within **t** registers:

three independent lifetimes (variables):
no data need to be preserved from one to the other



if placed within **S** registers:

three independent lifetimes (variables)
within one, same s register



"t" registers are preferable for childless lifetimes: no save-restore to stack 60

x0	zero	
x1	ra (return address)	"C" sp
x2	sp (stack pointer)	
x3	gp (global pointer)	
x4	tp (thread pointer)	
x5	t0 (caller saved)	
x6	t1 (caller saved)	RV "C" popular
x7	t2 (caller saved)	
x8	s0 / fp (or frame ptr)	
x9	s1 (callee saved)	
x10	a0 (1st arg/ret.val)	
x11	a1 (2nd arg/rv/tmp)	
x12	a2 (3rd arg / tmp)	
x13	a3 (4th arg / tmp)	
x14	a4 (5th arg / tmp)	
x15	a5 (6th arg / tmp)	
x16	a6 (7th arg / tmp)	
x17	a7 (8th arg / tmp)	
x18	s2 (callee saved)	
x19	s3 (callee saved)	
x20	s4 (callee saved)	
x21	s5 (callee saved)	
x22	s6 (callee saved)	
x23	s7 (callee saved)	
x24	s8 (callee saved)	
x25	s9 (callee saved)	
x26	s10 (callee saved)	
x27	s11 (callee saved)	
x28	t3 (caller saved)	
x29	t4 (caller saved)	
x30	t5 (caller saved)	
x31	t6 (caller saved)	

Συμβάσεις Χρήσης Καταχωρητών

- Σύμβαση Κλήσης Διαδικασιών
 - μεταξύ Compilers – δεν αφορά το hardware, δεν την επιβάλλει 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 για χρήση ως Temporaries
- Ο *RV32E (embedded)* έχει μόνον 16 καταχ.
- Ο *RVC (compressed)* συμπιέζει όσες εντολές χρησιμοποιούν μόνο τους **x0...2**, **x8...15**