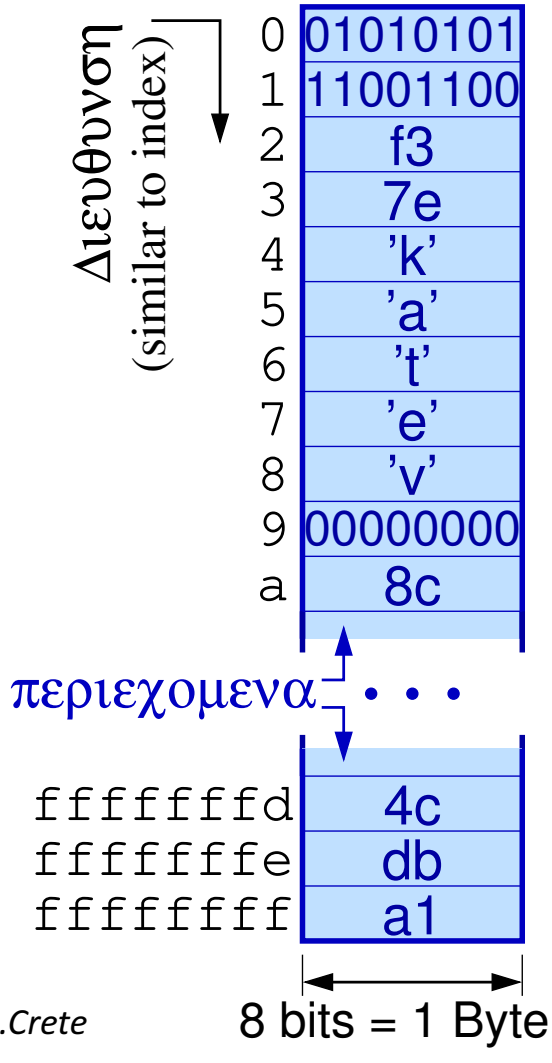
Η Μνήμη (Byte-Addressable), οι εντολές Load και Store στον RISC-V, η Διευθυνσιοδότησή τους και οι χρήσεις τους

03a (§3.1) – 22-24 Φεβ. 2021 – Μανόλης Κατεβαίνης

Byte-Addressable: κάθε Byte τη δική του Διεύθυνση

- Η Μνήμη είναι σαν ένας μεγάλος πίνακας (array) από Bytes
- Πρακτικά όλοι οι σημερινοί υπολογ. είναι *Byte-Addressable*, δηλαδή οι διευθύνσεις μνήμης αναφέρονται σε Bytes – όχι σε ολόκληρες λέξεις
  - γιά να μπορούν να επεξεργάζονται strings σε επίπεδο μεμονωμένων char
- Εδώ π.χ. 32-μπιτος επεξεργαστής
  - 32-μπιτη Διεύθ. ⇒ Μνήμη ≤ 4 GBytes
  - συνήθως: «πλάτος» επεξ. = Addr. width

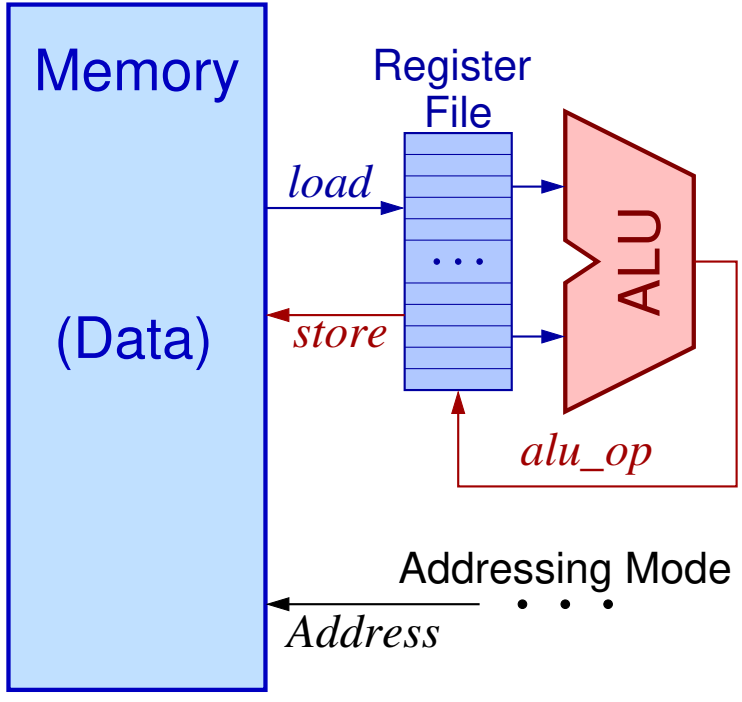


Οι εντολές load και store στις αρχιτεκτονικές RISC

- Στις αρχιτεκτονικές RISC, η Μνήμη προσπελάζεται μόνον με εντολές *Load* γιά ανάγνωση και εντολές *Store* γιά εγγραφή
- Σοβαρό θέμα:

Τρόπος Διευθυνσιοδότησης (Addressing Mode)

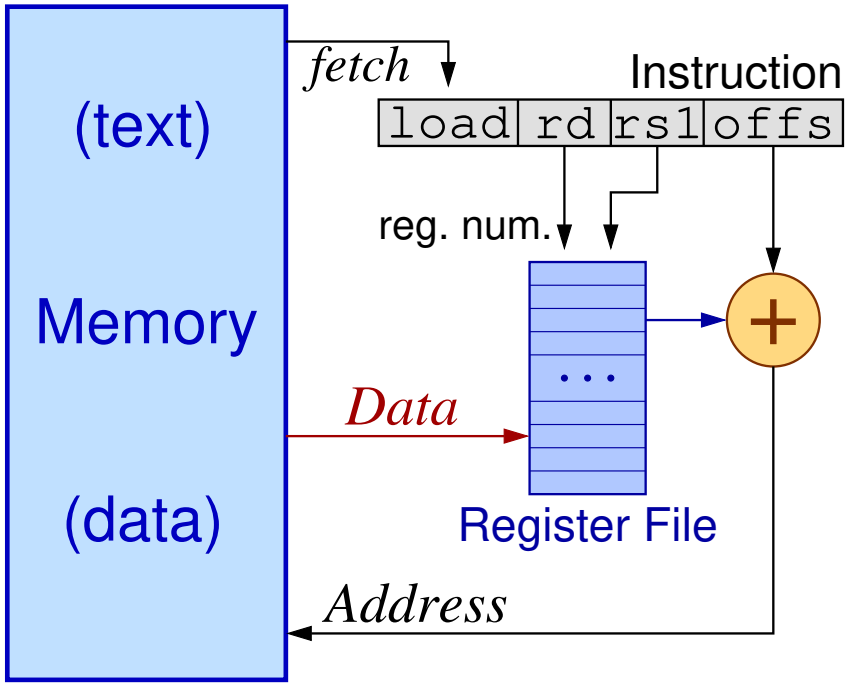
  - Γιά Δομές Δεδομένων, χρειάζεται και μεταβλητή διεύθυνση



- Αρχιτ. RISC: λίγοι, απλοί, γενικοί τρόποι διευθυνσιοδότ.
  - πάντα περιλαμβάνουν τουλάχιστον έναν καταχωρητή-pointer

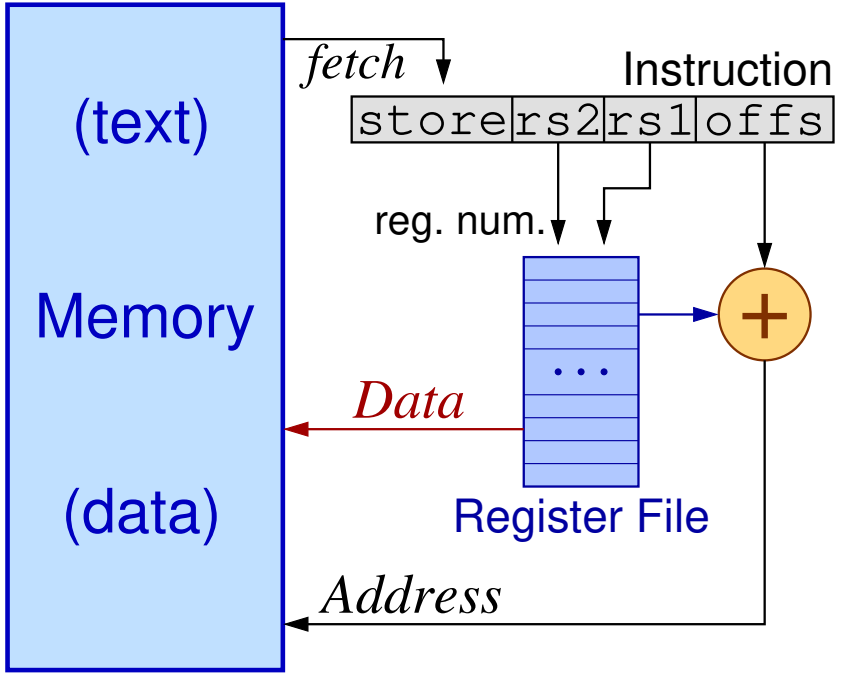
Οι εντολές Load και Store στον RISC-V

`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*

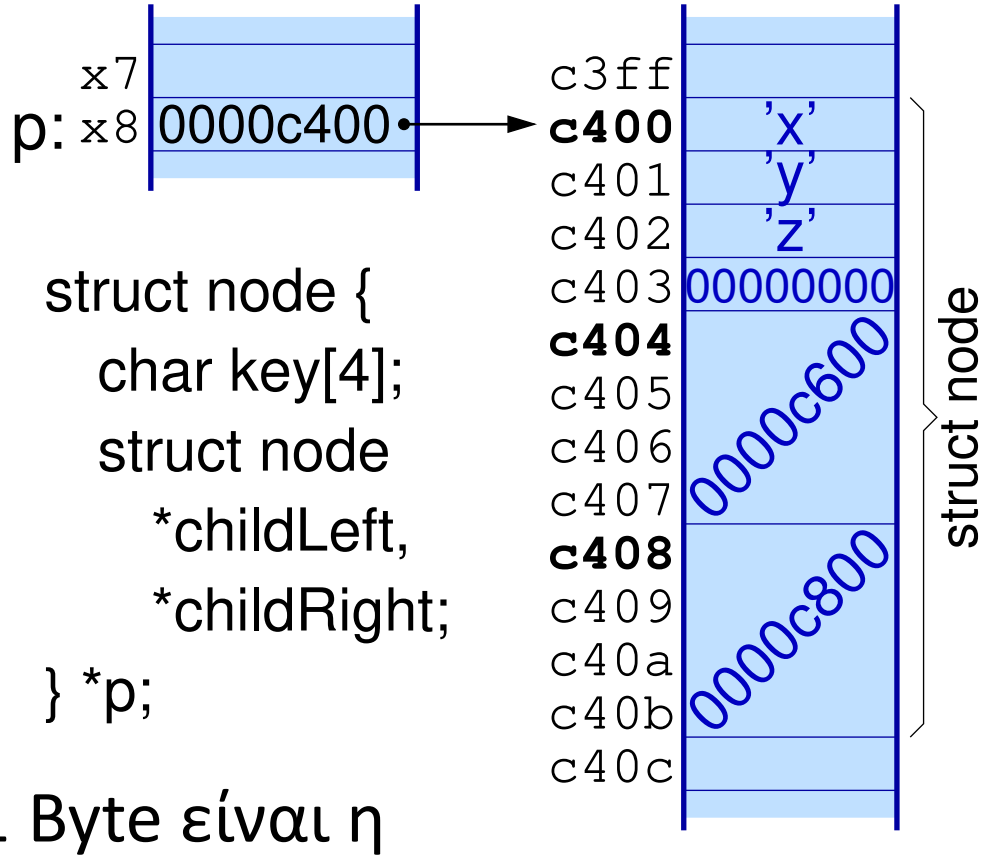
«Στενές» μεταφορές: τα υπόλοιπα bits του καταχωρητή;

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;

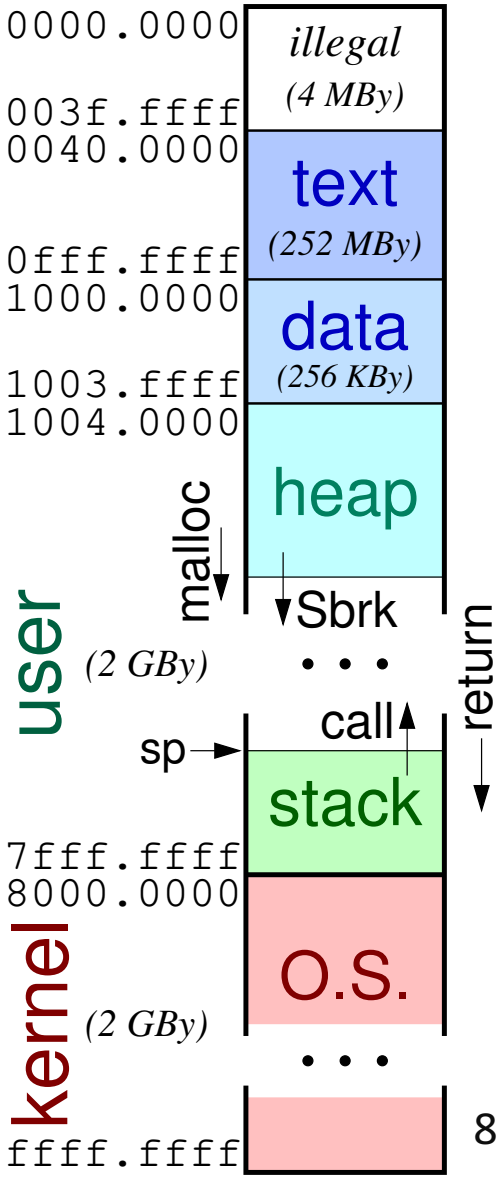


- Διεύθυνση ποσοτήτας >1 Byte είναι η διεύθ. εκείνου του Byte της που έχει την *μικρότερη* διεύθ.

Χρήσεις Μνήμης (memory layout)

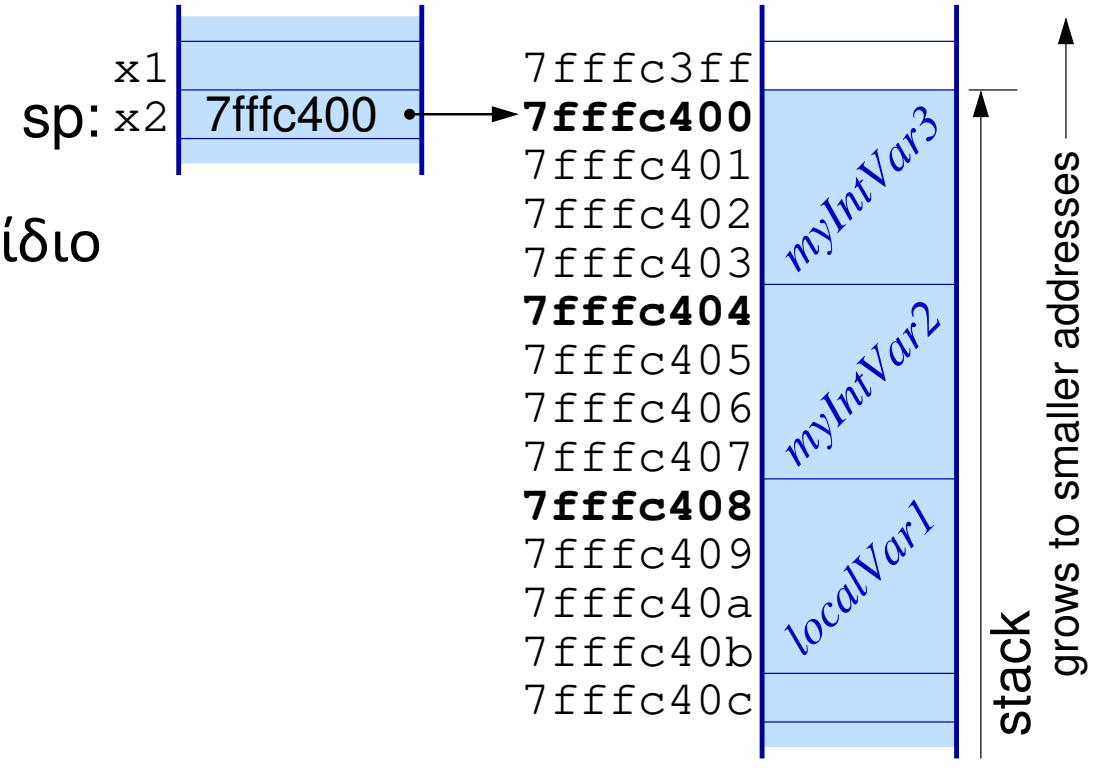
- Άνω ήμισυ: Λειτουργικό Σύστημα
- Σελίδα με “NULL pointer”: unallocated
  - σκοπός: debug NULL pointer dereference
- *Data*: statically allocated
- *Heap*: dynamically allocated
- Στοίβα & Heap μεγ. προς αντίθετες κατ.

  ⇒ ολική χρήση χώρου
- Οι διευθύνσεις δεξιά όπως στο RARS
  - αλλού αλλιώς, π.χ. 3G-1G, shared libaries,…



Χρήση 2: Προσπέλαση τοπικών μεταβλ. στη Στοίβα

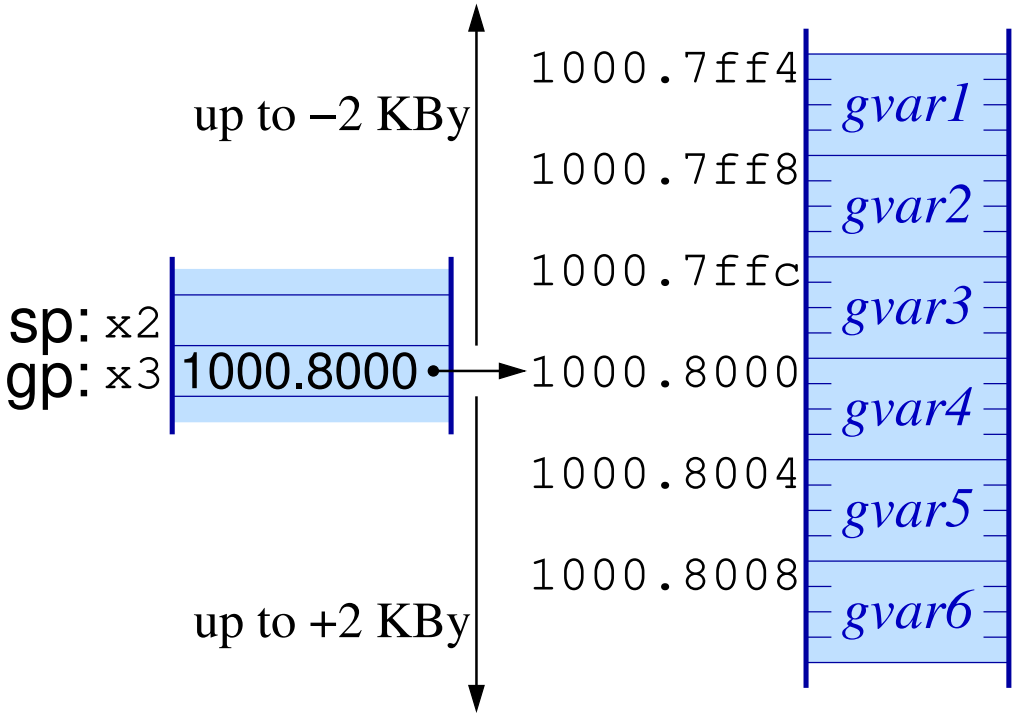
- `lw x5, 8(x2)`
- `lw x5, 8(sp)`
  - "x2" και "sp" είναι το ίδιο
  - tmp0 = localVar1;
- `sw x6, 4(sp)`
  - myIntVar2 = tmp1;
- `lw x7, 0(sp)`
  - tmp2 = myIntVar3;



"Local" variables in procedures: stack of activation frames

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

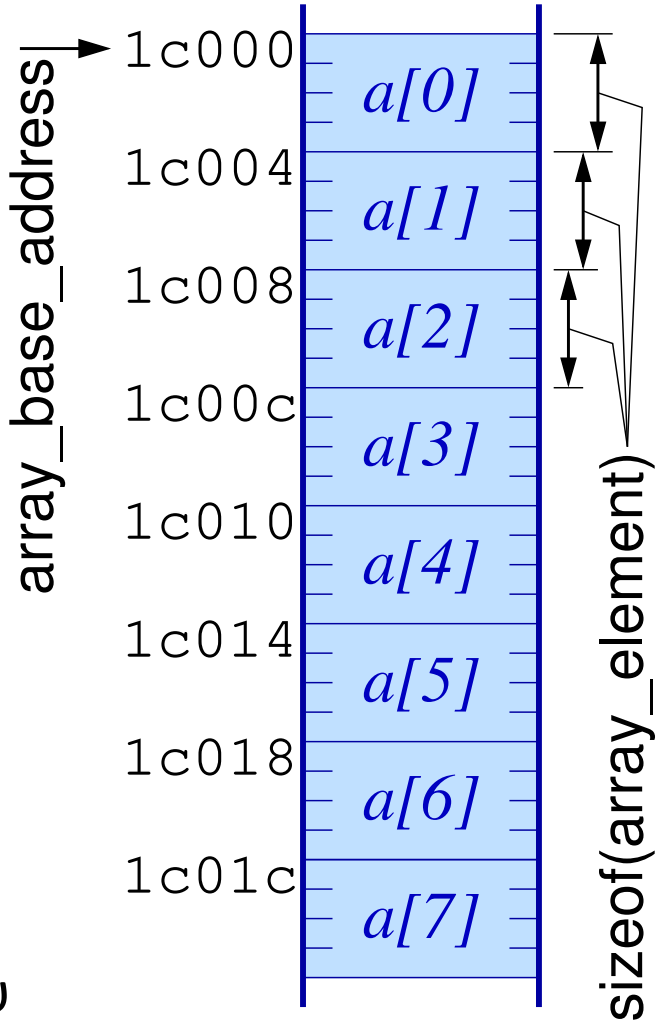
- Μεταβλητές ορισμένες εκτός διαδικασιών στη C: Στατικές, καθολικές (global)
- Χωριστά οι βαθμωτές (scalar – not arrays) εδώ, ώστε να χωρούν όλες, ει δυνατόν, σε 4 KBytes
- `x3` (**gp** – global pointer) δείχνει στη μέση του χώρου αυτού (12-bit signed offset)



- `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
  ⇒ πολλαπλασιασμός = αρ. ολίσθηση
- Όταν array of structures, κλπ.
  - κανονικός πολλαπλασιασμός, συχνά
  - Διεύθ. στοιχείου εντός Δομής i =
    Διεύθυνση Βάσης Πίνακα +
    i × sizeof(*struct*) + Offset_στοιχείου



Ένα απλοϊκό παράδειγμα πίνακα (βιβλίο p.69 / σελ.123-125)

- **A[12] = h + A[8];**
- sizeof(long long int) = 64 b = 8 By
- &(A[8]) = &(A[0]) + 8 × 8 = A + 64
- &(A[12]) = &(A[0]) + 12 × 8 = A + 96

```
ld  x5, 64(x22)
add x5, x21, x5
sd  x5, 96(x22)
```

- 64-μπιτος RISC-V
- long long int A[sz];
- `x5`: tmp
- `x21`: **h**
- `x22`: **A** = &(A[0]) (base addr. of A)

- Στην πραγματικότητα, σπανίως το index είναι σταθερά

Προσπελάσεις Πινάκων, συχνά

- Index = μεταβλητή σε καταχωρητή, sizeof(element) ≠ 1
  ⇒ απαιτείται χωριστή εντολή ολίσθησης ή πολλαπλασιασμού
- Διεύθυνση βάσης είτε σταθερά με πολλά bits είτε pointer
  ⇒ διεύθ. βάσης σε καταχωρητή
  ⇒ χωριστή εντολή: `add rTmp, rBase, rIndexSize`
  ⇒ εάν υπήρχε addr. mode $M[rs1+rs2]$ θα γλυτώναμε την add
- Εάν πίν. βαθμωτών (όχι struct) ⇒ 12-μπιτο Offset: άχρηστο

Όμως, συνήθως οι Compilers βελτιστοποιούν:

for (i=0; i<N; i++) { … a[i] … } ⇒

for (p=a; p<a+N; p++) { … *p … }

Γιατί μοναδικό Addressing mode *Offset+(rs1)* ?

- Γιά Πίνακες θα θέλαμε και *(rs1)+(rs2)* – *όμως*:
  - Γιά τις Store αυτό θα απαιτούσε 3 κατάχωρητές πηγής ⇒ κόστος
  - Όταν ο Compiler βελτιστοποιεί με pointers: περιττό
- Γιατί υποχρεωτικά μιά πρόσθεση στο δρόμο προς μνήμη;
  - Addr. mode με σκέτη (20-μπιτη;) Σταθερά;
    - μόνον γιά global scalars, αλλά γίνεται και μέσω **gp**
  - Addr. mode με σκέτο καταχωρητή;
    - γιά πίνακες/pointer-chasing χρήσιμο αλλά σπάνιο (pointers/struct)
  - Ναι μεν η απουσία πρόσθεσης θα επέτρεπε να γίνει έναν κύκλο νωρίτερα η πρόσβαση μνήμης, αλλά θα προκαλούσε και σημαντική ανομοιμορφία στην Pipeline, άρα το αποφεύγουν…