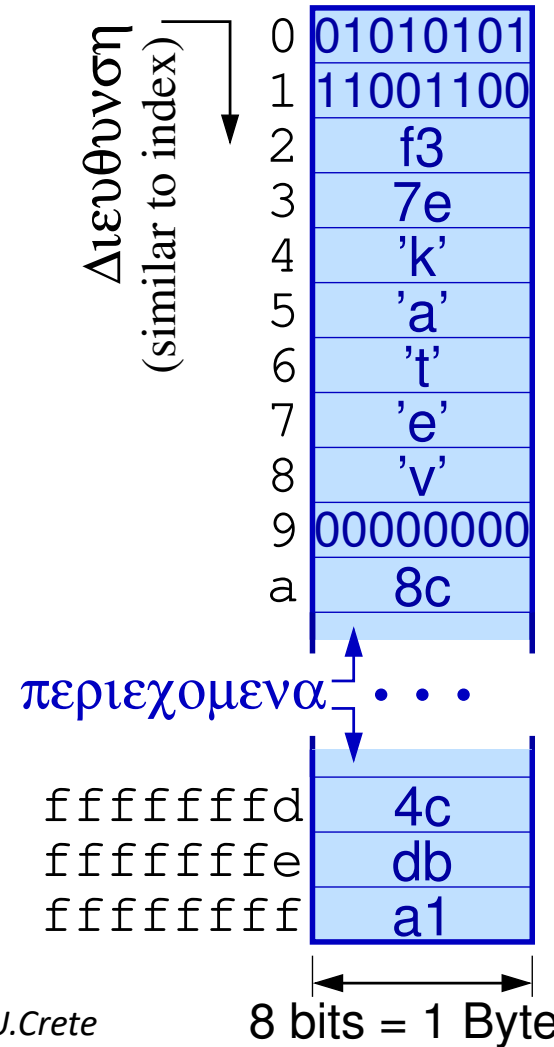


Η Μνήμη (Byte-Addressable),
οι εντολές Load και Store στον RISC-V,
η Διευθυνσιοδότησή τους και οι χρήσεις τους

03α (§3.1) – 16-18 Φεβ. 2026 – Μανόλης Κατεβαίνης

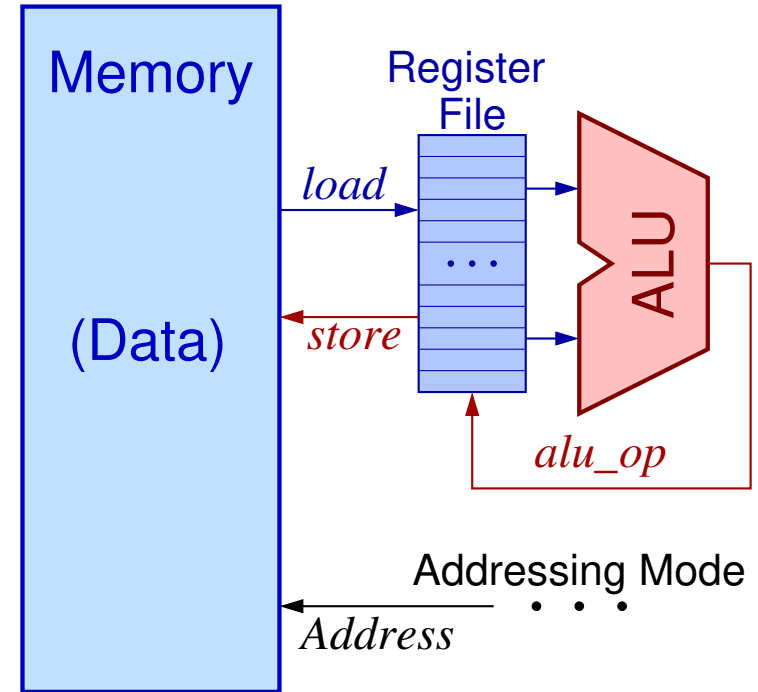
Byte-Addressable: κάθε Byte τη δική του Διεύθυνση

- 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
 - για να μπορούν να επεξεργάζονται strings σε επίπεδο μεμονωμένων char
- Εδώ π.χ. 32-μπιτος επεξεργαστής
 - 32-μπιτη Διεύθ. $\Rightarrow$ 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* για ανάγνωση και εντολές *Store* για εγγραφή
- Εντολές *Load* του RISC-V:
 - *lb* (load byte), *lbu* (lb unsigned)
 - *lh* (load half), *lhu* (lh unsigned)
 - *lw* (load word), *lwu* (lw unsigned)
 - *ld* (load double)
- Εντολές *Store* του RISC-V:
 - *sb* (store byte), *sh* (st. half), *sw* (st. word), *sd* (st. double)



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

Στον βασικό RISC-V, ***RV32***, με 32-μπιτους καταχωρητές:

- `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

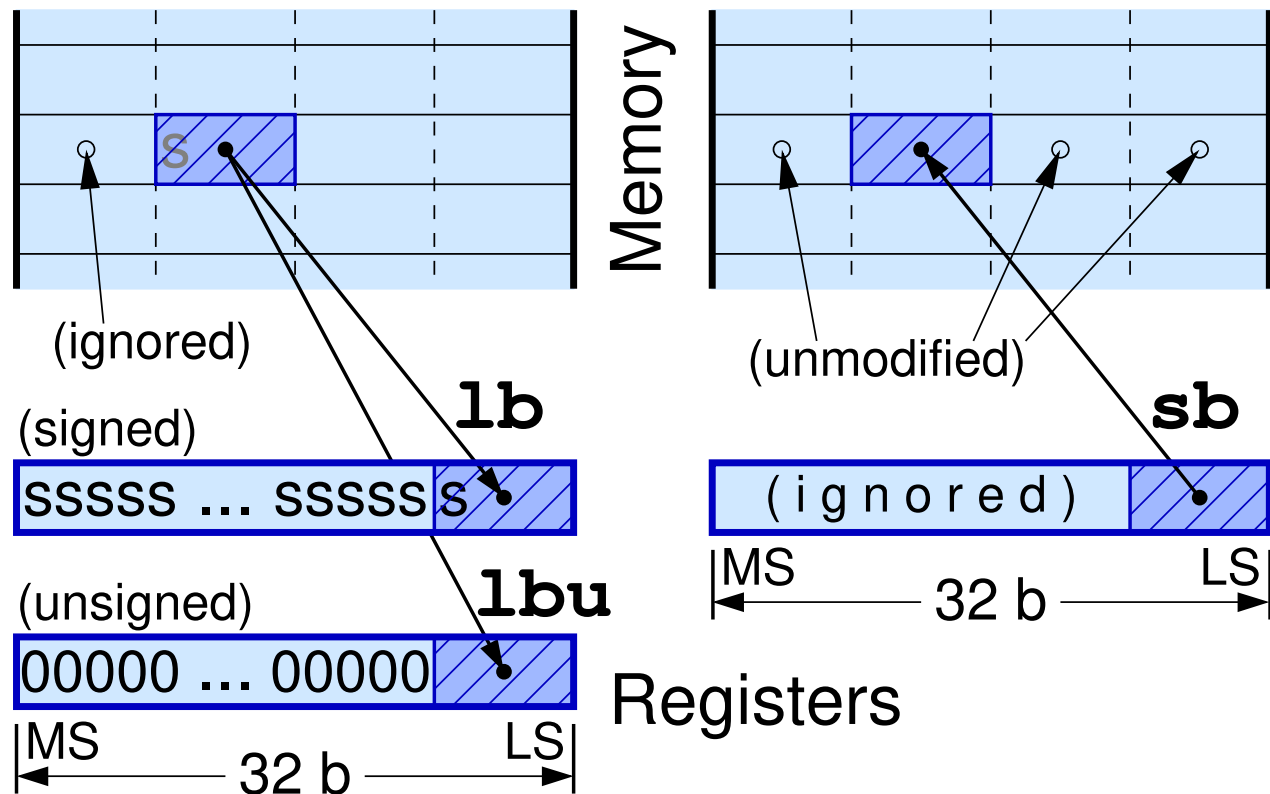
Στην 64-μπιτη επέκταση, ***RV64***, υπάρχουν και οι:

- `ld / sd` → load/store Double: διαβάζει/γράφει 8 Bytes
 - ο *RV32* δεν τις έχει διότι δεν χωράει Double σε καταχωρητή

Στην C: `int` → Word; `long long int` → Double

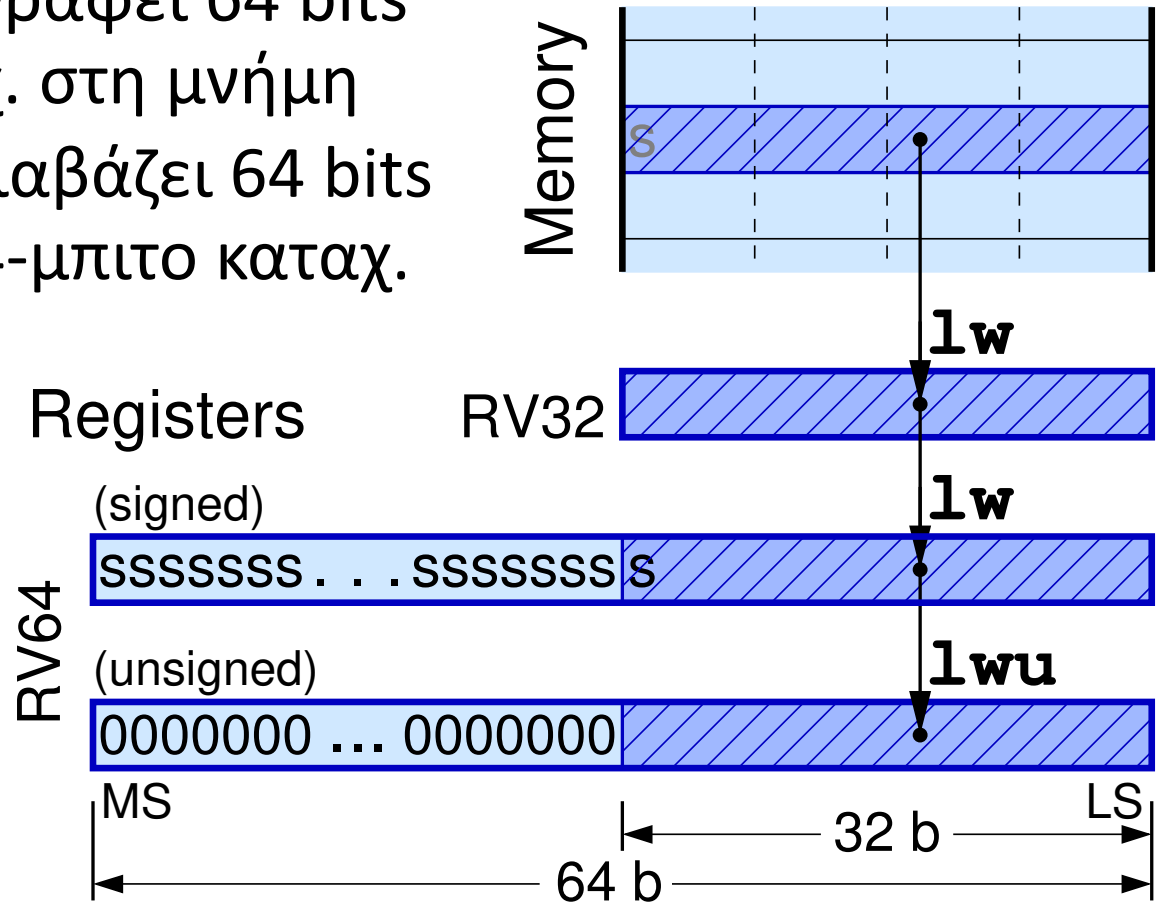
«Στενές» μεταφορές: τα υπόλοιπα bits του καταχωρητή;

- Από την πηγή κρατάμε μόνον όσα Bytes λέει η εντολή
- Στη μνήμη γράφουμε μόνον όσα Bytes λέει η εντολή
- Τον καταχωρητή, τον γράφουμε ολόκληρο:
ο καταχωρητής περιέχει πάντα μία και μόνο μία μεταβλητή (συνήθως: ακέραιο αριθμό για πρ. ALU)
- Ακέραιοι σε καταχωρητές: πάντα «δεξιά», δηλ. LS aligned (θέση $\times 2^0$)



Εντολές στον RV64 που δεν υπάρχουν στον RV32

- `sd` (store double): γράφει 64 bits από 64-μπιτο καταχ. στη μνήμη
- `ld` (load double): διαβάζει 64 bits από τη μνήμη σε 64-μπιτο καταχ.
- Η `lw` (load word), που στον RV32 είναι ίδια για signed ή unsigned, στον RV64 διαφοροποιείται:
 - `lw` → signed
 - `lwu` → unsigned



Θυμηθείτε / Σκεφτείτε:

- Σε τι μετράνε οι διευθύνσεις μνήμης;
- Τι σημαίνει 32-μπιτος ή 64-μπιτος επεξεργαστής;
- Load στενή ποσότητα:
 - γράφει όλον ή μέρος του καταχωρητή;
 - ποιο μέρος του καταχωρητή;
 - στον υπόλοιπο κτχ.;
- Store στενή ποσότητα:
 - γράφει όλη ή μέρος της λέξης μνήμης;
 - από ποιο μέρος του καταχωρητή;
 - ο υπόλοιπος καταχωρ.;

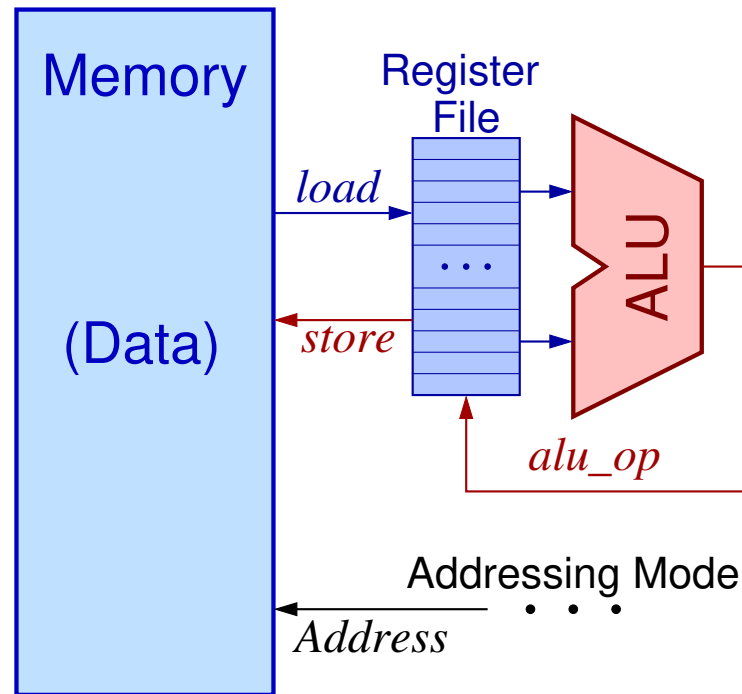
Θυμηθείτε / Σκεφτείτε:

- Σε τι μετράνε οι διευθύνσεις μνήμης; → σε Bytes
- Τι σημαίνει 32-μπιτος ή 64-μπιτος επεξεργαστής;
→ 32-bit ή 64-bit Addresses, in 32-bit ή 64-bit registers
- Load στενή ποσότητα:
 - γράφει όλον ή μέρος του καταχωρητή; → Όλον
 - ποιο μέρος του καταχωρητή; → το λιγότερο σημαντικό (LS)
 - στον υπόλοιπο κτχ.; → sign-extension ή zero-fill εάν “unsigned”
- Store στενή ποσότητα:
 - γράφει όλη ή μέρος της λέξης μνήμης; → Μέρος (όσο στενό λείει)
 - από ποιο μέρος του καταχωρητή; → το λιγότερο σημαντικό (LS)
 - ο υπόλοιπος καταχωρ.; → αγνοείται (δεν γράφεται στη μνήμη)

Addressing Modes στις αρχιτεκτονικές RISC

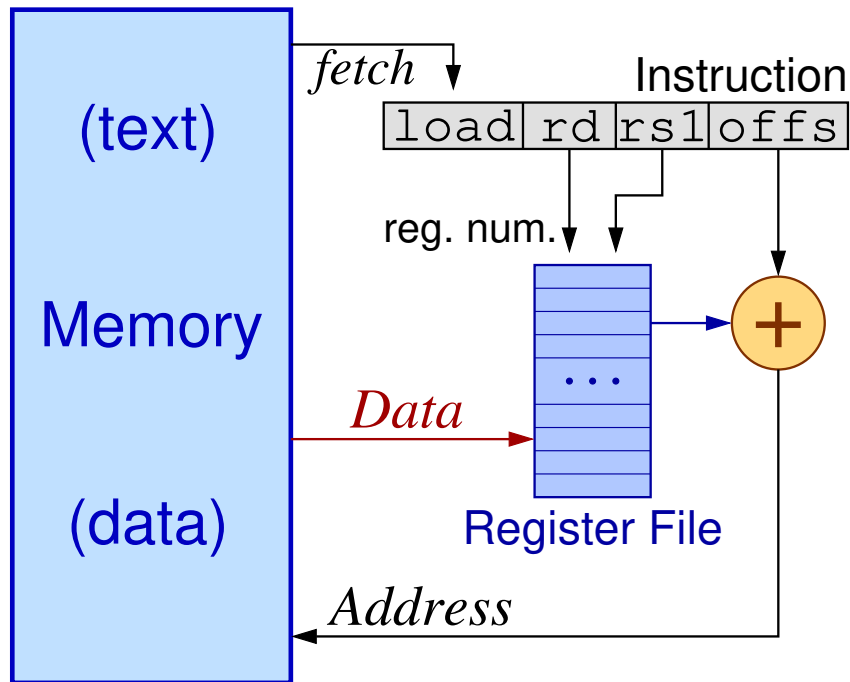
Τρόπος Διευθυνσιοδότησης:

- **CISC:** πολλά modes $\forall$ τελεστέο
 - σταθερά, καταχωρητής, @μνήμη
 - @σταθερή διεύθ. μν. (πολλά bits)
 - μέσω pointer σε καταχωρητή, με ή χωρίς offset, σταθερό ή μτβλ.
 - μέσω pointer(s) (& offst) @μνήμη?
- **RISC:** λίγα, απλά, γενικά modes
 - ALU: καταχ. ή σταθερά (λίγα bits)
 - Load/store: μέσω καταχ.: pointer ή/και αποφυγή πολλών bits
 - Υποψήφια modes: $M[Reg]$ ή $M[R1+R2]$ ή $M[Reg+Σταθερά]$
 - RISC-V: μόνον το $M[Reg+Σταθερά]$



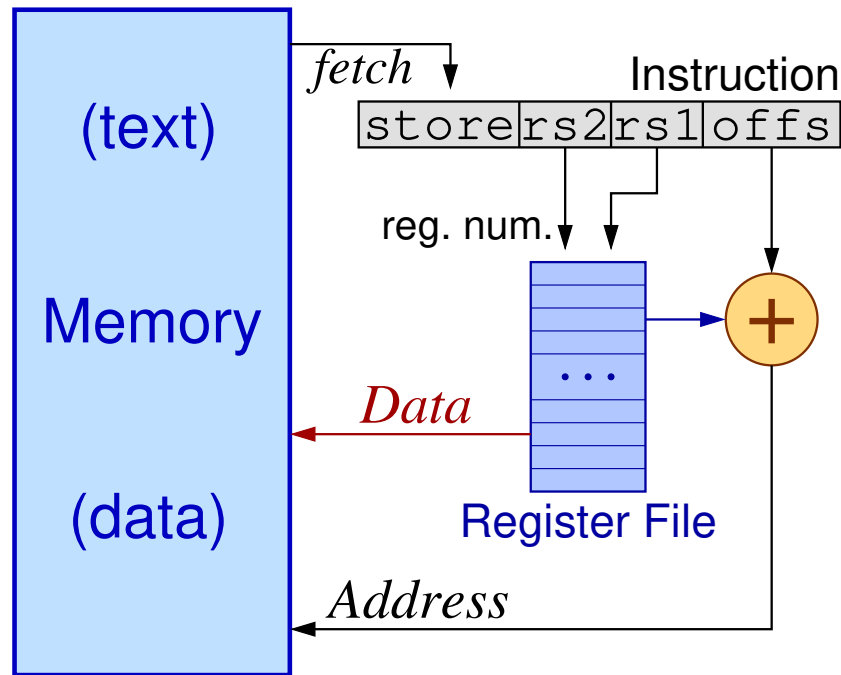
Οι εντολές Load και Store στον RISC-V

lw rd, offset(rs1)



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

sw rs2, offset(rs1)

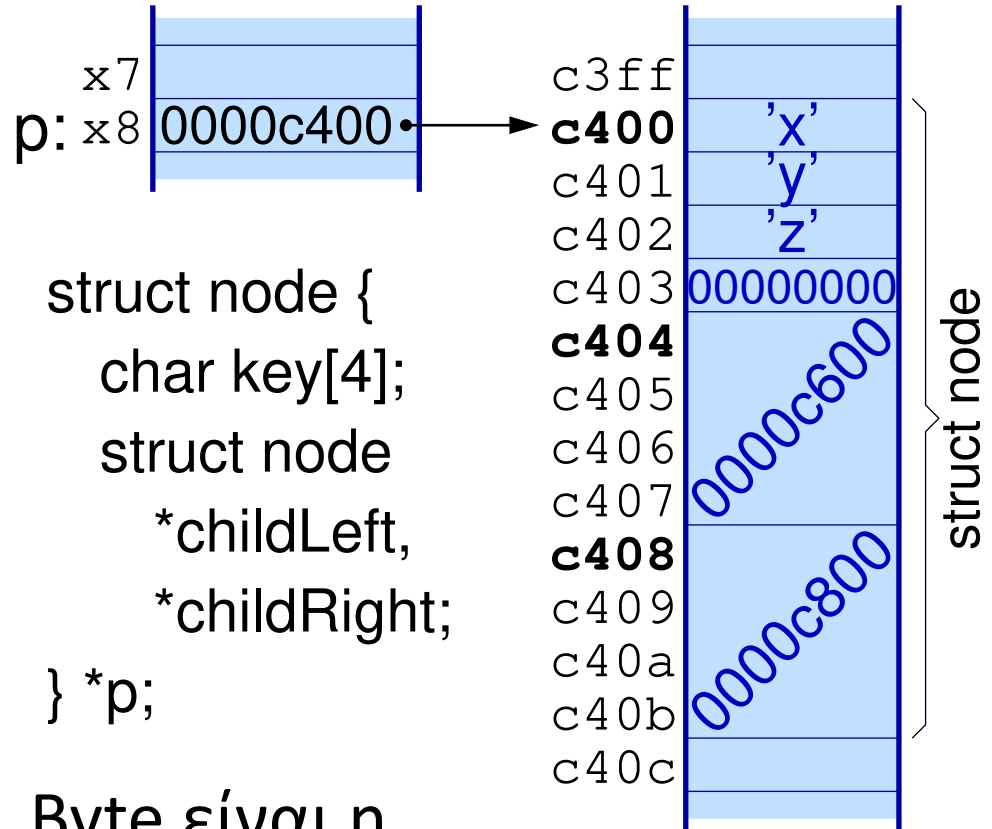


$$rs2 \rightarrow M[offset + (rs1)]$$

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

Χρήση 1: Προσπέλαση Στοιχείων Δομής μέσω Pointer

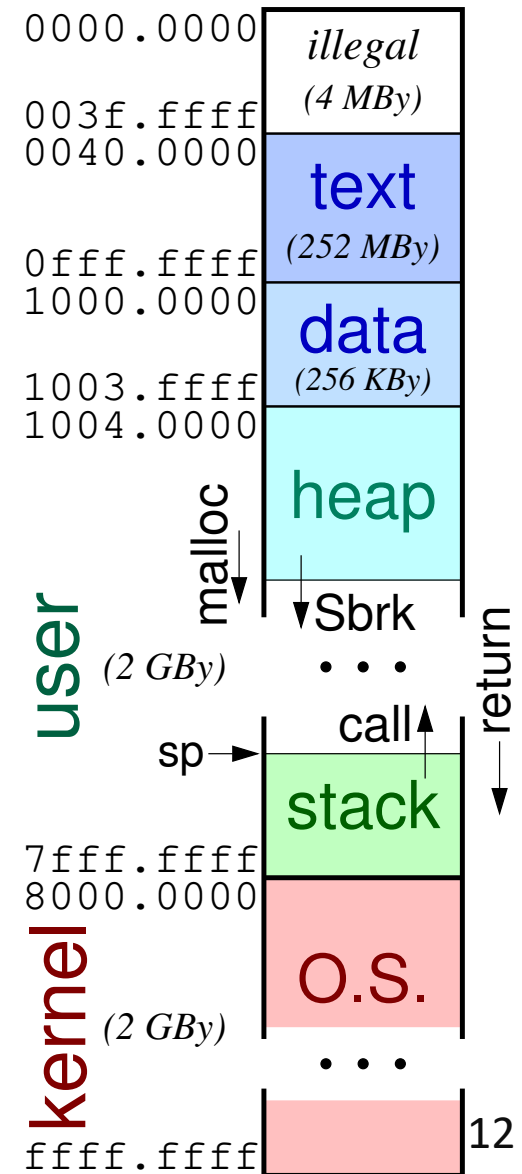
- `lbu x5, 0 (x8)`
 - `tmp0 = p->key[0];`
- `lbu x6, 1 (x8)`
 - `tmp1 = p->key[1];`
- `lw x7, 4 (x8)`
 - `tmp2 = p->childLeft;`
- `lw x8, 8 (x8)`
 - `p = p->childRight;`



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

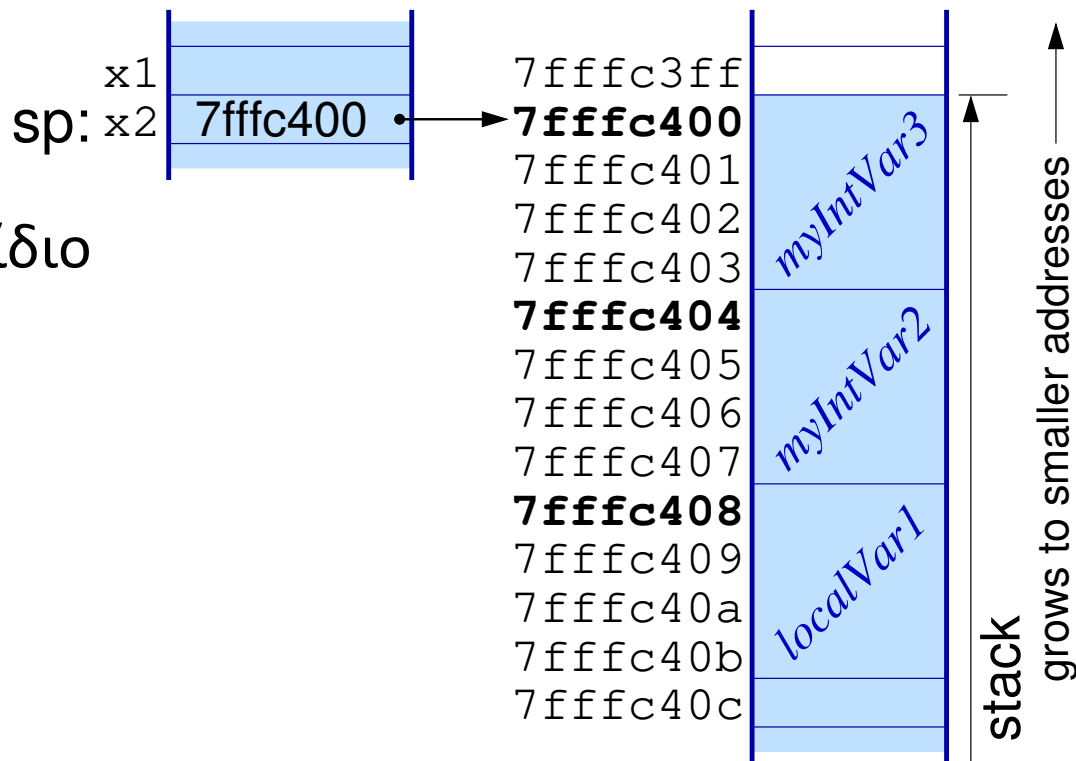
Χρήσεις Μνήμης (memory layout)

- 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
 - αλλιώς, π.χ. 3G-1G, shared libraries,...



Χρήση 2: Προσπέλαση τοπικών μεταβλ. στη Στοίβα

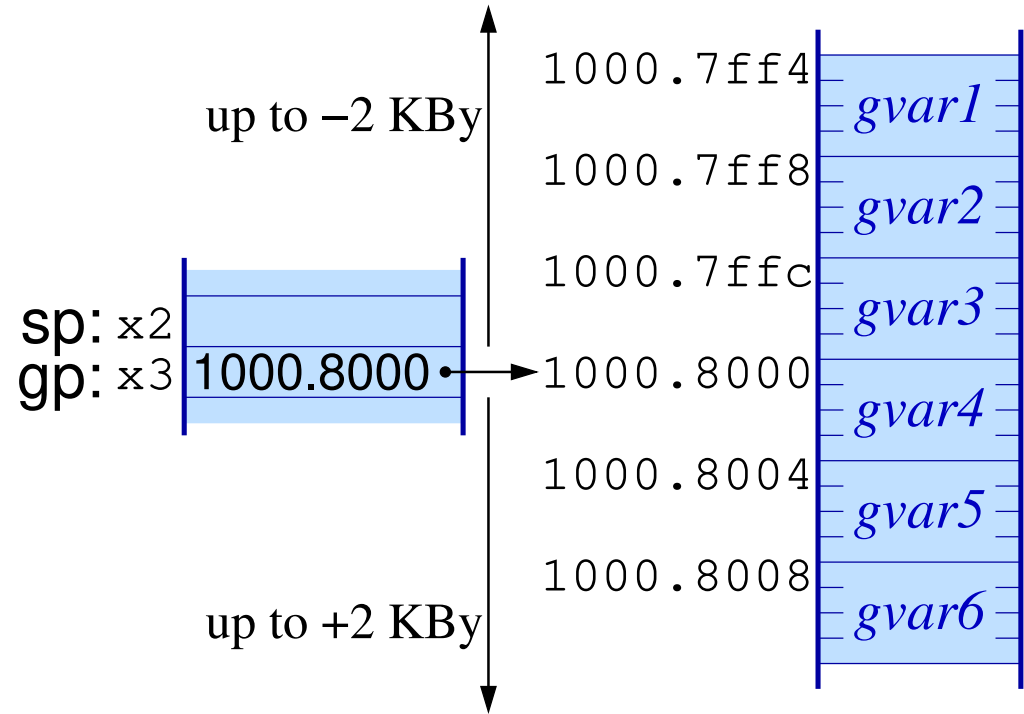
- `lw x5, 8(x2)`
- `lw x5, 8(sp)`
 - “x2” και “sp” είναι το ίδιο
 - `tmp0 = localVar1;`
- `sw x6, 4(sp)`
 - `myIntVar2 = tmp1;`
- `lw x7, 0(sp)`
 - `tmp2 = myIntVar3;`



“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;

Θυμηθείτε / Σκεφτείτε:

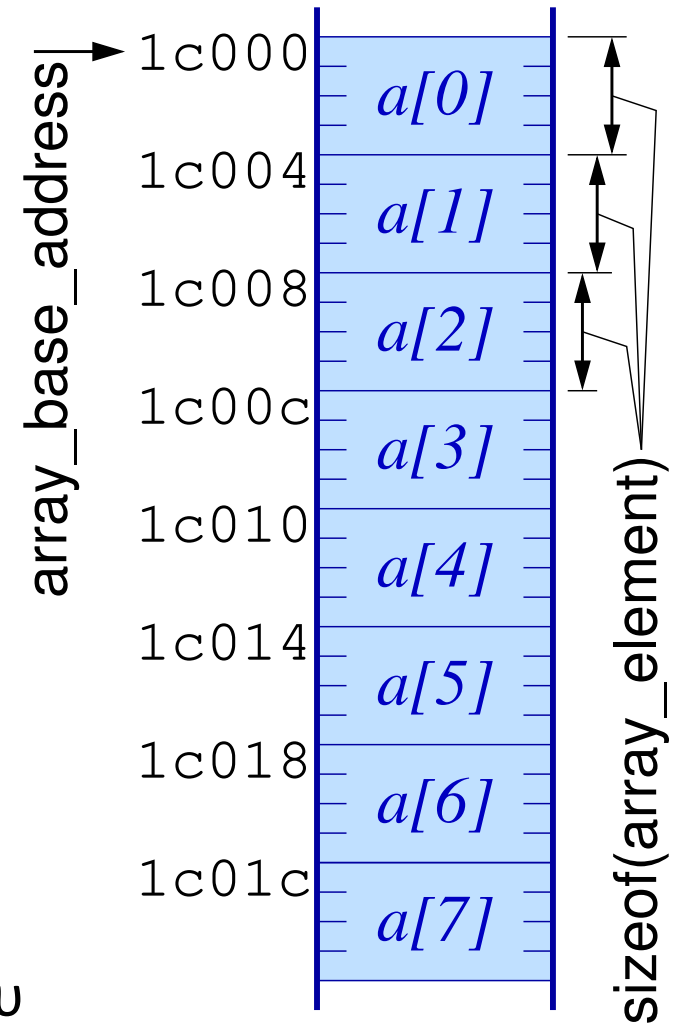
- Ποιο (μοναδικό) Addressing mode?
- Τι είναι η Σταθερά (Offset)?
- `struct xyz *p; p->member`
- Local variable in stack
- Global variable in data segment

Θυμηθείτε / Σκεφτείτε:

- Ποιο (μοναδικό) Addressing mode? → Register + Offset
- Τι είναι η Σταθερά (Offset)? → 12-bit, Signed
- `struct xyz *p; p->member` → register p + Offset
- Local variable in stack → stack pointer ($sp \equiv x2$) + Offset
- Global variable in data segment
→ global pointer ($gp \equiv x3$) $\pm$ Offset

Πίνακες (arrays)

- Διεύθυνση Στοιχείου i =
Διεύθυνση Βάσης +
 $i \times \text{sizeof}(\text{element})$
- Όταν βαθμωτά στοιχεία, συνήθ.
 $\text{sizeof}(\text{element})$ δύναμη του 2
 $\Rightarrow$ πολλαπλασιασμός = αρ. ολίσθηση
- Όταν array of structures, κλπ.
 - κανονικός πολλαπλασιασμός, συχνά
 - Διεύθ. στοιχείου εντός Δομής i =
Διεύθυνση Βάσης Πίνακα +
 $i \times \text{sizeof}(\text{struct}) + \text{Offset_στοιχείου}$



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

- $A[12] = h + A[8];$
- $\text{sizeof}(\text{long long int}) = 64 \text{ b} = 8 \text{ B}$
- $\&(A[8]) = \&(A[0]) + 8 \times 8 = A + 64$
- $\&(A[12]) = \&(A[0]) + 12 \times 8 = A + 96$

```
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, $\text{sizeof}(\text{element}) \neq 1$
⇒ 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 $Offset+(rs1)$?

- Για Πίνακες θα θέλαμε και $(rs1)+(rs2)$ – όμως:
 - Για τις Store αυτό θα απαιτούσε 3 κατάχωρητές πηγής $\Rightarrow$ κόστος
 - Όταν ο Compiler βελτιστοποιεί με pointers: περιττό
- Γιατί υποχρεωτικά μιά πρόσθεση στο δρόμο προς μνήμη;
 - Addr. mode με σκέτη (20-μπιτη;) Σταθερά;
 - μόνον για global scalars, αλλά γίνεται και μέσω **gp**
 - Addr. mode με σκέτο καταχωρητή;
 - για 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 ανομοιομορφία στην Pipeline, άρα το αποφεύγουν...