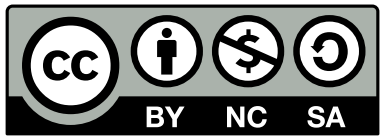
0110 - Adressage

Tableaux, pointeurs, pile et routines

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur
v241



Rappels

La mémoire

- Segments: `.text` et `.data`
- Directives de données: `.dword`, `.string`, `.space`, etc.
- Instruction d'accès mémoire: `ld`, `sd`, etc.

Les calculs

- Arithmétique et logique: `add`, `or`, `sll`, `mul`, etc.
- Structure de contrôles: `beq`, `j`, etc.
- Programmes de qualité: sans bogue, lisibles, voire efficaces

Plan

Tableaux et matrices

Tableaux: rappel

- Séquence d'éléments de taille fixe en mémoire
- L'adresse du tableau est l'adresse du premier éléments
- 100% programmatif:
  Tout est de la responsabilité du programme assembleur
  - Déclaration des données
  - Accès aux éléments

Accès aux éléments d'un tableau

Pointeur et index

- Un **pointeur** `s0` sur le tableau +
  Un **index** `s1` pour la case à traiter
  - Nécessite une multiplication `mul` (ou un décalage `sll`)
  - Permet Un accès direct aux éléments

Pointeur + décalage

- Un **pointeur** `s0` sur le tableau +
  Un **décalage** `s1` en octet pour la case à traiter
  - `la`: initialiser l'adresse du tableau `s0`
  - `add`: calculer l'adresse de l'élément `add t0, s0, s1`
  - `addi`: bouger le décalage `s1` de la taille d'un élément

Pointeur seul

- Un **pointeur** `s0` sur la case courante que l'on bouge directement
  - `la`: initialiser le pointeur au début du tableau
  - `addi`: bouger le pointeur de la taille d'un élément

Exercice: Rappel sur les tableaux

Écrire un programme `tri.s`

- Lire 10 nombres, les trier dans l'ordre croissant, puis les afficher
- Pseudocode du tri à bulle naïf

```
// Trier tableau t de n cases
for(i=0; i<n; i++)
  for(j=1; j<n; j++)
    if(t[j-1]>t[j])
      permuter(t, j-1, j);
```

Tableaux à deux dimensions

Matrices

- $n \times m$: n lignes et m colonnes
- (i, j) désigne la case de la ligne i et colonne j
- On indice à partir de 0 pour simplifier l'arithmétique
- Les éléments ont tous la même taille s

Exemple: matrice 2×3

```
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
```

Idée 1 : Tableaux de tableaux

Chaque ligne l

- Est un tableau de m éléments de taille s
- A une taille homogène de $m \times s$
- L'élément numéro j se trouve à l'adresse $l + j \times s$

La matrice M

- Est un tableau de n lignes (un tableau de tableau)
- Donc de taille $n \times (m \times s)$
- La ligne numéro i se trouve à l'adresse $M + i \times (m \times s)$
- La case (i, j) se trouve donc à l'adresse
  $M + i \times m \times s + j \times s = M + (i \times m + j) \times s$

Idée analogue: un grand tableau à une dimension

- Travailler avec un tableau M de $n \times m$ cases
- La case (i, j) se trouve à l'adresse $M + (i \times m + j) \times s$

Tableaux de tableaux

Pièges

- Tests de bornes de lignes et de colonnes

Exercice: programme `matrice.s`

```
.data
mat:    .word 11, 12, 13
        .word 21, 22, 23
        .eqv matN, 2 # Nombre de lignes
        .eqv matM, 3 # nombre de colonnes
```

- Afficher la matrice `mat`
- Bonus: Afficher sa transposée (en lab)

Idée 2 : Tableau d'adresses

Chaque ligne l

- Est un tableau indépendant
- L'élément numéro j se trouve à l'adresse $l + j \times s$

La matrice M

- Est un tableau d'adresses (**pointeurs**)
- La taille d'une adresse est a (4 octets en RV32, 8 en RV64)
- La ligne numéro i se trouve à l'adresse $mem[M + i \times a]$
- La case (i, j) se trouve à l'adresse $mem[M + i \times a] + j \times s$

Tailles d'architectures (RV32 vs. RV64)

Prendre correctement en compte la taille des pointeurs

- RV32: `.word`, `lw`, `sw`, `addi 4`, `slli 2`
- RV64: `.dword`, `ld`, `sd`, `addi 8`, `slli 3`

Exercice: tableau d'adresses

Faire un programme `matrice2.s`

```
ligne1: .word 11, 12, 13
ligne2: .word 21, 22, 23
mat:    .dword ligne1, ligne2
        .eqv matN, 2 # Nombre de lignes
        .eqv matM, 3 # nombre de colonnes
```

- Afficher la matrice `mat`
- Bonus: Afficher sa transposée (en lab)

Branchement indirect

Branchement indirect

Pseudoinstruction `jr rs1` (rappel)

- Branche à l'adresse dans le registre `rs1`
- Pseudoinstruction de `jalr rd, rs1` (sauve l'adresse de retour)
- L'adresse de branchement peut être stockée, calculée, mise-à-jour, etc.

Intérêt

- Pointeur de routine
- Aiguillage (instruction switch)
- Implémentation de langages de haut niveau (objet, fonctionnel, etc.)
- Implémentation d'interpréteurs efficaces

Exemple RISC-V

Qu'affiche ce programme ?

```
li a0, 421
auipc s0, 0
addi s0, s0, 16
jr s0
li a0, 42
li a7, 1 # PrintInt
ecall
```

Aiguillage

Instruction *switch* (Java, C, C++)

- Branche à partir de la valeur d'une variable
- Plusieurs cible possibles

Implémentations possibles

- Série de `if`/`else if` $O(n)$
- Recherche binaire: `if`/`else if` imbriqués $O(log(n))$
- Table de branchements: branchement indirect $O(1)$

Table de branchements (*branch table*)

- Une table de pointeurs vers du code machine
  - Attention à la taille des pointeurs
- Instruction de branchement indirect
- Avantage: temps **constant** indépendant du nombre de branches

Exercice

Écrire un programme `menu.s`

- Affiche un menu fonctionnel (mais qui ne fait rien)
- 4 options numérotés de 1 à 4
- Implémenter avec une table de branchement

```
menuStr:
    .ascii "******************\n"
    .ascii "*    M E N U     *\n"
    .ascii "* [1] raz         *\n"
    .ascii "* [2] ajouter     *\n"
    .ascii "* [3] soustraire  *\n"
    .ascii "* [4] quitter     *\n"
    .ascii "******************\n"
    .ascii "Votre choix: \0"
```

Appels et retours de routines

Rappel: Routines

- Synonymes: sous-programmes, fonctions, procédures, etc.
- Encapsuler du code pour le réutiliser
- Approche: un branchement avec mémoire
  - On branche à une adresse
  - Mais on se souvient de l'adresse de retour

Routine en RISC-V

- Pseudo-instructions `jal` (ou `call`) et `ret`
  - Instructions `jal` et `jalr` en vrai
- **Convention d'appel** de l'ABI (*Application Binary Interface*)
  - `a0` à `a7`: arguments et retours
  - `a0` à `a7` et `t0` à `t6`: potentiellement écrasés par la routine
  - `ra`: registre d'adresse de retour (`x1`)

Exemple

```
    li a0, 42
    jal inc # jal ra, inc
    call printInt
    li a0, 0
    call exit

# routine inc: incrémente a0
# Pas très interessante, c'est un exemple
inc:
    addi a0, a0, 1
    ret # jalr x0, ra, 0
```

Appel et retour RISC-V

Appel local

- `jal label` (pseudoinstruction = `jal ra, label`)
- Branchement **direct** à une étiquette `label`
  - Contrainte: `label` doit être à moins de 1Mo de distance
- Avec mémoire: l'adresse de l'instruction suivante est sauvegardée
  - On sait où revenir
  - Par « défaut » dans le registre `ra` (*return address*)
  - Note: ne pas perdre ou écraser `ra` sinon on ne peut pas revenir

Retour

- Retour: `ret` (*return*, pseudoinstruction)
- Branchement **indirect** à l'adresse stockée dans `ra`
  Vraisemblablement sauvée par un `jal` précédent
- = `jr ra` (branche à `ra` sans rien sauvegarder)
- (en vrai: `jalr x0, ra, 0`)

Appel et retour RISC-V

Appel lointain

Appel de bibliothèque: `call label` (pseudoinstruction)

- Fonctionne même si `label` est à plus de 1Mo de distance
- Pseudoinstruction: calcule l'adresse à l'exécution (utilise `t1`)
  $\rightarrow$ Branchement **indirect**
- `auipc` puis `jalr`

Note: le retour est le même, avec `ret`

Branchement lointain

Appel de bibliothèque: `tail label` (pseudoinstruction)

- Fonctionne comme un `j` lointain
- Ne sauvegarde pas l'adresse de retour
- Pseudoinstruction: `auipc` puis `jr` (utilise `t1`)
  $\rightarrow$ Branchement **indirect**

Pile d'exécution

Section: pile d'exécution

- Synonyme: **pile**, *call stack*
- Pour enregistrer de l'information sur les routines actives

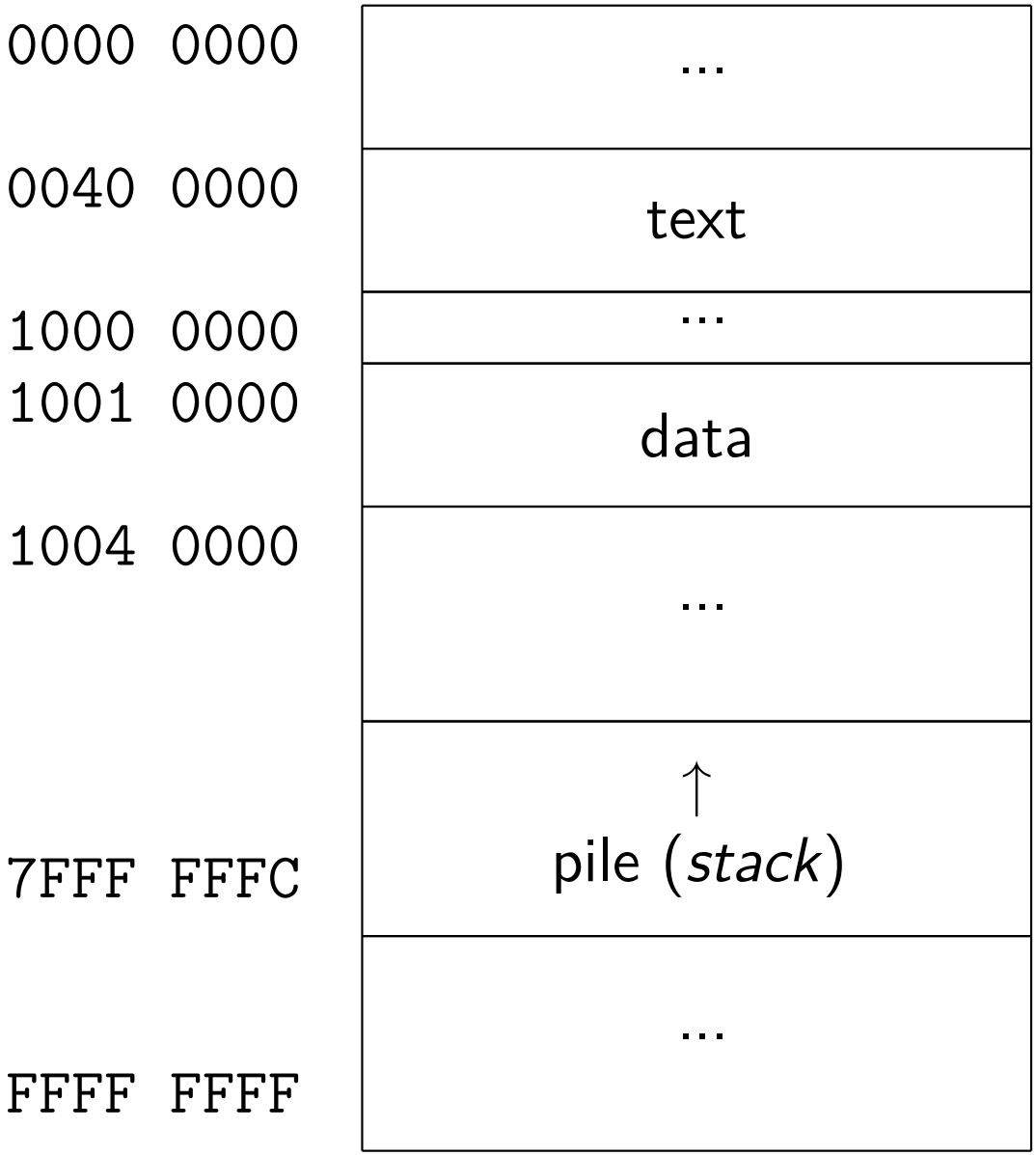
Section

- Section **dynamique** allouée par l'environnement d'exécution
- Dans des adresses hautes (loin de `.text` et `.data`)
- **Sommet de pile** pointé par le registre de pile `sp` (*stack pointer*)

Zone mémoire pour allocation dynamiques

- la pile croit quand `sp` décrémente (traditionnellement)
- Les adresses égales ou supérieures à `sp` sont utilisées
- Les adresses strictement inférieures à `sp` sont libre
- Réserver pour soi: décrémenter `sp`
- Libérer quand on a fini: incrémenter `sp`

Pile en RARS


0000 0000
...
0040 0000
text
1000 0000
...
1001 0000
data
1004 0000
...
7FFF FFFC
pile (stack)
...
FFFF FFFF

Pile en RISC-V

Convention RISC-V (ABI)

- La pile est une **convention**
- Le registre `sp` est général: `x2`

Pas d'instructions ou pseudoinstructions dédiée

- Utiliser les instructions générales
- `addi` pour incrémenter et décrémenter
- `load` et `store`
  - `sp` comme registre de base
  - Un décalage positif pour les accès

```
addi sp, sp, -8 # Alloue 8 octets dynamiquement
sd s0, 0(sp)    # Sauve `s0` sur la pile
ld s0, 0(sp)    # Charge `s0` depuis la pile
addi sp, sp, 8  # Restaure la pile (rend 8 octets)
```

Piles et routines en RISC-V

- Usage principal de la pile: sauvegarder les registres entre appels
- Sauver `s0` à `s11` si utilisés dans la routine
- Sauver aussi `ra` si appels de sous-routines dans la routine
- Pas besoin de sauver `sp` car les `addi` s'annulent

```
foo:    # Routine foo
    # Prologue
    addi sp, sp, -16 # Réserve 2 registres 64 bits
    sd s0, 0(sp)     # Sauve s0
    sd s1, 8(sp)     # Sauve s1
    # Reste de la routine qui utilise s0 et s1...
    # Épilogue
    ld s1, 8(sp)     # Restaure s1
    ld s0, 0(sp)     # Restaure s0
    addi sp, sp, 16  # Libère l'espace réservé
    ret
```

Bonnes pratiques

Prologues et épilogues

- Gérer la pile une fois au début de la routine: **prologue**
- Et une fois à la fin de la routine: **épilogue**
  - L'épilogue contient aussi le `ret`
  - Un seul épilogue même en cas de retours multiples
- On trouve parfois « *périlogue* » pour désigner les deux

Sauvegarde des registres

- Attention, la taille des registres varie entre RV32 et RV64
- Allouer le bon nombre d'octets dans le prologue
- Libérer le **même** nombre d'octets dans l'épilogue
- Sauvegarder et restaurer les registres aux bonnes **places**
- Et aux bonnes **tailles**
- Ne sauvegarder que les registres utilisés

Exercice: trouver les bogues (RV64)

```
foo:
    # Prologue
    addi sp, sp, 20
    sw s0, 0(sp)
    sw s1, 8(sp)
    sw a2, 16(sp)
    # Corps...
    # Épilogue
    addi sp, sp, -20
    lw s0, 0(sp)
    lw s0, 4(sp)
    lw a2, 8(sp)
```

Exercice: solution

- `sw` et `lw` utilisé au lieu de `sd` et `ld` (RV64)
- `addi`
  - mauvaises valeurs: 20 au lieu de 24
  - mauvais signe: empiler→soustraire, dépiler→additionner
  - mal placé à l'épilogue: doit être après les *load*
  - mauvais décalages à l'épilogue (on dirait du 32 bits)
- Fautes de frappe?
  - `a2` au lieu de `s0` ? (cf. disposition clavier)
  - à l'épilogue: deux fois `s0`, mais `s1` oublié
- Il maque le `ret`

Exercice

Écrire un programme `inc2`

- Écrire une routine `inc2` qui appelle 2 fois `inc`
- Afficher `inc2(42)`

Appels et pile en x86

- `sp` est un registre spécial (non général)
- Des instructions accèdent ou modifient directement `sp`
  - `call`, `ret`, `push`, `pop`, `enter`, `leave`, etc.
- Instruction `call`
  - L'adresse de retour est mise sur la pile et non dans un registre
  - Un `call` x86 coute parfois un peu plus cher
- Instruction `ret`
  - Dépile l'adresse de retour
  - Un `ret` x86 coûte parfois un peu plus cher
- Moins de registres disponibles donc besoin de plus de pile
  - Variables locales
  - Passage d'arguments sur la pile
  - Une autre fois

Arguments des programmes

Arguments des programmes

RARS

- *Settings>Program arguments provided to program*
- `a0` (`argc`)
  - `a0` initialisé au nombre d'arguments de la ligne de commande
- `a1` (`argv`)
  - `a1` initialisé avec l'adresse d'un tableau `argv`
  - Le tableau `argv` est de taille `argc`
  - Et contient des pointeurs sur des chaînes de caractères
  - Les chaines sont terminés par `'\0'` (classique)
  - Bogue RARS1.6: en mode 64 bits, les pointeurs dans `argv` sont quand même sur 32 bits
    - Lire avec `lw` et décaler de 4
    - Corrigé dans la version RARS du cours

Arguments des programmes Linux

Convention C (`main`). RARS s'en est inspiré

- Le point d'entrée du programme est une routine appelée `main`
- `argc` est le premier argument (`a0` selon l'ABI RISC-V)
- `argv` est le second argument (`a1` selon l'ABI RISC-V)
  - `argv[0]` est le nom du programme
  - Le premier argument de la ligne de commande est `argv[1]`

Linux sec (`_start`)

- ABI spéciale du noyaux Linux
- Le point d'entrée s'appelle `_start` (habituellement)
- `argc` au sommet de pile: `0(sp)`
- Les éléments d'`argv` directement en dessous
- Traditionnellement `_start` se charge d'appeler `main`
  - Avec les bons arguments
  - Plein de détails et de diables

Exercice

Écrire un programme `argv.s`

- Qui affiche le nombre d'arguments
- Puis qui affiche chacun des arguments

Modes d'adressage

Modes d'adressages

On sait lire et écrire des données en RISC-V

Mais beaucoup de complexité

- Quoi utiliser pour quelle situation ?
- Que proposent les autres architectures ?

Mode d'adressage

- Paramètre des instructions machines
- Comment l'UCT détermine les opérandes des instructions ?
  - RISC (*Reduced Instruction Set Computer*): RISC-V, ARM
    $\rightarrow$ Peu de mode d'adressages
  - CISC (*Complex Instruction Set Computer*): x86
    $\rightarrow$ Nombreux mode d'adressages

Adressage immédiat (ou littéral)

On donne l'opérande

- L'opérande est codée dans l'instruction
  La valeur de l'opérande est immédiatement disponible
- Pas d'accès mémoire supplémentaire
  Donc pas forcément considéré comme un mode d'adressage

Exemple RISC-V

- Toutes les instructions immédiates
- `addi s0, s0, 42`

Adressage immédiat

Utilisation

- Constantes (littérale ou via un symbole)

Contraintes

- Le nombre de bits de l'opérande peut être limité
  Par exemple 12 bits seulement en RISC-V (instruction de type I)
- Ou la taille des instruction peut être grande
  Exemple x86_64:
  `movabs rax,0x123456789abcdef0` fait 10 octets
  `48 b8 f0 de bc 9a 78 56 34 12`

Adressage à registre

Un registre contient l'opérande

- L'opérande est dans un registre
- Le numéro du registre est codé dans l'instruction
- Pas d'accès mémoire supplémentaire
  Donc pas forcément considéré comme un mode d'adressage

Exemple RISC-V

- `add a0, a1, a2`

Utilisation

- La grande majorité des instructions CISC
- La très grande majorité des instructions RISC

Adressage direct (ou absolu)

On donne l'adresse exacte de l'opérande en mémoire

- L'adresse absolue est codée dans l'instruction
- Plusieurs octets peuvent être lus ou modifiés
- cible = mem[adresse]

Exemple RISC-V

N'existe pas (en soit), mais pseudos instructions équivalentes

- `lw a0, label`
- `lb a1, 100`
- `call label`

Adressage direct

Utilisations

- Variables globales
- Appel de routines de bibliothèques

Contraintes

- Comme pour l'adressage immédiat
  - Beaucoup d'octets nécessaires pour coder une adresse
  - Et/ou limité à des adresses basses
- Plus complexe lors de l'édition de liens
  - Surtout si plusieurs instructions (pseudoinstructions)
  - Exemple: l'édition de liens en RISC-V
  - Une autre fois...

Adressage indirect à registre

Un registre contient l'adresse

- L'adresse absolue est stockée dans un registre
- Le numéro du registre est codé dans l'instruction

Exemples RISC-V

N'existe pas en RISC-V mais pseudos instructions équivalentes

- `lw a0, (a1)`
- `jr a0`

Adressage indirect à registre

Utilisation

- Déréférencements de pointeurs
- Branchements indirects (et retours de routines)

Avantages

- Efficace et compact en RISC-V
- Généraliste

Adressage relatif

PC + décalage

- L'adresse effective est calculée à l'exécution
- La cible est en mémoire avant ou après le compteur ordinal
- cible = mem[pc+décalage]

Exemples RISC-V

Seulement pour les branchements

- `beq a1, a1, label`
- `jal ra, label`

Pseudoinstruction pour les données

- `lw a1, label`

Adressage relatif

Utilisation

- Branchements (conditionnels et inconditionnels)
- Appels de routines locales
- Accès aux donnée PIC (pseudoinstruction en RISC-V)

Avantages

- Économe en nombre de bits
- Compatible avec PIC (*position-independent code*)

Adressage basé (ou base + décalage)

Registre + décalage

- L'adresse effective est calculée à l'exécution
- Le numéro d'un registre de **base** est codé dans l'instruction
- Un **décalage** est codé dans l'instruction
- cible = mem[base+décalage]

Exemples RISC-V

- `lw a1, 8(a0)`
- `jalr ra, a0, 8`

Adressage basé

Utilisation

- Accès à la pile
- Position statique dans un tableau
- Accès à un élément d'une structure (plus tard)

Avantage

- Efficace et compact en RISC-V
- Fait une addition implicite

Pseudoinstructions RISC-V

- Une pseudoinstruction présente (souvent) un adressage simple
- Correspond à une (ou des) instruction avec un adressage riche

Exercices

- Que font les instructions suivantes?
- Lesquelles sont des pseudoinstructions?
- À quelles vraies instructions elles correspondent?
  - `add a0, a1, a2`
  - `addi a0, a1, 8`
  - `li a0, 8`
  - `lw a0, 8`
  - `lw a0, (a1)`
  - `lw a0, 8(a1)`
  - `jr a0`
  - `lw a0, label`
  - `la a0, label`
  - `call label`

Solutions

- `add a0, a1, a2` → vraie instruction
- `addi a0, a1, 8` → vraie instruction
- `li a0, 8` → `addi a0, x0, 8`
- `lw a0, 8` → `lw a0, 8(x0)`
- `lw a0, (a1)` → `lw a0, 0(a1)`
- `lw a0, 8(a1)` → vraie instruction
- `jr a0` → `jalr x0, a0, 0`

Les suivantes nécessitent les valeur `H` et `L` en fonction de `label` et de l'adresse de l'instruction

- `lw a0, label` → `auipc a0, H; lw a0, L(a0)`
- `la a0, label` → `auipc a0, H; lw a0, L(a0)`
- `call label` → `auipc t1, H; jalr ra, t1, L`

Autres adressages

Base + index

- Deux registres (**base** et **index**) dans l'instruction
- cible = mem[base+index]
- Utilisation: index dans un tableau

Base + index + décalage

- Deux registres (**base** et **index**) dans l'instruction
- Un **décalage** (*offset*) dans l'instruction
- cible = mem[base+index+décalage]
- Utilisation: index dans un tableau, lui-même dans une structure ou dans la pile

Autres adressages

Base + index + décalage + facteur

- Deux registres (**base** et **index**) dans l'instruction
- Un **décalage** (*offset*) dans l'instruction
- Un **facteur** d'échelle (*scale*) dans l'instruction (seulement 1, 2, 4, 8 par exemple)
- cible = mem[base + (index × facteur) + décalage]
- Exemple en `x86_64`:

```
48 63 44 b7 08
movsxd rax,DWORD PTR [rdi+rsi*4+0x8] # syntaxe Intel
movslq 0x8(%rdi,%rsi,4),%rax # syntaxe AT&T
```

Adressage sur la pile

Certaines architectures offrent des instructions dédiés

- `push` et `pop` en x86 par exemple

Conclusion

Résumé

- Pas de nouvelles instructions ou directives cette semaine
- Mais on a vu les registres (ABI) `ra` (`x1`) et `sp` (`x2`)
- Utilisation avancée de la mémoire:
  tableaux, pointeurs, pile, arguments des programmes
- Bonus: nos premières routines (mais on y reviendra)

La prochaine fois

Instructions

- Plus d'instructions RISC-V
- Codage des instructions
- Révisions avant l'examen intra