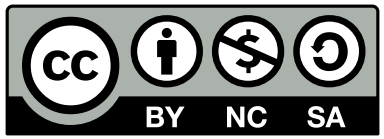
1000 Routines

programmes, bibliothèques, récursivité, pointeurs de routine, paramètres et variable locales dans la pile

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur
v233



Rappels

- Routines: `call` (`jal`) et `ret` (`jalr`)
- Pile (*stack*): segment mémoire + registre `sp`
- ABI (*Application Binary Interface*): `a0` à `a7` arguments
- Prologues et épilogue:`s0` à `s11` (et `ra`) sauvegardés par l'appelé
- Bibliothèques de routines
- Symboles et étiquettes

Plan

Routines

Composition d'une routine

Interface publique : permet l'imputabilité

- Signature: nombre et type des paramètres
  - Souvent explicite dans langage de haut niveau
- Documentation: comportement et règles d'utilisation
- Contrats: assertions formelles et semi-formelles
- API (*Application Programming Interface*) / ABI (*Application Binary Interface*)

Corps

- Du code (possiblement privé)
- Peut faire appel à d'autres routines
- Peut faire appel à lui même (récursivité)
- Peut changer tant que l'interface est respectée

Passage d'information

Invoquant et invoqué

- Peuvent communiquer de l'information

Moyens

- Paramètres
- Valeur de retour
- Modification de l'état global (mal)
  - Variables globales
  - Effets de bord (*side effects*)

Attention

- Documenter toute modification de l'état global
- Une modification non documentée est un bug

Pile obligatoire?

Pas d'appel interne

- Pas besoin de sauver `ra`
- Ne pas utiliser `s0` à `s11` $\rightarrow$ Pas besoin de les sauvegarder
- **Inconvénient**: ajout de `call` pour déboguer

Appel interne existant

- Sauvegarde nécessaire de `ra`
- Utiliser `s0` à `s11` qui ne seront pas perdu
  $\rightarrow$ Sauvegarde nécessaire de ces registres

Ne pas respecter l'ABI

- L'ABI est **convention** entre composantes logicielles
  - *Application Binary Interface*
  - Niveau langage machine
  - Bibliothèques, système d'exploitation, etc.
- On peut **ne pas** la respecter si on sait ce que l'on fait
  - **Fortement** déconseillé dans le cadre du cours

Exercice (rappel)

Écrire un programme `tab.s`

- Routine `readtab` qui lit un tableau d'entiers 32 bits
  - `a0` adresse du tableau
  - `a1` nombres d'éléments du tableau
  - Contrainte: utiliser `readInt` de `libs.s`
- Routine `sumtab` qui somme un tableau d'entiers 32 bits
  - Mêmes arguments
  - Retour: `a0` la somme des éléments du tableau
- Un programme principal qui lit un tableau et affiche la somme
  - Le tableau est alloué globalement dans la section `data`

Récursivité

Récursivité

- Voir Récursivité

Récursivité

- Voir Récursivité

Une routine s'appelle elle-même

- Tout fonctionne tout seul (c'est magique)
- Bien faire les **prologues** et **épilogues**
- Ne pas oublier la **condition d'arrêt**
  - Pour ne pas partir récursivité **infinie**
  - Pour ne pas **déborder** la pile

Source: SMBC (2019)

Exercice

Écrire un programmer `fib_rec.s`

- Version récursive (et inefficace!) de la suite de Fibonacci.
- Pseudocode:

```
int fib(int n) {
    if (n<=1) {
        return n;
    } else {
        return fib(n-1)+fib(n-2);
    }
}
```

Débordement de pile

Débordement de pile (*stack overflow*)

- Erreur classique de récursivité
- Exercice : comment faire déborder la pile ?

Débordement dans l'autre sens

- C'est idiot mais ça arrive
- *stack underflow*

Effets en pratique

Ça dépend des environnements

- De la mémoire inaccessible sera sans doute atteinte
- Erreur à l'exécution (*segmentation fault* par exemple)

Appel terminal (*tail call*)

Optimisation

- Un **appel** de routine (sauvegarde l'adresse de retour)
- qui précède un **retour**
- peut se simplifier en un **branchement** (sans sauvegarde)

```
jal foo     # jal ra, foo, 0
ret         # jr ra   / jalr x0, ra
# devient
j foo       # jal x0, foo, 0
```

- Les compilateurs optimisants le font pour vous
- Pseudoinstruction `tail`: branchement distant
- **Récursion terminale** même chose mais dans une routine récursive

Récursion terminale (*tail récursion*)

Appel terminal d'une fonction récursive

- On sait exactement ce qu'on appelle
- On peut réutiliser prologue et épilogues
  - Économie de temps CPU
  - Économie d'espace de pile
- Ça revient à transformer un appel récursif en une boucle

Inconvénients

- Utilisation limitée avec certains cas d'ABI
- Débogage plus complexe:
  les routines intermédiaires ont disparues de la pile

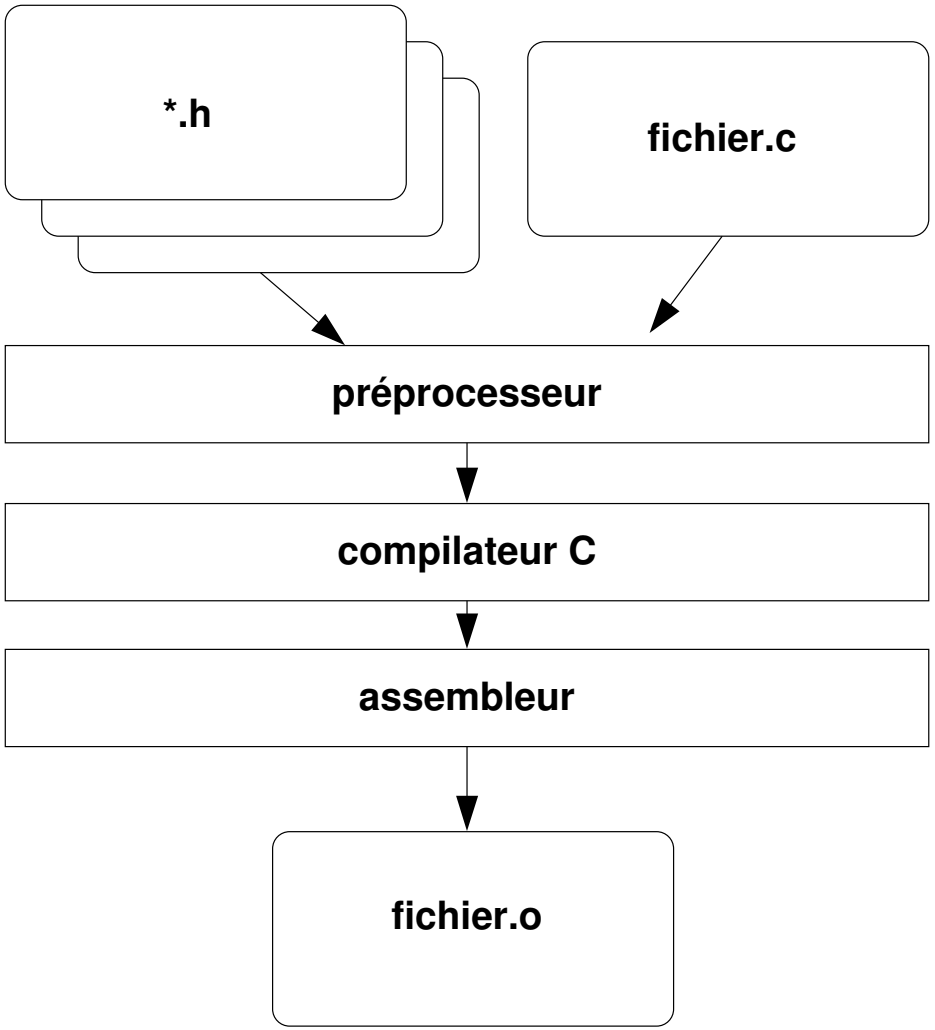
Programmes et bibliothèques

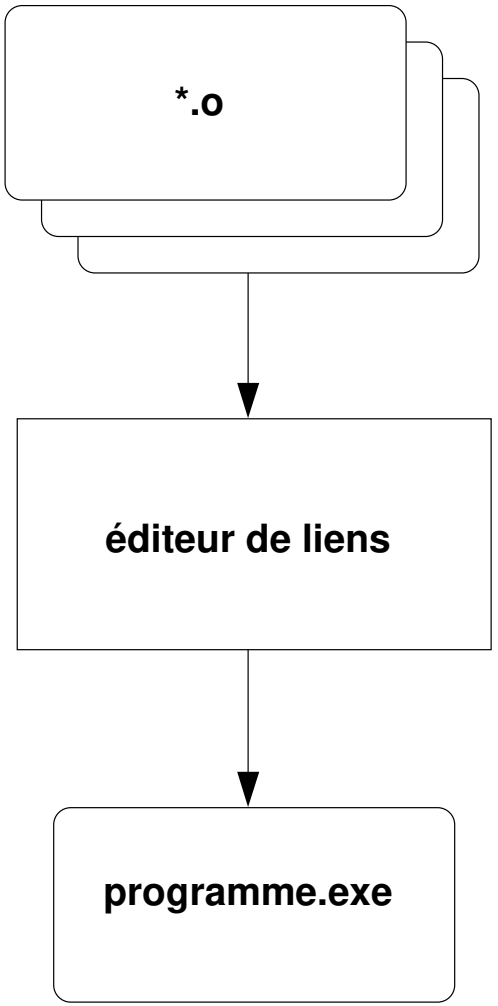
Utilisation de bibliothèques (rappel)

- Un programme principal
- Des bibliothèques fournissant des routines
- Combiner le tout pour avoir un programme exécutable

```
java -jar rars.java program.s lib1.s lib2.s
gcc program.s lib1.s lib2.s -static -nostdlib -o program
```

Schéma de compilation classique en C (rappel)





Compilation séparée

Assembler indépendamment chaque programme

- `gcc -c` (compile) et assemble, produit un fichier **objet** (`.o`)
- En fait `gcc` invoque simplement `as` (utiliser `-v` pour voir)

```
gcc -c program.s
gcc -c lib1.s
gcc -c lib2.s
```

Compilation séparée

Combiner les `.o` pour faire un exécutable

- On parle d'**édition de liens**
- `gcc` sans option comme `-c` produit un exécutable
  - Appelé `a.out` pour des raisons historiques
  - $\rightarrow$ Utilisez `-o` pour renseigner un meilleur nom
- En fait `gcc` invoque indirectement `ls`
  - `-v` montre `collect2` un utilitaire interne qui appelle `ld`
  - Beaucoup d'option et configuration complexe

```
gcc program.o lib1.o lib2.o -static -nostdlib -o program
```

- `-static` désactive l'utilisation de bibliothèques dynamiques
- `-nostdlib` désactive l'utilisation des bibliothèques par défaut

Export de symboles

Directive `.global`

- Exporte un symbole défini dans le fichier assembleur
- Pour qu'il soit utilisable (public)
- Outil `nm` (*names*) pour lister les symboles

Importation

- Pas besoin de déclarer les importations en `GNU as` et `RARS`
- Un symbole utilisé non défini dans le fichier est considéré externe

Point d'entrée

- Où le programme commence ?
- Cela dépend de l'environnent et des outils

RARS

- L'adresse du segment `text` du premier fichier (ou du fichier actif)
- Optionnellement au symbole global `main` (voir *Settings*)

GNU

- Par défaut (sans `-nostdlib`): au symbole global `main`
- Avec `-nostdlib`: au symbole global `_start`
  - Ou à l'adresse du segment `text` du premier fichier

libs.s

Exercice: lire et comprendre `libs.s`

- Pas de code magique
- Vous auriez-pu la développer vous-même :)

Bibliothèques binaires

- Prête à l'emploi et empaquetés

Bibliothèques statiques

- Intégrés lors de l'éditions de liens dans l'exécutable final
- Fichiers `.a` sous Linux: `find /lib/ -name '*.a'`

Bibliothèques dynamiques

- Non intégrée dans l'exécutable final
- Chargés durant l'exécution du programme
  - Lors du démarrage du programme
  - Ou paresseusement lors de son exécution
  - Ou programmativement (plugins par exemple)
- Fichiers `.so` sous Linux: `find /lib/ -name '*.so'`
- Fichiers `.dll` sous Windows

Bibliothèques standard GNU

`libc`, `crt0` et autre

- Contient les routines standard du langage C
  - Voir INF3135 pour le langage C
- Et les routines de démarrage d'un exécutable en C
  - Fournit le point d'entrée `_start`
  - `_start` appelle (indirectement) `main`
  - `main` est implémenté par le programme principal
- Initialise et configure de nombreuses affaires

Adresses des étiquettes

```
la s0, etiquette
sw s1, etiquette
```

Question: comment déterminer l'adresse de l'étiquette

Cas simple

- Dans un simulateur simple ou un système embarqués simple
- L'assembleur peut déterminer les valeurs des étiquettes

Cas réel

- Dans la vrai vie, la position finale des étiquettes n'est pas connu à l'assemblage, mais
  - à l'édition de liens
  - au chargement du programme
  - au chargement dynamique de bibliothèques pendant l'exécution du programme
- Le code machine généré à l'assemblage doit alors être prévu pour

Édition de liens et chargement dynamique

Assemblage

- On génère du code machine
- Les symboles inconnus sont laissés non-résolus
- Mais on note le symbole non résolu-quelque part

Relocation

Quand on connait les vraies valeurs des symboles:

- On « *bricole* » le code machine déjà généré
- On insère les bonne valeurs directement dans le binaire
- ... c'est compliqué

Conventions

Les règles et techniques de **relocations**

- Ne sont pas déterminées par l'**architecture**
- Mais par les conventions des outils et environnements (*ABI*)
- RISC-V ABIs Specification voir section 8.4. Relocations

Code indépendant de la position (PIC)

- Utilisé par les bibliothèques dynamiques
- Et certains exécutables (*position-independent executable*, PIE)

PC-relative

- Ne pas utiliser d'adresses absolues (qui sont inconnue)
- Mais utiliser des adresses relatives à l'instruction courante (`pc`)
- Exemple: `auipc` en RISC-V

Global Offset Tables (GOTs)

- Table à un position connue (*pc-relative*)
- Qui contient les vraies adresses des symboles
- *load* et *store* coutent un *load* de plus
- Le chargeur calcule et remplit la GOT

Pointeur de routine

Pointeur de routine (de fonction)

- On manipule l'**adresse** mémoire d'une routine
- On peut « *appeler* » cette adresse
- `jalr` en RISC-V

Attention: Respect de l'ABI

- La routine **réellement** appelée est inconnue
- Mais son ABI est **fixée d'avance**
  - On doit connaitre les attentes de la routine appelée
  - Il faut préparer les arguments et récupérer les valeurs de retour

Exercice: Écrire un programme `each.s`

- Qui implémente une routine d'itération `each`
  - `a0`: adresse d'un tableau
  - `a1`: nombre d'éléments du tableau
  - `a2`: taille d'un élément (en octets)
  - `a3`: adresse d'une routine d'action appelée à chaque élément
- ABI de la routine d'action appelée (`a3`)
  - `a0`: adresse de l'élément

```
.data
t:  .word 1, 8, 2, 10, 5, 5, -2, 2, 5, 4
.text
    la a0, t
    li a1, 10
    li a2, 4
    la a3, printWord
    jal each
```

Paramètres et variables locales

Utilisation de la pile

Idée : stocker plus que la sauvegarde de registres

- Stocker des variables locales
- Stocker des arguments si trop gros ou nombreux
- Stocker d'autres trucs...

Cadre de pile (*stack frame*)

- Le « morceau de pile » utilisé par une routine
- Ses sauvegardes de registres (dont `ra`)
- Ses arguments supplémentaires
- Ses variables locales supplémentaires
- La *pile* est une *pile* de cadres de routines **actives**

Variables locales dans la pile

- Si trop gros (plus grand qu'un registre)
- Ou si trop nombreux (plus que les registres)
- Ou si on a besoin d'une adresse
  (pour communiquer l'adresse de la donnée)

Variables locales vs. globale

- Ne **consomme** la mémoire que si la routine est **active**
  - Allocation systématique à l'entrée de la routine
  - Désallocation systématique à la sortie de la routine
- Compatible avec récursivité et concurrence
  - Chaque cadre de pile a ses propres variables locales
  - Plusieurs « *versions* » d'une variable locale coexistent

Accès aux variables locales dans la pile

Réserver/libérer la place

- `addi sp, sp, ...`
- Dans les prologues et épilogues
- Attention à l'**alignement** dans la pile: réserver plus si besoin

Adressage basé dans la pile

- `sp` + décalage
- Exemple: `sw a0, 12(sp)`
- Plus de lisibilité: Utiliser des `.eqv` pour nommer les décalages

```
.eqv toto, +12 # variable locale `toto`
# ...
sw a0, toto(sp)
```

Accès aux variables locales (`local.s`)

```
foo:
    # Variables locales
    .eqv toto, +8          # variable locale `toto`
    # Prologue
    addi sp, sp, -16       # sp, l1, alignement
    sd ra, 0(sp)
    # Corps (artificiellement simple)
    call readInt
    sw a0, toto(sp)        # affectation de toto
    # ...
    lw a0, toto(sp)        # lecture de toto
    call printInt
    # Épilogue
    ld ra, 0(sp)
    addi sp, sp, 16        # restauration de la pile
    ret
```

Exercice

Écrire un programme `incptr.s`

- Routine `incptr` qui incrémente le mot à l'adresse `a0`
- Utiliser `incptr` sur une variable locale

Exercice : écrire une routine `tabwork` qui

- Alloue un tableau de 4 entiers dans la pile (variable locale)
- L'initialise avec `readtab`
- Affiche sa somme grâce à `sumtab`
- Note: Étendez le programme `tab.s`

Paramètres et retours dans la pile

- Une affaire de **convention d'appel**; par **exemple**:

L'appelant

- Réserve de la pile additionnelle (`addi sp...`)
  - Pour paramètres et retours
- Initialise les paramètres dans la pile: adressage basé
- Appelle l'invoqué (`call`...)
- Lit les retours dans la pile: adressage basé
- Libère la pile additionnelle (`addi sp...`)

L'appelé

- Les paramètres sont déjà réservés (ne fait rien)
- Accède aux paramètres dans la pile: adressage basé
- Affecte les valeurs de retour: adressage basé

Paramètres et retours dans la pile (params.s)

```
    # appel de `s2 = foo(s1)`
    addi sp, sp, -16# un retour et un argument
    sd s1, 0(sp)     # sommet de pile pour l'argument
    jal foo          # appel
    ld s2, 8(sp)     # récupération du résultat
    addi sp, sp, 16 # libération de la pile
    #...

foo:
    ld t0, 0(sp)     # récupération de l'argument
    addi t0, t0, 1   # corps de la routine
    sw t0, 8(sp)     # sauvegarde du résulat
    ret
```

Convention d'appel

Nombreux détails à déterminer

- Où paramètres et retours sont rangés ?
- Dans quel ordre ?
- Qui nettoie la pile ?
- Qui sauvegarde quels registre ?
- Taille ? Alignement ? Boutisme ?
- À quoi correspondent les types (`int`, `long`, etc.) des langages de haut niveau (principalement C et C++)
- Ne rien laisser au hasard

Convention d'appel RISC-V

Les grandes lignes

- `sp` toujours aligné à 16 octets (128 bits)
- Arguments dans `a0` à `a7`, puis sur la pile
- Retours dans `a0` à `a1`

Le détail

- RISC-V ABIs Specification
  Voir *Chapter 2. Procedure Calling Convention*
- Plus compliqué s'il y a des flottants, des registres vectoriels, des données plus grosses
- Variations également possibles

Autre architectures

Pile x86

- Les instructions `call` et `ret` empile et dépile l'adresse de retour
  - Pas de registre `ra`
- Pas beaucoup de registres
  - Plus d'arguments et de variables locales dans la pile

Zoo des conventions d'appel x86

- Nombreuses conventions. fiche Wikipédia
- Dépendent des outils, systèmes et langages
- System V Application Binary Interface AMD64
  Pour Linux sur x86_64

Conclusion

Résumé

- Routines: récursivité, pointeur de fonction

L'ABI détermine les conventions pour

- Édition de liens, bibliothèques, résolution des symboles
- Pile: variables locales et paramètres

La prochaine fois

- Structures de données complexes
- Allocation dynamique sur le tas
- Liste chainée