ALMO

Convention d'appel des fonctions

plan

Convention d'appel des fonctions

La convention d'appel des fonctions est à la fin du document sur le langage d'assemblage :

https://www-soc.lip6.fr/trac/sesi-almo/chrome/site/docs/ALMO-mips32-archi-asm.pdf

Cette convention est définie dans le document qui définit le bon usage du MIPS (pages 3-46 à 3-49)

ftp://www.linux-mips.org/pub/linux/mips/doc/ABI/psABI_mips3.0.pdf

Le document contient pleins d'autres conventions,
seule la convention expliquée dans ce cours nous intéresse.

Programmation structurée

Un programme est structuré en fonctions

Une fonction est un morceau de programme

- qui reçoit des arguments
- qui rend un résultat
- qui peut accéder aux données globales du programme
- qui dispose d'un espace de travail propre.

Définition en C :

```
type_var_globale  var_globale;
type_retour nom_fonction (type_args args)
{
    type_var_locale  var_locale;
    instructions;
    return valeur_retour_fonction;
}
```

corps

bloc d'inst.

Usage :

l'appel d'une fonction est une instruction

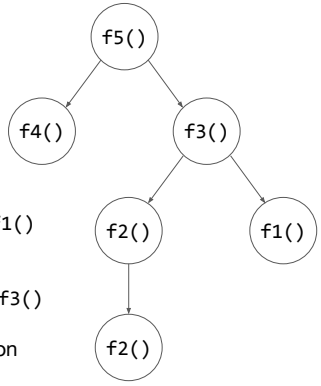
```
instruction_1;
instruction_2;
v = nom_fonction( arg);
instruction_4;
```


Une fonction peut appeler d'autres fonctions

Le comportement d'une fonction est défini par un bloc d'instructions qui peut contenir des appels d'autres fonctions.

```
f1() {
    …
}
f2() {
    f2();
}
f3() {
    f1();
    f2();
}
f4() {
    …
}
f5() {
    f4();
    f3();
}
```

Arbre d'appels



`f3()` est la fonction **appelante** de `f2()` et `f1()`

`f2()` et `f1()` sont les fonctions **appelées** de `f3()`

`f3()` est aussi la fonction **appelée** de `f5()`

Exécution d'un programme

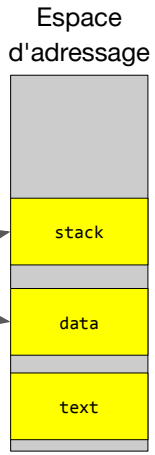
Un programme qui s'exécute est défini par

- un espace d'adressage avec les trois sections nécessaires à son exécution
  - text pour son code,
  - data pour ses données globales,
  - stack pour sa (ou ses) pile(s) d'exécution pour les données locales.
- Le programme a une fonction d'entrée (en C c'est `main()`).
  La valeur de retour de la fonction `main()` est la valeur de retour du programme, c'est un entier.
- La fonction `main()` va appeler, en général, d'autres fonctions.
- le *système d'exploitation* crée au moins un **fil d'exécution**, défini par :
  - une **pile d'exécution** des fonctions dans la section stack
  - **l'état de registres** du processeur dont :
    - le pointeur de programme (PC) désignant l'instruction en cours
    - le pointeur de pile (SP pour Stack Pointer)
    - les arguments de la fonction `main()`

Variables globales — Variables locales

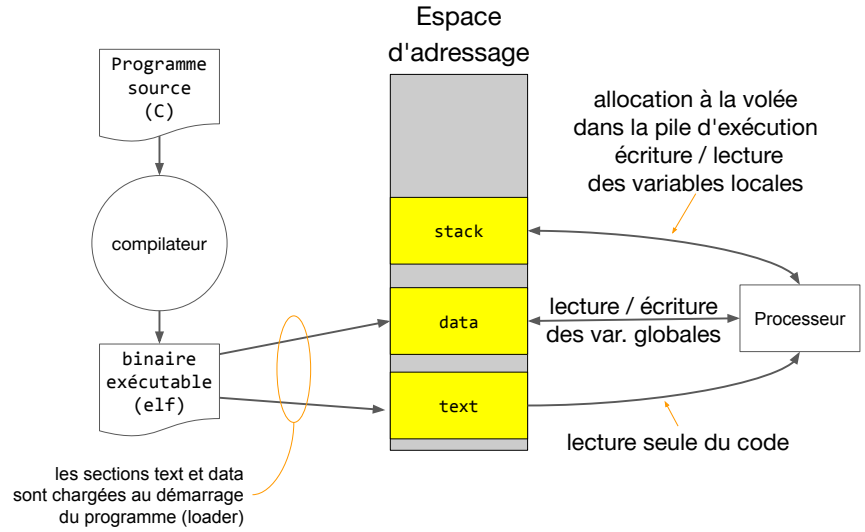
Une fonction peut utiliser :

- les variables globales du programme, lesquelles sont visibles (accessibles) de toutes les fonctions *
  - Les variables globales sont allouées à la compilation
  - Elles placées dans la section **`data`**
  - Leur valeur initiale est connue (0 par défaut)
  - Elles sont détruites à la terminaison du programme
- les variables locales, lesquelles sont visibles des instructions du bloc (et sous-blocs) où elles sont définies.
  - Les variables sont allouées à l'appel de la fonction
  - Elles placées dans le section **`stack`**
  - Leur valeur initiale est indéfinie par défaut
  - Elles sont détruites à la sortie de la fonction



* *en utilisant le mot **static** devant leur déclaration, leur visibilité est réduite au seul fichier où elles sont définies*

Allocation des variables



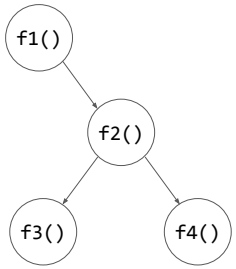
Services de la pile d'exécution

- La **pile d'exécution** offre de la place pour **3 services** :
  1. le passage des **arguments** des fonctions
  2. la sauvegarde du contenu des **registres** de travail du processeur
     En effet, les calculs sont faits dans les registres GPR (General Purpose Reg)
     En conséquence quand on appelle une fonction, les registres contiennent des valeurs appartenant à la fonction appelante.
     Or, la fonction appelée va aussi avoir besoin de registres pour ces calculs.
     Il faut donc sauver le contenu des registres quand on entre dans une fonction et il faut les restaurer en sortant, mais nous allons le voir, pas tous.
  3. l'allocation temporaire des **variables locales**
- Une fonction qui s'exécute, se réserve de la place dans la pile pour ces 3 services. Cette place est appelée le **contexte d'exécution** de la fonction.
- Le contexte d'exécution d'une fonction est donc constitué de 3 zones pour :
  1. la sauvegarde de certains registres de travail (qu'il faut restaurer en sortant)
  2. ses propres variables locales
  3. les arguments des fonctions **appelées** par la fonction courante.

Empilement des contextes 1/2

Soit le programme

```
f1( int a1 ) {
     int v1;
     f2( x2 );
     I1;
}
f2( int a2 ) {
     int v2;
     f3( x3 );
     f4( x4 );
     I2:
}
f3( int a3 ) {
     int v3;
     I3;
}
f4( int a4 ) {
     int v4;
     I4;
}
```



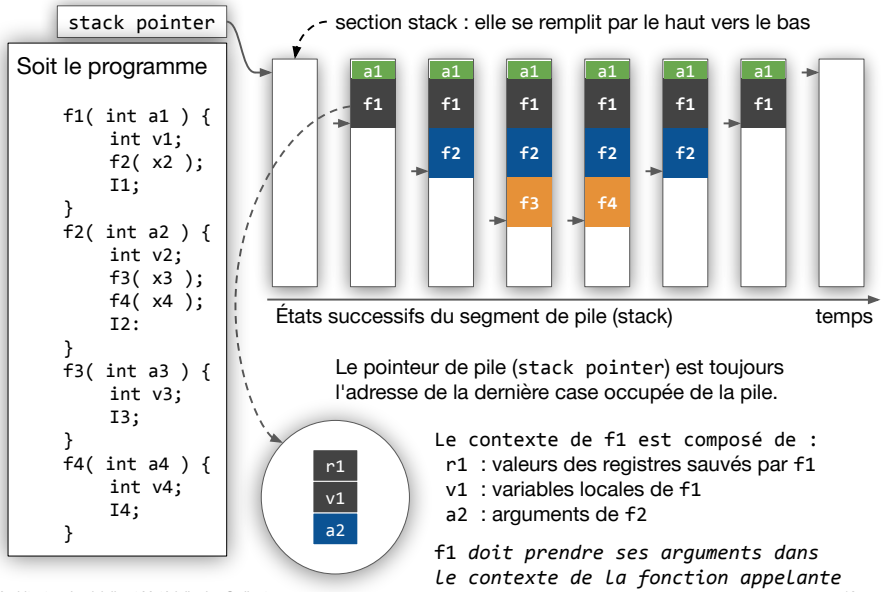
`Vocabulaire :`

`f2()` est la fonction :
- appelée de `f1()`
- appelante de `f3()` et `f4()`

`f3()` et `f4()`
- sont des fonctions **terminales**

L'exécution du programme consiste à
- entrer dans `f1()`
  - entrer dans `f2()`
    - entrer dans `f3()` sortir de `f3()`
    - entrer dans `f4()` sortir de `f4()`
  - sortir de `f2()`
- sortir de `f1()`

Empilement des contextes 2/2



Le pointeur de pile (`stack pointer`) est toujours l'adresse de la dernière case occupée de la pile.

```
Le contexte de f1 est composé de :
 r1 : valeurs des registres sauvés par f1
 v1 : variables locales de f1
 a2 : arguments de f2
```

`f1` doit prendre ses arguments dans le contexte de la fonction appelante

Allocation des contextes 1/2

Rappel : Le contexte d'exécution d'une fonction est constituée de 3 zones
1. `(r)` la sauvegarde de ces propres registres de travail
2. `(v)` ses propres variables locales
3. `(a)` les arguments des fonctions appelées

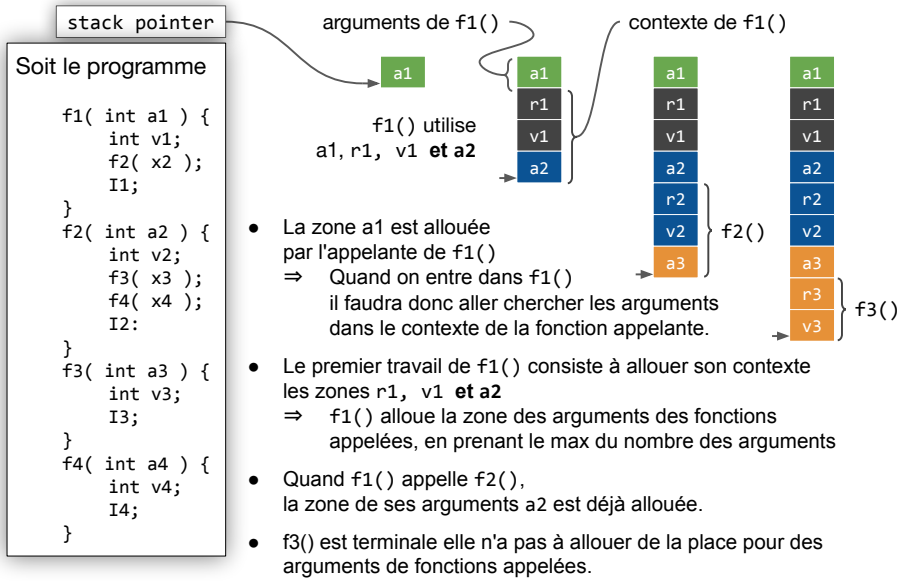
Soit
une fonction appelante `f1()` appelle une fonction appelée `f2()`
qui appelle une fonction `f3()`
`f2()` est donc l'appelée de `f1()` et l'appelante de `f3()`

Le contexte d'une fonction appelée `f2()` est alloué à l'entrée de `f2()` :

La fonction appelée `f2()` alloue de la place
- pour la sauvegarde de ses propres registres de travail
- pour ses propres variables locales
- **et aussi pour les arguments de la fonction** `f3()`

Attention : la zone des arguments de `f2()` est dans le contexte de `f1()`

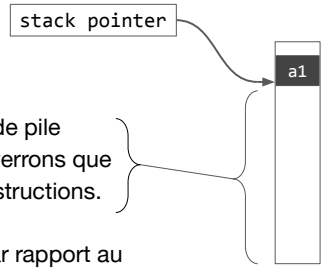
Allocation des contextes 2/2



- La zone a1 est allouée par l'appelante de `f1()`
  ⇒ Quand on entre dans `f1()` il faudra donc aller chercher les arguments dans le contexte de la fonction appelante.
- Le premier travail de `f1()` consiste à allouer son contexte les zones `r1, v1` **et a2**
  ⇒ `f1()` alloue la zone des arguments des fonctions appelées, en prenant le max du nombre des arguments
- Quand `f1()` appelle `f2()`, la zone de ses arguments `a2` est déjà allouée.
- f3() est terminale elle n'a pas à allouer de la place pour des arguments de fonctions appelées.

Pointeur de pile

Le pointeur de pile utilise le registre `$29`



A tout instant le pointeur de pile pointe sur la dernière case occupée dans la pile.

⇒ Les cases mémoires sous le pointeur de pile ne doivent jamais être utilisées, nous verrons que leur valeur peut changer entre deux instructions.

- L'accès aux données dans la pile se fait par rapport au pointeur de pile avec des déplacements positifs ou nuls

- Il n'y a **jamais de déplacement négatif** par rapport au pointeur de pile (d'accès sous le pointeur de pile).

```
lw  $x, imm ($29)  # imm >= 0
sw  $x, imm ($29)
```


Appel de fonction

L'appel d'une fonction se fait par les instructions :
`jal`, `jalr`, bgezal et bltzal.

`jal` signifie *jump and link*
jump parce qu'on saute à la première instruction de la fonction,
link parce qu'on doit se souvenir de l'adresse de l'instruction de retour
l'adresse de retour est stockée implicitement dans le registre `$31`

- `jal  label  : $31 ← PC+4  puis  PC ← label`
- `jalr  $r  : $31 ← PC+4  puis  PC ← $r`
- `bgezal $r, label  : si ($r>=0)  alors  $31 ← PC+4`
  `PC ← label`
  `sinon  PC ← PC+4`
- `bltzal $r, label  : si ($r<0)  alors  $31 ← PC+4`
  `PC ← label`
  `sinon  PC ← PC+4`

Retour de fonction

- Lorsqu'on entre dans une fonction, le registre $31 contient l'adresse de retour.
  Le retour dans la fonction appelante se fait par :

```
jr $31
```

- Lorsqu'on sort d'une fonction,

  $2 doit contenir la valeur de retour
  $3 également si c'est une valeur sur 64 bits

- Le registre $29 pointe au même endroit qu'à l'entrée.
  Si $29 a été modifié par la fonction, il devra être restauré.

Registres temporaires et persistants

L'ABI (Abstraction Binary Interface) du MIPS distingue deux types de registres :
1. Les registres persistants : $16 à $23 et $30
2. les registres temporaires : tous les autres

Un registre persistant n'est pas modifié par l'appel d'une fonction appelée.
Par exemple, si `f1()` place la valeur `42` dans le registre persistant `$18`, puis `f1()` appelle la fonction `f2()`, lorsque `f2()` retourne dans `f1()`, `$18` **contient toujours** `42`.

Un registre temporaire peut être modifié par l'appel d'une fonction appelée.
Par exemple, si `f1()` place la valeur `42` dans le registre temporaire `$10`, puis appelle la fonction `f2()`, lorsque `f2()` retourne dans `f1()`, `$10` **ne contient pas toujours** `42`.

Lorsqu'on entre dans une fonction `f()`, seuls les registres persistants qui vont être modifiés par l'exécution de la fonction `f()` doivent être sauvés dans le contexte de la fonction `f()` afin de les restaurer en sortant.
L'ordre des registres dans la pile est tel que le registre d'index le plus petit est à l'adresse la plus petite.
$29 et $31 ne sont pas des registres persistants mais ils sont également restaurés.

Spécialisation des registres

Le MIPS impose un usage pour ces registres dans le document MIPS ABI (Abstraction Binary Interface). C'est dans ce document qu'est défini la convention d'appel des fonctions que nous voyons ici en partie.

Le tableau ci-dessous résume les usages, nous allons le détailler plus loin, notez que chaque registre porte un nom symbolique en rapport avec son usage.

$0	Vaut 0 en lecture. Non modifié par une écriture
$1 (at)	Réservé à l'assembleur pour les macros.
$2, $3 (v0, V1)	Utilisés pour les calculs temporaires et la valeur de retour des fonctions,
$4 . $7 (a0 .. a3)	Utilisés pour le passage des arguments de fonctions, les valeurs ne sont pas préservées lors des appels de fonctions. Les autres arguments sont placés dans la pile.
$8..$15, $24, $25 (t0..t9)	Registres temporaires de travail, les valeurs ne sont pas préservées lors des appels de fonctions
$16 ,… $23, $30 (s0 … s8)	Registres persistants dont les valeurs sont préservées par les appels de fonctions
$26,$27 (K0, k1)	Réservés aux procédures noyau.
$28 (gp)	Pointeur sur la zone des variables globales (segment data)
$29 (sp)	Pointeur de pile
$31(ra)	Contient l'adresse de retour d'une fonction

Arguments d'une fonction

Une fonction appelante réserve une zone dans la pile pour TOUS les arguments des fonctions qu'elle appelle. Elle réserve donc une zone permettant de stocker tous les arguments de la fonction appelée en ayant le plus.

- Si la fonction appelante appelle une fonction à 1 argument entier et une fonction à 10 arguments, alors la fonction appelante réserve une zone pour les 10 arguments.
- L'ordre des arguments est imposé, le premier est à l'adresse la plus petite dans la zone des arguments.

L'ABI du MIPS impose une optimisation
Les 4 premiers arguments d'une fonction sont pas écrits dans la pile à la place qui leur a été allouée, ils sont placés dans les registres $4, $5, $6 et $7.

Donc, la fonction appelante a prévu la place dans la pile pour TOUS les arguments, même les 4 premiers, mais la fonction appelante ne place que les arguments au delà du 5ème dans la pile.

Forme générale d'une fonction

La fonction appelante a fait en sorte que :

- $29 pointe sur la case réservée au premier argument de la fonction appelée
- $4 à $7 contiennent la valeurs des 4 premiers arguments, les autres sont en pile

La fonction appelée est composé de trois parties et doit :

Prologue
1. — allouer de la place pour les registres persistants qu'elle utilise, ses variables locales et pour les arguments de ses propres fonctions appelées.
   - Soit **nr** le nombre de registres persistants (y compris $31)
   - Soit **nv** le nombre de variables locales
   - Soit **na** le nombre maximum d'arguments de ses propres fonctions appelées
     ATTENTION **a** n'est pas le nombre d'arguments de la fonction appelée
   - la place réservée est `(nr + nv + na) x 4` octets
— sauvegarder les registres persistants qu'elle utilise
— si besoin, écrire les arguments reçus dans les registres $4 à $7 dans la pile

Corps
2. — exécuter le corps de la fonction

Epilogue
3. — placer dans $2 la valeur de retour
— restaurer l'état des registres persistants
— restaurer le pointeur de pile
— revenir à la fonction appelante

un peu de code

```
fonction:   # ---------------------------- PROLOGUE
            addiu    $29,  $29, -X      # X = (nr + nv + na) x 4
            sw       $31, X-4 ($29)     # sauvegarde de l'adresse de retour
            sw       $RP, X-8 ($29)     # sauvegarde des registres persistants
            …
            sw       $4,  X ($29)       # sauvegardes des arguments (si besoin)
            …
                                  toujours positif !
            # ----------------------------- CORPS de fonction
            …

            # ---------------------------- EPILOGUE
            add      $2, $0, $R         # hyp: $R contient la valeur de retour
            lw       $31, X-4 ($29)     # restauration de l'adresse de retour
            lw       $RP, X-8 ($29)     # restauration des registres persistants
            addiu    $29, $29, X        # restauration du pointeur de pile
            jr       $31                # retour
```

Notez bien que lors de l'exécution d'une fonction

- le pointeur de pile n'est modifié que deux fois : 1 fois en entrant et 1 fois en sortant
- on ne restaure pas les registres $4, $5, etc, car ils sont temporaires

Appel d'une fonction

Dans le corps d'une fonction `f()`, quand on doit appeler une fonction `f()`,
il n'est jamais utile de déplacer le pointeur de pile pour les arguments de `g()`
puisque cela a déjà été fait dans le prologue de `f()`

- Si `g()` a moins de 4 arguments, on les écrit dans $4 à $7
  sinon les arguments à partir du 5ème sont placés dans la pile
- on exécute `jal g`
- au retour $2 contient la valeur de retour

Exemple : programmer la fonction `f()`

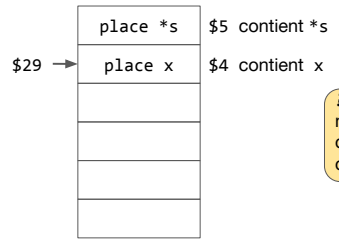
```
int g(int *t, int z);
void h(int *x);
int f(int x, int *s) {
    int v;
    v = g(s, 2);
    if ( v == x )
        return -1;
     v = h(s);
    return v;
}
```

1. Dessiner l'état de la pile telle qu'elle a été mise par la fonction appelante.
2. Ecrire le corps de la fonction
   Cette étape est faite avant le prologue parce qu'on ne connaît pas encore le nombre de registres persistants nécessaires.
   - Assignez des registres pour les variables en privilégiant les registres temporaires, sauf si c'est "moins efficace" que les persistants
   - Ecrire le code assembleur (1ère passe)
3. Écrire le prologue et l'épilogue
   - Déterminer **nr**, nv, na
   - Dessiner l'état de la pile après prologue
   - Ecrire le code assembleur du prologue et de l'épilogue
4. Une fois connu tous les registres, corriger le corps concernant les accès en pile


```
int g(int *t, int z);
void h(int *v);
int f(int x, int *s) {
    int v;
    v = g(s, 2);
    if ( v == x )
        return -1;
     v = h(s);
    return v;
}
```

fonction `f()`

1. Etat de la pile et argument à l'entrée f()



2. Ecrire le corps de la fonction
   - Assignez des registres
     `$8 : v`
     `$4 : x`
     `$5 : s`
   - assembleur du corps de `f()`

La place des arguments de g() est déjà réservée

?1 et *?2* ne seront connus qu'après l'écriture du prologue

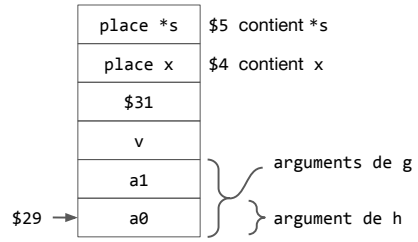
```
# v = g(s, 2);
    add  $4, $0, $5    # s
    li   $5,2          # 2
    jal  g             # appel
    add  $8, $0, $2    # retour

# if ( v == x ) return -1;
    lw   $4, ?1 ($29)  # x
    bne  $8, $4, suite
    li   $2, -1
    j    fin
suite:
    lw   $4, ?2 ($29)  # s
    jal  h             # appel
fin:
```



```
int g(int *t, int z);
void h(int *v);
int f(int x, int *s) {
    int v;
    v = g(s, 2);
    if ( v == x )
        return -1;
     v = h(s);
    return v;
}
```

fonction `f()`

Etat de la pile et argument après le prologue de f()



3. Ecrire le prologue et l'épilogue
   - déterminer **nr**, **nv**, et **na**
     - **nr** = 1 (seulement $31)
     - **nv** = 1 (seulement v)
     - **na** = 2 (= `MAX(2,1)`)
   - prologue

```
add  $29, $29, -16
sw   $31, 12 ($29)
sw   $4,  16 ($29)
sw   $5,  20 ($29)
```

   - épilogue

```
# la valeur de retour est déja
# dans $2 à la fin du corps
lw   $31, 12 ($29)
add  $29, $29, 16
jr   $31
```

4. Correction du code du corps
   - `?1 = 16`
   - `?2 = 20`

En résumé, nous avons vu :

- Un rappel de la programmation structurée
- Les trois sections nécessaires à l'exécution d'un programme
- La différence entre les variables globales et locales
- Les trois services offert par une pile d'exécution
- Le concept de contexte d'exécution et leur empilement
- L'usage du pointeur de pile
- Les instructions d'appel des fonctions et le retour des fonctions
- La différence entre les registres temporaires et persistants
- La spécialisation des registres
- La convention de passage des arguments avec optimisation
- La forme générale d'une fonction
- Un exemple de fonction en assembleur

Prochaine séance

Qu'est-ce qu'une chaîne de compilation ?

- Etapes
- Outils
- Fichiers