Les appels de fonctions dans MIPS

Fonction MIPS

Une fonction est un concept algorithmique utilisé pour structurer les programmes afin de les rendre plus faciles à comprendre et à manipuler. En fait, la fonction

- est un bloc d'instructions qui peut être appelé en cas de besoin à différents moments du programme.
- peut recevoir des paramètres et retourner des résultats. Les paramètres et les résultats agissent comme une interface entre la fonction appelée et le reste du programme.

Pour exécuter une fonction, le programme doit effectuer les étapes suivantes :

1. Le programme doit placer les paramètres à un endroit où la fonction appelée peut y accéder.
2. Transfère le contrôle à la fonction appelée.
3. Exécute les instructions de la fonction appelée.
4. La fonction appelée doit placer les résultats dans un endroit où l'appelant peut y accéder.
5. Renvoie le contrôle à l'appelant (à l'instruction suivant l'appel qui a été passé).

Les registres sont le lieu le plus rapide pour transmettre des paramètres et retourner des résultats. L'architecture MIPS respecte les conventions logicielles suivantes pour la transmission des paramètres et le renvoi des résultats :

- **`$a0-$a3`** : Quatre registres pour passer les arguments pour la fonction appelée.
- **`$v0-$v1`** : Deux registres pour renvoyer les résultats de la fonction appelée.
- **`$ra`** : Un registre contenant l'adresse de retour vers la fonction appelante.

Dans MIPS, l'instruction **`jal`** (jump-and-link) est utilisée pour lancer une fonction ; et l'instruction **`jr`** (jump register) permet de renvoyer le contrôle à la fonction appelante. Ainsi, pour appeler une fonction, nous utilisons l'instruction **`jal`** comme suit :

`jal label` où **`label`** est une étiquette qui marque le début de la fonction appelée.

L'instruction ci-dessus enregistre l'*adresse de retour* dans le registre **`$ra`** et saute à la première instruction dans la fonction appelée (c.-à-d. après l'étiquette **`label`**). L'*adresse de retour* est l'adresse de l'instruction qui apparaît après l'instruction **`jal`** dans la fonction appelante.

Pour retourner depuis une fonction, on utilise l'instruction :

```
jr $ra
```

Cette instruction affecte au registre **$pc** (compteur ordinal) la valeur stockée dans le registre **$ra**. C.-à-d. elle permet de sauter à l'adresse contenue dans le registre **$ra**.

La figure 1 ci-dessous montre un exemple de fonction en langage C (à gauche) qui vérifie si un caractère **ch** est une lettre minuscule ou non. La version équivalente dans le langage assembleur MIPS est présentée à droite. La fonction MIPS **islower** suppose que le paramètre **ch** est passé dans le registre **$a0**. Le résultat de la fonction est renvoyé dans le registre **$v0**.

```
int islower(char ch)
{
  if (ch>='a' && ch<='z')
    return 1;
  else
    return 0;
}
```

```
islower:
  blt $a0, 'a', else  # branche si $a0 < 'a'
  bgt $a0, 'z', else  # branche si $a0 > 'z'
  li  $v0, 1          # $v0 = 1
  jr  $ra             # retour à l'appelant
else:
  li  $v0, 0          # $v0 = 0
  jr  $ra             # retour à l'appelant
```

Figure 1. Exemple de fonction C et sa traduction en assembleur MIPS

Pour appeler la fonction **islower** dans MIPS, la fonction appelante doit d'abord copier le caractère **ch** dans le registre **$a0**, puis lancer l'appel de la fonction avec l'instruction **jal**.

```
move $a0, ...    # copie du caractère ch dans le registre $a0
jal   islower    # appel de la fonction islower
. . .            # returne ici après l'exécution de islower
```

En fait, l'architecture MIPS fournit une deuxième instruction pour l'appel de fonctions : l'instruction **jalr** (jump-and-link-register). En effet, à l'opposé de l'instruction **jal** où l'adresse de saut est codée dans l'instruction même via une étiquette, l'instruction **jalr** permet d'appeler des fonctions dont les adresses sont stockées dans des registres.

Le segment de pile et le registre « pointeur de pile »

Chaque programme possède trois segments lorsqu'il est chargé en mémoire par le système d'exploitation. Il y a le segment de code qui contient les instructions du programme en langage machine, le segment de données qui comprend les données et constantes du programme, et le segment de pile qui fournit une zone qui peut être allouée et libérée par des fonctions. Le programmeur n'a aucun contrôle sur l'emplacement de ces segments en mémoire.

Le segment de pile peut être utilisé par des fonctions

- pour transmettre de nombreux paramètres,
- pour allouer des variables locales, et
- pour enregistrer et conserver des registres entre les appels.

Sans ce segment, il serait impossible d'écrire des fonctions imbriquée et/ou récursives.

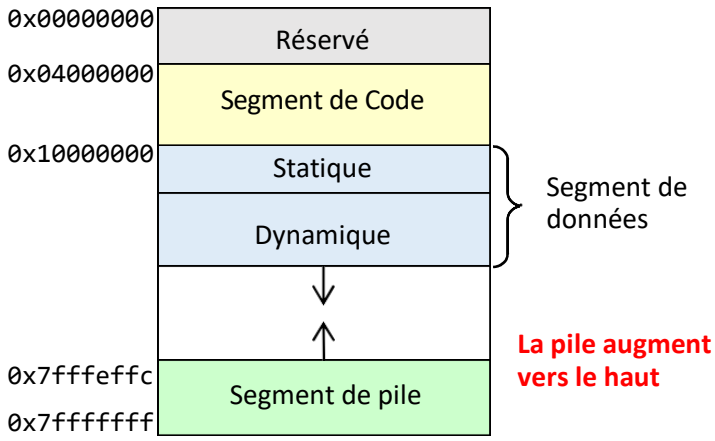

Figure 2. Les segments de Code, de Données et de Pile d'un programme

Lorsqu'un programme est chargé en mémoire, le système d'exploitation initialise le registre pointeur-de-pile **`$sp`** (registre **`$29`**) avec une adresse valide pour pointer vers le **haut de la pile**. Par exemple, lors de l'exécution d'un programme MIPS sous l'outil MARS, la valeur initiale du registre **`$sp`** est **`0x7fffeffc`**. Le segment de pile augmente vers des adresses mémoire plus faibles (*voir* Figure 3).

Une fonction peut réserver de l'espace sur la pile pour sauvegarder des registres et/ou pour allouer des variables locales. Cet espace est appelé une **trame de pile** (*anglais.* stack frame)**.** Pour allouer une trame de pile de **`n`** octets, nous **décrémentons** le pointeur de pile de **`n`** au début d'une fonction :

```
addiu $sp, $sp, -n        # n est un nombre (c.-à.-d une constante) d'octets
```

Pour libérer une trame de pile de **`n`** octets, nous **incrémentons** le pointeur de pile de **n** avant le retour d'une fonction :

```
addiu $sp, $sp,  n        # n est un nombre (c.-à.-d une constante) d'octets
```

La figure 4 ci-dessous illustre l'état du segment de pile avant l'appel, pendant l'exécution et après le retour d'un appel de fonction. Initialement, le registre pointeur-de-pile **`$sp`** pointe vers le haut de la *trame de pile* de la fonction appelante. Une fois l'appel est effectué, le registre **`$sp`** est décrémenté pour faire de l'espace sur le haut du segment de pile afin de stocker la *trame de pile* de la fonction appelée. Enfin, au retour de la fonction appelée, le registre **`$sp`** est incrémenté de manière qu'il pointe à nouveau vers la *trame de pile* de la fonction appelante libérant ainsi la mémoire allouée sur la pile.

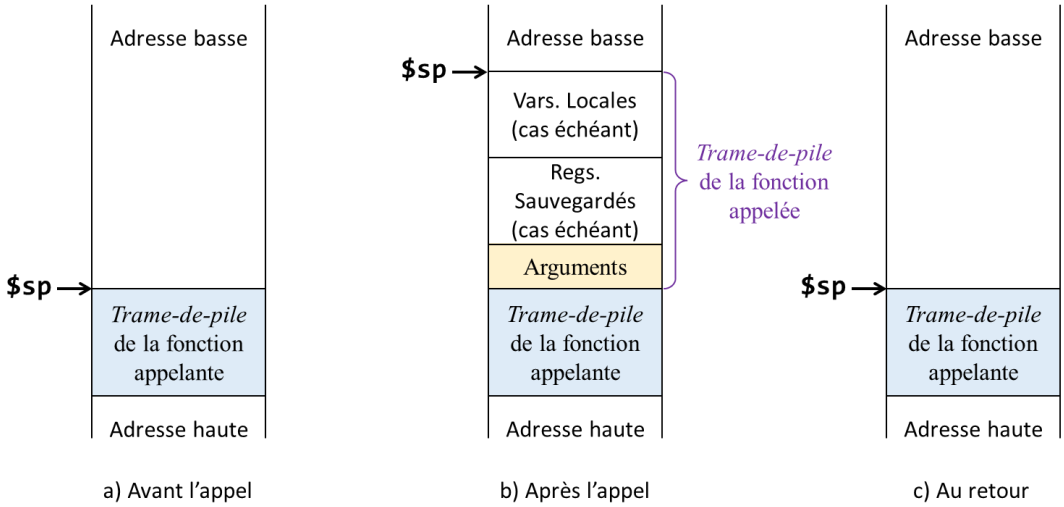

Figure 3. Allocation de pile (a) avant (b) pendant l'exécution, et (c) après le retour de la fonction appelée

Une *trame de pile* peut être utilisée pour passer des arguments à une fonction, enregistrer des registres entre les appels de fonction et/ou pour allouer de l'espace pour des variables locales déclarées dans la fonction. En particulier, le registre **$ra** doit être enregistré avant qu'une fonction puisse appeler une autre fonction parce que les instructions **jal** et **jalr** modifient le registre **$ra**. Les arguments d'une fonction sont généralement transmis dans les registres **$a0** à **$a3**. Toutefois, si une fonction a plus de quatre arguments, les arguments supplémentaires doivent être transmis sur la pile.

Un exemple de fonction **f()** qui alloue une *trame-de-pile* est illustré à la figure 5. La fonction n'est pas finale car elle appelle les fonctions **read()**, **reverse()** et **print()**. Par conséquent, l'adresse de retour (contenue dans le registre **$ra**) de la fonction **f()** doit être sauvegardée sur la pile. En outre, la *trame-de-pile* de la fonction **f()** doit réserver de l'espace pour le tableau local **array** (10 éléments entiers = 40 octets) comme illustré ci-dessous.

```
void f()
{
  int array[10];
  read(array, 10);
  reverse(array, 10);
  print(array, 10);
}
```

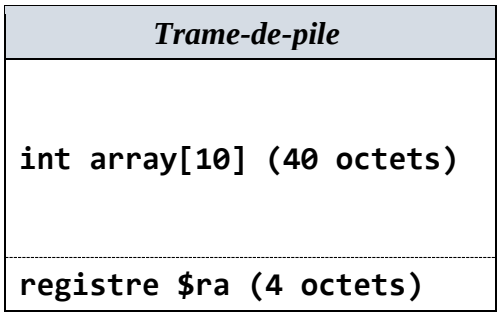

Figure 4. Exemple d'une fonction et de la *trame-de-pile* correspondante

La traduction de la fonction **f()** en langage assembleur MIPS est illustré à la figure 6. La fonction alloue une trame-de-pile de 44 octets. Le segment de pile est accessible à l'aide des mêmes instructions de lecture et d'écriture utilisées pour accéder au segment de données. Le registre pointeur-de-pile **$sp** contient l'adresse du haut de la pile. Différentes valeurs de déplacement sont utilisées pour accéder à différents éléments de la pile.

```
f:   addiu $sp, $sp, -44     # réserve une trame-de-pile de 44 octets
     sw    $ra, 40($sp)      # sauvegarde de $ra sur la pile
     add   $a0, $zero, $sp   # $a0 = pointeur de pile
     addi  $a1, $zero, 10    # $a1 = 10
     jal   read              # appel de la fonction 'read'
     add   $a0, $zero, $sp   # $a0 = pointeur de pile
     addi  $a1, $zero, 10    # $a1 = 10
     jal   reverse           # appel de la fonction 'reverse'
     add   $a0, $zero, $sp   # $a0 = pointeur de pile
     addi  $a1, $zero, 10    # $a1 = 10
     jal   print             # appel de la fonction 'print'
     lw    $ra, 40($sp)      # restitution de $ra depuis la pile
     addiu $sp, $sp, 44      # libérer la trame-de-pile de 44 octets
     jr    $ra               # retour à la fonction appelante
```

Figure 5. Traduction de la fonction **f()** en langage assembleur MIPS

Certains programmes MIPS utilisent le registre de pointeur de trame **`$fp`** (registre **`$30`**) pour pointer vers l'adresse de base d'une *trame-de-pile*. Cela peut être nécessaire si le pointeur de pile **`$sp`** change pendant l'exécution d'une fonction, ou si des tableaux et des objets sont alloués dynamiquement sur la pile.

Convention d'usage des registres dans MIPS

Une convention concernant l'usage des registres est nécessaire car les logiciels sont écrits par de nombreux programmeurs. En effet, chaque programmeur doit savoir comment les registres sont censés être utilisés, de telle sorte que son code n'entre pas en conflit avec des morceaux de code écrits par d'autres programmeurs.

Vous vous demandez peut-être pourquoi une telle convention est nécessaire aujourd'hui étant donné que les programmes sont écrits de nos jours à l'aide de langages évolués de programmation comme le C++ et le Java. Eh bien, c'est le compilateur qui doit le savoir. En effet, un programme peut être créé à partir de différents éléments compilés séparément. Pour compiler une fonction, le compilateur doit savoir quels registres sont utilisés pour transmettre des paramètres, quels registres sont utilisés pour renvoyer des résultats et quels registres doivent être conservés lors des appels de fonction. Ces règles d'utilisation des registres sont également appelées **conventions d'appel de fonction**.

L'électronique du processeur MIPS n'impose pas et ne vous empêchera pas d'ignorer ces règles, de ne pas conserver les registres, d'utiliser n'importe quel registre pour passer des paramètres et renvoyer des résultats. Cependant, si vous ignorez ces règles, vous rencontrerez facilement des problèmes et des bogues logiciels difficiles à détecter et éliminer.

Le tableau suivant présente la convention d'usage des registres des processeurs MIPS.

Numéro	Nom	Convention d'usage
$0	$0 ou $zero	Toujours égal à 0 – ne peut pas être modifié
$1	$at	Réservé (utilisé par l'outil Assembleur)
$2 - $3	$v0 - $v1	Arguments pour les **appels systèmes** (syscall) ou résultats retournés par une fonction
$4 - $7	$a0 - $a3	Arguments pour les **appels de fonctions**
$8 - $15 et $24 - $25	$t0 - $t9	Registres temporaires
$16 - $23	$s0 - $s7	Registres pour variables locales
$26 - $27	$k0 - $k1	Réservé pour le Mode Superviseur (Kernel mode) du processeur
$28	$gp	Global Pointer – Pointeur sur le segment data de la mémoire
$29	$sp	Stack Pointer – Pointeur de pile
$30	$fp ou $s8	Frame Pointer (on va ignorer ce registre dans ce cours)
$31	$ra	Utilisé implicitement par les instructions d'appel de fonctions **jal** et **jalr** pour la sauvegarde de l'adresse de retour

Contenus des registres qui doivent être conservés par la fonction appelée

Figure 6. Convention d'usage des registres du MIPS

Une fonction peut modifier les registres résultats **`$v0-$v1`**, les registres arguments **`$a0-$a3`** et les registres temporaires **`$t0-$t9`** sans sauvegarder leurs anciennes valeurs. Cependant, elle ne doit pas modifier les registres **`$s0-$s7`**, **`$gp`**, **`$sp`**, **`$fp`** et **`$ra`** sauf après avoir sauvegardé leurs anciennes valeurs en mémoire sur la pile.

Une fonction doit restaurer les états des registres **$s0–$s7**, **$gp**, **$sp**, **$fp** et **$ra** en récupérant leurs anciennes valeurs depuis la pile, juste avant de retourner à la fonction appelante. Les valeurs des registres **$sp** et **$fp** doivent être conservées si une nouvelle *trame-de-pile* est allouée par une fonction. Le registre **$ra** doit être conservé si une fonction fait appel à une autre fonction, car les instructions **jal** et **jalr** modifient le registre d'adresse de retour **$ra**.

Fonctions récursives dans MIPS

Une fonction récursive est une fonction qui fait appelle à elle-même. La figure 8 montre un exemple de fonction récursive (la fonction factorielle) en langage C et sa version en langage assembleur MIPS. Si (`n < 2`) alors la fonction s'achève et retourne à la fonction appelante. Il n'est pas nécessaire d'allouer une trame de pile dans ce cas. Cependant, si (`n >= 2`) alors la fonction factorielle alloue une trame de pile de 8 octets pour sauvegarder les registres **$a0** et **$ra** avant de s'appeler à nouveau.

Le registre **$a0** (argument **n**) est enregistré sur la pile car il est nécessaire de récupérer la valeur initiale de ce registre (utilisée à la ligne 12) après le retour de l'appel récursif (ce registre est modifié dans l'appel récursif à la ligne 8). Le registre **$ra** est sauvegardé sur la pile car sa valeur est modifiée par l'appel récursif (ligne 9).

```
    // version en langage C
    int fact(int n)
    {
      if (n<2) return 1;
      else return (n * fact(n-1));
    }


    # Version MIPS  (n ↔ $a0, n! ↔ $v0)
 0  fact:
 1        bge  $a0, 2, else    # aller à else si (n >= 2)
 2        li   $v0, 1          # $v0 = 1
 3        jr   $ra             # retour à la fonction appelante
 4  else:
 5        addi $sp, $sp, -8    # réserver une trame-de-pile de 8 octets
 6        sw   $a0, 0($sp)     # sauvegarder l'argument n
 7        sw   $ra, 4($sp)     # sauvegarder l'adresse de retour
 8        addi $a0, $a0, -1    # argument $a0 = n - 1
 9        jal  fact            # appel récursif fact(n-1)
10        lw   $a0, 0($sp)     # restaurer $a0 = n
11        lw   $ra, 4($sp)     # restaurer l'adresse de retour
12        mul  $v0, $a0, $v0   # $v0 = n * fact(n-1)
13        addi $sp, $sp, 8     # libérer la trame de pile
14        jr   $ra             # retour à la fonction appelante
```

Figure 7. Une fonction récursive en langage C et sa version en langage assembleur MIPS