Assembleur ARM avancé

Chouki TIBERMACINE

Chouki.Tibermacine@umontpellier.fr

Objectifs du chapitre

1. Donner l'intuition de la compilation des langages de haut niveau ;
2. Comprendre le mécanisme d'appel de fonction ;
3. Comprendre l'utilisation de la pile ;
4. Étudier la définition et l'utilisation de tableaux.

Haut niveau et bas niveau

Traduction assembleur

- Des constructions qui peuvent sembler élémentaires dans un langage tel que C ou Python (tests, boucles, fonctions) seront traduites par de nombreuses instructions assembleur
- Par ailleurs, des notions d'apparence pourtant simple n'ont pas d'équivalent. Par exemple, les variables de Python seront traduites en termes d'emplacements mémoire et de registres, une tâche non triviale (manipuler des adresses).

Conditionnelles

Code C

```
if (r1 < r2)
  r3 = r1;
else
  r3 = r2;
```

Code ARM

```
      cmp r1, r2
      blt then        // si r1 < r2 saut vers then
      mov r3, r2      // r3 := r2
      b done          // saut vers done
then: mov r3, r1      // r3 := r1
done:                 // suite du programme
```

Boucles

Code C

```
r2 = 0;
while (r1 > 0) {
  r2 = r2 + r1;
  r1 = r1 - 1;
}
```

Code ARM

```
       mov r2, #0     // r2 := 0
while: cmp r1, #0
       ble done       // si r1 <= 0 saut vers done
       add r2, r2, r1 // r2 := r2 + r1
       sub r1, r1, #1  // r1 := r1 - 1
       b while        // retour vers while
done:                 // suite du programme
```

Tableaux

Déclaration d'un tableau

En C : int a[100];

```
        .balign 4
a:      .skip 400
```

Accès à un élément du tableau

En C : a[3] = 3;

```
        ldr r1, =a
        mov r2, #3
        str r2, [r1, #+12]
```

Exercice

Écrire un programme qui remplit un tableau de 100 éléments avec les 100 premiers entiers positifs (0⟶99) et les affiche ensuite

Fonctions

Code C

```c
int succ (int x) {
  return x + 1;
}
```

Code ARM

```
/* Fonction succ */
succ:                   // r0 contient l'argument x
      add r0, r0, #1    // r0 contient le résultat
      bx lr             // retour à l'appelant (lr : registre qui contient
                        // l'adresse de l'instruction suivant l'appel)

/* Côté appelant */
      mov r0, #10       // On met la valeur de l'argument dans r0
      bl succ           // On appelle la fonction succ
```

Fonctions avec un corps qui modifie le registre lr

Code ARM

```
      .data
msg:  .string "Nouvelle valeur : %d\n"
      .text
succ: add r1, r0, #1
      ldr r0, =msg
      bl printf      // Instruction qui modifie lr
      mov r0, r1
      bx lr

      // Côté appelant :
      mov r0, #10
      bl succ
```

Problème avec l'instruction *bx lr* car lr a été modifié par *bl printf*

(Qu'est-ce qui va se passer à l'exécution ?)

Fonctions avec un corps qui modifie le registre lr

Code ARM

```
      .data
msg:  .string "Nouvelle valeur : %d\n"
      .text
succ: add r1, r0, #1
      ldr r0, =msg
      bl printf       // Instruction qui modifie lr
      mov r0, r1
      bx lr

      // Côté appelant :
      mov r0, #10
      bl succ
```

Problème avec l'instruction *bx lr* car lr a été modifié par *bl printf*
(boucle infinie, entre les 2 dernières instructions de succ)

Fonctions avec un corps qui modifie le registre lr

Solution : enregistrer en mémoire l'@ qui se trouve dans *lr*

```
        .data
return: .word 0 // Déclarer une variable return
msg:    .string "Nouvelle valeur : %d\n"
        .text
succ:   ldr r1, =return // Enregistrer dans r1 l'adresse de return
        str lr, [r1] // Enregistrer dans return l'adresse dans lr
        add r1, r0, #1
        ldr r0, =msg
        bl printf
        mov r0, r1
        ldr r1, =return // Récupérer de return l'adresse enregistrée
        ldr lr, [r1] // Remettre cette adresse dans lr
        bx lr
```

Variables locales et appels récursifs

- Si l'on veut déclarer des variables locales dans une fonction ?
- Si l'on veut faire des appels récursifs, comment on va gérer les adresses de retour (solution précédente : variable globale, écrasée à chaque appel) ?

Solution : utiliser la pile

- C'est quoi une pile ? c'est un espace en mémoire utilisé lors d'un appel de fonction
- L'adresse du sommet de la pile est stockée dans le registre *sp* (*stack pointer*)
- Cet espace est extensible "à souhait" par une fonction pour stocker des valeurs locales pour ses calculs
- Cet espace doit (par convention) être libéré par la fonction

Enregistrement de l'adresse qui se trouve dans lr dans la pile

Code ARM

```
foo: sub sp, sp, #8  // sp := sp - 8. On aggrandit la pile de 8 octets
                     // sommet de pile : adresse du dernier octet
                     // d'où la soustraction (sub). La pile grandit
                     // dans le sens des adresses décroissantes.
                     // Pourquoi 8 octets et non 4 ? standard ARM !
     str lr, [sp]    // On enregistre lr en sommet de pile
     ... // Code de la fonction foo
     ldr lr, [sp]    // On restitue lr depuis la pile
     add sp, sp, #8  // sp := sp + 8.
                     // pour réduire la taille de la pile de 8 octets
                     // et donc remettre le pointeur de pile à sa valeur
                     // d'origine
     bx lr
```

Variables locales

Code C

```c
int succ (int x) {
  int y;
  y = x + 1;
  return y;
}
```

Code ARM

```
succ: sub sp, sp, #8   // allocation mémoire pour la variable locale
      add r0, r0, 1    // opération
      str r0, [sp]     // enregistrement de la valeur dans la pile
      add sp, sp, #8   // désallocation
      bx lr            // retour
```

Convention d'appel

Règles

La convention d'appel régit la communication entre appelant et appelé lors de l'appel de fonction. Pour l'ARM, la convention proposée par le fabricant est la suivante :

- les arguments sont passés dans *r*0-*r*3, puis (s'il y en a plus de 4) sur la pile ;
- les valeurs de retour sont renvoyées dans *r*0 (et dans *r*1-*r*3 si renvoi de plusieurs/grandes valeurs). L'appelant ne doit pas faire d'hypothèse sur les valeurs dans ces registres, à moins qu'ils soient utilisés pour le passage de paramètres ;
- les registres restants (*r*4-*r*11 et *sp*), peuvent être modifiés par l'appelé mais leurs valeurs doivent être rétablies à la fin de la fonction (l'appelant fait l'hypothèse que ces registres ne sont pas altérés par l'appelé).

Exercice : Écrire un programme qui fait un appel à *printf* pour afficher *Hello World* ; ensuite debugger ce programme avec *gdb* pour voir l'état du registre r0 avant et après l'appel à *printf*

Sujet de TP

Exercices

- Écrire une fonction qui permute le contenu de deux variables entières de la zone de données au moyen d'une variable locale (en utilisant la pile) pour effectuer la permutation ;
- Écrire le code assembleur correspondant au code C suivant :

```
int sqr (int x) {
  return x * x;
}

int sum_sqr (int x, int y) {
  return sqr(x) + sqr(y);
}
```

Fonctions récursives

Code C

```c
int fact (int n) {
  if (n <= 0)
    return 1;
  else
    return n * fact (n - 1);
}
```

Fonctions récursives

Code ARM

```
fact:   sub sp, sp, #8
        str lr, [sp]    // enregistrer lr en sommet de pile
        cmp r0, #0
        ble fact0       // si r0 <= 0 saut vers fact0
        sub r0, #1      // r0 := r0 - 1
        mov r1, r0      // enregistrer r0 dans r1
        b fact          // appel récursif
endf:
        mul r0, r1      // r0 := r0 * r1 (r1 : résultat précédent)
        ldr lr, [sp]
        add sp, sp, #8
        bx lr           // retour à l'appelant

fact0:  mov r0, #1      // valeur de retour = 0
        b endf
```

Ce code est erroné. Où est l'erreur ?

Fonctions récursives

Pourquoi le code est-il erroné ?

- Parce que fact est récursive, plusieurs appels à la même fonction qui s'accumulent (appels actifs en même temps).
- L'appel récursif modifie *r*0 et *r*1 ; en fait, il en détruit le contenu. Or, celui-ci est utilisé après l'appel.
- Il faut donc sauvegarder le contenu de *r*0 avant l'appel, puis le restaurer après l'appel (comme pour *lr*).

Code ARM

```
fact:   sub sp, sp, #8
        str lr, [sp]
        cmp r0, #0
        ble fact0
        sub sp, sp, #8
        str r0, [sp]   // Enregistrer r0 dans la pile (empiler r0)
        sub r0, #1
        bl fact        // Appel récursif avec r0-1
endf:   ldr r1, [sp]   // Récupérer dans r1 la valeur empilée
        add sp, sp, #8
        mul r0, r1     // Multiplier r0 par r1 (résultat dans r0)
        ldr lr, [sp]
        add sp, sp, #8
        bx lr          // Retourner vers endf ou vers l'appelant de fact
fact0:  mov r0, #1
        sub sp, sp, #8
        str r0, [sp]   // Empiler la valeur 1 pour fact(0)
        b endf
```

Dérouler l'exécution d'un programme avec un appel : $fact(3)$

Sujet de TP -suite-

Exercices

- Écrire une fonction qui effectue récursivement la somme des n premiers entiers, où n est un entier passé en argument ;
- Écrire la fonction de Fibonacci :

$$fib(n) = \begin{cases} 1, \text{ si } n = 0, 1 \\ fib(n-1) + fib(n-2), \text{ sinon} \end{cases}$$