Université Joseph Fourier
Polytech' Grenoble

ALM Soft - Travail pratique No 3
Examen du code ARM produit par un compilateur C

Objectifs

Il s'agit dans cette expérience de concrétiser les notions de passage de paramètres en observant finement le travail effectué par le compilateur `arm-elf-gcc`.

On considère le programme `C` calculant le produit de deux entiers donné en annexe 1.

La procédure `multiplication` comporte trois paramètres :

– deux paramètres valeurs constituant les données du calcul et contenant la valeur des entiers dont on veut calculer le produit.
– un paramètre adresse dans lequel est rangé le résultat du calcul effectué par la procédure.

Par ailleurs, la procédure `multiplication` est programmée de façon récursive.

Travail Pratique

1. Copiez dans votre répertoire de travail le fichier `multrec.c` (présent sur le Moodle).
2. Comprendre l'algorithme de multiplication écrit sous forme récursive.
3. Compilez le programme `multrec.c` avec la commande suivante :
   `arm-elf-gcc -S -O0 multrec.c`.
   Vous obtenez un fichier `multrec.s` qui correspond au code ARM généré par le compilateur.
4. Complétez la compilation avec la commande suivante :
   `arm-elf-gcc -Wa,--gdwarf2,-L -o multrec multrec.s`
   Vous obtenez un fichier exécutable `multrec`
5. Vous pouvez alors exécuter le programme `multrec` : `arm-elf-run multrec`. Vérifiez que le programme réalise bien la multiplication de deux entiers positifs.
6. Lancer `arm-elf-gdb multrec` pour exécuter en pas à pas le programme `multrec`. Sous `gdb`, tapez ensuite : `target sim` puis `load`.

L'exécution en pas à pas a pour objectif de relever précisément le contenu de la pile à des endroits stratégiquement bien choisis. Vous pouvez par exemple suivre la démarche suivante :

1. Positionnez un point d'arrêt au début de la fonction `main` (`b main`).
2. Lancer l'exécution du programme (`run`).
3. Dessinez la pile au premier point d'arrêt. (L'exécution s'arrête en fait quelques instructions plus loin que main). N'oubliez pas de préciser les valeurs de `sp` (`r13`), `fp` (`r11`) et `ip` (`r12`).
4. Expliquez l'effet de la première instruction *stmfd sp !,* {*fp, lr*} qui suit l'étiquette *main.* Pourquoi cette instruction est-elle exécutée en début de programme ?

5. Dans `main`, que contient la mémoire à l'adresse égale à `fp - #12` avant l'appel à `scanf`. Et après ?
6. Montrez que c'est bien les adresses des variables (et non le contenu) n et t qui sont passées en paramètre à *scanf*.
7. Dans `main`, à quelles adresses sont rangées les variables `n`, `t` et `res` ? Dessinez la pile en faisant apparaitre les pointeurs `sp` et `fp`.
8. A quels endroits sont passés les paramètres effectifs lors du premier appel à la fonction `multiplication` ?
9. Dans la procédure `multiplication`, dessinez la pile :
   – à l'entrée de la fonction
   – avant l'instruction `cmp`
   – après un appel récursif
10. Dans `multiplication`, à quelles adresses sont rangés les paramètres `f` et `a` lors du calcul de `*f = a + r` ?
11. Quelle est la valeur de *pc* (Compteur programme) après l'exécution de l'instruction *ldmfd sp !,{fp, pc}* ? A quoi sert l'instruction précédente (*sup sp, fp, #4*) ?
12. Commentez précisément les instructions ARM résultant de la compilation.

Quelques commandes de gdb

Pour simuler en pas à pas, utilisez la commande `step` (`s`) et la commande `next` (`n`) lorsque vous ne voulez pas exécuter les instructions constituant le corps d'une procédure, par exemple pour l'exécution des `bl scanf` ou `bl printf`.

Pour demander la valeur contenue dans un registre, par exemple le pointeur de pile : `info reg $sp`.

Pour demander le contenu de la mémoire à partir d'une certaine adresse, utilisez la commande `x`. Par exemple, supposons que le pointeur de pile contienne la valeur `0x7fffe4` et que vous vouliez regarder le contenu de la pile à partir de cette adresse utilisez `x/6w 0x7fffe4` pour observer 6 mots de 32 bits à partir de l'adresse `0x7fffe4`.

Annexe 1

```
#include <stdio.h>

/* ************************************************************
 * multiplication :  donnee a, b : entier positif, un resultat f : entier
 *
 * apres l'execution de multiplication(a, b, f) : f = a * b
 * ************************************************************ */

void multiplication (int a, int b, int *f) {
int r;

  if (b==1)
     { *f = a; }
  else
     { multiplication (a, b-1, &r); *f = a + r; }
}

int main () {
int n, t, res;
```

```
   printf ("Donnez deux entiers positifs : \n");
   scanf ("%d %d", &n, &t);
        multiplication (n,t , &res);
   printf (" %d * %d = %d\n", n, t, res);
}
```

Annexe 2

```
     .file "multrec.c"
.text
.align 2
.global multiplication
.type multiplication, %function
multiplication:
@ args = 0, pretend = 0, frame = 16
@ frame_needed = 1, uses_anonymous_args = 0
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #16
str r0, [fp, #-12]
str r1, [fp, #-16]
str r2, [fp, #-20]
ldr r3, [fp, #-16]
cmp r3, #1
bne .L2
ldr r3, [fp, #-20]
ldr r2, [fp, #-12]
str r2, [r3, #0]
b .L4
.L2:
ldr r3, [fp, #-16]
sub r2, r3, #1
sub r3, fp, #8
ldr r0, [fp, #-12]
mov r1, r2
mov r2, r3
bl multiplication
ldr r2, [fp, #-8]
ldr r3, [fp, #-12]
add r2, r2, r3
ldr r3, [fp, #-20]
str r2, [r3, #0]
.L4:
sub sp, fp, #4
ldmfd sp!, {fp, pc}
.size multiplication, .-multiplication
.section .rodata
.align 2
.LC0:
.ascii "Donnez deux entiers positifs : \000"
.align 2
.LC1:
.ascii "%d %d\000"
.align 2
.LC2:
.ascii " %d * %d = %d\012\000"
.text
```

```
.align 2
.global main
.type main, %function
main:
@ args = 0, pretend = 0, frame = 12
@ frame_needed = 1, uses_anonymous_args = 0
stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #12
ldr r0, .L7
bl puts
sub r2, fp, #8
sub r3, fp, #12
ldr r0, .L7+4
mov r1, r2
mov r2, r3
bl scanf
ldr r1, [fp, #-8]
ldr r2, [fp, #-12]
sub r3, fp, #16
mov r0, r1
mov r1, r2
mov r2, r3
bl multiplication
ldr r1, [fp, #-8]
ldr r2, [fp, #-12]
ldr r3, [fp, #-16]
ldr r0, .L7+8
bl printf
sub sp, fp, #4
ldmfd sp!, {fp, pc}
.L8:
.align 2
.L7:
.word .LC0
.word .LC1
.word .LC2
.size main, .-main
.ident "GCC: (GNU) 4.4.0"
```