ARCHITURE DES MICROPROCESSEURS

TD5 : RISC-V Application Binary Interface

4. 1. Introduction

Dans ce dernier TD, nous allons explorer l'ABI RISC- V, afin de mieux appréhender le code généré par le compilateur et de pouvoir suivre une session de debug d'un code assembleur. Ce qui est vu dans ce TD peut être extrapolé pour d'autres architectures microprocesseurs (ARM , i686 x86-64 ...)

4. 2. Mise en place de la chaîne de compilation

La chaîne de compilation est déjà installée sur la VM.

Le compilateur

Il s'agit d'une chaîne de compilation croisée, c'est à dire s'exécutant sur x86-64 mais produisant du code binaire RISC-V.

3. 1. Dans un terminal exécutez la commande suivante **: riscv-none-embed-gcc --help**

3. 2. Quelle option permet de :

- Lancer seulement le préprocesseur C ?
- De compiler et d'assembler le code source C sans faire d'édition de lien?
- De compiler seulement le code source C ?
- De spécifier le nom du fichier de sortie ?

3. 3. L'option **–v --help** permet d'avoir une aide plus détaillée.

Que permettent les options :

- -debug
- -march
- -fverbose-asm
- -fomit-frame-pointer
- -mabi

(indication : utilisez | et grep pour filtrer la sortie de l'aide)

4. 4. Exploration du code assemblé et exploration de la structure de la pile

4. 1. Compilez seulement (voir question précédente) le programme suivant que vous placerez dans un fichier main1.c. Le fichier de sortie sera nommé main1.s.

```
int fct1( int a, int b, int c, int d, int e, int f ,int g, int h )
 {
   return(a+b+c+d+e+f+g+h);
 }
```

4. 2. De quel type de fonction s'agit-il ? (fonction feuille ou non, nombre de variables locales, nombre de paramètres en entrée ...)

Fonction feuille, pas de variables locales avec 5 paramètres

4. 3. La réponse à la question qui précède va vous permettre d'identifier la structure que doit avoir la pile.

Pour compiler le code :

riscv-none-embed-gcc -mabi=ilp32 -march=rv32i main.c -S -fverbose-asm -fomit-frame-pointer

Fonction feuille :

- Absence de la zone de sauvegarde de paramètres puisqu'on n'appelle pas de fonction
- pas de sauvegarde de $ra

Pas de variables locales :

- Absence de la zone pour l'allocation des variables locales

A priori la fonction n'est pas complexe et ne doit pas nécessiter l'usage de registres à sauvegarder :

- Absence de la zone pour la sauvegarde des registres s0 à s11.

Donc, a priori il n'y a pas besoin d'allouer d'espace sur la pile pour l'exécution de la fonction fct2.

Cependant nous avons vu à la question 4.3 que sans optimisation de compilation, le compilateur utilise $fp systématiquement. Pour éviter cet usage systématique il faut utiliser l'option : -fomit-frame-pointer .

```
fct1:
        addi    sp,sp,-32       #,,
        sw      a0,28(sp)       # a, a
        sw      a1,24(sp)       # b, b
        sw      a2,20(sp)       # c, c
        sw      a3,16(sp)       # d, d
        sw      a4,12(sp)       # e, e
        sw      a5,8(sp)        # f, f
        sw      a6,4(sp)        # g, g
        sw      a7,0(sp)        # h, h
# main.c:15:    return(a+b+c+d+e+f+g+h);
        lw      a4,28(sp)               # tmp80, a
        lw      a5,24(sp)               # tmp81, b
        add     a4,a4,a5        # tmp81, _1, tmp80
# main.c:15:    return(a+b+c+d+e+f+g+h);
        lw      a5,20(sp)               # tmp82, c
        add     a4,a4,a5        # tmp82, _2, _1
# main.c:15:    return(a+b+c+d+e+f+g+h);
        lw      a5,16(sp)               # tmp83, d
        add     a4,a4,a5        # tmp83, _3, _2
# main.c:15:    return(a+b+c+d+e+f+g+h);
        lw      a5,12(sp)               # tmp84, e
        add     a4,a4,a5        # tmp84, _4, _3
# main.c:15:    return(a+b+c+d+e+f+g+h);
        lw      a5,8(sp)                # tmp85, f
        add     a4,a4,a5        # tmp85, _5, _4
# main.c:15:    return(a+b+c+d+e+f+g+h);
        lw      a5,4(sp)                # tmp86, g
        add     a4,a4,a5        # tmp86, _6, _5
# main.c:15:    return(a+b+c+d+e+f+g+h);
        lw      a5,0(sp)                # tmp87, h
        add     a5,a4,a5        # tmp87, _15, _6
# main.c:16:  }
        mv      a0,a5           #, <retval>
        addi    sp,sp,32        #,,
        jr      ra                      #
        .size   fct1, .-fct1
```

On crée un espace de 32 octets sur la pile qui correspond à l'espace nécessaire pour stocker les huit paramètres

La taille de la pille est bien un multiple de 16

Les huit premiers paramètres sont passés par registres : a0..a7

On effectue l'addition des huit paramètres

On passe le résultat dans a0

On libère l'espace sur la pile

On retourne à l'appelant

4. 4.
Dessinez la zone mémoire de la pile : les adresses du bas de la pile au haut de la pile et leur contenu.

adresse	contenu
sp+32	
sp+28	
sp+24	
sp+20	
sp+16	
sp+12	
sp+8	
sp+4	
sp	

4. 5. Compilez seulement (voir question précédente) le programme suivant que vous placerez dans un fichier main2.c. Le fichier de sortie sera nommé main2.s :

```
int fct2( int a, int b, int c, int d, int e, int f ,int g, int h, int i, int j)
 {
   int x;
   int y;

   x=a+b;
   y=i+j;

   return(a+b+c+d+e+f+g+h+i+j+x+y);
 }
```

Mêmes questions 1 à 3 que précédemment. Faites apparaître les « frame » des fonctions (appelante et appelée)

4. 6. Sans utiliser l'option **-fomit-frame-pointer** compilez seulement le programme suivant que vous placerez dans un fichier main3.c. Le fichier de sortie sera nommé main3.s:

```
#include <alloca.h>
int fct2( int a, int b, int c, int d, int e, int f ,int g, int h, int i, int j)
 {
   int x;
   int y;

   x=a+b;
   y=i+j;

   return(a+b+c+d+e+f+g+h+i+j+x+y);
 }

void fct3(unsigned  char i){
            unsigned int *ptr;

            ptr=alloca(4);
            *ptr=fct2(1,2,3,4,5,6,7,8,9,10);

            }
```

Comment est allouée la variable ptr ?
Quel va être la structure de la pile lors de l'appel de cette fonction ?

4. 7. De nouveau, compilez seulement le programme suivant que vous placerez dans un fichier main4.c. Le fichier de sortie sera nommé main4.s:

```
int global[128];

int initialized_global=0x5555AAAA;

int fct2( int a, int b, int c, int d, int e, int f ,int g, int h, int i, int j)
 {
   int x;
   int y;

   x=a+b;
   y=i+j;

   return(a+b+c+d+e+f+g+h+i+j+x+y);
 }

int long  main()
 {
  int A=1;
  int B=2;
  int C=3;
  int D=4;
  int E=5;
  int F=6;
  int G=7;
  int H=8;
  int I=9;
  int J=10;


  initialized_global=0x12345678;
  global[10]=fct2(0,1,2,3,4,5,6,7,8,9);
  return(0);
 }
```

Examinez le code assembleur qui permet d'accéder aux variables **global et initialized_global** , commentez.

5 Edition de liens :

Ajoutez la déclaration de la variable globale suivante dans main.c :
Récupérez les fichiers : **td5.ld et crt.s**.
Assemblez le c runtime :

```
riscv-none-embed-gcc -mabi=ilp32 -march=rv32i  -no-common  -fomit-frame-pointer -c -x assembler crt.s -o crt.o
```

Compilez et assemblez le main.c :
riscv-none-embed-gcc -mabi=ilp32 -march=rv32i -no-common -c main4.c -fverbose-asm -fomit-frame-pointer -o main4.o

Construisez le binaire résultant :
riscv-none-embed-ld –relax -no-common -T td5.ld crt.o main4.o -Map td5.map -o main4.elf

Désassemblez le binaire résultant **: riscv-none-embed-objdump -SD main4.elf > main4.s**

5.1 Examinez td5.map , à quelles adresses se trouve les variables globales : global et initialized_global ?
5.2 Examinez main.s :

- à quelles valeurs sont initialisés les registres $sp et $gp ?
- à quelle adresse se trouvent le c runtime, la fonction main ?
- Examinez le code assembleur qui permet d'accéder aux variables **global et initialized_global** , commentez.