Compte-rendu du TD5 :

1) Introduction

Ce dernier TD va nous permettre d'explorer l'ABI RISC-V, afin de pouvoir mieux appréhender le code généré par le compilateur et de pouvoir suivre une session de debug d'un code assembleur.

2) Mise en place de la chaîne de compilation

Ajout du chemin vers les exécutables via la commande
source /soft/MENTOR/config_bashrc/.bashrc_riscv

3) Le compilateur

- L'option qui permet de lancer le processeur est : `-E`
- L'option qui permet de compiler et assembler le code source C sans faire d'édition de lien est : `-c`
- L'option qui permet de seulement compiler le code source C est : `-S`
- L'option qui permet de spécifier le nom du fichier de sortie est : `-o`

```
-E                       Preprocess only; do not compile, assemble or link.
-S                       Compile only; do not assemble or link.
-c                       Compile and assemble, but do not link.
-o <file>                Place the output into <file>.
```

- L'option `-g` permet d'accéder aux informations relatives au debug.
- L'option `-march` permet d'accéder au type d'architecture utilisé.
- L'option `-fverbose-asm` fournit du code assembleur commenté (traduction du programme).
- L'option `-fomit-frame-pointer` permet de
- L'option `-mabi` spécifie la convention d'appel ABI des entiers et flottants.

4) Exploration de la structure assemblée et exploration de la structure de la pile

```
 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);
}
```

La fonction étudiée est une fonction feuille, puisqu'elle n'appelle aucune autre fonction. Elle ne dispose pas de variables internes, mais a 8 paramètres en entrée.

```
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:3:     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:3:     return(a+b+c+d+e+f+g+h);
        lw      a5,20(sp)               # tmp82, c
        add     a4,a4,a5        # tmp82, _2, _1
# main.c:3:     return(a+b+c+d+e+f+g+h);
        lw      a5,16(sp)               # tmp83, d
        add     a4,a4,a5        # tmp83, _3, _2
# main.c:3:     return(a+b+c+d+e+f+g+h);
        lw      a5,12(sp)               # tmp84, e
        add     a4,a4,a5        # tmp84, _4, _3
# main.c:3:     return(a+b+c+d+e+f+g+h);
        lw      a5,8(sp)                # tmp85, f
        add     a4,a4,a5        # tmp85, _5, _4
# main.c:3:     return(a+b+c+d+e+f+g+h);
        lw      a5,4(sp)                # tmp86, g
        add     a4,a4,a5        # tmp86, _6, _5
# main.c:3:     return(a+b+c+d+e+f+g+h);
        lw      a5,0(sp)                # tmp87, h
        add     a5,a4,a5        # tmp87, _15, _6
# main.c:4: }
        mv      a0,a5   #, <retval>
        addi    sp,sp,32        #,,
        jr      ra              #
        .size   fct1, .-fct1
        .ident  "GCC: (xPack GNU RISC-V Embedded GCC, 64-bit) 8.3.0"
```

Espace nécessaire pour stocker les 8 paramètres

8 premiers paramètres passés par registre (a0…a7)

On passe le résultat dans a0

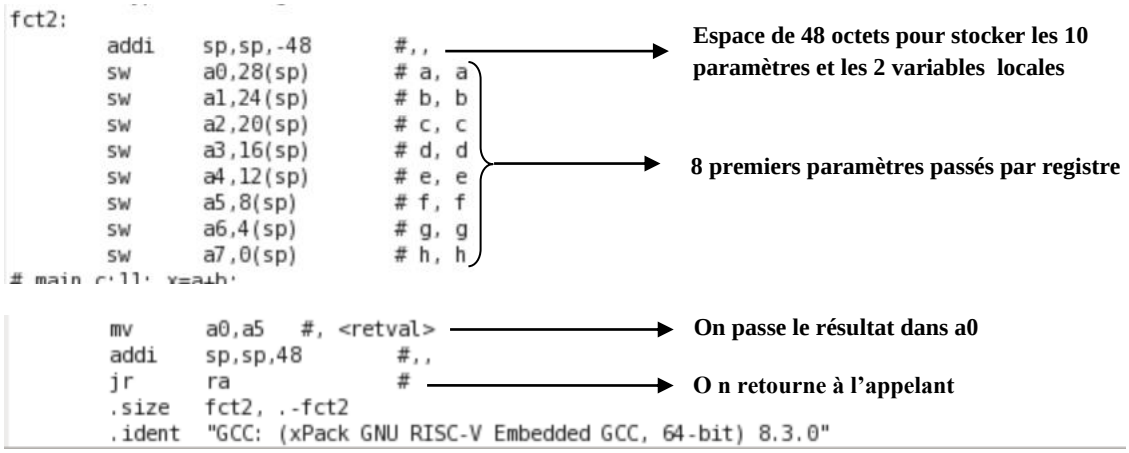
O n retourne à l'appelant

L'aspect de la zone mémoire de la pile est le suivant :

Adresse	Contenu
SP+28	a
SP+24	b
SP+20	c
SP+16	d
SP+12	e
SP+8	f
SP+4	g
SP	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);
}
```

Cette fois-ci encore, il s'agit d'une fonction feuille. Cette fois-ci, elle dispose de 10 paramètres en entrée, et 2 variables locales. Les 10 paramètres dépassant le nombre de registres à disposition (8), la zone mémoire de la pile aura la disposition suivante :



Adresse	Contenu	Appelant	Appelé
+4 SP+52	j		×
+0 SP+48	i		×
-4 SP+44	y	×	
-8 SP+40	x	×	
-12 SP+36			
-16 SP+32			
-20 SP+28	a		×
-24 SP+24	b		×
-28 SP+20	c		×
-32 SP+16	d		×
-36 SP+12	e		×
-40 SP+8	f		×
-44 SP+4	g		×
-48 SP	h		×

En effet, puisque la fonction ne fait appel qu'à 2 variables locales, elle n'ajoute que 4x32 = 128 bits puisque le nombre de variables est inférieur à 4. Les paramètres i et j sont donc ajoutés en libérant l'espace mémoire précédemment alloué aux variables locales et non utilisé.

```
#include <alloca.h>
void fct3(unsigned char i){
unsigned int *ptr;
ptr=alloca(4);
*ptr=fct2(1,2,3,4,5,6,7,8,9,10);
}
```

Dans cette fonction, la variable `ptr` est allouée via une fonction externe, `alloca(int )`. La fonction fct3 n'est donc pas une fonction feuille, puisqu'elle appelle une fonction externe.

```
fct3:
        addi    sp,sp,-64       #,,
        sw      ra,60(sp)       #,
        sw      s0,56(sp)       #,
        addi    s0,sp,64        #,,
        mv      a5,a0   # tmp74, i
        sb      a5,-33(s0)      # tmp75, i
# main.c:21: ptr=alloca(4);
        addi    sp,sp,-16       #,,
        addi    a5,sp,8 #, tmp76,
        addi    a5,a5,15        #, tmp77, tmp76
        srli    a5,a5,4 #, tmp78, tmp77
        slli    a5,a5,4 #, tmp79, tmp78
        sw      a5,-20(s0)      # tmp79, ptr
# main.c:22: *ptr=fct2(1,2,3,4,5,6,7,8,9,10);
        li      a5,10           # tmp80,
        sw      a5,4(sp)        # tmp80,
        li      a5,9            # tmp81,
        sw      a5,0(sp)        # tmp81,
        li      a7,8            #,
        li      a6,7            #,
        li      a5,6            #,
        li      a4,5            #,
        li      a3,4            #,
        li      a2,3            #,
        li      a1,2            #,
        li      a0,1            #,
        call    fct2            #
        mv      a5,a0   # _1,
        mv      a4,a5   # _2, _1
```

Utilisation de alloca() pour le pointeur ptr

Utilisation de la fonction fct2()

```
# main.c:22: *ptr=fct2(1,2,3,4,5,6,7,8,9,10);
        lw      a5,-20(s0)              # tmp82, ptr
        sw      a4,0(a5)        # _2, *ptr_5
# main.c:23: }
        nop
        addi    sp,s0,-64       #,,
        lw      ra,60(sp)               #,
        lw      s0,56(sp)               #,
        addi    sp,sp,64        #,,
        jr      ra              #
        .size   fct3, .-fct3
        .ident  "GCC: (xPack GNU RISC-V Embedded GCC, 64-bit) 8.3.0"
```

On libère l'espace sur la pile

On retourne à l'appelant

5) Edition de liens

5.1) En examinant le programme, on remarque que la variable `global` est initialisée dans le fichier `.bss`, tandis que la variable `initialized_global` est initialisée dans le fichier `.data`, conformément aux explications du cours.

```
.bss            0x0000000010000000      0x200
                0x0000000010000000              __bss_start = .
                0x0000000010000000              _gp = .
 *(.bss)
 .bss           0x0000000010000000        0x0 crt.o
 .bss           0x0000000010000000        0x0 main.o
 *(.sbss)
                0x0000000010000000              __bss_end = .
 COMMON         0x0000000010000000      0x200 main.o
                0x0000000010000000              global


.init           0x0000000010000200        0x0 load address 0x00000000000002c8
                0x0000000010000200              __data_start = .
 *(.data)
 .data          0x0000000010000200        0x0 crt.o
 .data          0x0000000010000200        0x0 main.o
                0x0000000010000200              __data_end = .

.sdata          0x0000000010000200        0x4 load address 0x00000000000002c8
 .sdata         0x0000000010000200        0x4 main.o
                0x0000000010000200              initialised_global
```

5.2)