Compte Rendu TP3&4 INF401

NGUYEN Minh Huy
LUU Nguyen Phuoc Loc

3.1.Déclaration de données en langage d'assemblage

donnees.s:

```
    .data
aa: .byte 65   @ .byte 0x41
oo: .byte 15   @ .byte  0x0f
cc: .asciz "bonjour"
rr: .byte 66   @ .byte 0x42
    .byte 3
T: .hword 0x1122
    .hword 0x3456
    .hword 0xfafd
xx: .byte 65
```

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-objdump -j .data -s donnees.o

donnees.o:     file format elf32-bigarm

Contents of section .data:
 0000 410f626f 6e6a6f75 72004203 11223456  A.bonjour.B.."4V
 0010 fafd41                               ..A
```

Les informations sont affichées en **hexadécimal (base 16)**, ce qui est standard pour l'affichage des données binaires dans un format lisible

Nombre d'octets par ligne :

- La première ligne contient **16 octets : 410f626f 6e6a6f75 72004203 11223456**
- La deuxième ligne contient **4 octets : fafd41**

Correspondance avec la zone .data :

- 41 → A(aa: .byte 65)
- 0F → 15(oo: .byte 15)
- 62 6F 6E 6A 6F 75 72 → "bonjour"
- 00 → \0 (terminaison de chaîne)
- 42 → B(rr: .byte 66)
- 03 → .byte 3 du rr
- 11 22 → T[0]: .hword 0x1122
- 34 56 → T[1]: .hword 0x3456
- FA FD → T[2]: .hword 0xfafd
- 41 → A(xx: .byte 65)

La chaine de caracteres(“bonjour”) est codée en ASCII

“bonjour” : 62 6f 6e 6a 6f 75 72 **00**

À la fin de la chaine, il y a un caractère nul (0x00)

donnees2.s:

```
    .data
aa: .byte 65   @ .byte 0x41
oo: .word 15   @ .byte  0x0f
cc: .asciz "bonjour"
rr: .byte 66   @ .byte 0x42
    .word 3
T: .word 0x1122
   .word 0x3456
   .word 0xfafd
xx: .word 65
```

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-
objdump -j .data -s donnees2.o
donnees2.o:     file format elf32-bigarm

Contents of section .data:
 0000 41000000 0f626f6e 6a6f7572 00420000  A....bonjour.B..
 0010 00030000 11220000 34560000 fafd0000  .....".."..4V.....
 0020 0041                                 .A
```

Maintenant, tous les entiers sont sur 32 bits

Correspondance avec la zone .data **:**

- 41 → A(aa: .byte 65)
- **0000000F → 15(oo: .word 15)**
- 62 6F 6E 6A 6F 75 72 → "bonjour"
- 00 → \0 (terminaison de chaîne)
- 42 → B(rr: .byte 66)
- **00000003 → .byte 3 du rr**
- **0000 1122 → T[0]: .word 0x1122**
- **3456 0000→ T[1]: .hword 0x3456**
- **FAFD 0000→T[2]: .hword 0xfafd**
- 41 →A(xx: .byte 65)

3.2 Acces a la memoire : echange memoire/registres

3.2.1 Lecture d'un mot de 32 bits

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run accesmem
0001f460
0000010a
```

-0001f460 : Cette instruction charge l'adresse de la variable xx dans le registre r5
-0000010a: Cette instruction charge dans le registre r6 le contenu de la mémoire à l'adresse pointée par r5

3.2.2 Lecture de mots de tailles differentes

```
   .data
D1:    .word  266
D2:    .hword 42
D3:    .byte 12

   .text
   .global main
main:
     LDR r3, LD_D1
     LDR r4, [r3]

     LDR r5, LD_D2
     LDRH r6, [r5]

     LDR r7, LD_D3
     LDRB r8, [r7]

  @ impression du contenu de r3
    MOV r1, r3
    BL EcrHexa32

  @ impression du contenu de r4
    MOV r1, r4
    BL EcrNdecimal32

  @ impression du contenu de r5
    MOV r1, r5
    BL EcrHexa32

  @ impression du contenu de r6
    MOV r1, r6
    BL EcrNdecimal16

  @ impression du contenu de r7
    MOV r1, r7
    BL EcrHexa32
```

```
 @ impression du contenu de r8
   MOV r1, r8
   BL EcrNdecimal8

fin:  B exit  @ terminaison immédiate du processus (plus tard on saura faire mieux)

LD_D1:   .word D1
LD_D2:   .word D2
LD_D3:   .word D3
```

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run
accesmem2
0001f494
266
0001f498
42
0001f49a
12
```

Remarque du Poly : les adresses sont toujours representees sur 32 bits.
-L'adresse de **D1**: **.word 266: 0001f494**
-L'adresse de **D2**: **.hword 42: 0001f498**
-L'adresse de **D3**: **.byte 12: 0001f49a**

Explication des différences :
Alignement en mémoire (On va rencontrer ce cas dans les exos suivants)
Les valeurs sont placées dans la mémoire en respectant leur alignement :
-.word (**4 octets)** doit être aligné sur une adresse multiple de 4 (**0x0001F494** est bien un multiple de 4)
-.hword (**2 octets)** doit être aligné sur une adresse multiple de 2 (**0x0001F498** est bien un multiple de 2)
-.byte (**1 octet**) peut être placé à n'importe quelle adresse.
Occupation mémoire
-D1 occupe 4 octets (0x0001F494 à 0x0001F497)
-D2 occupe 2 octets (0x0001F498 à 0x0001F499)
-D3 occupe 1 octet (0x0001F49A)

3.2.3 Ecriture en memoire

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run ecrmem
00000000
0
fffffff6
-10
00000000
0
0000fff6
-10
00000000
0
000000f6
-10
```

On obtient : **DW (32 bits), DH (16 bits), DB (8 bits)** ont la valeur initiale de 0 et la valeur après la modification est -10**.**

4.1 Un premier programme en langage d'assemblage

```
    .data
cc: @ ne pas modifier cette partie
    .byte 0x42
    .byte 0x4f
    .byte 0x4e
    .byte 0x4a
    .byte 0x4f
    .byte 0x55
    .byte 0x52
    .byte 0x00     @ code de fin de chaine
    @ la suite pourra etre modifiee
    .word 12
    .word 0x11223344
av: .asciz "au revoir..."

    .text
    .global main
main:

@ impression de la chaine de caractere d'adresse cc
    ldr r1, LD_cc
    bl EcrChaine

@ impression de la chaine "au revoir..."
    ldr r1, LD_av
    bl EcrChaine

@ modification de la chaine d'adresse cc
    ldr r0, LD_cc
    ldrb r1, [r0]
    add r1, r1, #0x20
    strb r1, [r0]
    ldrb r1, [r0, #1]
    add r1, r1, #0x20
    strb r1, [r0, #1]
    ldrb r1, [r0, #2]
    add r1, r1, #0x20
    strb r1, [r0, #2]
    ldrb r1, [r0, #3]
    add r1, r1, #0x20
    strb r1, [r0, #3]
    ldrb r1, [r0, #4]
    add r1, r1, #0x20
    strb r1, [r0, #4]
    ldrb r1, [r0, #5]
    add r1, r1, #0x20
```

```
   strb r1, [r0, #5]
   ldrb r1, [r0, #6]
   add r1, r1, #0x20
   strb r1, [r0, #6]

@ impression de la chaine cc modifiee
   ldr r1, LD_cc
   bl EcrChaine

fin: B exit  @ terminaison immédiate du processus (plus tard on saura faire mieux)

LD_cc: .word cc
LD_av: .word av
```

Rappel du TD1 : Pour convertir une lettre majuscule en minuscule en ASCII, on ajoute 0x20 (32 en décimal) à la valeur de cette lettre

ldr r0, LD_cc : Charge l'adresse de la chaîne de caractères dans le registre r0. LD_cc est une étiquette qui pointe vers l'adresse de la chaîne.

ldrb r1, [r0] : Charge le premier octet (caractère) de la chaîne dans le registre r1.

add r1, r1, #0x20 : Ajoute 0x20 (32 en décimal) à la valeur du registre r1. En ASCII, cela convertit une lettre majuscule en minuscule.

strb r1, [r0] : Stocke la nouvelle valeur de r1 (le caractère modifié) à l'adresse pointée par r0.

Les instructions suivantes répètent ce processus pour les caractères suivants de la chaîne, en utilisant des décalages pour accéder à chaque caractère successivement :

ldrb r1, [r0, #1] : Charge le deuxième octet de la chaîne.

add r1, r1, #0x20 : Convertit le caractère en minuscule.

strb r1, [r0, #1] : Stocke le caractère modifié.

ldrb r1, [r0, #2] : Charge le troisième octet de la chaîne.

add r1, r1, #0x20 : Convertit le caractère en minuscule.

strb r1, [r0, #2] : Stocke le caractère modifié.

Sortie de caracteres:

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run
caracteres
BONJOUR
au revoir...
bonjour
```

4.2 Alignements et ”petits bouts”

4.2.1 Questions d’alignements

Sortie de alignements1

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run
alignements1
1
2
4
67240205
8
```

-On note que en décimal, 0x0A0B0C0D = 168 496 141, cela contredit le résultat ci-dessus.
-Le probleme vient du fait que les mots de 32 bits doivent etre placés à des adresses multiples de 4
-On ajoute .balign 4 juste avant la declaration de l’entier a, et appelez ce nouveau programme alignements2.s

Sortie de alignements2

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run
alignements2
1
2
4
168496141
8
```

La valeur de a maintenant est “alignée”.
De même manière, on crée 2 programmes alignement3 et alignements4

Sortie de alignements3

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run
alignements3
17
22033
```

Sortie de alignements4

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run
alignements4
17
13398
```

0x3456 en hexadecimal est 13398, mais le résultat dans alignements3 n’est pas juste

4.2.2 Questions de ”petits bouts”

Par convention :

Big-endian : Le byte ayant la valeur la plus élevée est stocké à l'adresse la plus basse. Cela signifie que le premier élément dans la chaîne hexadécimale sera le byte ayant la valeur la plus élevée

Little-endian : Le byte ayant la valeur la plus basse est stocké à l'adresse la plus basse. Cela peut entraîner un stockage des bytes de manière "inversée", avec le premier byte ayant la valeur la plus basse

On a modifié accesmem et accesmem2 pour qu’ils affichent tous les adresses afin de trouver facilement ses adresses

Sortie de accesmem (en little-endian)

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run
accesmem
0001f498
0000010a
0001f494
00000018
0001f49c
0000002a
```

Sortie arm-eabi-objdump -j .data -s accesmem

```
1f448 00000000 00000000 00000000 00000000  ................
1f458 00000000 18000000 0a010000 2a000000  ............*...
1f468 25630a00 25630025 73002573 0a002530  %c..%c.%s.%s..%0
```

On obtient en observant en **big-endian**:

- xx : 266 (0x0a010000)
- aa : 24 (0x00000018)
- bb : 42 (0x2a000000)

Sortie de accesmem (en little-endian)

```
luunpl@im2ag-turing-01:[~/University/Semestre_4/401_INF/TP/TP3&4]: arm-eabi-run
accesmem2
0001f49c
266
0000010a
0001f4a0
42
002a
0001f4a2
12
0c
```

Sortie arm-eabi-objdump -j .data -s accesmem

```
1f488 00000000 00000000 00000000 00000000  ................
1f498 00000000 0a010000 2a000c00 25630a00  ........*...%c..
1f4a8 25630025 73002573 0a002530 38780a00  %c.%s.%s..%08x..
```

On obtient :

- **D1 : 266 (0a010000)**
- **D2 : 42 (2a00)**
- **D3 : 12 (0c)**