## ***3.1***

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: 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
```

C'est à comprendre en 3 parties; prenons l'exemple sur la première ligne de donnée:
-Les 2 premiers bytes indiquent les adresses mémoires en base hexadécimal.
-Les 16 bytes suivant représentent le contenu du fichier en base hexadécimal.
-Et les caractères à droite, sont l'interprétation si cela est possible, bien entendu, en caractère ASCII.

Passons à la correspondance:

| **Base HEXA** | **Données représentées** |
|---|---|
| 41 | caractère 'A' |
| 0f | l'entier 15 sur un octet mais non représentable par un caractère donc un point est affiché |
| 62 6f 6e 6a 6f 75 72 *00* | chaîne de caractère 'bonjour', *la fin de la chaîne est signalée par un octet nul non représentable par un caractère.* c'est la même représentation qu'en C. |
| 42 03 | caractère 'B' et l'entier 3 sur un octet |
| 11 22  34 56  fa fd | Le tableau T de trois entiers sur 16 bits représenté comme suit : **."  4V  ..** |
| 41 | l'entier 65 sur un octet |

Nouvelle représentation pour données2.O

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-objdump -j .data -s donnees2.o

donnees2.o:     file format elf32-bigarm

Contents of section .data:
 0000 00000041 0000000f 626f6e6a 6f757200  ...A....bonjour.
 0010 00000042 00000003 11223456 fafd0000  ...B....."4V....
 0020 0041                                 .A
```

Maintenant

| **Base HEXA** | **Données représentées** |
|---|---|
| 00000041 | caractère ‘A’ précédé de 3 points |
| 0000000f | l’entier 15 sur quatre octets mais non représentable par un caractère donc un point est affiché et précédé de 3 points. |
| 626f6e6a 6f757*200* | chaîne de caractère ‘bonjour’, *la fin de la chaîne est signalée par un octet nul non représentable par un caractère.* C’est la même représentation qu’en C. |
| 00000042 00000003 | caractère ‘B’ précédé de 3 points et l’entier 3 précédé de 3 points sur quatre octets |
| 11223456 fafd | Le tableau T de trois entiers sur 16 bits représenté comme suit : **.”  4V  ..** |
| 00000041 | l’entier 65 précédé de 3 points sur quatre octet |

***3.2***

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c es.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c accesmem.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -o accesmem accesmem.o es.o
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-run accesmem
0001f440
0000010a
```

Llors de l'exécution de la fonction EcrHexa32, les valeurs affichées sont 0001f440 et 0000010a, elles correspondent aux contenus actuels du registre r1.
La première valeur représente l'adresse associée à l'étiquette xx, tandis que la seconde indique la valeur stockée à cette adresse, (266 en hexadécimal). Les zéros en amont résultent de la codification sur 32 bits de l'entier.

***3.2.2***

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

    .text
    .global main
main:
        LDR r3, LD_D1
        LDR r4, [r3]
        MOV r1, r3
        BL EcrHexa32
        MOV r1, r4
        BL EcrNdecimal32

        LDR r5, LD_D2
        LDRH r6, [r5]
        MOV r1, r5
        BL EcrHexa32
        MOV r1, r6
        BL EcrNdecimal16

        LDR r7, LD_D3
        LDRB r8, [r7]
        MOV r1, r7
        BL EcrHexa32
        MOV r1, r8
        BL EcrNdecimal8


fin:    BX LR

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

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c es.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c accesmem2.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -o accesmem2 accesmem2.o es.o
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-run accesmem2
0001f474
266
0001f478
42
0001f47a
12
```

Les chiffres héxa et décimaux affichés dans les deux premières lignes représentent respectivement l'adresse mémoire et la valeur de la variable D1. Donc, la valeur 266 de D1 est enregistrée à l'adresse mémoire 0001f474.

Les chiffres héxa et décimaux affichés dans les deux lignes intermédiaires correspondent à l'adresse mémoire et à la valeur de la variable D2. La valeur 42 de D2 est stockée à l'adresse mémoire 0001f478.
Il est pertinent de noter que la valeur de la variable précédente, D1, étant codée sur 4 octets, explique que l'adresse de D2 soit celle de D1 + 4 .

Les chiffres héxa et décimaux affichés dans les deux dernières lignes désignent l'adresse mémoire et la valeur de la variable D3. La valeur 12 de D3 est enregistrée à l'adresse mémoire 0001f47a.
De même, pour  D2, étant codée sur 2 octets, cela explique que l'adresse de D3 soit celle de D2 + 2.

***3.2.3***

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c es.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c ecrmem.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -o ecrmem ecrmem.o es.o
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-run ecrmem
00000000
0
fffffff6
-10
0000
0
fff6
-10
00
0
f6
-10
```

```
        .data
DW: .word     0
DH: .hword    0
DB: .byte     0

        .text
        .global main
main:
        @ word
        @ word @ affiche avant
        LDR r0, LD_DW
        LDR r1, [r0]
        BL EcrHexa32
        BL EcrZdecimal32

        @ word @ range -10
        MOV r4, #-10
        LDR r5, LD_DW
        STR r4, [r5]

        @ word @ affiche apres
        LDR r0, LD_DW
        LDR r1, [r0]
        BL EcrHexa32
        BL EcrZdecimal32

        @ hword
        @ hword @ affiche avant
        LDR r0, LD_DH
        LDR r1, [r0]
        BL EcrHexa16
        BL EcrZdecimal16

        @ hword @ range -10
        MOV r4, #-10
        LDR r5, LD_DH
        STRH r4, [r5]

        @ hword @ affiche apres
        LDR r0, LD_DH
        LDR r1, [r0]
        BL EcrHexa16
        BL EcrZdecimal16

        @ byte
        @ byte @ affiche avant
        LDR r0, LD_DB
        LDR r1, [r0]
        BL EcrHexa8
        BL EcrZdecimal8

        @ byte @ range -10
        MOV r4, #-10
        LDR r5, LD_DB
        STRB r4, [r5]

        @ byte @ affiche apres
        LDR r0, LD_DB
        LDR r1, [r0]
        BL EcrHexa8
        BL EcrZdecimal8

fin:  B exit

LD_DW:  .word     DW
LD_DH:  .word     DH
LD_DB:  .word     DB
```

Les quatre premières lignes générées par la version modifiée du programme "ecrmem" demeurent inchangées.
Les huit lignes d'après résultent de l'exécution répétée du programme avec les valeurs d'adresse DH sur 16 bits et DB sur 8 bits.
Ces deux valeurs, tout comme DW, sont initialement affichées au début du programme avec des valeurs nulles, puis à la fin du programme, elles sont modifiées à -10. Les affichages pour ces variables sont également présentés en hexadécimal suivi de décimal.

L'adaptation du programme d'un mot de 32 bits à des mots de 16 et 8 bits a nécessité des modifications dans les instructions de stockage (STR).
Ainsi, elles deviennent STRH pour un mot de 16 bits, STRB pour un mot de 8 bits.
Cependant, le chargement d'une adresse dans un registre est effectué avec la

commande LDR, car quelle que soit la taille du mot stocké, une adresse demeure sur 32 bits. Les fonctions d'affichage des valeurs diffèrent également en fonction de la taille du mot à afficher.

## ***4.1***

```
        .data
cc:
        .byte 0x42
        .byte 0x4f
        .byte 0x4e
        .byte 0x4a
        .byte 0x4f
        .byte 0x55
        .byte 0x52
        .byte 0x00
        .word 12
        .word 0x11223344
au:
        .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_cc
        add r1, #16
        bl EcrChaine

        ldr r1, LD_au
        bl EcrChaine

@ modification de la chaine d'adresse cc
        ldr r2, LD_cc
        ldr r3, [r2]
        add r3, #32
        strb r3, [r2]
```

```
ldr r2, LD_cc
add r2, #1
ldr r3, [r2]
add r3, #32
strb r3, [r2]

ldr r2, LD_cc
add r2, #2
ldr r3, [r2]
add r3, #32
strb r3, [r2]

ldr r2, LD_cc
add r2, #3
ldr r3, [r2]
add r3, #32
strb r3, [r2]

ldr r2, LD_cc
add r2, #4
ldr r3, [r2]
add r3, #32
strb r3, [r2]

ldr r2, LD_cc
add r2, #5
ldr r3, [r2]
add r3, #32
strb r3, [r2]

ldr r2, LD_cc
add r2, #6
ldr r3, [r2]
add r3, #32
strb r3, [r2]
```

```
        ldr r1, LD_cc
        bl EcrChaine

fin:    B exit

LD_cc:  .word   cc
LD_au:  .word   au
```

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c es.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c caracteres.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -o caracteres caracteres.o es.o
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-run caracteres
BONJOUR
au revoir...
au revoir...
bonjour
```

L'adresse cc pointe vers la chaîne “BONJOUR”, définie en 8 octets, puis vers deux mots de quatre octets. Enfin, la chaîne de caractères "au revoir..." est également déclarée. La distance entre l'adresse cc et celle de la chaîne "au revoir…" se compose de 8 octets pour les huits caractères d'un octet et 8 octets pour les deux mots de quatre octets, soit un total de 16 octets. En incrémentant LD_cc de 16, on obtient l'adresse de la chaîne "au revoir...",

permettant ainsi son affichage. On peut aussi utiliser l'étiquette LD_au pour repérer la chaîne "au revoir…".

Pour obtenir le code ASCII d'une lettre en minuscule à partir de la même lettre en majuscule, il suffit d'ajouter 32. Ainsi, à l'aide de répétitions, on applique cette opération à chacune des 7 lettres de «BONJOUR» pour obtenir la version en minuscules, c'est-à-dire «bonjour». Le processus démarre à partir de l'adresse LD_cc, où le code ASCII de la lettre 'B' est stocké. En itérant, on ajoute 32 à la valeur stockée, en augmentant simultanément l'adresse de 1 à chaque étape pour passer à la lettre suivante.

### ***4.2.1***

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c alignements1.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -o alignements1 alignements1.o es.o
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-run alignements1
01
02
04
0402010d
08
```

après la première modif:

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c alignements2.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -o alignements2 alignements2.o es.o
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-run alignements2
01
02
04
0a0b0c0d
08
```

```
    .data
x:
    .byte   0x01
y:
    .byte   0x02
z:
    .byte   0x04
    .balign 4
a:
    .word   0x0a0b0c0d
b:
    .byte   0x08

    .text
    .global main
main:
    ldr r2, LD_x
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_y
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_z
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_a
    ldr r1, [r2]
    bl EcrHexa32

    ldr r2, LD_b
    ldr r1, [r2]
    bl EcrHexa8

fin:    B exit

LD_x:   .word   x
LD_y:   .word   y
LD_z:   .word   z
LD_a:   .word   a
LD_b:   .word   b
```

Les trois premières valeurs affichées correspondent correctement aux contenus des adresses LD_x, LD_y et LD_z. La dernière valeur, provenant de l'adresse LD_b, est également correcte. Cependant, une incohérence est observée avec la quatrième valeur affichée, stockée à l'adresse LD_a. La véritable valeur de la variable 'a' est 0A0B0C0D en hexadécimal, équivalant à 168496141 en décimal, et non à 67240205 comme mentionné précédemment. C'est un problème d'alignement qui peut être résolu en utilisant l'instruction ".balign 4".
(notre programme à côté:)

2ème retouche:

```
    .data
x:
    .byte   0×01
y:
    .byte   0×02
z:
    .byte   0×03
a:
    .hword  0×0a0b

    .text
    .global main
main:
    ldr r2, LD_x
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_y
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_z
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_a
    ldr r1, [r2]
    bl EcrHexa16

fin:    B exit

LD_x:   .word   x
LD_y:   .word   y
LD_z:   .word   z
LD_a:   .word   a
```

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c alignements3.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -o alignements3 alignements3.o es.o
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-run alignements3
01
02
03
010b
```

Les trois données x, y et z sont des octets qui remplissent les ¾ d'un mot de 32 bits. De ce fait, la donnée suivante a, qui contient un demi-mot de 16 bits, se retrouve sur une adresse qui n'est ni multiple de 4, ni multiple de 2. La valeur est alors mal alignée et la valeur affichée n'est pas la bonne.

3ème retouche:

```
    .data
x:
    .byte   0×01
y:
    .byte   0×02
z:
    .byte   0×03
    .balign 2
a:
    .hword  0×0a0b

    .text
    .global main
main:
    ldr r2, LD_x
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_y
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_z
    ldr r1, [r2]
    bl EcrHexa8

    ldr r2, LD_a
    ldr r1, [r2]
    bl EcrHexa16

fin:    B exit

LD_x:   .word   x
LD_y:   .word   y
LD_z:   .word   z
LD_a:   .word   a
```

```
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -c alignements4.s
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-gcc -o alignements4 alignements4.o es.o
rivetma@im2ag-turing-01:[~/inf401/tp34]: arm-eabi-run alignements4
01
02
03
0a0b
```

L'instruction ".balign 2" est alors nécessaire pour aligner une valeur stockée dans un demi-mot de 16 bits. Une fois bien alignée, la valeur est correctement affichée.

***4.2.2***

```
Part of Contents of section .data:
 1f438 00000000 18000000 0a010000 2a000000
```

À l'intérieur de la section de données du fichier "accessmem", trois mots de 32 bits sont explicitement définis : aa, qui détient la valeur décimale 24, xx, qui contient la valeur décimale 266, et bb, qui porte la valeur décimale 42.

Lorsqu'on examine l'adresse 1f43c, on trouve le mot aa, qui affiche la valeur hexadécimale 18 00 00 00 sur 32 bits, correspondant bel et bien à 24 en décimal. Cette correspondance découle de l'utilisation de la convention de stockage par "petits bouts" lors de la compilation, où les bits de poids faibles occupent les adresses les plus basses. Suivant une logique similaire, le mot xx est situé à l'adresse 1f440 et bb à l'adresse 1f444.

Du fait que ces trois mots occupent chacun 4 octets, la différence entre les adresses de aa et xx, ainsi que celles de xx et bb, est effectivement de 4.

```
Part of Contents of section .data:
 1f470 00000000 0a010000 2a000c00
```

Dans la section de données du fichier "accessmem2", trois mots sont explicitement définis. Le premier, nommé D1, occupe 4 octets et renferme la valeur décimale 266. Le second, appelé D2, est un mot de 2 octets contenant la valeur décimale 42. Enfin, le troisième, nommé D3, est un mot d'1 octet contenant la valeur décimale 12., ces trois mots sont alignés sur la ligne débutant à l'adresse 1f470.

D1 est repéré à l'adresse 1f474, exhibant la valeur hexadécimale 0a 01 00 00, équivalant à 266 en décimal, conformément à la convention de stockage par "petits bouts" adoptée lors de la compilation. Selon cette convention, les bits de poids faibles occupent les adresses les plus basses. De manière similaire, on note que D2 est à l'adresse 1f478 et D3 à l'adresse 1f47a.

Du fait que D1 a été déclaré sur 4 octets, la différence entre son adresse et celle de la variable suivante, D2, est bien de 4. De même, D2, déclaré sur 2 octets, présente une différence d'adresse de 2 avec D3.