Architecture des Ordinateurs (partiel)

March 23, 2024

Eléments de correction du partiel d'INF401, Mars 2024.
Sujet (cf. Moodle) ; Durée : 1h30 ; 1 doc A4 R/V personnel, manuscrit, autorisé.

1 Numération, opération en base 2, complément à 2

1.1 Question a

[43]:
```
!./show 16 5847
```

```
      Bit numbers
      1111 11
      5432 1098 7654 3210                            if natural     if signed

:     0001 0110 1101 0111        :  0x16d7 -->       5847 or       +5847
```

[44]:
```
!./show 16 -47
```

```
      Bit numbers
      1111 11
      5432 1098 7654 3210                            if natural     if signed

:     1111 1111 1101 0001        :  0xffd1 -->      65489 or         -47
```

1.2 Question b

rem. : il s'agit de nombres relatifs, il faut donc regarder la colonne "signed"

[45]:
```
!./show 16 0x1024
```

```
      Bit numbers
      1111 11
      5432 1098 7654 3210                            if natural     if signed

:     0001 0000 0010 0100        :  0x1024 -->       4132 or       +4132
```

[46]:
```
!./show 16 0xFACE
```

```
      Bit numbers
      1111 11
```

```
      5432 1098 7654 3210                              if natural     if signed

:     1111 1010 1100 1110        :  0xface -->      64206 or       -1330
```

La réponse est donc -1330 (nombre signé).

1.3 Question c

[9]:
```
!./add 8 0b01011001 0b01001011
```

```
         Bit numbers
         7654 3210                              if natural     if signed

         0101 1001 left    :  0x59 -->          89 or          +89
       + 0100 1011 right   :  0x4b -->          75 or          +75
   C=0 != 1011 0110 < c0=0   (in carries)
   V=1  ^ ---- ----
 Z=0 N=1->1010 0100  =     :  0xa4 -->         164 or          -92
```

L'opération est donc ;

- pour les naturels : 89+75=164 (correct), et
- pour les relatifs : 89+75=-92 (incorrect).

Les indicateurs sont donc :

- résultat non nul, Z=0
- bit de signe, N = 1
- bit de retenu, C = 0
- oVerflow, V = 1

2 Tours de Hanoï

Rappel du code fourni (quasi vide) :

[10]:
```
%%writefile arm.s
  .data
  @ déclaration des chaînes de caractères
  @ déclaration des tableaux initialisés
  .text
  .global main
main: push {lr}
  @ votre programme
  pop {lr}
  bx lr
```

```
Overwriting arm.s
```

On peut essayer de le compiler et de l'exécuter :

[11]:
```
!arm-linux-gnueabi-gcc -static -c es.s
!arm-linux-gnueabi-gcc -static -c arm.s
!arm-linux-gnueabi-gcc -static arm.o es.o -o arm.e
!qemu-arm  arm.e
!echo "(fin)"
```

```
(fin)
```

Le code compile, peut être exécuté, mais ne fait pas grand chose.

2.1 Question d

Déclaration d'une chaîne de caractères

[12]:
```
%%writefile arm.s
  .data
fleche: .asciz "->"                    @ Question d)
  .balign 4                            @ Question d)
  @ déclaration des tableaux initialisés
  .text
  .global main
main: push {lr}
  @ votre programme
  pop {lr}
  bx lr
LD_fleche: .word fleche                 @ Question d)
```

```
Overwriting arm.s
```

2.2 Question e

Déclaration des tableaux (seul le premier tableau était demandé dans le partiel).

[13]:
```
%%writefile arm.s
  .data
fleche: .asciz "->"
  .balign 4
tours: .word 31                        @ Question e)
   .word 0                             @ Question e)
   .word 0                             @ Question e)
avant: .word 2
   .word 0
   .word 1
apres: .word 1
   .word 2
   .word 0
  @ déclaration des tableaux initialisés
  .text
  .global main
```

```
main: push {lr}
  @ votre programme
  pop {lr}
  bx lr
LD_fleche: .word fleche
LD_tours: .word tours                    @ Question e)
LD_avant: .word avant
LD_apres: .word apres
```

```
Overwriting arm.s
```

[14]:
```
!arm-linux-gnueabi-gcc -static -c es.s
!arm-linux-gnueabi-gcc -static -c arm.s
!arm-linux-gnueabi-gcc -static arm.o es.o -o arm.e
!qemu-arm  arm.e
!echo "(fin)"
```

```
(fin)
```

Le code compile, peut être exécuté, mais ne fait toujours pas grand chose.

2.3 Question f

Affichage

rem. dans l'énoncé, c'est EcrZdecimal32 et EcrChaine qui sont indiqués (comme utilisés en Tp), mais pour avoir des affichages plus compacts, ici sont utilisés EcrNdecim32 et EcrChn qui ne passent pas à la ligne automatiquement

[27]:
```
%%writefile QuestionFLigne5.se
L5: mov R1, R6          @ Qestion F
  bl EcrNdecim32        @ Qestion F
  ldr R1, LD_fleche     @ Qestion F
  bl EcrChn             @ Qestion F
  mov R1, R7            @ Qestion F
  bl EcrNdecim32        @ Qestion F
  bl ALaLigne           @ Qestion F
```

```
Overwriting QuestionFLigne5.se
```

Pour vérifier que ce code est correct, il faut ajouter quelques lignes pour fixer arbitrairement R6 et R7 (juste pour vérifier, ce code ne sera pas pris en compte dans la suite) :

[28]:
```
%%writefile armQF.es
  .data
fleche: .asciz "->"
  .text
  .global main
main: push {lr}
  mov R6, #42
```

```
  mov R7, #1024
#include "QuestionFLigne5.se"
pop {lr}
  bx lr
LD_fleche: .word fleche
```

```
Overwriting armQF.es
```

[29]:
```
!arm-linux-gnueabi-cpp armQF.es -o armQF.s
!arm-linux-gnueabi-gcc -static -c es.s
!arm-linux-gnueabi-gcc -static -c armQF.s
!arm-linux-gnueabi-gcc -static armQF.o es.o -o armQF.e
!qemu-arm  armQF.e
!echo "(fin)"
```

```
42->1024
(fin)
```

L'affichage est bien celui attendu.

2.4 Question g

Affectation (tableau)

rem : pour avoir le code entier à la fin, les 2 affectations sont données.

[12]:
```
%%writefile QuestionGLigne10.se
L10:   @ tours[max]=tours[max]-top;  @ Qestion G
  ldr R0, LD_tours                   @ Qestion G
  ldr R1, [R0, R6, LSL #2]           @ Qestion G
  sub R1, R1, R8                     @ Qestion G
  str R1, [R0, R6, LSL #2]           @ Qestion G
      @ tours[min]=tours[min]+top;
  ldr R1, [R0, R7, LSL #2]
  add R1, R1, R8
  str R1, [R0, R7, LSL #2]
```

```
Overwriting QuestionGLigne10.se
```

Vérifier que ce code est correct en l'exécutant n'est pas immédiat (il faudrait ajouter des initialisations *(plus)* raisonnables pour R6, R7 et R8 et des affichages des tableaux : cela nécessite autant/plus de code que l'affectation ! il ne reste plus que 2 questions, continuons.)

2.5 Question h

Boucle

[13]:
```
%%writefile QuestionHLigne789.se
L7:     @ tant que (tours[max] | top) != tours[max] faire top=top>>1  fin tant␣
  ↪que
```

```
  ldr R0, LD_tours
  ldr R1, [R0, R6, LSL #2]
  orr R2, R1, R8
  cmp R1, R2
  beq L9
L8:
  mov R8, R8, ASR #1
  b L7
L9: nop
```

```
Overwriting QuestionHLigne789.se
```

2.6 Question i

Conditionnelle

[14]:
```
%%writefile QuestionILigne234.se
L2: @si tours[avant[tr]] < tours[apres[tr]] alors
    @  max=apres[tr]; min=avant[tr];
    @sinon max=avant[tr]; min=apres[tr]; fin si
  ldr R0, LD_tours
  ldr R1, LD_avant
  ldr R1, [R1, R5, LSL #2]
  ldr R2, [R0, R1, LSL #2]
  ldr R3, LD_apres
  ldr R3, [R3, R5, LSL #2]
  ldr R4, [R0, R3, LSL #2]
  cmp R2, R4
  BGE sinon
alors:
  mov R6, R3
  mov R7, R1
  b fsi
sinon:
  mov R6, R1
  mov R7, R3
fsi: nop
```

```
Overwriting QuestionILigne234.se
```

2.7 Vérification finale

Pour vérifier l'ensemble, on complète le code de la question e avec des inclusions de ce qui a été demandé après et avec ce qui n'a pas été demandé :

[31]:
```
%%writefile arm.se
  .data
fleche: .asciz "->"
```

```
  .balign 4
  @ déclaration des tableaux initialisés
tours: .word 31
   .word 0
   .word 0
avant: .word 2
   .word 0
   .word 1
apres: .word 1
   .word 2
   .word 0
  .text
  .global main
main: push {lr}
L0:  mov R5, #1
L1:  @ tant que tours[2] != 31 faire
  ldr R0, LD_tours
  ldr R0, [R0, #8]
  cmp R0, #31
  beq fin
@ votre programme
#include "QuestionILigne234.se"
#include "QuestionFLigne5.se"
L6:  mov R8, #16
#include "QuestionHLigne789.se"
#include "QuestionGLigne10.se"
L11: @ tr = tours[tr]
  ldr R0, LD_apres
  ldr R5, [R0, R5, LSL #2]
  b L1
fin:
  pop {lr}
  bx lr
LD_fleche: .word fleche
LD_tours: .word tours
LD_avant: .word avant
LD_apres: .word apres
```

```
Overwriting arm.se
```

[32]:
```
!arm-linux-gnueabi-cpp arm.se -o arm.s
!arm-linux-gnueabi-gcc -static -c es.s
!arm-linux-gnueabi-gcc -static -c arm.s
!arm-linux-gnueabi-gcc -static arm.o es.o -o arm.e
!qemu-arm  arm.e
!echo "(fin)"
```

```
0->2
```

```
0->1
2->1
0->2
1->0
1->2
0->2
0->1
2->1
2->0
1->0
2->1
0->2
0->1
2->1
0->2
1->0
1->2
0->2
1->0
2->1
2->0
1->0
1->2
0->2
0->1
2->1
0->2
1->0
1->2
0->2
(fin)
```

Le code compile, s'exécute sans erreur. Pour vérifier qu'il est correct, il faut maintenant prendre 5 disques de tailles différentes et 3 bâtons…

```
         |              |          |
         =              |          |
        ===             |          |
       =====            |          |
      =======           |          |
     =========          |          |
##########################################
```

3 Codage Quoted-Printable

Pour avoir un fichier avec tous les octets (fInit, cf. sujet), on peut imaginer le code suivant (hors sujet) :

[33]:
```
%%writefile prog.c
#include <stdio.h>

int main() {
unsigned char c=0;
printf("%c",c);
for(c=1;c;c++) {
  printf("%c",c);}
return 0;}
```

```
Writing prog.c
```

[38]:
```
!gcc prog.c -o prog.e
!./prog.e > fInit
```

3.1 Question j

Affichage sous hexdump (ou équivalent) :

[41]:
```
!hexdump -C fInit
```

```
00000000  00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f  |…|
00000010  10 11 12 13 14 15 16 17  18 19 1a 1b 1c 1d 1e 1f  |…|
00000020  20 21 22 23 24 25 26 27  28 29 2a 2b 2c 2d 2e 2f  | !"#$%&'()*+,-./|
00000030  30 31 32 33 34 35 36 37  38 39 3a 3b 3c 3d 3e 3f  |0123456789:;<=>?|
00000040  40 41 42 43 44 45 46 47  48 49 4a 4b 4c 4d 4e 4f  |@ABCDEFGHIJKLMNO|
00000050  50 51 52 53 54 55 56 57  58 59 5a 5b 5c 5d 5e 5f  |PQRSTUVWXYZ[\]^_|
00000060  60 61 62 63 64 65 66 67  68 69 6a 6b 6c 6d 6e 6f  |`abcdefghijklmno|
00000070  70 71 72 73 74 75 76 77  78 79 7a 7b 7c 7d 7e 7f  |pqrstuvwxyz{|}~.|
00000080  80 81 82 83 84 85 86 87  88 89 8a 8b 8c 8d 8e 8f  |…|
00000090  90 91 92 93 94 95 96 97  98 99 9a 9b 9c 9d 9e 9f  |…|
000000a0  a0 a1 a2 a3 a4 a5 a6 a7  a8 a9 aa ab ac ad ae af  |…|
000000b0  b0 b1 b2 b3 b4 b5 b6 b7  b8 b9 ba bb bc bd be bf  |…|
000000c0  c0 c1 c2 c3 c4 c5 c6 c7  c8 c9 ca cb cc cd ce cf  |…|
000000d0  d0 d1 d2 d3 d4 d5 d6 d7  d8 d9 da db dc dd de df  |…|
000000e0  e0 e1 e2 e3 e4 e5 e6 e7  e8 e9 ea eb ec ed ee ef  |…|
000000f0  f0 f1 f2 f3 f4 f5 f6 f7  f8 f9 fa fb fc fd fe ff  |…|
00000100
```

On retrouve bien au milieu tous les octets, de 00 à FF, avec les caractères ascii en dernière colonne à partir de l'octet 20 (et jusqu'à 7F). En première colonne, l'indication d'adresse dans le fichier (de 00 à F0 pour les premiers octets de chaque ligne, le dernier octet de chaque ligne a une adresse allant de 0F à FF, rem. : l'adresse et la valeur de l'octet sont identiques, c'est normal)

3.2 Question k

Il y a 256 octets (=2^8), donc le fichier a une taille (en octet) de 256 octets.

On peut le vérifier :

[40]:
```
!ls -l fInit
```

```
-rwxrwxrwx 1 bouhino bouhino 256 Mar 22 16:39 fInit
```

3.3 Question l

Les premiers octets sont 00, 01, 02, … Ce sont des octets non affichables, ils seront donc codés avec un égal (=) : `=00=01=02`, cependant, parmi les premiers octets les octets 10 et 13 sont un peu particuliers, ce sont des sauts de ligne (cf. sujet). La première ligne s'arretera donc avec l'octet 10 (et la seconde ligne avec l'octet 13). Ces octets (10 et 13) seront laissés intacts comme dit dans le sujet (et provoqueront donc un saut de ligne).

Le **début du fichier** sera donc :

```
=00=01=02=03=04=05=06=07=08=09
=0B=0C
=0E=0F=10=11=12=13...
```

Et si on voyait comment est fait le saut de ligne après `=09` (on pourrait le voir, par exemple, avec hexdump) on verrait l'octet 10.

Pour la fin du fichier, ce sont aussi des octets non affichables (FD, FE, FF), ils seront donc codés avec un égal (=).

La **fin du fichier** sera donc :

```
...=FD=FE=FF
```

Pour information (hors sujet), le fichier complet est :

```
=00=01=02=03=04=05=06=07=08=09
=0B=0C
=0E=0F=10=11=12=13=14=15=16=17=18=19=1A=1B=1C=1D=1E=1F !"#$%&'()*+,-./01234=
56789:;<=3D>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}=
~=7F=80=81=82=83=84=85=86=87=88=89=8A=8B=8C=8D=8E=8F=90=91=92=93=94=95=96=
=97=98=99=9A=9B=9C=9D=9E=9F=A0=A1=A2=A3=A4=A5=A6=A7=A8=A9=AA=AB=AC=AD=AE=AF=
=B0=B1=B2=B3=B4=B5=B6=B7=B8=B9=BA=BB=BC=BD=BE=BF=C0=C1=C2=C3=C4=C5=C6=C7=C8=
=C9=CA=CB=CC=CD=CE=CF=D0=D1=D2=D3=D4=D5=D6=D7=D8=D9=DA=DB=DC=DD=DE=DF=E0=E1=
=E2=E3=E4=E5=E6=E7=E8=E9=EA=EB=EC=ED=EE=EF=F0=F1=F2=F3=F4=F5=F6=F7=F8=F9=FA=
=FB=FC=FD=FE=FF
```

A noter :

- le saut de ligne en fin de première ligne est causé par un octet 10 (LF)
- le saut de ligne en fin de seconde ligne est causé par un octet 13 (CR)
- les sauts de ligne suivants sont causés par la suite de 2 octet 10, 13 (cf. sujet)
- il n'y a pas de saut de ligne à la fin de la dernière ligne (c'est une fin de fichier)

3.4 Question m

La réponse n'est pas immédiate, mais globalement :

- il y aura à peu près 32+128 caractères non affichable : ~160*3 caractères =XY,
- les autres caractères affichables directement, soit, à peu près : 256-32-128= 96

- ce qui donne environ 580 caractères, à répartir dans des lignes de 76 caractères, ce qui nécessite quelques caractères de plus (`= CR LF`)

Dans le détail :

- les deux premières lignes du fichier sont particulières, elles comprennent
  - les octets 0-8 : 10 octets initiaux représentés par 3*10 octets encodés
  - l'octet/caractère 10 de passage à la ligne (cr) : 1 octet encodé
  - les octets 11-12 : 9 octets initiaux représentés par 3*2 octets encodés
  - le caractère 13 de passage à la ligne (lf) : 1 octet encodé
- ensuite, on va avoir les autres caractères spéciaux (il en reste 18 + 1 (=) + 1 (del) + 128 (>127)), soit 148 caractères spéciaux qui vont demander 3*148 octets encodés
- et les caractères normaux 94 (128 -1 (=) - 1 (del) - 32)
- ces 2 derniers blocs vont nécessiter (94 + 3*148)/75 = 7 passage à la ligne (= CR LF), donc 3*7 octets

d'où 3*10 + 1 + 3*2 + 1 + 3*148 + 94 + 3*7 = 30 + 1 + 6 + 1 + 444 + 94 + 21 = 597 octets

3.5 Question n

Un fichier de lettres "normales" avec des passages à la ligne avant la limite (76), sera transmis tel que, donc avec 256 octets. C'est la taille minimum d'un fichier encodé pour un fichier initial de 256 octets.

A contrario, un fichier de caractères spéciaux nécessitera 256*3 octets et les passages à la lignes ((256*3)/75)*3, soit 768 + 30 = 798 octets.
C'est la taille maximum d'un fichier encodé pour un fichier initial de 256 octets.

Exemple avec un fichier ayant 256 caractères spéciaux (=), sous forme encodée cela donne :

```
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=3D=
=3D=3D=3D=3D=3D=3D
```

3.6 Question o

Le fichier initial comporte 256 octets qui doivent tous être traduits et vont occuper une certaine place.
Pour presque tous les caractères, cette traduction est imposée (ainsi que la place que cela va prendre).
Cependant pour le caractère de tabulation et celui de l'espace, selon qu'ils se trouvent en fin de ligne ou avant la fin de ligne, la traduction demande le égal (=) ou pas.
C'est un premier point qui peut changer la taille du fichier encodé.

Le fichier initial (fInit) comporte également 2 passages à la ligne initiaux qui peuvent être utilisés pour éviter les passages à la ligne forcés (= cr lf).
C'est une seconde marge pour avoir des fichiers plus longs ou plus courts.
En mettant ces 2 passages à la ligne ensemble, en début de fichier, les 2 passages à la ligne ne sont pas bien utilisés pour éviter les lignes longues, ils n'éviteront aucune ligne longue. Si ces 2 passages à la lignes sont mis un peu avant des limites de 76 caractères, cela permettra d'éviter 2 passages à la ligne forcée.

Globalement, il faut un peu moins de 8 lignes de caractères pour fEncoded.

Donc, dans le meilleur cas :

- avec les caractères tabulation et espace loin des fins de ligne,
- et les 2 sauts de lignes (cr et lf) du fichier initial placé un peu avant la limite de 76 caractères
- on obtient un économie de 2 octet (pour la tabulation) et 2*3 octets d'économie pour les sauts de lignes sur la taille du fichier fEncoded, soit une taille min de 589 octets.

Par ex. :

```
=00=01=02=03=04=05=06=07=08  =0B=0C=FA=FB=FC=FD=FE=FF
=0E=0F=10=11=12=13=14=15=16=17=18=19=1A=1B=1C=1D=1E=1F !"#$%&'()*+,-./01234
56789:;<=3D>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}=
~=7F=80=81=82=83=84=85=86=87=88=89=8A=8B=8C=8D=8E=8F=90=91=92=93=94=95=96=
=97=98=99=9A=9B=9C=9D=9E=9F=A0=A1=A2=A3=A4=A5=A6=A7=A8=A9=AA=AB=AC=AD=AE=AF=
=B0=B1=B2=B3=B4=B5=B6=B7=B8=B9=BA=BB=BC=BD=BE=BF=C0=C1=C2=C3=C4=C5=C6=C7=C8=
=C9=CA=CB=CC=CD=CE=CF=D0=D1=D2=D3=D4=D5=D6=D7=D8=D9=DA=DB=DC=DD=DE=DF=E0=E1=
=E2=E3=E4=E5=E6=E7=E8=E9=EA=EB=EC=ED=EE=EF=F0=F1=F2=F3=F4=F5=F6=F7=F8=F9
```

Dans le pire des cas, on perd encore 2 octets (par rapport à fEncoded) en codant l'espace en fin de ligne, soit 599 octets.

4 Commentaires

Des maladresses (pas vraiment des erreurs, mais…) :

- Question g (affectation) : pour le calcul des adresses, faire explicitement une multiplication par 4 (ou des additions multiples) n'est pas la meilleure solution, ce qui est attendu, c'est un décalage de 2 bits ; sous forme compact, cela donne une instruction complexe (c'est vrai) mais efficace `ldr R1, [R0, R6, LSL #2]`
- Exercice 2 (arm) : le sujet prévoyait que les variables Top, Max, Min, TR soient implantées dans les registres R5-R8 et non pas en mémoire, cela simplifiait le code et correspondait à la situation habituelle lors des TD 5-6 sur les structures de contrôle (pas d'accès à la mémoire) ; les accès à la mémoire (td 3-4) étaient vus à travers de l'accès aux tableaux (et à la chaîne de caractères)

Des points perdus par négligence :

- Question a (binaire) : lectures des bits dans le mauvais ordre (le premier bit obtenu, reste de la première division par 2 est le bit de parité, c'est le bit de poids faible, celui que l'on écrit en dernier)
- Question b (binaire) : quand le nombre est négatif, ne pas oublier de mettre un "-" à la fin

- Question d et e (arm, données): les déclarations en mémoire (.data) ne peuvent pas exister sans des déclarations en zone .text, il faut les indiquer (c'était demandé dans le sujet)
- Question f (affichage) : l'affichage des entiers demande la valeur dans R1, l'affichage des chaînes de caractère demande l'adresse de la chaîne de caractère dans R1, ne pas confondre, bien distinguer les 2 cas (une lecture mémoire en plus pour les entiers)
- Question h (boucle) : les boucles simples comportent une condition (simple) réduite à un `CMP Rx, Ry`sur lequel boucler, mais ici, la condition était plus complexe et nécessitait des calculs préalables au `CMP ...`, l'étiquette de début de boucle devait se situer au début de ces calculs (avant le `CMP ...`) pour, à chaque tour de boucle, effectuer l'ensemble de l'évaluation de la condition
- Question j (hexdump) : c'est une question proche du travail en tp, les réponses vraiment bonnes et complètes ont été rares (les tp sont importants, ils faut les faire avec attention, en comprenant bien ce qui se passe)
- Dans les exercices 2 et 3, les premières questions étaient de complexité progressive (facile au début) ; une bonne gestion de votre temps devait pouvoir accorder un peu de temps à ces premières questions (en tout , 5.5 points). Trop de copies ne comportaient pas de début de réponse pour les premières questions de l'exercice 3 (Questions j, k, l).

Des points non compris :

- Question c (addition) : l'addition est la même pour les naturels et les relatifs, les opérandes et le résultat ne changent pas, les bits Z, N, C et V non plus. C'est l'interprétation des opérandes de cette addition et du résultat qui nécessite de savoir si c'est pour des entiers naturels ou relatifs. La correction (=justesse) de l'interprétation de l'addition pour les naturels correspond à C, la correction de l'interprétation de l'addition pour les relatifs correspond à V.
- Question e (déclaration) : .skip réserve de la place, sans l'initialiser ; .word réserve de la place et permet de l'initialiser (ex: .word 42), ne pas faire les 2 en même temps (une solution plus rare consistait à initialiser le tableau par programme, c'était possible, mais il y avait plus simple, cf. corrigé idem au début du tp 3-4)
- Question h (boucle) : la condition de boucle comportait un "ou", certes, mais ce n'était pas une raison pour faire une traduction de "ou alors" ; ici, ce n'était pas adéquate. Est-ce l'effet de l'un des derniers cours sur la traduction des structures de contrôle (conditions complexes : et-puis (and-then), ou-alors (or-else))
- Question k (taille) : il n'y avait pas de difficulté, ici (réponse 256 octets !), mais attention à ne pas vouloir répondre avec les données du sujet (dans le sujet il était question de 76 lignes, et alors ? c'est effectivement une donnée du problème, mais qui n'intervient pas ici dans la question (vraiment pas), pourquoi apparaît-elle dans les réponses, c'est l'age du capitaine ?)

Des bons points :

- Exercice 1, en général, **bien** résolu (j'espère **rapidement** résolu aussi)
- Question g (affectation) : le double accès au tableau (tab[tab[…]]), c'est bien passé (les erreurs étaient souvent ailleurs)