UNIVERSITÉ GRENOBLE ALPES INF401

Licence de Sciences et Technologies - Parcours INM, MIN, MIN-Int Année 2024–2025

# Contrôle Continu UE INF401 :
## Introduction aux Architectures Logicielles et Matérielles

**13 Mars 2025**

Durée = 1h30

Documents autorisés : 1 A4 R/V personnel manuscrit autorisé ; calculettes et téléphones portables interdits.

**Remarques:** La plupart des questions sont indépendantes, si vous avez du mal avec l'une, passez à la suivante — faites (bien) attention à votre gestion du temps. Tant que possible, indiquez bien tous les détails et justifiez vos réponses. Le barème est donné à titre indicatif.

## Exercice 1 :
## Questions de cours : architecture des ordinateurs

*3 points*

Répondez (en une ou deux phrases) à chacune des questions suivantes.

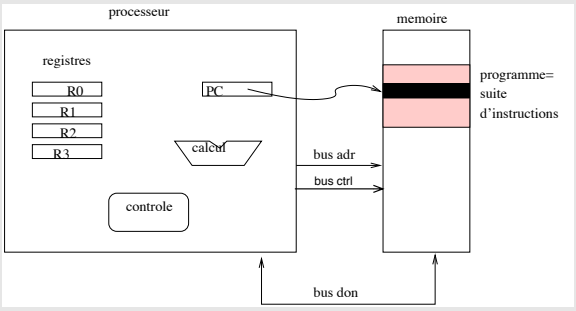
### Question 1.a */1*

Rappelez la caractéristique principale du modèle Von Neumann vu en cours.

Dessinez (rapidement) un schéma du modèle Von Neumann, tel que vu en cours.

Dans le modèle Von Neumann, la mémoire est utilisée pour stocker à la fois les données et les programmes (0.5pts)

Un schéma possible (0.5pts) :




Élements essentiels : une seule mémoire programme et données, une communication entre le proc et la mémoire, les registres PC et généraux.

### Question 1.b */1*

Rappelez la différence qu'il existe entre mémoire volatile et mémoire non-volatile.

La mémoire centrale manipulée lors de la programmation en assembleur `ARM` (par exemple, les zones `.text` et `.data`) est-elle volatile ou non-volatile ?

La mémoire non-volatile est persistente, c'est à dire qu'elle résiste à la mise hors-tension du circuit. La mémoire volatile est quant à elle effacée lors de la mise hors-tension, il faut la sauvegarder sur un support pérenne (mémoire non-volatile) si on veut être capable de la restaurer lors de la prochaine mise sous tension (*e.g.,* mémoire `swap` proposée par les systèmes d'exploitation).

La mémoire manipulée en `ARM` est de la mémoire RAM, volatile.

**Question 1.c** */1*

Rappelez le rôle du compteur de programme (registre `PC`) dans le processeur.

Comment est calculée la nouvelle valeur de `PC` lors d'une exécution séquentielle d'instruction ? Comment est calculée cette valeur lors d'une rupture de séquence ?

Le compteur de programme stocke l'adresse (en mémoire centrale) de la prochaine instruction à exécuter.

Lors d'une exécution séquentielle, `PC <- PC + x`, avec `x` la taille d'une instruction (4 octets en `ARM` 32 bits). Lors d'une rupture de séquence, la nouvelle valeur de `PC` peut être :
- directement spécifiée dans l'instruction (saut absolu)
- calculée à partir du `PC` actuel (saut relatif)

## Exercice 2 : Numération, opération en base 2 et en complément à 2 *5 points*

**Rappel important :** en machine, les entiers relatifs ($\mathbb{Z}$) sont représentés selon la méthode dite du complément à 2 (C2), il ne faut pas confondre cette méthode de représentation avec l'opération dite de complémentation. Pour les entiers naturels ($\mathbb{N}$), la base 2 est utilisée.

**Question 2.a** */2*

Déterminer le nombre de bits minimum pour représenter chacun des nombres suivants puis donner la représentation binaire et hexadécimale de ces entiers relatifs avec le nombre de bits choisi précédemment (codage en complément à 2) : $(+5019)_{10}$, $(-5019)_{10}$

$(+5019)_{10} =$ `01 0011 1001 1011`$_2$ (13 bits de valeur + 1 bit de signe)

$(+5019)_{10} =$ `139B`$_{16}$

$C1($`01 0011 1001 1011`$) =$ `10 1100 0110 0100` (n.b., on rajoute un bit pour le signe)

$C2($`01 0011 1001 1011`$) =$ `10 1100 0110 0101`

Donc $(-5019)_{10} =$ `10 1100 0110 0101`$_2$

$(-5019)_{10} =$ `2C65`$_{16}$

**Question 2.b** */1*

Donner la représentation binaire et la valeur décimale de l'entier relatif `FEED`$_{16}$ (donné sous forme hexadécimal).

```
$ show 16 0xFEED

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

```
1111 1110 1110 1101        :  0xfeed -->      65261 or        -275
```

**Question 2.c** */2*

Effectuer, en posant l'addition comme habituellement et en écrivant toutes les retenues, l'opération binaire suivante sur 1 octet :

$$\begin{array}{rr} & 1011\ 1101 \\ + & 0100\ 1011 \\ \hline = & \end{array}$$

Donner les 2 interprétations usuelles possibles de cette addition selon que les octets sont des entiers naturels ou relatifs. En déduire, la valeur des indicateurs (Z, N, C et V) en expliquant le sens de ces indicateurs.

```
$ add 8 0b10111101 0b01001011

    Bit numbers
      7654 3210                            if natural     if signed

      1011 1101 left   :  0x      bd -->        189 or         -67
    + 0100 1011 right  :  0x      4b -->         75 or         +75
  C=1 == 1111 1110 < c0=0   (in carries)
  V=0  ^ ---- ----
Z=0 N=0->0000 1000  =     :  0x       8 -->          8 or          +8
```

L'opération est donc :
— pour les naturels : 189+75=164 (incorrect), et
— pour les relatifs : -67+75=+8 (correct).
Les indicateurs sont donc :
— résultat non nul, Z=0
— bit de signe, N = 0
— bit de retenu, C = 1
— oVerflow, V = 0

## Exercice 3 : Programmation en langage ARM : détection de palindrôme *8 points*

Le but de cet exercice est d'implémenter un algorithme qui permet de déterminer si une chaîne de caractères est un palindrôme. Pour rappel, un palindrôme est un mot (ou une phrase) qui peut se lire de façon similaire de gauche à droite ou de droite à gauche. Dans cet exercice, on considérera des palindrômes "strictes" : on dira que le mot $A$ est un palindrôme si et seulement si, quelque soit la position $i$ d'un caractère dans $A$, $A_i == A_{size(A)-i}$[1].

L'algorithme à traduire est donc le suivant, pour déterminer si la chaîne `mot` est un palindrôme. On considérera également que la taille (**en nombre de caractères**) de la chaîne `mot` est contenue dans la variable `taille`.

1. De façon générale, on peut considérer qu'une phrase est un palindrôme sans considérer, par exemple, la position des espaces. Mais ici, on se restreint à un cas d'égalité stricte entre la chaîne $A$ et la chaîne $inverse(A)$

```
1  EcrChaine(mot)
2  ALaLigne()
3  pos = 0
4  estPalindrome = "Oui"
5  tantQue (pos < taille / 2)
6      si (mot[pos] != mot[taille - pos - 1]) alors
7          estPalindrome = "Non"
8      finsi
9      pos++
10 finTantQue
11 EcrChaine("Est ce que ")
12 EcrChaine(mot)
13 EcrChaine("est un palindrome ? ")
14 EcrChaine(estPalindrome)
15 ALaLigne()
```

Dans l'algorithme, les variables entières (`pos` et `taille`) sont des entiers naturels sur 1 mot de 4 octets. Elles sont respectivements placées dans les registres `r5` (pour `pos`) et `r6` (pour `taille`). Les chaînes `mot` et `estPalindrome` sont des chaînes de caractères, placées en mémoire. Plus particulièrement, on considère que :

- l'adresse de la chaîne `mot` est placée dans le registre `r7` au début du programme (l'allocation de la mémoire est déjà réalisé) ;
- l'adresse de la chaîne `estPalindrome` sera placée dans le registre `r8` (ce sera à vous de réaliser l'allocation et l'affectation à `r8` au cours des questions suivantes).

On rappelle que chaque caractère est codé sur un octet. Les affichages sont réalisés à l'aide des fonctions `EcrChaine` et `ALaLigne` correspondant aux fonctions du fichier `es.s` étudiées en cours et en TP, dont le fonctionnement est rappelé en annexe.

À vous maintenant de traduire le programme en langage d'assemblage `ARM`, en vous basant sur le squelette de code donné ci-dessous, ainsi que sur les questions qui suivent.

```
        .data
        @ à compléter : déclaration des chaînes de caractères

        .text
        .global main
main:   push {lr}
        @ à compléter : votre programme
        pop {lr}
        bx lr

        @ à compléter : déclaration des pointeurs relai
```

**Question 3.a** */1*

Donnez le code `ARM` (avec `balign` si nécessaire) pour déclarer les chaînes de caractères `"Oui"` et `"Non"`, dans la zone de données (`.data`), avec pointeurs relais dans la zone `.text`.

```
        .data
        oui: .asciz "Oui"
        non: .asciz "Non"
        .balign 4
```

```
        .text
        .global main
main:   push {lr}
            @ votre programme
        pop {lr}
        bx lr
LD_oui: .word oui
LD_non: .word non
```

**Question 3.b** */1*

Donnez la taille, en octets, de la chaîne de caractères `"Oui"` ainsi déclarée.

En règle générale, comment calculer la taille d'une chaîne de caractères en mémoire ?

La chaine de caractères `"Oui"` comporte 3 octets (`O`, `u` et `i`), et un octet spécial `0` de terminaison de chaîne de caractères. Soit 4 octets par chaîne.

De façon générale, chaque chaîne a besoin de $n + 1$ octets en mémoire, pour $n$ caractères.

**Question 3.c** */1*

Donnez le code `ARM` pour l'initialisation de la variable `estPalindrome` à `"Oui"` (ligne 4), et pour l'affichage de cette même variable `estPalindrome` (ligne 14).

```
    @ initialisation
    ldr r8, LD_oui

    @affichage
    mov r1, r8
    bl EcrChaine
```

**Question 3.d** */2*

Donnez le code `ARM` pour traduire l'instruction conditionnelle (lignes 6 à 8).

```
    ldrb r1, [r7, r5]   @ r1 <- mot[pos]
    sub r2, r6, r5      @ r2 <- taille - pos
    sub r2, r2, #1      @ r2 <- taille - pos - 1
    ldrb r2, [r7, r2]   @ r2 <- mot[taille - pos - 1]
    cmp r1, r2          @ r2 == r1
    beq finSi
    ldr r8, LD_non      @ estPalindrome <- "Non"
finSi:
```

**Question 3.e** */0.5*

On rappelle qu'effectuer une division par 2 d'un entier naturel représenté en binaire peut se faire simplement à l'aide d'un **décalage logique** d'un bit vers la droite.

Rappelez pourquoi il s'agit ici d'un décalage **logique** et non **arithmétique** vers la droite.

Les variables `pos` et `taille` sont des entiers naturels.

Ainsi, le décalage arithmétique à droite n'a pas de sens (propagation du bit de poids fort, qui ne correspond pas au bit de signe sur $\mathbb{N}$). Il faut bien faire un décalage logique.

**Question 3.f** */0.5*

Donnez l'instruction `ARM` permettant d'effectuer le calcul `r1 <- r6 » 1`, soit `r1 <- taille/2` (le symbole » représentant le **décalage logique à droite**).

Solution avec `MOV` : `MOV r1, r6, LSR #1`

Solution avec `LSR` : `LSR r1, r6, #1`

**Question 3.g** */2*

En se basant sur les réponses des deux questions précédentes, donnez le code `ARM` pour la boucle externe (lignes 5 à 10). Si vous n'avez pas su répondre aux questions précédentes, indiquez "Instructions 3.d" et/ou "Instruction 3.f" dans votre code, pour expliquer où s'insèrent les réponses aux questions **3.d** et **3.f**.

```
tantQue:
    MOV r1, r6, LSR #1  @ r1 <- taille/2 (Instruction 3.f)
    cmp r5, r1          @ pos < taille/2
    bcs finTantQue
    @ Instructions 3.d
    ldrb r1, [r7, r5]   @ r1 <- mot[pos]
    sub r2, r6, r5      @ r2 <- taille - pos
    sub r2, r2, #1      @ r2 <- taille - pos - 1
    ldrb r2, [r7, r2]   @ r2 <- mot[taille - pos - 1]
    cmp r1, r2          @ r2 == r1
    beq finSi
    ldr r8, LD_non      @ estPalindrome <- "Non"
finSi:
    @ fin Instructions 3.d
    add r5, r5, #1      @ pos++
    b tantQue
finTantQue:
```

## Exercice 4 : Représentation des couleurs

*4 points*

Le but de cet exercice est d'étudier le format le plus utilisé de représentation des couleurs en informatique : le format `RVB` (ou *RGB* en anglais). Ce format permet de représenter des couleurs

en décomposant chaque couleur en trois composantes : le rouge, le vert et le bleu. En particulier, on s'intéresse au format `RVB` usuel sur des machines 32 bits, où chaque composante (rouge, vert, bleu) est codée sur 8 bits.

Pour une couleur donnée, chaque composante est donc une valeur sur $[0, 255]$ : plus la valeur est proche de 255 (`0xff`), plus la couleur sera "concentrée" dans cette composante. On dénote donc chaque couleur par 3 valeurs hexadécimales : $0xR_1R_0V_1V_0B_1B_0$. Par exemple, la couleur `0xFF0000` représente un rouge pur (composante `R` au plus fort, composantes `V` et `B` au plus faible), et la couleur `0xFFFF00` représente du jaune (mélange de rouge et de vert). Un encodage équivalent sous forme de triplets de valeurs décimales $(R, V, B)$ (utilisé en question 4.b) sera, pour le rouge, $(255, 0, 0)$, et pour le jaune $(255, 255, 0)$.

Remarque : vous pouvez constater que le format proposé n'utilise pas les 32 bits disponibles. Les bits restants peuvent être utilisés pour coder une information de transparence (nommée *alpha channel*), par exemple dans le format `PNG`. Au final, chaque pixel est bien encodé sur **32 bits**, pour des raisons d'alignement.

**Question 4.a** */1*

Calculez le nombre (approximatif, en ordre de grandeur) de couleurs différentes représentables avec un tel format.

On a trois couleurs sur 8 bits, soit 24 bits de couleur. On peut donc représenter $2^{24} = 16777216$ (environ 16 millions de couleurs).

**Question 4.b** */1.5*

Sachant que le noir correspond à une absence de couleur, donnez le codage `RVB` du noir, du blanc et du magenta (mélange de rouge et de bleu).

Exprimer sous forme de triplet de valeurs décimales $(R, V, B)$ chacune des couleurs suivantes : `0xFF2234` et `0x001525`

— noir : `0x000000` ;
— blanc : `0xFFFFFF` ;
— magenta : `0xFF00FF`

`0xFF2234` $= (255, 34, 52)$

`0x001525` $= (0, 21, 37)$

**Question 4.c** */1*

Supposons une image composée de 1024×512 pixels, encodant chacun une couleur (en utilisant le format `RVB` défini ci-dessus). Donnez la taille (en méga-octets) d'une telle image.

$4 \times 1024 \times 512 = 4 \times 2^{10} \times 2^9 = 4 \times 2^{19} = 4 \times \frac{2^{20}}{2} = 2$ Mo

Si on considère 3 octets (et non 4) par pixel, on obtient 1.5 Mo (0.5 pts /1)

**Question 4.d** */0.5*

Pour l'expérience, nous avons créé une image de 1024×512 pixels, tous jaunes (`0xFFFF00`,

avec le format défini ci-dessus). Cette image a été enregistrée au format `JPG`, puis, à l'aide d'une rapide commande système (`du -h image.jpg`), nous avons mesuré que cette image pèse 4 Ko.

Que pouvez vous en déduire sur le format `JPG` ? Pensez vous que ce soit lié au fait que tous les pixels soient de la même couleur ?

Le format `JPG` inclut une compression, tous les pixels ne sont pas représentés.

En particulier, dans le cas d'une image uniforme, l'information à représenter est très faible : il faut seulement stocker la couleur en question, ainsi que l'information que tous les pixels sont de cette couleur.

# Annexes

## Fonction d'entrée/sortie

Rappel du fonctionnement des deux fonctions du fichier `es.s` utilisées dans l'exercice 2 :
— `bl ALaLigne` provoque un passage à la ligne dans l'affichage ;
— `bl EcrChaine` affiche la châine de caractères dont l'adresse est contenue dans `r1`.

## Principales instructions du processeur ARM

| Code | Nom | Explication du nom | Opération | Remarque |
|---|---|---|---|---|
| 0000 | `AND` | AND | et bit à bit | |
| 0010 | `SUB` | SUBstract | soustraction | |
| 0100 | `ADD` | ADDition | addition | |
| 1000 | `TST` | TeST | et bit à bit | pas `rd` |
| 1010 | `CMP` | CoMPare | soustraction | pas `rd` |
| 1100 | `ORR` | OR | ou bit à bit | |
| 1101 | `MOV` | MOVe | copie | pas `rn` |
| | `Bxx` | Branch | branchement conditionnel | xx = condition |
| | `LDR` | LoaD Register | lecture mémoire | |
| | `STR` | STore Register | écriture mémoire | |

L'opérande source d'une instruction `MOV` peut être une valeur immédiate notée `#5` ou un registre noté `Ri`, `i` désignant le numéro du registre. Il peut aussi être le contenu d'un registre sur lequel on applique un décalage de `k` bits ; on note `Ri, DEC #k`, avec `DEC` $\in$ {`LSL`, `LSR`, `ASR`, `ROR`}.

## Principaux codes conditions du processeur ARM

| cond | mnémonique | signification | condition testée |
|---|---|---|---|
| 0000 | EQ | égal | $Z$ |
| 0001 | NE | non égal | $\overline{Z}$ |
| 0010 | CS/HS | $\geq$ dans $\mathbb{N}$ | $C$ |
| 0011 | CC/LO | $<$ dans $\mathbb{N}$ | $\overline{C}$ |
| 1000 | HI | $>$ dans $\mathbb{N}$ | $C \wedge \overline{Z}$ |
| 1001 | LS | $\leq$ dans $\mathbb{N}$ | $\overline{C} \vee Z$ |