Chapitre 5

Série d'exercices n°5 - Théorie chapitre III

5.1 Avant-propos

Il est d'une importance capitale de remarquer que les problèmes de programmation ont rarement une seule solution, contrairement aux problèmes de quantification ou d'encodage de l'information ou d'encodage abordés dans les sessions d'exercice prédédentes. Dès lors, les solutions proposés dans ces correctifs ne sont que des suggestions, et l'information réellement utile consiste en les commentaires sur les raisonnements qui mènent à la construction de ces solutions. Pour le reste, une fois ces raisonnements acquis et intégrés, il existe en général plusieurs solutions valable à un problème de programmation et seuls la compilation et l'essai en contexte réel seront à même d'en juger la pertinence.

En particulier, les conventions employées lors de la rédaction de pseudo-code sont entièrement discutables et interchangeables. En effet, le pseudo-code n'est pas lié à des contraintes syntaxiques concrètes comme un langage de programmation réel ; il doit être cohérent par rapport à lui-même et être capable d'exprimer la logique de l'algorithme qu'il décrit, mais il n'a pas pour but d'être immédiatement compilable ou interprétable. Dans ces correctifs, on va explicitement employer plusieurs conventions de pseudo-code différentes afin d'illustrer ce principe.

5.2 Rappels mathématiques

— Grand-boutisme VS petit-boutisme : dans une architecture informatique manipulant de l'information, on doit définir si l'on travaille en "grand-" ou "petit-"boutiste. Ceci correspond à définir si l'on place le bit de poids fort respectivement au début ou a la fin d'un ensemble de cellules contenant une information individuelle. En général, les informations transitant sur les réseaux sont organisées de manière grand-boutiste, avec la cellule de poids fort en premier, et les informations dans les mémoires informatiques sont organisées de manière petit-boutiste, avec la cellule de poids faible en premier - mais ce n'est pas toujours le cas.

5.3 Mode d'emploi

- **Identifier l'organisation de la mémoire :**
  On doit déterminer si l'on est dans une architecture petit-boutiste ou grand-boutiste, pour savoir dans quel sens on doit lire des registres de 8 bits composant p.ex. un mot de 32 bits. On doit déterminer quels sont les cellules de mémoires disponibles dans notre architecture, et leur longueur : un mot de 32 bits est-il accessible ? est-ce un ensemble de 4 registres de 8 bits ?

- **Définir l'ensemble des instructions utilisables :**
  Pour chaque langage de programmation, on a a priori un ensemble différent d'instructions. Le Python, le Java et le C ne s'utilisent pas de la même manière, et la syntaxe comme le fonctionnement "interne" des opérations qui le composent ne sont pas les mêmes. Cependant, ces langages sont en général utilisables sur plusieurs architectures, car c'est le compilateur qui traduit p.ex. du code C en de l'assembleur propre à n différentes architectures de calcul. Dans le cadre de ce cours, on se place au niveau du langage assembleur, qui lui est a priori distinct pour chaque architecture de processeur ; on doit donc commencer par identifier quelles sont les instructions utilisables.

- **Identifier verbalement les opérations à programmer :**
  Lorsqu'on doit implémenter par une suite d'instructions une opération donnée en énoncé, il est toujours bon de réfléchir verbalement au problème avant de choisir les instructions à employer. Il faut être certain que l'on comprend parfaitement l'énoncé, et être capable de le découper en autant de petits blocs conceptuels que nécessaires pour l'implémenter facilement et sans aucune ambiguité sur le fonctionnement exact du programme à construire.

- **Connaître les valeurs d'entrée et leur localisation :**
  L'écrasante majorité des problèmes de programmation implique un certain nombre de valeurs d'entrée. En C, on peut parler des arguments d'une fonction. On doit savoir exactement quelle est la nature de ces valeur, et où elles se situent, car on n'a justement pas l'infrastructure de fonctions du C qui nous permet d'employer des arguments immédiatements. Ces valeurs sont-elles dans des registres ? Sont-elles en mémoire ? Comment y accède-t-on ?

- **Garder un inventaire des variables :**
  Il est toujours nécessaire d'avoir conscience de l'entiereté des variables qu'on manipule, et de leur valeur à tout instant dans l'exécution du programme. C'est encore plus vrai quand on programme avec des registres, car on n'a pas toutes les facilités d'un langage plus haut niveau comme le C où on peut librement déclarer des variables avec des noms et des types spécifiques.

- **Maîtriser le flot de contrôle :**
  Comme avec les variables, quand on travaille à plus bas niveau que des langages comme le C, on a en général pas accès à de belles clauses comme les "if(...) { ...}" qui permettent d'identifier et de contrôler visuellement l'ordre d'exécution des lignes d'un programme.

On doit faire avec les opérations élémentaires auxquelles on a accès, qui seront à priori des comparaisons et des sauts conditionnels, et on doit être parfaitement capable de simuler dans sa tête la trajectoire exacte du programme selon ces sauts conditionnels et les variables qu'ils impliquent.

5.4 Correctifs des exercices

5.4.1 Exercice 1

Énoncé : Un processeur d'architecture petit-boutiste accède à une mémoire dont le contenu est le suivant :

0x1237 :	0xC2
0x1236 :	0x20
0x1235 :	0x00
0x1234 :	0x00

1. Quelle est la valeur de l'entier sur 32 bits situé à l'adresse 0x1234 ?

2. Quelle est la valeur du nombre réel situé à cette adresse, en supposant qu'il soit encodé dans le format IEEE 754 en simple précision ?

Solution :

On commence par noter l'information qu'on est en petit-boutiste ; ceci signifie que les mots de plusieurs cellules d'information de long sont encodés tels que leur cellule de poids faible se situe à la première adresse mémoire dans l'ordre numérique croissant. On identifie aussi en observant le dessin des cellules de mémoire à considérer qu'on semble être dans une architecture où chaque cellule comporte 8 bits, i.e. un octet d'information.

(a) Puisqu'on est en petit-boutiste, la première cellule de mémoire d'un mot figurant à une adresse donnée est son élément de poids faible. On nous indique qu'un entier de 32 bits se situe à l'adresse 0x1234. On cherche donc à lire les informations dans $\frac{32}{8} = 4$ cellules de mémoire à partir de l'adresse 0x1234 ; on doit donc lire les cellules aux adresses 0x1234, 0x1235, 0x1236 et 0x1237, et les assembler dans l'ordre inverse pour respecter la convention de représentation utilisée dans les séances d'exercices précédentes, où le bit de poids fort se situe à gauche et le bit de poids faible à droite.

Adresse	0x1237	0x1236	0x1235	0x1234
Valeur	0xC2	0x20	0x00	0x00

On donc la représentation suivante d'un nombre encodé sur 32 bits en hexadécimal, qu'on supposera encodé selon la représentation la plus fréquente en complément à deux à défaut d'une indication contraire explicite :

$$\boxed{\texttt{0xC2200000}} \textit{ (complément à deux)}$$

On peut convertir cette représentation en un encodage binaire en transformant chaque chiffre hexadécimal en sa représentation équivalente sur quatre bits :

C	2	2	0	0	0	0	0
1100	0010	0010	0000	0000	0000	0000	0000

Pour interpréter ce nombre comme une représentation en complément à deux, on procède comme vu aux sessions d'exercices précédentes. Son bit de signe situé à l'extrême gauche étant à 1, il s'agit d'un négatif et il faut donc procéder comme d'habitude au complément de chaque bit, puis à l'ajout de +1 au résultat du complément pour lire la valeur absolue du nombre encodé. On a successivement :

1100 0010 0010 0000 0000 0000 0000 0000	Représentation en complément à deux
0011 1101 1101 1111 1111 1111 1111 1111	On complémente chaque bit
0011 1101 1110 0000 0000 0000 0000 0000	On ajoute +1

On lit alors la valeur absolue du nombre négatif comme $(2^{29} + 2^{28}) + (2^{27} + 2^{26} + 2^{24}) + (2^{23} + 2^{22} + 2^{21})$. Avec son signe, ce nombre est égal à -1.038.090.240.

(b) On procède comme au premier point pour obtenir la représentation binaire, puis on sépare la représentation complète selon les champs du standard IEEE 754 simple précision, comme suit :

Signe (1 bit)	Exposant (8 bits)	Mantisse (23 bits)
1	10000100	010...0

Ensuite, on l'interprète selon les conventions du standard IEEE 754 comme suit :

— Le bit de signe, à l'extrémité gauche du format, est à 1. On a donc un nombre négatif.

— L'exposant, sur les 8 bits suivants du format, peut être lu comme $e = 2^7 + 2^2 = 132$. La puissance à laquelle est élevé le nombre vaut donc $n = e - 127 = 132 - 127 = 5$. On a pas de valeur extrême de l'exposant, et on est donc dans le contexte d'une mantisse normalisée.

— On a une mantisse normalisée entièrement fixée à 0 sauf pour le second bit en partant de la gauche. On peut donc immédiatement lire que le nombre représenté vaut $2^{-2} = 0,25$. Puisqu'on a une mantisse normalisée, le nombre réellement encodé est 1 + le nombre représenté, soit $1 + 0,25 = 1,25$.

Au total, on assemble la valeur d'un réel égal à $(-1) \cdot 2^5 \cdot 1,25 = -40$.

5.4.2 Exercice 2

Énoncé : Pour cette série d'exercices, nous considérons un processeur fictif capable d'effectuer les opérations suivantes :

— Placer des valeurs constantes dans des registres.

— Lire et écrire des octets en mémoire, à des adresses constantes ou pointées par un registre. La destination des données lues et la source des données écrites sont toujours des registres.

— Effectuer des additions, des soustractions, et des comparaisons de valeurs de 8 bits contenues dans des registres.

— Réaliser un saut vers une autre partie du programme, soit de façon inconditionnelle, soit en fonction du résultat d'une comparaison.

Les exercices consistent à développer, pour chacun des problèmes suivants, un algorithme permettant de les résoudre, en n'employant que ces opérations. Les résultats peuvent être écrits en pseudocode. Le nom des registres et le nombre de registres disponibles peuvent être librement choisis. Les données d'entrée de chaque problème sont fournies dans des registres, ou placées en mémoire.

1. Écrire une valeur constante donnée dans tous les emplacements d'un tableau d'octets, donné par son adresse et sa taille.

2. Recopier un tableau d'octets, donné par son adresse et sa taille, vers une autre adresse donnée de la mémoire.

   Attention, il est possible que les zones de mémoire associées au tableau initial et à sa copie se recouvrent.

3. Retourner un tableau d'octets, donné par son adresse et sa taille, c'est-à-dire en permuter les éléments de façon à ce que le premier prenne la place du dernier, et vice-versa, le second la place de l'avant-dernier, et vice-versa, et ainsi de suite.

4. Calculer la somme de deux nombres représentés de façon petit-boutiste, à l'aide de n octets chacun. Les données d'entrée sont la valeur de n, et deux pointeurs vers la représentation des nombres.

5. Compter le nombre d'octets nuls dans un tableau d'octets d'adresse et de taille données.

6. En langage C, une chaîne de caractères est représentée par un tableau d'octets suivis par un octet nul. A partir d'un pointeur vers une telle chaîne de caractères, calculer sa longueur.

7. Convertir, dans une chaîne de caractères ASCII terminée par un zéro située à une adresse

donnée, toutes les lettres minuscules en majuscules. Les autres caractères ne doivent pas être modifiés.

8. Dans un tableau d'octet d'adresse et de taille données, calculer la longueur de la plus longue suite d'octets consécutifs identiques.

9. Déterminer si deux chaînes de caractères terminées par un zéro, d'adresses données, sont égales.

Solution :

(1) Pour rédiger efficacement les algorithmes répondant aux différents énoncés, on choisit d'abord des intitulés explicites et faciles à manipuler pour les opérations fictives constituant notre ensemble d'instructions :

writeConstToReg	Placer une constante dans un registre
readFromConstAdrToReg	Lire un octet à une adresse constante et le placer dans un registre
writeFromRegToConstAdr	Écrire un octet à une adresse constante à partir d'un registre
readFromPtrRegToReg	Lire un octet adressé par pointeur et le placer dans un registre
writeFromRegToPtrReg	Écrire un octet adressé par pointeur à partir d'un registre
addRegisters	Effectuer l'addition des valeurs dans deux registres
subRegisters	Effectuer la soustraction des valeurs dans deux registres
compRegisters	Comparer les valeurs dans deux registres
jump	Faire un saut de manière inconditionnelle
jumpIfFirstGreater	Faire un saut si la première opérande est la plus grande
jumpIfNotEqual	Faire un saut si les opérandes sont différentes

On choisit d'avoir plusieurs variantes d'opérations de saut conditionnel, car c'est le cas réel. De plus, on choisi arbitrairement que le résultat des opérations d'addition et de soustraction est placé dans le premier registre (l'opérande de gauche).

La convention exacte pour choisir les noms de ces opérations n'a aucune importance pour la machine. On peut juste attirer l'attention sur l'importance d'employer des noms qui indiquent aussi clairement que possible la fonction de l'opération, tout en étant aussi faciles d'utilisation que faire se peut. Par exemple, "readSomeMemoryCellAtThisAddress" est très explicite, mais assez laborieux à employer ; on lui préférera quelque chose comme "readAtAddress" qui conserve les mots-clés permettant d'identifier l'opération d'un coup d'oeil.

On choisit également arbitrairement que les registres de travail qu'on emploiera seront appellés $WorkA$, $WorkB$, etc... au besoin.

On nous demande de remplir toutes les cellules d'un tableau d'octets d'une unique valeur constante. On nous fourni l'adresse du tableau, et sa taille. Il est intéressant de remarquer que puisqu'on manie directement les cellules de mémoire contigües, on n'a pas à se préoccuper de si le tableau est en 2D, 3D ou au-delà ; tant qu'il a été alloué en mémoire en un unique bloc, on a besoin que de sa taille totale faite de la somme de la longueur de tous ses éléments selon toutes ses dimensions.

Les valeurs d'entrée sont l'adresse a du tableau à remplir, sa taille L, et la valeur X à utiliser pour le remplir. On suppose à défaut de plus d'information que ces valeurs sont respectivement fournies dans les registres $WorkA$, $WorkB$ et $WorkC$.

L'action à effectuer sur le tableau consiste en essence en un parcours de L octets, où L est la longueur du tableau, à partir d'une addresse a. Pour chacun de ces L octets on doit réaliser une opération d'écriture de la valeur X contenue dans $WorkC$ dans les registres des cellules du tableau qu'on localise à partir de leur adresse. Étant donné qu'on doit parcourir tout le tableau à partir de l'adresse de sa première cellule, on peut déjà prévoir qu'on pourra utiliser le registre $WorkA$ qui nous donne l'adresse du tableau comme registre de travail pour exécuter l'opération de remplissage comme suit :

```
writeFromRegToPtrReg( WorkB, WorkA ) ;
```

Pour parcourir le tableau, on doit manipuler le registre à l'adresse a, puis celui à l'adresse $a + 1$, etc... jusqu'à la dernière cellule à l'adresse $a + (L - 1)$. Pour ce faire, on a besoin d'un compteur qui progressera de 1 à chaque fois qu'on veut avancer d'un octet en parcourant les celulles aux adresses décallées de 0 à $L - 1$. Conceptuellement, le fonctionnement désiré est celui d'une boucle. Si, contrairement au C par exemple, on ne dispose pas dans notre ensemble d'instructions d'un "for(...)" ou un "while(...)" qui nous permettrait d'implémenter immédiatement la fonctionalité d'une boucle, on peut se débrouiller avec nos opérations de comparaison et de saut. En effet, on peut opérer comme suit :

```
Hypothèses    WorkA contient a
              WorkB contient L
              WorkC contient X

loop_init :   writeConstToReg( "1", WorkD ) ;
              addRegisters( WorkB, WorkA ) ; WorkB contient a+L
              subRegisters( WorkB, WorkD ) ; WorkB contient a+(L-1)

loop_start :  writeFromRegToPtrReg( WorkC, WorkA ) ;
              addRegisters( WorkA, WorkD ) ;
              jumpIfFirstGreater( WorkB, WorkA, "loop_start" );
```

À l'intitulé "loop_init", on initialise les variables nécéssaires pour travailler avec la boucle. On a choisi ici pour l'exercice de ne manipuler que des registres ; on aurait pu manipuler des

constantes en mémoire si les données de l'énoncé avaient pris ce format laissé au choix. On place un incrément fixe de +1 dans *WorkD*, puis on fait une addition et une soustraction des valeurs immédiatement à notre disposition pour finalement avoir $a + (L - 1)$ dans *WorkB*. De cette manière, on termine la phase d'initialisation avec deux registres de travail qui contiennent l'incrément de boucle et la cible de la boucle.

L'intitulé "loop_start" désigne le début de la boucle en elle-même. On voit qu'on y écrit la valeur contenue dans *WorkC* à l'adresse indiquée par *WorkA*, c'est-à-dire qu'on écrit la valeur *X* utilisée pour remplir le tableau dans l'adresse contenue par *WorkA*, qui au début du programme est l'adresse du tableau, i.e. de la première cellule du tableau.

Dans la première itération de la boucle, on incrémente ensuite cette adresse en effectuant la somme de *WorkA* et *WorkD* qui contient 1 (Dans un cas réel, on va plus vraisemblablement disposer d'une opération d'addition qui prend directement la constante "1" en argument). Ceci fait, *WorkA* contient donc l'adresse de la seconde cellule du tableau. On termine la première itération de la boucle en comparant cette adresse au contenu de *WorkB*, c'est-à-dire à l'adresse de la dernière cellule du tableau. Si le contenu de *WorkB* est plus grand que celui de *WorkA*, c'est-à-dire si l'adresse incrémentée de cette itération de la boucle reste plus petite ou égale à l'adresse de la dernière cellule du tableau, alors on saute à l'intitulé "loop_start" et on boucle donc pour une itération supplémentaire.

A la dernière itération, on rentre dans la boucle avec *WorkA* qui contient l'adresse de la dernière cellule, $a + L - 1$. On place la valeur de remplissage désirée à cette adresse, puis on incrémente l'adresse. La comparaison de *WorkB* et de *WorkA* identifiera cette fois-ci que *WorkA* contient $a + L$ qui est plus grand que $a + L - 1$ toujours inchangé dans *WorkB*, et on ne sautera donc pas à *loop_start*. On passera cette instruction et on continuera vers la suite du code, ou en l'occurence la fin du programme s'il n'est constitué que du pseudo-code défini ici.

On récapitule l'inventaire des variables comme suit :

— *WorkA* contient initialement l'adresse a du tableau, i.e. de sa première cellule, puis est incrémentée d'une unité à la fois pour pointer successivement sur chaque cellule jusqu'à prendre la valeur $a + L$ à la fin de la dernière itération de la boucle, i.e. l'adresse après la dernière cellule du tableau
— *WorkB* contient initialiement la longueur du tableau L, puis est manipulé pour contenir l'adresse $a + L - 1$ de la dernière cellule du tableau et est utilisée pour former l'invariant de la boucle - on aurait pu employer une constante en mémoire si on avait une opération de comparaison qui le permettait
— *WorkC* contient la valeur avec laquelle on doit remplir le tableau ; cette valeur n'est jamais modifiée - on aurait pu employer une constante en mémoire de manière équivalente
— *WorkD* contient 1 et est uniquement utilisé parce qu'on a qu'une opération d'addition qui demande en opérande deux registres - on aurait pu employer une constante en mémoire si on avait une opération d'addition qui le permettait

(2) Pour cet exercice, on utilise le même formalisme de pseudo-code que pour le précédent.

On nous demande de recopier l'entiereté d'un tableau à une nouvelle adresse, en précisant que cette nouvelle adresse peut mener à un recouvement du tableau original et de sa copie. Puisqu'on a pas de précision sur la politique à employer dans ce cas, on suppose par défaut qu'il convient simplement d'écraser les cellules du tableau original si nécéssaire. Le problème de cette opération est que si on écrase les données du tableau original progressivement par les cellules du début du tableau jusqu'à la dernière cellule de la zone de recouvrement, on perd à chaque fois les données écrasées, tel qu'on sera potentiellement bloqué lorsqu'il faudra également les recopier.

Les valeurs d'entrée sont l'adresse $a1$ du tableau à copier et l'adresse $a2$ de sa copie, ainsi que sa taille L. On suppose à défaut de plus d'information que ces valeurs sont respectivement fournies dans les registres $WorkA$, $WorkB$ et $WorkC$.

On peut exprimer l'entiereté des possibilités de recouvrement en définissant les relations entre a_1, l'adresse du tableau original, a_2, l'adresse de la copie à écrire, et L, la longueur du tableau (et donc de sa copie).

— Si $a2 + L < a1$, la copie se trouve entièrement avant le tableau original et on a aucun recouvrement. On peut recopier le tableau de n'importe quelle manière.
— Si $a1 < a2 + L < a1 + L$, la copie démarre avant l'original et se termine quelque part à l'intérieur de l'original. Si on copie le tableau élément par élément en commençant par le début et en avançant dans l'ordre des adresses croissantes, on écrasera uniquement des données déjà recopiées.
— Si $a2 = a1$, on est dans le cas trivial d'une copie à l'endroit actuel de l'original. Dans ce cas, il est inutile de faire quoi que ce soit, et on peut également copier le tableau élément par élément dans n'importe quel sens.
— Si $a1 < a2 < a1 + L$, la copie démarre à l'intérieur de l'original et se termine quelque part après l'original. Si on copie le tableau dans l'ordre croissant des adresses, on va écraser la fin de l'original avant de pouvoir la recopier et on sera donc bloqué. Une solution immédiate à ce problème est de copier le tableau dans l'ordre décroissant des adresses. De cette manière, on écrasera uniquement des données déjà recopiées.
— Si $a2 > a1 + L$, la copie se trouve entièrement après le tableau original et on a aucun recouvrement. On peut recopier le tableau de n'importe quelle manière.

En raisonnant verbalement sur le problème, on a identifié les problèmes éventuels et on a pu leur apporter une proposition de solution simple et efficace.

On remarque qu'on peut simplifier les contraintes listées ci-dessus en testant uniquement $a1 < a2$. Si c'est le cas, on copie le tableau cellule par cellule dans l'ordre décroissant des adresses ; cela correspond aux deux derniers éléments de la liste, le dernier n'imposant aucune contrainte sur la méthode employée. Si ce n'est pas le cas, on copie le tableau cellule par cellule dans l'ordre croissant des adresses. Cela correspond aux trois premiers éléments de la liste ; le second impose cette contrainte, tandis que le premier et le troisième sont indifférents vis-à-vis de la méthode employée. C'est une manière d'aboutir a un pseudo-code très simple à concevoir et à comprendre.

Pour comparer deux valeurs en utilisant notre ensemble d'instructions fictives, on doit les placer dans des registres puis utiliser la fonction de comparaison. Dans ce contexte précis, on sait que la comparaison que l'on veut effectuer doit mener à deux exécutions différentes d'un code par ailleurs sensiblement identique. Une manière de produire ce résultat est de définir un registre de travail qui contiendra un incrément d'adresse qui sera employé lors du parcours des tableaux ; cet incrément sera positif pour produire un parcours dans l'ordre croissant des adresses, et négatif sinon. Afin de définir cet incrément selon la comparaison entre $a1$ et $a2$, on peut dès lors immédiatement utiliser la fonction de saut conditionnel comme suit :

```
Hypothèses      WorkA contient a1
                WorkB contient a2
                WorkC contient L

case_finder :   jumpIfFirstGreater( WorkA, WorkB, "ascend" );

descend :       writeConstToReg( "-1", WorkD ) ;
                jump( "loop_start " );

ascend :        writeConstToReg( "+1", WorkD ) ;

loop_start :    (...)
```

A l'intitulé "case_finder", on compare $a1$ et $a2$ dans une fonction de saut conditionnel ; si $a1$ est plus grand que $a2$, on saute immédiatement à l'intitulé "ascend". Sinon, on passe cette étape et on arrive naturellement à l'intitulé "descend". Dans le premier cas, on insère un incrément positif +1 dans un quatrième registre de travail *WorkD*, puis on arrive naturellement à l'intitulé "loop_start" où il reste à insérer la boucle de parcours des tableaux. Dans le second cas, on insère un incrément négatif -1 dans *WorkD*, puis on saute inconditionnellement à l'intitulé "loop_start" pour ne pas exécuter le code de l'intitulé "ascend", i.e. le code du cas de progression dans le sens croissant.

Pour réaliser une boucle, on applique les mêmes principes qu'à l'énoncé précédent : on a besoin d'un compteur qui progresse à chaque itération et qu'on teste jusqu'à ce qu'il atteigne la valeur signalant la fin de la copie pour mettre fin à la boucle. Comme dans la solution à l'énoncé précédent, on peut utiliser comme compteur l'adresse de la cellule courante dans n'importe lequel des deux tableaux. Selon le sens de progression dans les tableaux, la cible à atteindre est soit la fin du tableau, soit son début. De la même manière, la première adresse à laquelle on opère est soit le début du tableau, soit la fin du tableau, respectivement. On peut donc implémenter cette boucle en ajoutant quelques clauses aux intitulés "ascend" et "descend", puis en utilisant les instructions suivantes :

```
Hypothèses      WorkA contient a1
                WorkB contient a2
                WorkC contient L

-------------------------------------------------------------------------
case_finder :   jumpIfFirstGreater( WorkA, WorkB, "ascend" );

descend :       writeConstToReg( "-1", WorkD ) ; WorkC contient -1

                addRegisters( WorkA, WorkC ) ; WorkA contient a1+L
                addRegisters( WorkA, WorkD ) ; WorkA contient a1+(L-1)

                addRegisters( WorkB, WorkC ) ; WorkB contient a2+L
                addRegisters( WorkB, WorkD ) ; WorkB contient a2+(L-1)

                writeFromRegToConstAdr( WorkA, "memoryAddress" ) ;
                readFromConstAdrToReg( "memoryAddress", WorkC ) ;
                addRegisters( WorkC, WorkD ) ; WorkC contient a1-1

                jump( "loop_start " );

ascend :        addRegisters( WorkC, WorkA ) ; WorkC contient a1+L
                writeConstToReg( "+1", WorkD ) ; WorkD contient +1

loop_start :    writeFromPtrRegToReg( WorkA, WorkE ) ;
                writeFromRegRegToPtrReg( WorkE, WorkB ) ;

                addRegisters( WorkA, WorkD ) ;
                addRegisters( WorkB, WorkD ) ;

                jumpIfNotEqual( WorkA, WorkC, "loop_start" );
```

Les clauses qu'on a ajouté avant la boucle font qu'au début de la boucle, on a les bonnes adresses de départ pour le parcours des deux tableaux dans *WorkA* et *WorkB*, le bon incrément dans *WorkD* et la bonne cible dans *WorkC*. On choisi arbitrairement d'utiliser l'adresse courante dans le premier tableau dans *WorkA* comme compteur de boucle. On remarque qu'on a du effectuer une manipulation passant par la mémoire pour déplacer l'adresse $a1$ de *WorkA* à *WorkC*, parce qu'on a pas d'opération de transfert de contenu d'un registre à l'autre ; dans une architecture réelle, on en a en général une. Cependant, c'est l'occasion de montrer un exemple de passage par la mémoire utilisant une cellule quelconque à une adresse *memoryAddress* qu'on sait être libre dans le segment de mémoire.

Dans la boucle, à l'intitulé "loop_start", on commence par copier le contenu de la cellule pointée par le registre *WorkA* dans un registre de travail temporaire *WorkE*, car on n'a pas d'opération pour copier directement une valeur entre deux adresses pointées par des registres. On

complète l'opération de copie de la cellule en copiant la valeur de $WorkE$ vers le registre pointé par $WorkB$. Ensuite, on incrémente les deux adresses dans $WorkA$ et $WorkB$ de l'incrément préalablement fixé à +1 ou -1 ; on progresse dans le sens croissant ou décroissant des adresses de cellules. Enfin, on compare l'adresse de la cellule courante du premier tableau dans $WorkA$ à la cible préalablement fixée dans $WorkC$. On retourne dans la boucle si ces valeurs sont différentes ; si elles sont égales, on a atteint la cible qui est soit juste après le tableau, soit juste avant, et on sort de la boucle, la copie étant achevée.

On récapitule l'inventaire des variables comme suit :

— $WorkA$ contient initialement l'adresse $a1$ du tableau original, i.e. de sa première cellule. Si on détermine qu'on doit effectuer une copie dans l'ordre croissant des adresses, cette valeur est ensuite incrémentée d'une unité à la fois pour pointer successivement sur chaque cellule jusqu'à prendre la valeur $a1 + L$ à la fin de la dernière itération de la boucle, i.e. l'adresse après la dernière cellule du tableau. Si on détermine qu'on doit effectuer une copie dans l'ordre décroissant des adresses, on manipule le registre pour contenir l'adresse $a1 + (L - 1)$ de la dernière cellule du tableau, puis on la décrémente progressivement pour parcourir le tableau en sens inverse jusqu'à finalement prendre la valeur $a1 - 1$ à la fin de la dernière itération de la boucle, i.e. l'adresse avant la première cellule du tableau.
— $WorkB$ contient initialement l'adresse $a2$ de la copie à effectuer. On agit sur cette valeur comme pour $a1$ dans $WorkA$, et elle évolue dans le sens croissant ou négatif des adresses au fil de la progression dans les tableaux jusqu'à prendre la valeur $a2 + L$ ou $a2 - 1$ à la dernière itération de la boucle.
— $WorkC$ contient initialement la longueur L du tableau à copier (et donc de sa copie). On l'utilise pour fixer le point de démarrage du parcours des tableaux, puis on le modifie pour contenir l'invariant permettant de mettre fin à la boucle - on aurait pu employer une constante en mémoire si on avait une opération de comparaison qui le permettait.
— $WorkD$ est utilisé pour contient l'incrément positif ou négatif employé dans la boucle et est uniquement utilisé parce qu'on a qu'une opération d'addition qui demande en opérande deux registres - on aurait pu employer une variable en mémoire si on avait une opération d'addition qui le permettait.
— $WorkE$ est utilisé comme un registre de travail temporaire, car on n'a pas d'opération pour copier directement une valeur entre deux adresses pointées par des registres.

(7) Comme précisé dans l'avant-propos, les conventions employées lors de la rédaction de pseudo-code sont arbitraires ; dans ce cas-ci, elles sont valables tant qu'elles restent cohérentes avec les spécifications exigées par le processeur. Cet exercice sera donc résolu avec un autre formalisme que nous allons définir ci-dessous :

(Registre) ← (Constante)	Placer une constante dans un registre
ptr (Registre ou adr. constante)	Déréférencer une adresse constante ou contenue dans un registre
(Registre) ← (RegA) ± (RegB)	Effectuer l'addition ou la soustraction de deux valeurs
JMP (étiquette)	Faire un saut de manière inconditionnelle
JMP (étiquette) if (condition)	Faire un saut conditionnel

*Remarque : l'instruction **ptr** vous rappelle peut-être l'opérateur de déréférencement " * " en langage C. En effet, celui-ci remplit une fonction semblable.*

Supposons également que les conditions peuvent être structurées de la même façon qu'en langage C (e.g. "Registre <= 10 && RegistreB != 0"). Quant aux registres, ils seront formulés de la façon suivante : *R_nomRegistre*.

Pour résoudre cet exercice, consultons d'abord la table ASCII et observons que les lettres minuscules de l'alphabet sont comprises sur l'intervalle [97,122]. Sachant que la majuscule correspondant à la lettre considérée dans cet intervalle se situe 32 positions avant celle-ci dans la table (la valeur de 'A' est 65, et celle de 'a' est 97 par exemple), l'idée serait alors de soustraire 32 à chaque valeur du tableau se trouvant dans l'intervalle [97,122]. Une solution envisageable serait dès lors celle ci :

```
Hypothèses  R_str : contient l'adresse de la chaîne

init :      R_current ← R_str
            R_offset ← 32
            R_inc ← 1

loop :      JMP "end" if ptr R_current == 0

            ;être dans [97,122] == ne pas être dans ]97,122[

            JMP "next" if (ptr R_current < 97 || ptr R_current > 122)
            ptr R_current ← ptr R_current - R_offset

next :      R_current ← R_current + R_inc
            JMP "loop"

end :
```

Analysons ce code étape par étape. Le registre *R_current* représente l'adresse de l'élément

traité à chaque itération. Faites donc bien attention que ce registre contient une adresse, et que pour pouvoir accéder à la valeur à laquelle cette adresse fait référence, il faut la déréférencer à l'aide de notre opérateur **ptr**. Aussi, celle-ci doit être incrémentée de 1 à chaque tour de boucle.

Donc nous devons affecter 1 à un registre *R_inc* puisque les opérandes des opérations arithmétiques de notre processeur fictif nous permettent uniquement de manipuler des valeurs contenue dans des registres et non des constantes. Ce registre nous permet donc d'avancer dans le tableau que nous manipulons, de façon similaire aux exercices précédents. Enfin, la valeur de décalage pour changer le caractère en majuscule est de 32. Nous affectons donc cette valeur au registre *R_offset*.

Maintenant que nous disposons de tous les registres dont nous avons besoin, il faut parcourir toute la chaîne (i.e. le tableau), jusqu'à la valeur zéro qui représente la fin de celle-ci. Il s'agit de la seule hypothèse que nous pouvons faire sur la structure du tableau. En effet, puisque la taille de la chaîne n'est pas donnée en entrée à notre programme, si celui-ci était de taille nulle, nous irions malheureusement lire un espace de mémoire qui ne nous appartient pas. Nous négligeons ce détail et supposons que la chaîne donnée n'est pas de taille nulle, et se termine forcément par un zéro.

Notez au passage qu'il s'agit d'un raisonnement à adopter dans ce genre de situation. Si vous trouvez un cas spécifique où votre programme pourrait ne pas fonctionner, relisez vos hypothèses (et donc l'énoncé) pour vérifier que votre solution est tout de même appropriée. Dans ce cas-ci, si nous disposions de la taille du tableau, il nous aurait été possible de ne pas entrer dans la boucle en adaptant le gardien par conséquent. Étant donné que cela n'est pas possible, nous devons nous contenter des spécifications du programme.

Une fois dans la boucle, il faut effectuer le décalage de 32 positions dans la table ASCII pour chaque lettre minuscule. Étant donné que nous ne disposons que d'opérations de sauts conditionnels, c'est-à-dire de modifications du flot d'interruption *lorsqu'une condition se réalise*, nous ne pouvons pas comme en C exécuter une instruction *si la condition est respectée* du type "do while (valeur sur [97,122])". En assembleur, nous sommes obligés de *sauter* certaines instructions selon la condition, et devons donc adopter notre condition à ce fonctionnement en en prenant le négatif, c'est-à-dire en passant une itération de la boucle de décalage si la valeur se trouve dans]97,122[. Il s'agit d'une pratique courante en langage Assembleur.

Finalement, à l'étiquette *next*, nous incrémentons l'adresse courante de 1 octet pour passer à la valeur suivante de la chaîne, et nous retournons au gardien de boucle.