Chapitre 6

Série d'exercices n°6 - Théorie chapitre III

6.1 Mode d'emploi

- **Décoder les modes d'adressage :**
  En pratique, on différencie trois type d'adressages pour les opérandes d'une instruction : l'adressage "immédiat" (*imm*) qui consiste à utiliser une valeur constante, l'adressage "registre (*reg*) qui consiste à utiliser la valeur contenue dans un registre, et l'adressage "mémoire" (*mem*) direct, indirect ou indirect indexé qui consiste à utiliser une adresse calculée à partir d'une combinaison de valeurs extraites de registres et de constantes, potentiellement au travers d'une opération mathématique. Dans le cas de l'adressage mémoire, il faut connaître les préfixes *qword, dword, word* et *byte* indiquant respectivement que l'espace mémoire à l'adresse indiquée doit être manipulé comme un ensemble de 64, 32, 16 ou 8 bits, c'est-à-dire un ensemble de 8,4,2 ou 1 cellules d'un octet (dans l'ordre petit-boutiste).

- **Connaître l'état des drapeaux et les opérations les affectant ou en dépendant :**
  Il est vital de connaître l'état des drapeaux à tout instant de l'exécution d'un programme assembleur. De nombreuses opérations comme ADD ou CMP les affectent, et les instructions de saut conditionnels largement employées en assembleur en dépendent directement. Dans l'architecture étudiée au cours, ces drapeaux sont CF, ZF, SF et OF, indiquant respectivement un report, un résultat égal à 0, le signe d'un résultat et un dépassement. Ils sont mis à jour pour de nombreuses instructions ; a priori, toutes celles impliquant une opération arithmétique.

- **Connaître l'état de la mémoire :**
  En assembleur, il n'y a quasiment aucune gestion automatique de la mémoire. On peut écrire, lire et écraser des valeurs en mémoire librement et à n'importe quelle adresse, sans aucune protection d'aucun type. Dès lors, il faut connaître la structure de la mémoire et est parfaitement conscient de la nature et du contenu des emplacements de mémoire qu'on utilise dans un programme.

- **Utiliser les registres de manière appropriée :**
  Les registres sont un type de mémoire particulier caractérisés par leur grande rapidité de lecture et d'écriture, ainsi que par leur nombre limité dans toute architecture. On veut les employer autant que possible, mais pour ce faire, on doit être efficace : on cherchera dans toute rédaction de programme assembleur à en employer le plus petit nombre strictement nécéssaire au bon fonctionnement du programme. On prendra également bien garde à comprendre leur structure exacte : par exemple, dans notre architecture, RAX est un registre de 64 bits et AX est un "second" registre qui correspond en réalité aux 16 bits de poids faible de RAX. Dès lors, toute opération sur RAX risque d'affecter immédiatement AX, et vice-versa.

6.2 Correctifs des exercices

6.2.1 Exercice 3.5.3 (1)

Énoncé :

Les instructions suivantes effectuent des transferts de 16 bits de données. On demande de décomposer chacune d'entre elles en deux instructions `MOV` réalisant la même opération, mais avec des opérandes de 8 bits.

1. `MOV AX, 0x1234`
2. `MOV AX, word ptr [0x1234]`
3. `MOV BX, CX`
4. `MOV word ptr [RBX], CX`
5. `MOV word ptr [RBP + RSI - 6], 0x32`

Solution :

1. `MOV AX, 0x1234` : Cette instruction déplace un mot de 4 valeurs hexadécimales, donc 16 bits, dans le registre AX. Cette valeur est une constante fournie explicitement, de manière littérale : on est dans le cas d'un **adressage immédiat** (*imm*). Le registre AX correspond lui-même aux 16 bits de poids faible du registre RAX ; cette opération est donc tout à fait légitime.

   Pour réaliser cette opération par blocs d'un octet à la fois, on doit séparer l'opérande de droite en une moitié correspondant au byte de poids fort, et une seconde moitié au byte de poids faible, et les déplacer dans les moitiés correspondantes du registre AX qui sont respectivement AH et AL. On a donc :

   ```
   MOV AH, 0x12
   MOV AL, 0x34
   ```

2. `MOV AX, word ptr [0x1234]` : Cette instruction déplace un **word**, c'est-à-dire un mot de 16 bits stocké en mémoire à l'adresse indiquée par la valeur constante, dans le registre AX. On renseigne une valeur par son adresse et sa taille : on est donc dans le cas d'un **addressage direct** (*mem*). Il est important de remarquer qu'une valeur qu'on renseigne par son adresse n'a bien de sens que si on en renseigne également la longueur par le qualificatif word, ou une de ses variantes. Si on ne connait pas la longueur de la représentation de la valeur, on ne peut pas savoir combien d'octets lire à partir de l'adresse fournie.

Pour réaliser cette opération par blocs d'un octet à la fois, on doit adresser les deux cellules de 8 bits composant la valeur de 16 bits séparément pour les déplacer individuellement dans AH et AL. Pour ce faire, on peut utiliser l'arithmétique de pointeur ; en effet, la cellule contenant le byte de poids fort de l'opérande de droite se situe une adresse après le byte de poids faible, puisque nous mémoire est organisée selon la convention petit-boutiste. Sachant qu'on doit donc adresser la même cellule que l'instruction d'origine pour le byte de poids faible, puis la cellule suivante pour le byte de poids fort, on peut réécrire deux opérations en adressage direct où l'on fait attention à renseigner cette fois-ci un **byte** puisqu'on manipule des octets :

```
MOV AH, byte ptr [0x1235]
MOV AL, byte ptr [0x1234]
```

3. `MOV BX, CX` : Cette instruction déplace le contenu du registre CX dans le registre BX. On renseigne une valeur par l'identifiant du registre qui la contient : on est donc dans le cas d'un **adressage registre** (*reg*). Comme pour AX et RAX, ces registres sont les 16 bits de poids faible de RCX et RBX, respectivement, et suivent par ailleurs les mêmes conventions d'utilisation. On ne manipule bien que des registres de 16 bits.

   Pour réaliser cette opération par blocs d'un octet à la fois, il suffit de séparer l'opération selon les bytes de poids fort et de poids faible des registres CX et BX qui sont respectivement CH et BH, et CL et BL. On manipule donc ces opérandes exactement comme on a fait pour AX jusqu'ici, tel qu'on a :

   ```
   MOV BH, CH
   MOV BL, CL
   ```

4. `MOV word ptr [RBX], CX` : Cette instruction déplace le contenu du registre CX vers un espace en mémoire dont l'adresse est contenue dans le registre RBX, et dont la longueur est celle d'un word, c'est-à-dire 16 bits. On renseigne un emplacement en mémoire par l'identifiant du registre (de 64 bits) qui contient son adresse et la longueur de cet emplacement : on est donc dans le cas d'un **adressage indirect** (*mem*). Comme pour l'adressage direct, l'opérande n'a de sens que si on indique bien la longueur de l'espace mémoire adressé par le qualificatif word, ou une de ses variantes. Dans ce cas-ci, on va écrire à cette adresse, et sans indication de la longueur de la valeur à écrire on ne saurait donc pas sur combien de cellules en mémoire écrire la valeur extraite de CX.

   Pour réaliser cette opération par blocs d'un octet à la fois, on doit comme fait précédemment utiliser l'arithmétique de pointeurs en sachant que l'octet de poids fort d'un espace mémoire de 16 bits de long se situe à l'adresse suivant celle de l'octet de poids faible. Cependant, puisqu'on est dans le cas d'un adressage indirect, on doit explicitement déléguer ce calcul d'adresses au processeur. Dans ce cas, on va adresser un emplacement en

mémoire par l'identifiant d'un registre qui contient une adresse de base, ainsi que par une opération qui vise à se déplacer depuis cette base vers une autre cellule : on est dans le cas d'un **adressage indirect indexé** (*mem*). On doit aussi prendre garde a bien renseigner la longueur de l'espace mémoire à écrire, qui devient un byte lorsqu'on sépare CX en ses deux moitiés CH et CL. L'opération à effectuer consiste en un simple incrément, et on a donc :

```
MOV byte ptr [RBX + 1], CH
MOV byte ptr [RBX], CL
```

5. `MOV word ptr [RBP + RSI - 6], 0x32` : Cette instruction déplace un mot de 2 valeurs hexadécimales exprimé comme une valeur littérale (**adressage immédiat**) à un emplacement en mémoire renseigné par son adresse, correspondant au résultat d'une opération impliquant les valeurs contenues dans les registres de 64 bits RBP et RSI et la soustraction d'une valeur numérique constante (**adressage indirect indexé**), et par sa taille, correspondant au qualificatif word indiquant un mot de 16 bits.

   On remarque que la valeur littérale 0x32 ne comporte ici pas explicitement 4 valeurs hexadécimales et donc 16 bits, tandis que l'opération est renseignée comme l'écriture d'un espace en mémoire de 16 bits. Dans ce sens, ce n'est pas un problème ; la valeur 0x32 peut être étenduee sur 16 bits. Différentes instructions peuvent avoir différentes règles d'extension des formats, par exemple en remplissant par la gauche avec des zéros, ou avec le bit de signe ; il faut se référer à la documentation technique exhaustive de l'architecture *x*86_64 pour en connaître les détails. On suppose ici que notre instruction MOV remplit par la gauche avec des zéros (Le bit de poids fort de 0x32 étant lui-même 0, une extension par copie du bit de signe donnerait le même résultat). On considère donc la valeur 0x0032 explicitement contenue sur 16 bits pour la suite de notre raisonnement.

   Pour réaliser cette opération par blocs d'un octet à la fois, on doit comme fait précédement utiliser l'arithmétique de pointeurs en sachant que l'octet de poids fort d'un espace mémoire de 16 bits de long se situe à l'adresse suivant celle de l'octet de poids faible. On doit aussi prendre garde a bien renseigner la longueur de l'espace mémoire à écrire, qui devient un byte lorsqu'on sépare 0x0032 en ses deux moitiés 0x00 et 0x32. L'opération à effectuer consiste en un simple incrément qui vient se superposer à l'opération actuelle tel que -6 devient -5, et on a donc :

```
MOV byte ptr [RBP + RSI - 5], 0x00
MOV byte ptr [RBP + RSI - 6], 0x32
```

6.2.2 Exercice 3.5.3 (2)

Énoncé :

Après l'exécution de la séquence d'instructions suivante, quel sera le contenu du registre AX ?

```
MOV RBX, 0
MOV RSI, 0
MOV AX, 0x100
MOV BX, AX
MOV word ptr [RBX + 1], AX
MOV word ptr [0x100], AX
MOV SI, word ptr [RBX + 1]
MOV AL, byte ptr [RSI]
MOV AH, byte ptr [RBX + RSI - 0x101]
```

Solution :

Pour comprendre la série d'instructions proposée, on va la décomposer en blocs plus facilement analysables.

```
MOV RBX, 0
MOV RSI, 0
MOV AX, 0x100
MOV BX, AX
```

Ces différentes opérations de déplacement de valeurs fixent RBX et RSI à zéro, tandis que AX est rempli avec 0x100 puis le propage dans BX. On note que puisque AX comporte 16 bits, la valeur 0x100 est étendue en 0x0100.

Pour l'instruction suivante, il faut se rappeller que RBX est un registre de 64 bits dont BX est la partie de poids faible de 16 bits. Dès lors, lorsqu'on a déplacé la valeur 0x0100 dans BX, on l'a aussi déplacé dans les 16 bits de poids faible de RBX. Comme on avait préalablement entièrement annulé RBX, son seul contenu non-nul est donc BX et on a que RBX = BX = 0x0100.

```
MOV word ptr [RBX + 1], AX
```

On déplace la valeur qui était dans AX, soit 0x0100, vers un espace en mémoire de 16 bits à une adresse dont la valeur est égale au contenu de RBX + 1 = 0x0100 + 1 = 0x0101.

Attention, comme on a écrit 16 bits à l'adresse 0x0101, on a en réalité écrit l'octet de poids faible de la valeur à l'adresse 0x0101 et l'octet de poids fort à l'adresse 0x0102, respectivement 0x00 et 0x01.

```
MOV word ptr [0x100], AX
```

On déplace la valeur qui était dans AX, soit 0x0100, vers un espace en mémoire de 16 bits à l'adresse 0x0100.

Comme on écrit une fois de plus 16 bits à une adresse donnée, on remplit en réalité deux cellules de mémoire. Ici, on place le byte de poids faible 0x00 à l'adresse 0x0100, et le byte de poids fort 0x01 à l'adresse 0x0101.

On a donc écrasé l'écriture de l'instruction précédente à l'adresse 0x0101, et à ce stade du programme, on a en mémoire :

- à l'adresse 0x0102 : la valeur 0x01 de l'avant-dernière instruction
- à l'adresse 0x0101 : la valeur 0x01 de l'instruction précédente
- à l'adresse 0x0100 : la valeur 0x00 de l'instruction précédente

```
MOV SI, word ptr [RBX + 1]
```

On déplace la valeur de 16 bits précédement enregistrée en mémoire à l'adresse RBX + 1 = 0x0101 dans le registre SI.

Dans cette première opération de lecture de la mémoire, on adresse un emplacement de mémoire de 16 bits à partir de l'adresse 0x0101. Ceci signifie qu'on va chercher une valeur de 16 bits faite d'un octet de poids faible lu à l'adresse 0x0101, et d'un octet de poids fort lu à l'adresse 0x0102. Étant donné l'état de la mémoire résultat des opérations précédentes, cette valeur déplacée vers le registre SI est donc 0x01 : 0x01 = 0x0101.

```
MOV AL, byte ptr [RSI]
```

On déplace la valeur de 8 bits en mémoire à l'adresse RSI dans le registre AL.

Comme pour les registres RAX et AX, le registre RSI est un registre de 64 bits dont SI est la partie de poids faible de 16 bits. Dès lors, lorsqu'on a déplacé la valeur 0x0101 dans SI, on l'a également déplacé dans la partie de poids faible de RSI. Comme on avait précédemment entièrement annulé RSI, on avait donc avant cette instruction que SI = RSI = 0x0101.

On effectue ici une lecture d'un mot de 8 bits à l'adresse RSI = 0x0101, et on déplace cette valeur dans le registre de 8 bits AL, qui est l'octet de poids faible du registre RAX (et donc également de AX). L'octet à l'adresse 0x0101 a précédement été fixé à la valeur 0x01, et donc a donc maintenant que AL = 0x01.

```
MOV AH, byte ptr [RBX + RSI - 0x101]
```

On calcule l'adresse égale à RBX + RSI - 0x101 et on insère la valeur à cette adresse dans le registre AH

Suite aux dernières opérations affectant ces registres, on a que RBX = 0x0100 et RSI = 0x0101. Dès lors, le calcul d'adresse se résout comme 0x0100 + 0x0101 - 0x0101 = 0x0100. On va lire l'octet à cette adresse qu'on a précédément fixé à la valeur 0x00, et on l'insère dans le registre AH.

Au total, à la suite de ces opérations, AX = AH : AL contient donc 0x00 : 0x01 = 0x0001.

On constate a posteriori que les premières instructions du programme visant à remplir de zéros les registres RBX et RSI nous on permis de les utiliser comme des adresses en ne manipulant que leurs 16 bits de poids faibles. En effet, si on ne les avait pas préalablement explicitement mis à zéro, on aurait aucune idée de ce qui se trouve dans les bits de poids forts de RBX et RSI, et donc aucune idée d'à quelle adresse de 64 bits un accès mémoire utilisant les valeurs contenues dans RBX et RSI comme adresses aurait accédé. C'est une pratique courante en assembleur, et il est important de la comprendre et de la maîtriser.

6.2.3 Exercice 3.5.3 (3)

Énoncé :

Pourquoi les instructions suivantes sont-elles incorrectes ?

1. `MOV RAX, CX`
2. `MOV byte ptr [EAX], CL`
3. `MOV 0x1234, AX`
4. `MOV EDX, qword ptr [RBP + RBX]`
5. `MOV RDX, RBP + RSI`
6. `MOV RDX, qword ptr[RBP - RSI - 9]`
7. `MOV RDX, qword ptr[2 * RBP + RBX + RCX]`

Solution :

On identifie individuellement les erreurs dans chacune des instructions suivantes :

1. `MOV RAX, CX` : Les opérandes de cette instruction sont RAX, un registre de 64 bits, et CX, un registre de 16 bit (CX correspondant aux 16 bits de poids faible du registre RCX de 64 bits). Cette opération ne correspond donc à aucune instruction réelle. On peut imaginer vouloir déplacer les 16 bits de CX vers une partie de RAX, mais dans ce cas, il reste encore une ambiguité sur la destination à l'intérieur de RAX. Dans tous les cas, cette formulation ne correspond pas à une syntaxe autorisée.
2. `MOV byte ptr [EAX], CL` : L'opérande de droite de cette instruction est un registre de 8 bits, et l'opération d'écriture est qualifiée par un byte, donc cette opération est a priori valide. Le problème se situe au niveau du registre EAX : ce registre correspond aux 32 bits de poids faible du registre RAX de 64 bits. Dans une architecture 64 bits, on ne peut absolument pas adresser d'espace mémoire par une valeur de 32 bits.
3. `MOV 0x1234, AX` : L'opérande de gauche est un littéral. Sans le qualificateur **ptr** indiquant que ce littéral doit être utilisé comme adresse, il ne peut être interpreté que comme une valeur constante. L'instruction MOV n'est pas définie dans ce cas de figure ; que pourrait-elle faire avec une valeur constante comme destination d'une valeur provenant d'un registre ?
4. `MOV EDX, qword ptr [RBP + RBX]` : L'opérande de droite de cette instruction est qualifiée comme un **qword**, c'est-à-dire un mot de 64 bits. Les deux registres RBP et RBX dont sont extraits l'adresse font bien 64 bits et sont donc parfaitement utilisables pour adresser la mémoire. La syntaxe employée pour les additionner est correcte. Le problème se situe au niveau de l'opérande de gauche : le registre EDX correspond aux 32 bits de poids faible du registre RDX de 64 bits. On ne peut pas déplacer une valeur de 64 bits dans un registre de 32 bits.

5. `MOV RDX, RBP + RSI` : L'instruction MOV n'est pas définie pour une syntaxe de ce type. Si l'on veut déplacer la somme des valeurs contenues dans RBP et RSI vers RDX, il faut d'abord effectuer leur somme en utilisant l'intruction ADD, puis utiliser le registre contenant le résultat de cette addition comme opérande de droite. Si l'on veut adresser un espace en mémoire à une adresse égale à la somme des valeurs de RBP et RSI, il faut employer la syntaxe de l'instruction précédente de la forme "ptr [...]".

6. `MOV RDX, qword ptr[RBP - RSI - 9]` : On a déjà développé les propriétés des registres RDX, RBP, RSI, et la syntaxe appropriée pour l'adressage indirect indexé. La seule erreur dans cette instruction consiste en le signe "-" précédent le registre RSI. En effet, cette opération de calcul d'adresse n'est strictement pas définie. Si l'on veut soustraire le contenu de RSI à RBP pour calculer l'adresse de l'espace mémoire dont on veut la valeur, il faut préalablement calculer l'opposé de RSI et l'y insérer, puis employer le signe "+" dans la syntaxe de calcul d'adresse.

7. `MOV RDX, qword ptr[2 * RBP + RBX + RCX]` : Comme pour l'instruction précédente, la seule erreur ici est dans la syntaxe exacte de calcul d'adresse en adressage indirect indexé. En effet, l'opération de calcul d'adresse n'accepte pas un registre comme troisième terme, mais uniquement une constante littérale.

6.2.4 Exercice 3.5.2 (3), (5) & (6)

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 :

(3) Pour résoudre cet énoncé, on utilise la convention de pseudo-code proposée dans la résolution de l'énoncé (1) de la série d'exercices n°5.

On nous demande de retourner un tableau d'octets donné par son adresse et sa taille. Cette opération peut être effectuée de plusieurs manières et ne présente a priori pas de cas à distinguer, si ce n'est éventuellement la différence entre un tableau de longueur paire ou impaire.

Les valeurs d'entrée sont l'adresse a du tableau à retourner et sa taille L. On suppose à défaut de plus d'information que ces valeurs sont respectivement fournies dans les registres $WorkA$ et $WorkB$.

En effet, si un tableau est de longueur paire, on peut le retourner en intervertissant le premier élément avec le dernier, puis le deuxième avec l'avant-dernier, etc ... jusqu'à intervertir l'élément d'indice $L/2 - 1$ (en comptant les éléments par des indices allant de 0 à $L - 1$) avec l'élément d'indice $L/2$. Si le tableau est de longueur impaire, on a un élément central qui ne doit pas être modifié par l'opération de retournement. Dans ce cas de figure, $L/2$ n'est pas un entier et on pourra intervertir chaque élément jusqu'à celui d'indice $[L/2]_{down} - 1$ où $[L/2]_{down}$ correspond à l'arrondi de $L/2$ vers le bas. L'élément suivant étant l'élément central, on a alors plus besoin de rien faire et on a terminé l'opération de retournement.

On voit immédiatement que cette structure de fonctionnement est celle d'une boucle, et que la discussion sur la parité de la longueur du tableau peut affecter la valeur cible de la boucle. En effet, on devra effectuer $L/2$ opérations d'échange de valeurs pour un tableau de longueur paire, et $[L/2]_{down} - 1$ opérations pour un tableau de longueur impaire ; la boucle doit donc utiliser un compteur qui doit s'arrêter après avoir été incrémenté un nombre correspondant de fois.

Dans un cas réel, on utiliserait vraisemblablement une simple fonction de décalage vers la droite pour diviser la longueur du tableau L en 2, et utiliser ce $L/2$ comme valeur de comparaison dans la boucle. Grâce aux propriétés des formats binaires, l'opération de décalage arrondi naturellement vers le bas, et le problème des tableaux impaires est donc immédiatement résolu sans effort additionnel.

Vu l'ensemble d'opérations limité dont on dispose ici, cependant, il faut être créatif pour définir une boucle qui s'arrêtera après $L/2$ itérations. C'est un cas un peu extrême d'un problème d'accès aux opérations habituellement disponibles en programmation que l'on rencontre de temps en temps dans les architectures parfois très minimalistes qu'on peut être amené à uti-

liser. Une solution possible est d'utiliser deux registres, l'un contenant 0 et l'autre $L-1$. Si on défini une boucle qui incrémente le registre contenant originellement 0 et décrémente le registre contenant originellement $L-1$ à chaque itération, ces registres vont progressivement contenir 1 et $(L-1)-1$, etc ... , jusqu'à contenir $L/2-1$ et $(L-1)-(L/2-1) = L/2$ après $L/2-1$ itérations.

À la prochaine itération, si L est pair, les deux registres échangeront leur valeur et le contenu du premier registre sera pour la première fois supérieure au second.

Si L est impair, après $[L/2]_{down} - 1$ itérations le premier registre contiendra $[L/2]_{down} - 1$ et le second $(L-1) - ([L/2]_{down} - 1) = [L/2]_{up}$. À la prochaine itération, donc après un total de $[L/2]_{up}$ itérations, les deux registres deviennent égaux à $[L/2]_{down}$, lorsque la boucle indique l'élément central. On ne doit rien faire avec cet élément, mais on peut également l'intervertir avec lui-même sans aucune conséquence vis-à-vis du résultat. À l'itération suivante, comme pour le cas pair, la valeur du premier registre devient alors pour la première fois supérieur au second.

On remarque que dans les deux cas de figure, on peut opérer avec une unique boucle qu'on fera se terminer lorsque la valeur contenue dans le premier registre deviendra supérieure à celle contenue dans le second. On peut implémenter ce fonctionnement comme suit :

```
Hypothèses    WorkA contient a
              WorkB contient L

loop_init :   writeConstToReg( "1", WorkC ) ;

              addRegisters( WorkB, WorkA ) ; WorkB contient a+L
              subRegisters( WorkB, WorkC ) ; WorkB contient a+(L-1)

loop_start :  readFromPtrRegToReg( WorkA, WorkD ) ;
              readFromPtrRegToReg( WorkB, WorkE ) ;

              writeFromRegToPtrReg( WorkD, WorkB ) ;
              writeFromRegToPtrReg( WorkE, WorkA ) ;

              addRegisters( WorkA, WorkC ) ;
              subRegisters( WorkB, WorkC ) ;

              jumpIfFirstGreater( WorkB, WorkA, "loop_start" );
```

On remarque que plutôt que d'utiliser de nouveaux registres, on part des valeurs disponibles dans *WorkA* et *WorkB* et on manipule ces registres pour opérer comme décrit ci-dessus. En effet, on peut les employer directement comme des pointeurs sur les cellules du tableau à inverser et les impliquer dans la comparaison menant à la fin de la boucle puisque toutes les expressions mathématiques développées ci-dessus sont identiques à une constante a près ajoutée des deux côtés, qui par exemple peut être l'adresse du tableau.

À l'intitulé "loop_init", on manipule la valeur fournie dans $WorkB$ pour qu'elle devienne $a + (L - 1)$ et indique donc l'adresse de la dernière cellule du tableau. On initialise également un registre de travail $WorkC$ destiné à contenir l'incrément utilisé dans la boucle.

Dans la boucle, à l'intitulé "loop_start", on commence par copier le contenu de la première cellule pointée par le registre $WorkA$ dans un registre de travail temporaire $WorkD$, et le contenu de la dernière cellule pointée par le registre $WorkB$ 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 $WorkD$ vers la cellule pointée par $WorkB$, et la valeur de $WorkE$ vers la cellule pointée par $WorkA$. Ensuite, on incrémente l'adresse dans $WorkA$ et on décrémente l'adresse dans $WorkB$. Enfin, on compare les deux adresses, et tant que celle dans $WorkB$ est supérieure à celle dans $WorkA$, on retourne dans la boucle pour la prochaine itération. Quand ce n'est plus le cas, on a dépassé la moitié du tableau en ayant éventuellement interverti l'élément central avec lui-même, et on a donc terminé de retourner le tableau ; la comparaison trouve que le contenu de $WorkB$ est plus petit que le contenu de $WorkA$ et on ne saute cette fois-ci plus dans la boucle, mais on accède au reste ou a la fin du programme.

On récapitule l'inventaire des variables comme suit :

— $WorkA$ contient initialement l'adresse a du tableau. Cette valeur est ensuite incrémentée d'une unité à la fois pour pointer successivement sur chaque cellule de la première moitié du tableau jusqu'à prendre la valeur $a + [L/2]_{up}$ à la fin de la dernière itération de la boucle.
— $WorkB$ contient initialement la longueur L du tableau. On manipule ce registre pour qu'il contienne l'adresse $a + (L - 1)$ de la dernière cellule du tableau. Cette valeur est ensuite décrémentée d'une unité à la fois pour pointer successivement sur chaque cellule de la seconde moitié du tableau dans l'ordre décroissant jusqu'à prendre la valeur $a+[L/2]_{down}-1$ à la fin de la dernière itération de la boucle.
— $WorkC$ est utilisé pour contenir l'incrément de 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 ou une constante littérale si on avait une opération d'addition qui le permettait.
— $WorkD$ est utilisé comme un registre de travail temporaire pour contenir la valeur issue de l'adresse $WorkA$, car on n'a pas d'opération pour copier directement une valeur entre deux adresses pointées par des registres. On a également explicitement besoin d'un espace de stockage intermédiaire afin de ne pas écraser la valeur à l'adresse $WorkB$ avant de pouvoir la recopier.
— $WorkE$ est utilisé comme un registre de travail temporaire pour contenir la valeur issue de l'adresse $WorkB$, pour la même raison.

(5) Pour résoudre cet énoncé, on utilise la convention de pseudo-code proposée dans la résolution de l'énoncé (7) de la série d'exercices n°5.

On nous demande de compter le nombre d'octets nuls dans un tableau d'octets renseigné par son adresse et sa taille. Il n'y a priori pas de difficulté conceptuelle particulière à effectuer cette opération ; on peut la réaliser en une simple boucle qui complète un nombre d'itérations égal à la longueur annoncée du tableau, d'une manière très similaire à l'énoncé précédent (3) ou à l'énoncé (1) résolu dans la série d'exercices n°5.

Les valeurs d'entrée sont l'adresse a du tableau à retourner et sa taille L. On suppose à défaut de plus d'information que ces valeurs sont respectivement fournies dans les registres *WorkA* et *WorkB*.

On implémente le fonctionnement décrit comme suit :

```
Hypothèses  WorkA : contient a
            WorkB : contient L

------------------------------------------------------------
init :      WorkC ← 1
            WorkD ← 0

            WorkB ← WorkB + WorkA ; WorkB contient a+L

loop :      JMP "end" if WorkA ≥ WorkB

            JMP "continue" if ptr(WorkA) ≠ 0
            WorkD ← WorkD + WorkC

continue :  WorkA ← WorkA + WorkC
            JMP "loop"

end :
```

Comme pour la solution développée pour l'énoncé (3), plutôt que d'utiliser de nouveaux registres, on part des valeurs disponibles dans *WorkA* et *WorkB* et on manipule ces registres pour réaliser le gardien de la boucle. On utilise *WorkA* comme un pointeur sur la cellule courante du tableau, et *WorkB* comme un pointeur sur la cellule après le tableau avant laquelle il faut s'arrêter.

A l'intitulé "init", on commence par initialiser deux registres de travail *WorkC* et *WorkD* destinés à contenir respectivement l'incrément utilisé dans la boucle, et le compte d'octets nuls trouvés dans le tableau. On manipule également la valeur fournie dans *WorkB* pour qu'elle devienne $a + L$ et indique donc l'adresse de la dernière cellule du tableau. Contrairement à la solution (3), on a dans ce formalisme de pseudo-code un saut conditionnel en "plus grand ou égal", ce qui nous permet d'économiser la ligne transformant $a + L$ en $a + (L - 1)$.

Au début de la boucle, à l'intitulé "loop", on commence cette fois-ci par le gardien de boucle. Contrairement aux solutions des énoncés (1) et (3), ce placement de l'instruction JMP nous protège contre un argument dégénéré correspondant à une taille de tableau nulle ou négative - on a donc de cette manière pratiqué un principe de programmation défensive, ce qui est toujours préférable à partir du principe que les arguments ne prendront jamais de valeurs problématiques. Le saut conditionnel compare les valeurs des registres *WorkA* et *WorkB*, c'est-à-dire les pointeurs sur la cellule courante et sur la cellule après le tableau, et lorsque l'adresse de la cellule courante dépasse l'adresse de la dernière cellule du tableau, on saute à l'intitulé "end' qui signale la fin du programme.

Ensuite, on compare la valeur pointée par l'adresse contenue dans le registre *WorkA*, c'est-à-dire la valeur de la cellule courante du tableau, à 0 ; si la valeur est différente de 0, on saute à l'intitulé "continue". Sinon, on exécute l'instruction suivante qui consiste à incrémenter le registre *WorkD* représentant le compteur d'octets nuls.

Enfin, au label "continue" qu'on atteint soit en passant l'instruction précédente, soit après avoir incrémenté le compteur d'octets nuls, on incrémente l'adresse de la cellule courante dans *WorkA* puis on saute à l'intitulé "loop" qui déterminera si on doit réaliser une itération suivante de la boucle ou non, selon si l'adresse de la cellule courante a dépassé le tableau ou non.

On récapitule l'inventaire des variables comme suit :

— *WorkA* contient initialement l'adresse a du tableau. Cette valeur est ensuite incrémentée d'une unité à la fois pour pointer successivement sur chaque cellule du tableau jusqu'à prendre la valeur $a + L$ à la fin de la dernière itération de la boucle.
— *WorkB* contient initialement la longueur L du tableau. On manipule ce registre pour qu'il contienne l'adresse $a + L$ et est utilisé dans le gardien de la boucle comme valeur à ne pas atteindre ou dépasser.
— *WorkC* est utilisé pour contenir l'incrément de 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 ou une constante littérale si on avait une opération d'addition qui le permettait.
— *WorkD* est utilisé pour contenir le compte d'octets nuls dans le tableau. Dans un contexte réel, il s'agirait d'un argument de sortie de la fonction qu'on a rédigé ; il conviendrait soit de préciser dans le prototype de cette fonction que le registre *WorkD* contient l'argument de sortie, soit de déplacer cette valeur à un endroit approprié pour un argument de sortie.

(6) Pour résoudre cet énoncé, on utilise la convention de pseudo-code proposée dans la résolution de l'énoncé (7) de la série d'exercices n°5.

On nous demande de compter le nombre de caractères d'une chaîne de caractère consistant en un tableau d'octets dont la fin est indiquée par un octet nul. On nous renseigne ce tableau par son adresse. Il n'y a priori pas de difficulté conceptuelle particulière à effectuer cette opération ; on peut la réaliser en une simple boucle qui parcourt les cellules du tableau dont on a l'adresse jusqu'à rencontrer une cellule égale à zéro.

L'unique valeur d'entrée est l'adresse *a* du tableau. On suppose à défaut de plus d'information que cette valeur est fournie dans le registre *R_str*.

On implémente le fonctionnement décrit comme suit :

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

init :      R_inc ← 1
            R_index ← 0

loop :      R_ad ← R_str + R_index
            JMP "end" if ptr R_ad == 0

            R_index ← R_index + R_inc
            JMP "loop"
end :
```

L'unique subtilité de cet exercice consiste en l'utilisation du registre *R_index*. En effet, comme son nom l'indique, il est employé comme un index à ajouter à l'adresse de la chaîne de caractère pour pointer sur la cellule courante du tableau. Cependant, il est aussi employé comme le compteur de caractères de la chaîne. En fait, ces deux fonctions sont strictement identiques dans cette solution au problème, puisque lorsque l'index atteint la valeur faisant pointer $a + R\_index$ sur le zéro terminant la chaîne, il est exactement égal à la longueur de la chaîne, i.e. au nombre de caractères valides du tableau.

A l'intitulé "init", on commence par initialiser deux registres de travail *R_inc* et *R_index* destinés à contenir respectivement l'incrément utilisé dans la boucle, et l'index dans la chaîne ainsi que le compte de caractères de la chaîne, comme expliqué ci-dessus.

Au début de la boucle, à l'intitulé "loop", on indexe l'adresse de départ du tableau pour adresser la cellule courante, puis on évalue le gardien de la boucle. Si le contenu de la cellule pointée par l'adresse indexée est nul, on saute à l'intitulé "end" qui signale la fin du programme. Si on ne rencontre pas encore de zéro, alors on exécute les deux instructions suivantes qui incrémentent l'index et qui font boucler le programme vers l'éventuelle prochaine itération de la boucle. Lorsqu'on sort de la boucle, *R_index* contient l'index qui ajouté à l'adresse *a* donne l'adresse du caractère nul. Cette valeur correspond bien à la longueur de la chaîne de caractère, et donc *R_index* est interpreté comme l'argument de sortie du programme.

On récapitule l'inventaire des variables comme suit :

— *R_str* contient l'adresse *a* du tableau. On aurait pu, comme dans les solutions précédentes, incrémenter progressivement ce registre afin de pointer sur les cellules successives du tableau, mais on a préféré employer un second registre *R_ad* afin d'illustrer au plus clairement le rôle double de *R_index*. En pratique cependant, on préférera économiser le nombre de registres qu'on utilise.
— *R_inc* est utilisé pour contenir l'incrément de 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.
— *R_index* contient une valeur qui est à la fois l'index de la cellule courante du tableau, et le nombre de caractères valides de la chaîne.
— *R_ad* est utilisé pour contenir l'adresse indexée égale à $a + R\_index$.