Chapitre 7

Série d'exercices n°7 - Théorie chapitre III

7.1 Correctifs des exercices

7.1.1 Exercice 3.5.1

Énoncé : Le plus simplement possible, décrivez l'effet des instructions x86-64 suivantes :

1. `AND word ptr[R8], 0xff`
2. `CMP EAX, EAX`
3. `PUSH qword ptr[RAX]`
4. `RET`

Solution :

1. `AND word ptr[R8], 0xff` : L'instruction AND calcule la fonction logique d'intersection (notée entre autres par "et", "&" ou "∩") entre chaque couple de bits de ses opérandes, et stocke le résultat de cette opération dans son opérande de gauche. La fonction d'intersection n'est positive que quand tous ses éléments sont vrais. Dans notre architecture, l'instruction AND calcule cette fonction par couple de bits tel que chaque bit à la position *n* du résultat sera donc fixé à 1 uniquement si les deux bits *n* des deux opérandes étaient eux-même à 1.

   Ici, on effectue un AND entre une valeur littérale égale à 0xFF, donc un mot de 8 bits entièrement rempli de 1, et la valeur de 16 bits (qualifiée par word) en mémoire à l'adresse contenue dans le registre R8. Cette opération n'a de sens que si le littéral explicitement défini sur 8 bits est étendu sur le format 16 bits de l'opérande de gauche. En supposant que le littéral 0xFF est étendu vers la gauche avec des zéros, on effectue donc en réalité un AND avec la valeur sur 16 bits 0x00FF. Étant donné la définition du AND présentée

ci-dessus, cette opération revient à conserver les 8 premiers bits de poids faible de l'opérande de gauche, et annuler ses 8 bits de poids fort, puisque $(x \ \& \ 1) = x$ et $(x \ \& \ 0) = 0$, quel que soit x. Le résultat est stocké à l'adresse correspondant à la valeur contenue dans R8, en écrasant la valeur originale, donc en perdant les 8 bits de poids fort.

Si l'on suppose plutôt que le littéral est complété par la gauche avec le bit de signe, alors 0xFF devient 0xFFFF sur 16 bits. Dans ce cas, par définition de la fonction d'intersection, on va comparer chaque bit à 1 et reproduire le mot de 16 bits de l'opérande de gauche à l'identique. En d'autres termes, cette opération n'aura aucun effet (sinon perdre du temps, ce qui est parfois désirable).

2. `CMP EAX, EAX` : L'instruction CMP calcule la différence $a - b$ entre ses deux opérandes de gauche a et de droite b dans l'unique but de les comparer. Plutôt que d'explicitement fournir le résultat de cette différence, elle met à jour l'ensemble de drapeaux qui indiquent la relation entre deux opérandes d'un calcul : ZF est fixé à 1 si les deux opérandes sont égales, quand $a - b = 0$, et SF est fixé à 1 si l'opérande de gauche est plus petite que l'opérande de droite, quand $a - b < 0$. Ces drapeaux sont ensuite utilisés par d'autres instructions dans leur fonctionnement normal, comme par exemple les instructions de saut conditionnel qui basent entièrement l'évaluation de leur condition de saut sur la valeur des drapeaux.

   Ici, on compare EAX à lui-même. On détermine donc immédiatement que le résultat de la différence calculée par CMP est strictement nul. En conséquence, CMP va donc fixer à 1 le drapeau ZF, et à 0 le drapeau SF.

3. `PUSH qword ptr[RAX]` : L'instruction PUSH manipule la **pile**, une zone spéciale de la mémoire qui est gérée en "LIFO", c'est-à-dire que lorsqu'on y place des valeurs dans des cellules successives, i.e. qu'on "empile" ces valeurs, on doit les récupérer dans l'ordre inverse. La dernière valeur empilée doit être récupérée en premier, etc, jusqu'à vider la pile de tout ce qu'on y a mis. Ce fonctionnement correspond aux opérations récurentes d'appels de fonctions qui nécéssitent de sauvegarder un ensemble de registres de travail pour garantir la bonne exécution du programme, puis de récupérer ces valeurs sauvegardées dans l'ordre inverse ; ceci permet un nombre quelconque de sauts et d'appels de fonctions dans le programme et donc des types de fonctionnement très puissants comme les récursions, qui seront abordées entre autres au cours de *Computation Structures*.

   Dans ce cas-ci, on empile une valeur qualifiée de **qword**, c'est-à-dire une valeur de 64 bits, qu'on va lire en mémoire à l'adresse contenue dans le registre RAX. On rappelle que l'instruction n'autorise **que** des opérandes de 64 bits. L'instruction PUSH décrémente de 8 la valeur contenue dans RSP afin de signaler que la pile va grandir de 8 cellules (on décrémente bien RSP car la pile croît dans le sens des adresses décroissantes et décroît dans le sens des adresses croissantes). Elle écrit ensuite 8 cellules successives d'un octet sur la pile comme suit : on commence à l'adresse correspondant à la valeur actuelle de RSP, qui désigne l'adresse du dernier octet de la pile, et on y place l'octet de poids faible de la

valeur de 64 bits à copier. Ensuite, on écrit le reste des octets selon la convention petit-boutiste, c'est-à-dire jusqu'à ce que l'octet de poids fort soit écrit sur la pile à l'adresse RSP+7, qui désigne le 8ème octet en comptant à partir de l'extrêmité de la pile.

4. `RET` : L'instruction RET est conçue comme l'instruction finale d'une fonction à laquelle on a accédé au moyen d'une instruction CALL. Elle va se charger de récupérer la dernière valeur de 64 bits placée sur la pile (comme par un POP) et de la placer dans RIP. Si la convention d'appel a été correctement respectée, cette valeur sur la pile est celle qui a été placée par le CALL employé pour accéder à la fonction, c'est-à-dire l'adresse de l'instruction suivant CALL dans la liste des instructions à exécuter dans le programme (i.e. la ligne suivant l'appel de fonction dans le code assembleur). Dans ce cas, RIP sera maintenant configuré de telle manière que l'exécution du programme pourra reprendre son cours purement linéaire à partir de la ligne suivant le CALL, après que l'appel à la fonction ait été résolu.

7.1.2 Exercice 5.3

Énoncé : Le plus simplement possible, décrivez l'effet des instructions x86-64 suivantes :

1. `IMUL AX`
2. `SUB R8D, -1`
3. `POP qword ptr[0x100]`
4. `CALL qword ptr[8 * RAX]`

Solution :

1. `IMUL AX` : L'instruction IMUL effectue une multiplication signée avec pour première opérande implicite le registre RAX ou un de ses sous-ensembles de taille égale à la seconde opérande, et comme seconde opérande explicite celle renseignée comme l'unique argument de cette instruction. En l'occurence, on utilise comme argument le registre AX. Ce registre de 16 bits correspond aux 16 bits de poids faible du registre RAX de 64 bits. Comme on renseigne un argument de 16 bits à l'instruction de multiplication, l'argument implicite sera le sous-ensemble de 16 bits de RAX, c'est-à-dire AX lui-même. Le résultat de la multiplication sera enregistré dans le couple de registres DX :AX, sur 32 bits, afin de garantir de pouvoir contenir le résultat de la multiplication comme développé dans le chapitre II.

   Ici, on aura donc les 16 bits de poids fort de la représentation signée de AX * AX dans DX, et les 16 bits de poids faibles dans AX.

2. `SUB R8D, -1` : L'instruction SUB effectue la soustraction entre deux opérandes, et enregistre le résultat dans l'opérande de gauche. Ici, on calcule donc la différence entre la valeur dans le registre R8B et le littéral -1. Ceci revient évidemment à faire la somme entre la valeur dans R8B et +1. En somme, on incrémente R8D.

3. `POP qword ptr[0x100]` : L'instruction POP manipule la **pile**, une zone spéciale de la mémoire qui est gérée en "LIFO", c'est-à-dire que lorsqu'on y place des valeurs dans des cellules successives, i.e. qu'on "empile" ces valeurs, on doit les récupérer dans l'ordre inverse. La dernière valeur empilée doit être récupérée en premier, etc, jusqu'à vider la pile de tout ce qu'on y a mis. Ce fonctionnement correspond aux opérations récurentes d'appels de fonctions qui nécéssitent de sauvegarder un ensemble de registres de travail pour garantir la bonne exécution du programme, puis de récupérer ces valeurs sauvegardées dans l'ordre inverse ; ceci permet un nombre quelconque de sauts et d'appels de fonctions dans le programme et donc des types de fonctionnement très puissants comme les récursions, qui seront abordées entre autres au cours de *Computation Structures*.

Dans ce cas-ci, on dépile une valeur qualifiée de **qword**, c'est-à-dire une valeur de 64 bits, qu'on va écrire en mémoire à l'adresse 0x100. On rappelle que l'instruction n'autorise **que** des opérandes de 64 bits. L'instruction POP lit donc 8 cellules successives d'un octet sur la pile en commençant à l'adresse correspondant à la valeur actuelle de RSP, qui désigne l'adresse du dernier octet placé sur la pile, puis en progressant dans l'ordre croissant des adresses puisqu'on enlève des données. Ces valeurs sont copiées dans la mémoire à partir de l'adresse 0x100 où sera placé l'octet de poids faible jusque l'adresse 0x107 où sera placé l'octet de poids fort, puisque notre mémoire est organisée selon la convention petit-boutiste. Une fois la copie effectuée, l'instruction incrémente de 8 la valeur contenue dans RSP afin de signaler que la pile vient de diminuer de 8 cellules (on incrémente bien RSP car la pile croît dans le sens des adresses décroissantes et donc décroît dans le sens des adresses croissantes).

4. `CALL qword ptr[8 * RAX]` : L'instruction CALL permet d'appeller des segments de code implémentant des fonctions. Elle va se charger de sauvegarder sur la pile la valeur actuelle de 64 bits du registre RIP, c'est-à-dire l'adresse de l'instruction suivant CALL dans la liste des instructions à exécuter dans le programme (i.e. la ligne suivante du code assembleur). Ensuite, elle va charger RIP avec l'adresse qui lui est confiée en argument et qui est sensé être l'adresse de la première instruction de la fonction à effectuer ; cette fonction devra elle-même se terminer par l'instruction RET qui effectuera les opérations inverses sur RIP et permettra le retour à l'instruction suivant CALL originellement enregistrée sur la pile, si l'on suit correctement les conventions d'appel de fonctions.

   Ici, on emploie l'adressage indirect indexé pour renseigner comme adresse de la première instruction de la fonction le contenu du registre RAX multiplié par 8. L'instruction CALL va donc sauver la valeur de RIP sur la pile, puis faire sauter l'exécution du programme à l'adresse 8*RAX. Il est intéressant de remarquer qu'on qualifie bien l'accès mémoire de qword, qui indique qu'on doit lire à l'adresse 8*RAX une valeur de 64 bits ; c'est en effet une condition indispensable pour que cette valeur puisse elle-même être utilisée comme une adresse dans notre architecture 64 bits.

7.1.3 Exercice 5.4

Énoncé : Décrivez, le plus simplement possible, l'effet des fragments de code assembleur x86-64 suivants. (On suppose que la pile est correctement configurée avant leur exécution.)

— Fragment 1 :

```
PUSH RBX
PUSH RAX
MOV  AL, byte ptr[RSP+0]
MOV  BL, byte ptr[RSP+7]
MOV  byte ptr [RSP+0], BL
MOV  byte ptr [RSP+7], AL
POP  RAX
POP  RBX
```

— Fragment 2 :

```
   PUSH RDX
   CMP  AX, 0
   JGE  p
n: MOV RDX, -1
   MOV DX, AX
   JMP f
p: MOV RDX, 0
   MOV DX, AX
f: MOV RAX, RDX
   POP RDX
```

Solution :

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

```
PUSH RBX
PUSH RAX
```

On utilise l'instruction PUSH pour sauver sur la pile la valeur des registres RBX et RAX.

Lorsqu'on place sur la pile les valeurs des registres RBX et RAX faisant tous les deux 64 bits, on empile 2 blocs de 8 octets. L'instruction PUSH s'occupe de réaliser la copie et de décrémenter la valeur du pointeur sur le dernier élément de la pile, RSP, de manière correspondante à l'ajout : dans ce cas, on ajoute au total 16 octets, et on verra donc RSP être décrémenté de 16.

```
MOV AL, byte ptr[RSP+0]
MOV BL, byte ptr[RSP+7]
```

On va chercher le dernier octet empilé sur la pile, pointé par RSP, et on le copie dans le registre AL. On va chercher le 8ème octet en partant de l'extrémité de la pile, pointé par RSP+7, et on le copie dans le registre BL.

La dernière instruction ayant modifié les 8 dernières cellules de la pile correspond au PUSH RAX du bloc d'instructions précédent. En effet, puisque RAX est un registre de 64 bits, les 8 octets à l'extrémité de la pile correspondent à l'entiereté de son contenu organisés selon la convention petit-boutiste de notre architecture. Les opérations de ce bloc d'instruction placent donc respectivement l'octet originel de poids faible de RAX dans AL, et l'octet originel de poids fort de RAX dans BL. On remarque que rien ne change dans RAX, puisque AL accueille la valeur originale de AL, mais que RBX est modifié au travers de BL.

```
MOV byte ptr [RSP+0], BL
MOV byte ptr [RSP+7], AL
```

On va chercher les valeurs des registres BL et AL, et on les place respectivement aux adresses RSP+0 et RSP+7 ; c'est-à-dire qu'on écrase les valeurs de l'octet de l'extrémité et du 8ème octet de la pile avec respectivement les valeurs contenues dans BL et AL.

On avait précédement placé l'octet de poids fort de RAX dans BL. Ici, on reprend cette valeur dans BL et on la réinjecte dans la pile, à l'endroit de l'octet de poids faible de la copie de RAX enregistrée sur la pile. De la même manière, on reprend l'octet de poids faible copié dans AL et on le réinjecte dans la pile, à l'endroit de l'octet de poids fort de la copie de RAX. En fin de compte, on a modifié la copie de RAX sur la pile tel que les octets de poids faible et de poids fort ont été inversé par rapport à la valeur originale de RAX.

```
POP RAX
POP RBX
```

On dépile le mot de 64 bits à l'extrémité de la pile, et on le copie dans RAX. On dépile le mot suivant de 64 bits, et on le copie dans RBX.

Étant donné les modifications que les instructions précédentes ont appliqué à la copie de RAX située à l'extrêmité de la pile, on réalise tout de suite qu'à la fin de ce fragment de code, la valeur désempilée copiée dans RAX correspond à sa valeur originelle, mais avec les octets de poids faible et de poids fort intervertis. La copie de RBX sur la pile a été inchangée, et on la

restaure donc à l'identique de son état avant le fragment de code.

On remarque que dans ce code, on a utilisé BL et AL comme des registres de travail, et que les opérations qu'on a fait dessus n'ont pas affecté la valeur de RAX et RBX dont ils font partie entre le moment avant et le moment après l'exécution de ce fragment de code. Ceci est garanti par l'utilisation correcte de la pile pour les registres de travail : on en sauvegarde la valeur sur la pile avant de les manipuler comme des espaces de calcul temporaires, et on les restaure à partir de la pile à la fin du calcul. Le registre RAX est lui bel et bien modifié, mais c'est visiblement le but explicite du fragment de code et c'est bien les modifications sur sa copie dans la pile qui sont conservées, plutôt que les manipulations sur AL alors qu'il était utilisé comme registre de travail.

Dans les architectures réelles, les registres de travail sont en général des registres distincts de registres de calcul explicitement manipulés par certaines opérations comme le sont RAX et RBX. On impose que la valeur de ces registres de travail soient sauvegardés à l'appel de la fonction, et restaurés après la fin de celle-ci s'ils ont été modifiés.

(2) On analyse le fragment de code de la même manière, bloc par bloc.

```
PUSH RDX
```

On utilise l'instruction PUSH pour sauver sur la pile la valeur du registre RDX. Connaissant les bonnes pratiques de l'assembleur, on peut s'attendre à ce que RDX soit utilisé comme registre de travail et restauré à la fin du fragment de code.

```
CMP AX, 0
JGE p
```

On compare d'abord la valeur de AX à 0. Ensuite, on invoque JGE ; l'opération de saut conditionnel prend en opérande une adresse de 64 bits. Cette adresse n'étant souvent pas connue explicitement, on emploie généralement des "étiquettes" comme l'intitulé "p" qui permettent d'associer un nom symbolique à une adresse qui ne sera déterminée numériquement que lors de la génération du code machine. On saute à l'adresse associée au label "p" si AX est plus grand ou égal à 0 (au travers de la valeur des drapeaux fixées par CMP).

Étant donné que ce bloc d'instructions réalisant une opération de saut conditionnel fait passer le flot d'exécution au label "p" si l'opérande AX est plus grande ou égale à 0, on peut inférer que "p" correspond à "positif", i.e. c'est une section du programme destiné à gérer les valeurs (non-strictement) positives. Inversément, si AX est strictement négatif, on n'effectue pas le saut mais on continue plutôt l'exécution séquentielle du programme, ce qui nous amène naturellement aux instructions du label "n". On infère de manière correspondante que "n" correspond donc à "négatif", et indique la section du programme destinée à gérer les valeurs (strictement) négatives.

```
n:  MOV RDX, -1
    MOV DX, AX
    JMP f
```

On place la valeur "-1" dans RDX, puis on déplace le contenu de AX dans DX, et on saute inconditionnellement au label "f".

Dans ce bloc, on construit la valeur complète sur 64 bits du registre RDX par morceaux. En effet, on commence par y insérer la valeur -1, qui comme vu au chapitre II implique un ensemble

de 1 en complément à deux. Ceci fait, on fixe les 16 bits de poids faible de RDX, c'est-à-dire le registre DX, à la valeur de AX.

On comprend en réalité que si AX est une valeur sur 16 bits négative (puisqu'on ne rentre dans cette section que si AX est strictement négatif), on vient de l'étendre à un support de 64 bits dans RDX en propageant le bit de signe égal à 1. En effet, les 16 bits de poids faible de RDX = -1 sont écrasés par la valeur de AX, et il ne reste pour l'entiereté des autres bits de poids fort que des 1 correspondant à une représentation en complément à deux d'un nombre dont les bits de poids forts de la représentation en valeur absolue sont nuls.

```
p:  MOV RDX, 0
    MOV DX, AX
```

On place la valeur "0" dans RDX, puis on déplace le contenu de AX dans DX.

On voit que dans cette section à laquelle on ne peut accéder que si AX est positif, on construit une valeur de 64 bits sur RDX où tous les bits sont à zéro, sauf les 16 bits de poids faible dans lesquels on recopie la valeur de AX. Comme pour la section "n", on a étendu la valeur de AX sur 64 bits, cette fois-ci en remplissant par la gauche avec des 0 puisque la valeur est positive ou nulle.

On remarque également qu'après cette section, on arrive naturellement au label "f", puisqu'il s'agit de l'instruction suivante.

```
f:  MOV RAX, RDX
    POP RDX
```

On place la valeur de RDX dans RAX, puis on dépile le dernier mot de 64 bits vers le registre RDX.

On accède à cette section à la suite des opérations des sections "n" ou "p" selon si AX avait été évalué comme négatif ou positif ; dans les deux cas, on a donc dans RDX la valeur de AX correctement étendue à 64 bits selon si cette valeur est négative ou positive. Dans cette section, on recopie cette valeur étendue assemblée dans RDX dans vers le registre RAX : ce faisant, on complète l'opération d'extension de AX à 64 bits en remplissant RAX avec la nouvelle représentation étendue.

Enfin, on dépile vers le registre RDX. Puisqu'on a pas manipulé la pile depuis l'empilement de la valeur originale de RDX, cette opération restaure RDX à son état de départ indépendamment de tout ce qui a pu lui arriver dans ce fragment de code. Ceci confirme notre hypothèse de départ que RDX est un simple registre de travail, et qu'on respecte la bonne pratique qui consiste à en garantir la valeur avant et après l'exécution du fragment de code.

7.1.4 Exercice 2 (troisième partie)

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 :

(8) 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.

La logique de l'algorithme proposé est la suivante : nous disposons en entrée d'un tableau de N octets que nous pouvons représenter formellement comme une suite $T = (e_0, ..., e_{N-1})$. Notre but est donc de renvoyer la taille de la plus longue sous-suite d'éléments identiques. Une bonne démarche à adopter lorsqu'il faut résoudre ce type de problèmes est de le généraliser et de réfléchir à une solution sans tout de suite penser aux contraintes du langage manipulé.

Une idée de résolution serait alors la suivante :

1. Choisir un élément de référence dans le tableau ;
2. Le comparer à l'élément suivant
   — Si cet élément est identique, on incrémente un compteur qui se réfère à la taille de la sous-suite actuelle ;
   — Sinon, cela signifie que la sous-suite considérée est terminée et qu'une autre sous-suite d'éléments commence. Dans ce cas, il faut réinitialiser le compteur et choisir cet élément comme nouvel élément de référence.
3. Mettre à jour le *Max* enregistré jusqu'à présent s'il est inférieur à la taille de la sous-suite que l'on vient de considérer ;
4. L'algorithme s'arrête une fois que l'on a parcouru tous les éléments de la suite donnée.

Maintenant que nous avons une idée d'algorithme de résolution, nous pouvons nous pencher sur l'implémentation. Notons pour commencer que l'on peut réutiliser les registres donnés en entrée au programme pour avancer dans le tableau. En effet, si on a *R_array* qui symbolise l'adresse du tableau en question, et *R_size* sa taille, afin d'éviter de passer par un registre intermédiaire, nous pouvons directement incrémenter la valeur de *R_array* pour avancer dans le tableau, et décrémenter *R_size* pour l'utiliser dans le gardien de boucle. Ainsi, une fois que *R_size* serait égal à 0, cela voudrait dire que l'on a parcouru tous les éléments du tableau.

```
Hypothèses   R_array : contient l'adresse du tableau
             R_size : contient la taille du tableau
-------------------------------------------------------
init :       R_max ← 0
             R_currentMax ← 0
             R_inc ← 1
             ; Affectation de l'adresse du tableau :
             R_lastAdr ← R_array

loop :       JMP "end" if R_size == 0

             R_lastVal ← ptr R_lastAdr
             R_currentVal ← ptr R_array
             JMP "nextVal" if R_lastVal != R_currentVal

             R_currentMax ← R_currentMax + R_inc
             JMP "updateMax"

nextVal :    R_currentMax ← 1
             R_lastAdr ← R_array

updateMax :  JMP "nextIter" if R_max > R_currentMax
             R_max ← R_currentMax

nextIter :   R_array ← R_array + R_inc
             R_size ← R_size - R_inc
             JMP "loop"
end :
```

Analysons ce programme étape par étape. Nous initialisons d'abord *R_max* et *R_currentMax* qui servent respectivement à garder en mémoire la taille de la plus longue sous-suite, et de la sous-suite considérée à l'itération actuelle. Comme il est possible que le tableau soit vide, on les initialise à 0. Ensuite, *R_inc* nous sert simplement à pouvoir incrémenter d'autres registres de 1. En effet, notre processeur fictif ne nous permet d'effectuer des additions entre registres uniquement. Finalement, on utilise un dernier registre *R_lastAdr* qui contiendra l'adresse du dernier élément considéré (i.e. que l'on compare à tous les suivants jusqu'à arriver à un élément différent).

Dans la boucle, on doit d'abord se protéger contre les tableaux vides. En effet, si le tableau était vide, on déréférencerait une zone de la mémoire qui ne nous est pas allouée, avec l'opération **ptr**. Nos hypothèses nous permettent cependant de contourner ce cas "limite". En effet, puisque nous disposons de la taille du tableau, nous pouvons attendre de d'abord entrer dans le corps de la boucle avant d'effectuer un quelconque déréférencement. Si le tableau est vide, sa taille est égale à 0, et nous n'entrons jamais dans le corps de la boucle.

```
            ...
loop :      JMP "end" if R_size == 0

            R_lastVal ← ptr R_lastAdr
            R_currentVal ← ptr R_array
            JMP "nextVal" if R_lastVal != R_currentVal

            R_currentMax ← R_currentMax + R_inc
            JMP "updateMax"

nextVal :   R_currentMax ← 1
            R_lastAdr ← R_array
            ...
```

Nous vérifions ensuite que l'élément actuellement considéré à l'adresse *R_lastAdr* soit bien égal à celui de l'adresse *R_array* (nous sommes dans le corps de la boucle donc nous avons la certitude que le tableau contient au moins un élément, et donc que nous pouvons déréférencer). Si c'est le cas, on incrémente *R_currentMax*. Sinon, cela veut dire que la séquence est terminée, et qu'il faut réinitialiser *R_currentMax* et considérer l'élément se trouvant à l'adresse *R_array*.

```
            ...
updateMax :  JMP "nextIter" if R_max > R_currentMax
             R_max ← R_currentMax
             ...
```

Aussi, à chaque itération, il est nécessaire de vérifier si la valeur de *R_currentMax* (i.e. la taille de la séquence considérée) est supérieure à la taille de la plus grande séquence, *R_max* enregistrée. Si c'est le cas, il faut la mettre à jour, et sinon, on passe à l'itération suivante.

```
            ...
nextIter :  R_array ← R_array + R_inc
            R_size ← R_size - R_inc
            JMP "loop"
```

À chaque itération, on passe à l'élément suivant du tableau, en incrémentant donc la valeur de *R_array*, et on diminue *R_size* d'une unité pour symboliser qu'il y a un élément en moins à parcourir. Finalement, on retourne à l'étiquette *"loop"* et on recommence.

Une autre approche pour résoudre ce type de problèmes est de d'abord passer par un pseudo-code intermédiaire, qui représente l'algorithme sans toutes les contraintes du processeur fictif, et d'ensuite effectuer une traduction dans le formalisme demandé. Par exemple, pour ce problème-ci, on pourrait imaginer le pseudo-code suivant :

```
T : adresse du tableau
N : taille du tableau

---

i = 0
max, currentMax = 0
lastAdr = T

while i < N do
   if T[i] == lastAdr[i] then
     currentMax = currentMax + 1
   else then
     currentMax = 1
     lastAdr = T+i

   if currentMax > max then
     max = currentMax

   i = i + 1

return max
```

Il suffit ensuite d'adapter cette approche plus *"user-friendly"* de l'algorithme aux contraintes imposées par l'environnement de développement, et donc dans ce cas-ci, il faut adapter les sauts conditionnels, les sommes de constantes, et la composition de la boucle avec les opérations permises par notre processeur fictif.

(9) 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 déterminer si deux chaînes de caractères terminées par un zéro sont égales.

Les valeurs d'entrée sont les adresses $a1$ et $a2$ des deux chaînes de caractères à comparer.

Ce problème ne présente a priori pas de difficulté conceptuelle particulière. On va profiter de sa simplicité et de notre connaissance des techniques développées dans les sections précédentes pour le résoudre immédiatement de manière intuitive.

On doit tout d'abord considérer l'état de départ de notre système et intialiser les registres dont on aura besoin pour réaliser l'opération demandée dans une section "init". On considère arbitrairement que les deux valeurs d'entrées sont placées dans des registres *R_str*1 et *R_str*2. On doit traverser deux tableaux en même temps, et on a donc besoin d'une boucle ; on initialise un registre *R_inc* qui sera utilisé comme un incrément sur les différentes variables devant progresser à chaque itération de la boucle. Si on a pas de précision explicite à ce sujet, on doit aussi certainement retourner la réponse à la question "les deux chaînes sont-elles identiques ?". On choisit arbitrairement que cette réponse figurera dans un registe *R_eq* utilisé comme un booléen. On l'initialise à 1 pour partir du principe que les deux chaînes sont égales, et on le fixera à 0 dès que l'on trouve une différence dans les chaînes. On pourrait le faire démarrer à 0 et le fixer à 1 uniquement si les deux chaînes sont entièrement égales de manière symétrique.

```
Hypothèses  R_str1 : adresse de la 1e chaine
            R_str2 : adresse de la 2e chaine
-------------------------------------------------
init :      R_inc ← 1
            R_eq ← 1
```

On sait que le parcours des deux chaînes doit s'arrêter dès qu'on atteint la fin d'une des deux chaînes, et dès qu'on a déterminé que les chaînes étaient différentes. On peut donc écrire un gardien de boucle à un intitulé "loop" qui amène à la fin du parcours qu'on appellera "endloop" si la valeur pointée par l'une des deux chaînes est un zéro de terminaison, ou si la valeur de vérité indiquant si les chaînes sont égales est annulée, comme suit :

```
loop :  R_val1 ← ptr R_str1
        R_val2 ← ptr R_str2
        JMP "endloop" if R_val1 == 0
        || R_val2 == 0 || R_eq != 1
```

On remarque qu'on a priori pas d'information explicite dans la définition d'architecture fictive vis-à-vis de la possibilité de faire plusieurs comparaisons d'un coup avec des opérateurs du type "&&" ou "| |". On pourrait prendre le parti de n'appliquer que ce qui est défini verbatim, c'est-à-dire de considérer qu'on ne peut faire qu'une comparaison entre deux registres à la fois. Cela reviendrait juste à étaler les différents tests de la comparaison en plusieurs sauts conditionnels successifs, sans impact sur la logique du programme, comme suit :

```
loop :  R_val1 ← ptr R_str1
        R_val2 ← ptr R_str2
        JMP "endloop" if R_val1 == 0
        JMP "endloop" if R_val2 == 0
        JMP "endloop" if R_eq != 1
```

Le corps de la boucle doit comparer les deux valeurs actuelles des deux chaînes, et indiquer l'inégalité pour mettre fin à la boucle si ces valeurs sont différentes. On peut réaliser cette fonction avec un second saut conditionnel qui évite l'annulation de *R_eq* si et seulement si les deux valeurs des chaînes sont égales. On indique la destination de ce saut par l'intitulé "next" pour signifier qu'on passe à l'étape suivante, après avoir effectué la comparaison :

```
JMP "next" if
R_val1 == R_val2
R_eq ← 0
```

À l'intitulé "next", on doit simplement progresser dans les deux tableaux. On incrémente donc les registres contenant les adresses sur les deux cellules courantes des deux tableaux, et on retourne au début de la boucle pour la prochaine itération ou la fin du parcours :

```
next :  R_str1 ← R_str1 + R_inc
        R_str2 ← R_str2 + R_inc
        JMP "loop"
```

Enfin, on arrive à l'intitulé "endloop" auquel on accède quand le gardien de la boucle n'est plus vérifié. Grâce aux sections précédentes, on sait qu'on arrive ici soit parce qu'on a trouvé une différence et que *R_eq* a été fixé à 0, soit parce qu'on a atteint la fin d'une des deux boucles en trouvant un zéro de terminaison. Dans ce second cas, on doit encore vérifier que la fin d'une chaîne est aussi la fin de l'autre - dans ce cas les chaînes sont bien identiques. Sinon, on doit annuler *R_eq* car les chaînes sont de longueurs différentes et donc par définition différentes. On implémente ce comportement avec un dernier saut conditionnel qui évite l'annulation de la valeur de vérité si et seulement si les deux valeurs sont égaux (et implicitement tous deux des zéros de terminaison) :

```
endloop :  JMP "ret" if
           R_val1 == R_val2
           R_eq ← 0

ret :
```

Au total, on a un programme qui se présente comme suit :

```
Hypothèses  R_str1 : adresse de la 1e chaine
            R_str2 : adresse de la 2e chaine

-----------------------------------------------
init :      R_inc ← 1
            R_eq ← 1

loop :      R_val1 ← ptr R_str1
            R_val2 ← ptr R_str2
            JMP "endloop" if R_val1 == 0
            || R_val2 == 0 || R_eq != 1

            JMP "next" if
            R_val1 == R_val2
            R_eq ← 0

next :      R_str1 ← R_str1 + R_inc
            R_str2 ← R_str2 + R_inc
            JMP "loop"

endloop :   JMP "ret" if
            R_val1 == R_val2
            R_eq ← 0

ret :
```

Cette solution répond à l'énoncé. Il y a cependant matière à discuter de la longueur du programme ; en effet, elle pourrait être considérablement réduite si l'on considère soigneusement les cas de figure dans lesquels les deux chaînes de caractères peuvent présenter des différences.

Les cas dans lesquels on doit mettre fin à la boucle correspondent à tous les cas où il existe au moins une différence dans les deux chaînes, ainsi que l'unique cas où les deux chaînes sont identiques. On remarque qu'en réalité, si les deux chaînes sont de longueurs différentes, elles sont par définitions différentes ; en pratique, l'une des chaînes présentera un zéro de terminaison alors que l'autre présentera une valeur strictement non-nulle. En d'autres mots, le cas de chaînes de

longueurs différentes est un sous-cas de l'ensemble des chaînes différentes et on peut le détecter exactement de la même manière que pour deux caractères "normaux".

En suivant ce raisonnement, on peut donc simplifier le gardien de boucle en une simple comparaison des valeurs des deux chaînes, et omettre la détection de la fin d'une des deux chaînes, puisqu'elle coïncide avec une différence. En fait, le seul cas où l'on voudra détecter la fin d'une des chaînes correspond au cas où les deux éléments sont égaux **et** sont des zéros de terminaisons. Dans ce cas et seulement dans celui-ci, on a donc atteint l'unique configuration où les deux chaînes sont parfaitement identiques.

On peut donc simplifier le gardien et le corps de la boucle de telle manière à obtenir le programme suivant :

```
Hypothèses  R_str1 : adresse de la 1e chaine
            R_str2 : adresse de la 2e chaine

-------------------------------------------------
init :      R_inc ← 1
            R_eq ← 1

loop :      R_val1 ← ptr R_str1
            R_val2 ← ptr R_str2
            JMP "diff" if R_val1 != R_val2

            JMP "ret" if R_val1 == 0

            R_str1 ← R_str1 + R_inc
            R_str2 ← R_str2 + R_inc
            JMP "loop"
diff :      R_eq = 0

ret :
```

Dans cette version plus optimisée du programme, on ne vérifie si on a atteint le zéro de terminaison qu'après avoir confirmé que les chaînes étaient toujours parfaitement égales - cette condition ne sera donc vérifiée que si les chaînes sont parfaitement identiques, à la fin du parcours. Dans tous les autres cas, on détectera une différence avant et on pourra sauter au nouvel intitulé "diff" qui annule la valeur de vérité signalant l'inégalité des chaînes.

On récapitule l'inventaire des variables comme suit :

— *R_str*1 et *R_str*2 contiennent l'adresse des deux chaînes à comparer. On incrément au fur et a mesure du parcours ces adresses afin de pointer sur les caractères successifs des deux chaînes.
— *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_eq* est manipulé comme un booléen correspondant à une valeur de vérité associée à la question "les chaînes sont-elles identiques ?". On part du principe qu'elles le sont en l'initialisant à 1, et on l'annule dans tous les cas où on détecte la moindre différence.
— *R_val*1 et *R_val*2 sont utilisés comme des registre de travail temporaires pour contenir les valeurs issues des deux chaînes de caractère afin de pouvoir calculer des comparaisons sur base de ces valeurs, parce qu'on a pas d'opération de comparaison acceptant en opérande des registres pointeurs dans notre architecture fictive.

(4) 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 calculer octet par octet la somme de deux nombres enregistrés en mémoire selon la convention petit-boutiste sur n octets chacun. On choisit arbitrairement de respecter la convention habituelle de l'opération d'addition en assembleur qui indique de placer le résultat dans l'opérande de gauche. Comme on a pas d'information sur les nombres, on suppose également qu'ils sont encodés en format non-signé.

Les valeurs d'entrée sont les adresses $a1$ et $a2$ des nombres à sommer et leur taille n.

La subtilité de cet exercice consiste à devoir maîtriser le mécanisme de report lors d'un dépassement arithmétique. Dans une architecture réelle, un registre spécial appellé "drapeau" sert à signaler lorsqu'une opération d'addition de deux opérandes de taille k mène à un dépassement arithmétique par rapport au format de k bits.

Par propriété de l'encodage binaire, la somme de deux registres de taille k peut mener au maximum à un dépassement d'une valeur égale à la puissance associé au $(k+1)$ème bit, 2^k. Par exemple, la somme de deux registres d'un octet (donc $k = 8$) qui contiendront chacun au maximum $2^8 - 1 = 255$ vaudra au plus $2 * 255 = 2^9 - 2 = 510$ qui est entièrement représentable sur $k + 1$ bits tel que le $(k + 1)$ième bit est fixé à 1 pour indiquer la présence de la puissance 2^8, et le reste du nombre $(2^9 - 2) - 2^8 = 510 - 256 = 254$ est entièrement représentable sur les k bits de départ.

Ici, on a a priori pas défini que notre architecture fictive comprenait des drapeaux et donc on doit être créatif afin de gérer les reports. Puisqu'on a pas accès aux mécanismes réels qui permettent de détecter un dépassement, on va comparer le résultat de la somme sur un octet aux opérandes de départ. En effet, si le résultat est plus petit que l'une de ces opérandes, c'est qu'il y a eu dépassement arithmétique et que le résultat que l'on lit sur un octet est en fait la version tronquée du résultat non-représentable sur un octet ; c'est une valeur plus grande que la capacité totale d'un format d'un octet, c'est-à-dire plus grande que la valeur 0xFF. Dans ce cas, c'est qu'il y a dépassement et on doit donc reporter le terme de puissance 2^k sur l'octet suivant. Puisqu'on considère une représentation d'un nombre faisant au total n octets de long, le report de la valeur de 2^k correspond à un ajout de 1 sur l'octet de poids supérieur. Par exemple, un report de 2^8 du premier octet est encodé comme 1 sur le second octet, puisque ce dernier indique les puissances de 2^8 à $2^{16-1} = 2^{15}$.

Il faut encore considérer le cas des sommes qui peuvent mener à un report à cause du report de l'octet de poids inférieur. En effet, si la somme des deux octets d'un poids donné est strictement supérieure aux opérandes de la somme, mais que le résultat de cette somme vaut lui-même $2^8 = 255$, on peut encore avoir un report à l'octet de poids supérieur à cause du report venant de l'octet de poids inférieur qui s'ajoute à 255 pour faire un total de 256 et qui doit donc être encodé comme un terme 1 à l'octet supérieur et un terme 0 à l'octet courant.

Pour considérer tous les cas, on peut procéder comme suit :

```
JMP "overflow" if R_sum < R_val
JMP "no_overflow" if R_sum ≠ R_full
JMP "no_overflow" if R_carry ≠ R_inc
```

En appliquant les différentes clauses de comparaison dans cet ordre, on se protège de tous les cas limites (comme une ou deux opérandes nulles) de la manière la plus simple. En suivant le flot d'exécution du code assembleur qui correspondant à cette séquence d'instructions JMP successives, on a en fait une situation qui revient à faire les tests suivants dans une syntaxe proche du C :

```
if( A+B % 256 < B )
  goto "overflow"
else
{
  if( A+B ≠ 255 )
    goto "no_overflow"
  else
  {
    if( R_carry ≠ 1 )
      goto "no_overflow"
  }
}
```

En suivant ces clauses, on n'arrive bien à l'instruction suivant les JMP (ou les if en syntaxe C) sans sauter que dans le cas où le résultat est plus grand que les opérandes, qu'il est égal à 255 et qu'il y a un report à comptabiliser de l'octet de poids inférieur. Cette intruction sera donc le début de la section correspondant au cas où il y a un report à propager sur l'octet de poids supérieur.

Enfin, puisqu'on est dans une architecture petit-boutiste, les octets sont arrangés du poids faible au poids fort dans le sens croissant des adresses. Dès lors, on peut effectuer l'addition octet par octet en incrémentant progressivement les adresses ds deux tableaux.

On peut donc implémenter le fonctionnement développé ci-dessus comme suit :

```
Hypothèses      R_nb1 : contient l'adresse de la première opérande
                R_nb2 : contient l'adresse de la deuxième opérande
                R_fin : contient la taille des opérandes

init :          R_inc ← 1
                R_full ← 255
                R_sum ← 0
                R_carry ← 0
                R_fin ← R_fin + R_nb1 ; R_fin contient (a1+L)

loop :          JMP "end" if R_nb1 ≥ R_fin

                R_sum ← ptr R_nb1
                R_val ← ptr R_nb2
                R_sum ← R_sum + R_val

                JMP "overflow" if R_sum < R_val
                JMP "no_overflow" if R_sum ≠ R_full
                JMP "no_overflow" if R_carry ≠ R_inc

overflow :      R_sum ← R_sum + R_carry
                R_carry ← 1
                JMP "next"

no_overflow :   R_sum ← R_sum + R_carry
                R_carry ← 0

next :          ptr R_nbr1 ← R_sum
                R_nb1 ← R_inc + R_nb1
                R_nb2 ← R_inc + R_nb2
                JMP "loop"
end :
```

A l'intitulé "init", on commence par initialiser les registres de travail *R_inc*, *R_full*, *R_sum* et *R_carry*. Le premier contiendra comme à l'habitude l'incrément pour faire avancer la boucle ; le second contiendra la constante 255 utilisée pour tester si le résultat de la somme remplit entièrement un octet et menace de déborder. Le troisième registre est utilisé pour effectuer la somme des deux octets courants et éventuellement du report venant de l'octet de poids inférieur s'il existe. Le quatrième registre indique la présence d'un report et est en essence manipulé comme un drapeau booléen associé au dépassement arithmétique lors de la somme. Enfin, on manipule *R_fin* pour contenir l'adresse de la cellule après la fin du premier nombre afin de l'utiliser dans le gardien de boucle.

Au début de la boucle, à l'intitulé "loop", on vérifie d'abord si le pointeur sur la valeur actuelle n'a pas dépassé le tableau. Si c'est le cas, c'est qu'on a fini le calcul, et on peut sauter au label "end" signalant la fin du programme. On note que si les tableaux fournis sont indiqués comme de longueur nulle, c'est-à-dire s'ils sont vides, on saute immédiatement à la fin du programme sans faire de calcul. Si on est dans un cas où on a pas encore dépassé la longueur des tableaux, on entre dans le calcul de l'octet courant.

On effectue ensuite le calcul en insérant à chaque fois la valeur adressée par les pointeurs sur les deux nombres dans un registre temporaire *R_val*, puisqu'on a pas d'opération de somme pouvant directement prendre en argument des pointeurs. On calcule la somme des deux opérandes dans le registre *R_sum*, puis on teste la valeur du résultat en la comparant successivement à la valeur d'une des opérandes puis à la valeur 255. On compare la somme à la valeur de l'opérande de droite encore contenue *R_val* pour la simple raison qu'elle y réside toujours. Enfin, si les clauses précédentes n'ont pas été vérifiées, on teste la présence d'un report de l'octet de poids inférieur afin de diriger le programme vers la section associée à la présence d'un report si la somme menace de déborder du support d'un octet.

Aux intitulés "overflow" et "no_overflow", on comptabilise le report éventuel dans la somme des deux termes courants, puis on fixe la valeur du registre associé à ce report à la valeur appropriée pour comptabiliser ou non le report sur l'octet de poids supérieur. On saute à l'intitulé "next" si nécéssaire, ou on passe simplement à celui-ci s'il s'agit de l'instruction suivante.

À l'intitulé "next", on met le résultat de la somme des octets courants à l'adresse de l'octet courant de la première opérande , puis on incrémente simplement les deux pointeurs pour adresser les cellules suivantes des deux nombres et on saute au gardien de boucle pour la prochaine itération ou la fin du programme.

On récapitule l'inventaire des variables comme suit :

— *R_nb*1 et *R_nb*2 contiennent l'adresse des deux nombres à sommer.On incrémente progressivement ces registre afin de pointer sur les octets successives des deux nombres.
— R_fin contient la taille annoncée des deux nombres, puis on le manipule pour contenir l'adresse de la cellule après le premier nombre et on l'utilise dans le gardien de boucle.
— *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_full est utilisé pour contenir la constante 255 employée dans les tests pour identifier un éventuel report.
— *R_sum* est utilisé pour contenir la somme des deux octets courants et éventuellement du report de l'octet de poids inférieur.
— *R_carry* est utilisé comme un drapeau booléen indiquant un report venant de l'octet de poids inférieur ou à propager à l'octet de poids supérieur.
— *R_val* est utilisé comme un registre de travail temporaire pour contenir les octets de la seconde opérande, parce qu'on a pas d'opération de somme acceptant en opérande des registres pointeurs dans notre architecture fictive.