Chapitre 8

Série d'exercices n°8 - Théorie chapitre IV

8.1 Mode d'emploi

— **Convention d'appel :**

- Les six premiers arguments éventuels de l'appel sont placés dans les registres respectifs RDI, RSI, RDX, RCX, R8 et R9. S'ils sont encodés sur 8, 16 ou 32 bits, ils occuperont les parties de poids faible correspondantes de ces registres.
- Si l'appel possède des arguments supplémentaires, alors ceux-ci sont placés sur la pile, en tant que valeurs représentées sur 64 bits. Ces arguments sont empilés dans l'ordre inverse où ils apparaissent dans l'appel de la fonction. Ils sont empilés avant que l'instruction `CALL` n'empile l'adresse de retour.
- La valeur de retour d'une fonction est placée dans le registre RAX, ou dans une partie de poids faible de ce registre si cette valeur est encodée sur 8, 16 ou 32 bits.
- Si une fonction modifie les registres RBX, RBP, R12, R13, R14 ou R15, alors elle doit se charger d'en sauvegarder la valeur, et de la restaurer avant de terminer son exécution.
- La fonction doit maintenir le pointeur de pile RSP à une adresse multiple de 16.

8.2 Correctifs des exercices

8.2.1 Exercice 1

Énoncé : Pour chacun des exercices de la section 3.5.2, traduire votre solution en un programme assembleur x86-64 complet, accompagné si nécessaire d'un programme C permettant de le tester.

Solution :

(1) On repart de la solution établie dans les séries d'exercices précédents en rappellant l'énoncé auquel le programme rédigé doit répondre : écrire une valeur constante donnée dans tous les emplacements d'un tableau d'octets, donné par son adresse et sa taille.

```
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" );
```

La logique du programme étant déjà donnée à travers le pseudo-code, il suffit de le traduire en langage Assembleur x86-64.

```
            .intel_syntax noprefix
            .text
            .global fill_tab
            .type fill_tab, @function
fill_tab:
            # RDI : adr. tableau (char*)
            # RSI : taille tableau (int)
            # RDX : valeur constante (char)
            MOV R8, 0
iterate:
            CMP R8D, ESI
            JGE end
            MOV byte ptr[RDI + R8], DL
            INC R8
            JMP iterate
end:
            RET
```

Tout d'abord, il faut d'abord spécifier que le programme est rédigé dans la variante syntaxique *Intel* du langage assembleur x86-64 [1]. Pour cette variante, l'option NOPREFIX permet d'écrire les noms de registres et les constantes numériques directement sans devoir les précéder d'un symbole spécial. Ensuite, la directive .TEXT signale le début d'un segment de code, ce qui signifie que la suite du programme contient des instructions exécutables placées dans la mémoire du programme. Finalement, la directive .GLOBAL FILL_TAB indique que l'on peut faire référence à l'étiquette FILL_TAB en dehors du programme, et la dernière directive indique qu'il s'agit d'une fonction.

Après avoir spécifié le type du programme réalisé, comme par convention les trois premiers arguments éventuels utilisés par une fonction en assembleur sont *RDI, RSI et RDX*, nous choisissons que *RDI* contient l'adresse du tableau de caractères (l'adresse, étant un pointeur, occupe donc les 64 bits du registre), *RSI* contient la taille du tableau (de type int, qui occupe donc 32 bits et donc la partie *ESI* du registre *RSI* uniquement), et *RDX* la valeur constant que l'on veut écrire à chaque case du tableau (la valeur constante, étant de type char occupe 1 Byte, donc la partie *DL* du registre *RDX*).

En effet, il est possible d'utiliser certaines parties spécifiques d'un registre, selon le nombre de bits nécessaires [2]. Par exemple, on peut soit choisir d'utiliser les 64 bits du registre *RDX* (et donc l'utiliser en entier), soit ses 32 bits de poids faible, ce que l'on note par *EDX*, soit ses 16 bits de poids faible (*DX*), soit le dernier octet de poids faible (*DL*), ou bien encore l'avant-dernier octet de poids faible (*DH*, les bits d'indices 8 à 15).

```
MOV R8, 0
```

Au lieu de manipuler l'adresse du tableau directement pour s'y déplacer, comme on le fait dans la solution rédigée en pseudo-code, il peut être intéressant d'utiliser plutôt un registre comme indice à incrémenter à chaque itération et à ajouter à l'adresse du tableau pour indexer une valeur donnée de celui-ci. C'est donc le rôle de *R8*, que nous initialisons à 0, et qui sera comparé à la taille du tableau à chaque itération puisqu'il évoluera pendant le parcours du tableau jusqu'à atteindre une valeur égale à sa longueur lorsqu'il pointera sur la première cellule de mémoire juste après le tableau.

```
iterate:
          CMP R8D, ESI
          JGE end
          MOV byte ptr[RDI + R8], DL
          INC R8
          JMP iterate
```

1. C'est la variante que nous avons choisie d'utiliser dans le cadre de ce cours.
2. cf. Partie 3.4.1 du cours théorique

La prochaine étape consiste à se déplacer dans tout le tableau et à y écrire la valeur contenue dans la partie *DL* de *RDX*. Pour ce faire, on compare d'abord le contenu de *R8* à la taille du tableau contenue dans *RSI*. Mais comme la taille du tableau est un *int* et qu'il ne faut utiliser que les 32 bits de poids faible pour la comparaison, il faut comparer la partie *R8D* de *R8* et la partie *ESI* de *RSI*.

On compare donc l'indice *R8D* à la taille pour vérifier si tout le tableau n'a pas déjà été parcouru. Si *R8D* est plus grand ou égal à *ESI*, il faut faire un jump à la fin du programme et renvoyer écrire l'instruction RET pour terminer l'exécution de la fonction [3]. C'est ce qui est respectivement symbolisé par les instructions :

```
CMP R8D, ESI
JGE end
```

Finalement, à chaque tour de boucle, il reste à écrire la valeur contenue dans *DL* (i.e. les 8 bits de poids faible de *RDX*) à l'adresse actuelle du tableau (i.e. la "*R8*-ième" case), et puis incrémenter le compteur, et finalement effectuer un jump inconditionnel au gardien de la boucle. On emploie donc un adressage mémoire indirect indexé tel que l'adresse à laquelle on doit écrire correspond à la somme des valeurs contenues dans *RDI* et *R8*. L'opération d'incrément en assembleur est bien plus immédiate que l'ensemble de directives employées en pseudo-code, puisqu'on s'abstrait des contraintes arbitraires de l'architecture fictive qui ne possédait pas d'instruction INC.

Note : La partie "difficile" de ce type d'exercices est la partie algorithmique. Une fois que l'on a une idée exacte de ce que l'on souhaite faire pour résoudre un problème, il suffit de l'appliquer en suivant le formalisme et la syntaxe de votre choix. Que ce soit du pseudo-code, de l'assembleur x86-64 ou du langage C, les contraintes techniques ne sont pas toujours aussi simples selon le niveau du langage utilisé, mais la logique de l'algorithme de résolution reste identique.

3. Lorsque le programme arrive à l'instruction RET, la convention d'appel veut que la valeur contenue dans *RAX* soit l'argument de sortie de la fonction. Or, dans ce cas-ci, la fonction ne renvoie rien, et la valeur contenue dans *RAX* n'a donc pas d'importance

(2) On repart de la solution établie dans les séries d'exercices précédents en rappellant l'énoncé auquel le programme rédigé doit répondre : recopier un tableau d'octets, donné par son adresse et sa taille, vers une autre adresse donnée de la mémoire, sachant qu'il est possible que les zones de mémoire associées au tableau initial et à sa copie se recouvrent.

```
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" );
```

Une traduction possible de cette solution en Assembleur x86-64 est :

```
                .intel_syntax noprefix
                .text
                .global copy_tab
                .type copy_tab, @function
copy_tab:
                # RDI : adr. tableau source (pointeur, 64b)
                # RSI : taille tableau source (int, 32b)
                # RDX : adr. tableau destination (pointeur)
                CMP RDI, RDX
                JL copy_dec

copy_inf:
                MOV R8, 0
copy_inc_loop:
                CMP R8D, ESI
                JGE end

                MOV R9B, byte ptr[RDI+R8]
                MOV byte ptr [RDX + R8], R9B

                INC R8
                JMP copy_inc_loop

copy_dec:
                MOV R8, 0
                MOV R8D, ESI
                DEC R8
copy_dec_loop:
                CMP R8, 0
                JLE end

                MOV R9B, byte ptr[RDI+R8]
                MOV byte ptr[RDX + R8], R9B

                DEC R8
                JMP copy_dec_loop
end:
                RET
```

Tout d'abord, les spécifications du programme sont les mêmes que dans l'exercice précédent, où seul le nom de la fonction change. Aussi, les trois premiers registres utilisés par convention pour contenir les paramètres d'une fonction sont *RDI, RSI* et *RDX*. Comme *RDI* et *RDX*

contiennent les adresses respectives du tableau source et du tableau de destination, les 64 bits des deux registres sont utilisés, et on les note donc bien comme *RDI* et *RDX*. Le registre *RSI*, quant à lui, contient la taille du tableau source, déclarée sur 32 bits. On exploitera donc ses 32 bits de poids faible en faisant référence à sa sous-partie *ESI*.

Ensuite, le pseudo-code propose une résolution où il n'y a qu'une seule boucle. C'est une bonne idée, dans la mesure où la plupart des instructions sont similaires dans les deux cas envisageables. Cependant, dans la solution proposée ci-après, nous allons le faire de façon plus élémentaire en séparant les deux cas possibles : soit le tableau source (*RDI*) se trouve à une adresse inférieure à celle du tableau de destination (*RDX*), soit il se trouve à une adresse plus grande ou égale. Il faut aussi tenir compte du fait qu'il est possible que les deux tableaux se chevauchent [4] ; en effet, *RDI* et *RDX* pourraient contenir des cases en commun, et c'est pour éviter d'écraser des valeurs à recopier dans *RDX* qu'il est indispensable de tenir compte de ces deux cas.

```
CMP RDI, RDX
JL copy_dec
```

C'est ce choix qui est effectué par ces deux instructions. Il y a un saut conditionnel *JL* (Jump if Less) vers l'étiquette *copy_dec* si l'adresse de *RDI* est inférieure à celle de *RDX*. Dans ce cas, il faut commencer par recopier les cases qui se trouvent à la fin de *RDI* en premier :

```
copy_dec:
                MOV R8, 0
                MOV R8D, ESI
                DEC R8
copy_dec_loop:
                CMP R8, 0
                JLE end

                MOV R9B, byte ptr[RDI+R8]
                MOV byte ptr[RDX + R8], R9B

                DEC R8
                JMP copy_dec_loop
```

On commence par initialiser les 64 bits du compteur *R8* à 0 avant d'y placer la taille du tableau source *ESI*. En effet, comme *ESI* contient une valeur définie sur 32 bits, si on se contentait de déplacer ces 32 bits dans *R8D*, il pourrait être dangereux de comparer *R8* en entier à 0 car ce qui

4. Représenter la situation sous la forme d'un dessin schématique peut être utile à la compréhension de cette logique. Elle est aussi expliquée dans la résolution en pseudo-code de cet exercice.

se trouve dans ses 32 bits de poids fort est inconnu. On décrémente ensuite cette valeur d'une unité, puisque l'indexage des tableaux commence par 0, et que l'on veut accéder à la dernière case du tableau *RDI* dont l'indice correspond bien à la longueur du tableau - 1.

Ensuite, il faut itérer un nombre de fois égal *ESI*. Pour cela, il suffit de décrémenter *R8* à chaque tour de boucle, et d'effectuer un *JLE* (Jump if Less or Equal) à la fin du programme lorsque *R8* arrive à 0.

Finalement, à chaque itération, il faut d'abord placer le contenu de la case actuelle de *RDI* dans un registre intermédiaire avant de la placer à la case correspondante de *RDX*. Pour ce faire, on utilise deux fois l'adressage mémoire indirect indexé en appliquant l'index contenu dans *R8* aux registres *RDI* puis *RDX* pour successivement lire la valeur du tableau d'origine puis écrire dans le tableau de destination.

```
MOV R9B, byte ptr[RDI+R8]
MOV byte ptr[RDX + R8], R9B
```

En effet, il est impossible d'effectuer une instruction *MOV* entre deux valeurs pointées. C'est une instruction qui peut être disponible sous d'autres architectures, mais qui ne l'est physiquement pas en Assembleur x86-64, ce qui nous oblige à passer par le registre intermédiaire *R9* (et plus spécifiquement, son dernier octet indiqué par *R9B*).

Finalement, il ne reste plus qu'à décrémenter la valeur de *R8*, et d'effectuer un saut inconditionnel vers l'étiquette qui correspond au gardien de la boucle, *copy_dec_loop*.

L'autre situation où *RDI* se trouve à une adresse supérieure ou égale à celle de *RDX* est traitée exactement de la même façon, sauf qu'il faut parcourir *RDI* en faisant varier l'indice *R8* de 0 jusque *ESI*-1 :

```
copy_inf:
                MOV R8, 0
copy_inc_loop:
                CMP R8D, ESI
                JGE end

                MOV R9B, byte ptr[RDI+R8]
                MOV byte ptr [RDX + R8], R9B

                INC R8
                JMP copy_inc_loop
```

La dernière instruction du programme, *RET*, consiste à nouveau à clôturer l'exécution de la fonction.

(3) On repart de la solution établie dans les séries d'exercices précédents en rappellant l'énoncé auquel le programme rédigé doit répondre : retourner un tableau d'octets, donné par son adresse et sa taille, c'est-à-dire en permuterles 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.

```
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" );
```

Une traduction possible de cette solution en Assembleur x86-64 est :

```
            .intel_syntax noprefix
            .text
            .global swap_tab
            .type swap_tab, @function

swap_tab:
            # RDI : adr tableau
            # RSI : taille tableau (int)
            MOV R8, 0
            MOV R9, 0
            MOV R9D, ESI
            DEC R9
```

```
loop_tab:
            CMP R9, R8
            JLE end

            MOV R10B, word ptr[RDI + R8]
            MOV R11B, word ptr[RDI + R9]

            MOV word ptr[RDI + R9], R10B
            MOV word ptr[RDI + R8], R11B

            INC R8
            DEC R9
            JMP loop_tab
end:
            RET
```

Les spécifications du programme restent les mêmes que celles des deux exercices précédents. Le programme prend en entrée deux arguments : l'adresse du tableau, et sa taille (déclarée sur un *int*). Par convention, les deux registres utilisés seront *RDI* et *RSI* (et plus spécifiquement *ESI* puisque la taille du tableau est contenue sur 32 bits).

Pour retourner le tableau d'octets, il nous faut deux registres qui nous serviront de compteurs. On choisit arbitrairement *R8*, que l'on initialise à 0, et qui nous permettra de considérer les éléments en partant du début du tableau, ainsi que *R9* qui sera initialisé à $ESI-1$ (donc d'abord à la valeur de *ESI* avant d'être décrémenté une fois) étant donné que l'indinçage des éléments commence par 0, et que l'on veut que *R9* fasse référence au dernier élément du tableau. Il ne faut pas oublier de d'abord l'initialiser à 0, car nous ne savons pas ce qui se trouve dans les 32 bits de poids fort de ce registre. En effet, si nous comparions *R8* et *R9* sans avoir formaté les 32 premiers bits de R9, il se pourrait qu'il y ait un résidu d'ancienne information contenue à cet endroit-là qui viendrait fausser le résultat de la comparaison.

L'idée de l'algorithme est la suivante : il faut permuter l'élément du tableau pointé par *RDI+R8* avec celui pointé par *RDI+R9* à chaque itération puis incrémenter *R8* et décrémenter *R9*. Lorsque *R8* et *R9* seront égaux ou que *R9* sera devenu plus petit que *R8* selon si les tableaux sont de longueurs impaires ou paires (comme discuté lors de la rédaction de la solution en pseudo-code), cela voudra dire que le premier élément aura pris la place du dernier, et vice-versa, le second la place de l'avant-dernier, et vice-verse, et ainsi de suite, comme demandé dans l'énoncé. C'est exactement ce qui se produit dans cet extrait de code :

```
loop_tab:
            CMP R9, R8
            JLE end

            MOV R10B, word ptr[RDI + R8]
            MOV R11B, word ptr[RDI + R9]

            MOV word ptr[RDI + R9], R10B
            MOV word ptr[RDI + R8], R11B

            INC R8
            DEC R9
            JMP loop_tab
```

On compare *R9* et *R8* et on effectue un *JLE* (Jump if Less or Equal, *R9*≤*R8*) à la fin du programme si la condition est validée.

Ensuite, on passe par des registres temporaires que l'on choisit arbitrairement : *R10* et *R11*. Étant donné que l'on manipule des octets, nous n'avons besoin que du dernier octet de ces registres, donc on spécifie *R10B* et *R11B*. Ces derniers servent à permuter la valeur contenue dans la "*R8*-ième" case du tableau avec la "*R9*-ième". Pour toutes les opérations de déplacement, on emploie l'adressage mémoire indirect indexé en sommant les contenus des registres *RDI* et *R8* ou *R9* pour pointer sur les cellules du tableau dont les valeurs doivent être échangées. Et une fois la permutation effectuée, on incrémente *R8* et on décrémente *R9* avant d'effectuer un saut inconditionnel au gardien de la boucle pour exécuter la prochaine itération ou arriver à la fin du programme.

Finalement, l'instruction *RET* indique que la fonction est terminée.