Chapitre 9

Série d'exercices n°9 - Théorie chapitre IV

9.1 Mode d'emploi

— **Gestion des contextes avec `RBP` :**

Les données relatives à un appel de fonction qui sont placées sur la pile sont organisées selon une *structure de pile (stack frame)*. Chaque appel de fonction place une telle structure sur la pile, et chaque retour de fonction retire de la pile la structure correspondant à l'appel qui se termine. Le contenu d'une structure de pile est indexé à partir de sa *base*, qui est représentée par le contenu du registre [1] RBP.

Les instructions suivantes à placer au début et à la fin de tout programme :

```
PUSH RBP
MOV RBP, RSP

(...)

POP RBP
```

créent la structure de pile, en empilant la base de la structure précédente et en situant la base de la structure courante, puis la détruisent et restaurent le contexte d'origine en dépilant la base de structure précédemment empilée.

1. Il s'agit de l'extension à 64 bits du registre BP provenant de l'architecture x86, dont le nom signifie *Base Pointer*.

9.2 Correctifs des exercices

9.2.1 Exercice 4.4.8

Énoncé :

(Examen de première session, 2019) Dans le cadre du développement d'une application de calcul statistique, on souhaite programmer une fonction `histogramme` capable de calculer l'histogramme d'un tableau d'octets donné. Cette fonction prend pour entrée

— l'adresse d'un tableau d'octets `ts`,
— la taille `n` de ce tableau, exprimée sur 64 bits,
— l'adresse d'un tableau `th` contenant 256 entiers de 64 bits, non initialisés.

À l'issue de l'exécution de cette fonction, chaque case de `th` doit contenir le nombre d'octets de `ts` égaux à son indice. Par exemple, la cinquième case `th[4]` de `th` contiendra le nombre d'octets de `ts` égaux à 4. La fonction ne retourne rien.

— Écrire, en pseudocode ou en langage C (au choix), un algorithme permettant de résoudre ce problème.
— Traduire cet algorithme en assembleur x86-64, en veillant à respecter la convention d'appel de fonctions des systèmes *Unix*.

Solution :

Pour comprendre cet énoncé, on détaille d'abord les instructions qui nous sont données. On nous demande en essence de réaliser un histogramme, c'est-à-dire un objet d'analyse statistique correspondant à un ensemble de boîtes. Chacune de ces boîtes est associée à un intervalle de valeurs, et contient le nombre de valeurs de cet intervalle présent dans les données dont est tiré l'histogramme. En général, on arrange ces boîtes selon l'ordre numérique croissant des intervalles de valeurs auxquelles elles sont associées afin d'en faciliter l'interprétation.

Par exemple, on pourrait construire un histogramme extrêmement simple fait de trois boîtes telles que la première contient le nombre de valeurs des données vivant sur l'intervalle [0,10[, la seconde le nombre de valeurs des données vivant sur l'intervalle [10,20[, et la troisième le nombre de valeurs des données vivant sur l'intervalle [20,30[. Si les données consistent en un tableau = [0, 1, 11, 12, 13, 21], on aura donc les valeurs 2, 3 et 1 dans la première, seconde et troisième boîte respectivement, ce qui correspond simplement au compte du nombre de valeurs présentes dans les données correspondant aux intervalles associés aux boîtes de l'histogramme.

Dans le cas de cet exercice, on définit un histogramme par un tableau `th` a 256 valeurs, donc a 256 boîtes. Sachant que les données à analyser sont elles-mêmes représentées par un tableau `ts` d'octets, c'est-à-dire de valeurs représentables 8 bits, les valeurs à considérer seront contraintes sur l'intervalle $[0,2^8\text{-}1] = [0,255]$. Dès lors, on en déduit immédiatement que chaque boîte de l'histogramme ne peut être définie de manière univoque que si chacune de ses 256

boîtes est associée à un intervalle d'une unique valeur parmi les 256 valeurs accessibles aux données. Autrement dit, la première boîte correspond à compter le nombre de valeurs égales à 0, la seconde le nombre de valeurs égales à 1, etc. Il n'y a aucune ambiguité sur l'ordre dans laquelle ces boîtes sont agencées, puisque dans ce cas où une boîte correspond à une valeur, l'indice de chaque boîte encode immédiatement cette valeur, ce qui permet une interpretation immédiate de l'histogramme - c'est la manière traditionnelle de concevoir des histogrammes de données à valeurs discrètes.

On remarque que c'est en essence ce que nous indique l'énoncé, lorsqu'il nous demande de placer dans chaque case de `th` le nombre d'octets de `ts` égal à son indice : il nous confirme que l'on doit réaliser un histogramme de 256 boîtes pour lequel l'indice de chaque boîte encode la valeur discrète présente dans les données dont chaque boîte doit contenir le compte.

On remarque également que l'histogramme est défini tel que ses valeurs correspondant au compte des données associées à chaque boîte sont représentées sur des entiers longs de 64 bits. De cette manière, on s'assure qu'on peut analyser des données potentiellement extrêmements grandes, telles qu'elles peuvent contenir jusque $2^64 - 1$ fois chacune des valeurs sur l'intervalle [0;255[. De manière correspondante, la taille du tableau `ts` est exprimée sur 64 bits pour lui permettre d'être aussi long que possible dans notre architecture x86_64.

On implémente d'abord la logique du problème en langage C en parcourant l'énoncé point par point.

```
void histogramme(char* ts, unsigned long int size, long int* th)
{
}
```

Cette déclaration correspond aux indications de l'énoncé : on a une fonction `histogramme` qui prend en entrée l'adresse du tableau d'octets `ts` (donc en C de type char, par exemple), la taille `n` de ce tableau exprimée sur 64 bits (donc en C de type unsigned long int, par exemple) et l'adresse d'un second tableau `th` d'entiers exprimés sur 64 bits (donc en C de type long int, par exemple) dont la longueur de 256 bits est connue et n'a donc pas besoin d'être passée en argument.

```
void histogramme(char* ts, unsigned long int size, long int* th)
{
  for( int i = 0 ; i < 256 ; i++ )
    th[i] = 0 ;
}
```

Comme on nous indique que le tableau d'histogramme `th` est a priori non initialisé, il faut d'abord le nettoyer de toute valeur résiduelle arbitraire. De plus, on sait qu'on peut déjà le remplir de 0 pour permettre ensuite de compter les données de `ts` de manière incrémentale. Cette simple boucle réalise cette opération en se basant sur la longueur connue de 256 éléments du tableau `th`.

```
void histogramme(char* ts, unsigned long int size, long int* th)
{
  for( int i = 0 ; i < 256 ; i++ )
    th[i] = 0 ;

  for( long int i = 0 ; i < n ; i++ )
    th[ts[i]]++ ;
}
```

Cette seconde boucle implémente le fonctionnement décrit dans l'énoncé : on doit placer dans chaque case de `th` le nombre de valeurs de `ts` égales à l'indice de la case de `th`. On le fait ici en un seul parcours de `ts`, de manière incrémentale, en se basant sur le fait que l'histogramme a été préalablement initialisé à 0. A chaque valeur `ts[i]` que l'on lit successivement, on sait qu'il y a une place dans l'histogramme `th` : cette place est l'élément de `th` dont l'indice est égal à la valeur de `ts`, autrement dit l'élément `th[ts[i]]`. Le problème serait en réalité plus compliqué si chaque boîte de l'histogramme ne correspondait pas exactement à une des valeurs discrètes accessibles au tableau `th`, mais c'est bien le cas ici.

On traduit ensuite ce programme rédigé en langage C en instructions assembleur, comme suit :

```
histogramme:
              # RDI : contient l'adresse de ts
              # RSI : contient la longueur n de ts
              # RDX : contient l'adresse de th
```

On sait par la convention d'appel que les arguments d'entrée du programme assembleur se trouveront dans l'ordre dans les registres `RDI`, `RSI` et `RDX`.

```
          MOV R12, 0 # R12 contiendra l'index dans th

loop_th:
          CMP R12, 0xFF
          JG init_ts

          MOV qword ptr[RDX + 8*R12], 0

          INC R12
          JMP loop_th
```

On réalise la première boucle qui nettoie et initialise à 0 chaque case de l'histogramme. Pour ce faire, on emploie un indice qu'on place dans un registre arbitraire `R12` qu'on compare à la

longueur connue du tableau th de 256 bits - on compare l'indice R12 à la constante hexadécimale 0xFF = 255, et on saute à la prochaine section qui sera la boucle sur le tableau ts lorsque l'indice dépasse cette valeur de 255, soit lorsqu'il atteint 256. On remarque que l'initialisation à 0 implique un MOV qualifié par un qword. En effet, les valeurs du tableau th sont des entiers de 64 bits prenant donc chacune 8 octets. De manière correspondante, le déplacement d'une case à l'autre sur base de l'indice contenu dans R12 doit se faire de 8 en 8 ; la n-ième case du tableau se trouve à l'octet d'indice 8*n par rapport à l'adresse de la première case. On doit également préalablement initialiser le registre d'indice R12 à 0 pour pointer sur la première case du tableau.

```
init_ts:
          MOV R12, 0 # On recycle R12 pour contenir l'indice dans ts
          MOV R13, 0 # R13 contiendra la valeur actuelle de ts
          # R14 contiendra la valeur actuelle de th

loop_ts:
          CMP R12, RSI
          JGE end

          MOV R13B, byte ptr[RDI + R12]
          MOV R14, qword ptr[RDX + 8*R13]
          INC R14
          MOV qword ptr[RDX + 8*R13], R14

          INC R12
          JMP loop_ts
```

On réalise la seconde boucle qui compte les valeurs de ts et les place dans la bonne case de th, c'est-à-dire la case d'indice égal à la valeur actuelle de ts. Pour ce faire, on parcourt le tableau th comme précédemment en comparant un indice placé arbitrairement dans le registre R12 qu'on recycle à la longueur du tableau que l'on a reçu en argument dans le registre RSI. Lorsque l'indice atteint la longueur du tableau, on saute à la fin du programme. On doit préalablement ré-initialiser le registre d'indice R12 qu'on recycle à 0 pour pointer sur la première case du tableau.

Ensuite, si on a pas atteint la fin du tableau ts, on déréférence la valeur actuelle de celui-ci en employant un adressage mémoire indirect indexé par le registre R12 et qualifié par un byte, puisqu'on déréférence bien un tableau d'octets. On place cette valeur dans le registre arbitraire R13, ou plus précisément sa sous-partie R13B correspondant à un octet. On doit préalablement initialiser le registre de valeur R13 à 0 pour être certain qu'il ne reste rien dans les octets de poids fort de celui-ci lorsqu'on en assigne que la sous-partie R13B.

On utilise alors la valeur du tableau ts contenue dans R13B comme indice dans le tableau th. Puisque les adresses de notre architecture x86_64 sont représentées sur 64 bits, on doit employer l'entiereté du registre R13. De plus, on doit appliquer un facteur *8 à cet indice indiquant une case

de l'histogramme, puisque chaque valeur de l'histogramme prend en réalité 8 octets en mémoire. De manière correspondante, on qualifie les MOV impliquant une case de l'histogramme par le qualificatif qword, puisqu'on manipule des valeurs de 64 bits.

On place la valeur extraite du tableau `th` dans un registre arbitraire `R14` avec MOV pour pouvoir l'incrémenter avec INC, et on replace ensuite avec MOV la valeur incrémentée à l'adresse d'origine dans le tableau `th`, à la case indiquée par la valeur courante du tableau `th` toujours inchangée dans le registre `R13`. On ne doit pas préalablement initialiser le registre de valeur `R14` puisqu'il sera entièrement écrasé par la nouvelle valeur qu'on y place.

Enfin, on incrémente le registre `R12` d'indice dans le tableau `ts` et on saute à l'intitulé indiquant le début de la boucle pour la prochaine itération ou la fin du parcours.

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

histogramme:
                # RDI : contient l'adresse de ts
                # RSI : contient la longueur n de ts
                # RDX : contient l'adresse de th

                PUSH RBP
                MOV RBP, RSP
                PUSH R12
                PUSH R13
                PUSH R14

                MOV R12, 0 # R12 contiendra l'index dans th puis dans ts
                MOV R13, 0 # R13 contiendra la valeur actuelle de ts
                # R14 contiendra la valeur actuelle de th
```

On doit ajouter les clauses habituelles de déclaration de syntaxe et de la fonction `histogramme` avec son attribut "global". On doit aussi sauver le contexte précédent et définir le nouveau contexte sur la pile à l'aide des instructions d'usage PUSH RBP et MOV RBP, RSP, et sauver sur la pile les valeurs des registres `R12, R13, R14` qu'on modifie pendant l'exécution du programme. On déplace également les initialisations de registres au début du programme pour plus de lisibilité.

```
end:
        POP R14
        POP R13
        POP R12
        POP RBP
        RET
```

Enfin, on doit rédiger la section de fin du programme. Celle-ci se chargera de récupérer la valeur des registres de travail depuis la pile dans l'ordre inverse de leur empilement au début du programme, puis de restaurer le contexte appellant en dépilant RBP et enfin de mettre fin à la fonction avec RET. On remarque qu'il n'y a ici pas d'argument de sortie dans le registre RAX puisque le résultat de l'opération est placé dans le tableau th dont l'adresse est passée en argument d'entrée.

Si on assemble les blocs rédigés jusqu'ici, on a la solution finale suivante :

```
                .intel_syntax noprefix
                .text
                .global histogramme
                .type histogramme, @function
histogramme:
                # RDI : contient l'adresse de ts
                # RSI : contient la longueur n de ts
                # RDX : contient l'adresse de th

                PUSH RBP
                MOV RBP, RSP
                PUSH R12
                PUSH R13
                PUSH R14

                MOV R12, 0 # R12 contiendra l'index dans th puis dans ts
                MOV R13, 0 # R13 contiendra la valeur actuelle de ts
                # R14 contiendra la valeur actuelle de th

loop_th:
                CMP R12, 0xFF
                JG init_ts

                MOV qword ptr[RDX + 8*R12], 0

                INC R12
                JMP loop_th
```

```
init_ts:
          MOV R12, 0 # On recycle R12 pour contenir l'indice dans ts

loop_ts:
          CMP R12, RSI
          JGE end

          MOV R13B, byte ptr[RDI + R12]
          MOV R14, qword ptr[RDX + 8*R13]
          INC R14
          MOV qword ptr[RDX + 8*R13], R14

          INC R12
          JMP loop_ts
end:
          POP R14
          POP R13
          POP R12
          POP RBP
          RET
```

9.2.2 Exercice 1 (seconde partie)

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 :

(4) Repartons de la solution établie dans la série d'exercices sur le pseudo-code. L'énoncé était : 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.

```
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 :
```

Une traduction possible de ce pseudo-code en Assembleur x86_64 est la suivante :

```
                .intel_syntax noprefix
                .text
                .global add_bytes
                .type Add_bytes, @function
                .type Inc_rax, @function
Add_bytes:
                # RDI adr tab 1
                # RSI adr tab 2
                # RDX taille des tab (int -> EDX)

                MOV R8, 0
                MOV RAX, 0 # report

iterate:
                CMP R8D, EDX
                JGE end_add

                MOV R9, RAX
                MOV RAX, 0

                ADD R9B, byte ptr[RDI+R8]
                CALL inc_rax

                ADD R9B, byte ptr[RSI+R8]
                CALL inc_rax

                MOV byte ptr[RDI + R8], R9
                INC R8
                JMP iterate

end_add:
                RET
================================================
Inc_rax:
                JNC end_inc_rax
                INC RAX
end_inc_rax:
                RET
```

L'algorithme étant expliqué dans la partie concernant le pseudo-code, elle ne sera pas revue en détails dans cette section-ci.

Tout d'abord, les spécifications du programme ne changent pas de celles des exercices précé-

dents, à la différence que cette fois-ci, une seconde fonction est déclarée, appelée `Inc_rax`. Cette fonction n'est pas indispensable, mais elle est utile afin d'éviter la redondance dans le code. En effet, en programmation, dès qu'il faut écrire des instructions très similaires (voire identiques) à plusieurs endroits différents, il est préférable de créer une fonction que l'on peut appeler à ces endroits plutôt que d'y réécrire ces mêmes instructions. Le langage Assembleur ne fait pas exception à cette règle, d'où l'intérêt d'une fonction supplémentaire. Son utilisation sera expliquée plus loin.

Par convention, les trois premiers paramètres d'une fonction en assembleur sont stockés dans les registres `RDI`, `RSI` et `RDX`. Dans le cadre de ce programme, ceux-ci contiennent respectivement l'adresse du premier tableau et l'adresse du second tableau déclarées sur 64 bits, ainsi que la taille des deux tableaux. Aussi, comme cette taille est déclarée sur un `int`, seuls les 32 bits de poids faible du registre sont utilisés ; on notera donc `EDX` au lieu de `RDX`.

Les deux tableaux d'octets quant à eux correspondent aux deux opérandes de la somme, où chaque octet représente une partie du nombre représenté par le tableau. Comme il est demandé de travailler en petit-boutiste, les bits de poids faible se trouvent aux plus petites adresses. Par exemple, les 8 premiers bits de poids faible du nombre représenté par le premier tableau se trouvent dans sa première case (i.e. à l'adresse `RDI`).

```
MOV R8, 0
MOV RAX, 0 # report
```

Les registres `R8` et `RAX` sont utilisés pour servir respectivement d'indice de parcours dans les tableaux, et pour garder la valeur du report en cours en mémoire. On utilise le registre RAX qui dans la convention d'appel correspond à l'argument de sortie de telle manière à avoir en sortie de la fonction la valeur du dernier report qui dépasse la taille des tableaux dont on effectue la somme. Grâce à cette information supplémentaire, on peut faire une somme "complète" en ayant la possibilité de traiter ce report dépassant le format des opérandes - ce n'est pas demandé dans l'énoncé, mais c'est une propriété intéressante qui ne coute rien de plus que d'utiliser RAX comme registre de travail.

```
iterate:
          CMP R8D, EDX
          JGE end_add

          MOV R9, RAX
          MOV RAX, 0

          ADD R9B, byte ptr[RDI+R8]
          CALL inc_rax

          ADD R9B, byte ptr[RSI+R8]
          CALL inc_rax

          MOV byte ptr[RDI + R8], R9
          INC R8
          JMP iterate
```

L'idée de la boucle principale est la suivante : à chaque itération, il faut faire la somme des octets actuellement considérés dans les deux tableaux. Par exemple, si `R8` vaut 0, il faut faire la somme du premier octet de `RDI` avec le premier octet de `RSI`. Et s'il y a un report, il faut l'enregistrer dans `RAX` pour pouvoir l'ajouter à la somme des deux octets suivants à considérer. Et pour savoir s'il y a eu un report, il faut simplement vérifier le *Carry Flag*.

En effet, il s'agit là d'une facilité présente en Assembleur x86_64 qui ne l'était pas dans le pseudo-code : l'utilisation des drapeaux. À chaque instruction, des drapeaux sont actualisés. Ceux-ci sont représentés par un seul bit ; si le bit est à 1, on dit que le drapeau est levé, et sinon, il est baissé. Par exemple, lorsqu'une instruction `ADD` est exécutée, elle mettra à jour les drapeaux CF, ZF, SF et OF. Le drapeau qui nous intéresse dans ce cas-ci est CF, le *Carry Flag*, qui indique si un report est survenu à la position n par une opération arithmétique sur n bits (et donc s'il faut ajuster la valeur obtenue dans le cas de notre programme, en ajoutant ce report à l'octet suivant).

```
CMP R8D, EDX
JGE end_add
```

Tout d'abord, le gardien de boucle vérifie si le compteur est arrivé à la taille du tableau. Si c'est le cas, on fait un saut à la fin du programme, puisque les deux tableaux ont été parcourus du début à la fin. Sinon, le JGE (*Jump if Greater or Equal*) est ignoré, et on entre dans le corps de la boucle.

```
MOV R9, RAX
MOV RAX, 0
```

Ensuite, le registre R9 est utilisé pour contenir le résultat de la somme des deux octets considérés lors de l'itération. Et plus précisément, puisqu'il s'agit d'octets qui sont manipulés, on utilisera l'écriture R9B. Ce registre doit être initialisé avec la valeur de RAX. En effet, à chaque itération, RAX contient la valeur du report de l'itération précédente. Il faut donc initialiser R9 avec sa valeur pour pouvoir compter ce report dans la somme des deux octets considérés lors de l'itération actuelle.

```
ADD R9B, byte ptr[RDI+R8]
CALL inc_rax

ADD R9B, byte ptr[RSI+R8]
CALL inc_rax
```

Il faut ensuite effectuer la somme des "R8-ièmes" cases de RDI et de RSI, tout en tenant compte du report de l'itération précédente. Pour ce faire, il faut d'abord sommer l'octet actuel de RDI à R9B, et puis l'octet actuel de RSI à ce même registre.

Cependant il y a une chose très importante à vérifier : la présence d'un nouveau report entre chaque somme. En effet, après avoir fait la somme de R9B et du "R8-ième" octet de RDI, il faut vérifier si un report a été observé. Pour ce faire, il faut vérifier le *Carry Flag*. Et finalement, si le drapeau est levé, il faut incrémenter la valeur de RAX.

Comme cette vérification de report doit être effectuée à deux endroits de la boucle (lors de la somme de R9B et de l'octet de RDI, et puis lors de la somme de R9B et de l'octet de RSI), il est opportun de créer une fonction qui sera appelée pour effectuer toutes ces instructions. C'est ce qui se passe avec la fonction Inc_rax, qui est appelée grâce à l'instruction CALL.

```
Inc_rax:
                JNC end_inc_rax
                INC RAX
end_inc_rax:
                RET
```

Cette fonction vérifie si le *Carry Flag* a été levé ou non. Si ce n'est pas le cas, alors un JNC (*Jump if Not Carry*) est effectué vers l'étiquette de fin de fonction end_inc_rax, et l'instruction RET effectue un retour vers l'endroit d'où la fonction a été appelée dans le programme (au niveau du CALL). Et si le drapeau a été levé, alors RAX est incrémenté, et la fonction s'arrête de même.

Il est important de noter que la fonction doit être appelée immédiatement après que les sommes aient été effectuées. En effet, il ne faudrait pas laisser d'autres instructions altérer la valeur du CF. Il faut donc que le CALL à la fonction soit réalisé tout de suite après l'instruction ADD.

```
MOV byte ptr[RDI + R8], R9
INC R8
JMP iterate
```

Finalement, les dernières instructions de la boucle consistent à placer la valeur du registre de la somme R9 à l'adresse de la "R8-ième" case de RDI, puisque celui-ci nous sert à stocker la valeur du résultat, en plus de servir d'opérande. Et puis il faut incrémenter R8 pour pouvoir accéder à la case suivante du tableau, avant de faire un Jump vers le gardien de boucle `iterate`.

Une fois les tableaux parcourus, l'instruction RET à l'étiquette end_add informera le programme que la fonction est terminée. Dans le programme que l'on propose en solution, la fonction appelante de Add_bytes peut récupérer la valeur contenue dans RAX. Cette valeur vaudra alors 0 si aucun report n'a été effectué sur le résultat de la somme des derniers octets de poids fort des nombres, ou bien la valeur du report si celui-ci a eu lieu. C'est alors à la fonction appelante de prendre cela en considération si elle souhaite utiliser cet argument de sortie additionel.

Une dernière chose importante à prendre en compte est la gestion de la pile. Généralement en Assembleur x86_64, lorsqu'une autre fonction est appelée au sein d'une fonction, et qu'il est possible que celle-ci utilise les mêmes registres que ceux utilisés par la fonction appelante, il faut préserver ces registres en les plaçant dans la pile grâce aux instructions PUSH, avant de les dépiler lorsque la fonction appelée est terminée, grâce à l'instruction POP. Par convention, on considère qu'il faut protéger les registres RBX, RBP, R12, R13, R14 et R15. Dans ce cas-ci, ces registres ne sont pas affectés et on ne doit donc pas les sauver ; il n'est donc pas nécessaire d'utiliser la pile pour préserver des informations susceptibles d'être écrasées, puisqu'il n'y en a pas.

(5) On repart de la solution établie dans les séries d'exercices précédents en rappelant l'énoncé auquel le programme rédigé doit répondre : compter le nombre d'octets nuls dans un tableau d'octets d'adresse et de taille données.

```
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

            WorkE ← ptr WorkA
            JMP "continue" if WorkE ≠ 0

            WorkD ← WorkD + WorkC

continue :  WorkA ← WorkA + WorkC
            JMP "loop"

end :
```

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

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

count_zeros:
                # RDI : adr tableau
                # RSI : taille tableau (int -> ESI)
                MOV RAX, 0
                MOV R9, 0
```

```
iterate:
            CMP R9D, ESI
            JGE end
            CMP byte ptr[RDI + R9], 0
            JNE continue
            INC RAX

continue:
            INC R9
            JMP iterate
end:
            RET
```

Les spécifications du programme restent les mêmes que celles des 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).

À la différence des exercices précédents, cette fois-ci il faut renvoyer une valeur : le nombre d'octets nuls du tableau. Par convention, c'est le registre `RAX` qui contiendra la valeur de retour de la fonction. L'idée pour résoudre cet exercice est donc d'initialiser `RAX` à 0, et de l'incrémenter à chaque fois que l'on tombe sur un octet nul dans le tableau.

On commence donc par initialiser `RAX` à 0, ainsi que `R9` qui nous servira d'indice pour parcourir le tableau, comme dans les exercices précédents.

```
iterate:
            CMP R9D, ESI
            JGE end
            CMP byte ptr[RDI + R9], 0
            JNE continue
            INC RAX
```

Au début de la boucle, le gardien compare le contenu du registre `R9` avec le contenu de `RSI`, qui représente la taille du tableau. Comme cette valeur est contenue sur 32 bits, il faut comparer les 32 bits de poids faible de ces deux registres ; d'où la notation `R9D` et `ESI`. Si le compteur a atteint la taille du tableau (i.e. tout le tableau a été parcouru), on effectue un JGE (Jump if Greater or Equal) à la fin du programme.

Ensuite, dans le corps de la boucle, on compare la "`R9`-ième" case de celui-ci en utilisant l'opérateur de déréférencement sur le byte situé à l'adresse `RDI+R9`, avec la valeur 0. L'instruction suivante est un JNE (Jump if Not Equal) à l'étiquette `continue`. Celle-ci sera exécutée si la case comparée du tableau est différente de 0. Elle aura pour effet d'ignorer l'instruction `INC RAX`, et donc de ne pas incrémenter le nombre de valeurs nulles du tableau.

```
continue:
            INC R9
            JMP iterate
end:
            RET
```

Finalement, le compteur R9 est incrémenté et un jump inconditionnel est effectué vers le gardien de boucle à l'étiquette `iterate`. La dernière instruction RET marque la fin de la fonction. On rappelle que RAX contiendra au moment de RET la bonne valeur de l'argument de sortie de la fonction, c'est-à-dire le nombre d'octets nuls dans le tableau passé en entrée.

(6) On repart de la solution établie dans les séries d'exercices précédents en rappelant l'énoncé auquel le programme rédigé doit répondre : à partir d'un pointeur vers une chaîne de caractères dont la fin est indiquée par un zéro de terminaison, calculer sa longueur.

```
Hypothèses  R_str : contient l'adresse de la chaîne
-----------------------------------------------------
init :      R_inc ← 1
            R_index ← 0

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

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

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

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

length_str:
            # RDI : adr tableau
            MOV RAX, 0
iterate:
            CMP byte ptr[RDI + RAX], 0
            JE end
            INC RAX
            JMP iterate
end:
            RET
```

Cet exercice-ci est très similaire au précédent. Les spécifications du programme restent les mêmes, et le programme prend en entrée un seul argument : l'adresse de la chaîne de caractères (i.e. du tableau, puisque les chaînes de caractères en langage C ne sont jamais que des tableaux d'octets). Par convention, le registre utilisé est `RDI`, et puisque l'adresse du tableau est une valeur contenue sur 64 bits, le registre entier est utilisé.

Le registre `RAX` est celui qui contiendra la valeur de retour de la fonction. Il faut donc l'initialiser à zéro. Ensuite, lors du parcours du tableau, il doit être incrémenté à chaque valeur différente

de zéro, c'est-à-dire chaque valeur de la chaîne avant le zéro de terminaison, et donc un nombre de fois égal à sa longueur. C'est exactement sur cette même idée qu'est structuré le pseudo-code. Il s'agit donc à nouveau d'adapter le contenu du pseudo-code en instructions compréhensibles par la machine, qui suivent le formalisme de l'Assembleur x86-64.

```
iterate:
          CMP byte ptr[RDI + RAX], 0
          JE end
          INC RAX
          JMP iterate
```

Puisque `RAX` doit finalement contenir la taille de la chaîne, il peut également être utilisé comme indice de parcours du tableau [2]. Et puisqu'il remplit également ce rôle, le gardien de boucle consiste en la comparaison de la "`RAX`-ième" case du tableau avec la valeur 0. On effectue ensuite un JE (Jump if Equal) à la fin du programme si c'est le cas. Sinon, on incrémente `RAX` pour symboliser que la case actuelle du tableau est différente de 0, et on effectue un jump inconditionnel vers le gardien de la boucle.

Si on atteint la fin du programme, l'argument de sortie aura bien été préalablement placé dans le registre `RAX` puisque sa valeur est incrémentée à chaque valeur non-nulle lue dans la chaîne de caractères. Lors de l'itération de la boucle menant à la fin du programme, `RAX` n'est pas incrémenté, ce qui veut dire que le zéro de terminaison n'est pas compté comme un caractère de la chaîne : c'est bien le comportement désiré.

2. Le cas d'une chaîne de caractères de taille nulle est-il couvert par ce programme ?