Initiation à l'assembleur 8086 sur système GNU/Linux

D'après Rémi HOARAU

Partie IV : La pile

I. Pourquoi et comment utiliser la pile ?

La pile est un morceau de mémoire RAM qui a été alloué afin d'y placer toutes les données nécesssaires à une fonction. Cette fonction peut par exemple avoir des variables locales, des paramètres, etc... Tout ceci est alors placé dans la pile. Lorsqu'une fonction a terminé son appel, la pile est détruite et on passe à une autre pile.

Deux instruction sont possibles ici : **push** et **pop**
push : permet de pousser un élément dans la pile
pop : permet de dépiler un élément dans la pile

En 64 bits :

- Je peux push un registre de 64 bits et de 16 bits
- Je peux push une valeur immédiate de 8 bits, 16 bits et 64 bits

En 32 bits :

- Je peux push un registre de 32 bits et de 16 bits
- Je peux push une valeur immédiate de 8 bits, 16 bits, 32 bits et 64 bits

Les deux registres qui interviennent dans la pile sont : **rsp** et **rbp**.

En 64 bits, le push va soustraire rsp par 8 et le pop va additionner rsp par 8

En 32 bits, le push va soustraire rsp par 4 et le pop va additionner rsp par 4

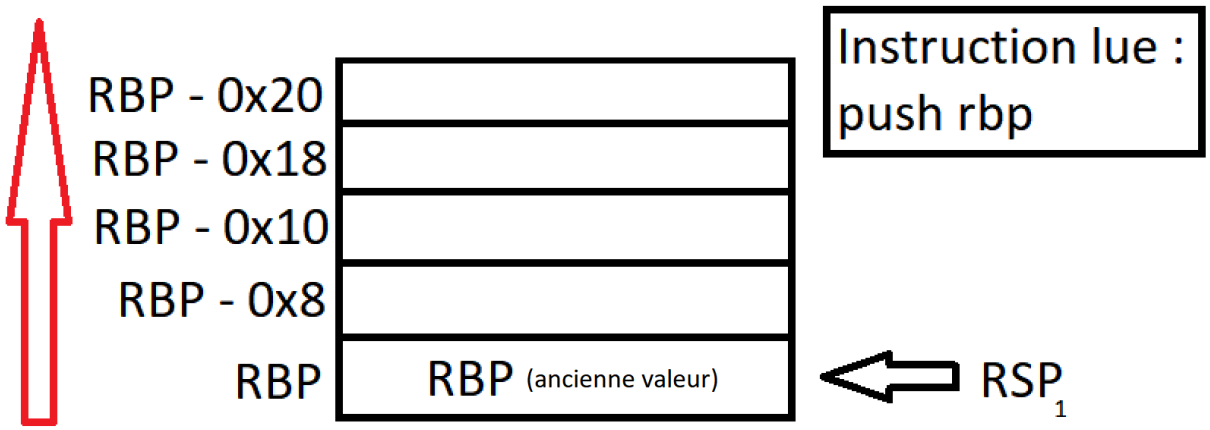
En règle générale, dans une fonction, nous retrouvons ce genre d'instructions en début de fonction (sur 64 bits) :

```
push rbp
mov rbp,rsp
sub rsp,0x10
```

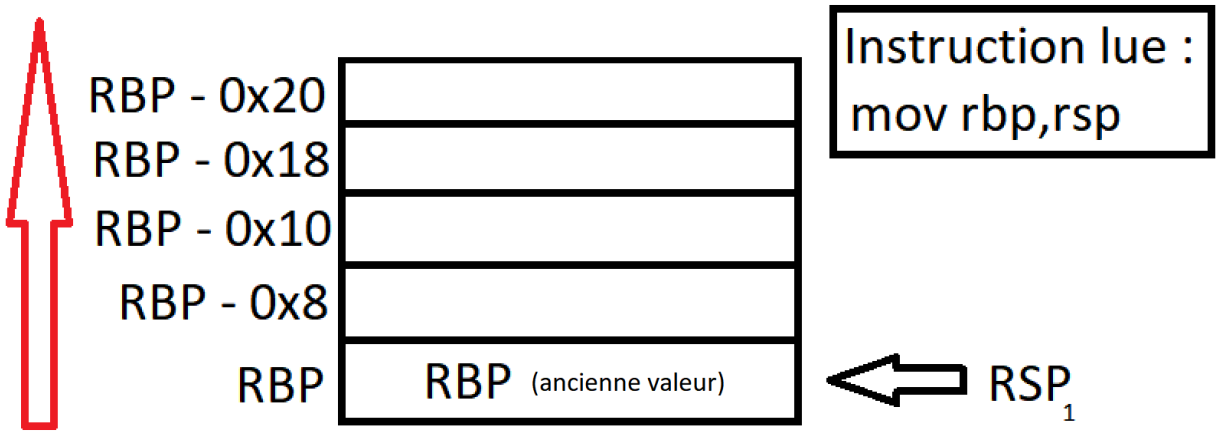
C'est ce qu'on appelle le prologue de la fonction. En fait, **rbp** représente une "ancre" où l'on va se fixer tout au long de la fonction afin de sélectionner les données que l'on veut. Dans la fonction, il pourrait y avoir d'autres push, et **rsp** en serait modifié. C'est pourquoi on push **rbp** dans la pile afin de conserver son ancienne valeur (donc l'ancre du cadre de pile de la fonction précédente), puis on place dans **rbp** l'adresse pointée par **rsp**. Là, on dit carrément à la machine <<Bon, dans ta fonction tu vas me faire tout un tas de trucs avec la pile mais moi j'm'en fous je sais que ici c'est là où s'arrête la pile à la fin de l'appel de la fonction, donc je vais marquer **rbp** par l'adresse de fin de pile>>

Le **sub rsp,0x10** signifie qu'il faut, dans cet exemple là, 0x10 octets (16 octets) à allouer dans la pile afin d'y placer les données locales à la fonction.

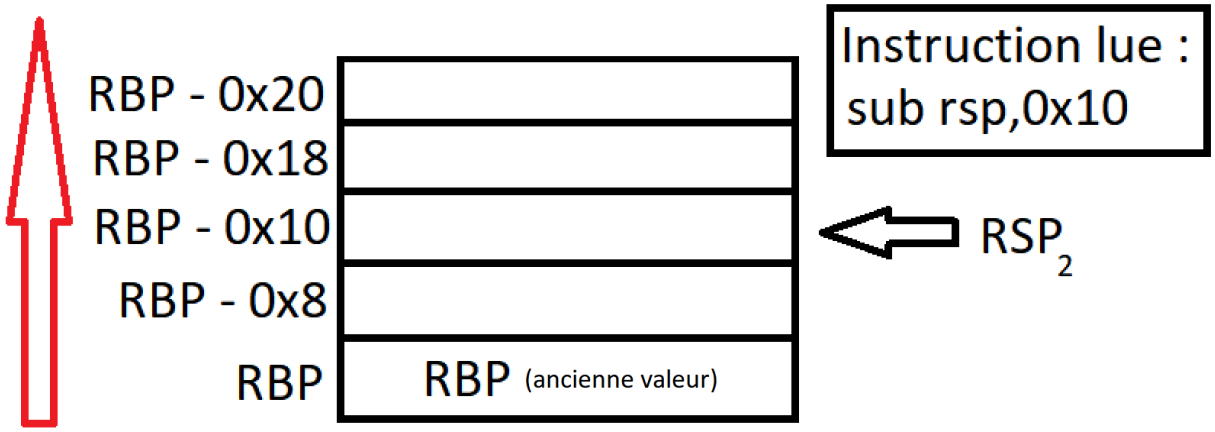
Voici un schéma récapitulatif :



==========

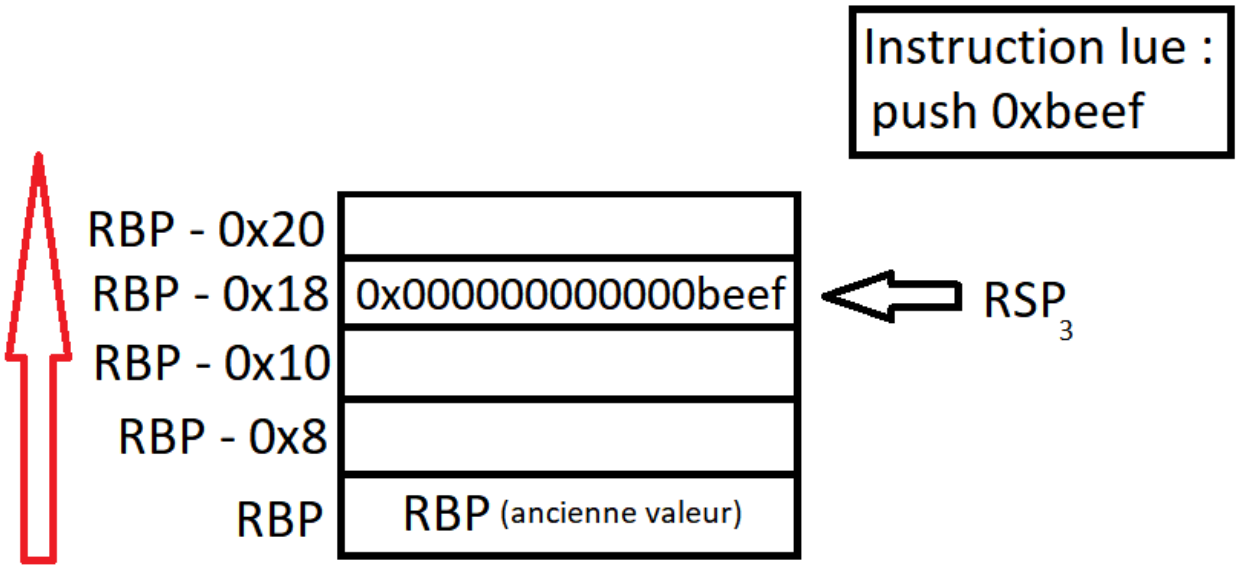


==========



A partir de là, tout ce qu'il y a entre **RBP** et **RSP2** est appelé le "**cadre de pile**", c'est ici où vont se trouver les variables locales de la fonction.

Voyon voir ce qu'il se passe lorsque l'on push par exemple **0xbeef** à la suite de ce prologue de fonction :



RSP est soustrait de 8 octets et on retrouve une multitude de 0 pour former en **RBP-0x18** un qword (8 octets).

Imagineons maintenant qu'à partir de là, on veuille quitter proprement la fonction. Comment fait-on ? On y ajoute ces instructions :

```
add rsp, 0x18
pop rbp
ret
```

Le **add rsp, 0x18** va nous permettre de placer **rsp** au niveau du **rbp** qui a été précédemment push. Ainsi, le cadre de pile est détruit. Enfin... pas complètement... Les données du cadre de pile (donc les données correspondant aux variables locales) sont toujours présentes dans la pile, mais puisque **rsp** ne peut plus atteindre ces éléments, on considère que les données ne sont plus atteignables et que les variables locales sont détruites. (C'est d'ailleurs ce qu'il se passe lorsque l'on essaie d'utiliser des variables locales d'une fonction en dehors de cette fonction : il y aura une erreur car la machine ne saura plus atteindre ces données)

Le **pop rbp** charge simplement RBP par l'ancienne valeur de RBP. Donc ici, **RBP = ancienne valeur de RBP d'un cadre de pile précédent**.

L'instruction **ret** est super importante... Contrairement aux apparences, elle ne retourne pas la valeur de retour de la fonction, mais elle retourne l'adresse de la prochaine instruction à exécuter une fois le **call** réalisé. Et cette adresse est stockée dans **RIP**. Donc techniquement, un **ret** est l'équivalent d'un **pop rip**.

Voici un exemple pour bien comprendre :

```
   0x55555555517c <main+19>:    mov    eax,0x0
=> 0x555555555181 <main+24>:    call   0x555555555060 <printf@plt>
   0x555555555186 <main+29>:    lea    rax,[rbp-0x20]
   0x55555555518a <main+33>:    mov    rsi,rax
```

Dans cet exemple, un appel à printf est réalisé à l'adresse 0x555555555181. Le **call** va réaliser deux instructions :

push 0x555555555186

jmp 0x555555555060

Le **push 0x555555555186** va pousser dans la pile l'adresse de la prochaine instruction à exécuter qui est **lea rax,[rbp-0x20]**.

Le **jmp 0x555555555060** va faire sauter **RIP** vers cette adresse, qui représente le début de la fonction **printf**. Nous verrons cette intruction plus tard.

A ce moment précis, printf va faire tout un tas de modifications dans la pile, donc **RSP** va décroître. Une fois arrivé à la fin de la fonction, nous retrouvons ceci :

```
   0x7ffff7e1cd44 <__printf+180>:       xor    rcx,QWORD PTR fs:0x28
   0x7ffff7e1cd4d <__printf+189>:       jne    0x7ffff7e1cd57 <__printf+199>
=> 0x7ffff7e1cd4f <__printf+191>:       add    rsp,0xd8
   0x7ffff7e1cd56 <__printf+198>:       ret
```

Comme convenu, **RSP** va croître de 0xd8 octets (216 octets) et il va directement pointer vers l'adresse de la prochaine instruction à exécuter après le call, que nous avons push précédemment. Donc **RSP** va pointer vers **0x555555555186**.

L'instruction ret va être exécutée, et va alors effectuer un **pop rip**.

RIP va alors pointer vers l'adresse **0x555555555186** et va pouvoir exécuter tout ce qu'il y a après la fonction **printf**. C'est un moyen de sortir proprement de la fonction.

Voilà, nous savons comment créer une fonction, allouer quelques octets pour les variables locales et sortir de proprement de cette fonction.

Mais qu'en est-il des paramètres et de la valeur de retour ?

II. Paramètres d'une fonction et valeur de retour

Pour le retour d'une fonction, c'est très simple : sur system V AMD64 ABI la valeur de retour est stockée dans rax pour une valeur de 64 bits et dans rdx:rax pour une valeur de 128 bits.

Il est alors possible de retourner uniquement une seule valeur. Cela peut être une adresse ou une valeur brute. C'est la raison pour laquelle en langage C on ne peut retourner qu'une seule valeur.

Les paramètres de fonctions dépendent du sytème et de l'architecture du processeur.

Par exemple, pour le system V AMD64 ABI, en utilisant GCC :

Argument Register Overview

Argument Type	Registers
Integer/Pointer Arguments 1-6	RDI, RSI, RDX, RCX, R8, R9
Floating Point Arguments 1-8	XMM0 - XMM7
Excess Arguments	Stack
Static chain pointer	R10

Pour l'architecture x86, par convention, les paramètres de fonction sont push dans la pile dans l'ordre inverse. Par exemple pour charger les paramètres de cette fonction :

```
fonction(a,b,c);          // langage C
```

il faudra push ces arguments dans la pile dans cet ordre précis :

```
push c
```

```
push b
```

push a

Je laisse quiconque de curieux approcher les conventions d'appels de fonctions sur ce site : https://en.wikipedia.org/wiki/X86_calling_conventions

Passons aux exercices.

III. Exercices

1. Ecrire une fonction en créant le symbole **_fonction** qui demande à l'utilisateur de saisir un octet, puis ajouter 0xaa à cet octet. Le résultat sera retourné dans **RAX**. Dans le symbole **_start**, il faudra appeler cette fonction. Les arguments seront passés dans l'ordre dans RDI, RSI, RDX, RCX, R8, R9 et l'architecture se basera sur du 64 bits.

Astuces :

- Il faudra utiliser le syscall **read** et l'instruction **add** pour additionner un registre à un autre.
- Plusieurs codes sont possibles. Il est d'ailleurs possible de réaliser la fonction sans cadre de pile ni prologue, puisque l'octet à saisir est contenu dans la section **.bss** et est globale à toute fonction. A vous de voir.

2. Ecrire une fonction en créant le symbole **_random** qui ne prends rien en paramètres et qui retourne un octet de manière aléatoire. Il faudra veiller à utiliser l'instruction **rdtsc**. Le résultat sera retourné dans EAX. Dans le symbole **_start**, il faudra appeler la fonction **_random**. L'architecture se basera sur du 32 bits.

Astuces :

- Faire attention à l'instruction **rdtsc** qui renvoie le timestamp dans **edx:eax**

3. Recoder la fonction suivante :

```
int _operation(int a,int b,int c)
{
    int d = a*b;
    int e = b*c;
    int f = d%e;
    return f-1;
}
```

Il faudra allouer assez d'espace dans la pile pour les 3 entiers en paramètres (a,b,c) ainsi que les 3 entiers de variables globales (d,e,f). Appeler cette fonction depuis le symbole **_start** en poussant les paramètres dans la pile. Le retour de la fonction se fera dans EAX. L'architecture choisie sera du 32 bits.

Astuces :

- Un entier correspond à un dword (4 octets)
- Il faudra utiliser l'instruction **mul** ou **imul** pour multiplier, **add** pour additionner, **idiv** ou **div** pour récupérer le reste de la division qui se trouvera dans **EDX** et **sub** pour soutraire.