Assembleur

Objectifs

- Comprendre les notions essentielles
- Être capable de lire et de comprendre du code simple pour architectures x86

- Pourquoi étudier l'assembleur ?
  - Prélude à l'étude de certaines attaques

Assembleur

- Langage de "bas niveau"
- Représente de manière lisible les instructions du processeur

Instruction assembleur

- Qu’est-ce qu’une instruction ?
  - Peut être vue comme une commande adressée au processeur
  - Opcode - Opération à effectuer
  - Opérande - Donnée passée en paramètre

- Une instruction est codée sur un certain nombre d’octets

Instruction x86

- CISC - Complex Instruction Set Computer
  - Nombreuses instructions
  - Instructions parfois "complexes"
  - Taille des instructions variable
  - X86, x64

- RISC - Reduced Instruction Set Computer
  - Peu d’instructions
  - Instructions simples
  - Taille des instructions fixe
  - ARM

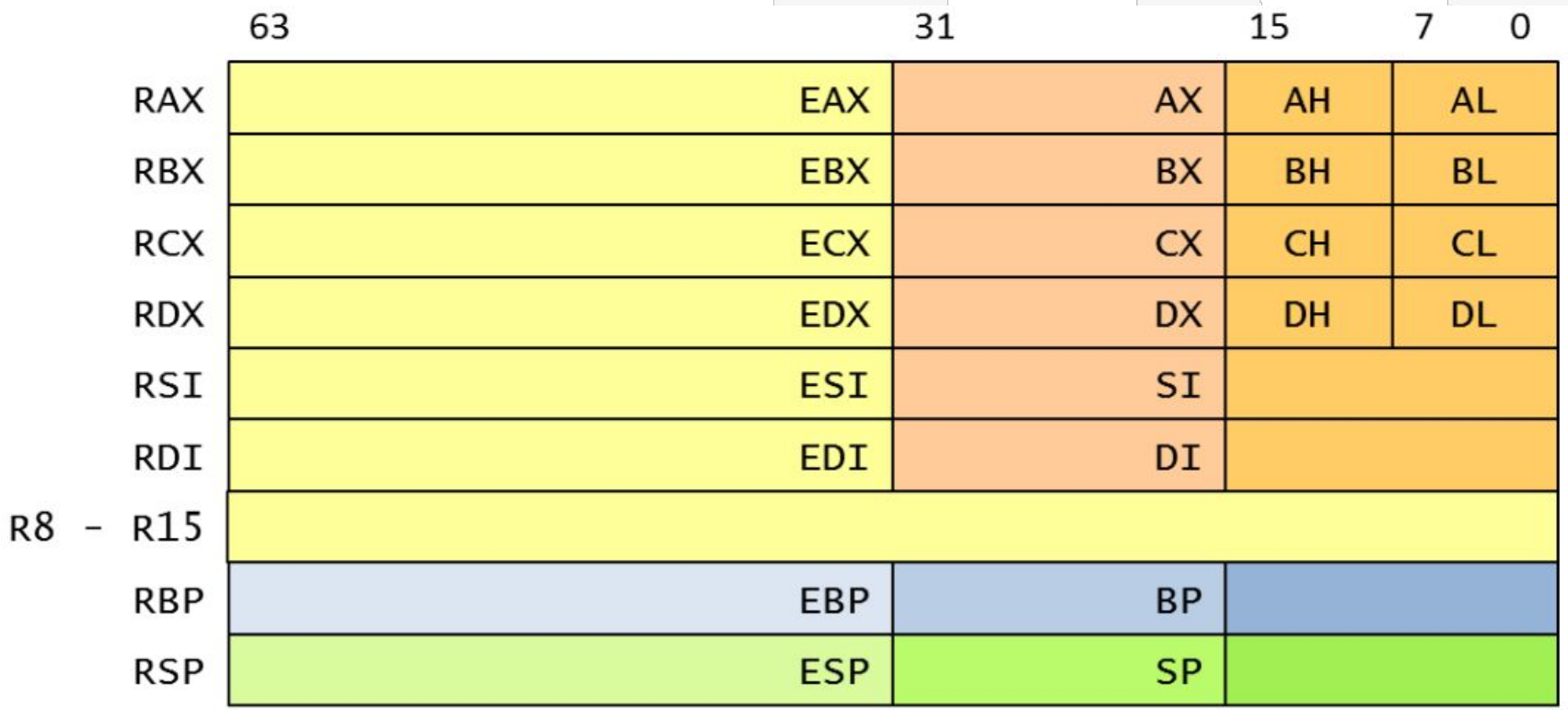
Registres du CPU

- Registre du CPU
  - Zones de stockage internes
  - Petite taille
  - Accès très rapide

- Plusieurs types
  - Registres génériques (General purpose registers)
  - Registres de segments
  - Registres de contrôle
  - Et d’autres...

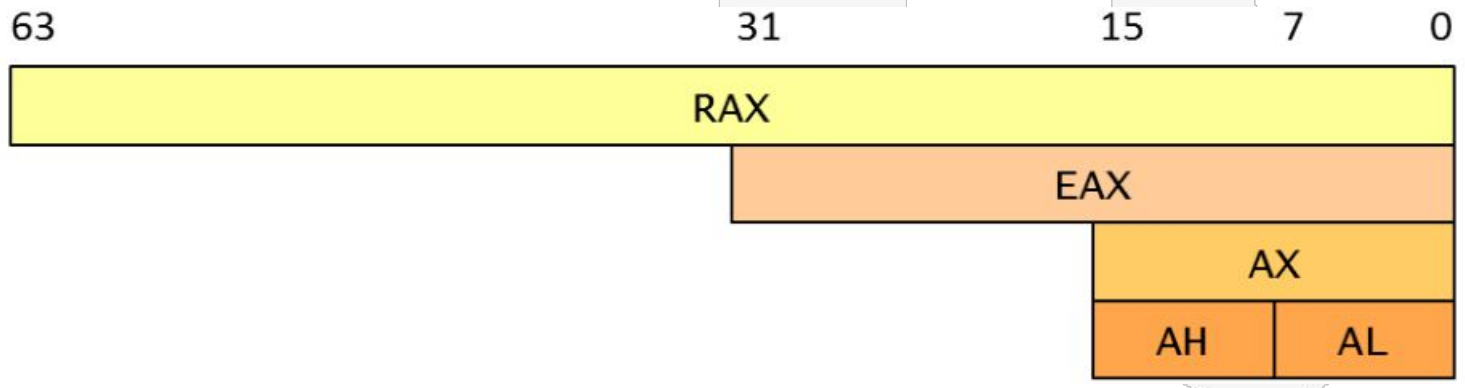
Registres génériques

- Registres utilisés pour le calcul arithmétique ou les accès à la mémoire



Registres génériques

- Registres subdivisés en sous-registres, permettant de manipuler des unités plus petites
  - RAX sur 64 bits
  - EAX sur 32 bits
  - AX sur 16 bits
  - AH et AL chacun sur 8 bits



Registres génériques

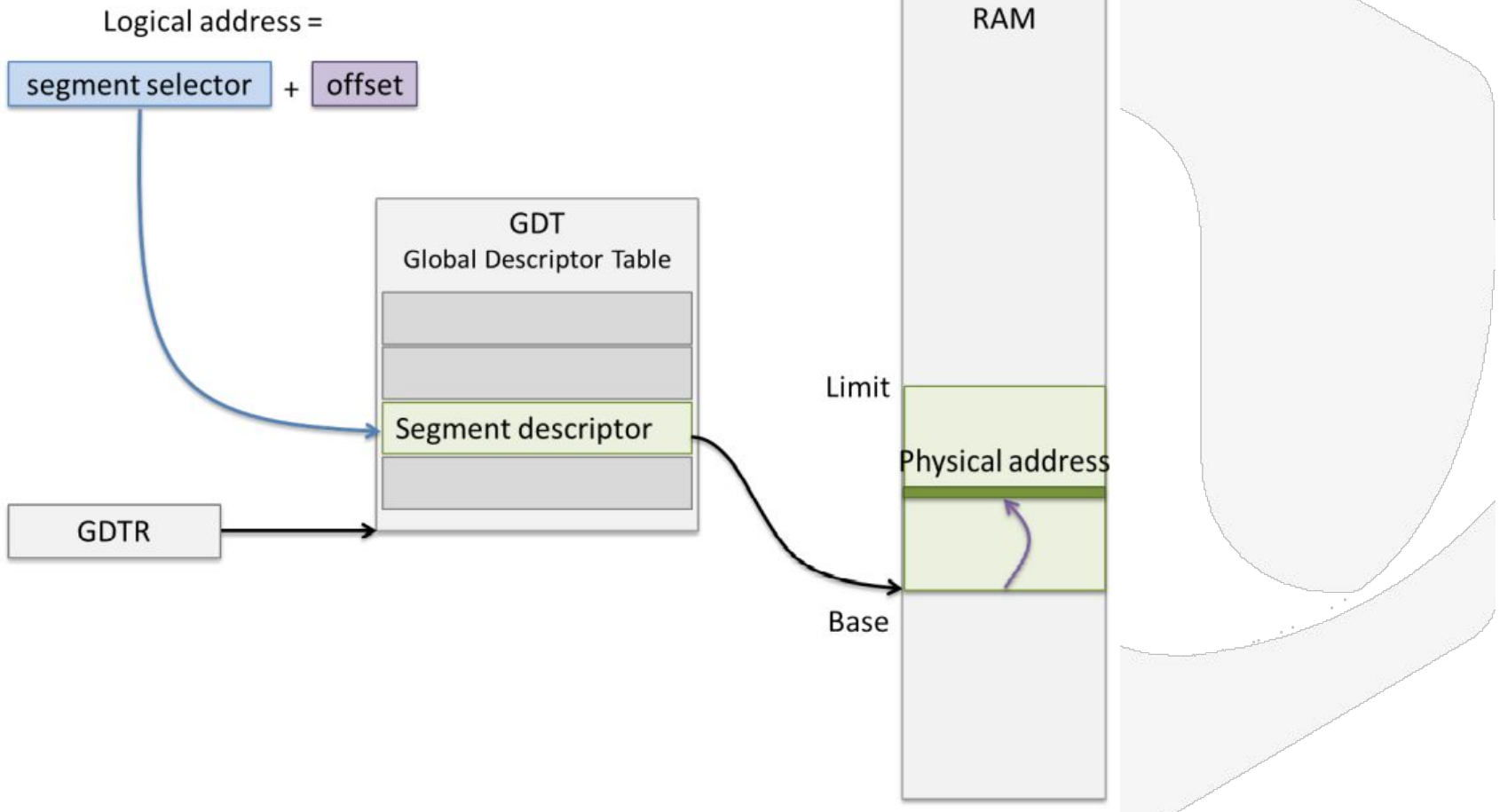
- Certains registres génériques sont réservés à une utilisation spécifique
  - RSP stocke le pointeur de pile (segment SS)
  - RBP pointeur sur le début des données dans la stack frame (segment SS)

- De nombreuses instructions utilisent des registres particuliers
  - Ex. : RCX, RSI et RDI utilisés par les instructions de manipulations de chaînes de caractères

Registres de segment

- Registres de 16 bits qui stockent un sélecteur de segment
  - CS - Segment de code
  - DS - Segment de données
  - SS - Segment de pile
  - ES - Segment de données
  - FS - Segment de données
  - GS - Segment de données

Rappel sur la segmentation



Nota : sur les architectures PC actuelles (64 bits), la base et la limite sont ignorées par le processeur

Registre d’état - RFLAGS

- Registre RFLAGS
  - Flags de statut, utilisables par l’utilisateur
  - Flags de contrôle, manipulés par le noyau

- Exemples
  - CF (Carry Flag) - Indique si la dernière opération a généré une retenue
  - ZF (Zero Flag) - Indique si le résultat de la dernière opération arithmétique ou logique vaut 0
  - IOPL (IO Privilege Flag) - Indique le niveau de privilège requis pour accéder aux I/O ports
  - IF (Interrupt Flag) - Détermine si le CPU doit traîter ou ignorer les interruptions

Le pointeur d’instruction - RIP

- Registre RIP
  - Aussi appelé compteur ordinal
  - Contient l’adresse logique de la prochaine instruction à exécuter

Autres registres

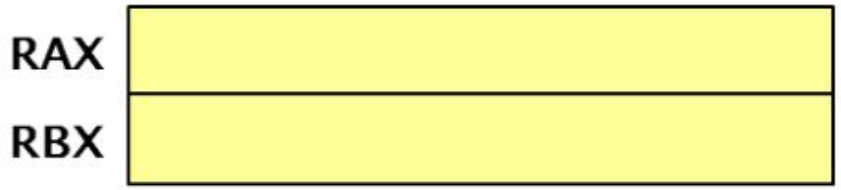
- D’autres registres
  - Registres pour le calcul sur les flottants
  - Registres de contrôle (CR0 - CR4), qui permettent de gérer la segmentation, la pagination, le cache, etc.
  - CR3 - Adresse physique du répertoire de page (32 bits) ou de la table PML4 (64 bits)
  - GDTR - Adresse linéaire de la GDT
  - IDTR - Adresse linéaire de l’IDT
  - Registres MSR spécifiques de configuration
  - Etc…

Assembleur x86 - Syntaxe

- Deux types de syntaxe :
  - INTEL
    - Ordre des opérandes : *destination, source*
  - AT&T
    - Ordre des opérandes : *source, destination*
    - Noms des registres précédés par %
    - Utilisée dans les projets GNU (ex. gcc)
- Il est important de comprendre les deux !

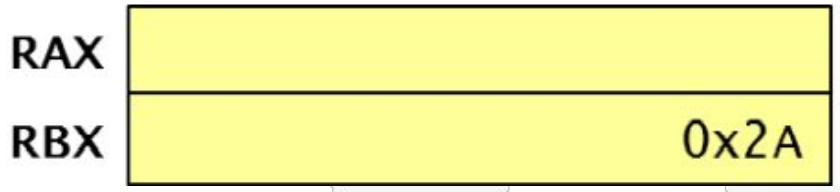
Un tout premier exemple

```
1  movl  $42 , %ebx
2  movq  $0xcafedeca , %rax
3  movb %al , %bh
```



Un tout premier exemple

```
1  movl  $42 , %ebx
2  movq  $0xcafedeca , %rax
3  movb %al , %bh
```



Un tout premier exemple

```
1  movl  $42, %ebx
2  movq  $0xcafedeca, %rax
3  movb %al, %bh
```

RAX	0xCAFEDECA
RBX	0x2A

Un tout premier exemple

```
1  movl $42, %ebx
2  movq $0xcafedeca, %rax
3  movb %al, %bh
```

RAX	0xCAFEDECA
RBX	0xCA2A

Instruction MOV

- **mov valeur, destination**
  - Copie de contenu

- Exemples
  - mov $1, %eax : Met la valeur 1 dans le registre EAX
  - mov 8(%rbp), %rbx : Copie la valeur située en mémoire, 8 octets après l'adresse pointée par RBP, dans RBX
  - mov $0x10, (%rsp) Ecrit la valeur 16 en mémoire, à l'adresse pointée par RSP

Exemples d’opérandes

- $0x4 : Entier en hexadécimal
- %eax : Contenu du registre EAX
- (%eax) : Contenu de la mémoire à l’adresse pointée par le registre EAX
- 8(%ebp) : Contenu de la mémoire 8 octets après l’adresse pointée par EBP
- fffffffc(%ebp) : Contenu de la mémoire 4 octets avant l’adresse pointée par EBP
- (%eax, %ecx, 2) : Contenu de la mémoire accédé de la manière suivante. On ajoute à l’adresse pointée par le registre EAX un offset, donné par le registre ECX, lequel est multiplié par 2.

Rappel - Représentation des entiers négatifs

- Calcul d'un nombre entier négatif
  - Complément à 2
  - Le bit de poids fort (le plus à gauche) permet de connaître le signe
  - On inverse les bits et on ajoute 1

- Exemple : comment représenter -4 ?
  - 4 en binaire : 00000100
  - On inverse les bits : 11111011
  - On ajoute 1 : 11111100

Instruction MOV - Utilisation de pointeur

```
1  mov 8(%rbx), %rax
```

Copie la valeur 0x9ABC, située à l'adresse 0x2008, dans le registre RAX

RAX	0
RBX	0x2000

0x2018	0x1234
0x2010	0x5678
0x2008	0x9ABC
0x2000	0xDEF0

Syntaxes AT&T et Intel

- Quelques exemples...
  - Intel
    - mov ebx, 0ffh
    - mov eax,[ecx]
    - add eax,[ebx+ecx*2h]
  - AT&T
    - movl $0xff, %ebx
    - movl (%ecx), %eax
    - addl (%ebx,%ecx,0x2),%eax

Appeler une fonction - 2 instructions essentielles

- **call adresse**
  - Appelle la fonction à l'adresse indiquée
  - L'adresse est le plus souvent un offset (déplacement) par rapport à l'adresse courante
  - Sauvegarde du pointeur d'instruction sur la pile

- **ret**
  - Retour à la fonction appelante en dépilant le pointeur d'instruction sur la pile

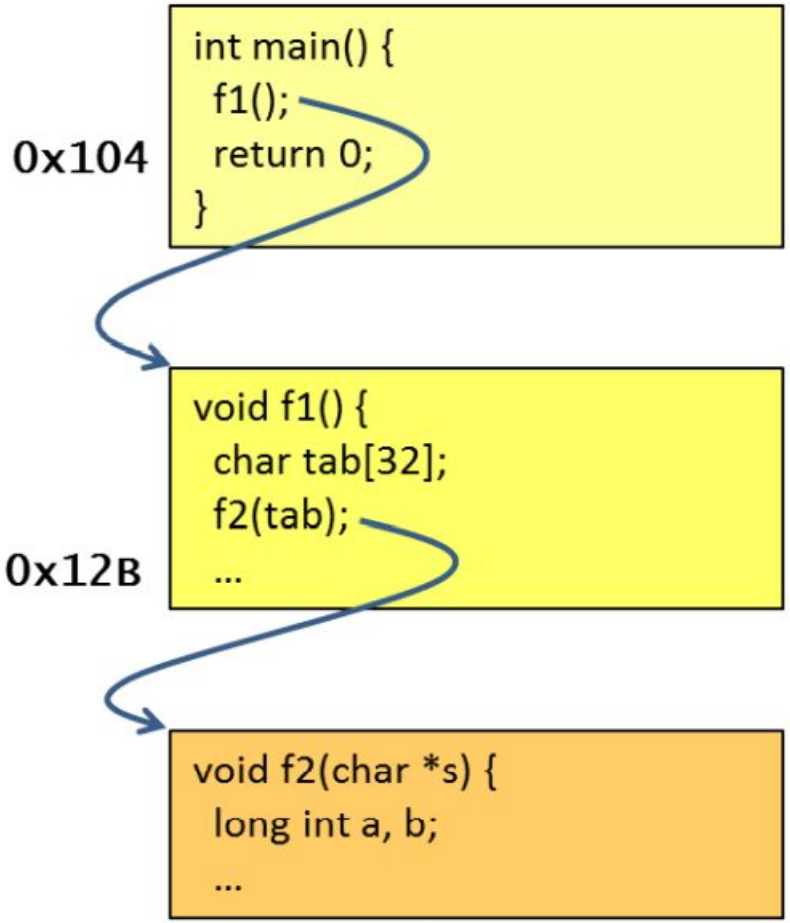
Appeler une fonction - La pile

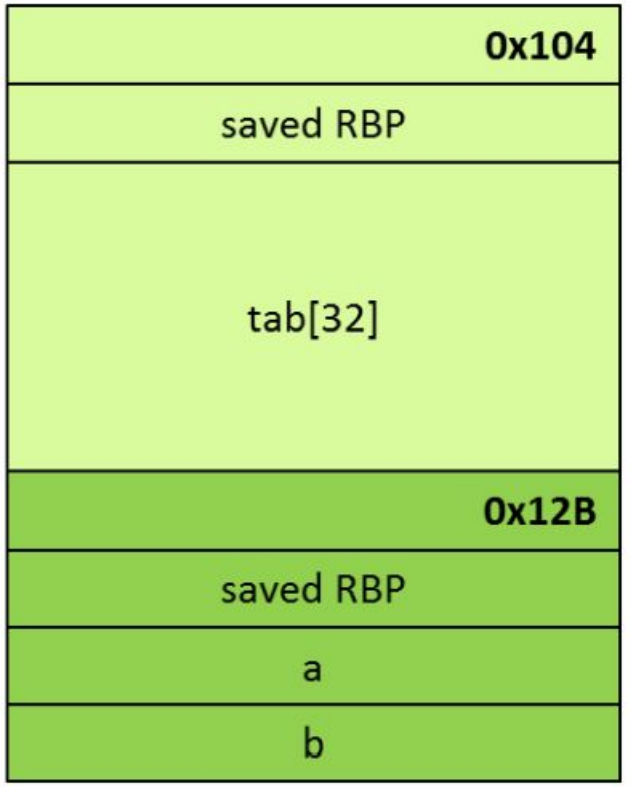
- Pile (stack)
  - Espace en RAM
  - Sert à stocker / sauvegarder de manière éphémère des données
  - Lors d'un appel de fonction (call), le processeur utilise automatiquement la pile pour sauvegarder le pointeur d'instruction

- La pile est également utilisée dans les programmes lors des appels de fonction pour
  - 1 : Passer des paramètres aux fonctions appelées
  - 2 : Stocker les variable locales utilisées dans les fonctions

- Rappel : les variables globales ne sont pas stockées sur la pile mais dans les sections de données (.rodata, .data et .bss)

Appeler une fonction - La pile





Appeler une fonction - Du C à l'assembleur

```
1  long addition (long a, long b)
2  {
3    return a+b;
4  }
5
6  int main()
7  {
8    long c;
9    c = addition(2, 3);
10   return 0;
11 }
```

Appeler une fonction - Du C à l'assembleur

Compiler le programme en mettant les optimisations à 0

```
1  gcc -O0 main.c
```

Affichage du code assembleur

```
1  objdump -d a.out
```

Une autre méthode : compiler du C vers l'assembleur

```
1  gcc -O0 -S main.c
2  less main.s
```

Appeler une fonction - Du C à l'assembleur

```
1  int main()
2  {
3    long c;
4    c = addition(2, 3);
5    return 0;
6  }
```

```
1  0000000000000679 <main>:
2   679:   55                      push   %rbp
3   67a:   48 89 e5                mov    %rsp,%rbp
4   67d:   48 83 ec 10             sub    $0x10,%rsp
5   681:   be 03 00 00 00          mov    $0x3,%esi
6   686:   bf 02 00 00 00          mov    $0x2,%edi
7   68b:   e8 d0 ff ff ff          callq  660 <addition>
8   690:   48 89 45 f8             mov    %rax,-0x8(%rbp)
9   694:   b8 00 00 00 00          mov    $0x0,%eax
10  699:   c9                      leaveq
11  69a:   c3                      retq
```

Appeler une fonction - Du C à l'assembleur

```
1  int main()
2  {
3    long c;
4    c = addition(2, 3);
5    return 0;
6  }
```

```
1  0000000000000679 <main>:
2   679:   55                      push   %rbp
3   67a:   48 89 e5                mov    %rsp,%rbp
4   67d:   48 83 ec 10             sub    $0x10,%rsp
5   681:   be 03 00 00 00          mov    $0x3,%esi
6   686:   bf 02 00 00 00          mov    $0x2,%edi
7   68b:   e8 d0 ff ff ff          callq  660 <addition>
8   690:   48 89 45 f8             mov    %rax,-0x8(%rbp)
9   694:   b8 00 00 00 00          mov    $0x0,%eax
10  699:   c9                      leaveq
11  69a:   c3                      retq
```

Passer des paramètres à une fonction

- Conventions pour les architecture 64-bit
  - Microsoft x64 calling convention
  - RCX, RDX, R8, R9 et arguments supplémentaires passés par la pile
  - La valeur de retour est stockée dans RAX

- System V AMD64 ABI
  - RDI, RSI, RDX, RCX, R8, R9 et arguments supplémentaires passés par la pile
  - Les valeurs de retour sont stockées dans RAX et RDX

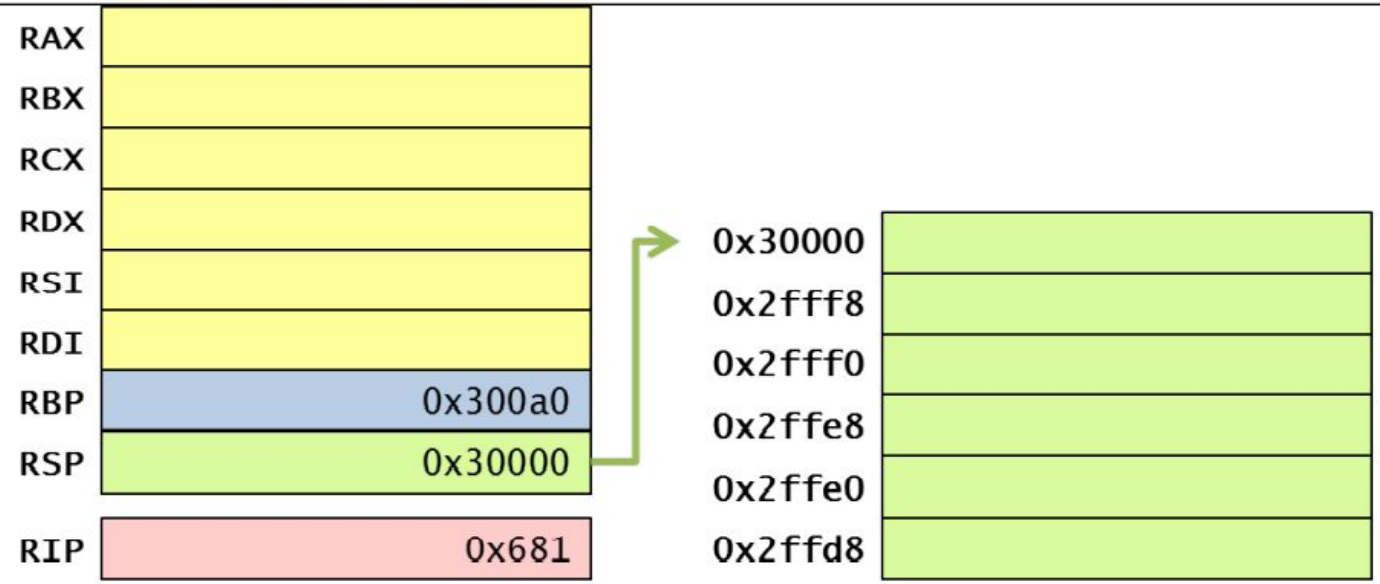
Appeler une fonction - Du C à l'assembleur

```c
int main()
{
  long c;
  c = addition(2, 3);
  return 0;
}
```

```
0000000000000679 <main>:
 679:   55                      push   %rbp
 67a:   48 89 e5                mov    %rsp,%rbp
 67d:   48 83 ec 10             sub    $0x10,%rsp
 681:   be 03 00 00 00          mov    $0x3,%esi
 686:   bf 02 00 00 00          mov    $0x2,%edi
 68b:   e8 d0 ff ff ff          callq  660 <addition>
 690:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 694:   b8 00 00 00 00          mov    $0x0,%eax
 699:   c9                      leaveq
 69a:   c3                      retq
```

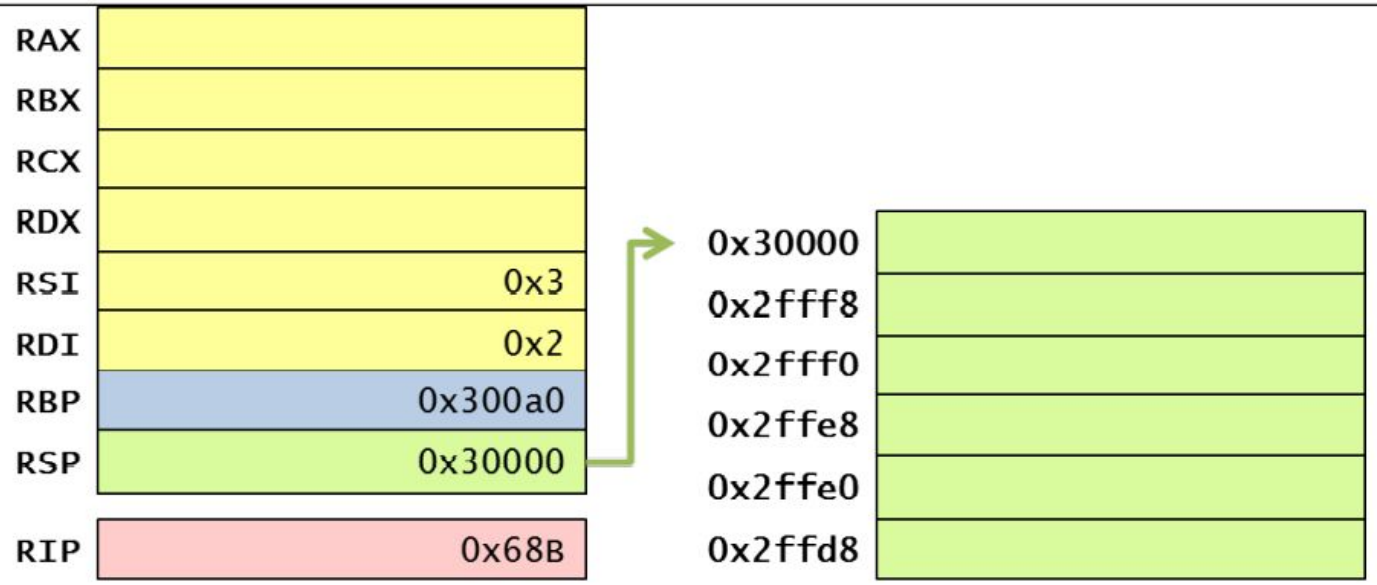
Du C à l'assembleur - Exécution en image - 1

```
1   679:    55                      push   %rbp
2   67a:    48 89 e5                mov    %rsp,%rbp
3   67d:    48 83 ec 10             sub    $0x10,%rsp
4   681:    be 03 00 00 00          mov    $0x3,%esi
5   686:    bf 02 00 00 00          mov    $0x2,%edi
6   68b:    e8 d0 ff ff ff          callq  660 <addition>
7   690:    48 89 45 f8             mov    %rax,-0x8(%rbp)
8   694:    b8 00 00 00 00          mov    $0x0,%eax
9   699:    c9                      leaveq
10  69a:    c3                      retq
```



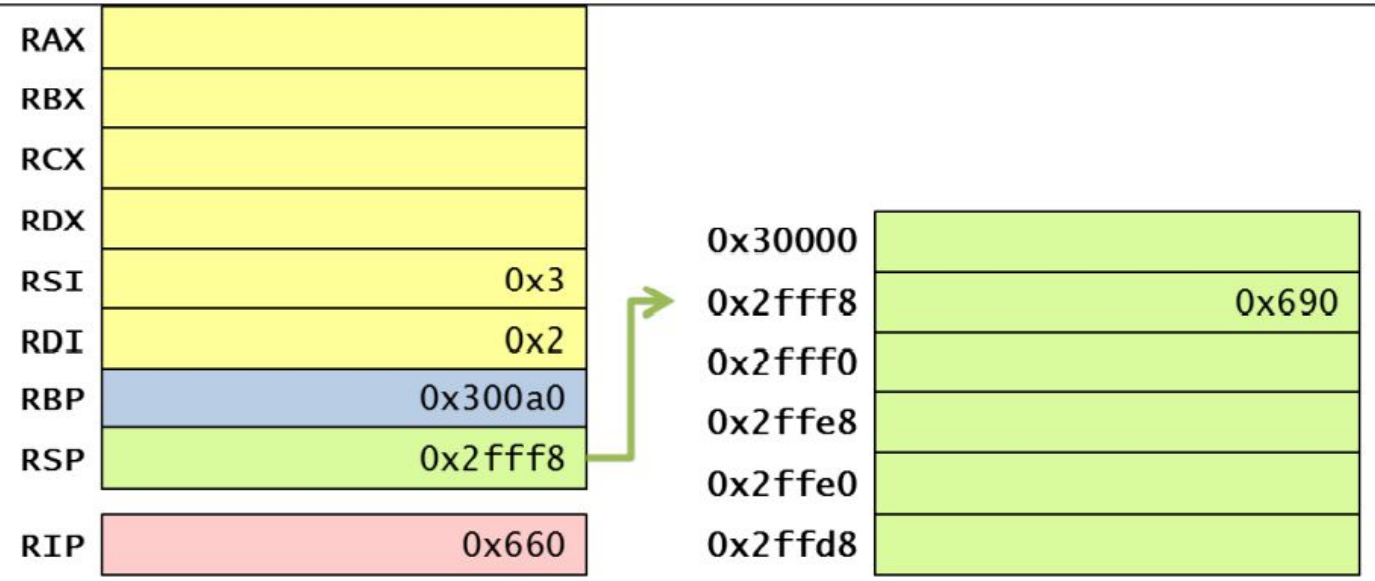
Du C à l'assembleur - Exécution en image - 2

```
1   679:    55                      push   %rbp
2   67a:    48 89 e5                mov    %rsp,%rbp
3   67d:    48 83 ec 10             sub    $0x10,%rsp
4   681:    be 03 00 00 00          mov    $0x3,%esi
5   686:    bf 02 00 00 00          mov    $0x2,%edi
6   68b:    e8 d0 ff ff ff          callq  660 <addition>
7   690:    48 89 45 f8             mov    %rax,-0x8(%rbp)
8   694:    b8 00 00 00 00          mov    $0x0,%eax
9   699:    c9                      leaveq
10  69a:    c3                      retq
```



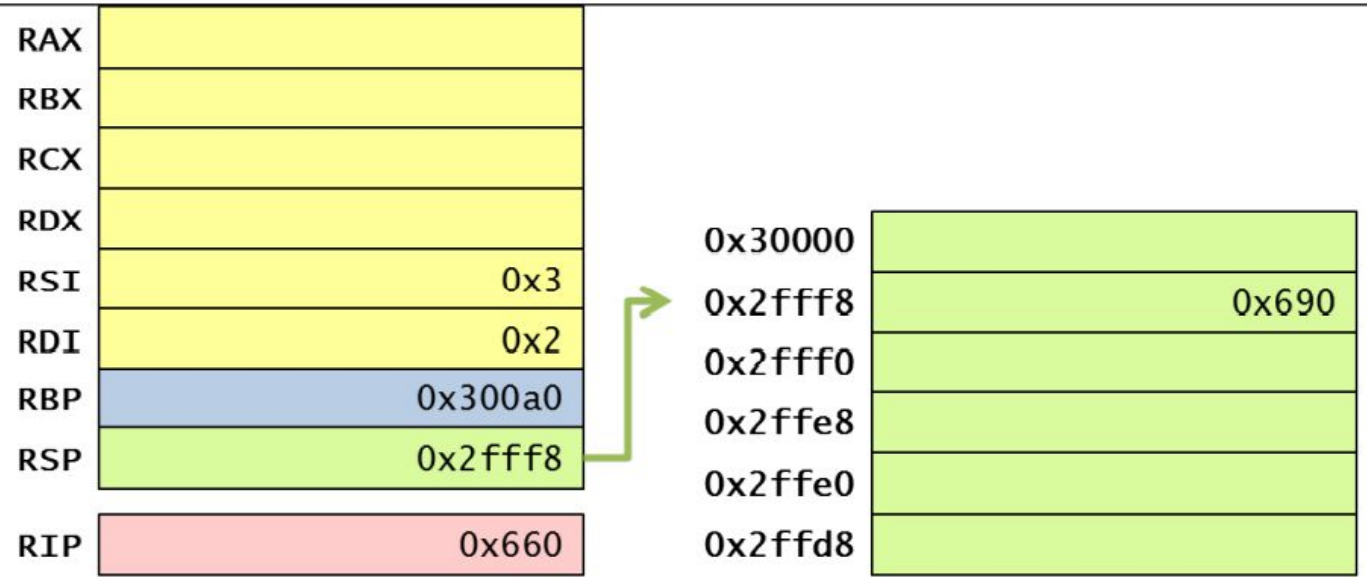
Du C à l'assembleur - Exécution en image - 3

```
0000000000000660 <addition>:
 660:   55                      push   %rbp
 661:   48 89 e5                mov    %rsp,%rbp
 664:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
 668:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
 66c:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
 670:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 674:   48 01 d0                add    %rdx,%rax
 677:   5d                      pop    %rbp
 678:   c3                      retq
```



Du C à l'assembleur - Exécution en image - 4

```
0000000000000660 <addition>:
 660:   55                      push   %rbp
 661:   48 89 e5                mov    %rsp,%rbp
 664:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
 668:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
 66c:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
 670:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 674:   48 01 d0                add    %rdx,%rax
 677:   5d                      pop    %rbp
 678:   c3                      retq
```



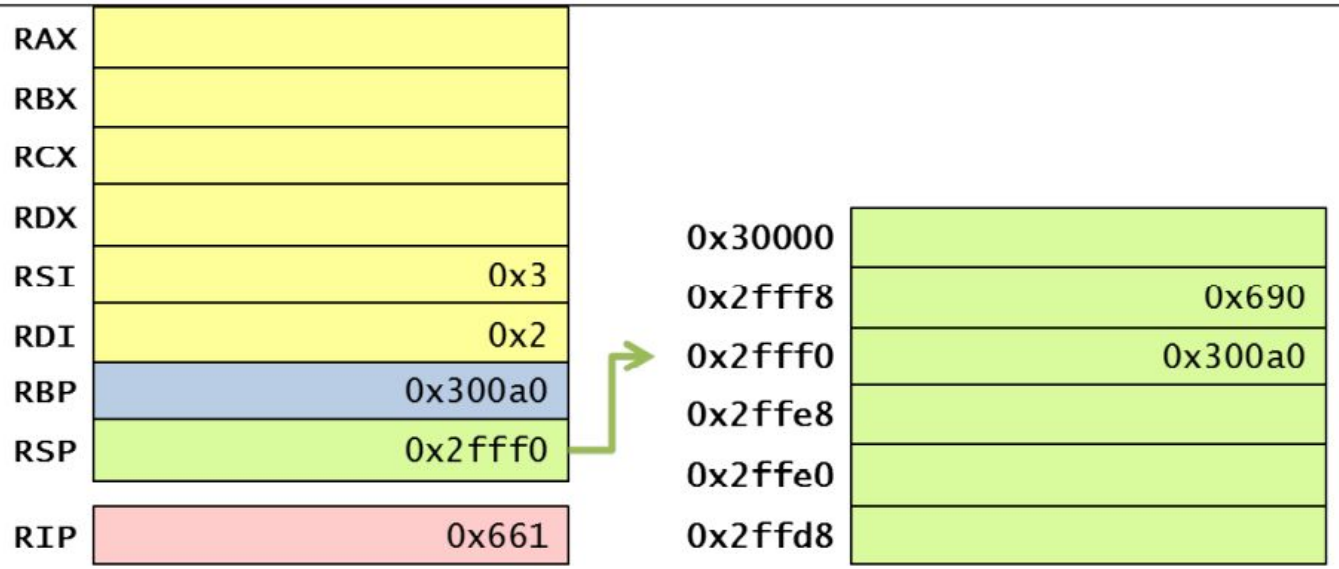
Instructions pour manipuler la pile

- **push valeur**
  - Décrémente le pointeur de pile puis stocke une valeur dans la pile
  - Sur presque l'ensemble des architectures, la pile croit vers les adresses basses

- **pop destination**
- Retire une valeur du sommet de la pile puis incrémente le pointeur

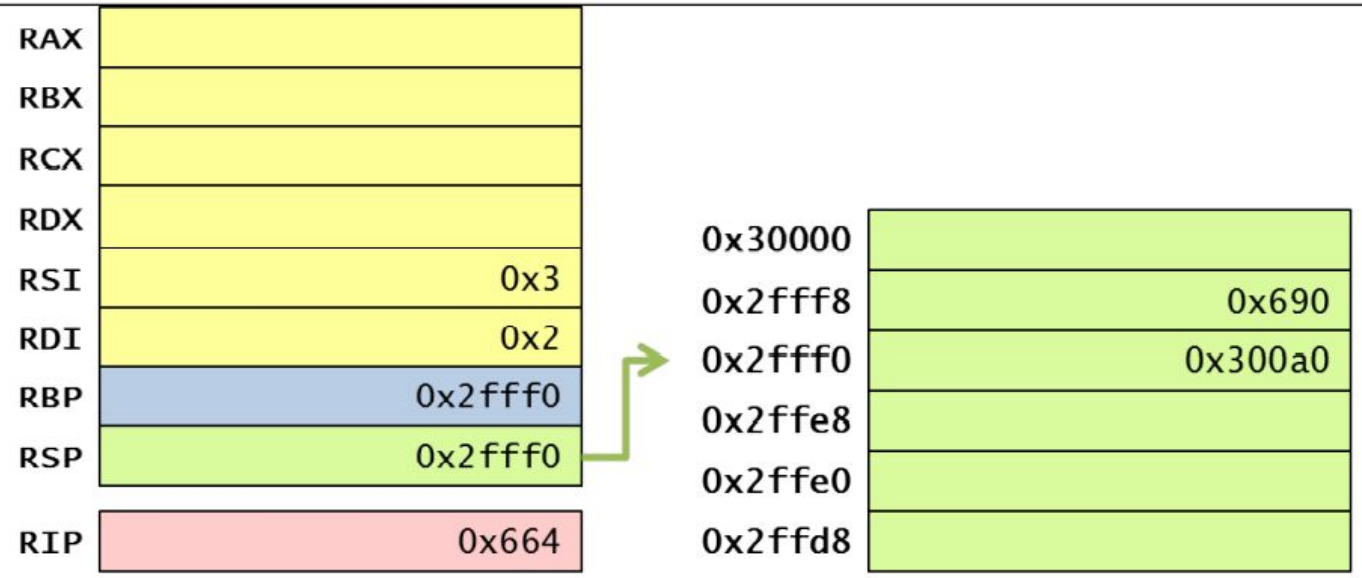
Du C à l'assembleur - Exécution en image - 5

```
1  0000000000000660 <addition>:
2    660:   55                      push   %rbp
3    661:   48 89 e5                mov    %rsp,%rbp
4    664:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
5    668:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
6    66c:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
7    670:   48 8b 45 f0             mov    -0x10(%rbp),%rax
8    674:   48 01 d0                add    %rdx,%rax
9    677:   5d                      pop    %rbp
10   678:   c3                      retq
```



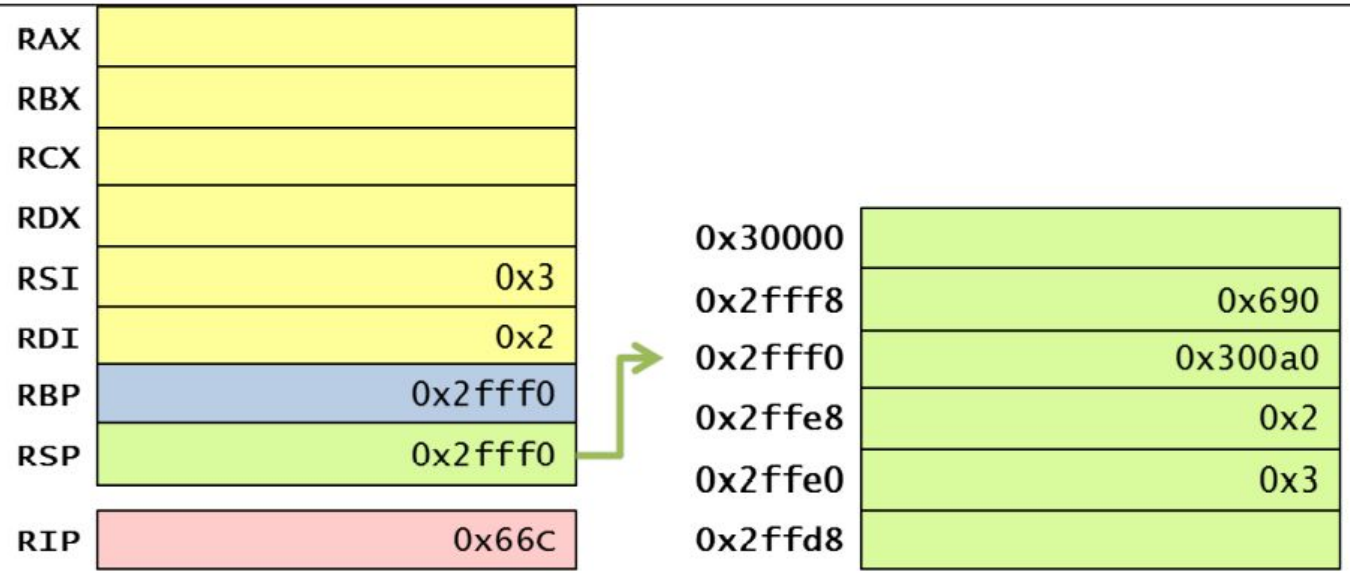
Du C à l'assembleur - Exécution en image - 6

```
1  0000000000000660 <addition>:
2    660:   55                      push   %rbp
3    661:   48 89 e5                mov    %rsp,%rbp
4    664:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
5    668:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
6    66c:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
7    670:   48 8b 45 f0             mov    -0x10(%rbp),%rax
8    674:   48 01 d0                add    %rdx,%rax
9    677:   5d                      pop    %rbp
10   678:   c3                      retq
```



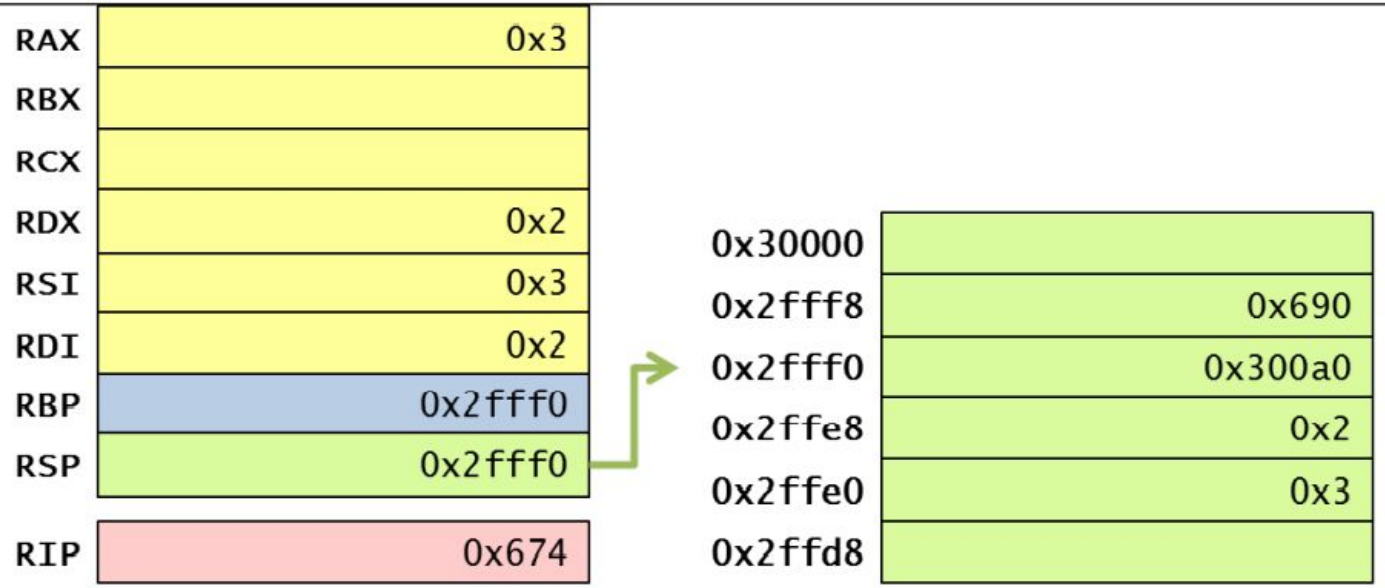
Du C à l'assembleur - Exécution en image - 7

```
1  0000000000000660 <addition>:
2    660:   55                      push   %rbp
3    661:   48 89 e5                mov    %rsp,%rbp
4    664:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
5    668:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
6    66c:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
7    670:   48 8b 45 f0             mov    -0x10(%rbp),%rax
8    674:   48 01 d0                add    %rdx,%rax
9    677:   5d                      pop    %rbp
10   678:   c3                      retq
```



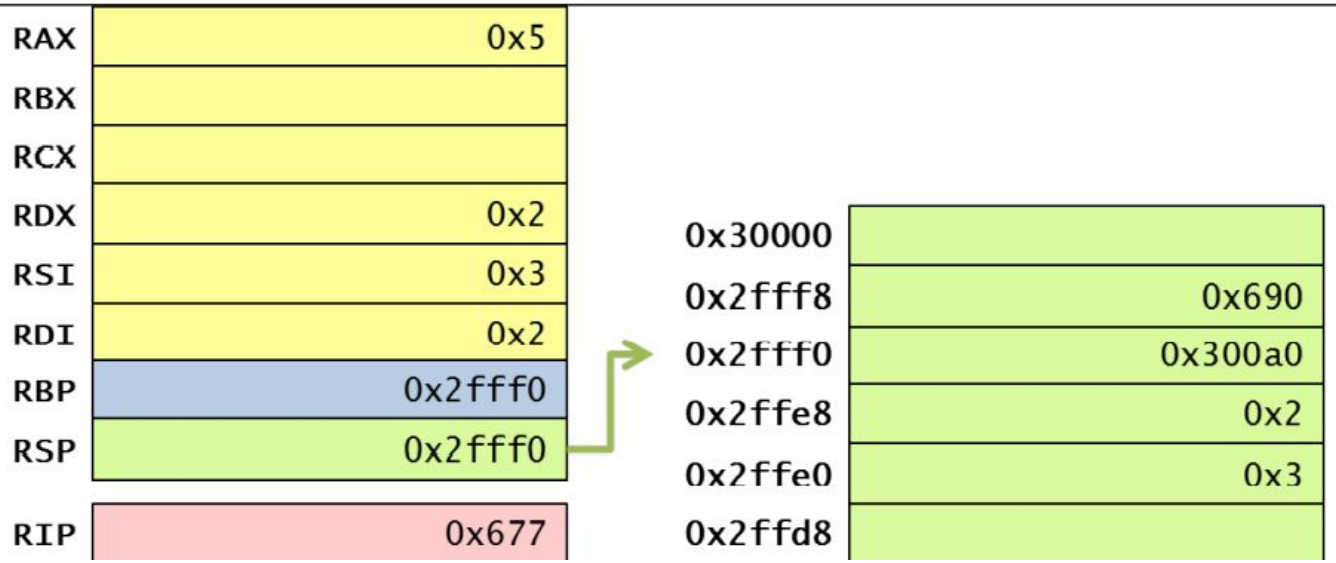
Du C à l'assembleur - Exécution en image - 8

```
1  0000000000000660 <addition>:
2    660:   55                    push   %rbp
3    661:   48 89 e5              mov    %rsp,%rbp
4    664:   48 89 7d f8           mov    %rdi,-0x8(%rbp)
5    668:   48 89 75 f0           mov    %rsi,-0x10(%rbp)
6    66c:   48 8b 55 f8           mov    -0x8(%rbp),%rdx
7    670:   48 8b 45 f0           mov    -0x10(%rbp),%rax
8    674:   48 01 d0              add    %rdx,%rax
9    677:   5d                    pop    %rbp
10   678:   c3                    retq
```



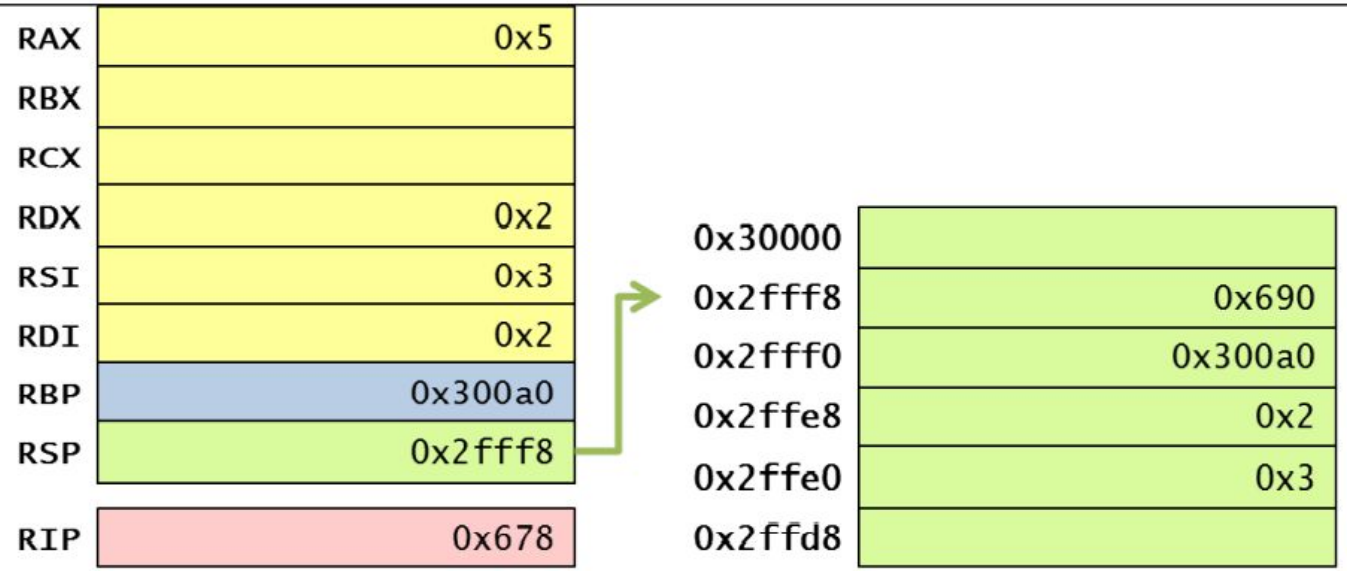
Du C à l'assembleur - Exécution en image - 9

```
1  0000000000000660 <addition>:
2    660:   55                      push   %rbp
3    661:   48 89 e5                mov    %rsp,%rbp
4    664:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
5    668:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
6    66c:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
7    670:   48 8b 45 f0             mov    -0x10(%rbp),%rax
8    674:   48 01 d0                add    %rdx,%rax
9    677:   5d                      pop    %rbp
10   678:   c3                      retq
```



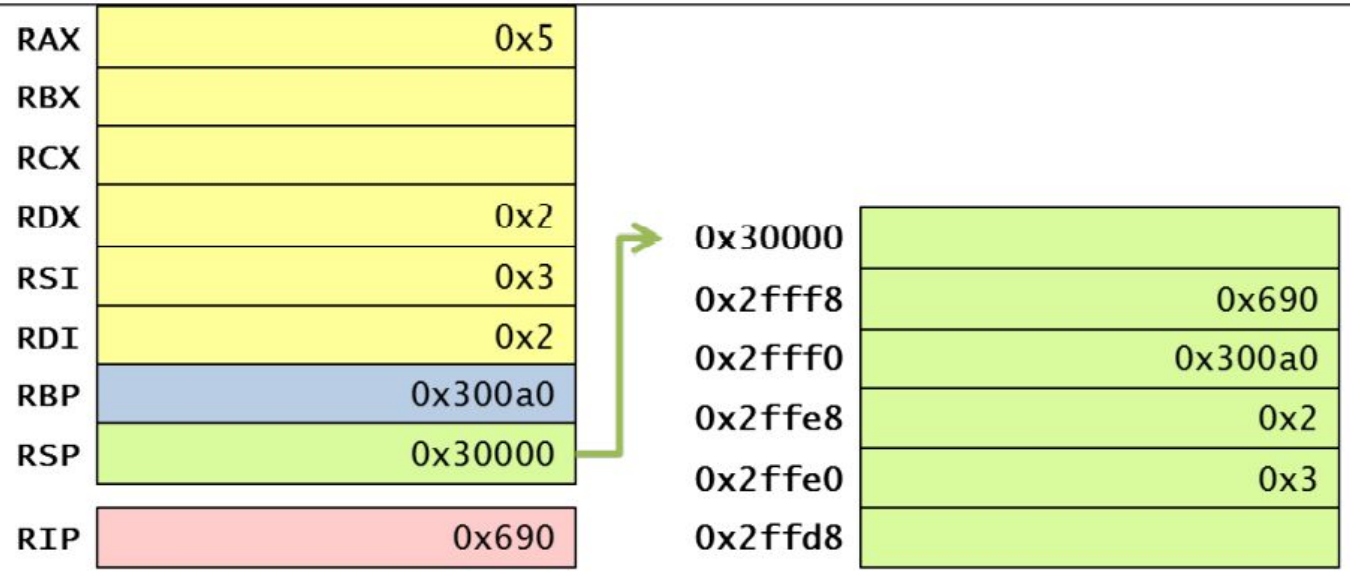
Du C à l'assembleur - Exécution en image - 10

```
1  0000000000000660 <addition>:
2    660:   55                      push   %rbp
3    661:   48 89 e5                mov    %rsp,%rbp
4    664:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
5    668:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
6    66c:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
7    670:   48 8b 45 f0             mov    -0x10(%rbp),%rax
8    674:   48 01 d0                add    %rdx,%rax
9    677:   5d                      pop    %rbp
10   678:   c3                      retq
```



Du C à l'assembleur - Exécution en image - 11

```
1   679:   55                    push   %rbp
2   67a:   48 89 e5              mov    %rsp,%rbp
3   67d:   48 83 ec 10           sub    $0x10,%rsp
4   681:   be 03 00 00 00        mov    $0x3,%esi
5   686:   bf 02 00 00 00        mov    $0x2,%edi
6   68b:   e8 d0 ff ff ff        callq  660 <addition>
7   690:   48 89 45 f8           mov    %rax,-0x8(%rbp)
8   694:   b8 00 00 00 00        mov    $0x0,%eax
9   699:   c9                    leaveq
10  69a:   c3                    retq
```



Outils d’analyse

- Outils sous Linux
  - gdb
  - strace
  - objdump
  - gcc -S
  - Appel système ptrace()
  - Radare - http://www.radare.org/
  - IDA - https://www.hex-rays.com/products/ida/

gdb

- Exécution de gdb
  - 1- Compiler avec gcc et l'option -g
  - 2- La commande gdb –args a.out param1 param2... Permet d'analyser le binaire a.out lancé avec plusieurs paramètres

- Commandes à lancer une fois dans GDB
  - help [commande] : Affiche l'aide
  - b main : Place un point d'arrêt pour stopper le programme à la fonction main()
  - r : Lance le programme dans le débogueur
  - l : Affiche le code C (si le programme a été compilé avec l'option –g)
  - s : Exécute le programme pas à pas
  - n : Exécution pas à pas mais sans rentrer dans les fonctions
  - p variable : Affiche le contenu de la variable variable
  - set variable=valeur : Affecte une valeur à une variable
  - x /8xb address : Affiche 8 octets en hexadécimal à partir de l'adresse donnée
  - disas : Désassemble
  - si : Exécute le programme assembleur pas à pas
  - ni : Exécute le programme assembleur pas à pas sans rentrer dans les fonctions

strace

- strace commande
  - Sous Linux, permet de tracer les appels systèmes

objdump

- objdump -d
  - Affiche les instructions de la section de code

- objdump -s -j .data -j .rodata
  - Affiche le contenu des sections .data et .rodata

Instructions d’accès à la mémoire

- mov valeur, destination
- lea adresse, destination
- push valeur
- pop destination

Instruction LEA

- **lea adresse, destination**
  - Calcul et copie d'une adresse

- Exemples
  - lea 8(%rbp), %rbx : Copie dans le registre RBP l'adresse dont la valeur est celle de RBP + 8

Différence entre LEA et MOV

- Attention à la différence entre les instructions MOV et LEA
  - **mov 8(%rbx), %rax** : Copie dans le registre RAX la valeur située en mémoire, 8 octets après l'adresse pointée par RBX
  - **lea 8(%rbx), %rax** : Copie dans le registre RAX l'adresse dont la valeur est celle de RBX + 8

Différence entre LEA et MOV

`mov 8(%rbx), %rax`	Copie la valeur 0x9ABC dans le registre RAX
`lea 8(%rbx), %rax`	Ecrit l'adresse 0x2008 dans le registre RAX

RAX	0
RBX	0x2000

0x2018	0x1234
0x2010	0x5678
0x2008	0x9ABC
0x2000	0xDEF0

Instructions arithmétiques et binaires

- **inc** destination
- **dec** destination
- **add** valeur, destination
- **sub** valeur, destination
- **xor** valeur, destination
- **and** valeur, destination

Instructions de saut

- **jmp offset**
  - L’offset permet de faire un saut relatif par rapport à la position courante
  - Saut à une adresse absolue rarement utilisé

Instructions de saut conditionnel

- **cmp valeur1, valeur2** , résultat stocké dans RFLAGS
- **test valeur1, valeur2** , test du ET bit à bit

- Saut en fonction du résultat de la dernière comparaison
  - je adresse : egalité
  - jne adresse : inégalité
  - jg adresse : strictement supérieur
  - jge adresse : supérieur ou égal
  - jl adresse : strictement inférieur
  - jle adresse : inférieur ou égal
  - jnz adresse : non nul
  - etc…

Instructions de saut conditionnel

```
1    cmpl $1, %eax     // Compare eax et 1
2    je .L1            // Si eax == 1, saut en .L1
3    xorl %ebx, %ebx   // Mise a 0 de ebx
4    jmp .L2           // Saut en .L2
5  .L1:
6    movl $42, %ebx    // Met 42 dans ebx
7  .L2:
```

Le code assembleur ci-dessus correspond au pseudo-code suivant

```
1  if (eax == 1)
2    ebx = 42;
3  else
4    ebx = 0;
```

Instructions d’appel de fonction et d’appel système

- **call adresse** et **ret**
- **lcall segment, adresse** et **lret**
- **int numéro d’interruption** et **iret**
- **sysenter** et **sysexit**
- **syscall** et **sysret**

Les notions essentielles

- Instructions, opérandes
- Registres
  - Pointeur d'instruction
  - Pointeur de pile
  - Registre de statut
- Accès aux données en mémoire
- Appel de fonctions
  - Utilisation de la pile
  - Variables locales
  - Convention pour le passage de paramètres