Programmation 1, leçon 2

Jean Goubault-Larrecq
ENS Cachan
goubault@lsv.ens-cachan.fr

24 septembre 2013

1 Architecture et assembleur

Nous allons maintenant regarder un peu plus en profondeur encore comment l'exécution d'un programme se passe sur une machine, en l'occurrence une machine à processeur Pentium 32 bits et tournant sous Linux. (Je dirai aussi ce qui change sur les machines 64 bits.)

Ce n'est pas seulement parce qu'il est intéressant de savoir comment ça se passe « en vrai », mais aussi parce que certaines notions, comme celles de *pointeurs* en C, sont plus faciles à comprendre lorsqu'on sait comment ça marche, et typiquement lorsque l'on a compris comment une mémoire est organisée, et ce que sont les adresses mémoire.

Reprenons l'exemple du programme `mycat` que nous avons déjà étudié en leçon 1. L'utilitaire `ddd` nous a permis de voir le programme s'exécuter, pas à pas. Maintenant, lorsque `ddd` se lance, cliquez le menu "Source/Display Machine Code". Ceci vous montrera un cadre ressemblant à celui de la figure 1. Vous pouvez exécuter le code en pas à pas stop en cliquant sur le bouton "Next", ce qui vous fera avancer d'une instruction C à la fois, soit en cliquant sur le bouton "Nexti", ce qui vous fera avancer d'une instruction assembleur à la fois. Notez qu'une instruction C correspond en général à plusieurs instructions assembleur.

Vous pouvez aussi voir le contenu des principaux registres du Pentium en cliquant sur le menu "Data/Status Displays...", puis sur la case "List of integer registers and their contents" (pas sur "List of all registers..."). Les plus importants seront `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, `%esp`, et `%eip`. Ce dernier est le *compteur de programme* (extended instruction pointer). Le registre `%esp` est le pointeur de pile, et `%ebp` est couramment utilisé pour sauvegarder la valeur de `%esp` en entrée de fonction. Les autres sont des registres à usage général. Tous contiennent des entiers 32 bits. Cette organisation des registres, ainsi que les instructions assembleur particulères que vous verrez sous `ddd` sont particulières au Pentium (en version 32 bits), mais le principe de l'assembleur est grosso modo le même sur toutes les machines.

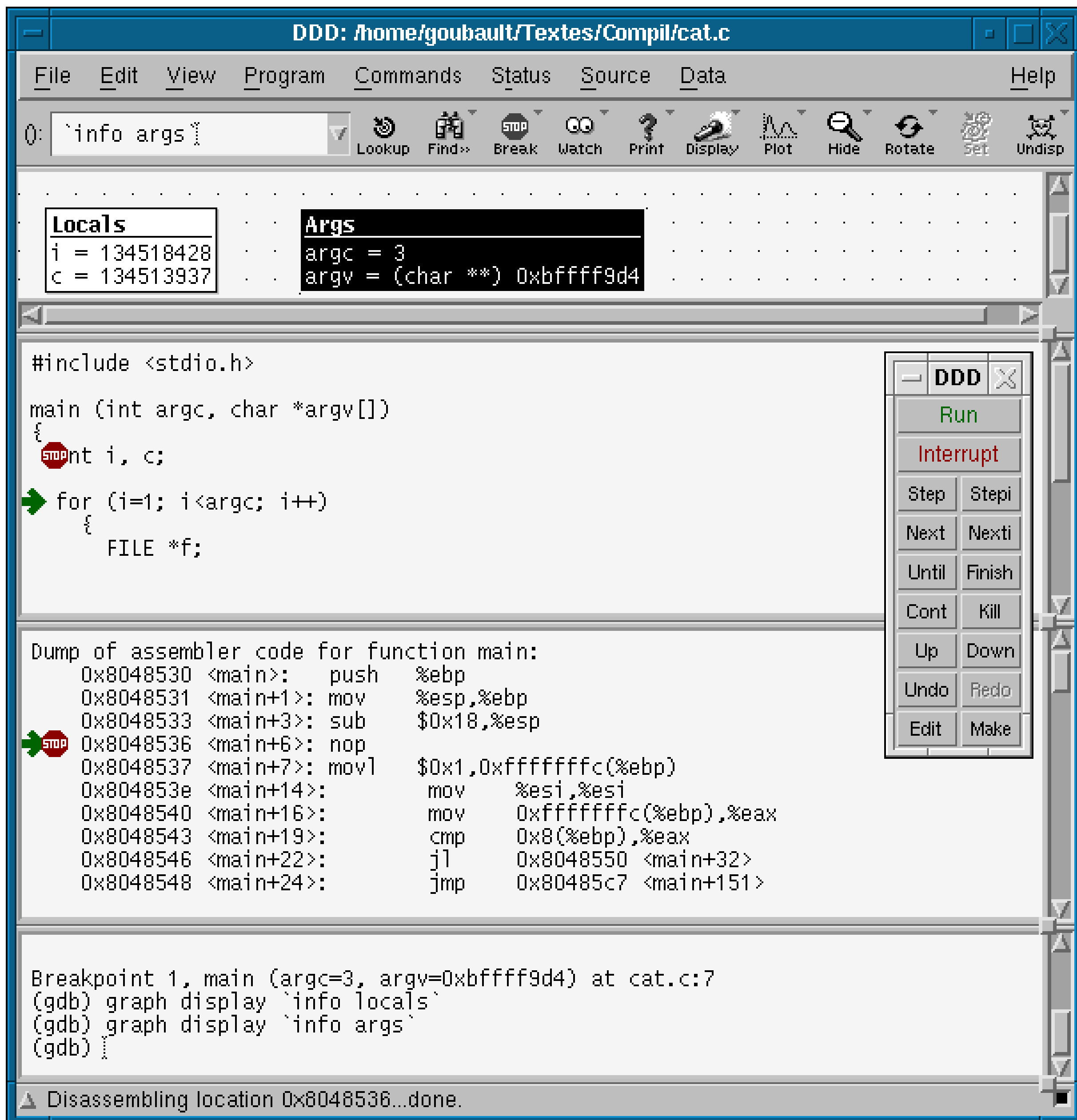
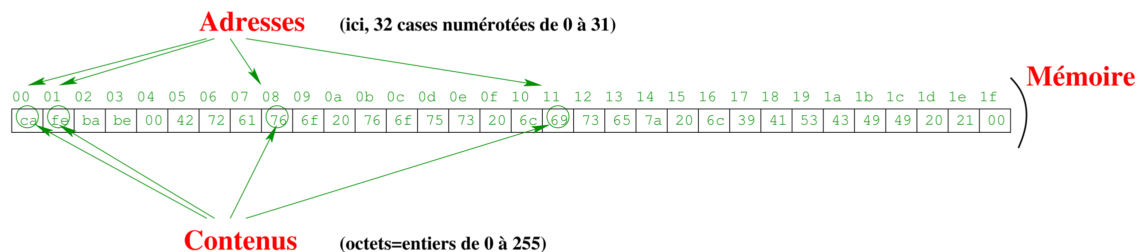


FIGURE 1 – Une session sous `ddd`, montrant ensemble code assembleur et code C

1.1 Mémoires

Dans un ordinateur, on trouve d'abord une *mémoire*. Il s'agit d'un gigantesque tableau de cases contenant des *octets*, c'est-à-dire des nombres à 8 chiffres en base 2. (Un chiffre en base 2 est 0 ou 1, et est traditionnellement appelé un *bit*.) Les indices de ce tableau sont des nombres, typiquement de 0 à $2^{32} - 1 = 4\ 294\ 967\ 295$ sur le Pentium 32 bits, qui code ces indices sur… 32 bits. (Sur les machines 64 bits, c'est bien sûr de 0 à $2^{64} - 1 = 18\ 446\ 744\ 073\ 709\ 551\ 616$.) Les indices de ce tableau sont traditionnellement appelés les *adresses*.



Octets :

Hexa	Dec.	Binaire	ASCII
`ca`	202	11001010	Ê
`fe`	254	11111110	þ
`76`	118	01110110	v
`69`	105	01101001	i

Electronique :

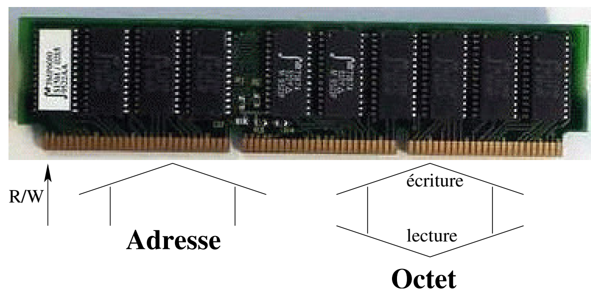


FIGURE 2 – La mémoire des ordinateurs

On peut voir en figure 2 un exemple de mémoire d'un ordinateur. Comme le montre le tableau du haut de la figure 2, une mémoire n'est donc rien d'autre qu'une fonction qui à chaque adresse associe un contenu, qui est un octet. Il est traditionnel de compter les adresses et de lire les octets en *hexadécimal*, c'est-à-dire en base 16 ; la lettre `a` vaut 10, `b` vaut 11, …, `f` vaut 15. Ainsi les adresses sont $0, 1, \ldots, 8, 9, a, b, c, d, e, f, 10, 11, \ldots, 1a, 1b, \ldots, 1f, 20, \ldots$ Lorsqu'il y aura ambiguïté dans la base, on utilisera la convention du langage C de faire précéder une suite de caractères parmi `0`, …, `9`, `a`, …, `f` des caractères `0x` pour montrer que le rester est un nombre hexadécimal. Par exemple, `0x10` dénote 16 ; ou bien `0xcafe` vaut $12 \times 256^3 + 10 \times 256^2 + 15 \times 256 + 14 = 51\ 966$.

La mémoire du haut de la figure 2 contient donc l'octet `0xca`$= 12 \times 16 + 10 = 202$ à l'adresse `0x0`$= 0$, l'octet `0xfe`$= 15 \times 16 + 14 = 254$ à l'adresse `0x1`$= 1$, et ainsi de suite.

Électriquement, une mémoire est un circuit ou un groupe de circuits ressemblant à celui en bas à droite de la figure 2. Les adresses sont envoyées sous forme de signaux électriques (par exemple, si le bit i de l'adresse vaut 1 alors on relie le contact électrique no. i au +5V, sinon à la masse). Un autre contact, `R/W`, est positionné à 1 si l'on souhaite lire l'adresse ainsi décrite,

auquel cas le contenu de l'octet à cette adresse sera présent en binaire sur huit autres contacts. Si `R/W` est positionné à 0, alors le circuit de mémoire s'attend en revanche à ce que la donnée à inscrire à l'adresse précisée soit déjà présente sur les huit contacts ci-dessus. Ceci est le principe général, il y a en fait d'autres contacts, et d'autres conventions (par exemple, le +5V peut en fait coder le 0 au lieu du 1, la tension peut ne pas être +5V, etc.)

Il est d'autre part important que tous les octets sont utilisés comme *codages*. On l'a dit en fin de leçon 1, et on le redit. Par exemple, l'octet `0xca` peut être vu comme :

- l'entier positif ou nul $12 \times 16 + 10 = 202$, comme ci-dessus (on dit que c'est un entier *non signé* ;
- ou bien l'entier compris entre $-2^7 = -128$ et $2^7 - 1 = 127$, et égal au précédent modulo $2^8 = 256$: ceci permet de représenter des entiers positifs ou négatifs dans un certain intervalle ; selon cette convention, `0xca` vaut alors $202 - 256 = -54$, et l'on parle d'*entier signé en complément à deux* ;
- ou bien le code du caractère "`Ê`" (dans la table ISO latin-1) : il s'agit d'une pure convention ;
- ou bien l'instruction assembleur `lret` ("long return") du processeur Pentium ;
- ou bien encore pas mal d'autres possibilités...

Tout dépend de ce que l'on fait avec l'octet en question : des opérations arithmétiques sur entiers signés, sur entiers non signés, des opérations de manipulations de chaînes de caractères, exécuter un programme, etc.

1.2 Le processeur

Le processeur est un automate, dont l'état interne est donné par le contenu de ses registres : on considérera que ceux du Pentium 32 bits sont `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, `%esp`, et `%eip`. Ce dernier est le compteur de programme.

À tout moment, le processeur est sur le point d'exécuter une instruction du *langage machine*. Il s'agit de l'instruction que l'on trouve en mémoire à l'adresse dont la valeur est le contenu du registre `%eip`. Dans le cas de la figure 3, le contenu du registre `%eip` et l'entier `0x08048530`, et l'on trouve en mémoire à cette adresse l'octet `0x55`, qui est le code de l'instruction `pushl %ebp` du Pentium.

Le processeur est cadencé par un oscillateur, qui s'appelle l'*horloge*, et en quelque sorte bat la mesure : à chaque tic d'horloge (à peu de choses près), le processeur va chercher l'instruction suivante, l'exécute, puis attend le prochain tic d'horloge.

Dans l'exemple de la figure 3, au prochain tic d'horloge, le processeur va exécuter l'instruction `pushl %ebp`, qui a pour effet d'empiler ("push") la valeur du registre `%ebp` (ici `0xbffffa48`) au sommet de la pile. Quelle pile ? Eh bien, le Pentium, comme beaucoup d'autres processeurs, maintient dans le registre `%esp` ("extended stack pointer") l'adresse du sommet d'une pile. Dans l'exemple, le sommet de la pile est donc en ce moment à l'adresse `0x18`, et empiler `%ebp` a pour effet de stocker sa valeur `0xbffffa48` aux adresses `0x17`, `0x16`, `0x15`, `0x14`, et de bouger le pointeur de pile jusqu'en `0x14`. Le résultat est montré en figure 4. Les valeurs ayant changé sont en rouge. À noter qu'après l'exécution de l'instruction, le compteur de programme `%eip` est incrémenté et pointe vers l'instruction suivante.

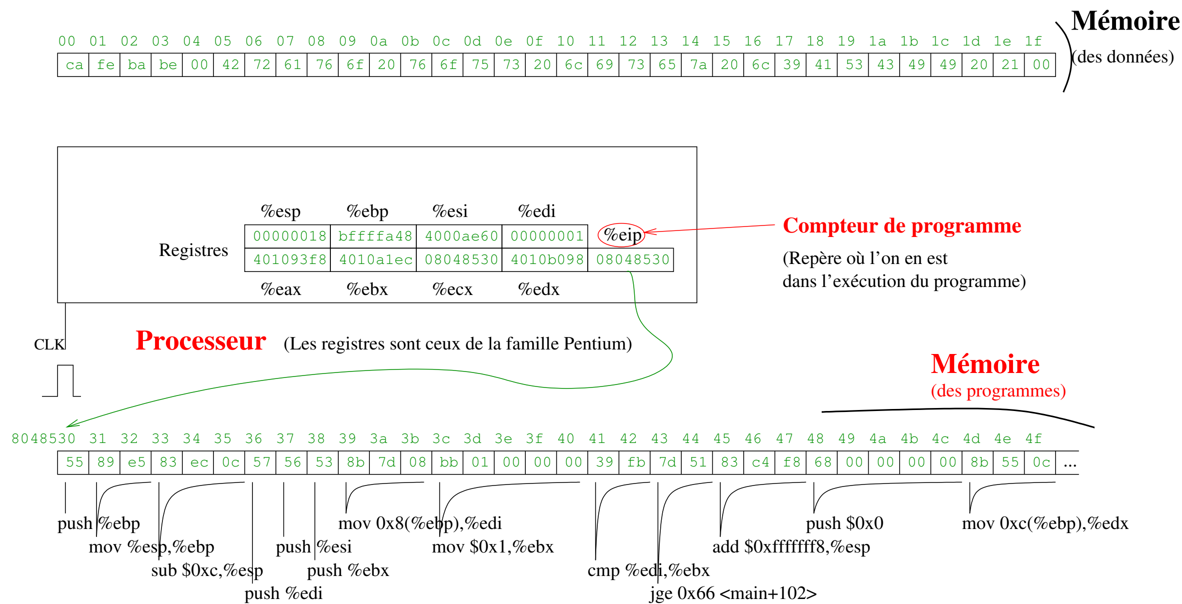


FIGURE 3 – Le processeur et la mémoire

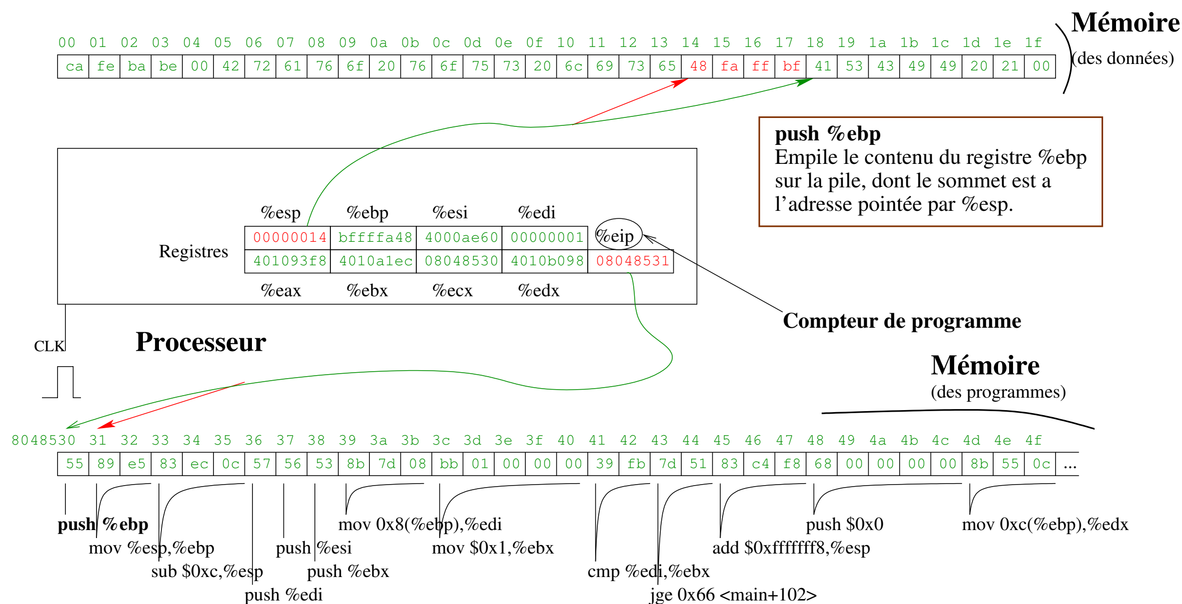


FIGURE 4 – Exécution de la première instruction

L'instruction suivante est `movl %esp, %ebp`. La sémantique de cette instruction est de recopier (`movl`="move", déplacer) le contenu du registre `%esp` dans le registre `%ebp`. Au total, les deux premières instructions de ce programme ont sauvegardé le contenu précédent du registre `%ebp` sur la pile, puis ont mis l'adresse du sommet de la pile dans `%ebp`, pour pouvoir s'y référer dans la suite.

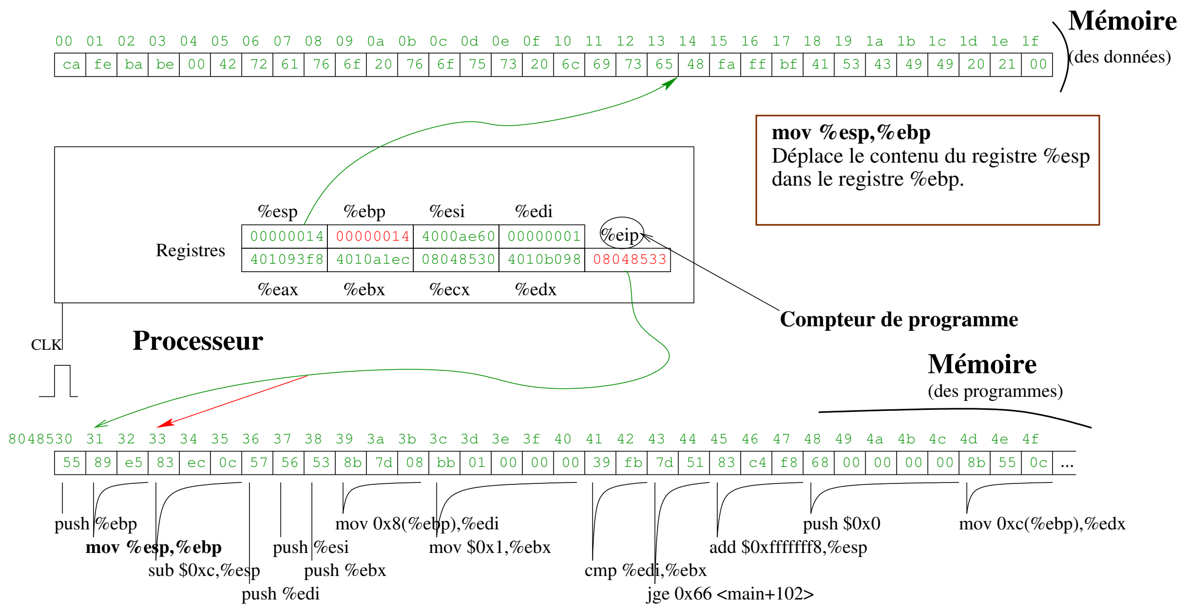
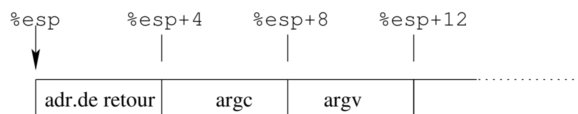


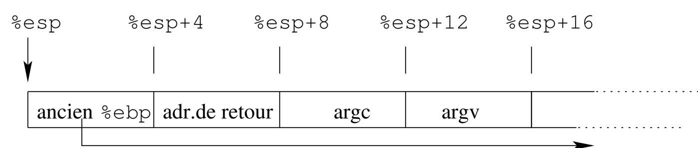
FIGURE 5 – Exécution de la deuxième instruction

La raison de ce mouvement est que, à l'entrée de la fonction `main` (voir le source C en figure 6), la pile ressemble à ceci :



Le paramètre d'entrée `argc` est donc stocké 4 octets plus loin que le sommet de pile, sur 4 octets (32 bits), et le second paramètre d'entrée `argv` est stocké 8 octets au-delà du pointeur de pile, sur 4 octets aussi. Les 4 octets à partir du sommet de pile stockent l'*adresse de retour*, c'est-à-dire la valeur du compteur de programme où il faudra reprendre l'exécution lorsque l'exécution de la fonction `main` sera terminée.

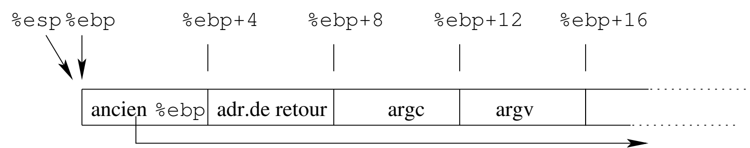
Après l'instruction `pushl %ebp`, la pile ressemble à :



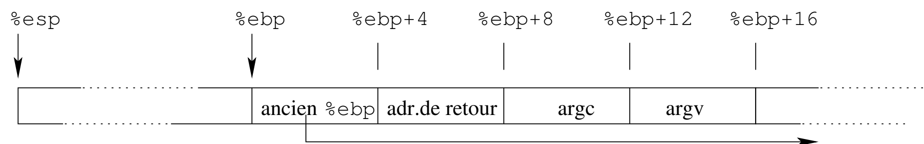
Puis, après l'instruction `movl %esp, %ebp`, elle est de la forme :

```
#include <stdio.h>

main (int argc, char *argv[])
{
  int i, c;

  for (i=1; i<argc; i++)
    {
      FILE *f;

      f = fopen (argv[i], "r");
      while ((c = fgetc (f))!=EOF)
        fputc (c, stdout);
      fclose (f);
    }
  fflush (stdout);
  exit (0);
}
```

FIGURE 6 – Un programme de concaténation en C



L'instruction `subl $0xc, %esp` a pour but de retirer `0xc`$= 12$ du registre `%esp`. En d'autre termes, ceci réserve 12 octets sur la pile. On a donc une pile de la forme :



Concrètement, la situation est comme montrée sur la figure 7. Au lieu de soustraire 12 octets, on aurait pu en ajouter -12, ce qui se serait fait avec l'instruction `addl $-12, %esp`, ou de façon équivalente, `addl $0xfffffff4, %esp`.

(Donc, que fait l'instruction `addl $0xfffffff8, %esp` un peu plus loin dans le programme ?)

L'intérêt de l'allocation des 12 octets avant `%ebp` est de permettre de réserver de la place pour les trois variables `i`, `c` et `f` de la fonction `main`, typiquement.

Lorsqu'on arrivera à la fin de la fonction `main`, on y trouvera les trois instructions :

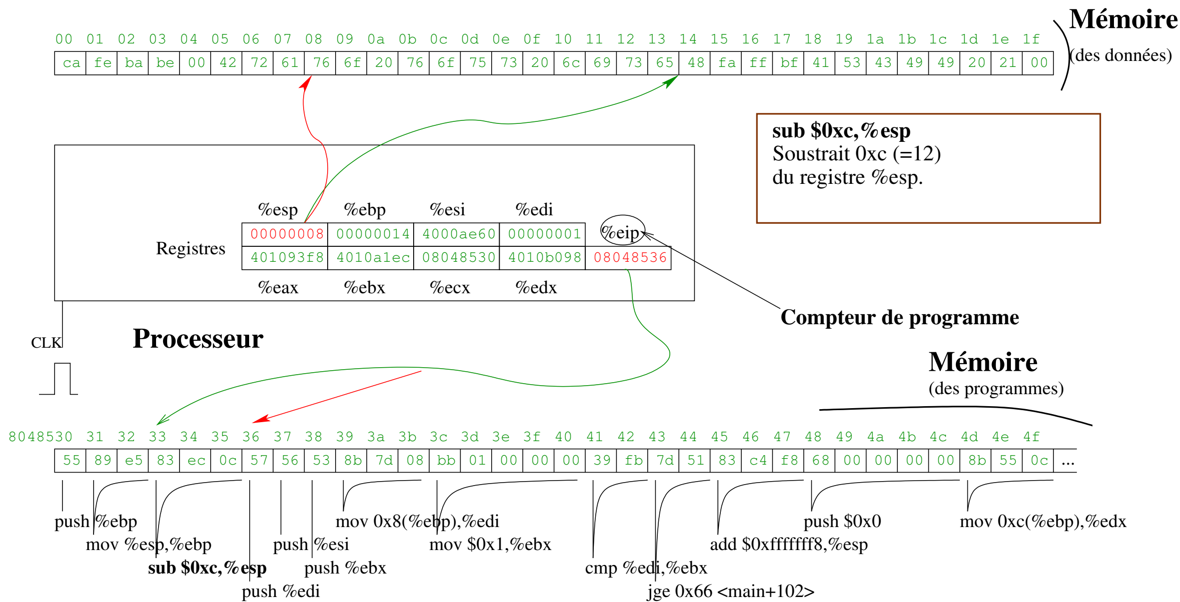
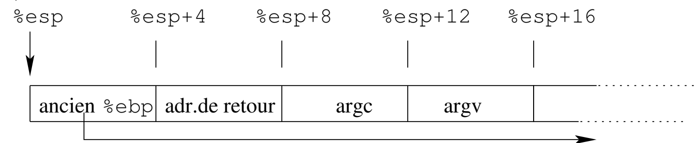
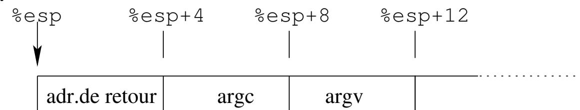


FIGURE 7 – Exécution de la troisième instruction

- `movl %ebp, %esp`, qui recopie le contenu du registre `%ebp` dans le registre `%esp`. Ceci a simplement pour effet de dépiler d'un coup tout ce qui avait été empilé au cours de la fonction `main`, et l'on revient à la situation :



- `popl %ebp`, l'instruction symétrique de `pushl %ebp`, qui dépile le sommet courant de la pile, et met la donnée 32 bits à l'ancien sommet de pile dans le registre `%ebp`. On revient à la situation :



- `ret`, qui retourne au programme appelant. Ceci dépile l'adresse au sommet de la pile, et s'y branche (autrement dit, met cette adresse dans le registre `%eip` de compteur de programme).

▸ **EXERCICE 1.1**

À l'aide de `ddd`, exécutez en pas à pas, avec l'instruction "Nexti", le programme `mycat` de la figure 6. Observez bien les contenus des registres. Vous pouvez aussi observer le contenu de la pile : regardez la valeur de `%esp` dans la liste des registres à l'entrée de la fonction `main`, disons `0xbffff8e0`, soustrayez disons 32 octets, ce qui donne `0xbffff9c0`, puis allez dans le menu "Data/Memory...", et demandez "View" "48" "hex" "words" "from : 0xbffff9c0" (remplacez par l'adresse réelle calculée plus haut). Déduisez-en ce que fait chaque instruction, si possible. Comparez avec la table de l'annexe A.

1.3 Modes d'adressage et formats d'instructions

Observons la différence entre les instructions `movl %esp, %ebp` et `subl $0xc, %esp`. Dans le premier cas, on recopie le contenu du registre `%esp` dans un autre registre : l'instruction `movl` recopie une donnée *source* vers une *destination*. Dans le second cas, on soustrait une donnée *source* du contenu de la *destination*.

Mais surtout, dans le premier cas, la source est un *registre*, à savoir le registre de sommet de pile `%esp`, alors que dans le second cas, il s'agit d'une constante (dite *immédiate*, parce que donnée, là, dans le code exécuté). Les modes typiques d'adressage sont :

– *immédiat* : la constante est donnée directement. La syntaxe correspondante dans l'assembleur Pentium consiste à faire précéder la constante par le signe `$`. Par exemple, `movl $0x14, %eax` met la constante `0x14`= 20 dans le registre `%eax`.
– *registre* : la donnée est dans un registre. Par exemple, `movl %eax, %ebx` recopie le contenu du registre `%eax` dans le registre `%ebx`.
– *absolu* : la donnée est à l'adresse donnée directement dans le code. À titre d'exemple, `movl 0x8f90ea48, %eax` recopie l'entier 32 bits stocké aux adresses `0x8f90ea48` à `0x8f90ea4b` (4 octets) dans le registre `%eax`.
– *indirect* : la donnée est à l'adresse que l'on trouve dans le registre spécifié. Par exemple, `movl (%ebx), %eax` récupère le contenu de `%ebx`, disons `0x8f90ea48`, puis lit l'entier 32 bits à cette adresse, c'est-à-dire de`0x8f90ea48` à `0x8f90ea4b` (4 octets), et le stocke dans le registre `%eax`.

Il y a quelques variantes de ces modes d'adressage. Dans le cas du mode absolu, il arrive que l'adresse ne soit pas stockée directement dans le code, mais sous forme d'un décalage entre la valeur courante de `%eip` (le compteur de programme) et l'adresse souhaitée : c'est l'adressage *relatif*, typiquement utilisé dans les instructions de saut (voir plus loin).

Le mode indirect connait aussi un certain nombre de déclinaisons de plus en plus raffinées. Par exemple, `movl 0x8(%ebx), %eax` ajoute d'abord l'*offset* (décalage) `0x8`= 8 à l'adresse stockée dans `%ebx` avant de récupérer la donnée qui s'y trouve. Dans l'exemple traité en section 1.2, `movl 0x8(%ebp), %eax` permet ainsi de récupérer la valeur de l'argument `argc`, une fois le prologue `pushl %ebp`, `movl %esp, %ebp` effectué ; de même, l'instruction `movl %0xc(%ebp), %eax` permet de récupérer (l'adresse) du tableau `argv` dans `%eax`. Le Pentium dispose d'une profusion de tels modes : `movl (%eax, %ebx, 4), %ecx` récupère dans `%ecx` le contenu de la mémoire aux 4 octets à partir de l'adresse obtenue en additionnant le contenu de `%eax` avec le contenu de `%ebx` multiplié par 4. . .

On a illustré ci-dessus l'utilisation des modes d'adressage dans le cas de la source. La destination peut être elle aussi spécifiée en mode registre, absolu, ou indirect. Elle ne peut pas être donnée en mode immédiat : on ne peut pas modifier la valeur d'une constante !

En général, toutes les combinaisons de modes d'adressage pour la source et la destination ne sont pas possibles. En théorie, l'idéal est de se référer à la documentation du processeur considéré, mais celle du Pentium est gigantesque. Il sera plus simple dans notre cas de compiler (avec `gcc`) quelques programmes assembleur simples : si `gcc` refuse, c'est que l'instruction n'existe pas.

Tous les processeurs disposent, en plus des instructions de déplacement (`movl` par exemple)

et des instructions arithmétiques et logiques (`addl`, `subl`, le et bit à bit `andl`, etc.), des instructions *de saut*. Sur le Pentium, l'instruction `jmp` ⟨dest⟩ détourne le contenu du programme vers l'adresse ⟨dest⟩ :

– `jmp 0x8f90ea48` se branche à l'adresse `0x8f90ea48` (adressage absolu), autrement dit met l'entier `0x8f90ea48` dans le registre `%eip`.
– `jmp *%eax` se branche à l'adresse qui est stockée dans `%eax` (adressage indirect). Autrement dit, ceci recopie `%eax` dans `%eip`. (On notera une irrégularité de syntaxe ici : on n'écrit pas `jmp (%eax)`, même si cela aurait été plus cohérent.)

▸ **EXERCICE 1.2**

L'instruction `leal` ("load effective address") ressemble à `movl`. Mais là où `movl` ⟨source⟩, ⟨dest⟩ recopie ⟨source⟩ dans ⟨dest⟩, `leal` ⟨source⟩, ⟨dest⟩ recopie l'*adresse* où est stocké ⟨source⟩ dans ⟨dest⟩. Donc, par exemple, `leal 0x8f90ea48, %eax` (adressage absolu) est équivalent à `movl $0x8f90ea48, %eax` (adressage immédiat), et `leal 4(%ebx), %eax` met le contenu de `%ebx` plus 4 dans `%eax`, au lieu de lire ce qui est stocké à l'adresse qui vaut le contenu de `%ebx` plus 4. Que fait `leal (%eax, %esi, 4), %ebx` ? Pourquoi les modes d'adressage immédiat et registre n'ont-ils aucun sens pour la source de `leal` ? L'instruction `jmp` ⟨source⟩ est-elle équivalente à `movl` ⟨source⟩`, %eip` ou bien à `leal` ⟨source⟩`, %eip` ? (À ceci près que ces deux dernières instructions n'existent pas, car il est impossible d'utiliser le registre `%eip` comme argument d'aucune instruction...)

Pour effectuer des tests (les `if` de C ou de Caml), il suffit d'utiliser une combinaison de deux instructions, `cmpl` et une instruction de la famille des *sauts conditionnels*. Par exemple, l'instruction C

```
if (i==j)
  a;
else b;
```

correspondra au code assembleur suivant, en supposant que `i` est dans le registre `%eax` et `j` dans le registre `%ebx` :

```
    cmpl %ebx, %eax      ; comparer i (%eax) avec j (%ebx)
    jne _b               ; si pas égaux (not equal), aller exécuter b,
                         ; à l'adresse _b.
    ... code de a ...    ; sinon exécuter a
    jmp _suite           ; puis passer à la suite du code, après le test.
_b:
    ... code de b ...
_suite:
    ... suite du code ...
```

On constate que `jne` branche à l'adresse (ici _b) en argument si `i` n'est pas égal à `j`, et continue à l'instruction juste après le `jne` sinon.

A contrario, `je` brancherait à l'adresse donnée en argument si `i` était égal à `j` ; `jl` brancherait si `i` était strictement inférieur (less) que `j` en tant qu'entier signé, `jge` si plus grand ou

égal (greater than or equal to), `jg` si plus grand strictement (greater), `jle` si plus petit ou égal (less than or equal to) ; `jb` (below), `jnb` (not below), `ja` (above), et `jna` (not above) sont les instructions correspondantes lorsque `i` et `j` sont des entiers *non* signés.

▸ **EXERCICE 1.3**
L'instruction `call` appelle un sous-programme (en C, un sous-programme, c'est une fonction). La seule différence avec l'instruction `jmp`, c'est que `call` empile d'abord le contenu de `%eip`, c'est-à-dire l'adresse qui est juste après l'instruction `call`. Donc `call` ⟨dest⟩, alors que le compteur de programme vaut, disons, N, est équivalent à `pushl $`n suivi de `jmp` ⟨dest⟩. Symétriquement, `ret` serait équivalente à `popl %eip` (si cette instruction existait) : que fait `ret`, concrètement ?

▸ **EXERCICE 1.4**
Montrer que `pushl` ⟨source⟩ est équivalente à `subl $4, %esp` suivi de `movl` ⟨source⟩`, (%esp)`. Proposer un équivalent de `popl` ⟨dest⟩.

▸ **EXERCICE 1.5**
Supposons que `%eax` contient l'entier 4 et `%ebx` contient l'entier -3 (en signé) $= 4\ 294\ 967\ 293$ (en non signé). Noter que le branchement `jle` sera pris après une instruction `cmpl %ebx, %eax`. Qu'en est-il dans les cas de `jg`, `jl`, `jge`, `ja`, `jna`, `jb`, `jnb` ?

Encore une fois, on rappelle que les instructions utiles du Pentium 32 bits sont récapitulées en annexe A. Les changements dans les architectures Intel 64 bits plus récentes sont donnés en annexe B.

A Guide de référence rapide de l'assembleur Pentium 32 bits

Les courageux pourront consulter `http://www.intel.com/design/intarch/techinfo/pentium/instsum.htm` pour une description complète du jeu d'instruction du Pentium. Nous n'aurons besoin dans le cours que de ce qui est décrit dans cette annexe.

Les registres : `%eax %ebx %ecx %edx %esi %edi %ebp %esp`.
Ils contiennent tous un entier de 32 bits (4 octets), qui peut aussi être vu comme une adresse. Le registre `%esp` est spécial, et pointe sur le sommet de pile ; il est modifié par les instructions `pushl`, `popl`, `call`, `ret` notamment.

Il y a aussi d'autres registres que l'on ne peut pas manipuler directement. (L'instruction `info registers` sous `gdb` ou `ddd` vous les montrera.) Le plus important est `%eip`, le *compteur de programme* : il contient en permanence l'adresse de la prochaine instruction à exécuter.

- `addl` ⟨source⟩, ⟨dest⟩ . . . . ⟨dest⟩= ⟨dest⟩+ ⟨source⟩ (addition)
  Ex : `addl $1, %eax` ajoute 1 au registre `%eax`.
  Ex : `addl $4, %esp` dépile un élément de 4 octets de la pile.
  Ex : `addl %eax, (%ebx, %edi, 4)` ajoute le contenu de `%eax` à la case mémoire à l'adresse `%ebx` $+ 4 *$ `%edi`. (Imaginez que `%ebx` est l'adresse de début d'un tableau `a`, `%edi` est un index `i`, ceci stocke `%eax` dans `a[i]`.)

- `andl` ⟨source⟩, ⟨dest⟩ . . . . ⟨dest⟩= ⟨dest⟩ `&` ⟨source⟩ (et bit à bit)
- `call` ⟨dest⟩ . . . . appel de procédure à l'adresse ⟨dest⟩

  Équivalent à `pushl $`a, où a est l'adresse juste après l'instruction `call` (l'adresse *de retour*), suivi de `jmp` ⟨dest⟩.

  Ex : `call printf` appelle la fonction `printf`.

  Ex : `call *%eax` (appel indirect) appelle la fonction dont l'adresse est dans le registre `%eax`. Noter qu'il y a une irrégularité dans la syntaxe, on écrit `call *%eax` et non `call (%eax)`.

- `cltd` . . . . conversion 32 bits → 64 bits

  Convertit le nombre 32 bits dans `%eax` en un nombre sur 64 bits stocké à cheval entre `%edx` et `%eax`.

  Note : `%eax` n'est pas modifié ; `%edx` est mis à 0 si `%eax` est positif ou nul, à -1 sinon.

  À utiliser notamment avant l'instruction `idivl`.

- `cmpl` ⟨source⟩, ⟨dest⟩ . . . . comparaison

  Compare les valeurs de ⟨source⟩ et ⟨dest⟩. Utile juste avant un saut conditionnel (`je`, `jge`, etc.). À noter que la comparaison est faite dans le sens inverse de celui qu'on attendrait. Par exemple, `cmp` ⟨source⟩, ⟨dest⟩ suivi d'un `jge` ("jump if greater than or equal to"), va effectuer le saut si ⟨dest⟩≥⟨source⟩ : on compare ⟨dest⟩ à ⟨source⟩, et non le contraire.

- `idivl` ⟨dest⟩ . . . . division entière et reste

  Divise le nombre 64 bits stocké en `%edx` et `%eax` (cf. `cltd`) par le nombre 32 bits ⟨dest⟩. Retourne le quotient en `%eax`, le reste en `%edx`.

- `imull` ⟨source⟩, ⟨dest⟩ . multiplie ⟨dest⟩ par ⟨source⟩, résultat dans ⟨dest⟩
- `jmp` ⟨dest⟩ . . . . saut inconditionnel : `%eip`=⟨dest⟩
- `je` ⟨dest⟩ . . . . saut conditionnel

  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩=⟨source⟩, continue avec le flot normal du programme sinon.

- `jg` ⟨dest⟩ . . . . saut conditionnel

  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩>⟨source⟩, continue avec le flot normal du programme sinon.

- `jge` ⟨dest⟩ . . . . saut conditionnel

  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩≥⟨source⟩, continue avec le flot normal du programme sinon.

- `jl` ⟨dest⟩ . . . . saut conditionnel

  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩<⟨source⟩, conti-

nue avec le flot normal du programme sinon.

- `jle` ⟨dest⟩ .... saut conditionnel
  Saute à l'adresse ⟨dest⟩ si la comparaison précédente (cf. `cmp`) a conclu que ⟨dest⟩≤⟨source⟩, continue avec le flot normal du programme sinon.

- `leal` ⟨source⟩, ⟨dest⟩ .... chargement d'adresse effective
  Au lieu de charger le contenu de ⟨source⟩ dans ⟨dest⟩, charge l'*adresse* de ⟨source⟩.
  Équivalent C : ⟨dest⟩=&⟨source⟩.

- `movl` ⟨source⟩, ⟨dest⟩ .... transfert
  Met le contenu de ⟨source⟩ dans ⟨dest⟩. Équivalent C : ⟨dest⟩=⟨source⟩.
  Ex : `movl %esp, %ebp` sauvegarde le pointeur de pile `%esp` dans le registre `%ebp`.
  Ex : `movl %eax, 12(%ebp)` stocke le contenu de `%eax` dans les quatre octets commençant à `%ebp` + 12.
  Ex : `movl (%ebx, %edi, 4), %eax` lit le contenu de la case mémoire à l'adresse `%ebx` + 4 ∗ `%edi`, et le met dans `%eax`. (Imaginez que `%ebx` est l'adresse de début d'un tableau `a`, `%edi` est un index `i`, ceci stocke `a[i]` dans `%eax`.)

- `negl` ⟨dest⟩ .... ⟨dest⟩=−⟨dest⟩(opposé)
- `notl` ⟨dest⟩ .... ⟨dest⟩=~⟨dest⟩(non bit à bit)
- `orl` ⟨source⟩, ⟨dest⟩ .... ⟨dest⟩= ⟨dest⟩ | ⟨source⟩ (ou bit à bit)
- `popl` ⟨dest⟩ .... dépilement
  Dépile un entier 32 bits de la pile et le stocke en ⟨dest⟩.
  Équivalent à `movl (%esp),`⟨dest⟩ suivi de `addl $4, %esp`.
  Ex : `popl %ebp` récupère une ancienne valeur de `%ebp` sauvegardée sur la pile, typiquement, par `pushl`.

- `pushl` ⟨source⟩ .... empilement
  Empile l'entier 32 bits ⟨source⟩ au sommet de la pile.
  Équivalent à `movl` ⟨source⟩, `-4(%esp)` suivi de `subl $4, %esp`.
  Ex : `pushl %ebp` sauvegarde la valeur de `%ebp`, qui sera rechargée plus tard par `popl`.
  Ex : `pushl` ⟨source⟩ permet aussi d'empiler les arguments successifs d'une fonction. (Note : pour appeler une fonction C comme `printf` par exemple, il faut empiler les arguments en commençant par celui de droite.)

- `ret` .... retour de procédure
  Dépile une adresse de retour a, et s'y branche. Lorsque la pile est remise dans l'état à l'entrée d'une procédure f, ceci a pour effet de retourner de f et de continuer l'exécution de la procédure appelante.
  Équivalent à `popl %eip`... si cette instruction existait (il n'y a pas de mode d'adressage permettant de manipuler `%eip` directement).

- `subl` ⟨source⟩, ⟨dest⟩ . . . . ⟨dest⟩= ⟨dest⟩− ⟨source⟩(soustraction)

  Ex : `subl $1, %eax` retire 1 du registre `%eax`.

  Ex : `subl $4, %esp` alloue de la place pour un nouvel élément de 4 octets dans la pile.

- `xorl` ⟨source⟩, ⟨dest⟩ . . . . . ⟨dest⟩= ⟨dest⟩ ^ ⟨source⟩ (ou exclusif bit à bit)

La gestion de la pile standard est la suivante. En entrée d'une fonction, on trouve en `(%esp)` l'adresse de retour, en `4(%esp)` le premier paramètre, en `8(%esp)` le deuxième, et ainsi de suite. La fonction doit commencer par exécuter `pushl %ebp` puis `movl %esp, %ebp` pour sauvegarder `%ebp` et récupérer l'adresse de base dans `%ebp`. A ce stade et dans tout le reste de la fonction l'adresse de retour sera en `4(%ebp)`, le premier paramètre en `8(%ebp)`, le second en `12(%ebp)`, etc.

La fonction empile et dépile ensuite à volonté, les variables locales et les paramètres étant repérées par rapport à `%ebp`, qui ne bouge pas.

A la fin de la fonction, la fonction rétablit le pointeur de pile et `%ebp` en exécutant `movl %ebp, %esp` puis `popl %ebp` (et enfin `ret`).

B L'assembleur Intel/AMD 64 bits

Par rapport à l'assembleur Pentium 32 bits, les différences principales sont :

- Les registres contiennent 64 bits (8 octets), et non plus 32 (pour dire l'évidence) ;
- Il y a davantage de registres : `%rax %rbx %rcx %rdx %rsi %rdi %rbp %rsp` qui sont les équivalents des 8 registres 32 bits (et qui contiennent en fait ces derniers dans leurs 32 bits de poids faible), plus 8 autres registres nommés `%r8` à `%r15`.
- Les instructions 64 bits ont un `q` au lieu d'un `l` à la fin de leur mnémonique. On écrira donc `xorq %rax, %rbx` pour effectuer un ou exclusif 64 bits, plutôt que `xorl %eax, %ebx`, qui n'opère que sur 32 bits.

  Les instructions `call`, `ret`, `jmp` sont aussi remplacées par `callq`, `retq`, `jmpq` respectivement. Les instructions de saut conditionnel (`jg`, `jge`, `jl`, `jle`) restent inchangées.
- Les instructions `cltd` et `idivl` (division 32 bits) sont remplacées par `cqto` et `idivq`. Explicitement :
  - `cqto` effectue une conversion 64 bits → 128 bits, et convertit le nombre 64 bits dans `%rax` en un nombre sur 128 bits stockés à cheval entre `rdx` et `rax` (`rax` n'est pas modifié, comme avec `cltd`) ;
  - `idivq` ⟨dest⟩divise le nombre 128 bits stocké en `%rdx` et `%rax` par le nombre 64 bits ⟨dest⟩, retourne le quotient en `%rax` et le reste en `%rdx`.
- La convention d'appel des fonctions change… ce n'est pas à proprement parler une caractéristique du processeur, plutôt du système d'exploitation, mais tous les systèmes obéissent aux recommandations du constructeur (`http://www.x86-64.org/documentation/abi.pdf`).

  Donc, au lieu d'empiler tous les paramètres d'une fonction sur la pile, le premier étant le plus près du sommet de pile, les processeurs Intel/AMD 64 bits s'attendent à ce que les

paramètres soient passés comme suit :
- Paramètre 1 : dans le registre `%rdi` (pas sur la pile !)
- Paramètre 2 : dans le registre `%rsi`
- Paramètre 3 : dans le registre `%rdx`
- Paramètre 4 : dans le registre `%rcx`
- Paramètre 5 : dans le registre `%r8`
- Paramètre 6 : dans le registre `%r9`
- Les autres paramètres, s'il y en a : sur la pile, le paramètre 7 étant le plus proche du sommet de pile.

Je simplifie outrageusement ici, et l'explication ci-dessus n'est valide que dans le cas de programmes qui ne manipulent que des entiers ou des adresses (pas de nombres à virgule flottante par exemple).

Lors d'un appel de fonction, les seuls registres qui sont garantis revenir inchangés après le retour de la fonction sont `%rsp %rbp %rbx %r12 %r15`. Tous les autres doivent être sauvegardés (typiquement, empilés sur la pile avant appel et dépilés après) par la fonction appelante. Réciproquement, toute fonction doit sauvegarder (sur la pile) les valeurs des registres `%rbp %rbx %r12 %r15` s'il est prévu qu'elle les modifie lors de son exécution.

Finalement, une petite subtilité : toute fonction qui en appelle d'autres devra sauvegarder sur la pile les registres `%rdi %rsi %rdx %rcx %r8 %r9` avant de faire un appel, si elle souhaite pouvoir réaccéder à ses propres paramètres. Mais elle n'est pas obligé de les sauvegarder tous : il suffit de sauvegarder les $\min(n, 6)$ premiers de la liste, où n est le nombre maximal de paramètres passés aux fonctions appelées.

- On n'est pas censé utiliser `%rbp` comme `%ebp` dans le cas 32 bits. A la place, `%rbp` est un registre ordinaire. En entrée de fonction, la fonction va allouer l'espace maximal dont elle aura jamais besoin lors de son exécution (nombre de variables fois 8, typiquement) grâce à une soustraction sur `%rsp`. Le but est que `%rsp` ne bouge pas dans toute la fonction. Toutes les variables locales et les paramètres sont références par rapport à `(%rsp)`.

  De plus, le pointeur de pile `%rsp` est censé être un multiple de 16 avant chaque appel de fonction, par l'instruction `callq`. Ceci demande à éventuellement allouer un peu plus de mémoire sur la pile que rigoureusement nécessaire.