Logiciel de base
Première année par alternance

Responsable : Christophe RIPPERT
`Christophe.Rippert@Grenoble-INP.fr`

Programmation assembleur sur l'architecture x86

Introduction

On présente dans ce document les bases de la programmation sur l'architecture x86, la version 32 bits de l'architecture Intel.

Présentation générale

Le x86 est un processeur 32 bits : cela signifie que le mot de donnée de base et les adresses en mémoire sont codés sur 4 octets.
Le x86 contient un nombre limité de registres 32 bits. On détaille ci-dessous ceux qui nous serviront en pratique :

- `%eax` (accumulateur) dont les bits 15..0 s'appellent `%ax`, les bits 15..8 s'appellent `%ah` et les bits 7..0 s'appellent `%al` ;
- `%ebx` (index de base), avec les sous-registres `%bx`, `%bh` et `%bl` comme pour `%eax` ;
- `%ecx` (compteur), idem pour `%cx`, `%ch` et `%cl` ;
- `%edx` (données), idem pour `%dx`, `%dh` et `%dl` ;
- `%esi` (index de source), dont les bits 15..0 s'appellent `%si` (et dont les bits 15..8 et 7..0 n'ont pas de nom : ça sera le cas pour tous les registres ci-après) ;
- `%edi` (index de destination), idem pour `%di` ;
- `%esp` (pointeur de pile), idem pour `%sp` ;
- `%ebp` (pointeur de base), idem pour `%bp` ;
- `%eip` (compteur programme), idem pour `%ip` ;
- `%eflags` (indicateurs), idem pour `%flags` ;

Les registres du x86 sont fractionnables. Par exemple, le nom `%ax` peut-être utilisé dans un programme assembleur pour accéder aux 16 bits de poids faibles de `%eax` (les 16 bits de poids forts ne portent pas de nom particulier). De plus, on peut accéder aux 8 bits de poids faibles de `%ax` (et donc aussi de `%eax`) en utilisant le nom `%al`. Le nom `%ah` permet d'accéder aux 8 bits de poids fort de `%ax`.
Les 4 registres généraux (`%eax`, `%ebx`, `%ecx` et `%edx`) peuvent être utilisés pour effectuer des calculs quelconques. Les registres `%esi` et `%edi` ont des rôles particuliers lorsqu'on utilise certaines instructions, mais dans notre cas on les utilisera comme des registres de calcul généraux.
Les registres de pile (`%esp` et `%ebp`) ont des fonctions particulières que l'on détaillera plus tard. On ne doit jamais les utiliser pour effectuer des calculs.
Le registre `%eip` pointe en permanence sur la prochaine instruction à exécuter (c'est le compteur programme que l'on trouve dans toutes les architectures). On ne peut pas le manipuler directement dans un programme assembleur.
Enfin, le registre des indicateurs `%eflags` contient notamment les indicateurs `Z`, `C`, ... communs à la majorité des processeurs.
Les adresses sont codées sur 32 bits : la taille maximale de la mémoire adressable est donc $2^{32} = 4GiO$.
Le bus de données est aussi sur 32 bits, mais on peut accéder à des données sur 32, 16 et 8 bits : on devra donc systématiquement préciser la taille des données manipulées via un suffixe de taille ajouté à la fin des instructions :

- `l` (long) pour les données sur 32 bits ;
- `w` (word) pour les données sur 16 bits ;
- `b` (byte) pour les données sur 8 bits ;

Le x86 est un processeur little-endian, ce qui signifie que dans un mot mémoire, les bits de poids faibles sont stockés en premier. Par exemple, si la valeur (sur 32 bits) 0x12345678 est stockée à l'adresse 0x1000, on trouve en mémoire :

Adresses	0x1000	0x1001	0x1002	0x1003
Valeurs	0x78	0x56	0x34	0x12

Instructions courantes

L'instruction de copie

`mov src, dst` copie une valeur d'un endroit à un autre :

- d'un registre vers un autre registre : `movl %eax, %ebx` : copie le contenu du registre 32 bits `%eax` dans le registre `%ebx` ;
- d'un registre vers la mémoire : `movb %al, 0x1234` : copie le contenu du registre 8 bits `%al` dans la case mémoire 8 bits d'adresse `0x1234` ;
- de la mémoire vers un registre : `movw 0x1234, %bx` : copie le contenu de case mémoire 16 bits d'adresse 0x1234 dans le registre `%bx` ;
- une constante vers un registre : `movb $45, %dh` : copie la valeur sur 8 bits 45 dans le registre `%dh` ;
- une constante vers la mémoire : `movl $0x1234, 0x1234` : copie la valeur sur 32 bits 0x00001234 dans la case mémoire 32 bits d'adresse 0x1234.

Il n'est par contre pas possible de copier une valeur de la mémoire vers la mémoire.

Opérations arithmétiques

Il existe de nombreuses opérations arithmétiques, dont on détaille les plus courantes :

- Addition : `add`, par exemple `addw %ax, %bx` calcule `%bx := %bx + %ax` ;
- Soustraction : `sub`, par exemple `subb $20, %al` calcule `%al := %al - 20` ;
- Négation : `neg`, par exemple `negl %eax` calcule `%eax := -%eax` ;
- Décalage à gauche : `shl`, par exemple `shll $1, %eax` décale la valeur sur 32 bits contenue dans le registre `%eax` d'un bit vers la gauche, en insérant un 0 à droite ;
- Décalage arithmétique à droite : `sar`, par exemple `sarl $12, %ebx` décale la valeur sur 32 bits contenue dans le registre `%ebx` de 12 bits vers la droite, en propageant le bit de signe à gauche ;
- Décalage logique à droite : `shr`, par exemple `shrl $4, %ebx` décale la valeur sur 32 bits contenue dans le registre `%ebx` de 4 bits vers la droite, en insérant des 0 à gauche ;
- Conjonction logique : `and`, par exemple `andw $0xFF00, %cx` calcule : `%cx := %cx AND 0xFF00` ;
- Disjonction logique inclusive : `or`, par exemple `orb $0x0F, %al` calcule `%al := %al OR 0x0F` ;
- Disjonction logique exclusive : `xor`, par exemple `xorl %ax, %ax` calcule `%ax := %ax XOR %ax` ;
- Négation logique : `not`, par exemple `notl %ecx` calcule l'opposé bit-à-bit de `%ecx`.

Comparaisons

On utilise les comparaisons avant un branchement conditionnel :

- Comparaison arithmétique : `cmp`, par exemple `cmpl $5, %eax` compare `%eax` avec 5, en effectuant la soustraction `%eax - 5` sans stocker le résultat ;
- Comparaison logique : `test`, par exemple `testb $0x01, %bl` effectue un et bit-à-bit entre la constante 1 et `%bl`, sans stocker le résultat.

Les comparaisons ne stockent pas le résultat de l'opération effectuée, mais mettent à jour le registre des indicateurs `%eflags`, qui est utilisé par les branchements conditionnels.

Branchements

Le branchement le plus simple est le branchement inconditionnel : `jmp destination`.
Pour préciser la destination d'un branchement, on utilise une étiquette :

```
movl $0, %eax
jmp plus_loin
```

```
    movl $5, %edx
plus_loin:
    addl $10, %eax
```

Les branchements conditionnels se basent sur l'état d'un ou plusieurs indicateurs contenus dans le registre `%eflags` pour déterminer si le saut doit être effectué ou pas :

— `je etiq` saute vers l'étiquette `etiq` ssi la comparaison a donné un résultat égal ;
— `jne etiq` saute ssi le résultat était différent ;

Les indicateurs à tester sont parfois différents selon si on travaille sur des entiers signés ou naturels. Pour les naturels, on utilisera :

— `ja etiq` saute ssi le résultat de la comparaison était strictement supérieur (*jump if above*) ;
— `jb etiq` saute ssi le résultat de la comparaison était strictement inférieur (*jump if below*) ;

Et pour les entiers signés :

— `jg etiq` saute ssi le résultat de la comparaison était strictement supérieur (*jump if greater*) ;
— `jl etiq` saute ssi le résultat de la comparaison était strictement inférieur (*jump if less*) ;

On peut composer les suffixes et obtenir ainsi plusieurs mnémoniques différents pour la même instruction : `jna` (*jump if not above*) est strictement équivalent à `jbe` (*jump if below or equal*).
Le tableau ci-dessous résume les différents branchements conditionnels qu'on utilisera :

Comparaison	Entiers naturels	Entiers signés
$>$	`ja`, `jnbe`	`jg`, `jnle`
$\geq$	`jae`, `jnb`	`jge`, `jnl`
$<$	`jb`, `jnae`	`jl`, `jnge`
$\leq$	`jbe`, `jna`	`jle`, `jng`
$=$	`je`, `jz`	
$\neq$	`jne`, `jnz`	

Adressages mémoire

Les modes d'adressage sont les différents mécanismes fournis par le processeur pour accéder à la mémoire. On détaille ici les plus couramment utilisés en pratique.

L'adressage immédiat (constantes)

Ce n'est pas un accès mémoire à proprement parler, car il sert à mettre une constante dans un registre (la constante étant codée directement dans l'instruction).
Par exemple : `movl $5, %eax` charge la constante (sur 32 bits) 5 dans le registre `%eax`.

L'adressage direct (variables globales)

Lorsqu'on souhaite accéder à une variable globale, on utilise un mode d'adressage très simple, qui consiste à utiliser directement l'étiquette correspondant au nom de la variable globale. Par exemple, si on écrit en C :

```
int32_t x = 5;

void fct(void)
{
    x++;
}
```

on peut traduire ce code comme suit en assembleur :

```
fct:
    ...
    addl $1, x
    ...
```

```
// .data indique que ce qui suit correspond a des donnees
    .data
// la directive .comm sert a reserver de la place pour une variable globale
.comm x, 4
```

Cela suppose bien sûr que la variable `x` a été initialisée quelque-part avant d'appeler la fonction `fct`.

L'adressage indirect (pointeurs)

On manipule fréquemment dans les programmes C des variables qui contiennent elles-mêmes des adresses d'autres variables : on parle alors de pointeurs.
On peut manipuler ce type de variables facilement en assembleur grâce à l'adressage indirect, qui consiste à stocker l'adresse de la variable pointée dans un registre, que l'on appelle généralement registre de base. Attention, le registre servant de base doit obligatoirement être sur 32 bits sur x86.
Par exemple, si `%eax` contient l'adresse d'une variable de type `int32_t x`, on écrit : `movl $5, (%eax)` pour copier 5 dans `x`.
Évidemment, il faut pouvoir copier l'adresse de `x` dans `%eax` initialement. Pour cela, on utilise l'instruction `lea` (*Load Effective Address*) comme dans l'exemple suivant :

```
int32_t x;
int32_t *ptr;

void fct(void)
{
    // on copie l'adresse de x dans ptr
    ptr = &x;
    // on copie 5 dans la case mémoire pointée par ptr
    // cette ligne est donc equivalente a "x = 5"
    *ptr = 5;
}
```

que l'on peut traduire en assembleur par le code suivant (en supposant que la variable `ptr` est stockée dans le registre `%eax`) :

```
fct:
    ...
    // on copie l'adresse de x dans %eax
    leal x, %eax
    // on copie 5 dans la case mémoire pointée par %eax
    movl $5, (%eax)
    ...

    .data
.comm x, 4
```

Attention : il faut bien différencier les deux lignes suivantes :

```
    // copie L'ADRESSE de x dans %eax
    leal x, %eax
    // copie LA VALEUR de x dans %eax
    movl x, %eax
```

Il est possible d'ajouter un déplacement à l'adresse contenue dans le registre. Ce déplacement doit être une constante entière signée. On utilisera beaucoup ce mode d'adressage pour accéder à des variables locales. Par exemple : `movl -8(%ebp), %eax` copie dans le registre `%eax` l'entier sur 32 bits situé dans la case mémoire dont l'adresse est calculée en enlevant 8 au contenu du registre `%ebp`.

L'adressage indirect avec index (tableaux)

Dans ce mode, on utilise deux registres 32 bits pour stocker l'adresse de la variable que l'on souhaite manipuler.
L'adresse de la variable est calculée selon la formule : $adresse = base + index \times type + deplacement$ où :

- la base est la valeur contenue dans le premier registre (registre de base) ;
- l'index est la valeur contenue dans le deuxième registre (registre d'index) ;
- le type est une constante entière valant forcément 1, 2 ou 4 ;
- le déplacement est une constante entière quelconque qui, comme indiqué ci-dessus, n'est pas multipliée par le type.

Si le déplacement est 0, on peut ne pas l'écrire. Par exemple `movl 0(%edx, %ecx, 4), %eax` est équivalent à `movl (%edx, %ecx, 4), %eax`.
Si on ne précise pas le type, il vaut 1 par défaut. Par exemple, `movl 5(%edx, %ecx, 1), %eax` est équivalent à `movl 5(%edx, %ecx), %eax`.
Ce mode d'adressage est très utile pour accéder à des tableaux : le registre de base contient l'adresse du début du tableau, le type représente la taille d'un élément du tableau et le registre d'index contient l'indice de la case à accéder. Si c'est un tableau de structures, l'index permet de passer d'une structure à la suivante et le déplacement permet d'accéder directement au champ désiré de la structure.
Attention, dans tous les cas, les registres de base et d'index doivent obligatoirement être des registres 32 bits sur x86. Par exemple, si on écrit en C :

```
int32_t tab[10];
int32_t i;

void fct(void)
{
    ...
    if (tab[i] == tab[i + 1]) {
        ...
    }
    ...
}
```

la comparaison entre `tab[i]` et `tab[i + 1]` pourra s'écrire :

```
    ...
    leal tab, %edx
    movl i, %ecx
    movl (%edx, %ecx, 4), %eax
    cmpl 4(%edx, %ecx, 4), %eax
    jne fin_if
    ...
```

Appels de fonctions

On doit respecter un certain nombre de conventions de gestion de la pile d'exécution lorsqu'on veut appeler des fonctions C depuis du code assembleur ou l'inverse.

Les pointeurs de pile

Le x86 comprend 2 registres dont le rôle est lié à la pile d'exécution : `%esp` et `%ebp`.

- `%esp` est le pointeur de pile : il contient en permanence l'adresse de la dernière case occupée dans la pile d'exécution. On ne le manipule pas directement en général pour éviter de risquer de déséquilibrer la pile.
- `%ebp` est le pointeur de base : il contient l'adresse de la base du contexte d'exécution de la fonction en cours d'exécution. Dans notre cas, il pointera en permanence sur la case dans laquelle on a sauvegardé le `%ebp` précédent. On se sert de `%ebp` pour accéder aux variables locales et aux

paramètres de la fonction en cours d'exécution, via un adressage indirect avec déplacement (e.g. `-4(%ebp)`, `8(%ebp)`, ...).

Instructions d'appel et de retour

L'instruction call permet d'appeler une fonction en précisant son étiquette : par exemple `call fact` appelle la fonction `fact`. Cette instruction a le même comportement qu'un branchement inconditionnel (`jmp`) mais en plus, elle empile automatiquement l'adresse de retour dans la pile d'exécution. L'adresse de retour est simplement l'adresse de l'instruction suivant le `call` dans la fonction appelante.
L'instruction réciproque s'appelle `ret` : elle est équivalente au mot clé `return` que l'on trouve dans de nombreux langages de programmation. Cette instruction dépile l'adresse de retour et revient donc à l'instruction suivant le `call` dans la fonction appelante.

Instructions de sauvegarde et restauration dans la pile

On utilise l'instruction `push` pour empiler une valeur, c'est-à-dire :
- déplacer le pointeur de pile `%esp` qui pointe sur la dernière case occupée ;
- copier la valeur dans la case mémoire pointée maintenant par `%esp`.

Par exemple, `pushl %eax` est équivalent à :

```
subl $4, %esp
movl %eax, (%esp)
```

L'instruction réciproque s'appelle `pop` et elle permet de dépiler la valeur en sommet de pile. Par exemple, `popl %eax` est équivalent à :

```
movl (%esp), %eax
addl $4, %esp
```

Gestion des registres

Le compilateur GCC classe les registres généraux du processeur dans 2 catégories :
- les registres *scratch* sont des registres de calcul qui peuvent être utilisés librement dans les fonctions en assembleur ;
- les registres *non-scratch* sont des registres précieux dans lesquels le compilateur peut stocker des valeurs importantes.

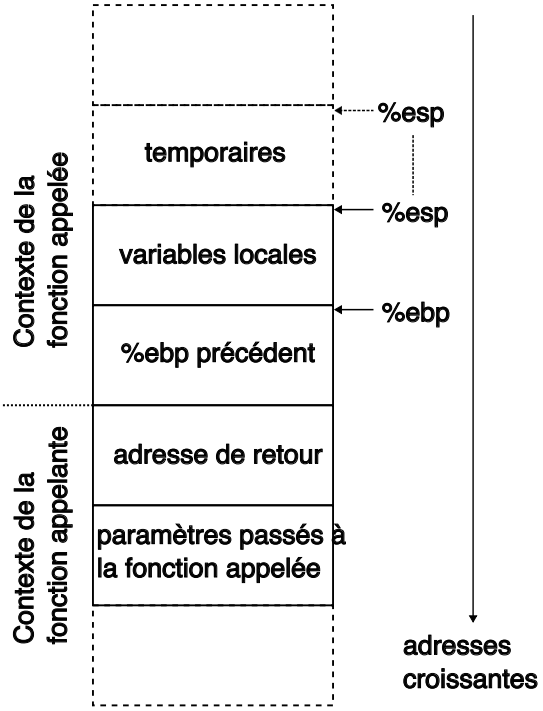
Sur x86 :
- les registres *scratch* sont `%eax`, `%edx`, `%ecx` et `%eflags` ;
- les registres *non-scratch* sont `%ebx`, `%esi`, `%edi`, `%ebp` et `%esp`.

Cette convention est importante lorsqu'on appelle une fonction C depuis du code assembleur :
- la fonction en assembleur ne doit pas laisser de valeurs importantes dans les registres *scratch* sans les sauvegarder avant d'appeler la fonction C : en effet, l'appel à la fonction C pourrait parfaitement détruire le contenu de ces registres, car GCC considère qu'il peut s'en servir comme bon lui semble ;
- la fonction en assembleur peut par contre laisser des valeurs importantes dans les registres *non-scratch*, car on est sûr qu'ils ne seront pas modifiés par l'appel à la fonction C.

Et réciproquement bien sûr lorsqu'on écrit une fonction assembleur appelée depuis du C : il ne faut pas utiliser les registres *non-scratch* sans les avoir sauvegardés au préalable, car le compilateur s'attend à retrouver les valeurs qu'il y avait placé avant l'appel à la fonction assembleur.

Conventions pour le cadre de pile

Lorsqu'on écrit un programme complet en assembleur, en mélangeant des fonctions C et assembleur qui s'appellent les unes les autres, on doit respecter des conventions de gestion de la pile d'exécution.
Sur x86, les paramètres de la fonction qu'on souhaite appeler sont mis dans la pile par l'appelant de la fonction (par des instructions `push`). Un cadre de pile a la structure suivante :



Notez qu'en 32 bits, la pile doit être alignée sur des adresses multiples de 4 : il suffira de réserver un nombre d'octets multiples de 4 pour les variables locales et les éventuels temporaires.
Dans ce schéma, les adresses sont croissantes vers le bas, mais les valeurs sont empilées en haut, car il s'agit d'une pile au sens algorithmique du terme. Les différentes zones sont détaillées ci-dessous :

- les paramètres sont ceux passés à la fonction en cours d'exécution, ils sont mis par l'appelant de la fonction (par des instructions `push`) ;
- l'adresse de retour est l'adresse de l'instruction suivant l'appel de la fonction en cours d'exécution dans la fonction appelante, cette adresse est mise dans la pile automatiquement lors de l'appel de la fonction avec l'instruction `call` ;
- le `%ebp` précédent est la sauvegarde du pointeur de base de la fonction appelante, il est sauvegardé à l'entrée de la fonction par l'instruction `enter` ;
- les variables locales de la fonction en cours d'exécution, si elles existent, sont localisées dans son cadre de pile, à des adresses fixes par rapport à `%ebp`, on peut réserver l'espace nécessaire en déplaçant le pointeur de pile `%esp` de la taille à réserver en utilisant le premier paramètre de l'instruction `enter` ;
- on pourrait comme en 64 bits réserver de la place dans le cadre de pile pour sauvegarder des registres *non-scratch* si besoin (zone des temporaires), mais la convention en 32 bits est plutôt d'utiliser des `push` et `pop` pour stocker les temporaires au-dessus de la position initiale du registre `%esp`.

Le cadre de pile d'une fonction est mis en place en deux temps :

- dans la fonction appelante, qui copie les éventuels paramètres supplémentaires qu'elle va passer à la fonction appelée et empile l'adresse de retour en exécutant `call` ;
- dans la fonction appelée, qui réserve la place nécessaire à son contexte d'exécution.

Lorsqu'on écrit le code d'une fonction, on commence par l'instruction `enter $N, $0` qui a pour rôle de réserver l'espace nécessaire au contexte d'exécution. Le premier paramètre précise le nombre `N` d'octets à réserver pour le contexte (le deuxième paramètre sera toujours 0 dans notre cas). L'instruction `enter $N, $0` est en fait équivalente à la suite d'instructions suivante :

```
pushl %ebp
movl %esp, %ebp
subl $N, %esp
```

Le `pushl` correspond à la sauvegarde du `%ebp` précédent dans le schéma du cadre de pile ci-dessus. Ensuite, on fait pointer `%ebp` sur la case où on a sauvegardé son ancienne valeur, et on décrémente `%esp`

de `N` octets : c'est cette opération qui réserve la place nécessaire au cadre de pile. On rappelle que la pile descend vers les adresses décroissantes, d'où le fait qu'on effectue des soustractions.
L'instruction réciproque de `enter` s'appelle `leave`. Elle s'utilise sans paramètre et a pour effet de détruire le cadre de pile mis en place par `enter`. Elle est équivalente à la séquence d'instructions :

```
movl %ebp, %esp
popl %ebp
```

Elle commence par faire pointer `%esp` sur la case contenant l'ancienne valeur de `%ebp` : c'est comme cela qu'on détruit le cadre de pile (la mémoire n'est bien sûr pas vidée et contient toujours les valeurs précédentes, mais ces valeurs ne doivent plus être utilisées par le programme). Ensuite, on restaure l'ancienne valeur de `%ebp` en la dépilant.
Si on prend l'exemple classique du PGCD, on pourra écrire en assembleur x86 :

```
    // .text precise que ce qui suit est du code (pas des donnees)
 .text
    // .globl rend l'etiquette publique
    .globl pgcd
    // uint32_t pgcd(uint32_t a, uint32_t b)
    // a : %ebp + 8
    // b : %ebp + 12
    // etiquette designant le debut de la fonction
pgcd:
    // on decale les instructions d'une tabulation
    enter $0, $0
    // while (a != b) {
while:
    movl 12(%ebp), %eax
    cmpl 8(%ebp), %eax
    je fin_while
    // if (a > b) {
    movl 12(%ebp), %eax
    cmpl 8(%ebp), %eax
    ja else
    // a = a - b;
    movl 12(%ebp), %eax
    subl %eax, 8(%ebp)
    jmp fin_if
else:
    // b = b - a:
    movl 8(%ebp), %eax
    subl %eax, 12(%ebp)
fin_if:
    jmp while
fin_while:
    // return a;
    movl 8(%ebp), %eax
    leave
    ret
```

La séquence d'appel de cette fonction sera (si dans la fonction appelante, on écrit `pgcd(15, 10);`) :

```
...
pushl $10
pushl $15
call pgcd
addl $8, %esp
...
```

On note bien qu'on doit :

— copier les paramètres dans l'ordre inverse de leur déclaration, de façon à ce que le premier paramètre (dans l'ordre de déclaration) soit le plus « proche » à partir de la fonction appelée ;

— rééquilibrer la pile après l'appel en ajoutant à `%esp` la taille des paramètres qu'on avait empilés.

Si on oublie de rééquilibrer la pile, le code fonctionnera vraisemblablement, mais en cas de nombreux appels récursifs, on risque de provoquer un débordement de pile.

Exercices

Récupérer les sources fournies sur la page principale du cours.
Note : la version de Valgrind installée ne reconnait pas l'instruction `enter` : il faut donc remplacer cette instruction par la séquence `pushl` / `movl` / `subl` équivalente décrite ci-dessus.

Encore des histoires d'alignement mémoire...

Soit le code C suivant qui déclare un caractère et un entier, les passent en paramètres à une fonction `affiche_asm` écrite en assembleur et dont le code C équivalent est donné en commentaires. Cette fonction assembleur va donc légèrement modifier les deux paramètres puis les passer en paramètres à la fonction `affiche_c` qui les affiche à l'écran.

```
#include <stdio.h>
#include <inttypes.h>

extern void affiche_asm(char, uint16_t);
/*
  void affiche_asm(char c, uint16_t s)
  {
      affiche_c(c + 1, s - 1);
  }
*/

void affiche_c(char c, uint16_t s)
{
    printf("Caractere = %c, Short = %" PRIu16 "\n", c, s);
}

int main(void)
{
    char c = 'a';
    uint16_t s = 1024;
    affiche_asm(c, s);
    return 0;
}
```

Regardez le code assembleur de la fonction `affiche_asm` ci-dessous et déduisez-en comment GCC passe des paramètres de taille inférieure à 32 bits en mode 32 bits. On rappelle qu'en 32 bits, la pile doit toujours être alignée sur des adresses multiples de 4.

```
    .text
    .globl affiche_asm
affiche_asm:
    enter $0, $0
    movl $0, %eax
    movw 12(%ebp), %ax
    subw $1, %ax
    pushl %eax
    movl $0, %eax
```

```
movb 8(%ebp), %al
addb $1, %al
pushl %eax
call affiche_c
addl $8, %esp
leave
ret
```

Appels croisés C/assembleur

On reprend la fonction factorielle récursive qu'on a déjà écrite en mode 64 bits, pour l'écrire cette fois-ci en 32 bits. Pensez bien à vérifier quand le résultat de factorielle ne tient pas sur 32 bits, et à appeler dans ce cas la fonction `erreur_fact` fournie dans le fichier `fact.c`.

Utilisation de la bibliothèque C

On travaille maintenant sur la fonction `palin` qui détecte les palindromes et qu'on a déjà traité dans la séance sur les fonctions. Implantez donc une version 32 bits de cette fonction, en gardant bien en tête les conventions d'appels de fonction de l'ABI 32 bits pour l'appel à `strlen`.

Manipulation de listes chaînées

On reprend l'exercice de la dernière séance sur les listes chaînées. Implantez donc en assembleur 32 bits une fonction `inverse` qui inverse une liste chaînée et une fonction `decoupe` qui découpe une liste chaînée en deux listes : l'une contenant tous les nombres pairs et l'autre tous les nombres impairs de la liste initiale.

Tri du nain de jardin

On va reprendre enfin l'exercice déjà traité sur le tri du nain de jardin, mais cette fois en implantant une fonction `tri_nain` en assembleur 32 bits. Commencez par implanter cette fonction sans aucune optimisation, puis ajoutez une fonction `tri_nain_opt` où vous chercherez à minimiser les accès mémoire en utilisant au mieux les registres disponibles (moins nombreux qu'en assembleur 64 bits).