Logiciel de base
Première année par alternance

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

Introduction à l'architecture Intel

Introduction

Le but de cette séance est de découvrir l'architecture Intel x86_64 / x86_32 que l'on va utiliser ce semestre pour écrire du code assembleur.
L'architecture Intel est une architecture à jeu d'instructions complexe (*CISC* en anglais, par opposition aux processeurs *RISC* plus courants) : cela signifie qu'il supporte beaucoup d'instructions, dont certaines sont capables d'effectuer des opérations très complexes. Dans le cadre de ce cours, on n'utilisera qu'un tout petit sous-ensemble du jeu d'instruction supporté par le processeur.
Cette architecture a aussi la caractéristique de préserver une compatibilité ascendante : cela signifie que les processeurs Intel actuels supportant l'architecture 64 bits x86_64 peuvent aussi exécuter des programmes compilés pour l'architecture 32 bits x86_32 (et même pour l'architecture x86 16 bits du 8086).
On étudiera donc deux versions de l'architecture Intel : la version 64 bits (utilisée actuellement sur la plupart des machines grand public) et la version 32 (qu'on trouve encore sur certains systèmes mobiles). L'intérêt d'étudier ces deux versions est qu'elles proposent des conventions d'utilisations assez différentes tout en gardant une base d'instructions et de registres communs, et elles sont donc plus faciles à apprendre et à comparer que deux processeurs totalement différents.
En pratique, on écrira de petits programmes en assembleur x86_64, que l'on interfacera avec des programmes écrits en C, et on utilisera l'architecture x86_32 pour la partie mini-projet de système dans la dernière partie du cours.

Bases sur l'architecture Intel

Le langage assembleur

L'assembleur est le langage compréhensible par un humain qui se situe le plus prêt de la machine.
Le processeur exécute des instructions écrites en binaire comme vous l'avez vu dans le cours d'architecture des ordinateurs. Il est très difficile pour un humain de se souvenir que la séquence

0100100011000111110000000000010100000000000000

copie la valeur 5 dans un registre appelé `%rax`. C'est pour cela qu'on utilise l'assembleur, qui est représentation textuelle des instructions supportées par le processeur : dans le cas de cet exemple, on écrira `movq $5, %rax`, ce qui est nettement plus clair.
L'assembleur est un langage sans structure de contrôle (i.e. pas de `if`, `while`, etc.) dont chaque instruction peut-être traduite directement dans la séquence binaire équivalente compréhensible par le processeur. Le logiciel qui réalise cette traduction s'appelle aussi un assembleur. Il en existe beaucoup supportant l'architecture Intel : on utilisera l'assembleur GAS qui est l'assembleur intégré à GCC.

Le processeur

Le premier processeur supportant l'architecture x86_64 (aussi connue sous le nom d'AMD64 du nom du fabricant de ce premier processeur) est sorti en 2003 et présente la propriété d'être compatible avec les processeurs des séries Pentium (1993), 80386 (1985) et 8086 (1978). Cette compatibilité ascendante explique en partie la complexité de l'architecture Intel.
L'architecture x86_64 est une architecture « 64 bits » : cela signifie que le mot de donnée de base et les adresses en mémoire sont codés sur 8 octets. L'architecture x86_32 est quant à elle une architecture 32 bits : le mot de données et les adresses mémoires sont donc codés sur 4 octets.

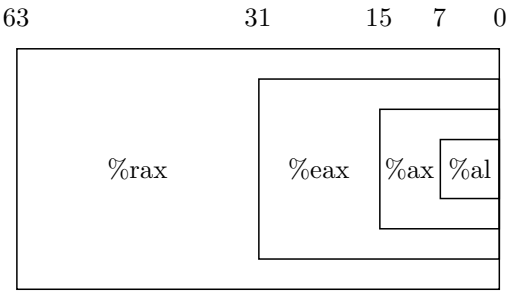
Les processeurs conforment à l'architecture Intel x86_64 contiennent seulement 14 registres généraux pouvant être utilisés librement pour effectuer des calculs (les processeur RISC en ont généralement beaucoup plus). Ces registres ont la particularité de pouvoir être fractionnés en sous-registres de taille 32, 16 ou 8 bits. L'architecture x86_32 contient quant à elle 6 registres généraux.
Par exemple, le registre `%rax` est un registre 64 bits mais :

— les 32 bits de poids faibles (i.e. les bits 31..0) sont directement en utilisant le nom `%eax` ;
— les 16 bits de poids faibles (i.e. les bits 15..0) sont accessibles grâce au nom `%ax` ;
— et les 8 bits de poids faibles (i.e. les bits 7..0) sont accessibles par le nom `%al`.

Il s'agit d'un héritage des processus 80386 (processeur 32 bits) et 8086 (processeur 16 bits). Attention, pour ceux qui connaissent l'architecture x86, les noms de registres `%ah`, `%bh`, `%ch` et `%dh` ne doivent pas être utilisés dans les programmes x86_64.
Il est important de comprendre qu'il s'agit bien du même registre 64 bits : simplement certaines parties de ce registre portent des noms spécifiques. Ainsi, par exemple :

— si on copie une valeur 64 bits dans `%rax`, tout le registre sera modifié ;
— si on copie une valeur 32 bits dans `%eax`, seuls les bits 31..0 de `%rax` seront modifiés ;
— si on copie une valeur 16 bits dans `%ax`, seuls les bits 15..0 de `%rax` seront modifiés ;
— si on copie une valeur 8 bits dans `%al`, seuls les bits 7..0 de `%rax` seront modifiés.

Le schéma ci-dessous illustre l'emboitement des registres `%rax`, `%eax`, `%ax` et `%al` :



Le même schéma se retrouve pour les autres registres du processeur :

— `%rbx` (64 bits) se découpe en `%ebx` (32 bits), `%bx` (16 bits) et `%bl` (8 bits) ;
— `%rcx` (64 bits) se découpe en `%ecx` (32 bits), `%cx` (16 bits) et `%cl` (8 bits) ;
— `%rdx` (64 bits) se découpe en `%edx` (32 bits), `%dx` (16 bits) et `%dl` (8 bits) ;
— `%rsi` (64 bits) se découpe en `%esi` (32 bits), `%si` (16 bits) et `%sil` (8 bits) ;
— `%rdi` (64 bits) se découpe en `%edi` (32 bits), `%di` (16 bits) et `%dil` (8 bits).

Les 8 autres registres généraux ne sont utilisable que sur une architecture 64 bits et portent des noms plus réguliers :

— `%r8` (64 bits) se découpe en `%r8d` (32 bits), `%r8w` (16 bits) et `%r8b` (8 bits) ;
...
— `%r15` (64 bits) se découpe en `%r15d` (32 bits), `%r15w` (16 bits) et `%r15b` (8 bits).

Les versions 64 bits des registres (par exemple `rax`) ne sont utilisables que sur l'architecture x86_64 : sur x86_32, les registres s'arrêtent à 32 bits (par exemple `%eax`).
Il existe enfin d'autres registres dont on détaillera l'intérêt plus tard :

— `%rsp` est le pointeur de sommet de pile et `%rbp` le pointeur de cadre de pile : ces registres seront présentés en détail lorsque l'on apprendra à écrire des fonctions en assembleur ;
— `%rip` est le compteur-programme (le `PC` de vos cours d'architecture) : il ne peut pas être modifié directement, à part par des instructions de saut ;
— `%rflags` est le registre des indicateurs (qui contient par exemple les indicateurs C et Z que vous avez vu en architecture), lui aussi ne peut pas être manipulé directement.

Ces différents registres seront toujours manipulés sur 64 bits quand on écrira de l'assembleur x86_64 et sur 32 bits quand on écrira de l'assembleur x86_32 (dit autrement, vous ne devez jamais utiliser leurs parties 16 bits et 8 bits, ça n'aurait pas de sens).

La mémoire

Sur l'architecture x86_64, les adresses sont codées sur 64 bits : la taille maximale de la mémoire adressable est donc $2^{64} = 18446744073709551616$ soit environ 18 milliards de milliards de cases mémoire. Il s'agit

bien sûr d'une limite théorique : il n'existe pas encore de machine disposant d'autant de mémoire ! L'architecture x86_32 supporte quant à elle des adresses allant jusqu'à $2^{32} = 4294967296$, c'est à dire 4 GiO : cette taille mémoire paraissait encore énorme il y a quelques années, mais c'est devenu maintenant une taille courante sur la plupart des machines grand public, et même insuffisante pour de gros serveurs de calcul.
Le bus de données de l'architecture x86_64 est aussi sur 64 bits, mais on peut accéder à des données sur 64, 32, 16 et 8 bits : on devra donc systématiquement préciser la taille des données manipulées. L'architecture x86_32 fonctionne de façon similaire, avec une taille maximum de donnée de 32 bits bien sûr.
L'architecture Intel est conçue pour des processeurs *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

Programmation en assembleur

L'architecture Intel implante plusieurs centaines d'instructions. Certaines de ces instructions sont capables de réaliser des opérations très puissantes, comme par exemple effectuer des calculs mathématiques complexes en une seule opération. Dans le cadre de ce cours, on n'utilisera qu'un sous-ensemble très restreint des instructions fournies.

L'instruction de copie

`mov src, dst` copie une valeur d'un endroit à un autre, spécifiquement :
- d'un registre à un autre ;
- d'un registre vers la mémoire ("store") ;
- de la mémoire vers un registre ("load") ;
- une constante dans un registre ("move immediate") ;
- et même une constante vers la mémoire.

Il n'est par contre pas possible de copier une valeur de la mémoire vers un autre endroit en mémoire : cette restriction est valable pour la quasi-totalité des instructions, qui ne pourront jamais avoir deux opérandes en mémoire.
Comme on l'a vu, l'architecture Intel permet de manipuler des données sur 64 (pour le x86_64), 32, 16 ou 8 bits : on doit donc spécifier systématiquement la taille des données manipulées, en utilisant des suffixes de taille, qui seront collés à la fin de l'instruction :
- `q` est le suffixe pour une donnée sur 64 bits : `movq %rax, %r10` copie les 64 bits de `%rax` dans les 64 bits de `%r10` ;
- `l` est le suffixe pour une donnée sur 32 bits : `movl $0x12345678, %eax` charge la constante hexadécimale 0x12345678 dans le registre 32 bits `%eax` ;
- `w` est le suffixe pour une donnée sur 16 bits : `movw %ax, a` copie le contenu du registre 16 bits `%ax` dans la variable globale 16 bits nommée `a` ;
- `b` est le suffixe pour une donnée sur 8 bits : `movb $15, b` copie la valeur 15 dans la variable globale 8 bits nommée `b`.

Il est **indispensable** d'utiliser les suffixes de taille lorsqu'on écrit du code Intel : si on ne le fait pas (et qu'on écrit par exemple `mov` au lieu de `movq`, `movl`, etc.), c'est l'assembleur qui choisira la taille des données manipulée (en fonction de ce qui lui semble « logique ») mais il n'est pas sûr qu'il choisisse celle que l'on sous-entendait, et on risque de se retrouver avec une erreur très difficile à localiser.

Opérations arithmétiques

Il existe de nombreuses opérations arithmétiques, dont on détaille les plus courantes :
- Addition : `add`
- Soustraction : `sub`
- Négation : `neg`

— Décalage à gauche : `shl` décale une valeur d'un nombre de bits N compris entre 1 et 63 (sur x86_64) ou 31 (sur x86_32), en complétant avec des 0 à droite (ce qui revient à multiplier la valeur initiale par 2^N)
— Décalage arithmétique à droite : `sar` décale une valeur d'un nombre de bits N compris entre 1 et 63 (ou 31), en complétant avec **le bit de signe** à gauche (ce qui revient à diviser la valeur initiale par 2^N **si cette valeur est un entier signé**)
— Décalage logique à droite : `shr` décale une valeur d'un nombre de bits N compris entre 1 et 63 (ou 31), en complétant avec des 0 à gauche (ce qui revient à diviser la valeur initiale par 2^N **si cette valeur est un entier naturel**)
— Conjonction logique : `and`
— Disjonction logique inclusive : `or`
— Disjonction logique exclusive : `xor`
— Négation logique : `not`

Toutes les opérations fonctionnent sur le même schéma que l'instruction de copie : `op src, dst` effectue le calcul `dst = dst op src`. On note bien que le deuxième opérande de l'instruction est à la fois la destination et le premier opérande de l'opération, ce qui a un impact pour les opérations non-commutatives (comme la soustraction par exemple).

Comparaisons

On utilise les comparaisons avant un branchement conditionnel, pour implanter un `if` ou un `while` :
— Comparaison arithmétique : `cmp`, par exemple `cmpq $5, %rax` compare `%rax` avec 5, en effectuant la soustraction `%rax - 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 `%rflags`, 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 :

```
    movq $0, %rax
    jmp plus_loin
    movq $5, %r10
plus_loin:
    addq $10, %rax
```

Pour implanter des `if` et `while`, on utilise les branchements conditionnels, qui se basent sur l'état d'un ou plusieurs indicateurs contenus dans le registre `%rflags` 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 (*not equal*).

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`	

On trouvera sur la page principale une documentation détaillant la liste exhaustive de tous les branchements conditionnels existants sur x86_64.

Traduction systématiques des structures de contrôle classiques

On peut traduire les structures de contrôles des langages de haut-niveau de façon très systématique, ce qui réduit le risque d'erreur. Dans le cadre de ce cours, on demandera toujours de traduire systématiquement les algorithmes donnés en C, sans chercher à « optimiser » le code (sauf indication contraire).

Structure d'un `if`

Soit par exemple le code C suivant, où `x` est une variable globale de type `(u)int64_t` initialisée auparavant :

```
if (x == 5) {
    x = x + 2;
} else {
    x = x - 4;
}
```

Le code assembleur x86_64 correspondant prendra toujours la forme ci-dessous :

```
if:
    cmpq $5, x
    jne else
    addq $2, x
    jmp fin_if
else:
    subq $4, x
fin_if:
```

Structure d'une boucle `while`

On suppose dans le programme ci-dessous que `x` est une variable de type `int64_t` déclarée globalement et initialisée auparavant :

```
while (x > 5) {
    x = x - 1;
}
```

Le type de `x` impose d'utiliser le bon branchement conditionnel dans le code assembleur :

```
while:
    cmpq $5, x
    jle fin_while
    subq $1, x
    jmp while
fin_while:
```

Structure d'une boucle `for`

Même supposition pour `x` que dans le programme précédent, et on suppose aussi que `i` est une variable globale de type `uint64_t` non-initialisée :

```
for (i = 0; i < 5; i ++) {
    x = x + 4;
}
```

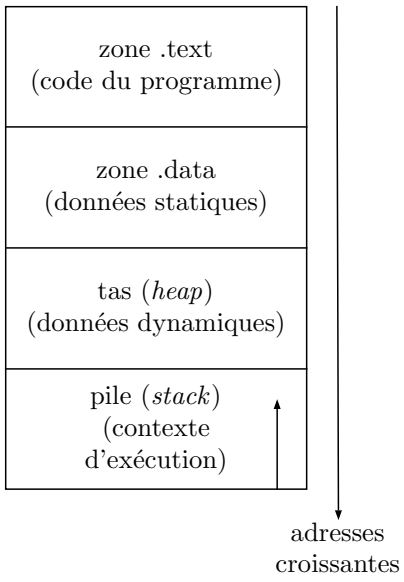
Le code aura une forme similaire à celle d'un `while` :

```
    movq $0, i
for:
    cmpq $5, i
    jae fin_for
    addq $4, x
    addq $1, i
    jmp for
fin_for:
```

Représentation d'un programme en mémoire

Principe général

Lorsqu'on compile ou assemble un programme, on obtient un fichier binaire contenant le code de ce programme en langage machine, ainsi que les données dont il aura besoin à l'exécution. Ce fichier binaire respecte un certain format, dont ELF et Mach-O par exemple.
Lorsqu'on lance l'exécution de ce programme (en tapant `./mon_prog` dans un terminal par exemple), le fichier binaire est chargé en mémoire et un certain nombre d'opérations dépassant le cadre de ce cours sont effectuées afin de rendre le programme exécutable. Ce qui nous intéresse ici est la représentation finale du programme dans la mémoire, juste avant que le processeur commence à exécuter son code machine.
On parle traditionnellement de processus pour désigner un programme en cours d'exécution dans la mémoire du système. De façon volontairement simplifiée, un processus a en mémoire la structure suivante :



Lorsqu'on écrit un programme en C, on ne se préoccupe pas de savoir dans quelle zone du processus va se retrouver telle fonction ou telle donnée : c'est le travail du compilateur. Mais lorsqu'on écrit du code assembleur, on doit définir précisément la structure du programme. On utilise pour ça des directives, qui sont des commandes qui ne font pas partie du langage assembleur mais qui aide le programme d'assemblage à faire son travail.

La zone `text`

Cette zone contient le code en langage machine de toutes les fonctions du programme. Lorsqu'on écrit de l'assembleur, on est par défaut dans cette zone. Si l'on souhaite le préciser explicitement, on peut utiliser la directive `.text` avant de commencer à écrire du code assembleur, comme par exemple dans le code ci-dessous :

```
/*
void fct(void)
{
    // du code...
}
*/
    .text
fct:
    // du code...
```

La zone `data`

Directives de réservation d'espace statique

Cette zone contient les données allouées statiquement dans le programme, c'est à dire les variables globales du langage C. La directive assembleur permettant de signifier le début de la zone `data` est tout simplement `.data`.
En assembleur, pas plus que dans la mémoire physique de la machine, il n'existe pas de types de données à proprement parler. On travaille toujours en termes d'espace mémoire réservé pour les variables : on ne sait pas si `x` est un entier ou un réel, on sait juste que cette variable occupe 8 octets en mémoire. En pratique, on n'utilisera que des entiers, des tableaux d'entiers et des chaines de caractères dans ce cours. Si on veut réserver de l'espace pour une variable entière `x` codée sur 8 octets (un entier 64 bits donc), on utilisera la syntaxe suivante : `.comm x, 8`.
On peut aussi se servir de la directive `.comm` pour réserver de l'espace pour des tableaux : par exemple, on peut allouer un tableau `tab` de 10 entiers sur 32 bits chacun avec `.comm tab, 10 * 4` (vous noterez que l'assembleur sait effectuer des opérations simples sur des constantes, cela peut permettre d'améliorer la lisibilité du code).
L'assembleur que l'on utilise fourni aussi une facilité syntaxique pour déclarer des chaines de caractères constantes dans la zone `data` : `chaine: .asciz "azerty"`.
La directive `.asciz` :

- réserve autant d'octets en mémoire qu'il y a de caractères dans la chaine `azerty` plus 1 caractère pour le `\0` final ;
- initialise le contenu de la mémoire avec les caractères `azerty` et place le 0 final.

Cette directive permet donc de traduire directement les chaines de caractères du C : ici, la constante sera accessible directement en utilisant l'étiquette `chaine`.

Correspondance entre les types du C et les directives assembleur

Le langage C définit des types entiers « classiques » comme `int`, `long`, `short`, etc. L'inconvénient de ces types est que leur taille peut dépendre de l'architecture sous-jacente. Pour éviter ce problème, on utilisera les types de tailles fixes définis dans `inttypes.h`. On donne ci-dessous la correspondance entre les types du C et les directives assembleur à utiliser dans la zone `.data` :

Type C99	Taille en octets	Directive assembleur
`int64_t x`, `uint64_t x`	8	`.comm x, 8`
`int32_t x`, `uint32_t x`	4	`.comm x, 4`
`int16_t x`, `uint16_t x`	2	`.comm x, 2`
`int8_t x`, `uint8_t x`	1	`.comm x, 1`
`char c`	1	`.comm c, 1`
pointeurs sur x86_64	8	`.comm ptr, 8`
pointeurs sur x86_32	4	`.comm ptr, 4`

On notera bien que les types pointeurs ont toujours la même taille (8 octets sur x86_64 et 4 octets sur x86_32) quelque-soit la taille des données pointées : un pointeur vers une structure complexe ou vers un simple caractère est toujours codé sur 8 octets sur x86_64 par exemple.

Le tas

Le tas (*heap* en anglais) est la zone dans laquelle les fonctions `malloc` et `calloc` prennent la mémoire à réserver dynamiquement quand on les appelle. Cette zone a en général une taille prédéfinie lors de la création du processus et peut potentiellement grandir grace à un mécanisme appelé « mémoire virtuelle » que vous étudierez dans le cours de système.

La pile

La pile (*stack* en anglais) d'exécution est la zone dans laquelle on stocke notamment les variables locales aux différentes fonctions du programme (on rappelle que la zone `data` ne contient que les variables globales). Cette zone est organisé comme une pile (au sens algorithmique du terme), et part donc de la fin de l'espace mémoire du processus pour remonter vers le début à chaque fois que l'on empile des valeurs.
La pile a une taille fixe qui est généralement une constante dépendant du système utilisé. A noter que dans de nombreux systèmes, on ne vérifie pas que l'on n'empile pas plus de données qu'il n'y a d'espace réservé pour la pile, et on peut parfaitement arriver à écraser d'autres données, ou même du code, si on en empile trop : on parle alors de « débordement de pile » (*stack overflow* en anglais), une technique couramment utilisée pour exploiter des failles de sécurité dans les systèmes.
On verra plus tard dans ce cours comment on peut utiliser des variables locales et gérer la pile d'exécution.

Etude d'un exemple : le PGCD

Récupérez les sources fournies et décompressez-les pour extraire les fichiers. Ouvrez le fichier `fct_pgcd.s` qui contient le code assembleur de la fonction PGCD.
Soit le programme en C ci-dessous qui calcule le PGCD de `a` et `b`, deux variables globales de type `uint32_` définies par exemple dans le programme principal :

```
uint32_t pgcd(void)
{
    while (a != b) {
        if (a < b) {
            b = b - a;
        } else {
            a = a - b;
        }
    }
    return a;
}
```

En assembleur, on traduit ce code comme suit :

```
    .text
    .globl pgcd
pgcd:
    enter $0, $0
    // while (a != b) {
while:
    movl a, %eax
    cmpl b, %eax
    je fin_while
    // if (a < b) {
    movl a, %eax
```

```
    cmpl b, %eax
    // jmp ssi not(a < b)
    jnb else
    // b = b - a;
    movl a, %eax
    subl %eax, b
    jmp fin_if
else:
    // a = a - b:
    movl b, %eax
    subl %eax, a
fin_if:
    jmp while
fin_while:
    // return a;
    movl a, %eax
    leave
    ret
```

Point d'entrée et étiquettes publiques

Le programme commence par la directive `.globl` qui est liée à la déclaration de l'étiquette `pgcd` :

```
    .globl pgcd
pgcd:
```

Par défaut, toutes les étiquettes déclarées dans un programme assembleur sont privées et invisibles à l'extérieur du fichier. Or la fonction `pgcd` doit être visible pour pouvoir être appelée par le programme principal. C'est le but de directive `.globl` qui rend l'étiquette `pgcd` publique.

En-tête et fin de fonction

```
enter $0, $0
...
leave
...
```

Toutes les fonctions que l'on écrira en assembleur commencent par l'instruction `enter` et se terminent par l'instruction `leave` : on comprendra plus tard à quoi servent ces instructions.

Valeur renvoyée

```
movl a, %eax
ret
```

Par convention, une fonction en assembleur renvoie sa valeur de retour dans le registre `%rax`, ou une de ses fractions si on renvoie une valeur de taille inférieure à 64 bits (ici, on renvoie un entier sur 32 bits, donc on utilise `%eax`). Si on écrit une fonction qui ne renvoie rien (`void`), il suffit de ne rien copier dans `%rax` : son contenu sera de toute façon ignoré par la fonction appelante.
L'instruction `ret` est l'équivalent du `return` en C et en Ada, et rend la main à la fonction appelante.

Déclaration des variables

Les variables globales `a` et `b` sont déclarées dans le programme principal, mais elles sont accessibles depuis le code assembleur (en C, tous les symboles sont publics sauf s'ils sont explicitement rendus privés par la directive `static`). On peut donc y accéder exactement comme si elles étaient définies dans la zone `.data` du fichier assembleur.

Exercices

On verra plus tard qu'il existe des conventions d'utilisation des registres : on ne peut pas utiliser n'importe-quel registre n'importe-comme si on veut interfacer les programmes assembleur avec du C. Dans tous les exercices ci-dessous vous n'utiliserez que les registres `%rax` (64 bits), `%eax` (32 bits) et `%al` (8 bits) pour vos valeurs temporaires (vous n'avez pas besoin de plus d'un registre pour ces exercices).

Utilisation de GDB pour la mise au point de programmes assembleur

Le fichier `pgcd.c` contient un programme principal écrit en C qui permet de lire les valeurs initiales de `a` et `b` et qui appelle la fonction PGCD écrite en assembleur dans le fichier `fct_pgcd.s`.
Vous pouvez compiler ces deux fichiers et créer le binaire en tapant simplement : `make pgcd`.
On va utiliser GDB pour tracer l'exécution du programme instruction par instruction :

- charger le programme dans GDB en tapant `gdb ./pgcd` ;
- ajouter un point d'arrêt au début de la fonction PGCD en utilisant la commande `break pgcd` ;
- afficher les valeurs des variables `a` et `b` en utilisant les commandes `display a` et `display b` : l'affichage ne se fera que lors de l'exécution suivante ;
- faites de même pour afficher le contenu du registre `%eax` : `display $eax` (attention, c'est bien le signe `$` qu'il faut utiliser devant le nom du registre, et pas un `%`) ;
- lancer l'exécution en tapant `run 15 10` : noter que l'exécution s'arrête sur le point d'arrêt positionné sur `pgcd` : GDB affiche alors la prochaine instruction à exécuter (i.e. elle n'a pas encore été exécutée) ;
- continuer l'exécution du programme pas à pas avec la commande `step` en vérifiant les valeurs des variables et du registre après l'exécution de chaque instruction qui les modifie ;
- vérifier qu'on saute bien vers `else` la première fois qu'on exécute l'instruction `jnb else` mais pas la deuxième ;
- vérifier aussi qu'on fait bien deux itérations de la boucle `while` avant d'en sortir (i.e. le branchement conditionnel `je fin_while` n'est pas effectué lors des deux premières itérations) ;
- vérifier que le PGCD (5) est bien dans `%eax` à la fin de la fonction, juste avant d'exécuter l'instruction `ret` ;
- ensuite on peut terminer proprement l'exécution du programme en tapant `continue`.

On fournit sur la page principale du cours un aide-mémoire pour GDB qui contient beaucoup plus d'information que ce qui vous sera nécessaire ce semestre.

Somme des 10 premiers entiers

Traduire en assembleur x86_64 de façon systématique le code C suivant qui calcule la somme des 10 premiers entiers naturels. Vous écrirez le code assembleur dans le fichier `fct_somme.s` et vous compilerez en tapant `make somme` pour utiliser le programme principal `somme.c` fourni. Testez bien l'exécution pas à pas du programme avec GDB : vous pouvez afficher le contenu d'une variable sur 8 bits en utilisant par exemple `display (char)res`.

```
uint8_t i, res;

uint8_t somme(void)
{
    res = 0;
    for (i = 1; i <= 10; i++) {
        res = res + i;
    }
    return res;
}
```

Notez bien que dans cet exercice, les variables globales `i` et `res` ne sont pas définies dans le programme principal : vous devez donc compléter la zone `.data` du programme assembleur pour les définir, en utilisant la directive `.comm` comme vu précédemment.

Quand on traduit un programme du C vers l'assembleur, on indiquera toujours la ligne de C en commentaire avant la séquence d'instructions assembleur correspondante, par exemple :

```
// res = 0;
movb $0, res
```

Multiplication simple

On va maintenant implanter l'opération `res := x * y` par additions successives. On donne ci-dessous le code C que l'on va devoir traduire le plus littéralement possible en assembleur, en complétant le fichier `fct_mult.s`.

```
uint64_t mult_simple(void)
{
    res = 0;
    while (y != 0) {
        res = res + x;
        y--;
    }
    return res;
}
```

On fournit un programme principal `mult.c` qui va tester toutes les versions de la multiplication avec d'abord les valeurs passées sur la ligne de commande (par exemple `./mult 10 5`) puis avec 111111111 et 111111111, et finalement 2 et 2^{31}, et afficher le temps de calcul.
Les variables globales `x` et `y` sont déclarées dans `mult.c`, mais `res` doit être définie dans la zone `.data` du programme assembleur.

Multiplication egyptienne

Refaite le même exercice en implantant cette fois-ci l'algorithme de la multiplication égyptienne :

```
uint64_t mult_egypt(void)
{
    res = 0;
    while (y != 0) {
        if (y % 2 == 1) {
            res = res + x;
        }
        x = x * 2;
        y = y / 2;
    }
    return res;
}
```

Pour implanter les opérations modulo, multiplier et diviser par 2, vous aurez sûrement besoin des instructions suivantes :

- `test A, B` compare A et B en effectuant l'opération `A and B` ;
- `shl $n, A` décale A de n bits vers la gauche ;
- `shr $n, A` décale A de n bits vers la droite (c'est un décalage logique, pas arithmétique).

Indication pour le test de la parité de `y` : l'instruction `test` effectue une conjonction logique (`and`) entre la constante 1 (qui s'écrit en binaire sur 64 bits "00..(60x 0 au milieu)..01") et `y`, ce qui peut donner 2 résultats :

- soit 1 ssi le bit 0 de `y` vaut 1, c'est à dire ssi `y` est impair ;
- soit 0 ssi le bit 0 de `y` vaut 0, c'est à dire ssi `y` est pair.

Si le résultat du test est 0, alors le *flag* `Z` passe à 1, et on peut utiliser l'instruction `jz` qui effectue le branchement ssi `Z` vaut 1 pour sauter le code dans le `if` ci-dessus.

Multiplication native

On va maintenant finir par implanter la multiplication en utilisant l'instruction de multiplication native du processeur, en traduisant le code ci-dessous :

```
uint64_t mult_native(void)
{
    res = x * y;
    return res;
}
```

Le processeur fournit une instruction de multiplication dont la syntaxe est un peu particulière. Si on suppose qu'on ne travaille ici que sur des valeurs sur 64 bits, lorsqu'on écrit `mulq x`, on effectue le produit de la valeur de la variable `x` avec le contenu du registre `%rax` : ce registre est donc un paramètre implicite de l'instruction.
Le résultat (sur 128 bits vu qu'on multiplie deux nombres sur 64 bits), est toujours stocké dans `%rdx:%rax`, c'est à dire que les 64 bits de poids fort du résultat sont dans `%rdx`, et les 64 bits de poids faibles dans `%rax` (par exemple si le résultat de la multiplication tient sur 64 bits, `mul` va écrire le résultat dans `%rax` et 0 dans `%rdx`. Si vous aviez mis quelque chose d'important dans `%rdx`, la donnée est perdue !).
Vous pouvez consulter la documentation de l'instruction `mul` sur la page principale si vous souhaitez en savoir plus.
Là encore, vous devez utiliser GDB pour vérifier que votre programme fonctionne. Vous n'avez pas besoin de vérifier que la multiplication ne déborde pas des 64 bits de `%rax`.

Génération de nombres aléatoires

Dans cet exercice, vous pouvez utiliser le registre `%r10` (ou ses fractions) en plus du registre `%rax`.
On va implanter un registre à rétro-action linéaire pour générer des nombres pseudo-aléatoires. Pseudo-aléatoire signifie que les nombres qu'on va générer seront en fait périodiques : au bout d'un certain nombre d'itération du générateur, on reviendra sur une valeur déjà trouvée.
Il y a deux fonctions à implanter en assembleur dans un fichier `fct_lfsr.s` :

- la fonction `void init_aleat(void)` qui initialise le générateur ;
- la fonction `uint16_t aleat(void)` qui renvoie à chaque appel un nouveau nombre aléatoire (sur 16 bits dans cet exercice).

Le programme principal de test est fourni dans le fichier `lfsr.c`. Ce fichier contient notamment la déclaration de la variable globale `etat` qui contient la valeur courante du générateur et qu'on va devoir mettre à jour depuis les fonctions assembleur.
Le générateur étant pseudo-aléatoire, les nombres qu'on va énumérer le seront toujours dans le même ordre : il est donc important de démarrer à un point imprévisible de la séquence, sinon on obtiendra toujours la même succession de valeurs.
Pour implanter la fonction `init_aleat`, vous utiliserez l'instruction `rdtsc` de l'architecture Intel. Cette instruction renvoie dans `%edx:%eax` le nombre de cycles écoulés depuis le démarrage de la machine. Vous utiliserez donc cette valeur pour initialiser la variable `etat`.
Pour la fonction `aleat`, il suffit d'implanter la formule suivante pour mettre à jour la valeur de l'état à chaque appel : `etat = (etat >> 1) ^ (-(etat & 1) & 0xB400)`.
On rappelle que `^` est l'opérateur XOR en C.