Guide as 32bits Pour l'Impatient(e)

"as - the portable GNU assembler.", /usr/bin/man 2005.
"Le Guide *as* Pour l'Impatient(e) permet à chacun d'évaluer les limites de ses propres réflexions sur l'optimisation des programmes.", nj 2004.

Point de départ : une opération (souvent un calcul) à optimiser avec soins.
Objectif : définir cette opération par une succession d'instructions pour contrôler avec précision son exécution. Il est courant pour cela de recourrir à la programmation **asm**(*i.e.* assembleur).

1 Compilation pas à pas

La programmation **asm** commence par l'écriture d'un programme ou d'une fonction en assembleur. Pour un programme **asm** dans un fichier source nommé **prog.s**, la compilation peut se décomposer en deux phases :

- l'assemblage ou la traduction du fichier source **prog1.s** en fichier objet (un fichier objet est une succession de codes hexadécimaux représentant des instructions pour le processeur) ; cette phase peut se réaliser à l'aide de la commande **as --s prog1.s -o prog1.o**
- l'édition de liens ou le regroupement de fichier(s) objet(s) en un fichier éxecutable ; Cette phase peut se réaliser à l'aide de la commande **ld prog1.o -o prog1**

lien utile :

- http ://webster.cs.ucr.edu/AoA/index.html

2 Registres d'un processeur

Les processeurs 32 bits d'Intel (*big-endian*) sont composés de :

- quatre registres à usages généraux sur 32 bits, nommés **%eax**, **%ebx**, **%ecx** et **%edx**, adressés sur leur 16 premiers bits sous les noms **%ax**, **%bx**, **%cx** et **%dx** (eux-même adressés en deux parties de 8 bits sous les noms **%ah**, **%al**, **%bh**, **%bl**, **%ch**, **%cl**, **%dh** et **%dl**)
- quatre registres spécifiques sur 32 bits, nommés **%esi**, **%edi**, **%esp** et **%ebp**, adressés sur leur 16 premiers bits sous les noms **%si**, **%di**, **%sp** et **%bp**
- six registres de segments sur 16 bits, nommés **%cs**, **%ss**, **%ds**, **%es** , **%fs** et **%gs** (peu utilisés dans les architectures 32 bits)
- un registre de contrôle sur 32 bits nommé **%eflags**, composé de 8 bits utiles à la programmation ;

  %eflags :

			...				of	df	if		sf	zf		af		pf		cf

  par ordre de position, ces 8 bits sont : **cf** en 0 (carry flag ou bit de retenue, mis à 1 quand une opération produit une retenue), **pf** en 2 (parity flag ou bit de parité, mis à 1 quand un résultat contient un nombre pair de bits à 1), **af** en 4 (auxiliary flag ou bit de retenue auxiliaire), **zf** en 6 (zero flag ou bit de valeur nulle, mis à 1 quand un résultat vaut zéro), **sf** en 7 (sign flag ou bit de signe, mis à 1 quand un résultat est négatif), **if** en 9 (interruption flag ou bit d'interruption), **df** en 10 (direction flag ou bit de direction, indique la direction des opérations sur les chaînes de caractères) et **of** en 11 (overflow flag ou bit de débordement, mis à 1 quand le signe du résultat est l'inverse du signe attendu)
- un registre de statut sur 32 bits nommé **%eip**, contenant l'adresse de la prochaine instruction à exécuter

3 Arithmétique, logique et déplacement

Le format général est **OpCode[b,w,l] op1 [, op2]**. Le suffixe des instructions présentées indique la taille des opérandes associés : **b** pour 8 bits, **w** pour 16 bits et **l** pour 32 bits.

- **movl %reg1, %reg2** : copie le contenu du registre **%reg1** dans le registre **%reg2**
- **incl %reg** : incrémente le contenu du registre **%reg**
- **decl %reg** : décrémente le contenu du registre **%reg**
- **addl %reg1,%reg2** : **%reg2** reçoit la somme des contenus des registres **%reg1** et **%reg2**
- **subl %reg1,%reg2** : **%reg2** reçoit le contenu de **%reg2** moins le contenu de **%reg1**
- **cmpl %reg1,%reg2** : réalise **subl** sans modifier le contenu de **%reg2**
- **imull %reg1,%reg2** : **%reg2** reçoit le contenu de **%reg1** multiplié par le contenu de **%reg2**
- **idivl %reg** : **%eax** reçoit le quotient de la division de **%eax** par **%reg** et **%edx** reçoit le reste de cette division
- **sarl $1,%reg** : décale 1 fois le contenu de **%reg** vers la droite (s'écrit également **sarl %reg**)
- **sall $1,%reg** : décale 1 fois le contenu de **%reg** vers la gauche (s'écrit également **sall %reg**)
- **xorl %reg1,%reg2** : **%reg2** reçoit le ou-exclusif de **%reg1** et de **%reg2**
- **andl %reg1,%reg2** : **%reg2** reçoit le et-logique de **%reg1** et de **%reg2**
- **orl %reg1,%reg2** : **%reg2** reçoit le ou-logique de **%reg1** et de **%reg2**
- **notl %reg** : **%reg** reçoit le non-logique de **%reg**

4 Instructions de branchement

- **jmp LABEL** : passe sans condition à l'instruction placée en **LABEL**
- **jb LABEL** : passe en **LABEL** si **cf** = 1 (*Jump if Below* équivaut à **jnae** pour *Jump if Not Above or Equal* et **jc** pour *Jump if Carry*)
- **je LABEL** : passe en **LABEL** si **zf** = 1 (*Jump if Equal* équivaut à **jz** pour *Jump if Zero*)
- **jo LABEL** : passe en **LABEL** si **of** = 1 (*Jump if Overflow*)
- **jp LABEL** : passe en **LABEL** si **pf** = 1 (*Jump if Parity* équivaut à **jpe** pour *Jump if Parity Even*)
- **js LABEL** : passe en **LABEL** si **sf** = 1 (*Jump if Signed*)
- **ja LABEL** : passe en **LABEL** si **cf** = **zf** = 0 (*Jump if Above* équivaut à **jnbe** pour *Jump if Not Below or Equal*)
- **jg LABEL** : passe en **LABEL** si **zf** = 0 et **sf** = **of** (*Jump if Greater* équivaut à **jnle** pour *Jump if Not Less or Equal*)
- **jge LABEL** : passe en **LABEL** si **sf** = **of** (*Jump if Greater or Equal* équivaut à **jnl** pour *Jump if Not Less*)
- **jl LABEL** : passe en **LABEL** si **sf** = **of** = 1 (*Jump if Less* équivaut à **jnge** pour *Jump if Not Greater or Equal*)
- **jle LABEL** : passe en **LABEL** si **zf** = 1 ou **sf** = **of** = 1 (*Jump if Less* équivaut à **jng** pour *Jump if Not Greater*)

5 Directives

- **.data** : déclare le segment de données initialisées (**.bss** si non initialisées)
- **.text** : déclare le segment de texte
- **mot :** : déclare un label (ou une étiquette) de branchement appelé **mot**
- **.globl : _start** déclare le label **_start** comme point d'entrée du programme

Dans un segment de données :

- **.long 7** : déclare un entier sur 32 bits de valeur 7
- **.string "abc"** : déclare une chaîne de caractères **abc**
- **.float 0.1** : déclare un flottant de valeur **0.1**
- **.long 7,13** : déclare deux entiers sur 32 bits
- **.space 8** : déclare un espace de 8 octets

– **v1 : .long 7** : déclare un entier à l'adresse définie par le label **v1**

6 Accès mémoire

Pour **%reg1** = 0x01, **%reg2** = 0x03 :
– **(%reg1)** : est la valeur située à l'adresse 0x01
– **16(%reg1)** : est la valeur située à l'adresse 0x11
– **(%reg1, %reg2)** : est la valeur située à l'adresse 0x04
– **(%reg1, %reg2, 4)** : est la valeur située à l'adresse 0x0D
– **16(%reg1, %reg2, 4)** : est la valeur située à l'adresse 0x1D

Après déclaration d'un label **L0 : .long 1** :
– **movl L0, %reg** : copie 1 dans le registre **%reg**
– **movl $L0, %reg** : copie l'adresse de la valeur égale à 1 dans le registre **%reg** (l'instruction **leal** permet d'accéder à l'adresse d'une valeur *i.e.* **leal L0, %reg**)

7 Valeurs constantes

La déclaration des valeurs constantes peut précéder les deux segments du programme. L'utilisation d'une constante est obligatoirement précédée de sa déclaration.
– **v1 = 13** : déclare une constante égale à 13
– **movl v1, %reg** : copie 13 dans le registre **%reg**
– **movl $v1, %reg** : copie l'adresse de la valeur v1 dans le registre **%reg**
– **movl $12, %reg** : copie 12 dans le registre **%reg**

8 Pile et fonction

La pile est gérée par les registres **%esp** (dernière valeur empilée) et **%ebp** (base de la pile). Elle croit dans le sens décroissant des adresses.
– **pushl %reg** : empile le contenu de **%reg**
– **popl %reg** : dépile la tête de la pile et la copie dans **%reg**

Pour chaque fonction, l'espace appelé *stack frame*, délimité par **%esp** et **%ebp**, définie l'espace associé aux variables locales. La définition d'un *stack frame* de 3 variables (*i.e.* 12 bits) corresponds à l'exécution des instructions suivantes :
– **pushl %esp** : sauvegarde la base de la pile
– **movl %esp, %ebp** : déplace la base de la pile
– **subl $12, %esp** : réserve 12 bits dans la nouvelle pile

Les trois instructions spécifiques aux appels de fonction sont :
– **call L0** : appelle la fonction associée au label **L0** (*i.e.* empile **%eip** puis place **%eip** sur la première instruction de la fonction associée au label **L0**)
– **leave** : restaure le cadre de pile précédent (*i.e.* **movl %ebp,%esp** puis **popl %ebp**)
– **ret** : définie la fin d'un appel de fonction (*i.e.* **popl %eip**) ; la valeur de retour éventuelle est posée dans **%eax**

Les paramètres d'une fonction sont empilés dans l'ordre inverse avant exécution de **call**. Dans la fonction appelée :
– **4(%ebp)** : adresse de retour de la fonction
– **8(%ebp)** : 1er paramètre de la fonction
– **12(%ebp)** : 2ième paramètre de la fonction
– **4+4*n(%ebp)** : Nième paramètre de la fonction

9 Appel système

Les appels systèmes sont accessibles via l'interruption **0x80**. L'instruction **int 0x80** permet d'effectuer un appel système défini par les registres suivants :

- **%eax** : numéro de l'appel système (numéros listés dans /usr/include/asm/unistd.h)
- **%ebx, %ecx, %edx, %esi, %edi** : 5 premiers paramètres

10 Programme asm de bienvenue

```
# affichage d'un message sur la sortie standard
# gcc bonjour.s -o bonjour

.data   # segment de donnees
msg:    .ascii  "bonjour!\n"
        taille = . - msg # taille en octets depuis msg

.text   # segment d'instructions
        # le linker necessite le point d'entree _start
        # ld -e npe permet de redefinir le point d'entree en "npe"
        # gcc necessite par defaut main
.global main
main:
        movl $4,%eax
        movl $1,%ebx
        movl $msg,%ecx
        movl $taille,%edx
        int     $0x80

        movl    $1,%eax
        movl    $0,%ebx
        int     $0x80
```

11 Interaction entre fonction asm et programme C

```
# fonction d'addition assembleur
# n - 2005

.text
.globl asmAdd
asmAdd:
 pushl %ebp # sauvegarde ebp
 movl %esp, %ebp
 pushl %ecx # sauvegarde ecx et ebx
 pushl %ebx
 movl 8(%ebp), %ebx    # 1er arg
 movl 12(%ebp), %ecx   # 2eme arg

 addl %ebx, %ecx
 movl %ecx, %eax # retour = eax

 popl %ebx  # restaure ebx et ecx
 popl %ecx
 movl %ebp, %esp
 popl %ebp  # restaure ebp
 ret
```

```
/* programme C
 * utilisant une fonction asm
 *
 * gcc -c foncAsmAdd.s -o foncAsmAdd.o
 * gcc appAsmAdd.c foncAsmAdd.o -o appAsmAdd
 *
 * n - 2005
 */

extern int asmAdd(int, int);

int main(void) {
  int a = 10, b = 11, c = 0;
  c = asmAdd(10,11);
  printf("10+11= %d\n", c);
  return 0;
}
```