Fiche de référence assembleur

Rémi Bonnet

1 Instructions assembleurs

1.1 Opérateurs binaires

- **addl** ⟨**source**⟩**,** ⟨**dest**⟩ : Effectue *dest*=*source*+*dest*
- **andl** ⟨**source**⟩**,** ⟨**dest**⟩ : Effectue *dest*=*source* & *dest*
- **leal** ⟨**source**⟩**,** ⟨**dest**⟩ : Effectue *dest*=&*source* (ceci charge l'adresse de source dans dest)
- **movl** ⟨**source**⟩**,** ⟨**dest**⟩ : Effectue *dest*=*source*
- **imull** ⟨**source**⟩**,** ⟨**dest**⟩ : Effectue *dest*=*source*∗*dest*
- **orl** ⟨**source**⟩**,** ⟨**dest**⟩ : Effectue *dest*=*dest*|*dest*
- **subl** ⟨**source**⟩**,** ⟨**dest**⟩ : Effectue *dest*=*dest*−*source*
- **xorl** ⟨**source**⟩**,** ⟨**dest**⟩ : Effectue *dest*=*source* xor *dest*

1.2 Sauts conditionnels

Un saut conditionnel comprend deux instructions : une instruction de comparaison, suivie immédiatement d'une instruction de saut.

Exemple :

```
cmp %eax , %ebx
je instr
```

Un saut vers **instr** sera effectué si **%eax = %ebx**.

- **cmp** ⟨**source**⟩**,** ⟨**dest**⟩. Compare *source* et *dest*.
- **je** ⟨**target**⟩. Saute à ⟨target⟩ si la comparaison précédente vérifiait $dest = source$
- **jg** ⟨**target**⟩. Saute à ⟨target⟩ si la comparaison précédente vérifiait $dest > source$
- **jge** ⟨**target**⟩. Saute à ⟨target⟩ si la comparaison précédente vérifiait $dest \geq source$
- **jl** ⟨**target**⟩. Saute à ⟨target⟩ si la comparaison précédente vérifiait $dest < source$
- **jge** ⟨**target**⟩. Saute à ⟨target⟩ si la comparaison précédente vérifiait $dest \leq source$
- **jmp** ⟨**target**⟩. Saute à ⟨target⟩.

1.3 Manipulation de pile, fonctions

- **pushl** ⟨**source**⟩ : Ajoute *source* sur le dessus de la pile.
- **popl** ⟨**dest**⟩ : Récupère le sommet de la pile dans *dest*.
- **call** ⟨**target**⟩ : Place l'adresse de l'instruction suivante sur le sommet de la pile, puis fait un saut vers *target*.
- **leave** : Macro pour **movl %ebp, %esp** suivi de **popl %ebp**.
- **ret** : Récupère la valeur du sommet de la pile et fait un saut vers cette valeur.

1.4 Division

Une division doit être effectuée en deux étapes. **cltd** prépare la division en maintenant le contenu de %eax à cheval sur %eax et %edx. **idivl** ⟨**div**⟩ effectue la division de %eax par *div*, place le quotient dans

%eax et le reste dans %edx.

```
movl arg, %eax
ctld
idivl %ebx
```

- **ctld** : Met le contenu de %eax à cheval sur %eax et %edx.
- **idivl** ⟨**div**⟩ : Divise un nombre à cheval sur %eax et %edx par *div*, met le quotient dans %eax et le reste dans %edx.

2 Accéder à la mémoire

Dans les instructions, ⟨source⟩ et ⟨dest⟩ peuvent être :
- **%reg**, un registre (eax, ebx, ecx, edx, esp, ebp).
- **n(%reg)**, le contenu à l'adresse $reg + n$
- **n(%reg1, %reg2, p)**, le contenu à l'adresse $reg1 + reg2 * p + n$. Attention, il n'est pas possible d'utiliser cette syntaxe à la fois pour ⟨source⟩ et ⟨dest⟩.
- **label**, le contenu pointé par *label*.
- **$label**, l'adresse pointée par *label*.
- **$n**, une constante n (en décimal).
- **$0xhex**, une constante *hex* (en hexadécimal).

3 Convention d'appel des fonctions

Plusieurs convention d'appel sont possibles. Nous suggérons (fortement) la convention suivante :
- La sauvegarde des registres est à la charge de l'appelant.
- Les paramètres sont passés sur la pile.
- La valeur de retour est stockée dans %eax.

Nous organiserons la pile de la manière décrite dans la figure 1, qui est la convention unaniment utilisée. Ceci consiste à utiliser la forme suivante pour les fonctions :

```
fonction:
        pushl %ebp
        movl %esp, %ebp

        ; Corps de la fonction

        leave
        ret
```

(1) correspond à l'état de la pile avant l'appel. Après empilement des paramètres, nous sommes en (2), et après l'appel suivi de **pushl %ebp** et **movl %esp, %ebp**, la pile ressemble à (3). Les variables locales peuvent alors être allouées selon (4). Enfin, après **leave** et **ret**, nous sommes de retour en (2).

On notera que dans le corps de la fonction, les paramètres peuvent être accédés par **8(%ebp)**, **12(%ebp)**, etc... et que les variables locales peuvent être accédées par **0(%ebp)**, **-4(%ebp)**, etc...

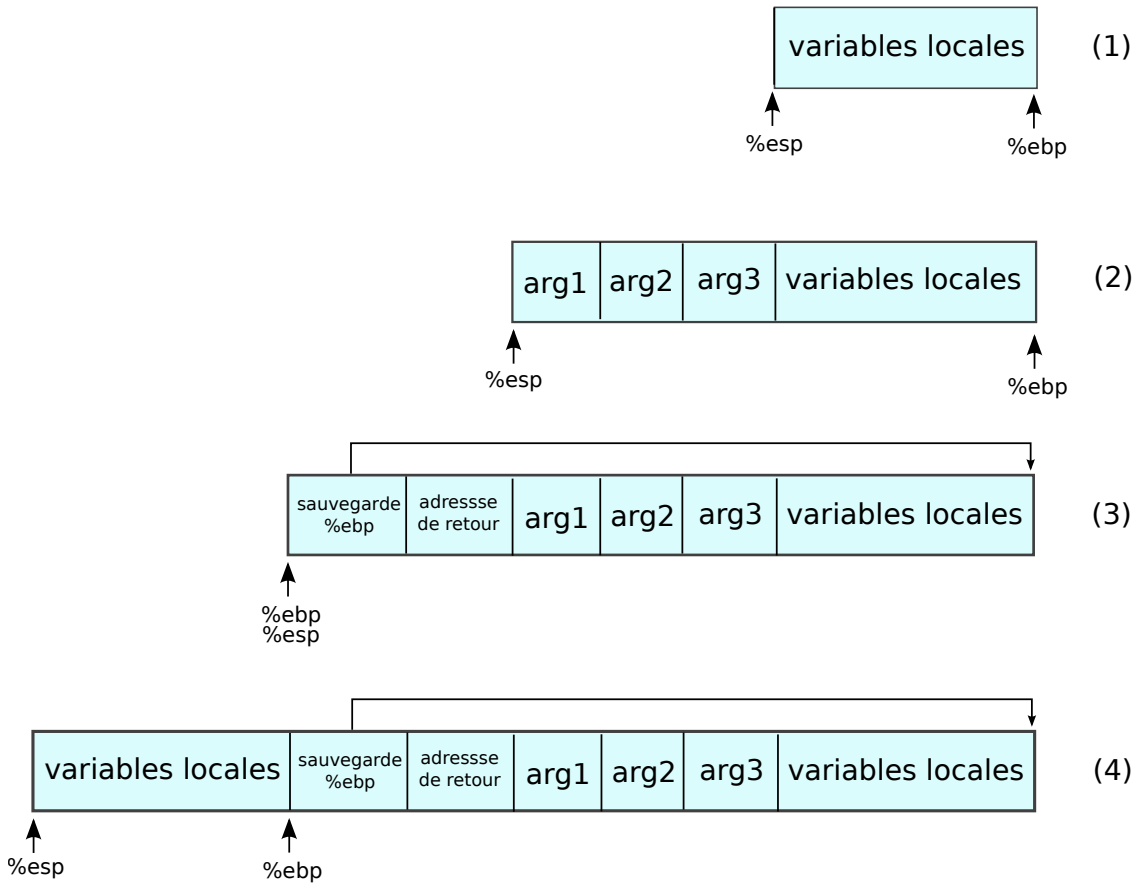

FIGURE 1 – Structure de la pile au cours d'un appel de fonction.

4 Pseudo-opérations

Les pseudo-opérations sont des commandes destinées à l'assembleur lui indiquant comment assembler les instructions et données qui se trouvent dans votre programme.

- **.align** ⟨**size**⟩ : Demande à l'assembleur de sauter autant d'octets que nécessaire pour que l'adresse suivante soit un multiple de *size*. Les processeurs 32-bits ont tendance à apprécier quand toutes les adresses qu'ils manipulent sont des multiples de 4. Certains peuvent même refuser de travailler avec des adresses non-alignées.
- **.ascii** ⟨**chaine**⟩ : Place la chaîne de caractère *chaine*, sans zéro terminal.
- **.asciz** ⟨**chaine**⟩ : Place la chaîne de caractère *chaine*, avec un zéro terminal.
- **.bss** : Déclare une section bss. Cette section est définie comme contenant uniquement des zéros. Utile pour allouer un large espace de stockage (vous ne devriez toutefois pas en avoir besoin).
- **.comm** ⟨**name**⟩**,** ⟨**size**⟩ : Déclare un symbole commun, nommé *name*, de taille *size*. Si un symbole commun avec le même nom est déclaré plusieurs fois, un seul emplacement sera quand même alloué.
- **.data** : Déclare une section de données. Les sections de données peuvent être lues et écrites à l'exécution. Cette instruction peut être utilisée plusieurs fois. Dans ce cas, toutes les sections de données seront fusionnées en une seule.
- **.global** ⟨**symbol**⟩ : Déclare un symbole externellement visible. Les symboles représentant des fonctions qui peuvent être appelées de programme extérieurs doivent être déclarés globaux.
- **.long** ***val*** : Déclare un entier initialisé à *val*.
- **.string** ***chaine*** : Place la chaîne de caractère *chaine*, avec un zéro terminal.
- **.text** : Déclare une section de code. Les sections de code ne sont pas sensées être écrivables à l'exécution. Cette instruction peut être utilisée plusieurs fois. Dans ce cas, toutes les sections de code seront fusionnées en une seule.
- **.type** ⟨**symbol**⟩**,** ⟨**type**⟩ : Déclare *symbole* comme ayant le type *type* (utilisé par exemple pour les débogueurs). *type* peut être **@object** ou **@function**.

5 Appels systèmes

Un appel système Linux est émis avec l'instruction **int $0x80**. Le type d'appel système doit être stocké dans %eax, les paramètres de l'appel système dans %ebx, %ecx, %edx (dans l'ordre).

Voici rapidement deux appels systèmes utiles :

- ID=1 : **exit**. Paramètres :
  - **%ebx** : Valeur de retour du programme.
- ID=4 : **write**. Paramètres :
  - **%ebx** : Canal d'écriture (1 = stdout, 2 = stderr)
  - **%ecx** : Adresse de la chaîne à écrire.
  - **%edx** : Longueur de la chaîne à écrire.

À noter que vous devriez éviter d'utiliser des appels systèmes. Appeler les fonction C associées (exit, printf,...) est bien plus simple, et portable sur d'autres systèmes.

6 Structure de base d'un programme assembleur

```
.data                           # Debut de la section data (variables).
hello:  .string "hello world!\n"
.align 4                        # La prochaine variable doit etre alignee
len:    .long 13

.text                           # Debut du code
main:
```

```
.global main
.type main, @function
        movl $4, %eax
        movl $1, %ebx
        leal hello, %ecx     # Place l'adresse de hello dans ecx
        movl len, %edx       # Place le contenu de len dans edx
        int $0x80            # Appel systeme

        pushl $0
        call exit            # Appel d'une fonction C
```