# TP6: assembleur x86-64

Lucca Hirschi

19 et 20 octobre 2016

d'après un sujet de David Baelde

Cette semaine constitue la dernière étape de notre descente vers les entrailles de nos machines : nous allons pratiquer un peu l'assembleur x86 64bits. Ce sera le langage cible du projet compilation, qui commencera juste après les vacances.

---

**Exercice - 1** *Observer*

---

On va commencer par observer et comprendre l'assembleur généré par GCC à partir d'un programme simple. Dans un répertoire neuf dédié à ce TP, récupérez sur la page du TP le fichier d'exemple `aha_write.c` ainsi que le Makefile fourni. Ce dernier permet d'exécuter automatiquement les bonnes commandes pour convertir le C en assembleur puis en binaire exécutable. Les lignes pertinentes pour cela sont les suivantes :

```
# Pour produire un fichier xxx.s a partir de xxx.c,
# faire "gcc ..." avec $+ = "xxx.c" et $@ = "xxx.s".
%.s: %.c
	gcc -S -fno-asynchronous-unwind-tables $< -o $@
# Meme principe pour compiler un .s en binaire.
%: %.s
	gcc -ggdb $+ -o $@
```

Le premier appel de GCC comporte, outre l'option `-S` qui dit de produire de l'assembleur, une option au nom interminable qui a pour effet de produire un code assembleur aussi simple que possible. Le second appel est déja connu, il permet de produire un binaire en n'oubliant pas de demander les informations de déboguage.

**1-** Compiler le code C en faisant `make aha_write`, ou simplement `make` si `aha_write` est indiqué dans la variable `BINARIES` du Makefile. Ouvrez le fichier `aha_write.s` et essayez de vous y repérer un peu — attention, ne modifiez ce fichier qu'après l'avoir renommé, car il risque sinon d'être regénéré à partir du source C.

L'utilisation des registres `%rsp` et `%rbp` devrait vous parler. Dans tous les cas, n'hésitez pas à ouvrir, à portée de main, le mémo x86-64 disponible sur la page du TP, ainsi que la feuille de TD pointée dans ce mémo.

À noter : On utilise `write` plutôt que `printf` car cette dernière fonction a des arguments variables, et donc une convention d'appel plus complexe. Il peut être utile de jeter un oeil à `man 2 write` pour comprendre le sens des trois arguments de cette fonction.

**2-** Lancer `ddd aha_write`. Mettre en place l'environnement de déboguage comme suit :

- Mettez un breakpoint (clic droit) au début de la fonction `main`.
- Lancez l'exécution (F2).
- Affichez l'état des registres (Status / Registers).
- Activez si besoin la zone d'affichage des données (View / Data Window).
- Affichez 16 octets, en hexadécimal, à partir de l'adresse `$rsp` (Data / Memory, utiliser le bouton Display pour valider).

Exécutez enfin le programme, pas à pas (F5), en essayant de prévoir avant chaque pas ce qui va changer dans les données affichées, ainsi que les sauts dans le code.

---

**Exercice - 2** *Modifier*

---

Nous nous proposons de faire quelques modifications au code assembleur précédent. Copiez le fichier `aha_write.c` en `my_aha.c`, pour éviter de modifier un fichier (qui sera peut être re)généré.

**1-** Comme vous pouvez le remarquer, GCC stocke le compteur sur 4 octets (car il est déclaré `int` et non `long int`). Modifiez le code pour le stocker sur 8 octets.

Les conventions d'appel x86-64 ne sont pas contraignantes sur l'utilisation du registre `%rbp`. Il est commun de l'utiliser, comme GCC, pour désigner le début de la *frame*, c'est à dire de la partie de la pile qui concerne la fonction en cours d'exécution. Mais ce n'est pas nécessaire ; ce choix n'impacte pas l'interopérabilité entre fonctions.

**2-** Modifiez le code pour ne plus utiliser le registre `%rbp` pour accéder au compteur `i`. À la place, on le désignera par décalage à partir de `%rsp`. Compilez (on pourra ajouter `my_aha` à la variable `BINARIES` dans le Makefile) et testez, déboguez si besoin comme vu précédemment.

**3-** On veut maintenant cesser d'utiliser `%rbp` pour stocker la valeur de `%rsp` après la sauvegarde de la valeur initiale de `%rbp` via `pushq %rbp`. On va donc supprimer l'instruction `movq %rsp %rbp`…mais si l'on ne fait que ça, cela ne "marche plus" ; que faut-il modifier/supprimer aussi ?

**4-** Utiliser le registre `%rbp` pour stocker le compteur `i`, actuellement stocké sur la pile, ce qui constitue un gain d'efficacité. Faut-il prendre des précautions au moment de l'appel à `write` ? peut-on se passer du `pushq %rbp` au début ?

---

**Exercice - 3** *Bonus : encore plus bas, les appels système*

---

Le code assembleur vu jusqu'ici fait appel à des fonctions de la librairie standard C (en l'occurrence `write`) et définit une fonction `main` qui n'est pas le "vrai" point d'entrée du binaire produit : en réalité cette fonction est appelée par une routine qui s'occupe de passer `argc` et `argv` comme il faut, et de terminer l'exécution après retour de la fonction.

Mais nous pouvons nous passer de tout cela !

**1-** La fonction `write` est un emballage assez direct au dessus de l'*appel système* du même nom. Un appel système est une fonctionalité très primitive fournie par le noyau. Pour faire un tel appel on met le code de l'appel dans le registre `%rax` (pour write c'est 1), les arguments de l'appel dans les registres d'arguments comme pour un appel de fonction (les arguments de l'appel système write sont les mêmes que pour la fonction) et enfin on utilise l'instruction `syscall`. Débarassez-vous ainsi de l'appel de la fonction `write`.

**2-** Nous allons maintenant nous débarasser de la fonction "intermédiaire" `main`. Renommez votre fichier avec une extension `.S` au lieu de `.s` ; le Makefile contient une recette différente pour cette extension, qui demande à gcc de compiler un code assembleur "nu" sans aucune référence à la librairie standard C. Vous pouvez maintenant renommer `main` (si le coeur vous en dit), supprimer les instructions de gestion de pile et de retour de fonction (puisque ce n'est plus vraiment une fonction). Compilez, testez. Normalement vous observez une erreur de segmentation, due au fait que l'exécution ne s'arrête pas (et va chercher à exécuter du code qui n'existe pas ou est mal défini). Remédiez à cela en effectuant l'appel système exit (code 60) avec 0 pour unique argument.

---

**Exercice - 4** *Programmons un peu*

---

Il s'agit maintenant d'écrire quelques algorithmes simples directement en assembleur. Vous pouvez partir d'une structure de code généré à partir d'un programme C vide, qui définit juste une fonction `main` avec ses deux arguments.

**1-** Codez une fonction `putstr` qui prend comme unique argument l'adresse d'une chaîne de caractère, calcule sa longueur, et l'affiche au moyen d'un appel à `write`. Testez avec une fonction main qui fait un `print` de `argv`.

**2-** Ecrivez une fonction `putint` qui prend en argument un entier et qui l'affiche en hexadécimal.

**Exercice - 5** *Encore un mini-compilo*

**1-** Ecrivez une fonction OCaml `compile_aux` de type `propf -> (string*int) list -> unit` qui prend une formule propositionnelle (ex. $x \vee (y \wedge (x \vee z))$) ainsi qu'un environnement qui à chaque formule associe son *offset* depuis `%rbp` et affiche le code assembleur qui permet d'évaluer cette formule. Le type des propositions et des listes d'associations se trouvent en annexe dans les figures 1 et 2.

**2-** Ecrivez une fonction `compile` en OCaml qui prend en argument une formule propositionnelle et qui renvoie le code assembleur qui demande des valeurs pour chaque variable puis qui évalue la formule pour cette valuation. Le code que j'obtiens avec ma version est donnée dans l'annexe.

# A Types

```
type propf =
  | Var of string
  | Or of boolexpr * boolexpr
  | And of boolexpr * boolexpr
  | Not of boolexpr
```

Figure 1 – Type des formules

```
(string * int) list
```

Figure 2 – Type des listes d'assocations

# B Exemple d'output de ma fonction `compile`

```
.data
say:
        .string "Variable values:\n"
        .align  8
var:
        .string "  %s: %d\n"
        .align  8
res:
        .string "Result: %d\n"
        .align  8
x1:
        .asciz  "x1"
        .align  8
x2:
        .asciz  "x2"
        .align  8
x3:
        .asciz  "x3"
        .align  8

.text
        .global main
main:
        .type   main, @function
        pushq   %rbp
```

```
        movq    %rsp, %rbp
        subq    $48, %rsp       # make room for 3 pvars + 3 local vars
        movq    24(%rbp), %rax  # load argv
        movq    8(%rax), %rax   # load *argv[1]
        movq    %rax, %rdi
        movq    $0, %rax
        call    atoi
        movq    %rax, -24(%rbp) # save x1
        movq    24(%rbp), %rax  # load argv
        movq    16(%rax), %rax  # load *argv[2]
        movq    %rax, %rdi
        movq    $0, %rax
        call    atoi
        movq    %rax, -16(%rbp) # save x2
        movq    24(%rbp), %rax  # load argv
        movq    24(%rax), %rax  # load *argv[3]
        movq    %rax, %rdi
        movq    $0, %rax
        call    atoi
        movq    %rax, -8(%rbp)  # save x3
        movq    $say,%rdi
        movq    $0, %rax
        call    printf
        movq    -24(%rbp), %rax
        movq    %rax, 16(%rsp)
        movq    %rax, %rdx
        movq    $x1, %rsi
        movq    $var, %rdi
        movq    $0, %rax
        call    printf          # print x1
        movq    -16(%rbp), %rax
        movq    %rax, 16(%rsp)
        movq    %rax, %rdx
        movq    $x2, %rsi
        movq    $var, %rdi
        movq    $0, %rax
        call    printf          # print x2
        movq    -8(%rbp), %rax
        movq    %rax, 16(%rsp)
        movq    %rax, %rdx
        movq    $x3, %rsi
        movq    $var, %rdi
        movq    $0, %rax
        call    printf          # print x3
# Compile_aux:
        movq    -24(%rbp), %rax
        pushq   %rax
        movq    -16(%rbp), %rax
        popq    %rbx
        orq     %rbx, %rax
        pushq   %rax
        movq    -24(%rbp), %rax
        pushq   %rax
        movq    -8(%rbp), %rax
        popq    %rbx
        orq     %rbx, %rax
        popq    %rbx
        andq    %rbx, %rax
        pushq   %rax
        movq    -16(%rbp), %rax
        pushq   %rax
        movq    -8(%rbp), %rax
```

```
        popq    %rbx
        andq    %rbx, %rax
        popq    %rbx
        orq     %rbx, %rax
# Affchage du resultat:
        movq    %rax, %rsi
        movq    $res, %rdi
        movq    $0, %rax
        call    printf
        movq    8(%rsp), %rax
        call    exit
```