# TP - 7/8

15/16 octobre 2014

On se rapproche encore de la machine avec le langage assembleur (pour architecture x86-64). On commence par exécuter et étudier de l'assembleur généré par *gcc* à partir d'un programme C à l'aide de *ddd*. Ensuite, ça sera à vous d'écrire de l'assembleur.

Rappel : le projet (qui commence juste après les vacances) consiste à écrire en OCaml un compilateur de *C--* vers de l'assembleur x86-64.

## 1 A partir de codes C

On commence en douceur avec l'assembleur généré par *gcc* et *ddd*.

---

**Exercice - 1** *Aperçu*

---

**1-** Récupérez le petit programme C à cette adresse. Compilez le avec les options

```
-S -fno-asynchronous-unwind-tables
```

qui permettent de générer non pas du code machine mais de l'assembleur (la seconde option sert à rendre le code produit plus concis). Vous obtenez un fichier `max.s`. Repérez dans ce fichier la fonction `max`, les variables $x$ et $y$ et les chaînes de caractères affichées (un peu d'aide : suivez les labels ainsi que les instructions `jle` et `jmp`).

**2-** Lancez *ddd* sur ce programme (rappel : vous devez compilez avec l'option `-ggdb` et lancez *ddd* sur l'exécutable). Dans le menu Source sélectionnez Machine Code, et agrandissez la fenêtre contenant l'assembleur. Placez un break point au début de la fonction `main`. Vous pouvez afficher les valeurs des registres en allant dans Sources puis Registers. Lancez le programme, puis appuyez sur `Nexti` qui a pour effet d'exécuter une instruction asssembleur à la fois. Le code assembleur est un peu différent, reconnaisez-vous la structure du code ? (un peu d'aide : 10 vaut $0xa$)

## 2 A la main

A vous de jouer. Vous allez avoir besoin du cours de Jean Goubault (surtout la leçon 2 et son annexe) et sans doute cette fiche de référence. Attention, souvenez-vous qu'en x86-64, les registres sont un peu différents : un petit résumé. De même, comme l'a expliqué Jean, les instructions 64 bits terminent par un "q" et non par un "l".

Vous aurez besoin d'écrire des fonctions. Vous pouvez suivre la convention de votre choix : soit vous utilisez le maximum de registres pour les paramètres, soit vous les passez par la pile. Faites cependant attention aux fonctions C, vous devez utiliser en priorité les 6 registres pour paramètres.

---

**Exercice - 2** *Print*

---

**1-** En utilisant l'appel système *write*, écrivez un "Hello World" (sans utiliser `printf`).

**2-** Ecrivez une fonction `myprint_len` en assembleur qui prend en argument l'adresse d'une chaîne de caractères ainsi que sa taille et qui affiche cette chaîne (sans utiliser `printf`). Ecrivez à la suite, une fonction `main` qui teste votre fonction.

**3-** Ecrivez une fonction `myprint_int` qui prend en argument un entier et qui l'affiche.

• Vous aurez sans doute besoin de `printf`. On appelle cette fonction avec `call printf`. Vous devez bien utiliser `rdi` en premier argument et `rsi` en second. Avant d'appeler `printf`, assurez vous que `rax` vaut 0.

**4-** (*) Ecrivez une fonction `mystrlen` qui prend en argument une chaîne de caractère terminant par $0x0$ et qui renvoie sa taille. Testez en écrivant un `main` qui appelle cette fonction et qui affiche son résultat.

• Vous aurez sans doute besoin de `movzbq` qui déplace un octet vers sa destination et remplit les octets manquants par des 0. Vous aurez également besoin d'une boucle. Construisez la en utilisant les instructions `cmp`, `je` et `jmp`.

**5-** (*) Vous pouvez maintenant écrire une fonction `myprint` qui prend en argument une chaîne de caractères et qui l'affiche (sans utiliser `printf`).

---

**Exercice - 3** *Input*

---

**1-** (*) Ecrivez uen fonction `askval` qui demande à l'utilisateur une valeur (0 ou 1) et qui l'affiche en retour.

• Vous pouvez utiliser la fonction `getline` (même ussage qu'en C).

# 3 Avec OCaml

---

**Exercice - 4** *Génération d'assembleur*

---

**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

**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
```