Assembleur

hondet

November 5, 2020

Contents

1 Différents assembleurs et différentes syntaxes

Il existe plusieurs assembleurs, comme le NASM, le YASM, le MASM &c. Nous utiliserons le GNU Assembler, noté GAS ou AS. Sa documentation est disponible à l'adresse `http://sourceware.org/binutils/docs/as/`. Il utilise la syntaxe AT&T, alors que la plupart des autres mentionnés utilisent la syntaxe Intel. Le GAS peut lire de l'assembleur en syntaxe Intel, pourvu que le code contienne la directive `.intel_syntax`.

2 Les registres et appels 64 bits

L'architecture 64 bits dispose des 16 registres suivants, `rax,rbx,rcx,rdx,rsi,rdi,rbp,rsp` et `r8` jusqu'à `r15`. Les fonctions en 64 bits opèrent sur des *quadwords*, et sont donc postfixées d'un `q`, donc `movq`, `addq`, &c.

3 Bonjour tout le monde

On commence par le fichier assembleur suivant,

```
    .data
moremsg:
    .string     "Hello, world\n"
moremlen:
    .int        13
lessmsg:
    .string     "Goodby, world\n"
lessmlen:
    .int        14

    .text
    .global main
main:
    cmpq    $42, %rax
    jge     less
    /* Prepare for syscall */
    movq    $1, %rdi /* File descriptor */
    movq    moremlen(%rip), %rdx /* Size of string */
    leaq    moremsg(%rip), %rsi
    call    write@plt /* System call */
    ret
less:
    movq    $1, %rdi /* File descriptor */
    movq    lessmlen(%rip), %rdx /* Size of string */
    leaq    lessmsg(%rip), %rsi
    callq   write@plt
    ret
```

qu'on peut compiler avec

```
gcc -g hello.s -o hello.out
```

On pourra utiliser le Makefile suivant pour n'avoir qu'à faire `make hello.out`,

```
.POSIX:
CC = gcc

.SUFFIXES: .s .out

.s.out:
${CC} -g -o $@ $<
```

Les instructions de la forme `write@plt` et `data(%rip)` permettent d'avoir des PIE pour *Position Independant Executables* (voir `https://en.wikipedia.org/wiki/Position-independent_code` pour plus d'information). Si la compilation ne fonctionne pas, c'est peut-être car le compilateur n'est pas configuré pour.

On peut examiner le déroulement du programme avec `ddd`. Pour voir les registres et leur contenu, faites un click droit sur une ligne du code source, et "set breakpoints"; cliquez ensuite sur "Run"; et une fois au *breakpoint*, cliquez sur "Status" puis sur "Register". Cliquez sur "Stepi" pour avancer instruction par instruction. Remarquez le changement de valeur du registre `rcx` ou `rdi`.

4 Conventions d'appels System V AMD64 ABI

Ces conventions sont suivies par Solaris, Linux, FreeBSD et macOS. Ces conventions doivent être respéctées pour s'interfacer avec le système d'exploitation.

- Les arguments des fonctions sont passés dans les registres `rdi, rsi, rdx, rcx, r8, r9, r10`. Les arguments supplémentaires sont passés sur la pile.
- Les arguments de type float sont passés dans les registres `xmm0` à `xmm7`.
- Si l'appelée utilise les registres `rbx`, `rbp`, ou `r12-15`, leur valeur originale doit être restaurée avant de quitter le corps de la fonction (les autres registres sont supposés être pris en charge par la fonction appelante).

5 Une somme variadique

Écrivons petit à petit une fonction qui somme un nombre variable d'arguments.

5.1 Une simple addition: pas de pile

1. Écrire un programme qui renvoit un entier arbitraire fixé dans le code (c'est-à-dire le place dans `rax`). Pour vérifier son bon fonctionnement, en supposant que l'exécutable se nomme `a.out`

   ```
   ./a.out
   echo $?
   ```

   devrait afficher l'entier choisi (`$?` contient le dernier code de retour).

2. Implémenter l'addition de deux entiers uniquement à l'aide de l'opérateur d'incrémentation et de décrémentation. Les arguments pourront dans un premier temps être placés dans des registres (donc déclarés dans le code), par exemple,

   ```
   movq    $3, %rbx
   movq    $2, %rcx
   ```

   pour par la suite additioner 3 et 2. Le résultat devra être le code de retour.

3. Utiliser des constantes déclarées dans la section `.data` plutôt que des valeurs codées en dur.

4. Introduire (si vous ne l'avez pas fait naturellement) une fonction (*i.e.* un label) `add`.

5.2 Une addition variadique: amenez la pile

1. Écrire une fonction `adds` prenant ses deux arguments sur la pile (vous pouvez utiliser `popq`, même s'il existe un autre moyen).

2. Écrire une fonction `addv` qui prend son premier argument dans `rdi` et le reste sur la pile et les somme. Si `rdi = n`, `addv` somme les `n` arguments de la pile.

3. Modifiez votre fonction pour utiliser le moins de registres possible, et utiliser la pile. Cela nécessite de s'allouer de la place sur la pile en entrant dans la fonction pour y stocker des variables locales. On va donc fair entrer en jeu la gestion des *stack frames*. Concrètement, on va utiliser le registre `rbp` pour indiquer le **fond** de la pile pour la

fonction (et donc l'adresse la plus élevée à laquelle la fonctions peut accéder). Pour l'utiliser, on va, pour tout appel de fonction nécessitant des variables locales,

(a) sauvegarder le pointeur de base de pile `rbp` pour pouvoir le restaurer en sortant,

(b) mettre le pointeur de base de pile à la même valeur que le pointeur de sommet de pile

Ainsi, `rbp` indique la base d'une pile à laquelle on peut accéder. En se débrouillant bien, on n'utilise que les registres `rsi`, `rax` et `rbp`.

En pratique, il sera aussi nécessaire de décaler `rsp` pour le sortir de l'espace qu'on se réserve sur la pile, pour pouvoir appeler d'autres fonctions.

6 S'il en faut plus

Je vous invite à regarder le TP donné il y a deux ans qui consiste à écrire un débogueur, disponible à `http://www.lsv.fr/~hondet/6-bonus.pdf`.

7 Références

- `https://cs.lmu.edu/~ray/notes/gasexamples/`
- `http://sourceware.org/binutils/docs/as/`
- `http://www.lsv.fr/~goubault/CoursProgrammation/prog1_lecon2.pdf`
- Syntaxe GAS, `https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax`
- Un guide sur l'assembleur 32bits `http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html`
- Un autre cours, `http://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf`