

[ASM] Kit de base

Introduction

Le **langage d'assembleur** est le langage le plus bas-niveau et le plus proche du langage machine.

Cependant, contrairement aux autres langage, chaque **architecture** possède son **jeu d'instruction** et donc son propre langage d'assembleur.

Ce cours traitera essentiellement de l'architecture **Intel** et peut-être **AT&T**.



Installation d'un assembleur (nasm)

Linux

Pour installer **nasm** sur Debian :

```
sudo apt install -y nasm build-essential
```

Registres

Registres de données

Un registre de données permet de stocker 16 bits soit 2 octets avec les bits forts placés à gauche et les bits faibles à droite.

| Nom raccourcis | Nom complet | Fonction |
|----------------|----------------|--|
| AX | Accumulator | Opérations arithmétiques et d'entrées/sorties. |
| BX | Base register | Adressage mémoire. |
| CX | Count register | Compteur de boucle. |
| DX | Data register | Extension d'AX. |

Registre d'états

Ce type de registre contient des flags permettant de suivre des états.

Sections

Data

C'est ici que nos constantes seront définies dans le code comme dans l'exemple ci-dessous :

```
SECTION .data
msg    db    'Hello world!', 0Ah
```

La valeur **0A** correspond au caractère `\n` et le **h** spécifie qu'il s'agit d'une valeur hexadécimale.

Text

C'est ici que nous définissons le point d'entrée (le label qu'il faut lancer pour démarrer le programme) :

```
SECTION .text
global _start
```

Convention d'appel

Voici la liste des registres et leur utilité lors de l'appel à vos fonctions :

| arch | syscall NR | return | arg0 | arg1 | arg2 | arg3 | arg4 | arg5 |
|--------|------------|--------|------|------|------|------|------|------|
| arm | r7 | r0 | r0 | r1 | r2 | r3 | r4 | r5 |
| arm64 | x8 | x0 | x0 | x1 | x2 | x3 | x4 | x5 |
| x86 | eax | eax | ebx | ecx | edx | esi | edi | ebp |
| x86_64 | rax | rax | rdi | rsi | rdx | r10 | r8 | r9 |

Libc

En architecture Intel x86, selon la convention d'appel, nous devons procédé comme suit pour afficher **"Hello world"** à l'écran :

```
_start:
    MOV eax, 4 ; Appel à SYS_WRITE (OPCODE 4 définit dans la libc)
    MOV ebx, 1 ; La valeur 1 correspond à la sortie standard STDOUT
    MOV ecx, msg ; Met le contenu de la constante msg dans le registre ecx
    MOV edx, 13 ; Indique la taille de la chaîne de caractère incluant le null byte (ici 13)
    INT 80h ; Lance une interruption de la libc pour exécuter nos instructions
```

Voici comment fermer un programme proprement avec la libc :

```
_start:
    ; Close program
    MOV eax, 1 ; Appel à SYS_CLOSE dans la libc (OPCODE 1)
    MOV ebx, 0 ; Code de retour 0
    INT 80h
```

Les jeux d'instructions

| Instructions | Descriptions |
|--------------|--|
| MOV | Déplace une valeur dans un registre sous le format suivant : MOV dst, src |
| CALL | Fait appel à une subroutine (fonction) |
| INT | Fait une interruption (pour appeler la libc) |
| PUSH | Pousse une valeur sur la stack |
| POP | Retire une valeur de la stack (format LIFO) |
| SUB | Effectue une soustraction entre deux adresses |
| ADD | Effectue une addition entre deux adresses |
| CMP | Compare deux valeurs |
| JE | "Jump if Equal". Se rend sur la routine ou le label indiqué si les deux valeurs comparées sont identiques. |
| JZ | "Jump if the Zero flag is set". Se rend sur la routine ou le label indiqué si le flag zero est défini. |
| JMP | Saute à la routine ou au label mentionné. |
| INC | Incrémente une valeur ou une adresse. |
| RET | Définit la fin d'une fonction. |

Arguments

Les arguments sont passés sur la stack de sorte à ce que les derniers éléments posés sur la stack soit le **nombre d'argument**, le **nom de l'exécutable**, le **premier argument**, le **deuxième** etc.