Langage assembleur

Du binaire au langage de bas niveau

La mémoire dans le modèle de Von Neumann

Dans l'architecture de Von Neumann, une seule mémoire, la **mémoire vive** (**RAM**), reçoit à la fois **les données** et **le programme**, c'est-à-dire la suite des instructions à exécuter.

Or la mémoire ne contient que des bits. L'octet `01000001`, lu comme un entier, vaut 65 ; lu comme un caractère codé en ASCII, c'est la lettre `A` ; lu par le processeur comme une instruction, c'est un ordre, dont le sens dépend du processeur.

Rien, dans l'octet lui-même, n'indique la bonne lecture. C'est l'**usage** qui décide : le processeur traite comme une instruction le mot rangé à l'adresse indiquée par son **compteur de programme**, et comme une donnée le mot qu'une instruction lui demande de lire. Le programmeur et le système d'exploitation doivent donc ranger instructions et données à des endroits bien séparés.

Les instructions machine

Principe

Une **instruction machine** est un mot binaire formé de deux parties :

```
code de l'operation | operandes
```

Le **code de l'opération** (*opcode*) dit ce qu'il faut faire ; les **opérandes** disent sur quoi agir : un registre, une valeur, une adresse en mémoire. L'ensemble des opérations qu'un processeur sait exécuter forme son **jeu d'instructions** ; chaque famille de processeurs a le sien.

Pour un petit processeur imaginaire, l'instruction `00110010 00001010` pourrait se lire ainsi : le premier octet est le code de l'opération « ajouter une valeur au registre R2 », le second octet, 10 en décimal, est la valeur à ajouter.

Une instruction occupe un ou plusieurs **mots** de la mémoire. Le mot est la quantité d'information que le processeur traite d'un seul coup : 8 bits pour un processeur 8 bits, 64 bits pour les processeurs des ordinateurs actuels. Ces suites de bits forment le **langage machine**, le seul que le processeur comprenne.

Opérations de base

On retrouve presque toujours les mêmes familles d'instructions :

— **transfert** : placer un nombre dans un registre, copier un registre, lire ou écrire un mot de la mémoire ;

— **calcul** : additionner, soustraire, appliquer une opération logique bit par bit ;

— **comparaison** : comparer deux valeurs, ce qui met à jour des indicateurs (résultat nul, négatif…) ;

— **saut** : poursuivre l'exécution ailleurs dans le programme, sans condition ou selon le résultat de la dernière comparaison.

Le langage assembleur

Une suite de 0 et de 1 est très difficile à écrire et à relire. On a donc donné à chaque code d'opération un nom court, un **mnémonique** : `ADD` pour additionner, `SUB` pour soustraire, `MOV` pour placer une valeur, `LDR` pour lire en mémoire… C'est le **langage assembleur**. L'instruction binaire du processeur imaginaire devient :

```
ADD R2, R2, #10
```

On lit : « ajouter 10 au contenu du registre R2, et ranger le résultat dans R2 ». Le symbole `#` signale une valeur écrite dans l'instruction, et non une adresse.

Ce court programme lit deux nombres au clavier, les additionne et affiche le résultat :

```
INP R0, 2
INP R1, 2
ADD R2, R0, R1
OUT R2, 4
HALT
```

Chaque instruction `INP` met le programme en pause : l'exécution ne reprend que lorsqu'un nombre a été tapé dans la zone *Input* et validé avec la touche Entrée.

Un programme appelé lui aussi **assembleur** traduit ce texte en langage machine, presque ligne pour ligne. En résumé :

— le **langage machine** est la suite de bits que le processeur exécute directement ;

— le **langage assembleur** écrit les mêmes instructions avec des noms lisibles ;

— l'**assembleur** (le programme) traduit le langage assembleur en langage machine.

Comme le langage machine, un langage assembleur est propre à une **famille de processeurs**. Le simulateur AQA a le sien ; la fiche d'exercices utilise celui d'un vrai processeur, le **MIPS** : mêmes idées, mais d'autres noms (`li $t0, 10` au lieu de `MOV R0, #10`). C'est un **langage de bas niveau** : chaque ligne correspond à une opération élémentaire du processeur.

Les compilateurs et interpréteurs

Écrire un programme entier en assembleur reste long et fastidieux. On a donc créé des **langages de haut niveau** (C, Python, JavaScript, etc.), qui manipulent des variables, des listes ou des fonctions plutôt que des registres et des adresses. Plus un langage éloigne le programmeur des détails de la machine, plus son **niveau d'abstraction** est élevé.

Le processeur ne comprend que son langage machine : tout programme doit être traduit, de l'une des trois façons suivantes.

— La **compilation** : un compilateur traduit une fois pour toutes le programme complet et produit un fichier exécutable (exemple : le langage C).

— L'**interprétation** : un interpréteur traduit et exécute les instructions une à une, à chaque lancement (exemple : les premières versions de Basic).

— La **méthode mixte** : le programme est traduit en un code intermédiaire, le *bytecode*, exécuté par une **machine virtuelle** ; le même bytecode tourne sur tout ordinateur qui possède cette machine virtuelle (exemples : Python, Java).

L'ASSEMBLEUR EN PRATIQUE

Le simulateur AQA

Le simulateur AQA, écrit par Peter Higginson et utilisable dans un navigateur, représente une machine de Von Neumann simplifiée, que l'on programme en assembleur. Deux règles changent tout par rapport à Python :

— il n'y a pas de variables : une valeur est rangée soit dans l'un des treize registres `R0` à `R12`, soit dans un mot de la mémoire, désigné par son adresse ;

— le processeur ne calcule qu'avec des registres : une valeur rangée en mémoire doit d'abord être chargée dans un registre.

Syntaxe des opérandes

Syntaxe	Signification
`Rn`	le registre numéro n
`#n`	la valeur n elle-même
`n`	le mot de la mémoire d'adresse n

Quelques instructions

Instruction	Effet
`MOV R1, #25`	met la valeur 25 dans R1
`LDR R1, 20`	lit le mot d'adresse 20 et le copie dans R1
`STR R1, 22`	écrit le contenu de R1 dans le mot d'adresse 22
`ADD R3, R1, R2`	calcule R1 + R2 et range le résultat dans R3
`SUB R3, R1, #1`	calcule R1 − 1 et range le résultat dans R3
`CMP R1, #0`	compare R1 à 0 et met à jour les indicateurs
`BGT etiquette`	saute à l'étiquette si la comparaison a donné « plus grand »
`OUT R1, 4`	affiche le contenu de R1
`HALT`	arrête le programme

Premier programme

```
MOV R0, #25
MOV R1, #17
ADD R2, R0, R1
OUT R2, 4
HALT
```

Ce programme place 25 dans R0 et 17 dans R1, range leur somme, 42, dans R2, l'affiche, puis s'arrête.

Une boucle

```
        MOV R0, #5
boucle: OUT R0, 4
        SUB R0, R0, #1
        CMP R0, #0
        BGT boucle
        HALT
```

Ce programme affiche 5, 4, 3, 2 et 1 : tant que R0 reste strictement positif, le saut `BGT` renvoie l'exécution à l'étiquette `boucle`. Il joue le rôle d'une boucle `while` de Python.

Que retenir ?

— Un programme rangé en mémoire est une suite d'**instructions machine**, que le processeur exécute directement.

— Une instruction machine se compose d'un **code d'opération** et d'**opérandes** : des valeurs, des registres ou des adresses en mémoire.

— Le **langage assembleur** écrit ces instructions avec des noms lisibles ; le programme **assembleur** les traduit en langage machine.

— Chaque famille de processeurs a son propre **jeu d'instructions**, limité à des opérations simples : transfert, calcul, comparaison, saut.

— Un **compilateur** traduit un programme écrit en langage de haut niveau, comme C, en langage machine ; Python est traduit en bytecode, exécuté par une machine virtuelle.