

## Le langage assembleur

### Langage assembleur

Le langage machine est une suite de bits. Par exemple pour faire une addition, on peut avoir à écrire quelque chose comme 11100010100000100001000001111101. Il est donc très difficile de programmer directement avec ce langage.

C'est pourquoi les informaticiens ont développé un langage un peu plus compréhensible, appelé **assembleur**. L'instruction précédente s'écrit alors "**ADD** R1, R2, #125" et veut dire "ajouter 125 au contenu du registre R2 et mettre le résultat dans R1".

Chaque type de processeur possède son langage assembleur, mais les instructions de bases sont très similaires. Le but ici n'est pas d'apprendre un langage en particulier, mais de comprendre ces instructions de base.

### Opérations arithmétiques et logiques

Nous allons prendre comme exemple, un sous ensemble du langage des processeurs ARM. Pour la suite, et sauf mention contraire, **dest** et **op1** désignent des registres et **op2** un registre ou une valeur immédiate.

Les registres utilisés seront R0 à R12. Les valeurs immédiates peuvent être :

- un entier en base 10 : #92
- un entier en binaire précédé par **0b** : #0b00101
- un entier en hexadécimal précédé par **0x** : #0xF3

Voici les opérations arithmétiques et logiques :

Syntaxe	Explications	Commentaires
<b>MOV</b> dest, op1	dest = op1	op1 peut être une valeur immédiate
<b>ADD</b> dest, op1, op2	dest = op1 + op2	
<b>SUB</b> dest, op1, op2	dest = op1 - op2	
<b>AND</b> dest, op1, op2	dest = op1 et op2	Le et est fait bit à bit
<b>OR</b> dest, op1, op2	dest = op1 ou op2	Le ou est fait bit à bit
<b>EOR</b> dest, op1, op2	dest = op1 xor op2	Le xor est fait bit à bit

**EXERCICE 1 :** On considère le programme ci-dessous :

```
MOV    R0, #10
MOV    R1, #3
ADD    R2, R0, R1
ADD    R2, R0, R2
```

- 1) Quelle est la valeur de R2 à la fin de l'exécution de ce programme?
- 2) Écrire un programme Python faisant la même chose.

**EXERCICE 2 :** Écrire un programme en assembleur correspondant au programme Python ci-dessous, en précisant à quelle variable correspond chaque registre.

```
a = 9
b = a + 5
c = b - 3
a = b + c
```

## Tests et sauts

Afin de stocker les résultats des tests, il y a quatre bits spéciaux qui sont mis à jour lors des opérations suivantes :

- N vaut 1 si le résultat est négatif et 0 sinon ;
- Z vaut 1 si le résultat est nul et 0 sinon ;
- C vaut 1 s'il y a une retenue et 0 sinon ;
- V vaut 1 s'il y a un dépassement (overflow) et 0 sinon.

Ces valeurs sont modifiées par les instructions suivantes :

Syntaxe	Explications	Commentaires
<b>CMP</b> op1, op2	op1 = op2?	Met à jour NZCV en faisant op1 – op2
<b>CMN</b> op1, op2	op1 = –op2?	Met à jour NZCV en faisant op1 + op2
<b>TST</b> op1, op2		Met à jour NZ en faisant op1 et op2
<b>TEQ</b> op1, op2		Met à jour NZ en faisant op1 xor op2

**TST** permet de tester la parité d'un nombre en faisant **TST** op1, #1.

Il est également possible de modifier les valeurs de NZCV en rajoutant un S à la suite des opérateurs arithmétiques ou logiques. Ainsi, "**SUBS** R1, R2, #7" permet d'éviter de faire "**SUB** R1, R2, #7" puis "**CMP** R1, #0".

Ces informations sont utilisées pour effectuer des sauts dans le programme. Pour cela, il est possible d'insérer des **labels** dans le code du programme. Lors d'une instruction de saut, si les conditions sont vérifiées, le programme saute à l'instruction indiquée par le label donné, sinon, il continue à l'instruction suivante. Les instructions de saut sont :

Syntaxe	Explications	Commentaires
<b>B label</b>	Saut inconditionnel	Le saut est toujours exécuté
<b>BEQ label</b>	Saut si Z = 1	Cas d'égalité
<b>BNE label</b>	Saut si Z = 0	Cas d'inégalité
<b>BPL label</b>	Saut si N = 0	Valeur positive ou nulle
<b>BGE label</b>	Saut si N = V	Cas supérieur ou égal
<b>BGT label</b>	Saut si N ≠ V et Z = 0	Cas strictement supérieur
<b>BLE label</b>	Saut si N = V et Z = 1	Cas inférieur ou égal
<b>BLT label</b>	Saut si N ≠ V	Cas strictement inférieur

Pour la plupart des opérations arithmétiques et logiques, il est possible d'exécuter l'opération uniquement si un des cas ci-dessus est vérifié, en rajoutant le suffixe correspondant. Par exemple **MOVEQ** ne sera exécuté que si Z = 1.

**EXERCICE 3 :** On considère le programme ci-dessous :

	<b>MOV</b>	R0, #10
	<b>MOV</b>	R1, #3
	<b>CMP</b>	R0, R1
	<b>BGE</b>	<b>label2</b>
<b>label1</b>	<b>SUB</b>	R2, R1, R0
	<b>END</b>	
<b>label2</b>	<b>SUB</b>	R2, R0, R1
	<b>END</b>	

- 1) Quelle est la valeur de R2 à la fin de ce programme?
- 2) Donner des valeurs de R0 et de R1 qui font sauter au label label2?
- 3) Donner des valeurs de R0 et de R1 qui permet d'exécuter le code suivant label1?
- 4) Que fait ce programme?
- 5) Écrire un programme Python faisant la même chose.

**EXERCICE 4 :** Écrire un programme en assembleur correspondant au programme Python ci-dessous, en précisant à quelle variable correspond chaque registre.

```
x = 4
y = 8
if x == 10:
    y = 9
else :
    x = x + 1
z = 6
```

---

### *Un exemple de programme*

---

Le programme suivant permet de réaliser la multiplication des valeurs se trouvant dans les registres R0 et R1. Le résultat se trouve dans R2 à la fin de l'exécution. R1 doit être positif.

```
init    MOV    R0, #4      ; R0 = 4
        MOV    R1, #5      ; R1 = 5
        MOV    R2, #0      ; R2 = 0, le résultat
boucle  CMP    R1, #0      ; Si R1 = 0, on a fini
        BEQ    fin        ; On passe à la fin si R1 = 0
        ADD    R2, R0, R2  ; Sinon, R2 = R2 + R0
        SUB    R1, R1, #1  ; R1 = R1 - 1
        B      boucle     ; On recommence
fin     END                ; Fin du programme
```

L'idée est de faire  $R2 = R0 + R0 + \dots + R0$  autant de fois que nécessaire. Pour cela, on enlève 1 à R1 à chaque fois qu'on ajoute R0 à R2. Une fois que R1 est à 0, on peut finir le programme avec la commande **END**.

**EXERCICE 5 :** Écrire un programme Python réalisant la même chose que le programme ci-dessus. Vous pouvez faire une fonction prenant R0 et R1 en paramètres.

**EXERCICE 6 :** Écrire un programme en assembleur correspondant au programme Python ci-dessous, en précisant à quelle variable correspond chaque registre.

```
x = 0
while x < 3:
    x = x + 1
```

**EXERCICE 7 :** Déterminer ce que réalise le programme suivant. Les valeurs en entrées sont dans R0 et R1. Les valeurs en sortie sont dans R0 et R2.

```
init    MOV    R0, #16
        MOV    R1, #3
        MOV    R2, #0
boucle  CMP    R0, R1
        BLT    fin
        SUB    R0, R0, R1
        ADD    R2, R2, #1
        B      boucle
fin     END
```

---

## Gestion de la mémoire vive

---

En pratique, le nombre de variables utilisées par un programme dépasse souvent le nombre de registres. Il est donc nécessaire de stocker les valeurs en mémoire vive et de les copier dans les registres pour faire les calculs. Il faut donc associer une adresse à chaque variable. C'est le travail du compilateur ou de l'interpréteur qui transforme le programme C, Python, ou écrit dans un autre langage de haut niveau, en code machine.

Les commandes permettant de faire cela pour un processeur ARM sont les suivantes :

Syntaxe	Explications
<b>LDR</b> dest, [source]	Copie la valeur contenue à l'adresse source dans le registre dest
<b>STR</b> source, [dest]	Copie la valeur contenue dans le registre source à l'adresse dest

Dans les deux cas, source et dest doivent être des registres.

Par exemple, le programme suivant :

```
x = 15
y = 37
x = x + y
```

peut se traduire par :

<b>MOV</b>	R0, #15	; x = 15
<b>MOV</b>	R12, #256	; adresse de x
<b>STR</b>	R0, [R12]	; on sauvegarde la valeur de x
<b>MOV</b>	R0, #37	; y = 37
<b>MOV</b>	R12, #260	; adresse de y
<b>STR</b>	R0, [R12]	; on sauvegarde la valeur de y
<b>MOV</b>	R12, #256	; adresse de x
<b>LDR</b>	R1, [R12]	; on lit la valeur de x
<b>ADD</b>	R0, R0, R1	; on fait x+y
<b>STR</b>	R0, [R12]	; on sauvegarde la valeur de x

Dans le cas d'un programme compilé, comme en C++, le compilateur pourra essayer de limiter le nombre de lectures et d'écritures afin d'optimiser l'exécution du programme.

---

## Pour aller plus loin

---

Puisqu'un des registres sert à stocker la position de la prochaine instruction à exécuter, il est possible d'enregistrer cette position dans un autre registre, de sauter à une autre partie du programme, comme un fonction, puis de revenir à la position précédente pour continuer l'exécution du programme. Ainsi, l'instruction "**MOV** R12, PC" copie l'adresse de la prochaine instruction dans le registre R12. Certains langages incluent une commande spéciale qui fait cela pendant un saut. Par exemple, pour les processeurs ARM, les instructions **BL**, **BLEQ**, **BLNE**... copient la valeur de PC dans un registre spécial LR et ainsi, "**MOV** PC, LR" permettra de revenir à la position précédente.

<b>MOV</b>	R0, #5	; R0 = 5
<b>BL</b>	<b>double</b>	; on double R0
<b>ADD</b>	R0, R0, #1	; on rajoute 1
<b>BL</b>	<b>double</b>	; on double R0
<b>END</b>		; R0 = 22
<b>double</b>	<b>ADD</b>	R0, R0, R0
	<b>MOV</b>	PC, LR