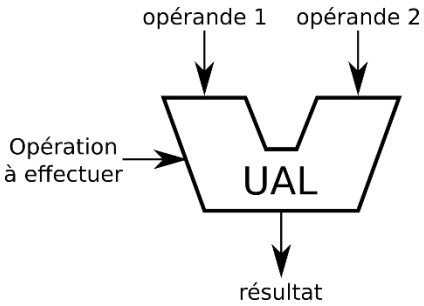
Assembleur : un peu de théorie

Rappelons que les unités arithmétiques et logiques (UAL) sont chargées d'effectuer les opérations arithmétiques (addition, soustraction, changement de signe etc.), les opérations logiques (and, or, xor, not etc.) et les tests (test d'égalité, de supériorité etc.).

1. Les UAL ne manipulent que deux opérandes

Une des propriétés des UAL – qui va avoir une conséquence importante sur le langage assembleur – est qu'elles ne travaillent en pratique qu'avec deux *opérandes* (deux *valeurs* si vous préférez).
Par exemple on ne peut pas demander à une UAL de faire le calcul :

- 2+7-5+99



En effet ce calcul comporte quatre *opérandes* (ou quatre *valeurs*). Il va falloir décomposer le travail en demandant à l'UAL trois calculs successifs avec deux *opérandes* :

- 2+7 (=9)
- 9-5 (=4)
- 4+99 (=103)

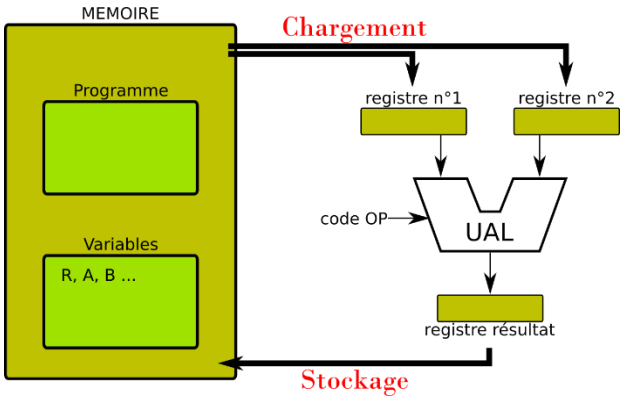
Cette caractéristique fondamentale, très présente lorsqu'on fait de l'assembleur, explique pourquoi les UAL sont souvent représentées sous la forme ci-contre.

2. Les opérandes ne sont souvent pas des noms de variables, le résultat non plus

a) Avec des registres : le cas des processeurs RISC

Pour des raisons d'optimisation diverses et en simplifiant le propos (il existe des livres entiers sur la question), les opérandes – juste avant le calcul - doivent souvent être stockés dans une mémoire très limitée en taille. Typiquement il y a deux *registres* pour les opérandes et un *registre* pour le résultat, sachant qu'un *registre* est une mémoire ne pouvant contenir … qu'une seule valeur !

Une conséquence de l'existence de ces *registres* (ou *accumulateurs*) est que les valeurs des variables doivent être chargées explicitement dans les registres avant de faire un calcul. Et que la valeur du résultat devra être mémorisée explicitement dans une variable (si besoin). Ainsi un langage assembleur ne comprendra pas une intruction telle que :



- R = A+B

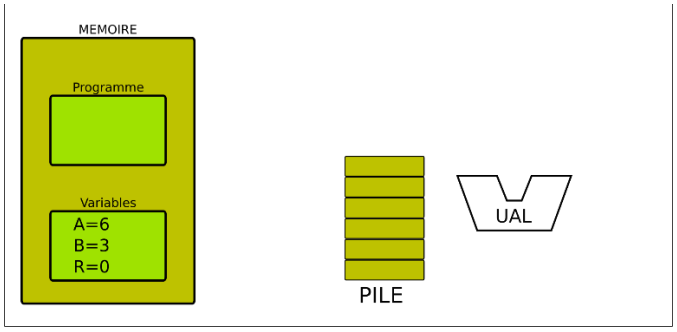
En assembleur sur un processeur de type RISC il faut faire quelque chose comme :

- Charger A dans registre n°1
- Charger B dans registre n°2
- Effectuer une somme
- Mémoriser le registre résultat dans la variable R

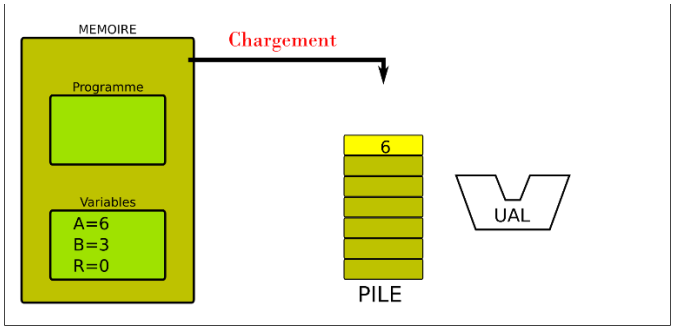
b) Avec une pile : le cas de l'assembleur IJVM

Il existe de nombreuses variantes au cas des registres ci-dessus, en particulier une variante où il y a une pile de valeurs (comme une pile d'assiettes) qui remplace les registres. Voici dans le cas d'une pile, un exemple de quatre instructions à exécuter en assembleur pour effectuer l'équivalent de R = A + B :

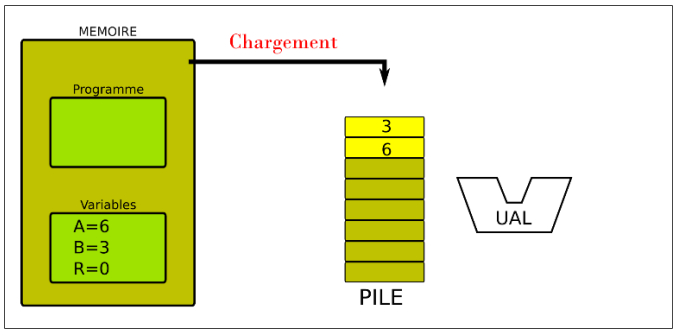
Instructions pour effectuer l'affectation R = A + B en assembleur IJVM



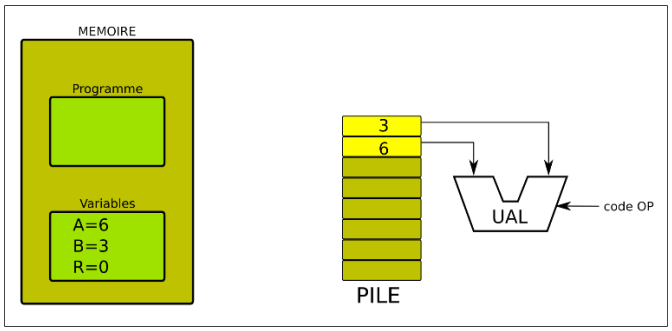
État initial



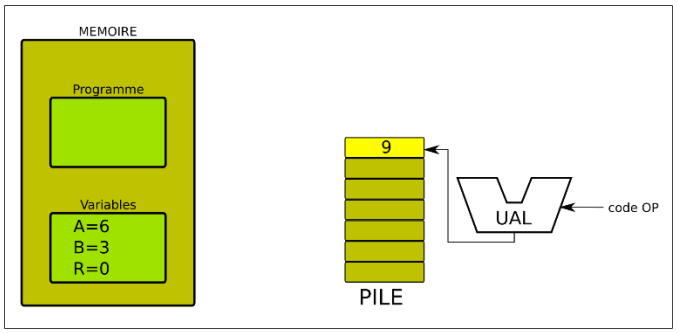
1 : Instruction de chargement de la valeur de A



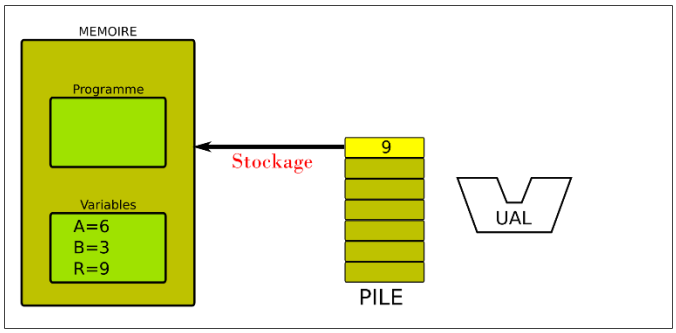
2 : Instruction de chargement de la valeur de B



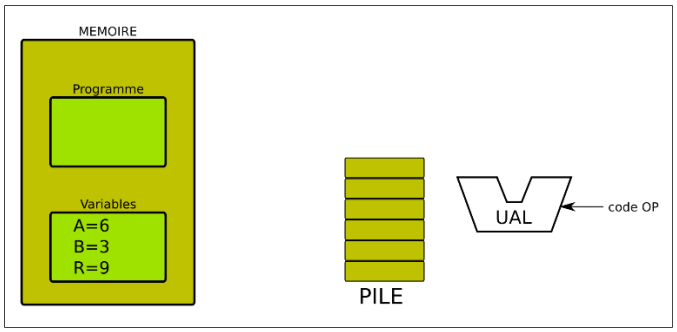
3 : Instruction d'addition
(implicitement sur les deux valeurs au sommet)



(implicitement le résultat "efface" les opérandes)



4 : Instruction de stockage
(implicitement du sommet de la pile
qui est effacé suite au stockage)



État final

TP assembleur

Voici les intructions IJVM que nous utiliserons :

BIPUSH X : place la valeur X au sommet de pile

ILOAD VAR : place la valeur de la variable VAR au sommet de la pile

IADD : additionne les deux valeurs au sommet de la pile et remplace ces deux valeurs par le résultat

ISUB : effectue la soustraction des deux valeurs au sommet de la pile et remplace ces deux valeurs par le résultat. Attention à l'ordre! C'est le haut de la pile qui est retranché.

ISTORE VAR : affecte la valeur du sommet de la pile à la variable VAR et efface le sommet de la pile.

POP : efface le sommet de la pile

DUP : duplique la valeur du sommet de la pile en la rajoutant une fois au sommet de pile.

GOTO etiq : Saut dans le programme à une position indiquée via l'étiquette mentionnée

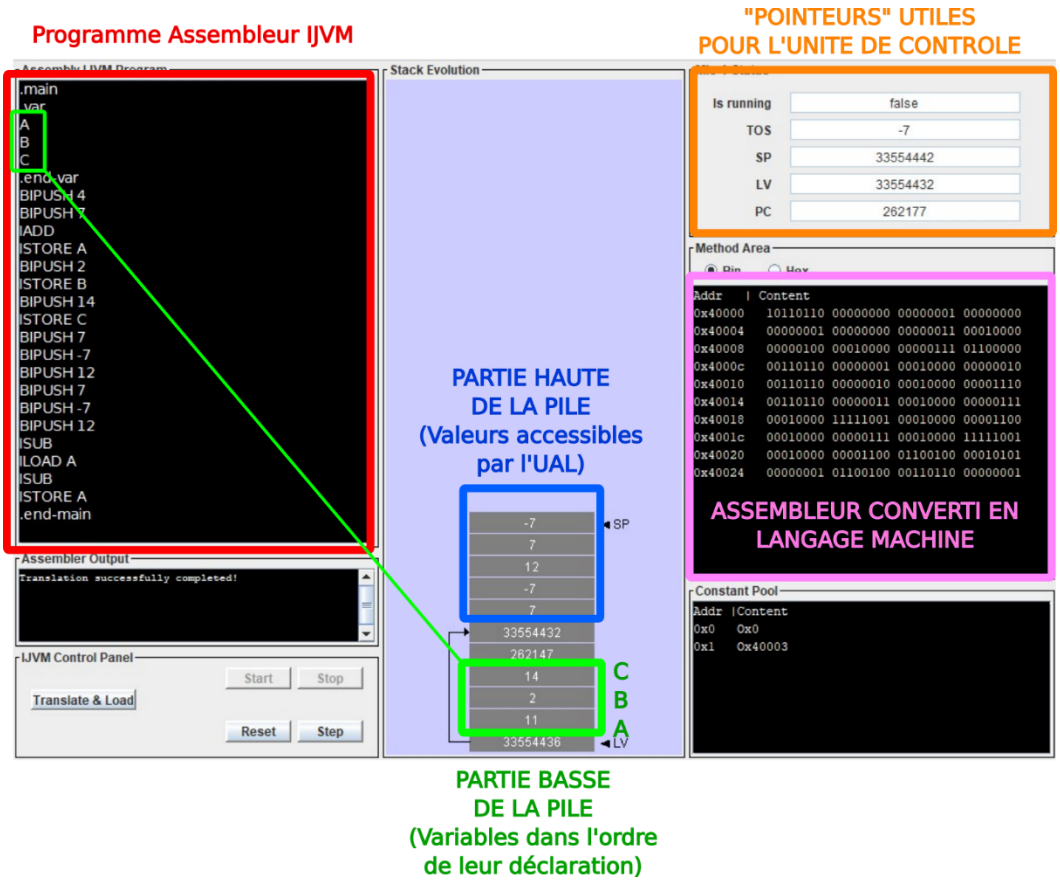
IFEQ etiq : Supprime le sommet de la pile puis effectue un saut à l'étiquette mentionnée si le sommet de la pile supprimé était égal à zéro.

IFLT etiq : idem IFEQ avec un saut si le sommet de la pile supprimé était strictement inférieur à zéro.

NOP : ne rien faire

Voici un petit rappel de ce qui vous a été présenté au vidéoprojecteur concernant le simulateur IJVM.

Rappelez vous que la pile est comme une pile d'assiettes : on ajoute des valeurs systématiquement par le haut et on retire des valeurs systématiquement par le haut.



Exercice 1 :

Avec l'émulateur IJVM, ouvrir le fichier exercice1_IJVM. Le programme saisi est représenté ci-contre. L'exécuter en mode pas à pas et répondre aux questions suivantes en observant les états successifs de la pile et les valeurs successives des variables :

```
.main
.var
A
B
C
.end-var
BIPUSH 7
ISTORE A
BIPUSH -1
BIPUSH 44
BIPUSH 33
BIPUSH 11
BIPUSH 11
IADD
IADD
IADD
ISTORE B
ISTORE C
.end-main
```

- Si l'on avait déclaré quatre variables A, B, C et D entre les lignes `.var` et `.end-var`, qu'est ce que ça aurait changé dans la partie basse de la pile après le premier clic sur "Step" ?
- À quelle valeur sont initialisées les variables après le premier clic sur "Step" ?
- Quel est le plus grand nombre de valeurs stockées dans la partie haute de la pile ?
- Quel est l'état de la partie haute de la pile après la première instruction IADD ? (Écrire les valeurs les unes au-dessus des autres)
- En fin d'exécution, quelle est valeur de B ? À quelle somme de quatre nombres correspond cette valeur ?
- Que se passe-t-il quand on clique sur le bouton radio Hex ? Le langage machine est-il plus lisible en représentation hexadécimale ou en représentation binaire ?

Exercice 2 :

```
.main
.var
X
Y
Z
.end-var
BIPUSH 5
BIPUSH 10
BIPUSH 99
ISTORE X
ISTORE Y
ISTORE Z
BIPUSH 7
ILOAD X
IADD
ILOAD Y
IADD
ILOAD Z
ISUB
ISTORE X
.end-main
```

Exécuter la séquence d'instructions ci-contre à la main.

Donner les valeurs des variables X, Y, et Z à la fin de la séquence d'instructions ainsi que l'état final de la partie haute de la pile.

Verifier votre résultat grâce au fichier exercice2_IJVM.

Exercice 3 :

Écrire une séquence d'instructions qui permet d'effectuer le calcul suivant : 55 + 45 – 4 – 9 et de stocker le résultat dans une variable K.

On pourra utiliser seulement la variable K ou – au choix – la variable K et d'autres variables.

On fera attention à l'ordre de la soustraction. C'est le sommet de la pile qui est retranché à la valeur du dessous.

Exercice 4 :

```
.main
.var
P
.end-var
BIPUSH 2
DUP
IADD
DUP
IADD
DUP
IADD
DUP
IADD
DUP
IADD
DUP
IADD
ISTORE P
.end-main
```

À la main, dérouler la séquence d'instructions ci-contre. Que fait-elle ? Quelle est la valeur affectée à la variable P en fin de séquence ?

On pourra vérifier son résultat grâce au fichier exercice4_IJVM.

Exercice 5 :

```
.main
.var
P
.end-var
BIPUSH 2
deb_boucle: DUP
IADD
GOTO deb_boucle
ISTORE P
.end-main
```

Avec l'émulateur IJVM, ouvrir le fichier exercice5_IJVM et exécuter la séquence d'instruction ci-contre en mode pas à pas (en cliquant sur "Step").

`deb_boucle:` est ce qu'on appelle une étiquette.

Que provoque l'instruction GOTO deb_boucle dans le déroulement de la séquence d'instructions ?

Dans IJVM, quel pointeur de l'unité de contrôle est particulièrement affecté par le GOTO ?

On répondra en dessinant une flèche au crayon rouge dans l'encadré ci-dessous :

```
1  .main
2  .var
3  P
4  .end-var
5  BIPUSH 2
6  deb_boucle: DUP
7  IADD
8  GOTO deb_boucle
9  ISTORE P
10 .end-main
```

Quel est le lien avec l'exercice 4 ?

Quel est l'énorme problème qui survient si on exécute ce programme ?

On dit que ***l'instruction assembleur GOTO est un saut simple***. En particulier cela permet de faire des sauts "en arrière" (qui sont utilisés par les boucles for ou while en langage de haut niveau).

Pour obtenir des boucles qui se terminent, les assembleurs disposent très souvent d'une ou plusieurs instructions de saut conditionnel. En IJVM les sauts conditionnels sont réalisés grâce aux instructions IFLT et IFEQ

Les instructions de sauts (simples ou conditionnels) affectent directement ***le pointeur PC (program counter)*** de l'unité de contrôle qui indique l'instruction à exécuter.

Nous allons voir qu'en assembleur, l'utilisation conjointe de saut simple et de saut conditionnel permet d'obtenir l'équivalent de la boucle for, de la boucle while ainsi que de l'instruction conditionnelle if … else...

Exercice 6 : équivalent assembleur du if … else

Fonctionnement de l'instruction `IFLT: etiq` en langage IJVM :

- Si le sommet de la pile est strictement négatif, un saut est effectué jusqu'à l'étiquette `etiq`
- Sinon le programme continue normalement

En lui adjoignant un `GOTO` on obtient un `if … else …`:

```
1  .main
2  .var
3  A
4  B
5  .end-var
6  BIPUSH 50
7  IFLT etiq1
8  BIPUSH 99
9  ISTORE A
10 GOTO etiq2
11 etiq1: BIPUSH 98
12 ISTORE A
13 etiq2 : BIPUSH 44
14 ISTORE B
15 .end-main
```

```
1  .main
2  .var
3  A
4  B
5  .end-var
6  BIPUSH -50
7  IFLT etiq1
8  BIPUSH 99
9  ISTORE A
10 GOTO etiq2
11 etiq1: BIPUSH 98
12 ISTORE A
13 etiq2 : BIPUSH 44
14 ISTORE B
15 .end-main
```

Pour chacun des deux programmes assembleur ci-dessus :

- Indiquer quelle est la valeur stockée au sommet de la pile juste avant l'exécution de la ligne 7 et, surtout, quel est le signe de cette valeur.
- Vérifier que le registre PC de l'unité de contrôle est convenablement modifié.
- Dérouler alors la séquence d'instructions. Préciser les valeurs affectées aux variables A et B à la fin de l'exécution.
- En considérant que la valeur placée en sommet de pile à la ligne 6 est stockée dans une variable X, donner un équivalent du programme ci-dessus en langage python avec `if … else …`.

Exercice 7 : équivalent assembleur de la boucle while

```
.main
.var
U
.end-var
BIPUSH 7
ISTORE U
etiq1: NOP
BIPUSH 100
ILOAD U
ISUB
IFLT etiq2
ILOAD U
BIPUSH 6
IADD
ISTORE U
GOTO etiq1
etiq2: ILOAD U
.end-main
```

- En vous aidant si besoin du schéma ci-dessous dérouler la séquence d'instructions ci-contre. Vérifier si besoin votre résultat avec le fichier exercice7_IJVM.
- Vérifier que le registre PC de l'unité de contrôle est convenablement modifié.
- Préciser quelle est la valeur affectée à la variable U en fin d'exécution.
- Quel est l'intérêt de l'instruction NOP ?
- Proposer un équivalent en langage python du programme ci-contre.