

Exercice 1. On donne le code assembleur suivant, ainsi que sa traduction en byte-code :

```
.method add(x)
ILOAD x
BIPUSH 5 // ici
IADD
IRETURN
.end-method
.main
BIPUSH 42
BIPUSH 2 // la
INVOKEVIRTUAL add // et la aussi
.end-main
```

- Combien de paramètres admet la méthode add ? combien de variables locales ? Que fait cette méthode ?
- A quelle adresse commence la méthode main ?
- A quelle adresse commence la méthode add ?
- Vérifier maintenant avec le simulateur. Observer (et comprendre !) l'état de la pile à chaque étape.

Exercice 2. Dans le simulateur sur Célène (Emulateur_IJVM.zip) expérimentez les appels de procédures ; lancez leurs exécutions pas à pas. Saisissez la procédure suivante qui additionne 5 à une valeur passée en paramètre.

```
.method add(x)
ILOAD x
BIPUSH 5
IADD
IRETURN
.end-method
.main
BIPUSH 0
BIPUSH 2
INVOKEVIRTUAL add
.end-main
```

- Comment est passé le paramètre?
- Comment est récupéré le résultat?
- Comment est traduite cette fonction en code exécutable? Voici un exemple:

Addr	Content
0x40000	0xb6 0x00 0x01 0x00
0x40004	0x01 0x00 0x00 0x10
0x40008	0x00 0x10 0x02 0xb6
0x4000c	0x00 0x02 0x00 0x02
0x40010	0x00 0x00 0x15 0x01
0x40014	0x10 0x05 0x60 0xac

Constant Pool	
Addr	Content
0x0	0x0
0x1	0x40003
0x2	0x4000e

Dans cet exemple:

A l'adresse 0x40000 on trouve le code 0xb6 de début d'exécution d'une méthode, en l'occurrence *main*. Les noms des méthodes sont référencés comme les constantes. On peut donc trouver leurs adresses dans la zone mémoire dévolue aux constantes. Ici c'est la constante 0x00 0x01 qui est lancée. Celle-ci se trouve à l'adresse 0x40003. Elle commence par 0x00 0x01 0x00 0x00 0x10

Les 2 premiers octets encodent le nombres de paramètres ici 1 pour *main* (*il y en a toujours un par défaut*)

Les 2 octets suivants encodent le nombres de variables locales ici 0 pour *main*

Ensuite on trouve les code d'instruction IJVM: 0x10 pour le BIPUSH, 0xb6 pour le INVOKEVIRTUAL.

Ce dernier est suivi par 0x00 0x02 qui est le numéro de la seconde constante qui représente le *add(x)*. Son adresse est 0x4000e. En 0x4000e on trouve 0x00 0x02 0x00 0x00 0x15

Les 2 premiers octets encodent le nombres de paramètres ici 2 pour *add* (*il y en a toujours un par défaut*) Les 2 octets suivants encodent le nombres de variables locales ici 0 pour *add*

Exercice 3. On donne le pseudo-code JAVA suivant :

```
main() {
    a=15;
    b=3;
    b=add(a,b)
    a=6
}
add(a,b){
    return a+b;
}
```

- Donner la valeur des variables à la fin de l'exécution.
- Traduire en assembleur puis en byte-code.
- Vérifiez ENSUITE à l'aide du simulateur.

Exercice 4. Donner le code assembleur IJVM correspondant au code hexadécimal ci-dessous. Traduire en pseudo-code JAVA. Vérifiez ensuite à l'aide du simulateur.

Method Area

Addr	Content
0x40000	0xb6 0x00 0x01 0x00
0x40004	0x01 0x00 0x00 0x10
0x40008	0xff 0x10 0x01 0x10
0x4000c	0x03 0xb6 0x00 0x02
0x40010	0x00 0x03 0x00 0x00
0x40014	0x15 0x01 0x15 0x02
0x40018	0x64 0x9b 0x00 0x08
0x4001c	0x15 0x01 0xa7 0x00
0x40020	0x05 0x15 0x02 0xac

Constant Pool

Addr	Content
0x0	0x0
0x1	0x40003
0x2	0x40010

Exercice 5.

On donne le pseudo-code JAVA suivant :

```
main{
    int a, b ;
    a=15;
    b=3;
    b=double(max(a,b)+double(b));
}
max(a,b){
    if (a<b)
        return b;
    else
        return a;
}
double(a){
    return 2*a;
}
```

- Traduire le code en assembleur puis en byte-code.
- Vérifiez à l'aide du simulateur.
- Observer (et comprendre !) l'état de la pile à chaque étape.

Exercice 6. On donne le pseudo-code JAVA suivant :

```

main{
    variables locales : a, b;
    a=15;
    b=3;
    return triple(max(a,b));
}
max(a,b){
    if (a<b)
        return b;
    else
        return a;
}

```

```

triple(a){
    return 3*a;
}

```

- Traduire le code en assembleur puis en byte-code.
- Vérifiez à l'aide du simulateur.
- Observer (et comprendre !) l'état de la pile à chaque étape.

Exercice 7. Voici un code hexadécimal exécutable de l'architecture IJVM :

Method Area

Addr	Content
0x40000	0xb6 0x00 0x01 0x00
0x40004	0x01 0x00 0x00 0x10
0x40008	0x00 0x10 0x08 0xb6
0x4000c	0x00 0x02 0x00 0x02
0x40010	0x00 0x00 0x15 0x01
0x40014	0x10 0x01 0x9f 0x00
0x40018	0x22 0x15 0x01 0x10
0x4001c	0x02 0x9f 0x00 0x1b
0x40020	0x10 0x00 0x15 0x01
0x40024	0x10 0x01 0x64 0xb6

0x40028	0x00 0x02 0x10 0x00
0x4002c	0x15 0x01 0x10 0x02
0x40030	0x64 0xb6 0x00 0x02
0x40034	0x60 0xa7 0x00 0x05
0x40038	0x10 0x01 0xac 0x00

Constant Pool

Addr	Content
0x0	0x0
0x1	0x40003
0x2	0x4000e

- A quelle adresse débute la procédure (pas celle du main évidemment)? (donnez l'adresse de la déclaration du nb de paramètres de la procédure variable locales et aussi l'adresse du début du code exécutable de cette procédure)
- Donner le code IJVM correspondant.
- Donner le code de haut niveau (style JAVA ou C) correspondant.
- Que calcule ce code? (Ici c'est pour la valeur 6)
- Si on avait changé la valeur 6 qui est à l'adresse 0x4000A en la remplaçant par la valeur 8, qu'obtenait-on en sommet de pile à la fin de l'exécution ?

Dans cet exercice, on considère que les variables *a*, *b*, *c*, *d* sont placées en mémoire respectivement aux adresses 1, 2, 3, 4 calculées à partir de la position stockée dans le registre LV.

Exercice 8. Donner le code assembleur IJVM correspondant au code hexadécimal ci-dessous. Donnez ensuite le code JAVA correspondant.

Method Area

Addr	Content
0x40000	0xb6 0x00 0x01 0x00
0x40004	0x01 0x00 0x04 0x10
0x40008	0xfd 0x36 0x01 0x10
0x4000c	0x05 0x36 0x02 0x10
0x40010	0x08 0x15 0x01 0x60

0x40014	0x36 0x03 0x15 0x02
0x40018	0x15 0x03 0x64 0x99
0x4001c	0x00 0x0a 0x15 0x01
0x40020	0x36 0x04 0xa7 0x00
0x40024	0x07 0x10 0x00 0x36
0x40028	0x04 0x10 0x00 0x36
0x4002c	0x01 0x00 0x00 0x00

Constant Pool

Addr	Content
------	---------

0x0	0x0
0x1	0x40003

Exercice 9.

- Ecrire en pseudo-code JAVA une méthode mul qui prend 2 nombres entiers comme paramètres, et qui retourne le produit de ces deux nombres, et ceci SANS UTILISER LA MULTIPLICATION (vous proposerez plusieurs versions)
- Traduire en assembleur le pseudo-code JAVA suivant (avec les versions de la méthode mul de la question précédente)

```
main{
    variables locales : a, b,
    c;
    a=15;
    b=3;
    c= a + mul(a,b);
}
mul(a,b){
    return a*b;
}
```