

# Architectures des microprocesseurs

Emmanuel Melin  
Université d'Orléans

Bibliographie:  
« Architecture de l'Ordinateur »  
Andrew Tanenbaum  
Ed. PEARSON  
Cote bibliothèque:004.2 TAN

Grand merci au Professeur Thierry PAQUET pour les schémas  
(Univ. Rouen)

- Pour maîtriser la complexité d'un ordinateur...
  - La machine ne sait travailler qu'avec des représentations binaires et un ensemble d'instructions réduit : langage machine L0 pour la machine M0

Title:(Unknown)  
Creator:(Unknown)  
CreationDate:(Unknown)  
CreationDate:(Unknown)  
LanguageLevel:2

- Pour maîtriser la complexité d'un ordinateur...
  - La machine ne sait travailler qu'avec des représentations binaires et un ensemble d'instructions réduit : langage machine L0 pour la machine M0
  - Nécessité de construire un langage de plus haut niveau : L1
  - Machine virtuelle M1: celle qui pourrait exécuter le langage L1

Title:(Unknown)

CreationDate:(Unknown)

CreationDate:(Unknown)

LanguageLevel:2

?

- 2 solutions:

- Traduction:

- pour que l'exécution du programme écrit en L1 soit possible (bien que M1 n'existe pas) il faut le traduire en L0 qui sera exécuté sur la Machine M0

Title:(Unknown)

Creator:(Unknown)

CreationDate:(Unknown)

CreationDate:(Unknown)

LanguageLevel:2

- 2 solutions:

- Interprétation:

→ pour que l'exécution du programme écrit en L1 soit possible (bien que M1

Title:(Unknown)

Creator:(Unknown)

CreationDate:(Unknown)

CreationDate:(Unknown)

LanguageLevel:2

n'existe pas) il faut créer la machine M1 en L0 et ce programme sera exécuté sur la Machine M0

- Combinable:

## Exemple:

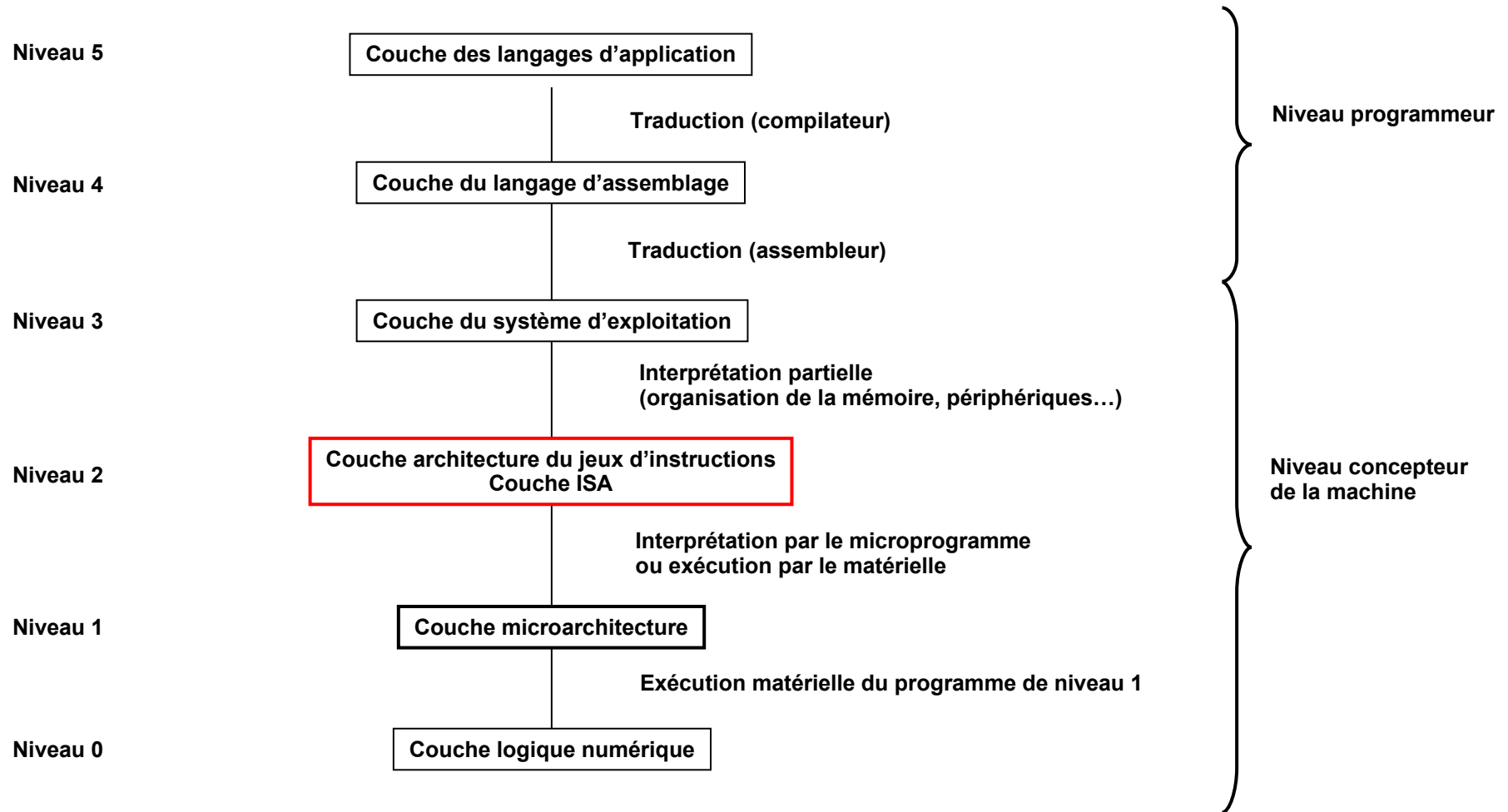
- Le Python ou le BASH sont directement interprétés par un programme.
- Le C ou le C++ sont compilés en langage exécutable par le processeur.
- Pour réaliser des programmes exécutables sur toutes les plateformes, le langage java est compilé dans un langage intermédiaire de type « P-Code » (code pré-compilé). Il est désigné par « bytecode Java ».
- Le bytecode Java est ensuite interprété sur une machine virtuelle: la JVM.
  - Il existe pour chaque plateforme des JVM. (par exemple dans les JRE de Oracle)
  - Plus rare : il existe également des processeurs qui implémentent le jeu d'instruction de la JVM

# Organisation multi-niveaux

- Le principe conceptuel du découpage en plusieurs couches de complexités croissantes est omniprésent en informatique.
  - Les systèmes informatiques (matériel + logiciel).
  - La conception des systèmes d'exploitation (noyau et couches Unix)
  - Les réseaux informatiques : les 7 couches de la norme Open System Interconnection de l'ISO
  - Les méthodes de conception de systèmes d'information (méthode Merise)
  - Les compilateurs (code source, intermédiaire, objet)
  - Les Types Abstraits de Données et les langages Orientés-Objet
  - Les systèmes transactionnels multi-niveaux



# Architectures actuelles en six couches



# La couche ISA

## Instruction Set Architecture

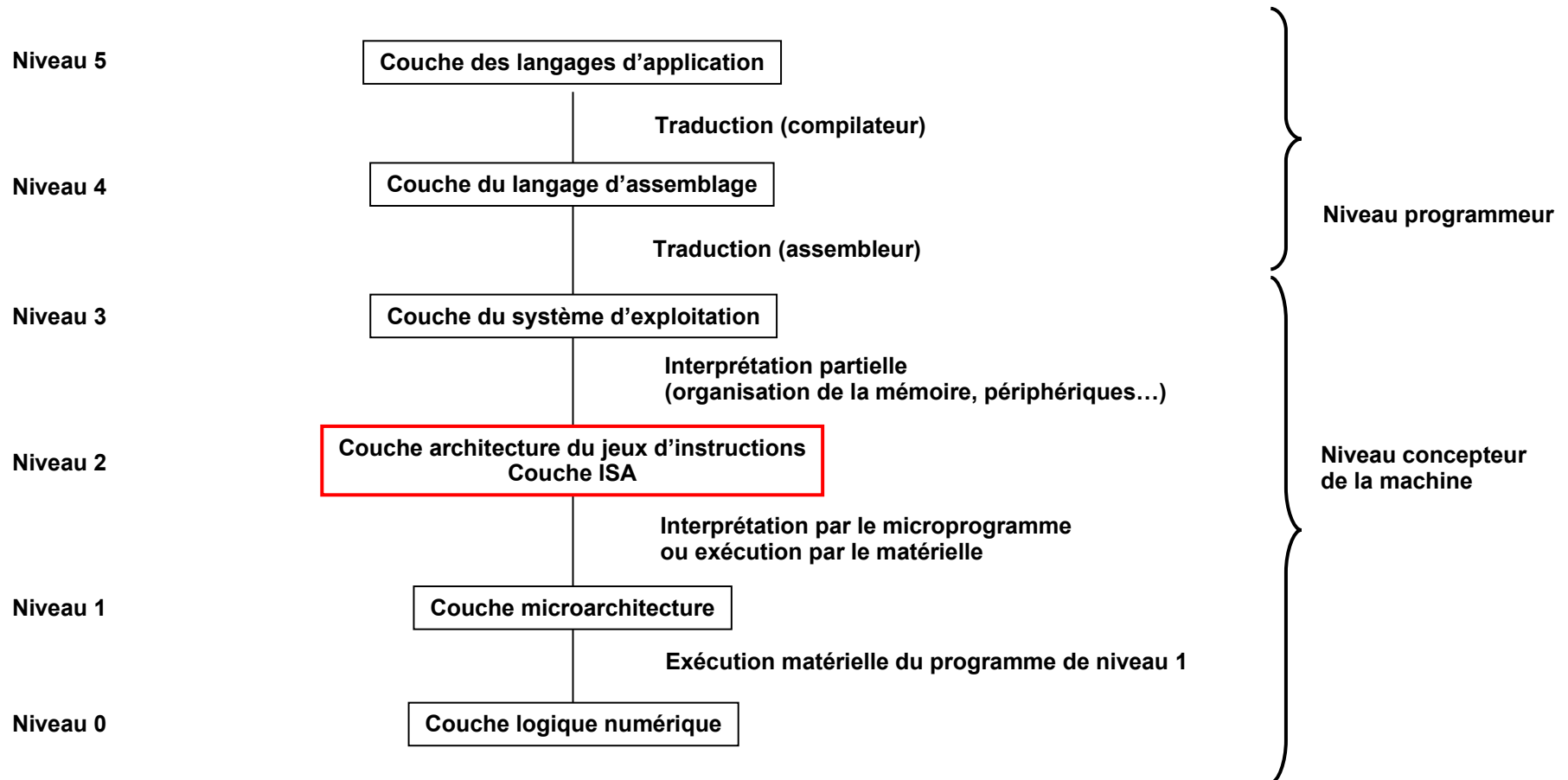
La couche ISA est la plus ancienne à avoir été développée

A l'origine c'était la seule couche fonctionnelle

C'est l'interface entre le logiciel et le matériel

C'est le langage intermédiaire commun aux différents langages de haut niveau

Les compilateurs produisent des programmes en langage ISA qui sont ensuite exécutés soit par le matériel soit par le micro-programme



# Propriétés de la couche ISA

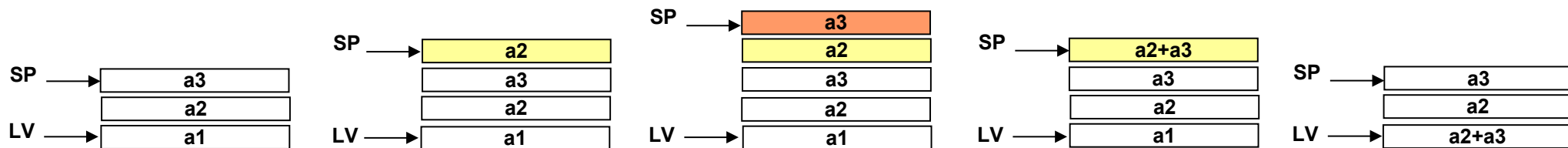
- La couche ISA est la vision qu'a le compilateur de la machine, ou le programmeur en langage d'assemblage
- Pour être efficace, le compilateur doit connaître l'organisation de la mémoire de la machine, les registres, le jeu d'instruction, les types de données manipulées
- L'architecture de la machine (pipeline, architecture superscalaire,...) n'est pas visible mais les développeurs de logiciels ou de compilateurs doivent connaître les points forts et faibles pour optimiser les programmes
- Certaines machines ont leur couche ISA décrite formellement dans une documentation (SPARC V9, JVM,...). Cela permet différentes implémentations de machines par différents constructeurs et selon différentes architectures. Ces machines sont toutes compatibles du point de vue des programmes qu'elles exécutent et des résultats qu'elles fournissent
- Principales instructions de la couche ISA
  - LOAD, STORE : déplacement de données entre la mémoire et les registres
  - MOVE : recopie de données entre registres
  - Instructions arithmétiques, booléennes, comparaison
  - Branchement ou sauts conditionnel ou inconditionnels

# La pile d'opérands

- On peut utiliser une pile pour stocker les opérandes d'une opération arithmétique
- Exemple :

$$a1 = a2 + a3$$

- 1- placer a2 au sommet de la pile (push)
- 2- placer a3 au sommet de la pile (push)
- 3- extraire les deux variables au sommet de la pile (pop)
- 4- effectuer l'addition et placer le résultat dans la pile
- 5- ranger le résultat du sommet de la pile dans la variable a1



- Pratiquement toutes les machines utilisent des piles pour stocker les variables locales
- Très peu utilisent une pile d'opérandes: la JVM en fait partie

# Commande IJVM ISA BIPUSH

- instruction : BIPUSH X
  - place la valeur X (en hexa ou en décimal) en sommet de pile

- exemple:

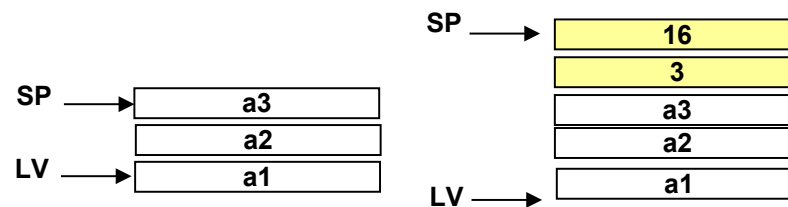
```
.main
```

```
bipush 3
```

```
bipush 0x10
```

```
halt
```

```
.end-main
```



# Commande IJVM ISA DUP

- instruction : DUP
  - place la valeur du sommet de la pile en sommet de pile

- exemple:

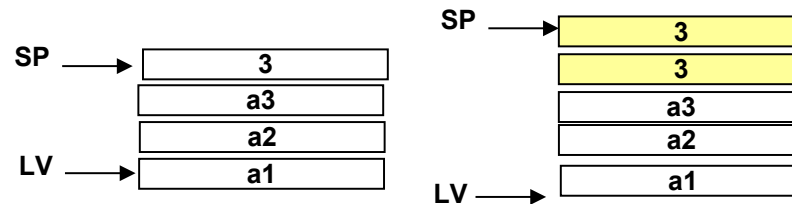
```
.main
```

```
bipush 3
```

```
dup
```

```
halt
```

```
.end-main
```

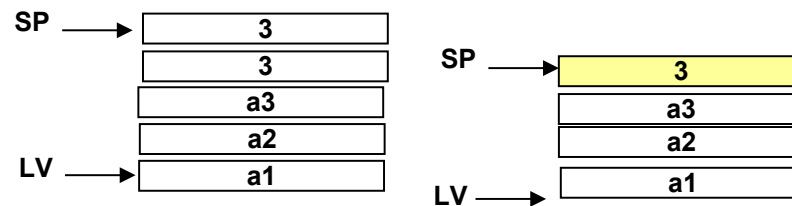


# Commande IJVM ISA POP

- instruction : POP
  - supprime le sommet de la pile

- exemple:

```
.main  
bipush 3  
dup  
pop  
halt  
  
.end-main
```

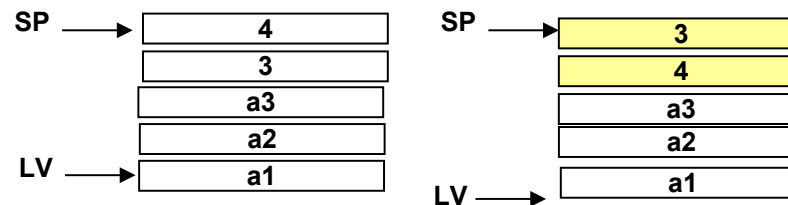


# Commande IJVM ISA SWAP

- instruction : SWAP
  - permute les 2 valeurs du sommet de la pile

- exemple:

```
.main  
bipush 3  
bipush 4  
swap  
halt  
.end-main
```

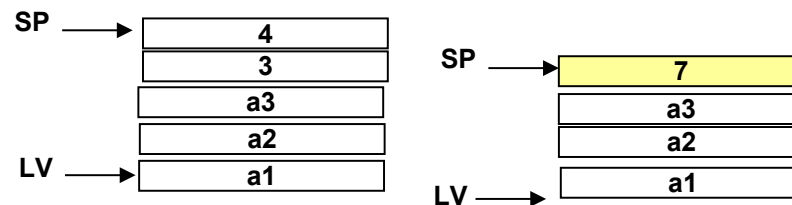




# Commande IJVM ISA IADD

- instruction : IADD
  - Dépile le sommet de la pile 2 fois et...
  - place la valeur de leur addition en sommet de pile
- exemple:

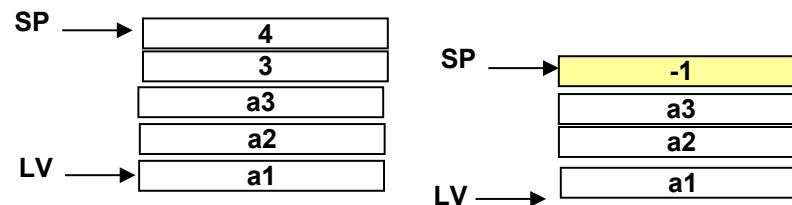
```
.main  
bipush 3  
bipush 4  
iadd  
halt  
.end-main
```



# Commande IJVM ISA ISUB

- instruction : ISUB
  - Dépile le sommet de la pile 2 fois et...
  - place la valeur de leur soustraction en sommet de pile
- exemple:

```
.main  
bipush 3  
bipush 4  
isub  
halt  
.end-main
```

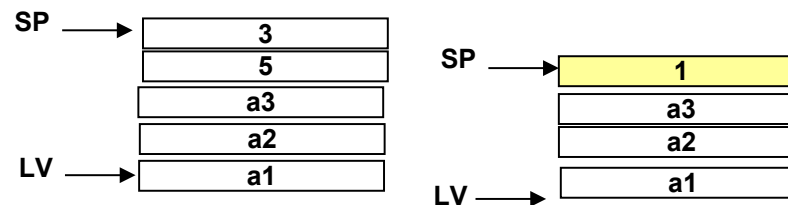


# Commande IJVM ISA IAND

- instruction : IAND
  - Dépile le sommet de la pile 2 fois et...
  - place la valeur de leur opération and bit à bit en sommet de pile

- exemple:

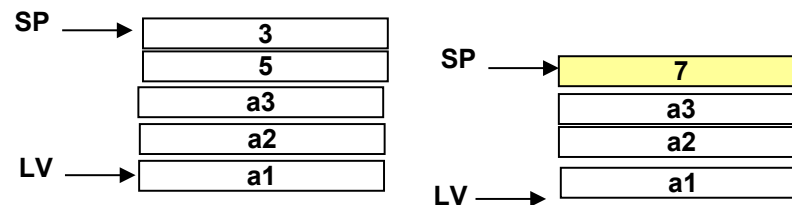
```
.main  
bipush 5  
bipush 3  
iand  
halt  
.end-main
```



# Commande IJVM ISA IOR

- instruction : IOR
  - Dépile le sommet de la pile 2 fois et...
  - place la valeur de leur opération and bit à bit en sommet de pile
- exemple:

```
.main  
bipush 5  
bipush 3  
ior  
halt  
.end-main
```



# Commande IJVM ISA ISTORE VAR

- instruction : ISTORE var
  - Dépile le sommet de la pile...
  - et range dans les variables locales (via un label « var »)

- exemple:

```
.main
```

```
.var
```

```
i
```

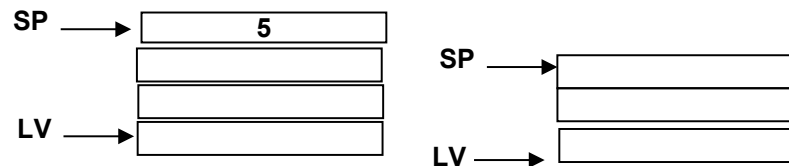
```
.end-var
```

```
bipush 5
```

```
istore i
```

```
halt
```

```
.end-main
```

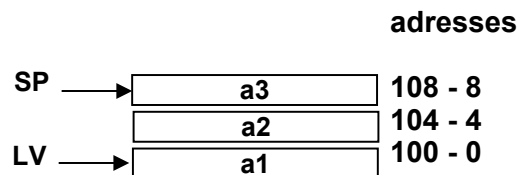


Où est la valeur 5??

# La notion de pile système

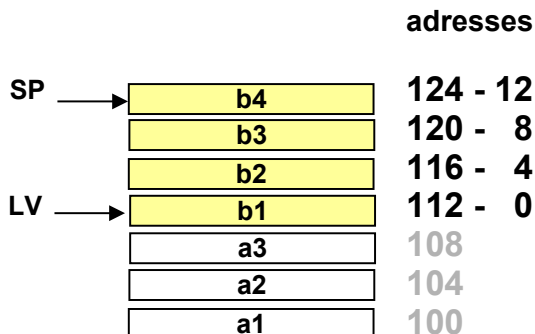
- La notion de procédure introduit la notion de variables locales visibles uniquement de l'intérieur de la procédure.
- Comment gérer l'espace mémoire réservé à ces variables de façon à supporter les appels récursifs?
- Une zone de la mémoire stocke les variables locales.
- Ces variables n'ont pas d'adresse absolue
- Un registre pointeur **LV** (Local variables pointer) pointe sur le début de la zone des variables locales de la procédure courante (la première adresse des variables locales)
- Le registre de pointeur de pile **SP** pointe le sommet de la zone mémoire occupée par les variables locales (la dernière adresse des variables locales)

dans la procédure A  
3 variables locales  
a1, a2, a3



absolue - locale

dans la procédure B appelée par A  
4 variables locales  
b1, b2, b3, b4



dans la procédure C appelée par A  
5 variables locales  
c1, c2, c3, c4, c5



# Le modèle mémoire de l'IJVM

- L'IJVM peut être vue comme un ensemble de 4 Go ou 1 Giga mots
- Aucune adresse n'est manipulée directement
- Les adresses sont manipulées à travers des pointeurs
- Les instructions n'accèdent à la mémoire qu'à travers ces pointeurs
- 4 zones mémoires sont prédéfinies

- 1 pool de constantes
  - zone non modifiable chargée en mémoire au lancement du programme
  - **CPP** contient l'adresse du début du pool
- Le bloc de variables locales
  - zone mémoire de taille fixe allouée lors de l'appel de la procédure.
  - C'est la pile des données.
  - Le registre **LV** pointe sur la première variable de la pile
  - Le registre **SP** pointe sur le sommet de la pile
- La pile d'opérandes
  - Elle est placée au-dessus de la pile des données
  - **Pour une procédure, pile d'opérandes et pile de données forment un seul bloc**
  - **SP** pointe sur le sommet de la pile d'opérandes
- La zone méthodes
  - Zone qui contient le programme
  - Un registre **PC** contient l'adresse de l'instruction courante

Pool de  
constantes

operande  
courant de la  
pile 2

variables  
locales du  
bloc 2

zone de  
variables  
locales du  
bloc 1

zone de  
méthodes

- CPP, LV, SP pointent des mots
- PC pointe des octets

# Commande IJVM ISA ISTORE VAR

- instruction : ISTORE var
  - Dépile le sommet de la pile...
  - et range dans les variables locales (via un label « var »)

- exemple:

main

.var

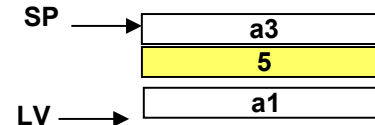
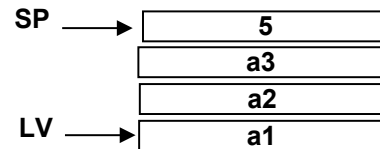
i

.end-var

bipush 5

istore i

halt



•end-main



# Commande IJVM ISA ILOAD VAR

- instruction : ILOAD var
  - Récupère la valeur via le label « var »...
  - et la place en sommet de pile

- exemple:

main

.var

i

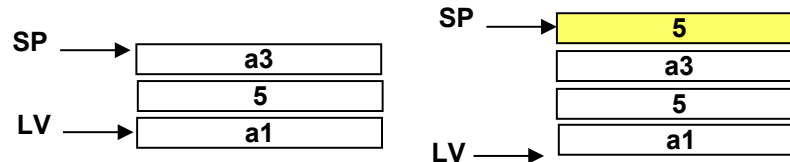
.end-var

bipush 5

istore i

iload i

halt



# Commande IJVM ISA IINC VAR X

- instruction : IINC var x
  - additionne une constante « x » à une variable locale « var »

- exemple:

main

.var

i

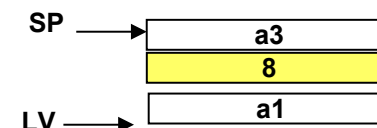
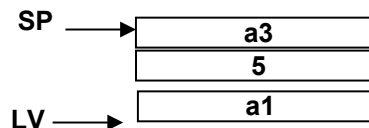
.end-var

bipush 5

istore i

iinc i 3

halt



.end-main

# Déclarations dans un programme IJVM

```
.constant  
mymax 100  
.end-constant
```

```
main  
  .var  
  i  
  .end-var  
  bipush 5  
  istore i  
  iinc i 3  
  halt  
  .end-main
```

# Déclarations dans un programme IJVM

```
.constant  
mymax 100  
.end-constant
```

```
.main  
.var  
i  
.end-var  
bipush 5  
istore i  
iinc i 3  
halt  
.end-main
```

**NB: il est aussi possible  
de déclarer des  
procédures:**

```
.method cmp(p1,p2)  
.var  
temp  
.end-var  
//corps de la  
//procédure  
IRETURN  
.end-method
```

# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM
j=1; k=2; i= j + k; k = 0; j = j - 1;	1 BIPUSH 1 2 ISTORE j 3 BIPUSH 2 4 ISTORE k 5 ILOAD j 6 ILOAD k 7 IADD 8 ISTORE i 9 BIPUSH 0 10 ISTORE k 11 ILOAD j 12 BIPUSH 1 13 ISUB 14 ISTORE j

# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM
j=1;	1 BIPUSH 1
k=2;	2 ISTORE j
i= j + k;	3 BIPUSH 2
k = 0;	4 ISTORE k
j = j – 1;	5 ILOAD j
	6 ILOAD k
	7 IADD
	8 ISTORE i
	9 BIPUSH 0
	10 ISTORE k
	11 ILOAD j
	12 BIPUSH 1
	13 ISUB
	14 ISTORE j

# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM	
j=1;	1	BIPUSH 1
k=2;	2	ISTORE j
i= j + k;	3	BIPUSH 2
k = 0;	4	ISTORE k
j = j - 1;	5	ILOAD j
	6	ILOAD k
	7	IADD
	8	ISTORE i
	9	BIPUSH 0
	10	ISTORE k
	11	ILOAD j
	12	BIPUSH 1
	13	ISUB
	14	ISTORE j

# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM	
j=1;	1	BIPUSH 1
k=2;	2	ISTORE j
i= j + k;	3	BIPUSH 2
k = 0;	4	ISTORE k
j = j - 1;	5	ILOAD j
	6	ILOAD k
	7	IADD
	8	ISTORE i
	9	BIPUSH 0
	10	ISTORE k
	11	ILOAD j
	12	BIPUSH 1
	13	ISUB
	14	ISTORE j



# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM	
j=1;	1	BIPUSH 1
k=2;	2	ISTORE j
i= j + k;	3	BIPUSH 2
k = 0;	4	ISTORE k
j = j - 1;	5	ILOAD j
	6	ILOAD k
	7	IADD
	8	ISTORE i
	9	BIPUSH 0
	10	ISTORE k
	11	ILOAD j
	12	BIPUSH 1
	13	ISUB
	14	ISTORE j

# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM	
j=1;	1	BIPUSH 1
k=2;	2	ISTORE j
i= j + k;	3	BIPUSH 2
k = 0;	4	ISTORE k
j = j - 1;	5	ILOAD j
	6	ILOAD k
	7	IADD
	8	ISTORE i
	9	BIPUSH 0
	10	ISTORE k
	11	ILOAD j
	12	BIPUSH 1
	13	ISUB
	14	ISTORE j

# Les ruptures de flot de contrôle

- instruction : GOTO
  - Saut dans le programme
  - à une position
  - indiquée via un LABEL

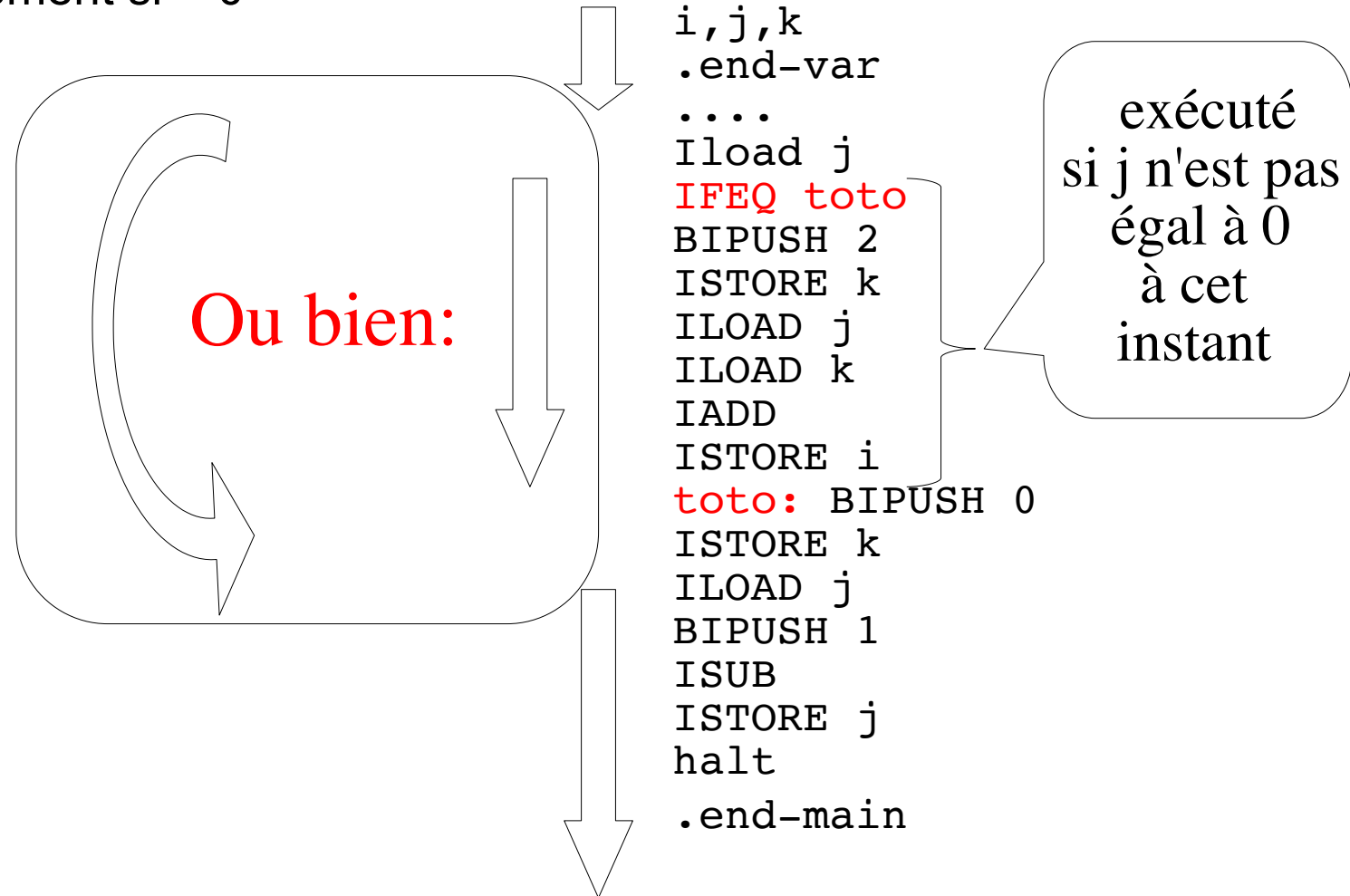
Voir Simulateur

```
main
.var
i,j,k
.end-var
BIPUSH 1
ISTORE j
GOTO toto
BIPUSH 2
ISTORE k
ILOAD j
ILOAD k
IADD
ISTORE i
toto: BIPUSH 0
ISTORE k
ILOAD j
BIPUSH 1
ISUB
ISTORE j
halt
.end-main
```

Jamais  
exécuté!

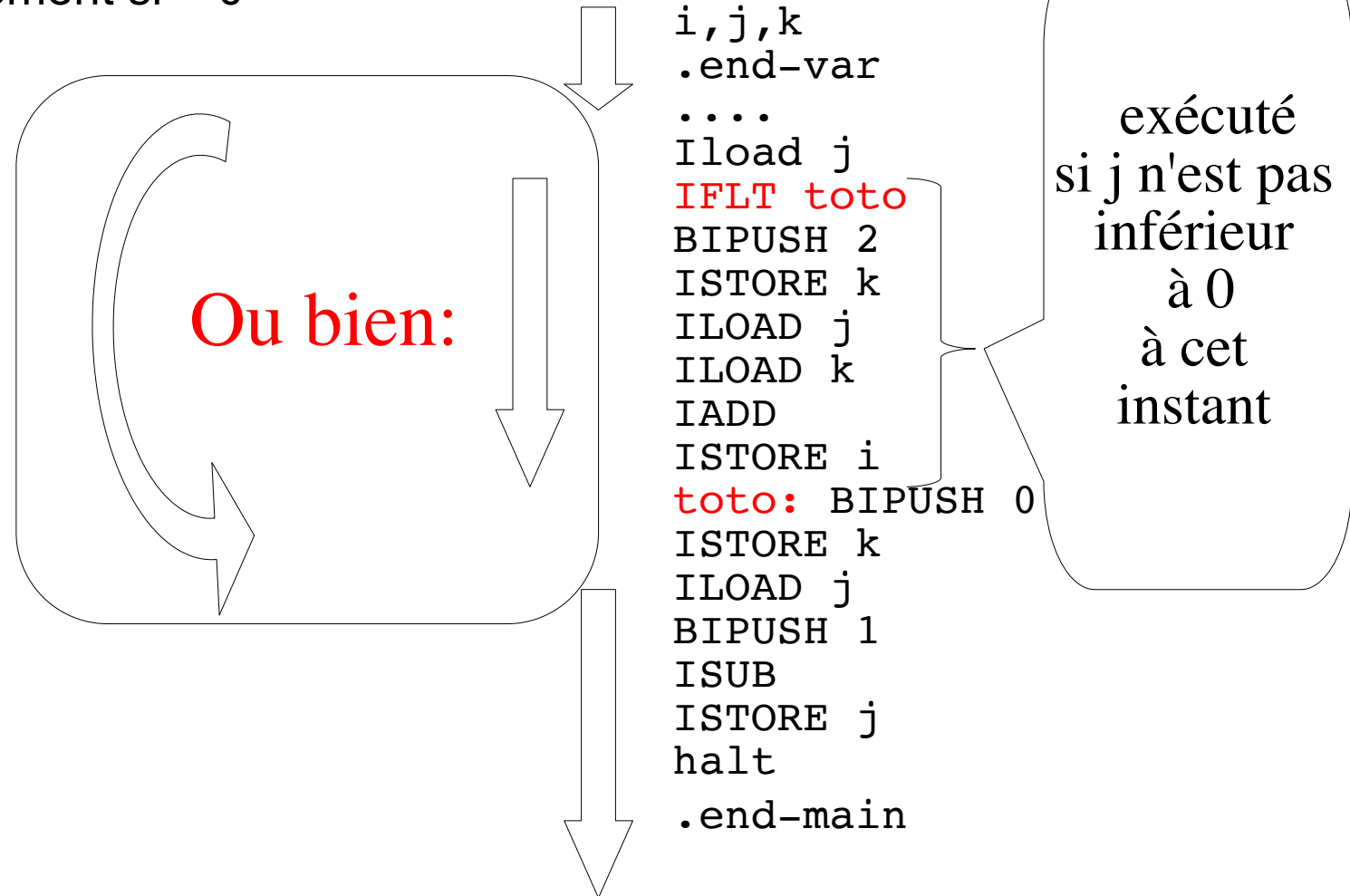
# Les ruptures de flot de contrôle

- instruction : IFEQ label
  - pop un mot de la pile
  - et branchement si = 0



# Les ruptures de flot de contrôle

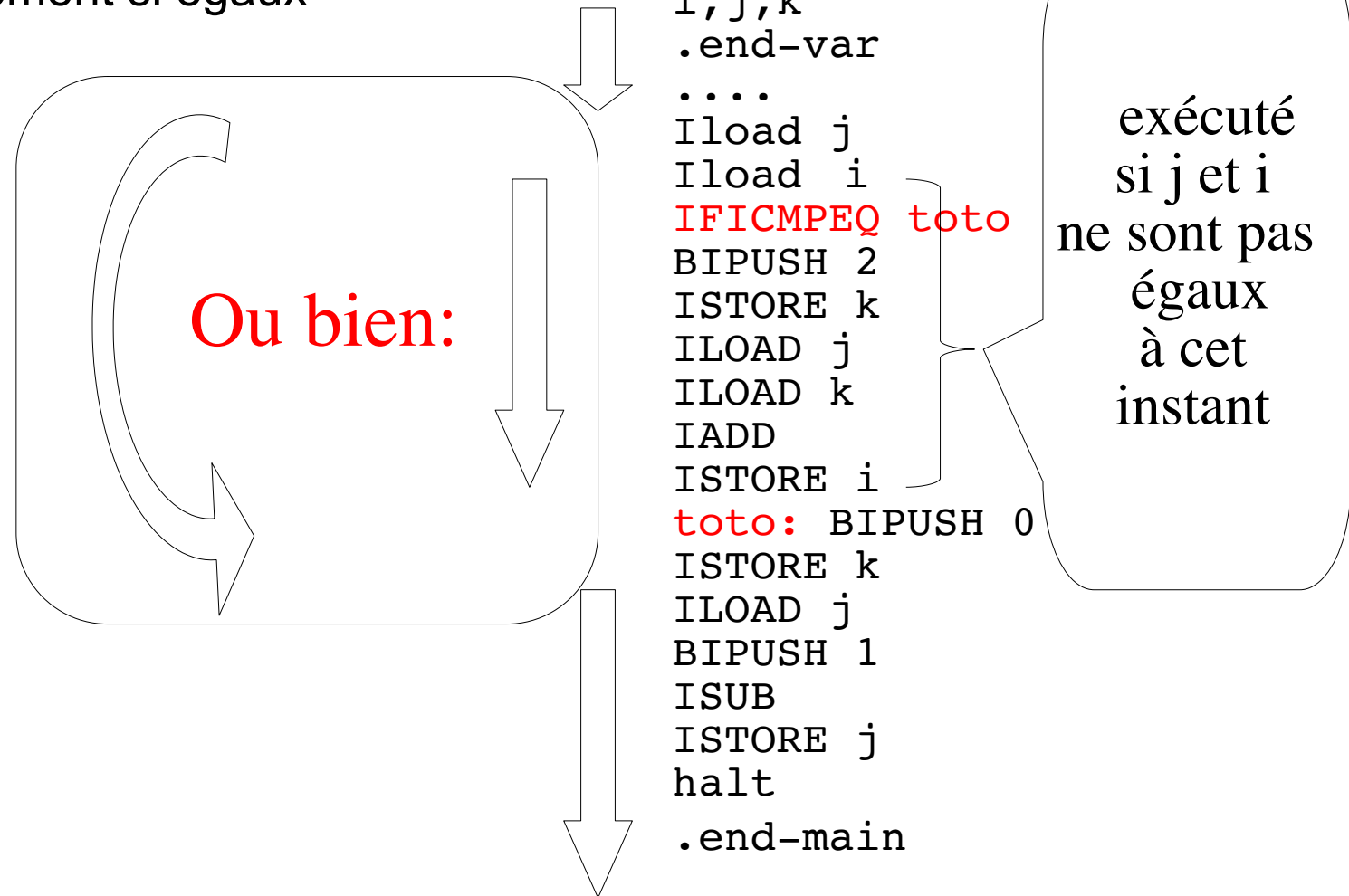
- instruction : IFLT label
  - pop un mot de la pile
  - et branchement si  $< 0$



# Les ruptures de flot de contrôle

- instruction :IFICMPEQ label

- pop deux mot de la pile
- et branchement si égaux



# Pourquoi il n'y a pas de Goto dans nos langages?

Go-to statement considered harmful (Edgar Dijkstra)

in Commun. ACM 11 (1968), 3: 147–148

## Go To Statement Considered Harmful

**Key Words and Phrases:** go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

**CR Categories:** 4.22, 5.23, 5.24

### EDITOR:

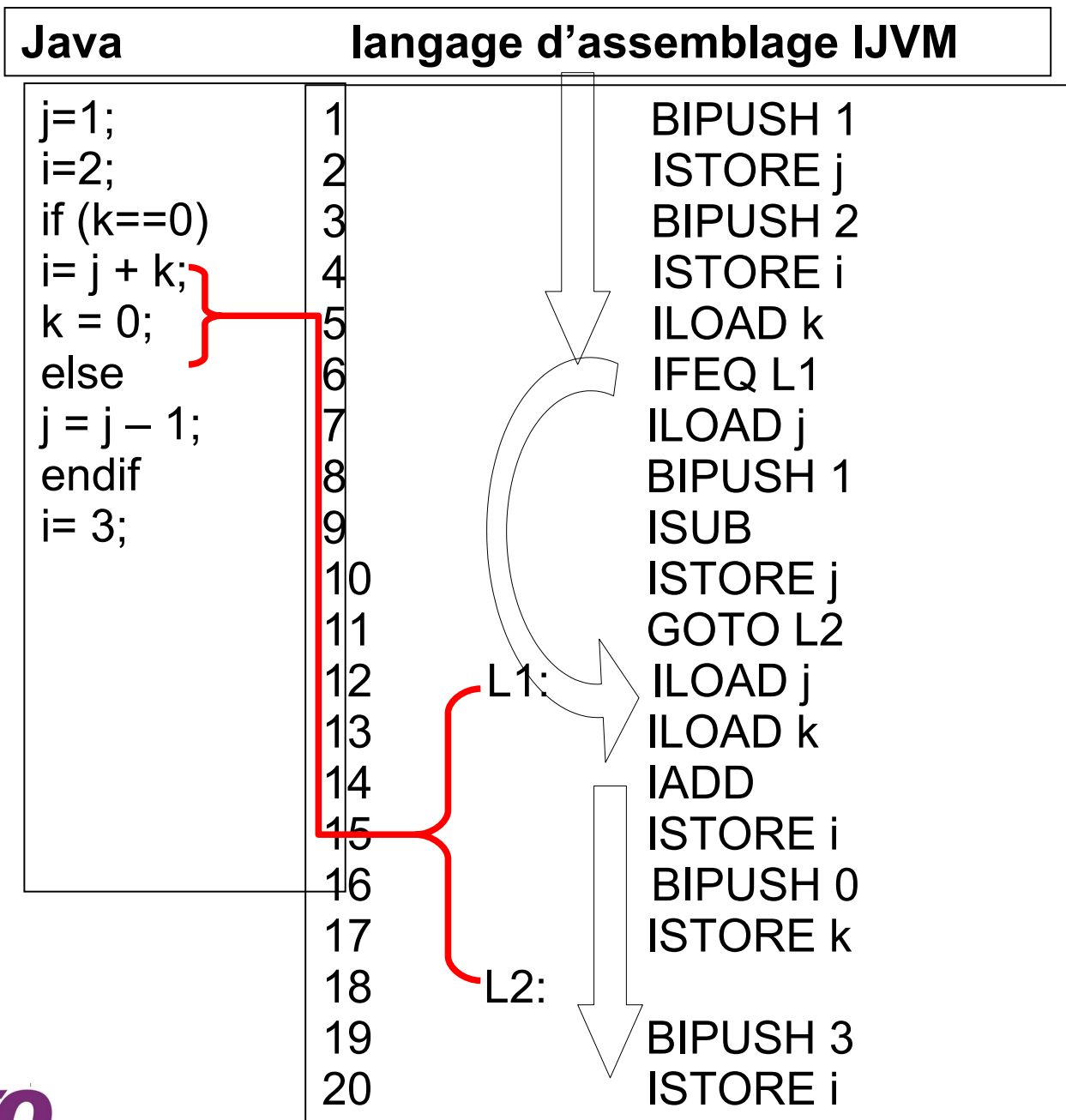
For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all “higher level” programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

# Compilation d'un IF

Java	langage d'assemblage JVM	
j=1;	1	BIPUSH 1
i=2;	2	ISTORE j
if (k==0)	3	BIPUSH 2
i = j + k;	4	ISTORE i
k = 0;	5	ILOAD k
else	6	IFEQ L1
j = j - 1;	7	ILOAD j
endif	8	BIPUSH 1
i= 3;	9	ISUB
	10	ISTORE j
	11	GOTO L2
	12	L1: ILOAD j
	13	ILOAD k
	14	IADD
	15	ISTORE i
	16	BIPUSH 0
	17	ISTORE k
	18	L2: BIPUSH 3
	19	
	20	ISTORE i



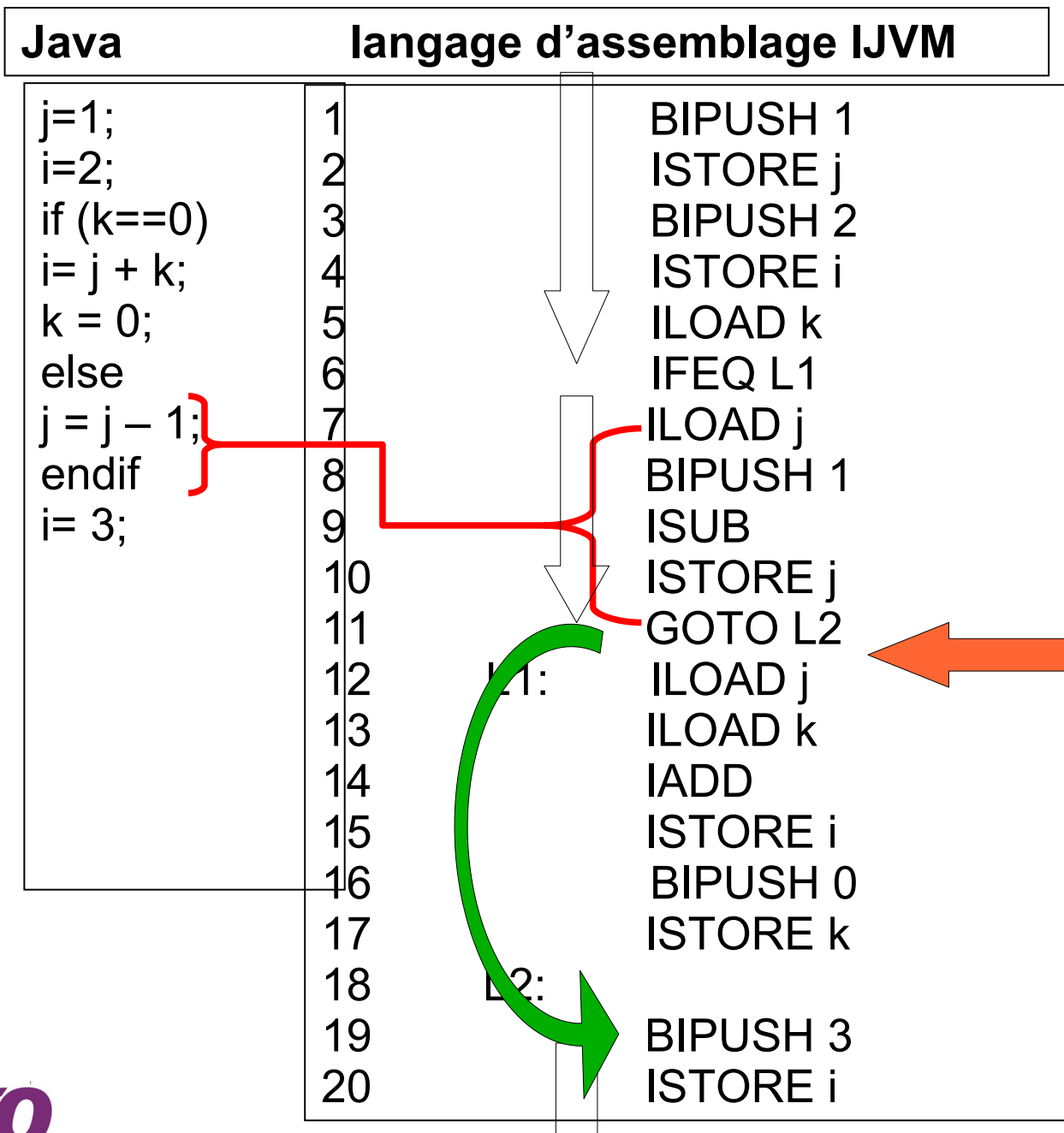
# Compilation d'un IF



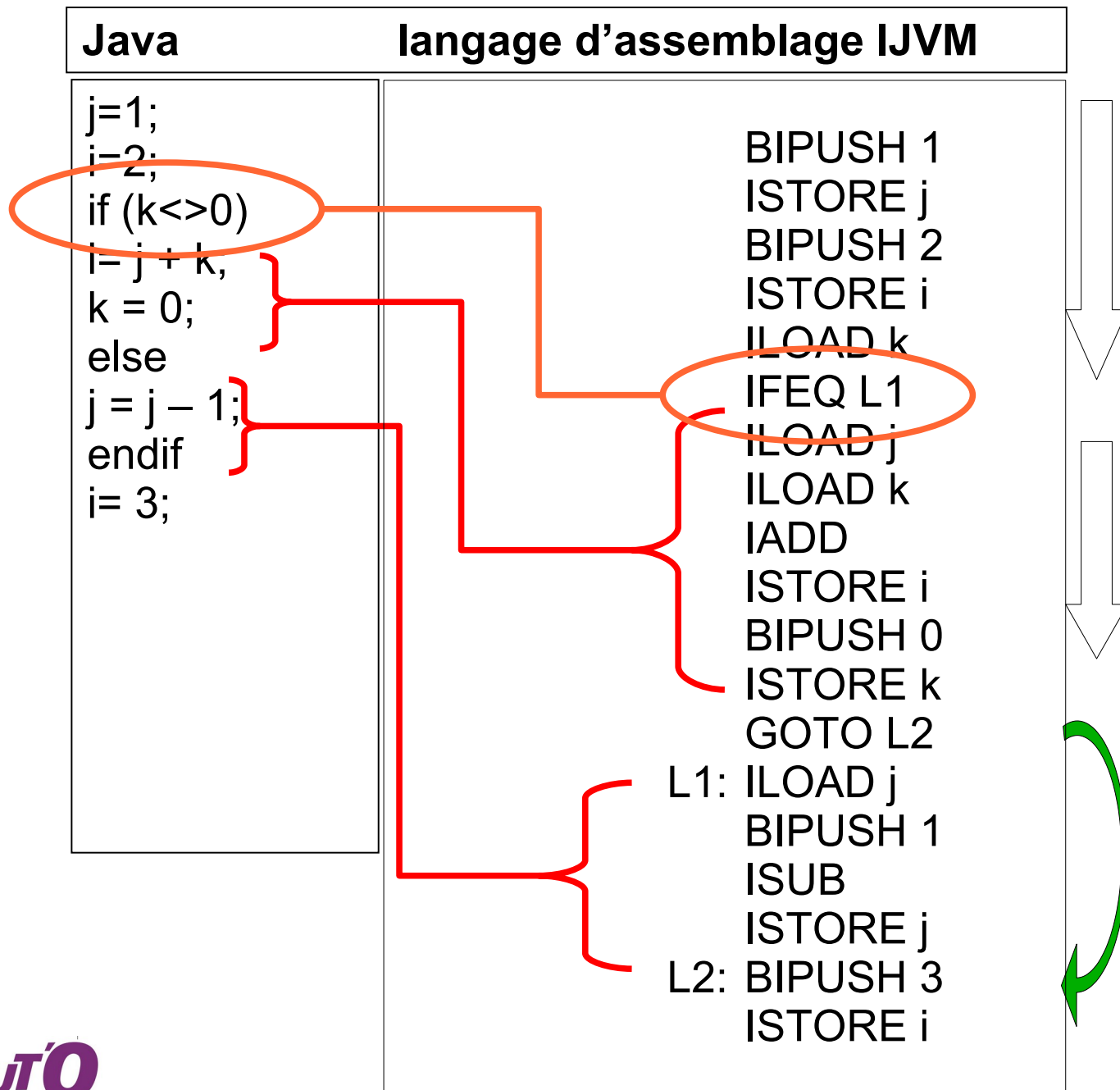
# Compilation d'un IF

Java	langage d'assemblage JVM	
j=1;	1	BIPUSH 1
i=2;	2	ISTORE j
if (k==0)	3	BIPUSH 2
i = j + k;	4	ISTORE i
k = 0;	5	ILOAD k
else	6	IFEQ L1
j = j - 1;	7	ILOAD j
endif	8	BIPUSH 1
i = 3;	9	ISUB
	10	ISTORE j
	11	GOTO L2
	12	L1: ILOAD j
	13	ILOAD k
	14	IADD
	15	ISTORE i
	16	BIPUSH 0
	17	ISTORE k
	18	L2: BIPUSH 3
	19	
	20	ISTORE i

# Compilation d'un IF



# Compilation d'un IF avec condition inversée



# Méthode générale

Test	Instruction ISA	Inversion ordre du code par rapport au JAVA
$X == 0$	IFEQ	OUI
$X \neq 0$	IFEQ	NON
$X < 0$	IFLT	OUI
$X \geq 0$	IFLT	NON

# Méthode générale

Test	Equivalent	Instruction ISA	Inversion ordre du code par rapport au JAVA
$X == Y$	$X - Y == 0$	IFEQ	OUI
$X <> Y$	$X - Y <> 0$	IFEQ	NON
$X < Y$	$X - Y < 0$	IFLT	OUI
$X >= Y$	$X - Y >= 0$	IFLT	NON

# Méthode générale

Test	Instruction ISA	Inversion ordre du code par rapport au JAVA
$X == 0$	IFEQ	OUI
$X <> 0$	IFEQ	NON
$X < 0$	IFLT	OUI
$X \geq 0$	IFLT	NON

$$X \leq 0 \Leftrightarrow -X \geq 0$$

$$X > 0 \Leftrightarrow -X < 0$$

# Méthode générale

Test	Instruction ISA	Inversion ordre du code par rapport au JAVA
$X == 0$	IFEQ	OUI
$X \neq 0$	IFEQ	NON
$X < 0$	IFLT	OUI
$X \geq 0$	IFLT	NON

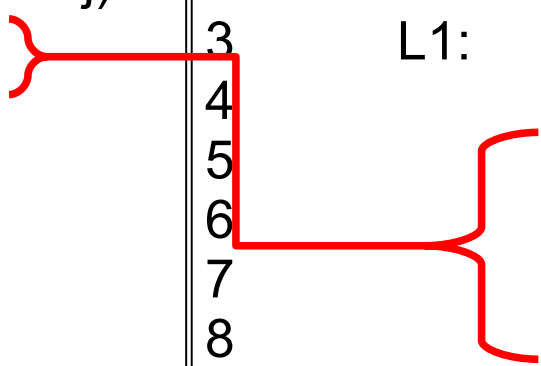
$$X \leq 0 \Leftrightarrow -X \geq 0$$

$$X > 0 \Leftrightarrow -X < 0$$

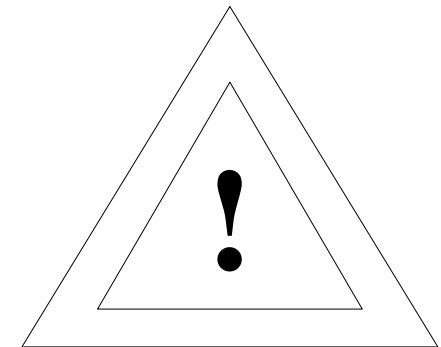
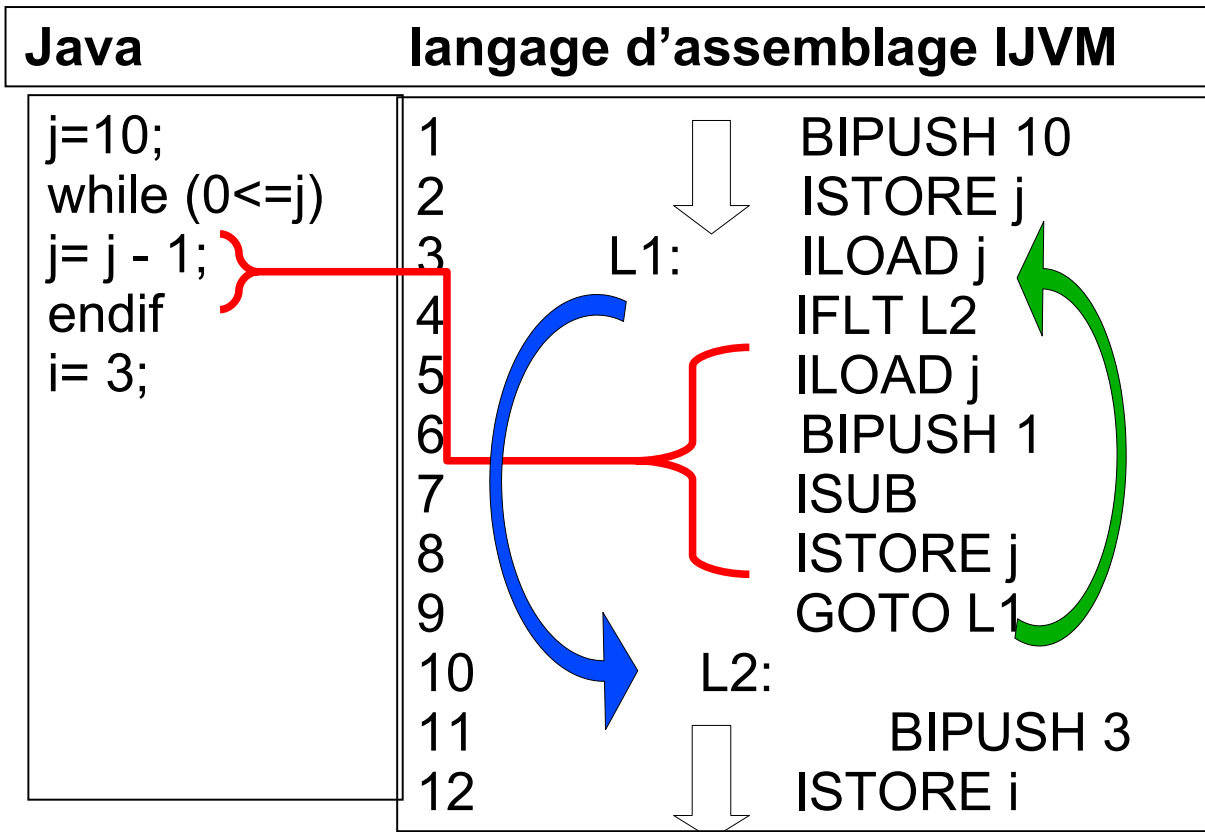


# Compilation d'une boucle while

Java	langage d'assemblage JVM	
j=10;	1	BIPUSH 10
while (0<=j)	2	ISTORE j
j= j - 1;	3	L1: ILOAD j
endif	4	IFLT L2
i= 3;	5	ILOAD j
	6	BIPUSH 1
	7	ISUB
	8	ISTORE j
	9	GOTO L1
	10	L2:
	11	BIPUSH 3
	12	ISTORE i

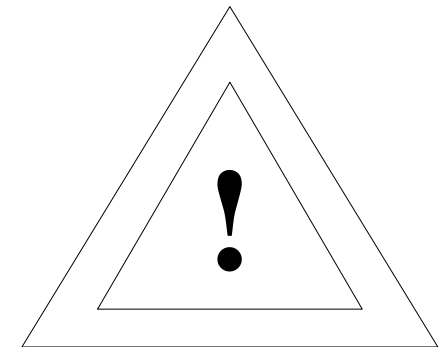
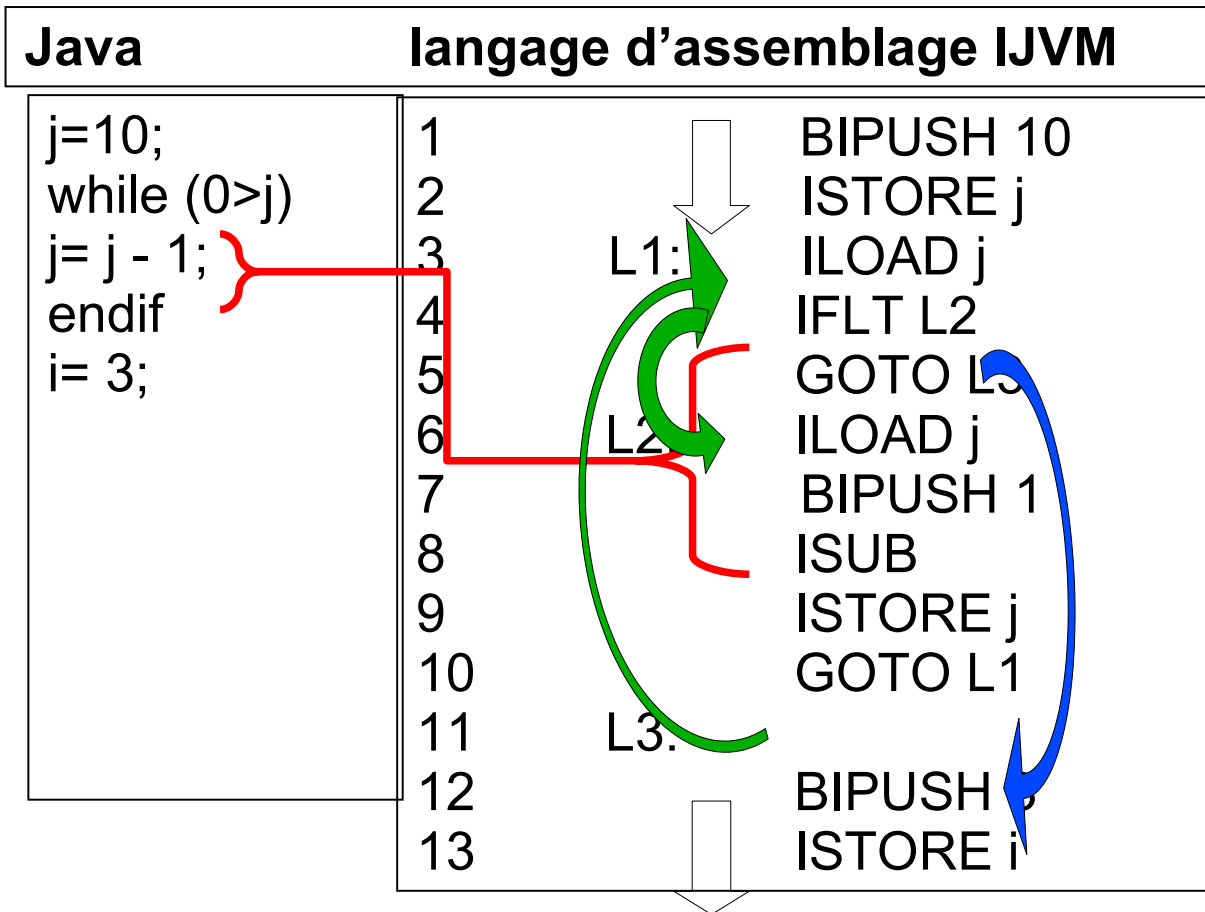


# Compilation d'une boucle while



Inversion  
du test

# Compilation d'une boucle while (variante)



Pas d'  
inversion  
du test

# Les instructions de l'IJVM utilisent la pile

- Chaque instruction comprend un code opération et parfois un code opérande
- Ici le code correspond au livre de Tanenbaum

Hex.	Mnémonique	description
0x10	BIPUSH octet	push un octet dans la pile
0x59	DUP	duplique le mots du sommet de la pile dans la pile
0xA7	GOTO offset	branchement inconditionnel
0x60	IADD	pop 2 mots de la pile et push leur somme dans la pile
0x7E	IAND	pop 2 mots de la pile et push leur ET dans la pile
0x99	IFEQ offset	pop un mot de la pile et branchement si = 0
0x9B	IFLT offset	pop un mot de la pile et branchement si < 0
0x9F	IF_ICMPEQ offset	pop 2 mots de pile et branchement si égaux
0x84	IINC numvar const	additionne une constante à une variable locale
0x15	ILOAD numvar	push une variable locale dans la pile
0xB6	INVOKEVIRTUAL dep	invoque une méthode
0x80	IOR	pop 2 mots de la pile et push leur OU dans la pile
0xAC	IRETURN	retour de méthode avec une valeur entière
0x36	ISTORE numvar	pop un mot de la pile et range dans les variables locales
0x64	ISUB	pop les 2 mots ds la pile et push la différence dans la pile
0x13	LDCW index	Push une constante depuis la zone de constantes dans la pile
0x00	NOP	ne fait rien
0x57	POP	efface le mot au sommet de la pile
0x5F	SWAP	permutte les 2 mots au sommet de la pile
0xC4	WIDE	préfixe d'instruction

# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM	byte code
i = j + k; If (i == 3) k = 0; else j = j - 1	0 ILOAD j 1 ILOAD k 2 IADD 3 ISTORE i 4 ILOAD i 5 BIPUSH 3 6 IFICMPEQ L1 7 ILOAD j 8 BIPUSH 1 9 ISUB 10 ISTORE j 11 GOTO L2 12 L1: BIPUSH 0 13 ISTORE k 14 L2:	

# Les instructions de l'IJVM utilisent la pile

- Chaque instruction comprend un code opération et parfois un code opérande
- Ici le code correspond au livre de Tanenbaum

Hex.	Mnémonique	description
0x10	BIPUSH octet	push un octet dans la pile
0x59	DUP	duplique le mots du sommet de la pile dans la pile
0xA7	GOTO offset	branchement inconditionnel
0x60	IADD	pop 2 mots de la pile et push leur somme dans la pile
0x7E	IAND	pop 2 mots de la pile et push leur ET dans la pile
0x99	IFEQ offset	pop un mot de la pile et branchement si = 0
0x9B	IFLT offset	pop un mot de la pile et branchement si < 0
0x9F	IF_ICMPEQ offset	pop 2 mots de pile et branchement si égaux
0x84	IINC numvar const	additionne une constante à une variable locale
0x15	ILOAD numvar	push une variable locale dans la pile
0xB6	INVOKEVIRTUAL dep	invoque une méthode
0x80	IOR	pop 2 mots de la pile et push leur OU dans la pile
0xAC	IRETURN	retour de méthode avec une valeur entière
0x36	ISTORE numvar	pop un mot de la pile et range dans les variables locales
0x64	ISUB	pop les 2 mots ds la pile et push la différence dans la pile
0x13	LDCW index	Push une constante depuis la zone de constantes dans la pile
0x00	NOP	ne fait rien
0x57	POP	efface le mot au sommet de la pile
0x5F	SWAP	permutte les 2 mots au sommet de la pile
0xC4	WIDE	préfixe d'instruction

# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM	byte code
i = j + k;	0 ILOAD j	0x15 0x02
If (i == 3)	1 ILOAD k	0x15 0x03
k = 0;	2 IADD	0x60
else	3 ISTORE i	0x36 0x01
j = j - 1	4 ILOAD i	0x15 0x01
	5 BIPUSH 3	0x10 0x03
	6 IFICMPEQ L1	0x9F 0x?? 0x??
	7 ILOAD j	0x15 0x02
	8 BIPUSH 1	0x10 0x01
	9 ISUB	0x64
	10 ISTORE j	0x36 0x02
	11 GOTO L2	0xA7 0x?? 0x??
	12 L1: BIPUSH 0	0x10 0x00
	13 ISTORE k	0x36 0x03
	14 L2:	

# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM	byte code
i = j + k;	0 ILOAD j	0x15 0x02
If (i == 3)	1 ILOAD k	0x15 0x03
k = 0;	2 IADD	0x60
else	3 ISTORE i	0x36 0x01
j = j - 1	4 ILOAD i	0x15 0x01
	5 BIPUSH 3	0x10 0x03
	6 IF_ICMPEQ L1	0x9F 0x00 0x0D
	7 ILOAD j	0x15 0x02
	8 BIPUSH 1	0x10 0x01
	9 ISUB	0x64
	10 ISTORE j	0x36 0x02
	11 GOTO L2	0xA7 0x?? 0x??
	12 L1: BIPUSH 0	0x10 0x00
	13 ISTORE k	0x36 0x03
	14 L2:	

Saut de  
13 octets



# Compilation d'un programme java pour l'IJVM

Java	langage d'assemblage IJVM	byte code
i = j + k;	0 ILOAD j	0x15 0x02
If (i == 3)	1 ILOAD k	0x15 0x03
k = 0;	2 IADD	0x60
else	3 ISTORE i	0x36 0x01
j = j - 1	4 ILOAD i	0x15 0x01
	5 BIPUSH 3	0x10 0x03
	6 IF_ICMPEQ L1	0x9F 0x00 0x0D
	7 ILOAD j	0x15 0x02
	8 BIPUSH 1	0x10 0x01
	9 ISUB	0x64
	10 ISTORE j	0x36 0x02
	11 GOTO L2	0xA7 0x00 0x07
	12 L1: BIPUSH 0	0x10 0x00
	13 ISTORE k	0x36 0x03
	14 L2:	

Saut de  
13 octets

Saut de  
7 octets

# Les instructions de l'IJVM utilisent la pile

- Chaque instruction comprend un code opération et parfois un code opérande
- Ici le code correspond au livre de Tanenbaum

Hex.	Mnémonique	description
0x10	BIPUSH octet	push un octet dans la pile
0x59	DUP	duplique le mots du sommet de la pile dans la pile
0xA7	GOTO offset	branchement inconditionnel
0x60	IADD	pop 2 mots de la pile et push leur somme dans la pile
0x7E	IAND	pop 2 mots de la pile et push leur ET dans la pile
0x99	IFEQ offset	pop un mot de la pile et branchement si = 0
0x9B	IFLT offset	pop un mot de la pile et branchement si < 0
0x9F	IF_ICMPEQ offset	pop 2 mots de pile et branchement si égaux
0x84	IINC numvar const	additionne une constante à une variable locale
0x15	ILOAD numvar	push une variable locale dans la pile
0xB6	INVOKEVIRTUAL dep	invoque une méthode
0x80	IOR	pop 2 mots de la pile et push leur OU dans la pile
0xAC	IRETURN	retour de méthode avec une valeur entière
0x36	ISTORE numvar	pop un mot de la pile et range dans les variables locales
0x64	ISUB	pop les 2 mots ds la pile et push la différence dans la pile
0x13	LDCW index	Push une constante depuis la zone de constantes dans la pile
0x00	NOP	ne fait rien
0x57	POP	efface le mot au sommet de la pile
0x5F	SWAP	permuté les 2 mots au sommet de la pile
0xC4	WIDE	préfixe d'instruction

## ET le saut arrière?

Java	langage d'assemblage IJVM	byte code
<pre> j=10; while (0&lt;=j) j= j - 1; endif i= 3; </pre>	<pre> 1      BIPUSH 10 2      ISTORE j 3      L1: ILOAD j 4      IFLT L2 5      ILOAD j 6      BIPUSH 1 7      ISUB 8      ISTORE j 9      GOTO L1 10     L2: 11     BIPUSH 3 12     ISTORE i </pre>	

## ET le saut arrière?

Java	langage d'assemblage JVM	byte code
j=10;	1 BIPUSH 10	0x10 0x0A
while (0<=j)	2 ISTORE j	0x36 0x02
j= j - 1;	3 L1: ILOAD j	0x15 0x02
endif	4 IFLT L2	0x9B 0x00 0x0D
i= 3;	5 ILOAD j	0x15 0x02
	6 BIPUSH 1	0x10 0x01
	7 ISUB	0x64
	8 ISTORE j	0x36 0x02
	9 GOTO L1	0xA7 0xFF 0xF4
	10 L2:	
	11 BIPUSH 3	0x10 0x03
	12 ISTORE i	0x36 0x01
	13	
	14	

Saut de  
-12 octets

# Les instructions de l'IJVM utilisent la pile

- Chaque instruction comprend un code opération et parfois un code opérande
- Ici le code correspond au livre de Tanenbaum

Hex.	Mnémonique	description
0x10	BIPUSH octet	push un octet dans la pile
0x59	DUP	duplique le mots du sommet de la pile dans la pile
0xA7	GOTO offset	branchement inconditionnel
0x60	IADD	pop 2 mots de la pile et push leur somme dans la pile
0x7E	IAND	pop 2 mots de la pile et push leur ET dans la pile
0x99	IFEQ offset	pop un mot de la pile et branchement si = 0
0x9B	IFLT offset	pop un mot de la pile et branchement si < 0
0x9F	IF_ICMPEQ offset	pop 2 mots de pile et branchement si égaux
0x84	IINC numvar const	additionne une constante à une variable locale
0x15	ILOAD numvar	push une variable locale dans la pile
0xB6	INVOKEVIRTUAL dep	invoque une méthode
0x80	IOR	pop 2 mots de la pile et push leur OU dans la pile
0xAC	IRETURN	retour de méthode avec une valeur entière
0x36	ISTORE numvar	pop un mot de la pile et range dans les variables locales
0x64	ISUB	pop les 2 mots ds la pile et push la différence dans la pile
0x13	LDC_W index	Push une constante depuis la zone de constantes dans la pile
0x00	NOP	ne fait rien
0x57	POP	efface le mot au sommet de la pile
0x5F	SWAP	permutte les 2 mots au sommet de la pile
0xC4	WIDE	préfixe d'instruction

# Les Fonctions (ou Méthodes)

```
.constant  
quatre 4  
.end-constant
```

```
.main
```

```
.var
```

```
i
```

```
j
```

```
k
```

```
.end-var
```

```
BIPUSH 1
```

```
LDCW quatre
```

```
BIPUSH 5
```

```
INVOKEVIRTUAL toto
```

```
BIPUSH 3
```

```
IADD
```

```
.end-main
```

```
.method toto(x,y)
```

```
.var
```

```
loc
```

```
j
```

```
.end-var
```

```
ILOAD x
```

```
BIPUSH -1
```

```
ISUB
```

```
ISTORE loc
```

```
ILOAD loc
```

```
ILOAD x
```

```
ILOAD y
```

```
IADD
```

```
IADD
```

```
IRETURN
```

```
.end-method
```

## Method Area

☐ Bin ☒ Hex

Addr	Content
0x40000	0xb6 0x00 0x02 0x00
0x40004	0x01 0x00 0x03 0x10
0x40008	0x01 0x13 0x00 0x01
0x4000c	0x10 0x05 0xb6 0x00
0x40010	0x03 0x10 0x03 0x60
0x40014	0x00 0x03 0x00 0x02
0x40018	0x15 0x01 0x10 0xff
0x4001c	0x64 0x36 0x03 0x15
0x40020	0x03 0x15 0x01 0x15
0x40024	0x02 0x60 0x60 0xac

## Constant Pool

Addr	Content
0x0	0x0
0x1	0x4
0x2	0x40003
0x3	0x40014