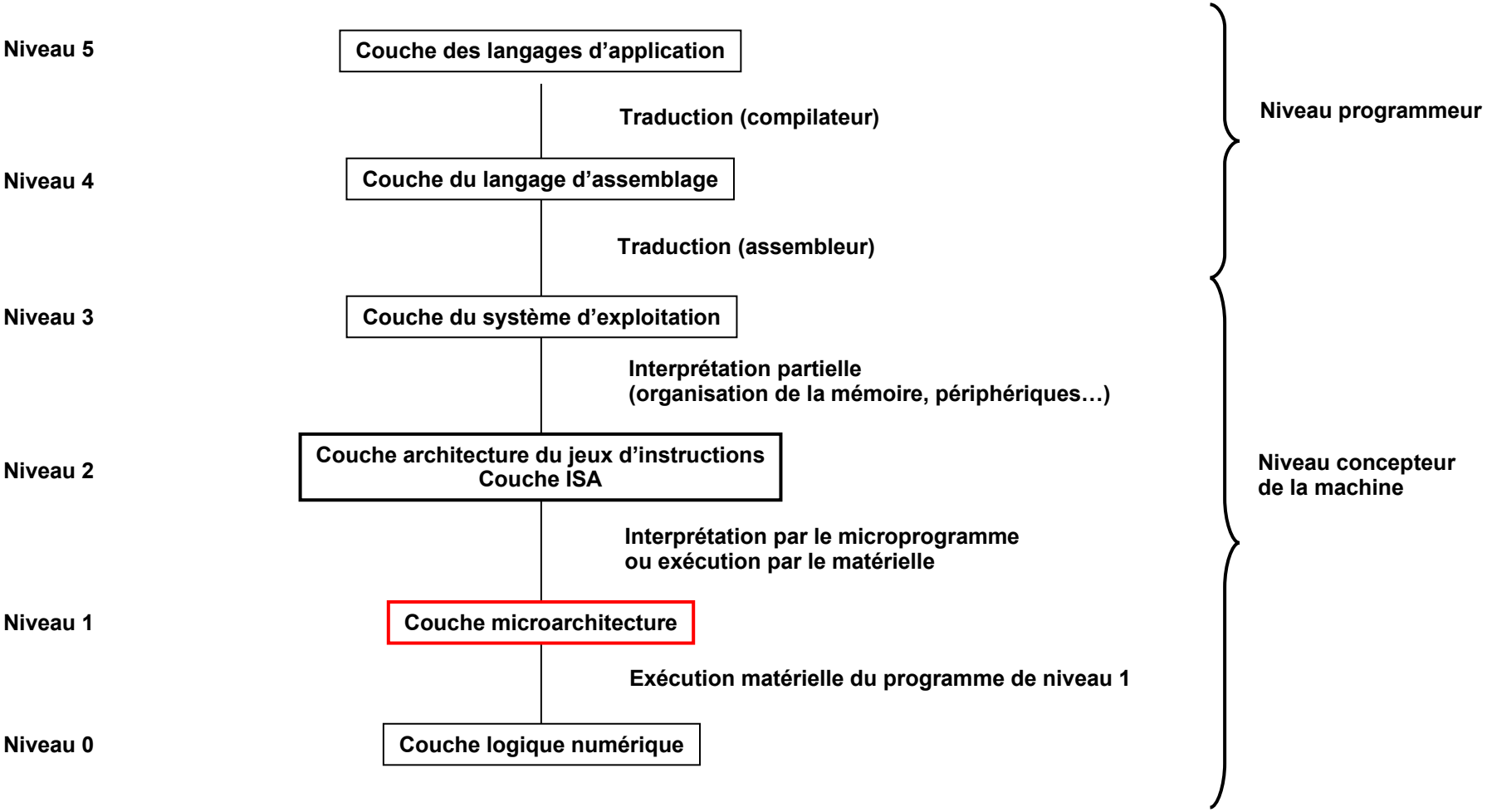
Architectures des microprocesseurs

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 (Univ. Rouen)

Architectures actuelles en six couches



Introduction

Rôle de la couche microarchitecture

- Exécution d'instructions complexes
- interprétation par un microprogramme
  1- extraction des instructions de la mémoire
  2- décodage des instructions
  3- exécution pas à pas

- commander les composants matériels de la microarchitecture selon les instructions de la couche ISA

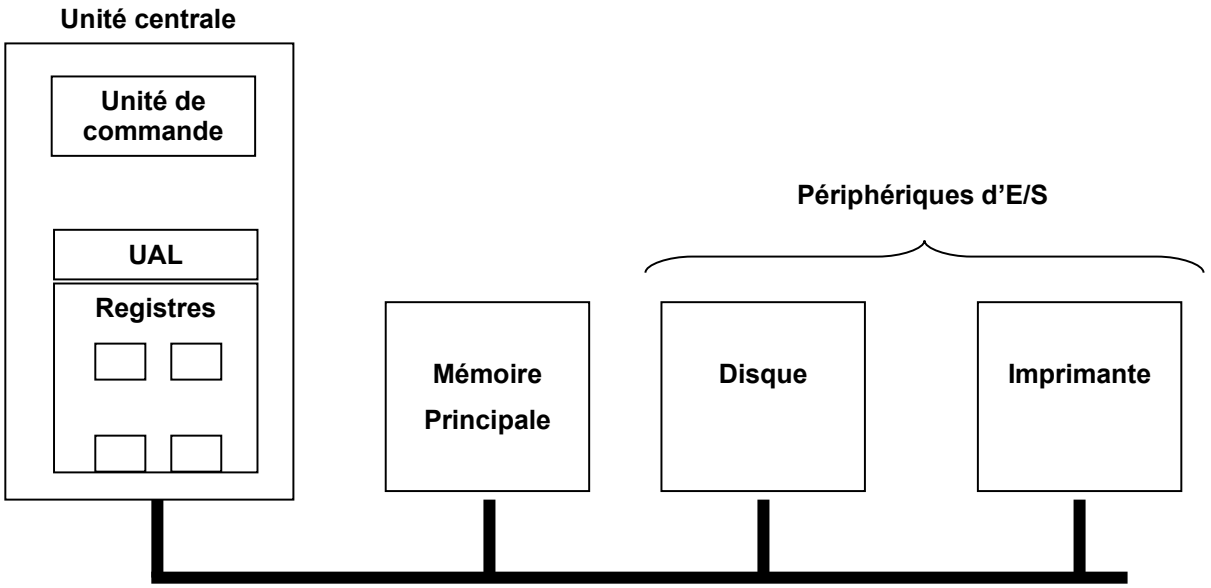
Approche

- On voit les choses comme un problème de programmation
  une instruction ISA est une fonction appelée par un programme principal
- L'état ou contexte du microprogramme
  Ensemble de variables partagées par toutes les fonctions (CP, Registres…)
- On s'appuie sur un modèle d'exécution des instructions
  cycle d'extraction/exécution
- Il n'y a pas de principes généraux qui sous-tendent la couche microarchitecture
  on s'appuie sur un exemple : *l'Integer Java Virtual Machine (IJVM)*

Machine de Von Neuman

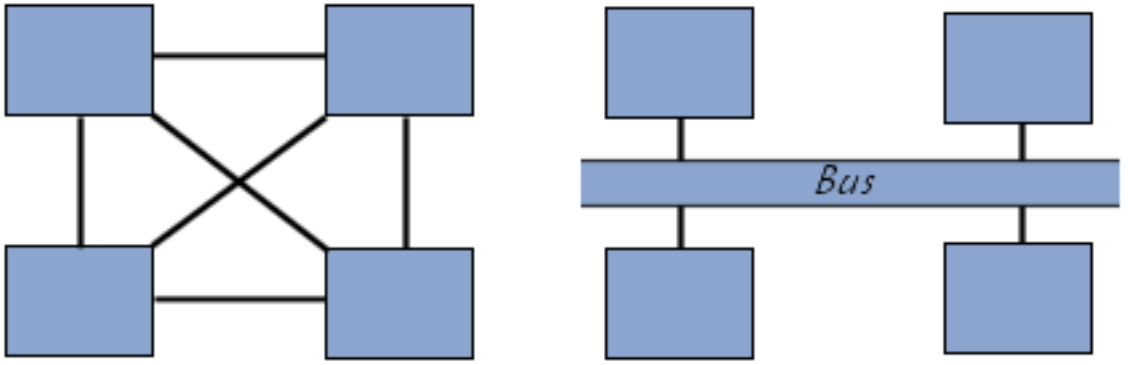
- arithmétique binaire
- programme numérique rangé en mémoire

première machine : EDSAC



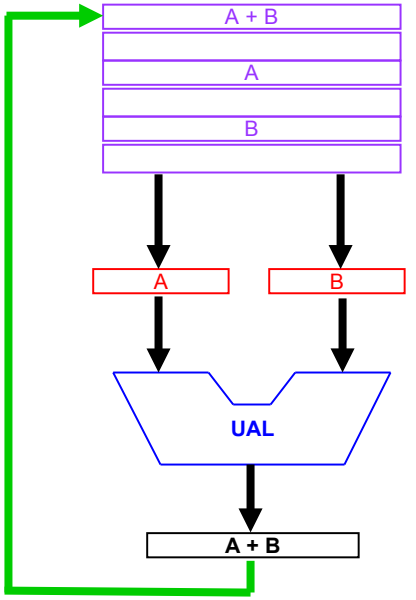
Notion de Bus

- On appelle bus, en informatique, un ensemble de liaisons physiques (câbles, pistes de circuits imprimés, ...) pouvant être exploitées en commun par plusieurs éléments matériels afin de communiquer.
- Les bus ont pour but de réduire le nombre de "voies" nécessaires à la communication des différents composants
- Caractériques:
  - Fréquence
  - Largeur



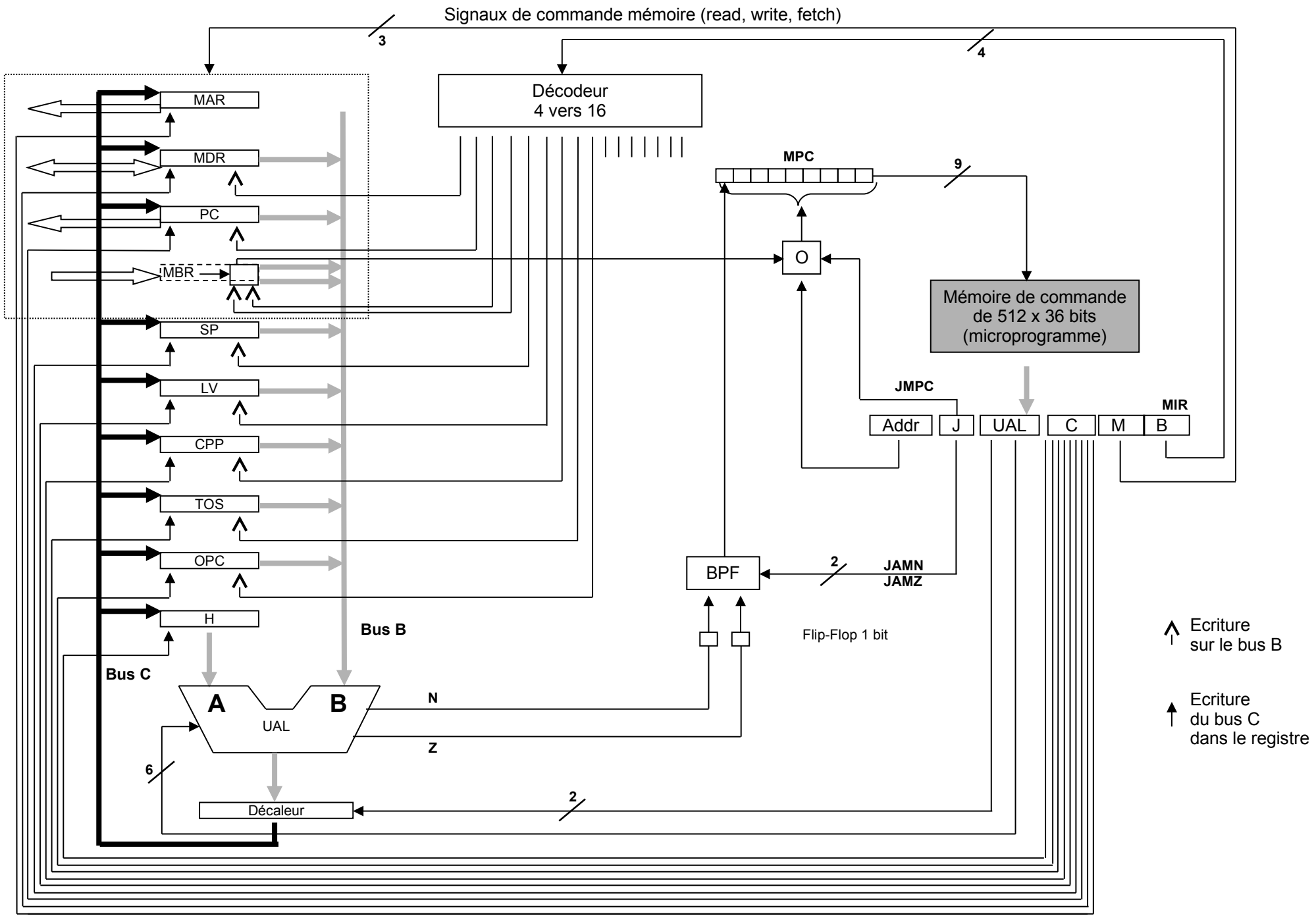
Organisation générale de l'unité centrale

Chemin des données
Registres
Bus
UAL
Mot mémoire



Cycle du chemin des données
Mot mémoire
Registres
UAL
Bus
Mot mémoire

La micro-architecture MIC1

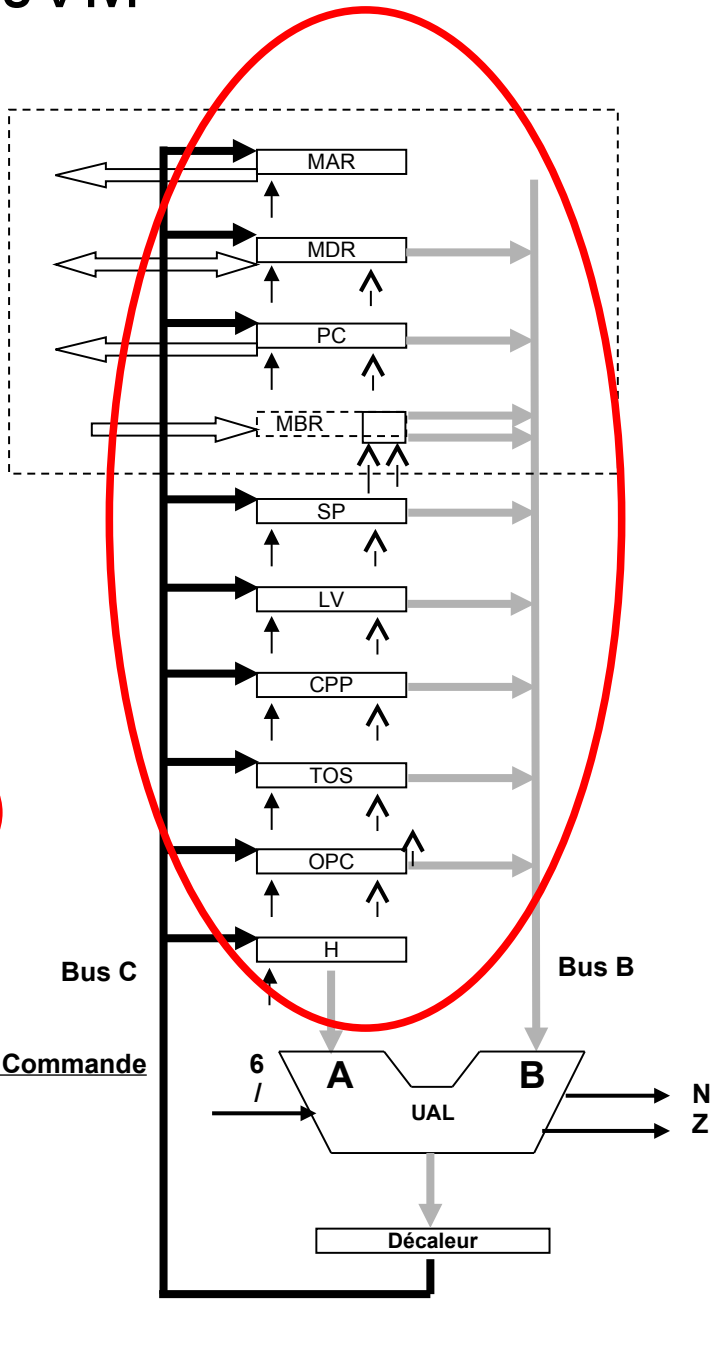


Modèle d'exécution de l'IJVM

Le chemin des données

- Regroupe l'UAL et les registres
- Registres 32 bits accessibles uniquement par le microprogramme
  - PC compteur ordinal
  - MAR: Memory Address Register
  - MDR: Memory Data Register (32 bits)
  - MBR: Memory Byte Register (8 bits)

- Tous les registres sauf MAR sont reliés au bus B pour écriture
- La sortie de l'UAL est écrite sur le bus C
- Toute donnée disponible sur le bus C peut être écrite dans un ou plusieurs registres simultanément
- L'UAL est commandée par 6 bits: F0 F1 INVA ENA ENB INC
  - F0 F1 permettent: A ET B, A OU B, $\overline{B}$, A + B
- H est un registre de maintien

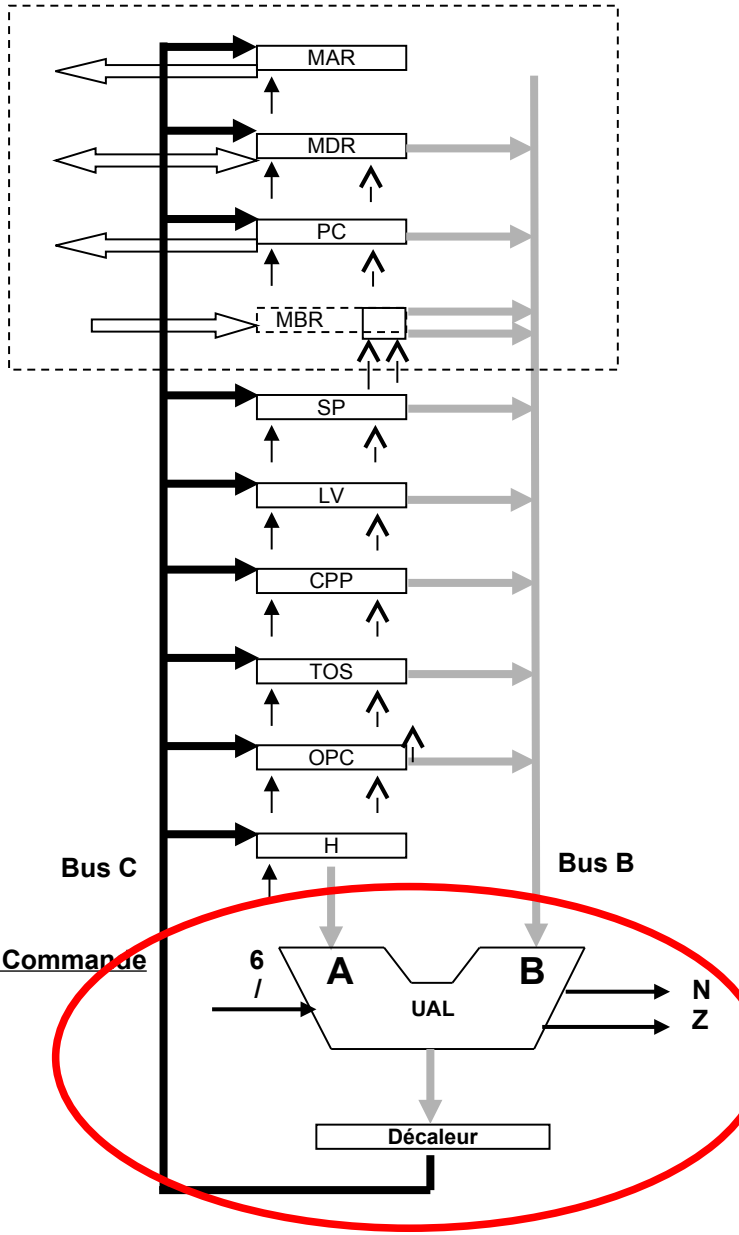


Modèle d'exécution de l'IJVM

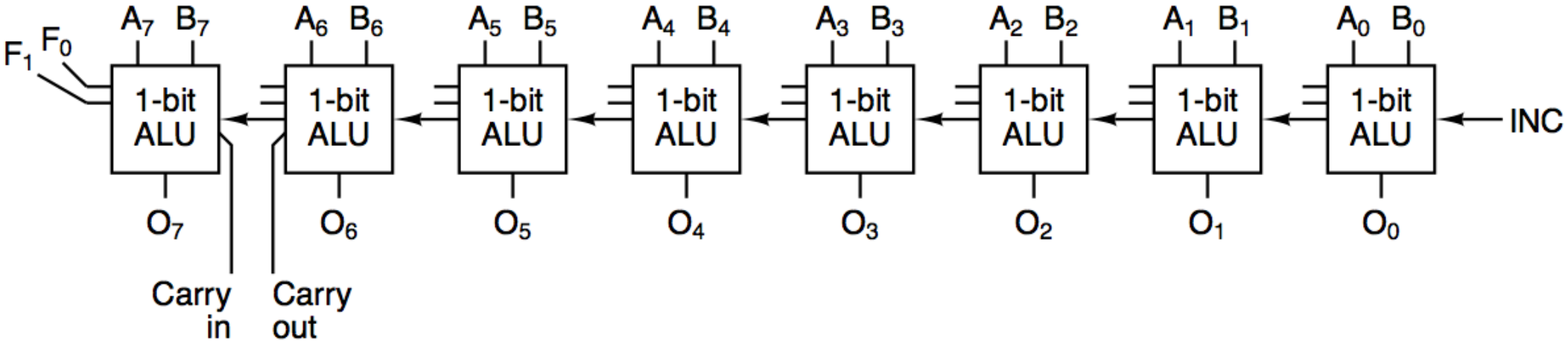
Le chemin des données

- Regroupe l'UAL et les registres
- Registres 32 bits accessibles uniquement par le microprogramme
  - PC compteur ordinal
  - MAR: Memory Address Register
  - MDR: Memory Data Register (32 bits)
  - MBR: Memory Byte Register (8 bits)

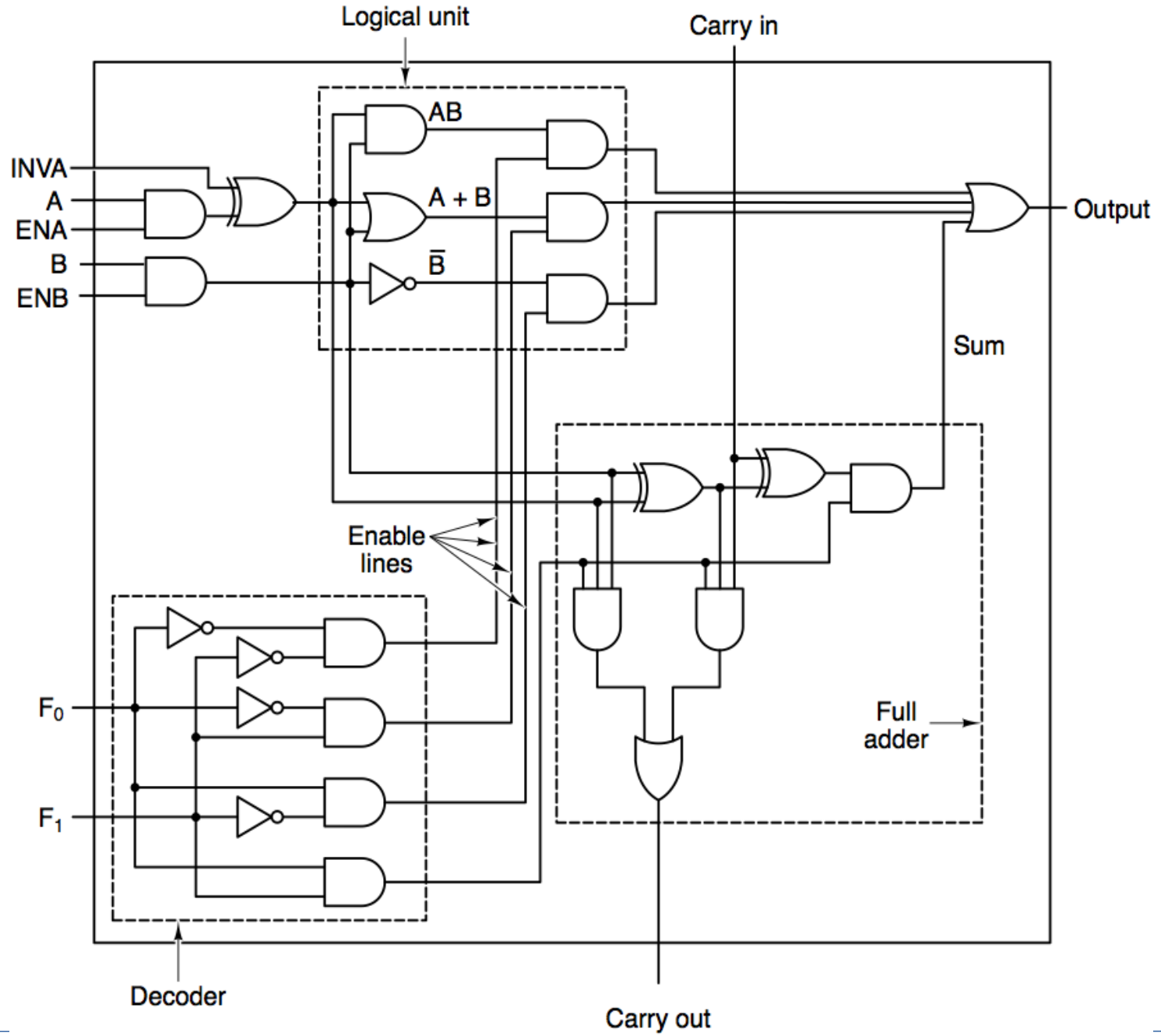
- Tous les registres sauf MAR sont reliés au bus B pour écriture
- La sortie de l'UAL est écrite sur le bus C
- Toute donnée disponible sur le bus C peut être écrite dans un ou plusieurs registres simultanément
- L'UAL est commandée par 6 bits: F0 F1 INVA ENA ENB INC
  - F0 F1 permettent: A ET B, A OU B, $\overline{B}$, A + B
- H est un registre de maintien



Passage à n bits



Une Unité Arithmétique et Logique



Modèle d'exécution de l'IJVM

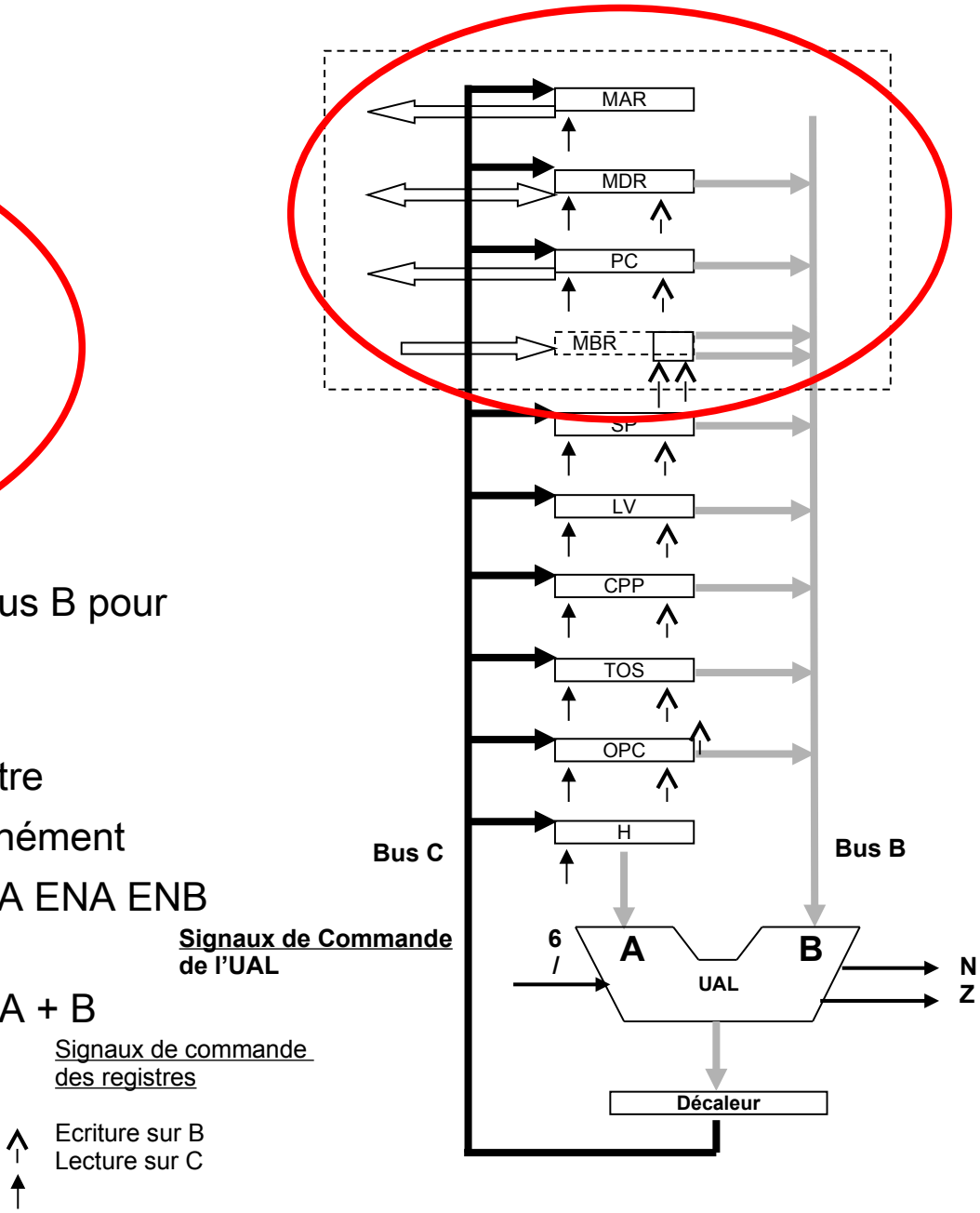
Fonctions de l'UAL

Fonction	Fo	F1	ENA	ENB	INVA	INC
A	0	1	1	0	0	0
B	0	1	0	1	0	0
$\overline{A}$	0	1	1	0	1	0
$\overline{B}$	1	0	1	1	0	0
A + B	1	1	1	1	0	0
A + B+1	1	1	1	1	0	1
A + 1	1	1	1	0	0	1
B + 1	1	1	0	1	0	1
B - A	1	1	1	1	1	1
B - 1	1	1	0	1	1	0
- A	1	1	1	0	1	1
A ET B	0	0	1	1	0	0
A OU B	0	1	1	1	0	0
0	0	1	0	0	0	0
1	1	1	0	0	0	1
- 1	0	1	0	0	1	0

Modèle d'exécution de l'IJVM

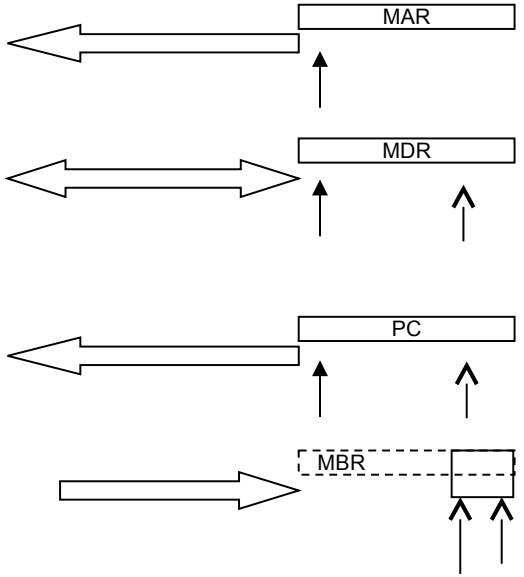
Le chemin des données

- Regroupe l'UAL et les registres
- Registres 32 bits accessibles uniquement par le microprogramme
  - PC compteur ordinal
  - MAR: Memory Address Register
  - MDR: Memory Data Register (32 bits)
  - MBR: Memory Byte Register (8 bits)
- Tous les registres sauf MAR sont reliés au bus B pour écriture
- La sortie de l'UAL est écrite sur le bus C
- Toute donnée disponible sur le bus C peut être écrite dans un ou plusieurs registres simultanément
- L'UAL est commandée par 6 bits: F0 F1 INVA ENA ENB INC
  - F0 F1 permettent: A ET B, A OU B, $\overline{B}$, A + B
- H est un registre de maintien



Modèle d'exécution de l'IJVM

Les opérations avec la mémoire



La communication avec la mémoire se fait par deux canaux de transmission

- Le port de 32 bits est commandé par 2 registres

MAR : Memory Address Register = registre d'adresses mémoire

MDR: Memory Data Register = registre de données mémoire

- Le port de 8 bits est commandé par le registre PC

PC contient l'adresse d'un octet en mémoire

L'octet à l'adresse indiquée par PC est déplacé dans les 8 bits de poids faible du registre MBR

Ce port 8 bits ne peut faire que des lectures en mémoire, les écritures sont impossibles

- Les registres sont commandés par deux signaux

Validation de la sortie des registres sur le bus B *(MAR n'écrit jamais sur le bus B)*

Ecriture de l'entrée présente sur le bus C dans le registre *(MBR ne lit jamais le bus C)*

Le registre MBR dispose de 2 signaux de commande spécifiques

- Trois signaux de commande supplémentaires autorisent la lecture ou l'écriture en mémoire

Modèle d'exécution de l'IJVM

Les opérations avec la mémoire (suite)

- MAR contient des adresses de mots de 32 bits
- PC contient des adresses de mots de 8 bits

si PC=2 alors c'est l'octet n° 2 qui est transféré dans les 8 bits de poids faible de MBR

Si MAR=2 alors ce sont les 4 octets de 8 à 11 qui sont transférés dans MDR

L'ensemble MAR / MDR commande la lecture / écriture des données de la couche ISA

L'ensemble PC / MBR permet de lire le programme exécutable de la couche ISA

Le cycle complet du chemin des données

- Prendre le contenu d'un ou de 2 registres et les placer sur les entrées A et B de l'UAL via le bus
- Faire se propager les données à travers l'UAL et le décaleur
- Obtenir le résultat sur le bus C
- Ecrire les données présentes sur le bus C dans les registres appropriés
- Si autorisé faire une lecture mémoire aussitôt après le chargement de MAR
- La donnée lue en mémoire n'est disponible dans MDR ou MBR qu'à la fin du cycle suivant. La lecture en mémoire déclanchée à la fin du cycle k n'est valide qu'au cycle k+2 ou plus tard encore!!

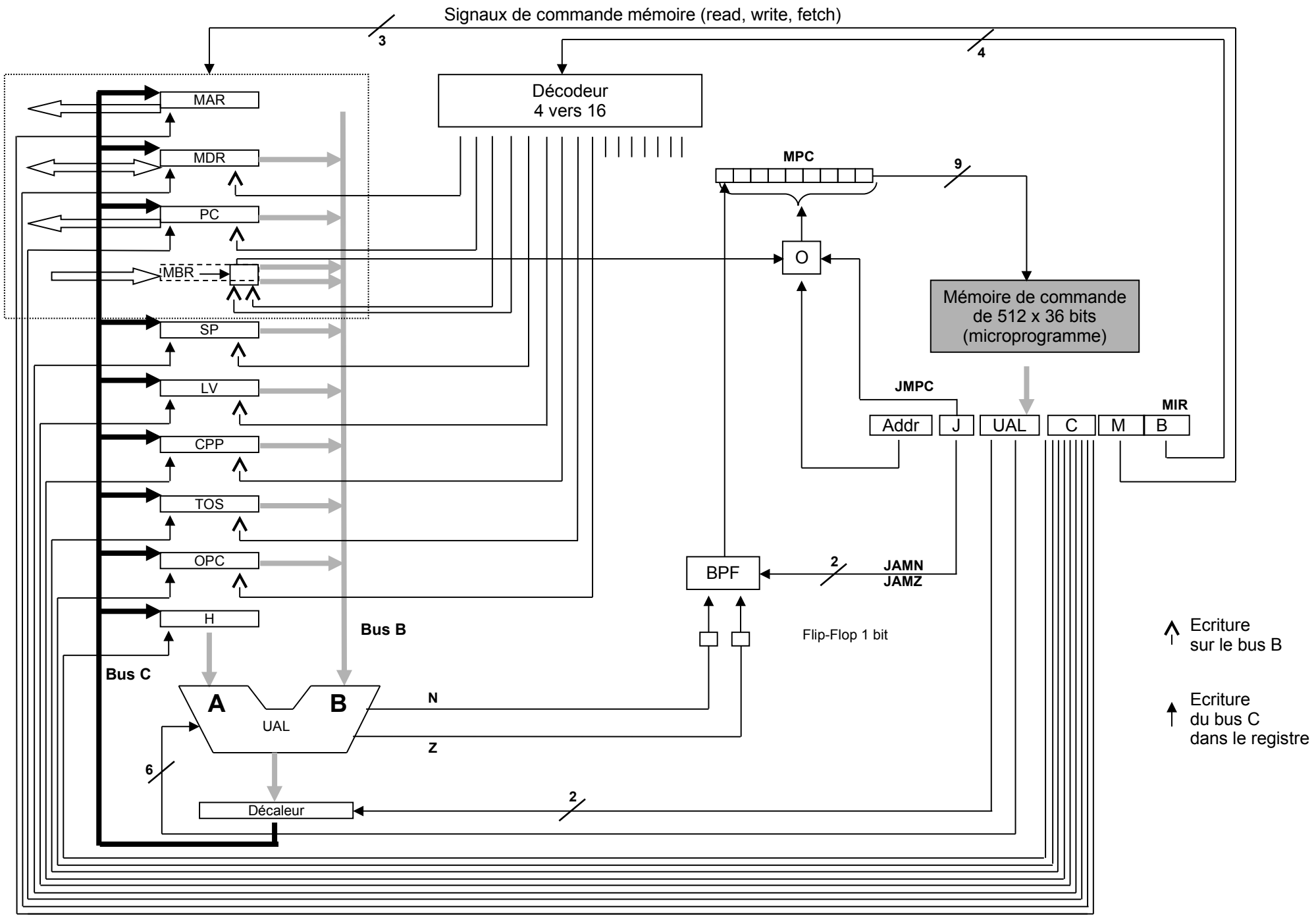
Rôle du micro-programme

- Exécute une séquence de micro-instructions qui réalise une instruction ISA
- La séquence de micro-instructions est mémorisée dans la mémoire de commande ou mémoire de micro-programme
- La mémoire de micro-programme est structurée en mots de 36 bits
- Le séquencement des micro-instructions ne suit pas l'ordre des micro-instructions dans la mémoire. Chaque micro-instruction spécifie la prochaine micro-instruction à exécuter (champ *adresse*).

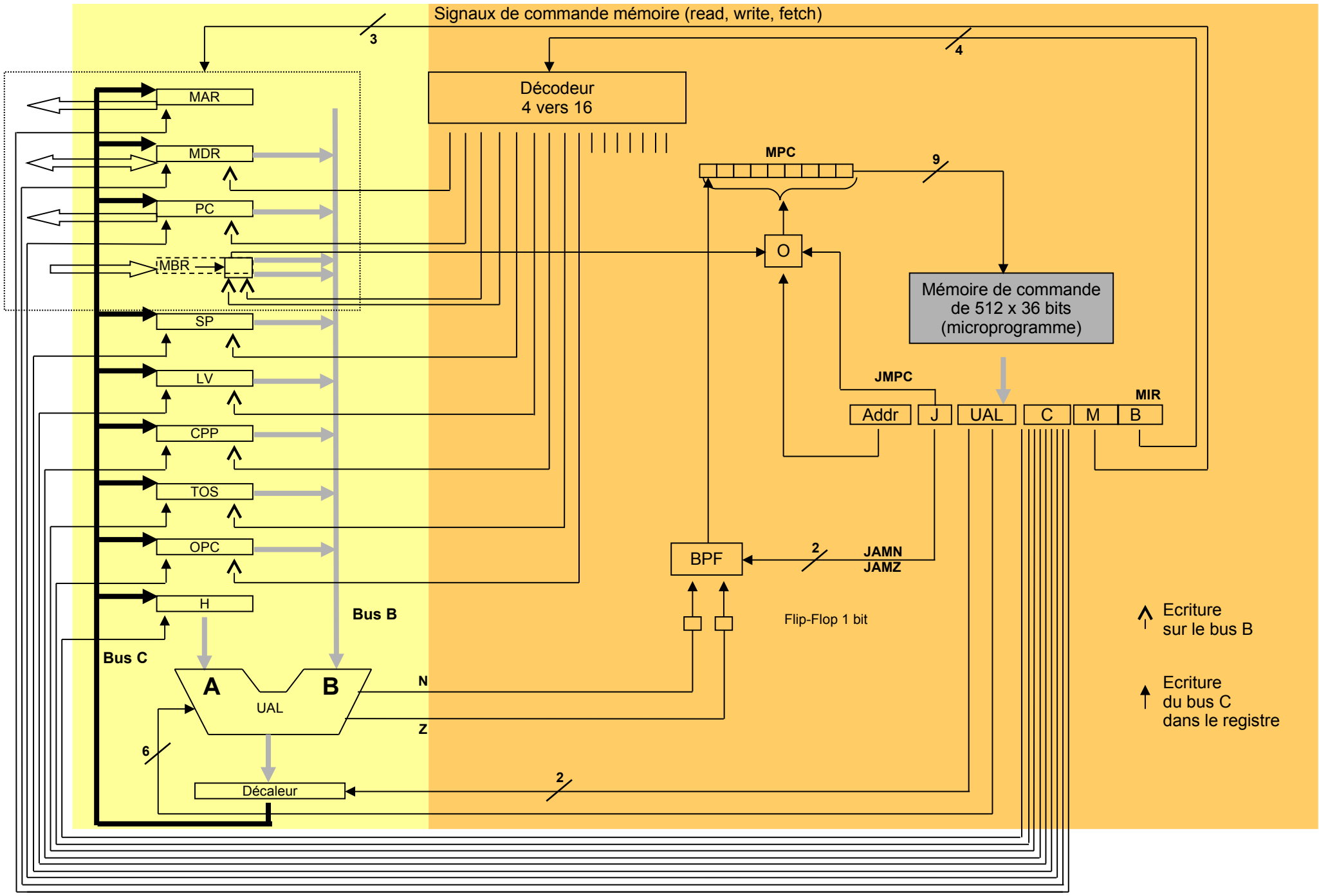
La mémoire de micro-programme

- C'est une mémoire en lecture seule
- *MPC (MicroProgram Counter)* est le registre d'adresse de la mémoire de commande
- *MIR (MicroInstruction Register)* contient la micro-instruction courante

La micro-architecture MIC1

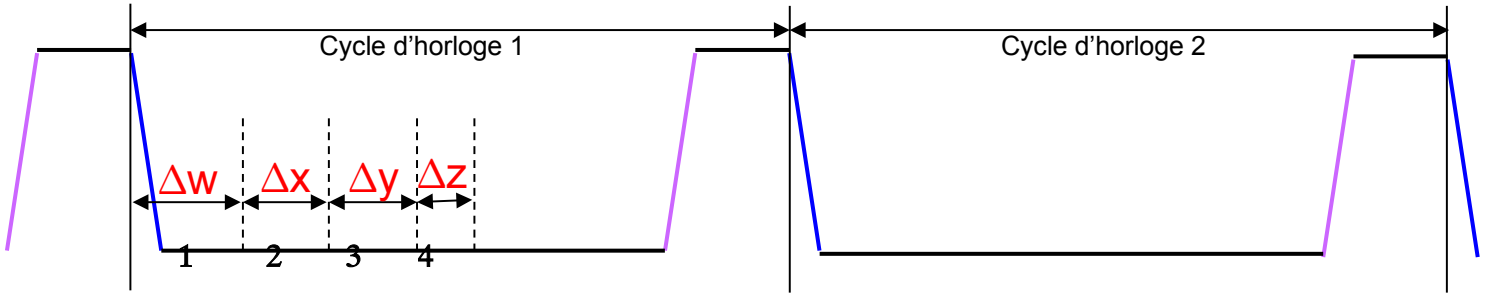


La micro-architecture MIC1



Modèle d'exécution de l'IJVM

Chronogramme du chemin des données



Une impulsion brève débute chaque cycle d'horloge qui se décompose en 4 sous-cycles

1)Le front descendant de l'impulsion active les signaux de commande: ils sont actifs après Δw

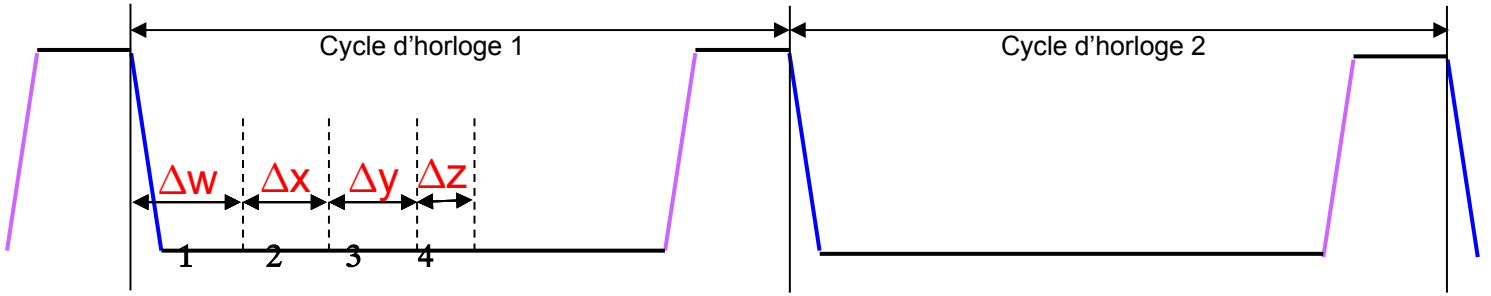
2) Le registre d'entrée est sélectionné et connecté au bus B.Le registre H est sélectionné (éventuellement) et connecté à l'entrée A de l'UAL. Les entrées de l'UAL sont stables au bout d'un temps Δx

3)L'opération est ensuite réalisée par l'UAL et le décaleur Δy

4)Le résultat est disponible sur le bus C au bout d'un temps Δz

Modèle d'exécution de l'IJVM

Chronogramme du chemin des données



- La donnée disponible sur le bus C peut être chargée dans les registres à la transition montante de la prochaine impulsion
- Le registre d'entrée est déconnecté au bus B sur la transition montante de la prochaine impulsion
- Le chargement des registres à partir de la mémoire se fait également sur la transition montante de l'impulsion

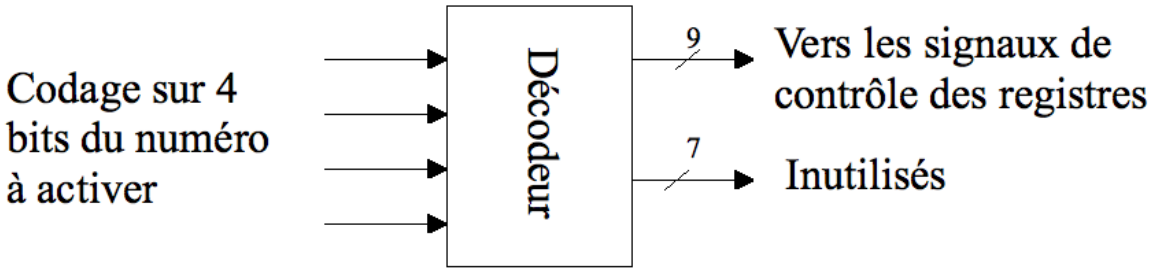
Les sous-cycles sont implicites ils ne représentent qu'un enchaînement d'actions et ne sont pas cadencés par une horloge. La difficulté est d'assurer que le temps global du cycle d'horloge soit supérieur à

$$\Delta w + \Delta x + \Delta y + \Delta z$$

La micro-instruction

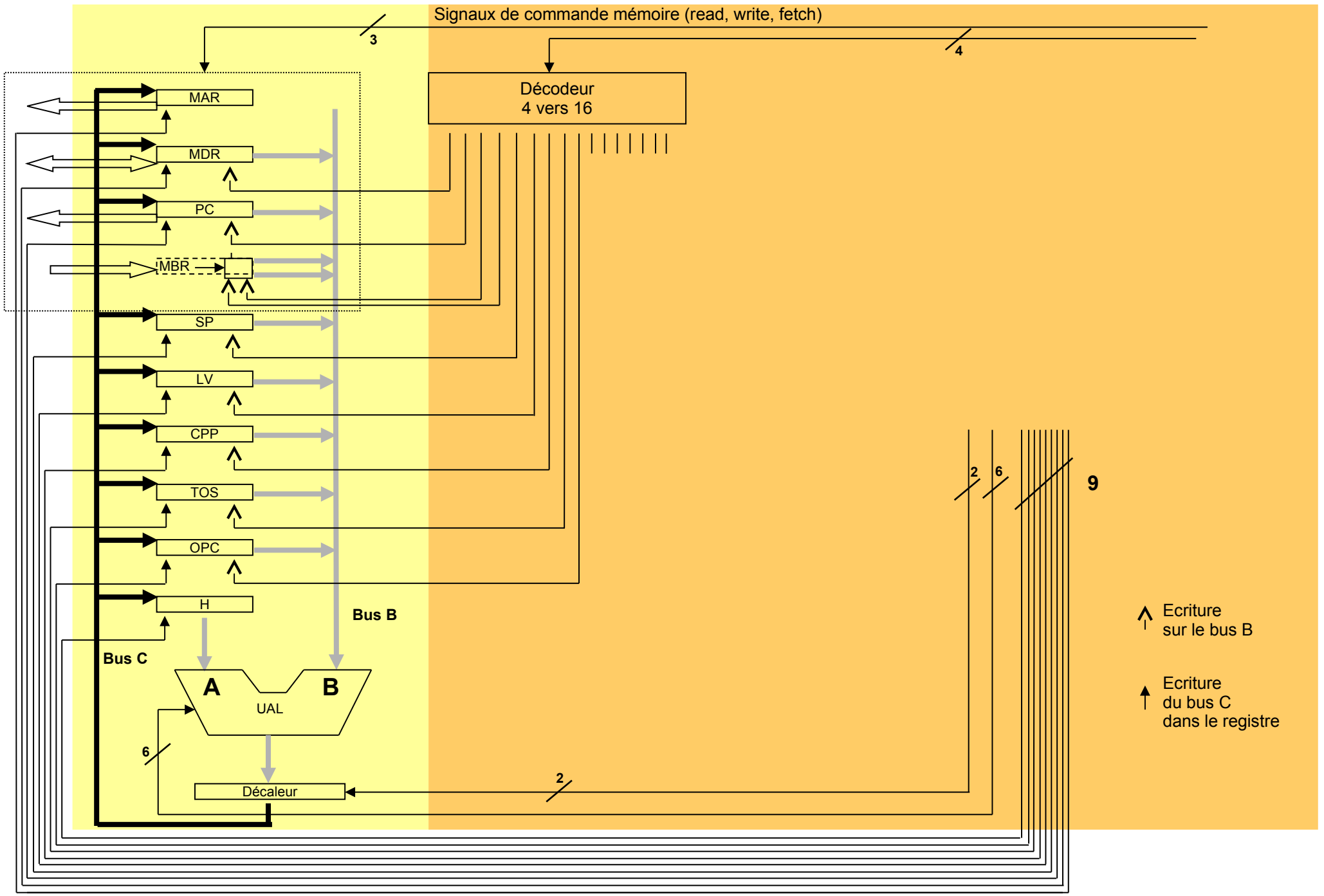
La commande du chemin des données

- Il y a 29 signaux de commande
  - 9 signaux pour commander l'écriture à partir de C dans l'un des 9 registres
  - 9 signaux pour commander la copie d'un des 9 registres sur le bus B
  - 8 signaux pour commander l'UAL et le décaleur : 6 + 2
  - 2 signaux pour autoriser la lecture ou l'écriture en mémoire
  - 1 signal pour commander la recherche de la prochaine instruction (fetch)
- Il faut les activer de façon à réaliser l'instruction souhaitée
  - Comme un seul des 9 registres peut être écrit sur B à un instant, 4 bits suffisent pour coder le registre concerné en utilisant un décodeur 1 parmi 16 dont 7 possibilités ne sont pas utilisées. Il suffit de 4 bits au lieu de 9 ! Donc 24 signaux de commande suffisent!



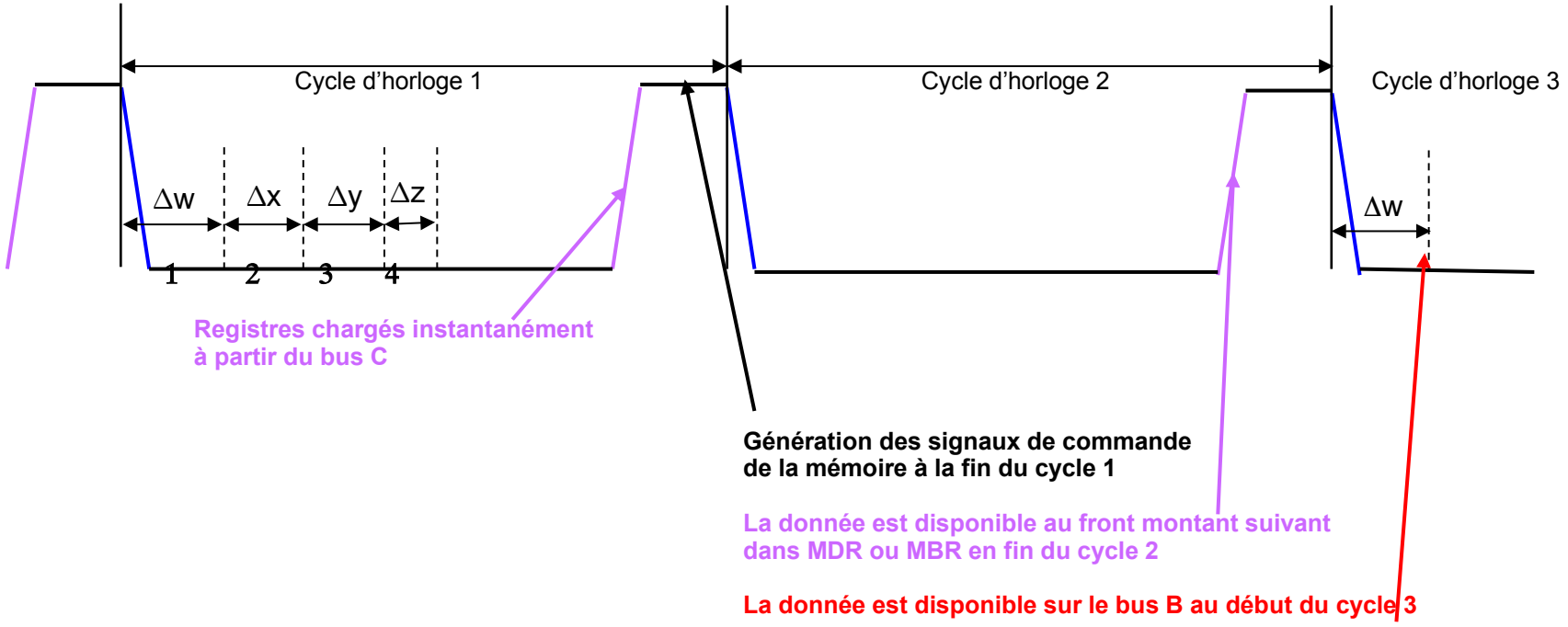
On a alors 29-9+4=24 signaux pour contrôler le chemin de

La micro-architecture MIC1



La micro-instruction

Le cycle complet du chemin des données (suite)



- si le temps d'accès à la mémoire est de l'ordre d'un cycle d'horloge alors la donnée demandée à la fin du cycle k n'est disponible sur le bus B qu'au début du cycle k+2
- si une lecture mémoire n'est pas terminée au front montant de l'horloge les registres MDR ou MBR contiennent encore la donnée précédente: attention aux erreurs si on n'est pas certain du contenu!

La micro-instruction

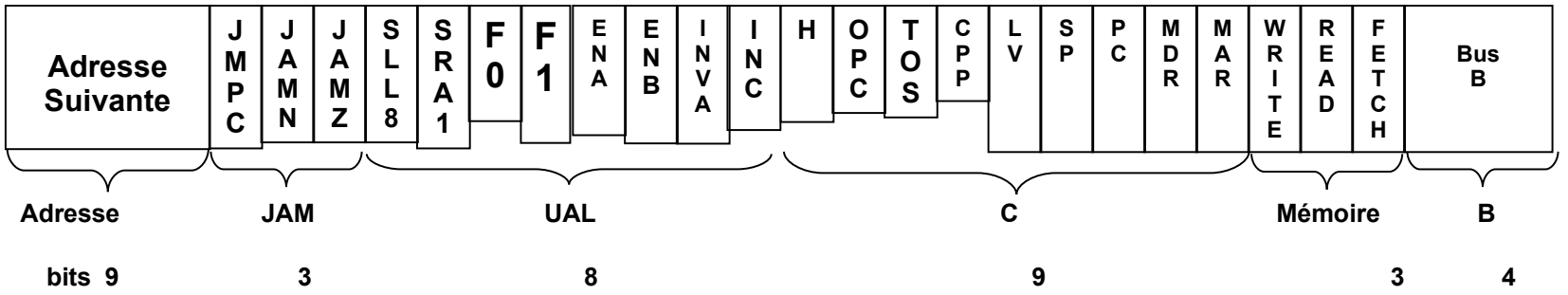
Format de la micro-instruction

- **Elle doit commander le chemin des données au cycle courant**
  - Il faut 24 bits que l'on peut regrouper sur 4 champs distincts

  - ***UAL*** définit l'opération à réaliser par l'UAL et le décaleur 8 bits
  - ***C*** définit le ou les registres chargés avec la donnée du bus C 9 bits
  - ***Mem*** définit des fonctions mémoire 3 bits
  - ***B*** définit le registre à vider sur B 4 bits

- **Elle doit déterminer ce qui doit être fait au cycle suivant**
  - ***Adresse*** contient l'adresse de la micro-instruction suivante 9 bits
  - ***JAM*** définit le mode de sélection de la micro-instruction suivante 3 bits

- **Il faut 36 bits pour coder la micro-instruction**



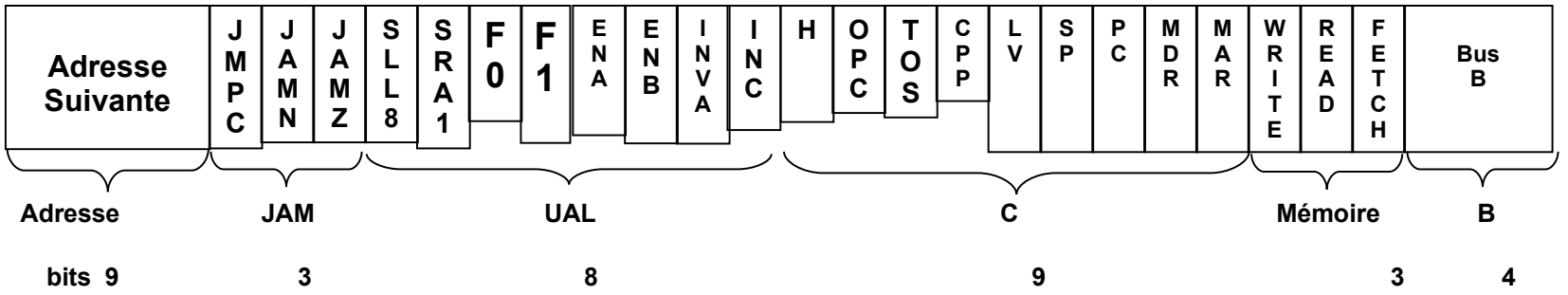
Registres sur B

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 rien

La micro-instruction

********Generated MIC-1 Binary Instructions**

000000010000011010100000010000000001
000000011000011010100000010000010001
001100001000000000000000000000001111
000101110000011011101000000000010001
000000100000000101000000010010100101
000000101000000000000000000000001111
000000110000000101000000010010100000
000000111000001101010000000010000101



Registres sur B

0 = MDR	**5 = LV**
1 = PC	**6 = CPP**
2 = MBR	**7 = TOS**
3 = MBRU	**8 = OPC**
4 = SP	**9-15 rien**

La micro-instruction

******Generated MIC-1 Binary Instructions**

000000010 000 00110101 000000100 000 0001

saut à 2/pas de jump/B+1/ vers PC/ pas d'acces memoire / depuis PC

PC=PC+1

000000011 000 00110101 000000100 001 0001

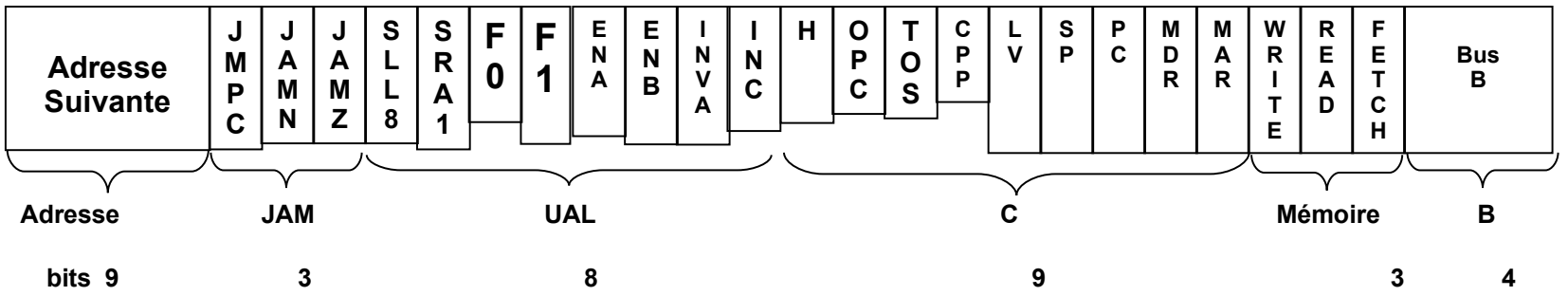
saut à 3/pas de jump/B+1/ vers PC/ Fetch / depuis PC

PC=PC+1;Fetch

000001110 000 00111100 000000001 010 0110

saut à 14/pas de jump/A+B/ vers MAR/ Rd / depuis CPP

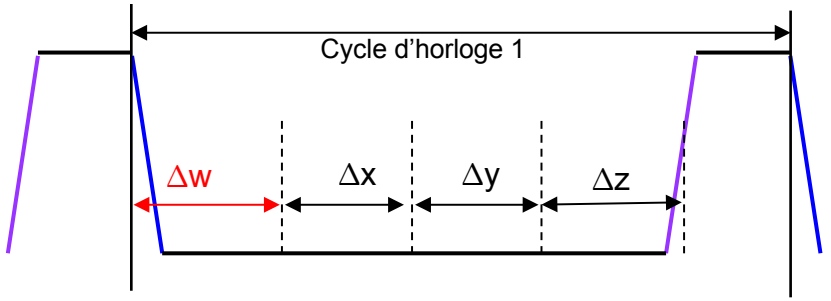
MAR=CPP+H;rd



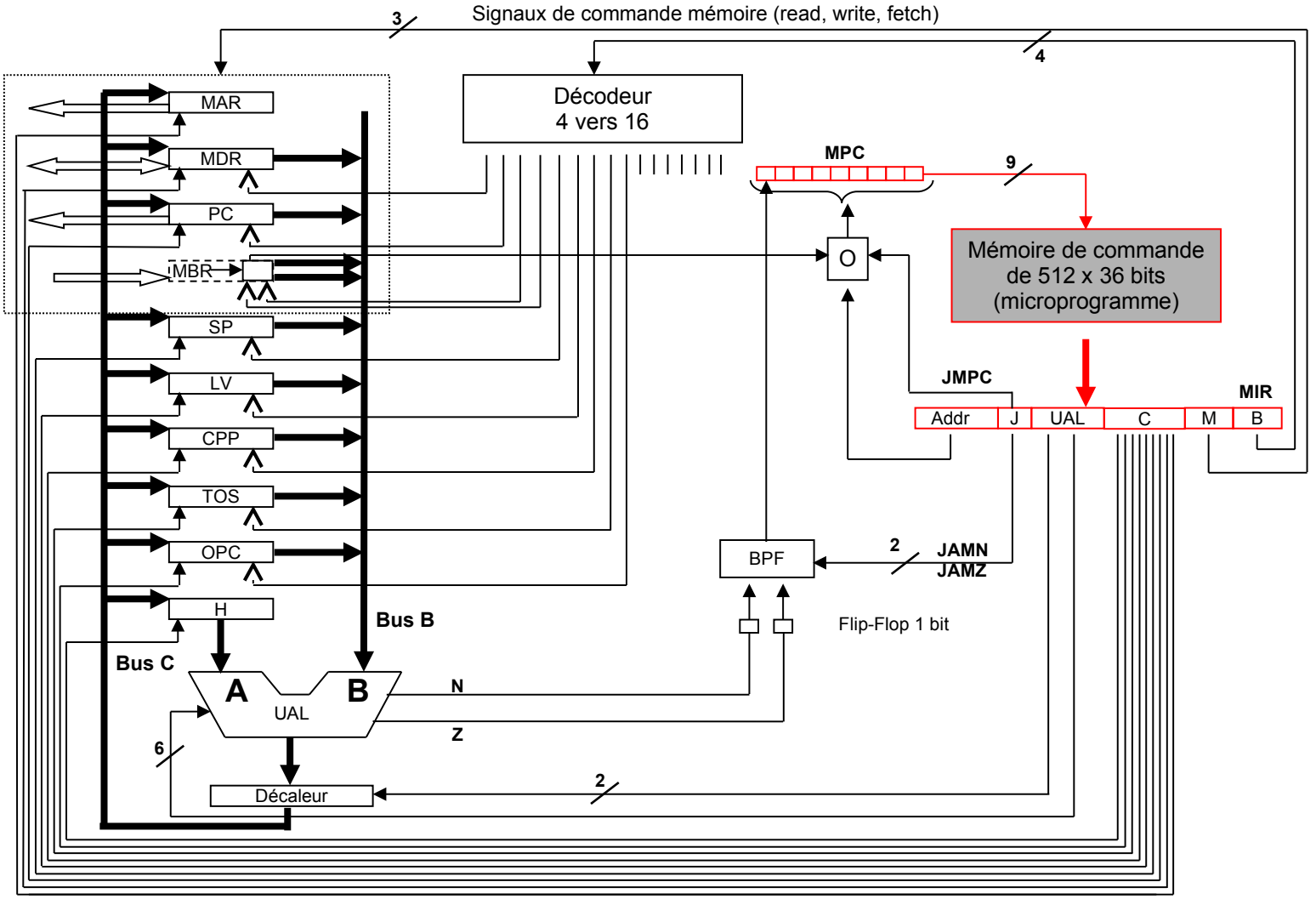
Registres sur B

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 rien

Fonctionnement de la micromachine



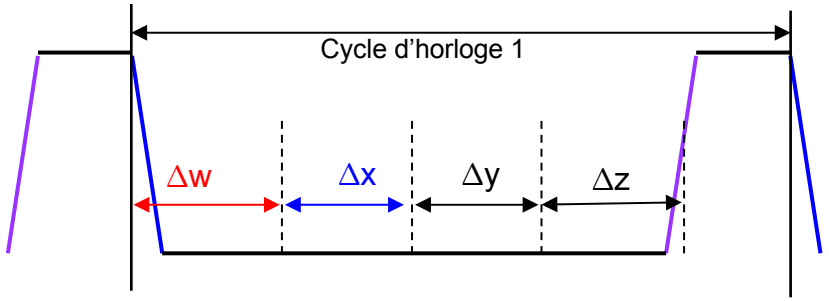
Sous-cycle 1
- MIR est chargé avec la micro instruction pointée par MPC



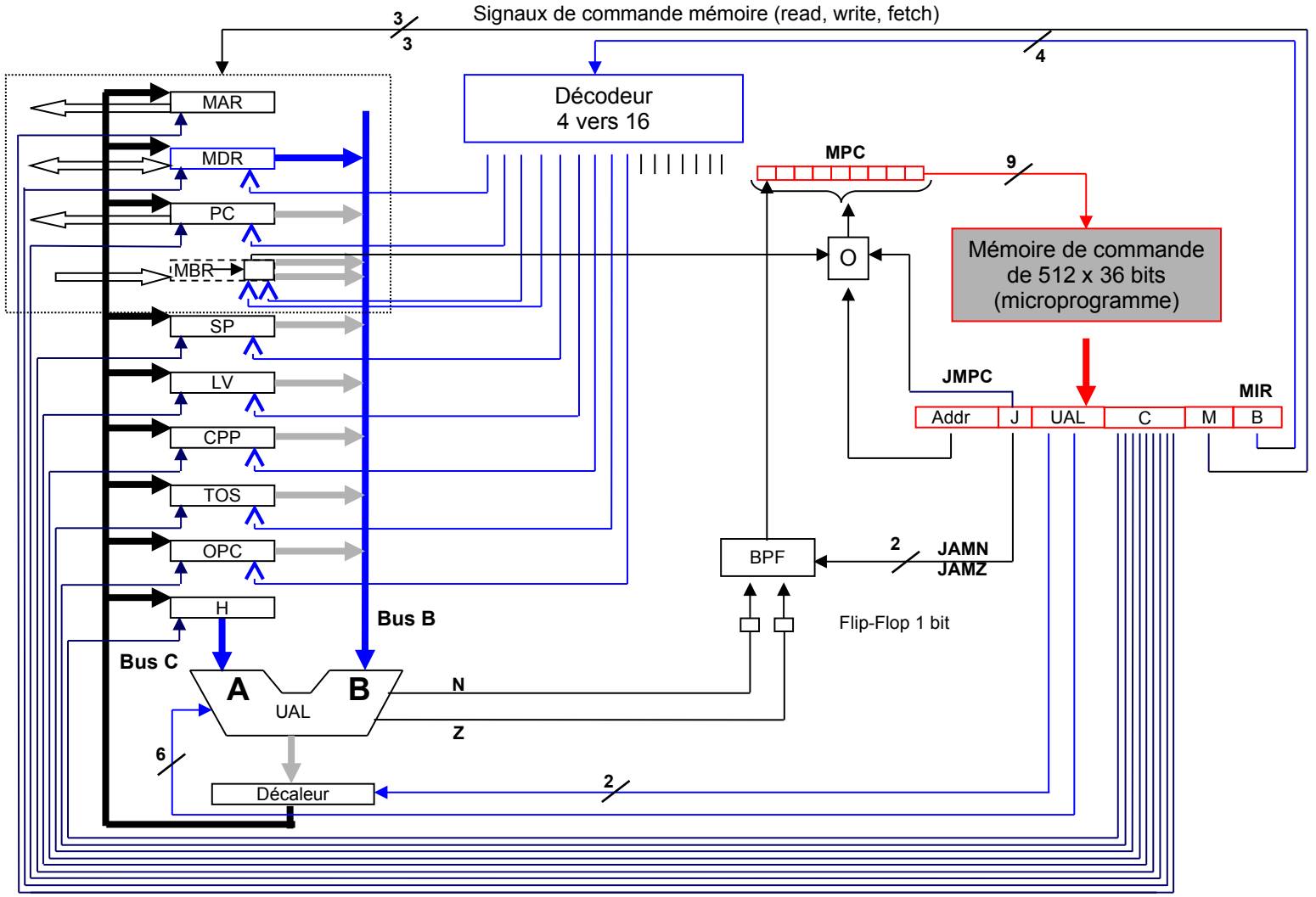
Ecriture sur le bus B

Ecriture du bus C dans le registre

Fonctionnement de la micromachine



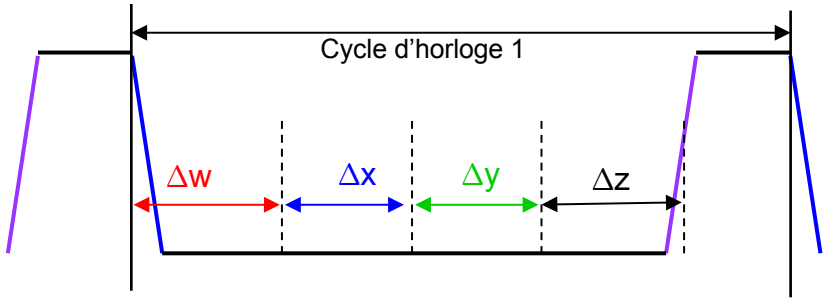
Sous-cycle 1
Sous cycle 2
Les divers signaux de commande se propagent
- Un registre est vidé sur le bus B
- les entrée de l'UAL sont stables
- la sélection de l'opération de l'UAL est réalisée
- le registre où sera sauvegardé le résultat est sélectionné



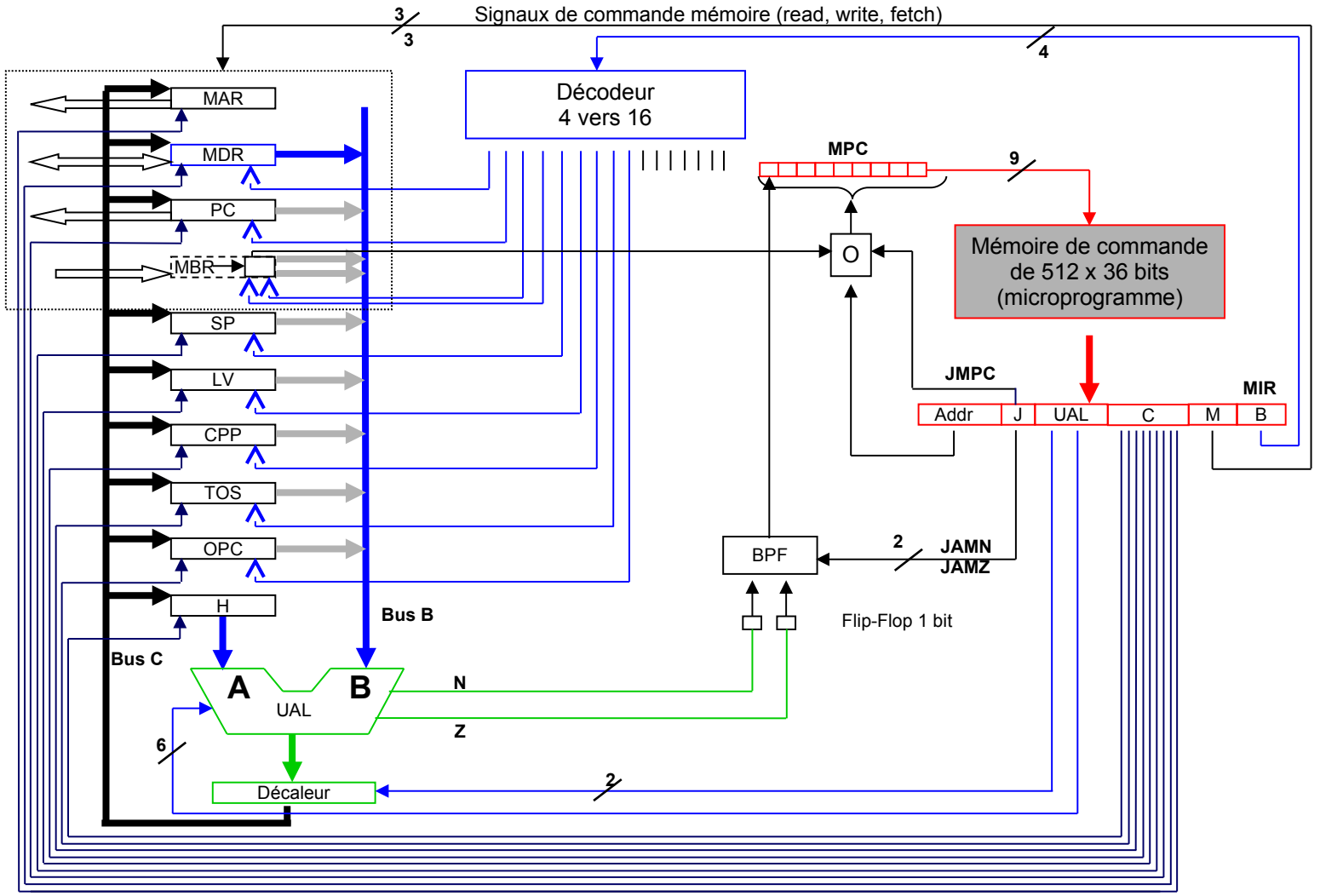
Ecriture
sur le bus B

Ecriture
du bus C
dans le registre

Fonctionnement de la micromachine



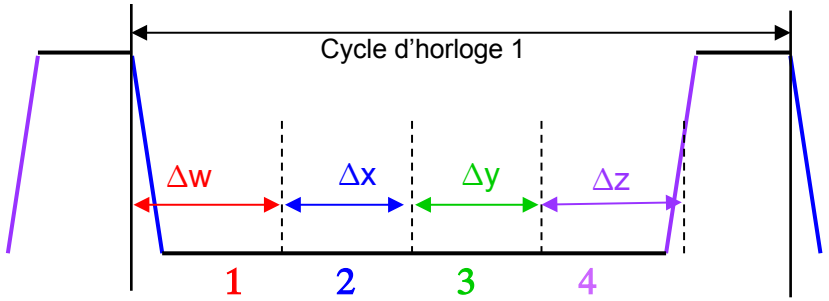
Sous-cycle 1
Sous cycle 2
Sous-cycle 3
- L'opération est réalisée par l'UAL
- Le résultat est disponible à la sortie du décaleur
- Les bits N et Z sont positionnés



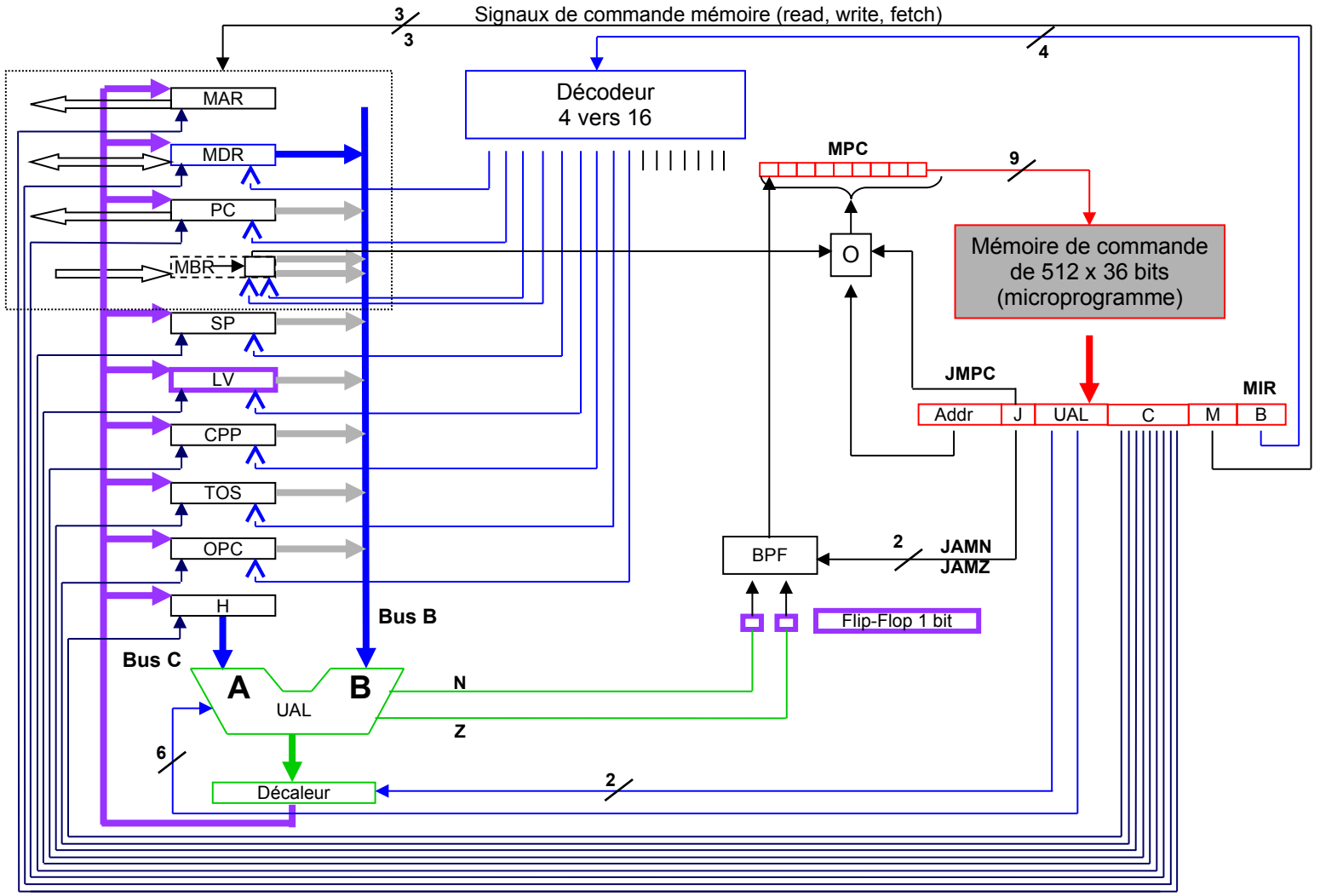
Ecriture sur le bus B

Ecriture du bus C dans le registre

Fonctionnement de la micromachine



Sous-cycle 1
Sous cycle 2
Sous-cycle 3
Sous-cycle 4
- la sortie du décaleur se propage sur le bus C
- la donnée sur C est chargée sur le front montant
- les flip-flop sont activés sur le front montant



Ecriture
sur le bus B

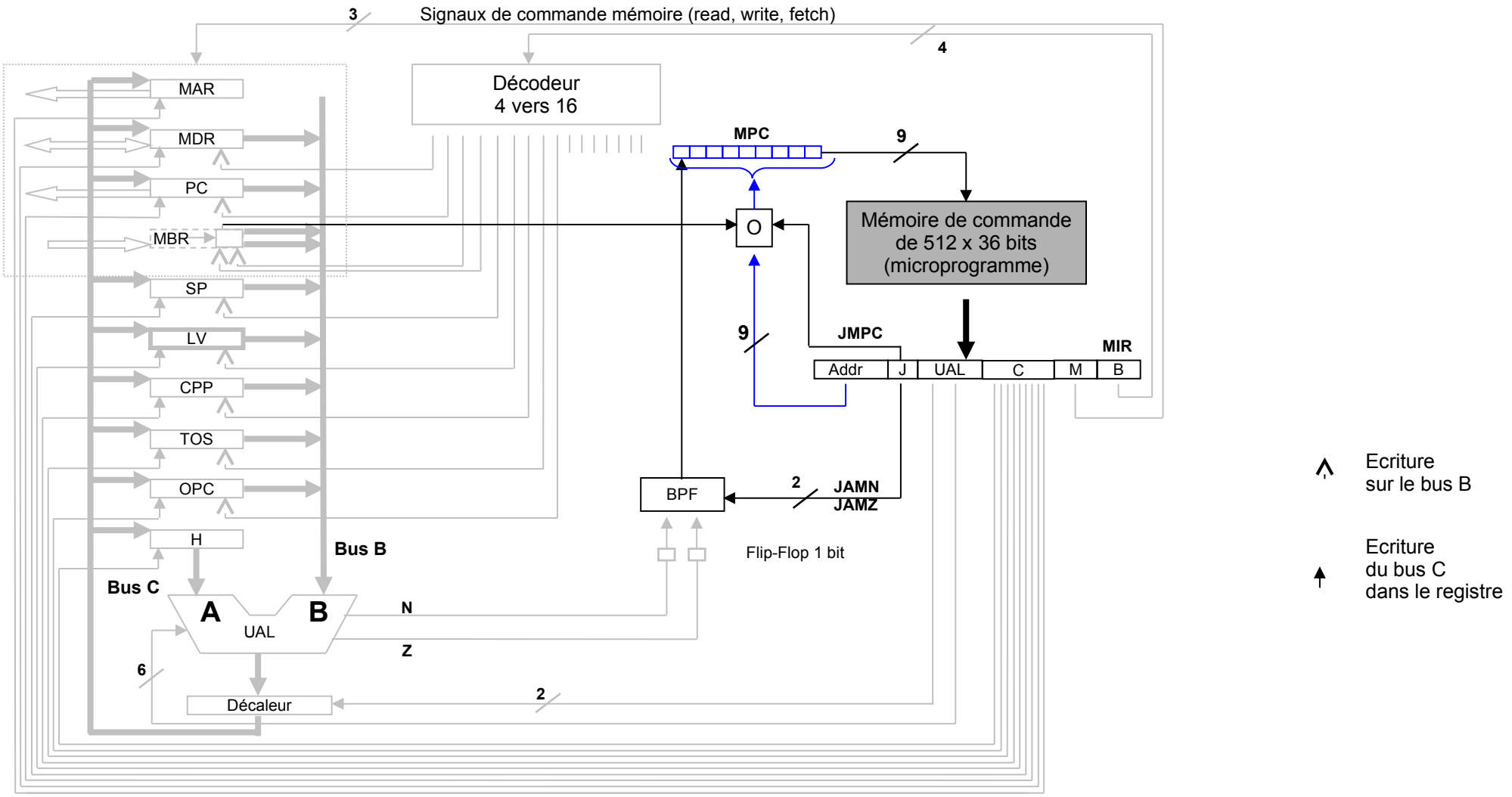
Ecriture
du bus C
dans le registre

Séquencement des micro instructions

Mise à jour du MPC (micro program counter)

A la fin du sous cycle 1 MIR est chargé. Le calcul de l'adresse de la prochaine micro-instruction peut débuter

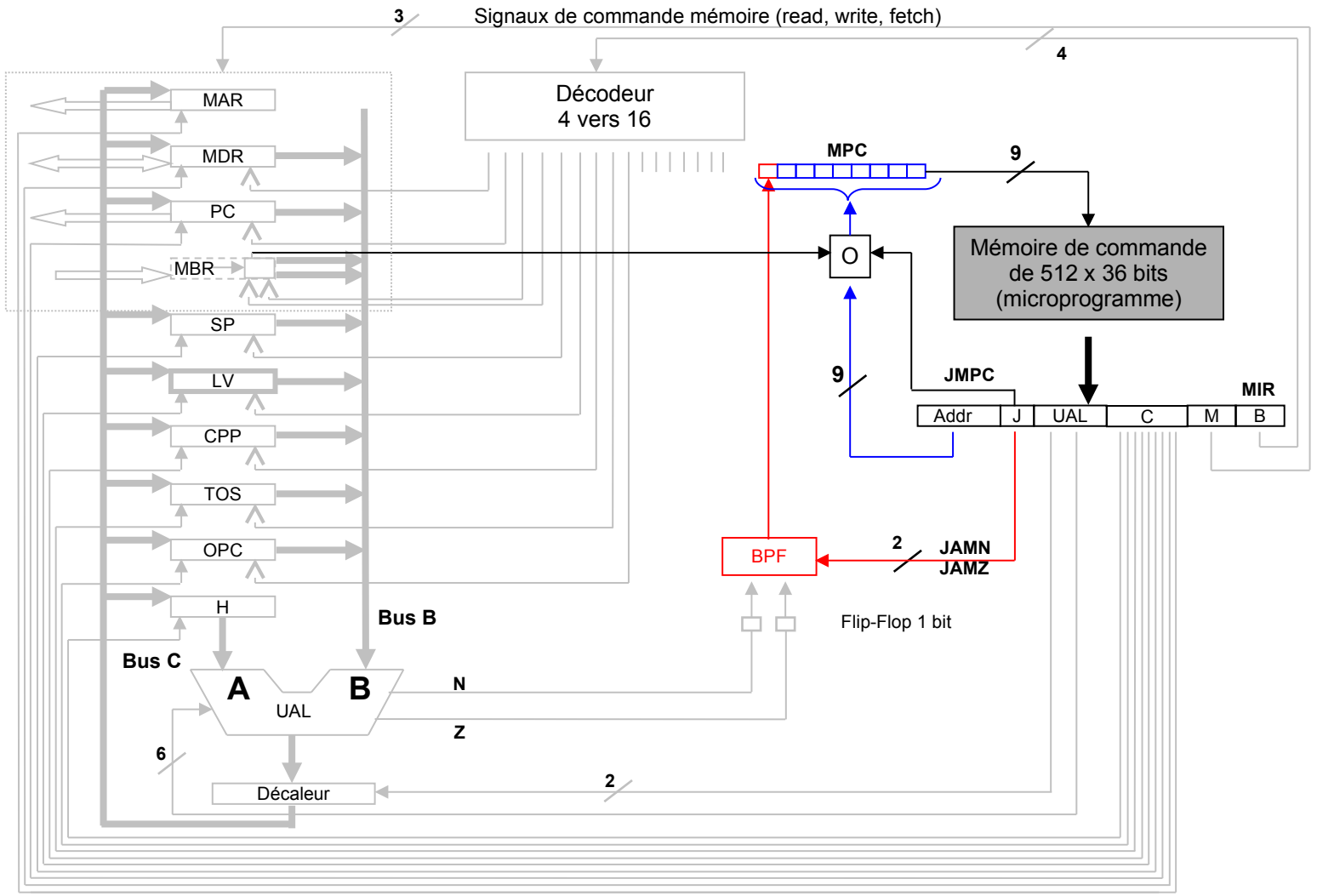
1- copie des 9 bits du champ addr de MIR dans MPC



Séquencement des micro instructions

Mise à jour du MPC (micro program counter)

2- **si** un ou plusieurs des bits du JAM sont à 1 **alors**
BPF = (*JAMN* ET *N*) OU (*JAMZ* ET *Z*) = Bit de Poids Fort de MPC



Ecriture
sur le bus B

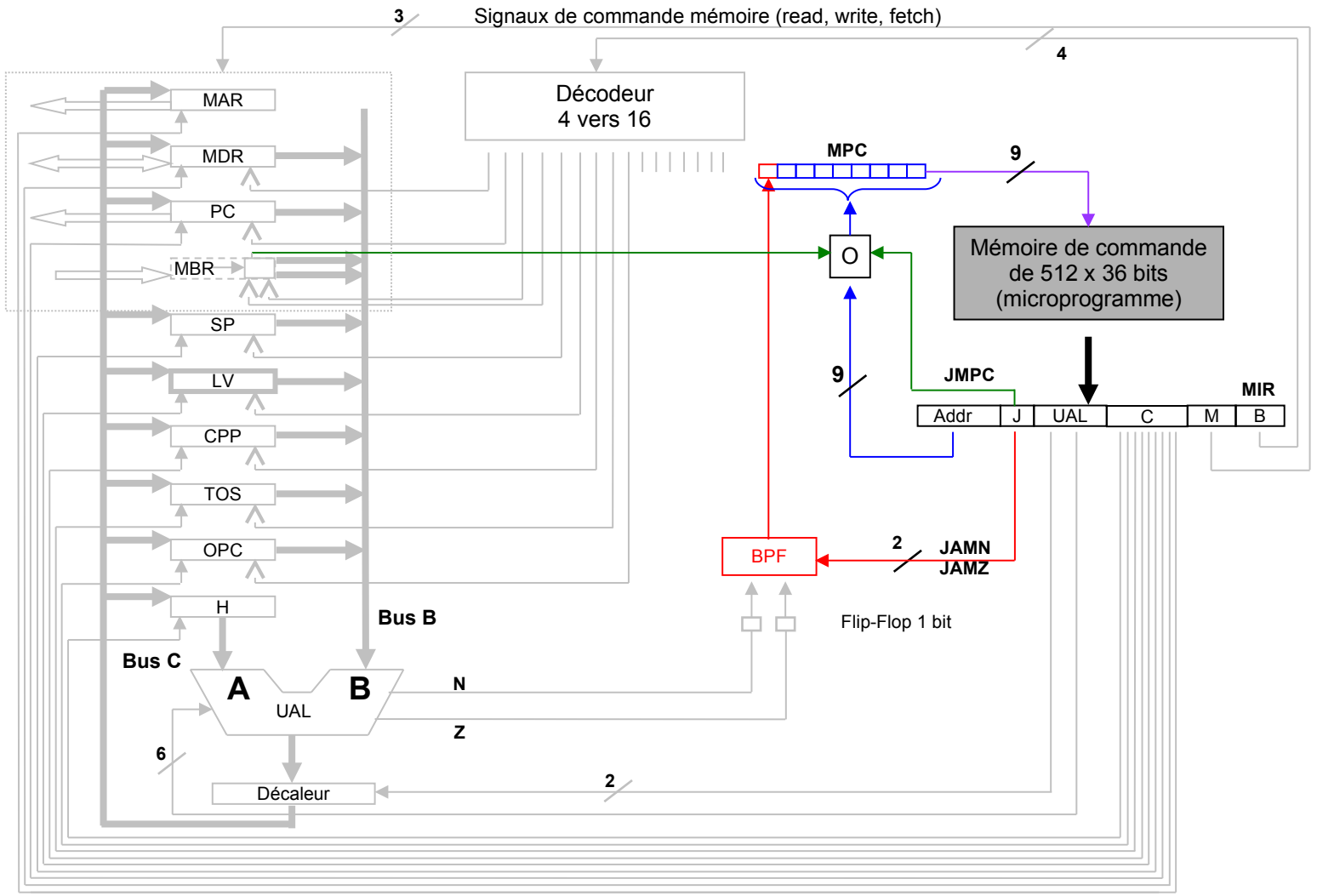
Ecriture
du bus C
dans le registre

Séquencement des micro instructions

Mise à jour du MPC (micro program counter)

3- MPC permet d'adresser la prochaine micro-instruction. Elle se trouve soit à l' *Adresse suivante* soit à l'*Adresse suivante* avec BPF *à 1*

Il y a donc deux instructions suivantes possibles selon le résultat du test qui détermine BPF



Ecriture
sur le bus B

Ecriture
du bus C
dans le registre

Séquencement des micro instructions

Mise à jour du MPC (micro program counter)

4- **si** JMPC=1 **alors**

MPC = Addr OU MBR (8 bits de poids faible)

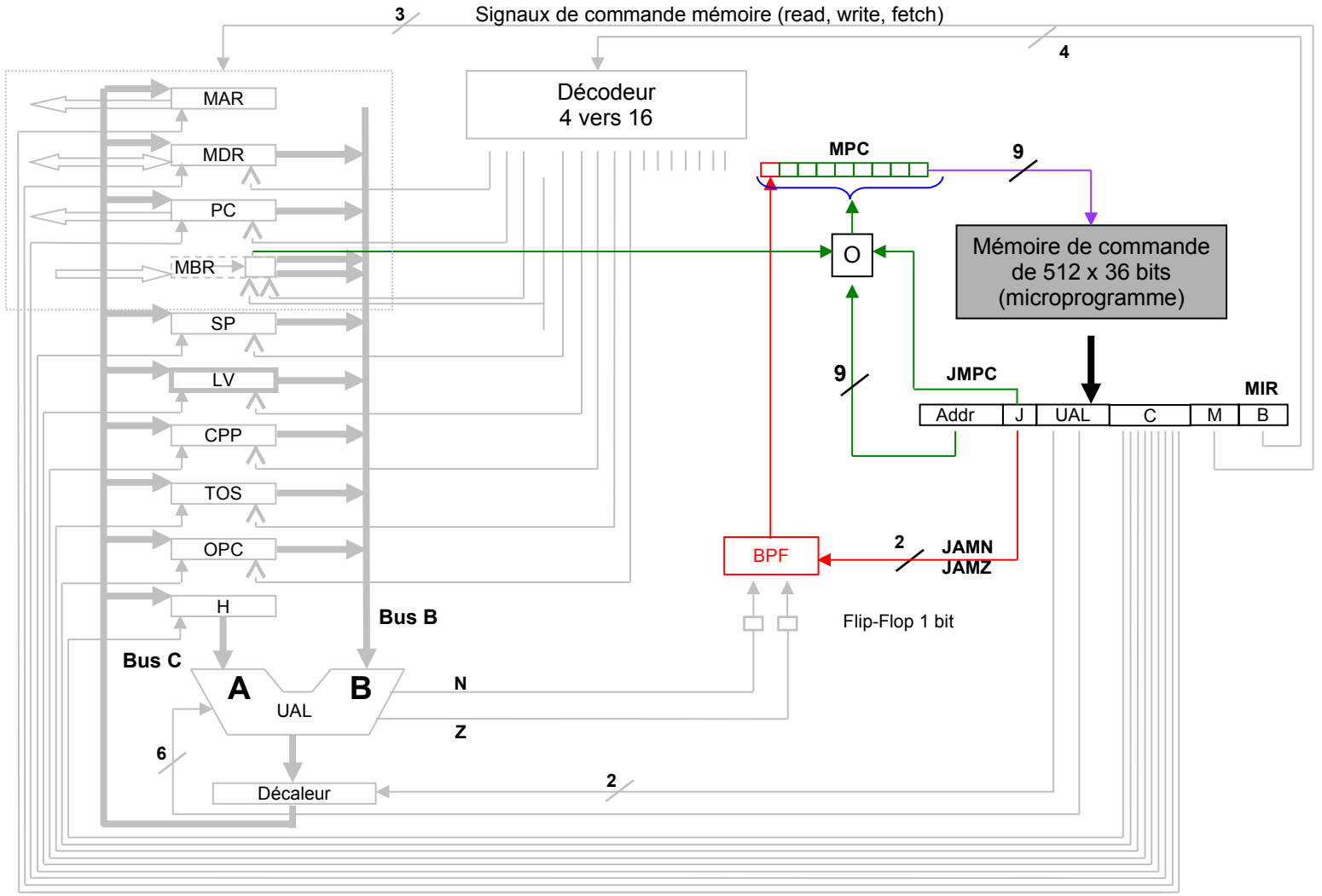
MPC (bit de poids fort) = Addr (bit de poids fort) OU (JAMN ET N) OU (JAMZ ET Z)

Lorsque JMPC=1 si Addr (8 bits de poids faible)=0x00 et JAMN=0 et JAMZ =0 alors l'adresse de l'instruction suivante vaut soit 0x100+MBR soit 0x000+MBR (utile si deux versions d'une même instruction existent

si Addr ≠ 0x00 alors branchement dépendant d'un état du microprogramme et de MBR

si JAMN ET JAMZ =1 on peut adresser 512 adresses différentes

si JAMN=0 ET JAMZ =0 on peut adresser 256 adresses différentes



Ecriture sur le bus B

Ecriture du bus C dans le registre

Implémentation du microprogramme

- Il s'agit de définir les micro-instructions qui réalisent le microprogramme de chaque instructions de la macro-archirecture
- On définit un langage de micro-instructions plus facile à manipuler que des code binaires sur 36 bits : le langage de micro-assemblage

- Exemple
  - SP = MDR copie de MDR dans SP
  - MDR = H + SP addition de H et de SP placée dans MDR

- Il faut veiller à ce que chaque micro-instruction du langage de micro-assemblage soit effectivement réalisable en un cycle d'horloge

  - MDR = SP + MDR impossible en un cycle car une addition fait intervenir H
  - H = H – MDR ne peut pas être réalisée par la microarchitecture

Le langage de micro-assemblage

MAL : *Micro Assembly Language*

- DEST désigne n'importe quel registre pouvant lire C
- SOURCE désigne n'importe quel registre pouvant écrire sur B

- Liste des opérations permises par MAL

$$DEST = H$$
$$DEST = SOURCE$$
$$DEST = \overline{H}$$
$$DEST = \overline{SOURCE}$$
$$DEST = H + SOURCE$$
$$DEST = H + SOURCE + 1$$
$$DEST = H + 1$$
$$DEST = SOURCE + 1$$

$$DEST = SOURCE - H$$
$$DEST = SOURCE - 1$$
$$DEST = -H$$
$$DEST = H \ AND \ SOURCE$$
$$DEST = H \ OR \ SOURCE$$
$$DEST = 0$$
$$DEST = 1$$
$$DEST = -1$$

- Opérations d'accès à la mémoire
  - Lecture et écriture de mots de 4 octets par les registre MAR et MDR
    rd et wr
  - Lecture d'un code opération d'un octet en mémoire par les registres PC et MBR
    fetch
  - Les deux types d'opérations peuvent intervenir en parallèle

Le langage de micro-assemblage
MAL : *Micro Assembly Language*

- Déroulement concurrent de micro-instructions

  MAR = SP; rd

  Le ; désigne un déroulement concurrent (en parallèle).

  Cela suppose que le matériel soit capable de réaliser effectivement ces micro-instructions en parallèle

  Exemple d'enchaînement impossible

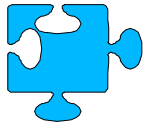
  1. MAR = SP; rd
  2. MDR = H

  La première micro-instruction demande un accès à la mémoire. Il sera en principe satisfait lors du cycle suivant et le registre MDR devrait être affecté par la valeur lue en mémoire

  Mais la seconde micro-instruction charge MDR avec H au cours du même cycle qui affectera MDR avec la donnée lue en mémoire.

  Il se produit donc un conflit de chargement du registre MDR au cours dans la seconde micro-instruction

Le langage de micro-assemblage

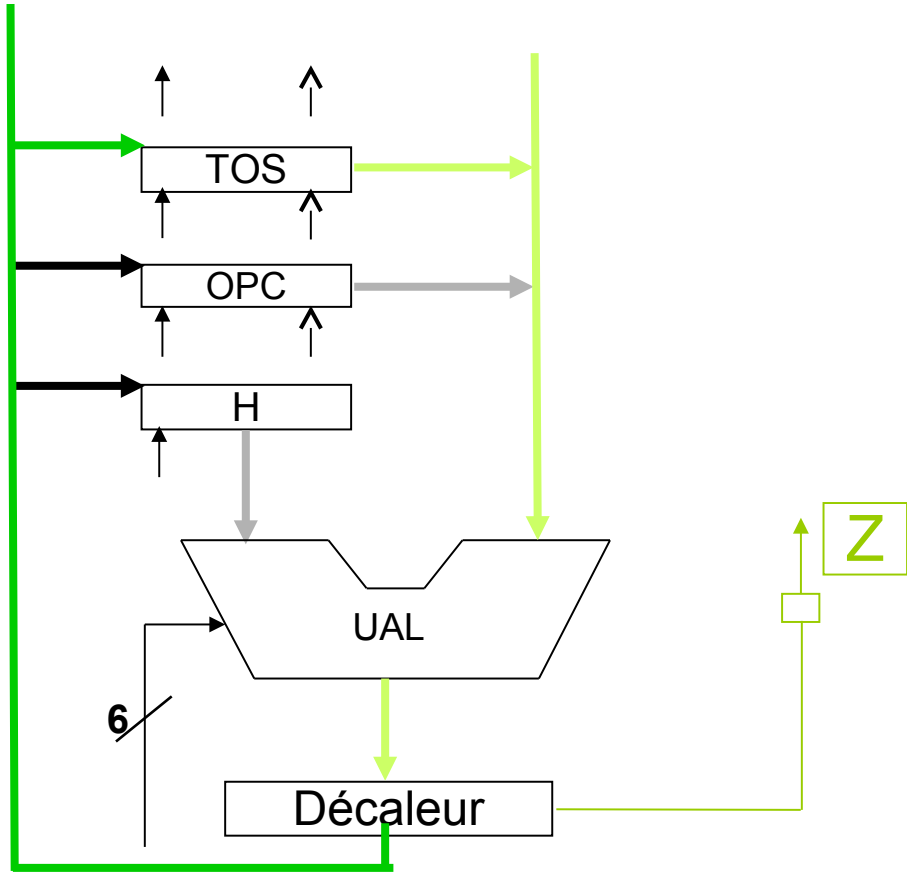


- Branchement inconditionnel

  goto label peut être incluse dans n'importe quelle micro-instruction pour désigner explicitement la micro-instruction suivante

- Compare la valeur d'un registre à zéro

  Z = TOS en faisant passer le registre dans l'UAL le bit Z est positionné si le contenu du registre est nul le bit Z est mémorisé grâce au Flip-Flop (il n'est pas mémorisé dans un registre)



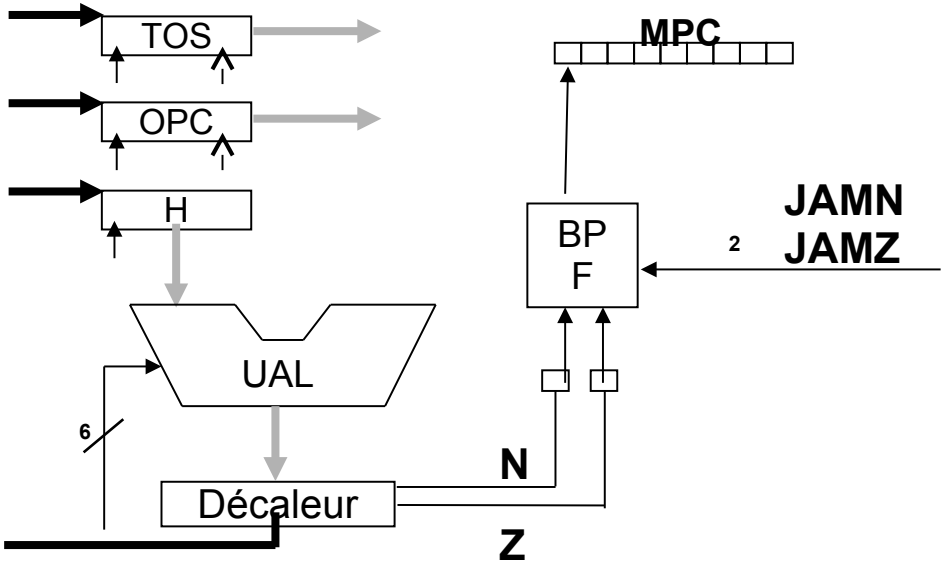
Le langage de micro-assemblage

- Branchement conditionnel

  if (Z) goto L1; else goto L2

  if (N) goto L1; else goto L2

  par construction, les deux adresses possibles L1 & L2 ne diffèrent que par leur bit de poids fort – elles doivent être distantes de 256 octets exactement dans la mémoire de microprogramme



- Comparaison + Branchement

  peut être réalisé en une seule micro-instruction

  Z=TOS; if (Z) goto L1; else goto L2

Le langage de micro-assemblage

- Branchement à une micro-instruction spécifiée par MBR

  goto (MBR OR *addr*)

  lorsque le bit JMPC est positionné

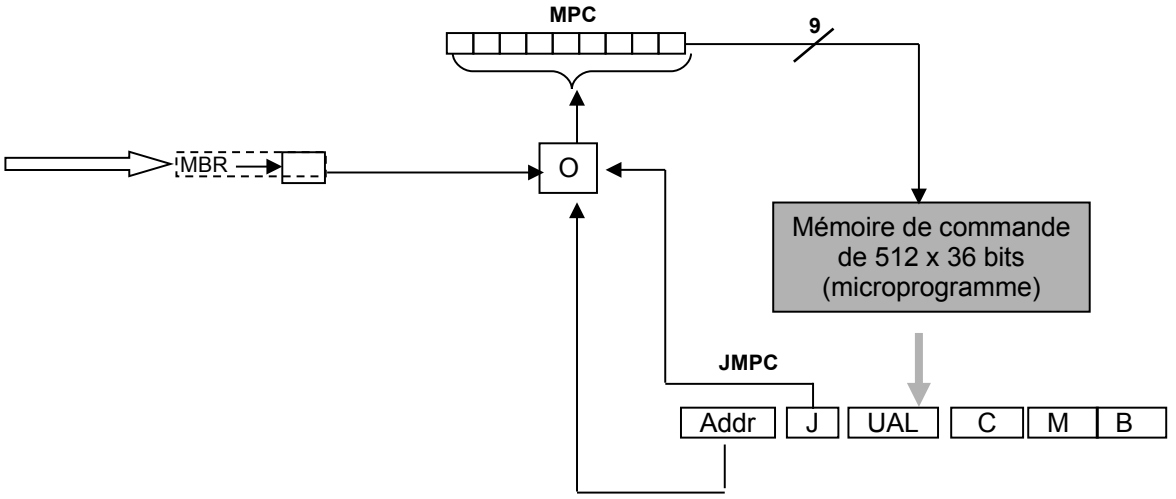
  le registre MPC est chargé avec le contenu de MBR OU *addr*

  Ce mécanisme est utilisé par exemple pour gérer les instructions *wide*

  si *addr* est à 0x00 alors MPC est chargé avec le contenu de MBR

  dans ce cas la micro instruction est équivalente à

  goto MBR



Implémentation du microprogramme

- 112 micro-instructions suffisent à définir les instructions de l'IJVM sur l'architecture Mic-1

- Utilisation des registres
  - CPP référence le pool de constantes
  - LV référence le bloc de variables locales (Local Variables)
  - SP référence le haut de la pile (Stack Pointer)

  - PC contient l'adresse du prochain octet du flux d'instructions (Program Counter)
  - MBR reçoit le flux des instructions par octets provenant de la mémoire

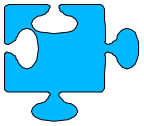
  - TOS et SP sont des registres complémentaires
    - TOS (Top Of Stack) contient le contenu de la mémoire référencée par SP (haut de la pile) pour un gain de temps

  - OPC est un registre temporaire

```
Main1     PC = PC + 1; fetch; goto (MBR)          (MBR contient code d'opération)extrait octet suivant; branchement

nop1      goto Main1                              Ne fait rien
iadd1     MAR = SP = SP – 1; rd                   Lit le 2ème mot de la pile
iadd2     H = TOS                                 H = sommet de pile
iadd3     MDR = TOS = MDR + H; wr; goto Main1     Additionne les 2 mots,actualise TOS ;écrit nouveau sommet

isub1     MAR = SP = SP – 1; rd                   Lit 2ème mot de la pile
isub2     H = TOS                                 H =sommet de pile
isub3     MDR = TOS = MDR – H; wr; goto Main1     Soustrait, actualise TOS ;écrit nouveau sommet

iand1     MAR = SP = SP – 1; rd                   Lit 2ème mot de la pile
iand2     H = TOS                                 H =sommet de pile
iand3     MDR = TOS = MDR AND H; wr; goto Main1   Effectue AND, actualise TOS ;écrit nouveau sommet

ior1      MAR = SP = SP – 1; rd                   Lit 2ème mot de la pile
ior2      H = TOS                                 H =sommet de pile
ior3      MDR = TOS = MDR OR H; wr; goto Main1    Effectue OR, actualise TOS ;écrit nouveau sommet

dup1      MAR = SP = SP + 1                       MAR et SP =futur sommet de pile
dup2      MDR = TOS; wr; goto Main1               Ecrit nouveau sommet

pop1      MAR = SP = SP - 1; rd                   Lit 2ème mot de pile
pop2                                              Attend disponibilité du 2ème mot
pop3      TOS = MDR; goto Main1                   Actualise TOS avec nouveau sommet de pile

swap1     MAR = SP - 1; rd                        MAR =position 2ème mot de pile;lit le 2ème mot
swap2     MAR = SP                                MAR =sommet de pile
swap3     H = MDR; wr                             H =2ème mot; écrit nouveau sommet
swap4     MDR = TOS                               MDR =ancien sommet
swap5     MAR = SP - 1; wr                        Ecrit ancien sommet en 2ème position de pile
swap6     TOS = H; goto Main1                     Actualise TOS avec nouveau sommet
```

```
bipush1     SP = MAR = SP + 1                 (MBR =octet à empiler) MAR et SP =nouveau sommet
bipush2     PC = PC + 1; fetch                Prépare PC ; extrait prochain code d'opération
bipush3     MDR = TOS = MBR; wr; goto Main1   TOS et MDR =octet signé étendu; écrit nouveau sommet

i1oad1      H = LV                            (MBR =index)H =base des variables locales
i1oad2      MAR = MBRU + H; rd                MAR =adresse variable locale;lit variable
iload3      MAR=SP=SP+1                       MAR et SP=nouveau sommet
i1oad4      PC = PC + 1; fetch; wr            Extrait prochain code d'opération; écrit nouveau sommet
iload5      TOS = MDR; goto Main1             Actualise TOS

istore1     H= LV                             (MBR = index) H = base des variables locales
istore2     MAR = MBRU + H                    MAR = adresse variable locale
istore3     MDR = TOS; wr                     MDR = sommet de pile ; écrit dans variable
istore4     SP = MAR = SP – 1; rd             Lit 2e mot de pile
istore5     PC = PC + 1; fetch                Extrait code d'op.suivant
istore6     TOS = MDR; goto Main1             Actualise TOS avec nouveau sommet

wide1       PC=PC + 1;fetch;goto(MBR OR 0x100) Extrait octet suivant (operande ou code d'o p.)
                                              Branchement avec QQ bit de MPC active

wide_i1oad1 PC = PC + 1 ; fetch               (MBR contient 1er octet index) Extrait 2e octet index
wide_i1oad2 H = MBRU <<8                      H = 1er octet étendu non signé décalé de 8 bits
wide_i1oad3 H = MBRU OR H                     H = index de 16 bits de la variable locale
wide_i1oad4 MAR = LV + H; rd; goto i1oad3     MAR = adresse var. loc.a empiler ; lit variable

wide_istore1   PC = PC + 1; fetch             (MBR :1er octet index)Extrait 2e octet index
wide_istore2   H = MBRU <<8                   H = 1er octet étendu non signé décalé de 8 bits
wide_istore3   H = MBRU OR H                  H = index de 16 bits de la variable locale
wide_istore4   MAR = LV + H; goto istore3     MAR = adresse variable locale cible
```

Implémentation du microprogramme



- La boucle principale du micro-programme

Label	Opérations	Commentaires
Main1	PC = PC + 1; fetch; goto MBR	Boucle principale

On suppose que PC a été initialisé avec l'adresse d'un emplacement mémoire contenant la première instruction de programme à exécuter

Tous les retours vers Main1 devront avoir préalablement placé dans PC l'adresse du prochain code opération et MBR chargé par ce code opération

La boucle principale se déroule sur un seul cycle d'horloge

en parallèle on exécute

incrémentation de PC pour adresser l'instruction suivante

fetch déclenche une lecture en mémoire pour accéder à la prochaine instruction.

(Le nouvel octet dont on demande l'accès en mémoire à ce moment ne sera disponible qu'à la troisième micro-instruction)

goto MBR : on débute le micro-programme de l'instruction présente dans le registre MBR

Déroulement de l'instruction *dup* de l'IJVM

dup duplique le sommet de la pile

Label	Opérations	Commentaires
dup1	MAR = SP = SP + 1	on incrémente le sommet de la pile
dup2	MDR = TOS; wr;	La valeur du sommet de la pile est copiée dans MDR qui sera écrit dans le nouveau sommet de pile

Déroulement de l'instruction *bipush* de l'IJVM

bipush <n> n octet signé

Label	Opérations	Commentaires
bipush1	SP = MAR = SP + 1	Incrémente le sommet de la pile MBR est chargé avec l'opérande (octet n)
bipush2	PC = PC + 1; fetch	Incrémente le compteur d'instruction et lance l'extraction en mémoire du prochain code opération
bipush3	MDR = TOS = MBR; wr; goto Main1	L'octet dans MBR est copié dans TOS et dans MDR. Une écriture de l'octet au sommet de la pile est lancée. Retour à la boucle principale

Attention: l'octet signé doit être porté à 32 bits lors de la copie de MBR dans TOS

Déroulement de l'instruction *iadd* de l'IJVM

Label	Opérations	Commentaires
iadd1	MAR = SP = SP – 1; rd	Lit le 2ème mot dans la pile
iadd2	H = TOS	H est chargé avec le sommet de la pile
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Effectue l'addition; actualise TOS et écrit le résultat en haut de la pile en mémoire

Le sommet de la pile est déjà mémorisé dans TOS

Il faut lire en mémoire le second élément de la pile

- à la fin du premier cycle d'horloge on déclenche un accès en mémoire à l'adresse contenue dans MAR : rd dans iadd1
- la valeur sera disponible dans MDR à la fin du second cycle (fin de iadd2).
- au cours du premier cycle on décrémente SP et on place son contenu dans MAR
- après décrémentation SP pointe sur ce qui sera le nouveau sommet de la pile après la fin de l'addition
- c'est à cette adresse qu'il faudra écrire le résultat de l'addition par wr dans iadd3
- durant le second cycle on place TOS dans H (registre de maintient pour l(addition)
- durant le cycle 2 l'accès en mémoire pour lire le 2nd mot de la pile se déroule
- le cycle 3 effectue l'addition de H avec MDR qui a été chargé avec le 2nd mot de la pile en fin du cycle 2
- Le résultat de l'addition est placé dans TOS et dans MDR
- On déclenche une écriture du résultat au sommet de la pile wr dans iadd3
- Iadd3 termine l'instruction et saute à la boucle principale

durant le déroulement de iadd1 le champs adresse est placé dans MPC qui pointe alors sur iadd2

idem pour iadd2

iadd3 pointe sur Main1 : MPC est chargé avec l'adresse de Main1

Déroulement de l'instruction *iload* de l'IJVM

iload <varnum> **varnum** est le numéro du mot de 32 bit dans le pool de variables locales

label	opérations	commentaires
iload1	H = LV	H est initialisé avec le pointeur de début de zone de variables locales pendant ce temps MBR = varnum
iload2	MAR = MBRU + H; rd	MAR est chargé avec l'adresse de la variable locale à charger en haut de la pile La lecture en mémoire est lancée
iload3	MAR = SP = SP + 1	Le pointeur de sommet de la pile est incrémenté et copié dans MAR
iload4	PC = PC + 1; fetch; wr	La lecture en mémoire s'est terminés et MDR contient la variable en question On peut lancer sont écriture en mémoire enhaut de la pile Parallèlement on incrémente le compteur de programme pour aller cherche l'instruction suivante en mémoire
iload5	TOS = MDR; goto Main1	Le registre TOS est actualisé avec le contenu du nouveau sommet de la pile On saute à la boucle principale

Instructions étendues

```
wide1          PC = PC + 1 ; fetch ;goto (MBR OR 0x100)
                  Extrait octet suivant (opérande ou code
                  d'o p.)

                  Branchement avec 9ième bit de MPC
                  activé


wide_iload1       PC = PC + 1 ; fetch    (MBR contient
                  1er octet index) Extrait 2e octet index
wide_iload2       H = MBRU <<8    H = 1er octet étendu
                  non signé décalé de 8 bits
wide_iload3       H = MBRU OR H      H = index de 16 bits
                  de la variable locale
wide_iload4       MAR = LV + H; rd; goto iload3
MAR = adresse var. loc.a empiler ; lit variable
```

Rupture de séquence d'instructions

```
goto1       OPC = PC - 1          Préserve l'adresse
            du code d'opération actuel
goto2       PC = PC + 1; fetch    (MBR = 1er octet
            index) Extrait 2ème octet index
goto3       H= MBR <<8            H = 1er octet signé
            décalé de 8 bits
goto4       H = MBRU OR H         H =offset 16 bits de
            branchement
goto5       PC = OPC + H; fetch PC =base
            constantes+index;extrait code
            d'opération
goto6       goto Mainl            Attente
            disponibilité du code d'opération dans
            MBR
```

Conditionnelle

```
if_icmpeq1      MAR = SP = SP – 1 ; rd    Lit 2e mot de pile
if_icmpeq2      MAR = SP = SP – 1         MAR et SP = futur
                                          sommet de pile
if_icmpeq3      H = MDR; rd               H = mot lu ; lit
                                          nouveau sommet de
                                          pile
if_icmpeq4      OPC = TOS                 Préserve
                                          temporairement ancien
                                          sommet dans OPC
if_icmpeq5      TOS = MDR                 Actualise TOS avec
                                          nouveau sommet
if_icmpeq6      Z = OPC – H; if (Z) goto T; else goto F
                                          Teste;branchement
                                          vers T si Z
                                          activé,sinon vers F
```

Conditionnelle

```
T    OPC = PC – 1; goto goto2    Comme pour goto1,utile pour
                                 adresse cible

F    PC = PC + 1                 Saut 1er octet offset, PC
                                 pointe sur 2ème octet

F2   PC = PC + 1; fetch          PC pointe sur prochain
                                 code d'opération;extrait code

F3   goto Mainl                  Attente de disponibilité du
                                 code dans MBR
```

Conditionnelle

microProgram

```
main1:Z=not(TOS); if (N)  goto T ; else goto F
F: MAR=1; goto main1
T:MDR=1
```

```
POS|MIC-INSTRUCTION
0  |00000000100000010001000000010001111
1  |00000000001000101100000000000000111
256|00000000100000010001000000100001111
```

Mir

```
000000001 000001000100000001000001111
```

```
   ADDR     JJJSSFFEEIIHOTCLSPMMWRF   B
            MAALRO1NNNN  PPOPVPCDARE   U
            PMMLA   ABVC  CSP  RRIAT   S
            CNZB1      A          TDC
                                  EH   B
```

Appel de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Début d'un appel
de procédure

Pool de
constantes

SP

LV

Pile de main

Zone de
méthodes

Appel de procédure

Code appelant
(Main)

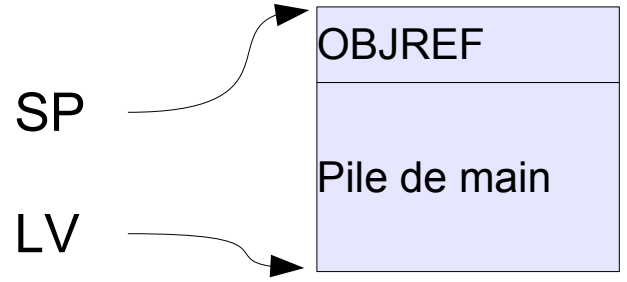
```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Pool de constantes

On empile un pointeur vers l'objet qui possède la procédure



Zone de méthodes

Appel de procédure

Code appelant
(Main)

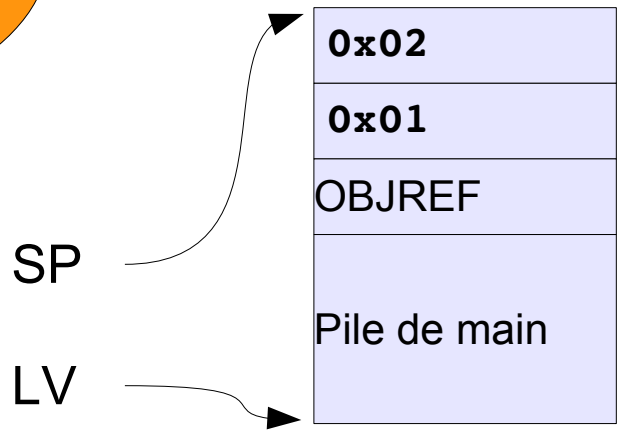
```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE r sultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

On place deux arguments
Sur la pile

Pool de
constantes



Zone de
méthodes

Appel de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Appel
De la procédure add

Pool de
constantes

add →

Ptr sur procédure

Zone de
méthodes

code procédure

SP →

LV →

0x02
0x01
OBJREF
Pile de main

Appel de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Ici il y a 2 arguments MAIS
le nombre d'arguments retenu par la procédure est
3 (=**α**) car on compte OBJREF

Pool de constantes

add → | Ptr sur procédure |

Zone de méthodes

code procédure :
| Nb arguments=α |
| Nb var. Loc. = β |
| code |

SP →
SP- **α** +1 →
LV →

| 0x02 |
| 0x01 |
| OBJREF |
| Pile de main |

Appel de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

A la place de OBJREF
On place un pointeur de liaison
vers le nouveau sommet
de la pile

Ptr.liaison
=
SP+β+1

SP+β+1

Place pour
les var.loc.

SP

SP- α +1

LV

0x02
0x01
Ptr de liaison
Pile de main

Pool de
constantes

add

Ptr sur procédure

Zone de
méthodes

code
procédure

Nb arguments=α
Nb var. Loc. = β
code

Appel de procédure

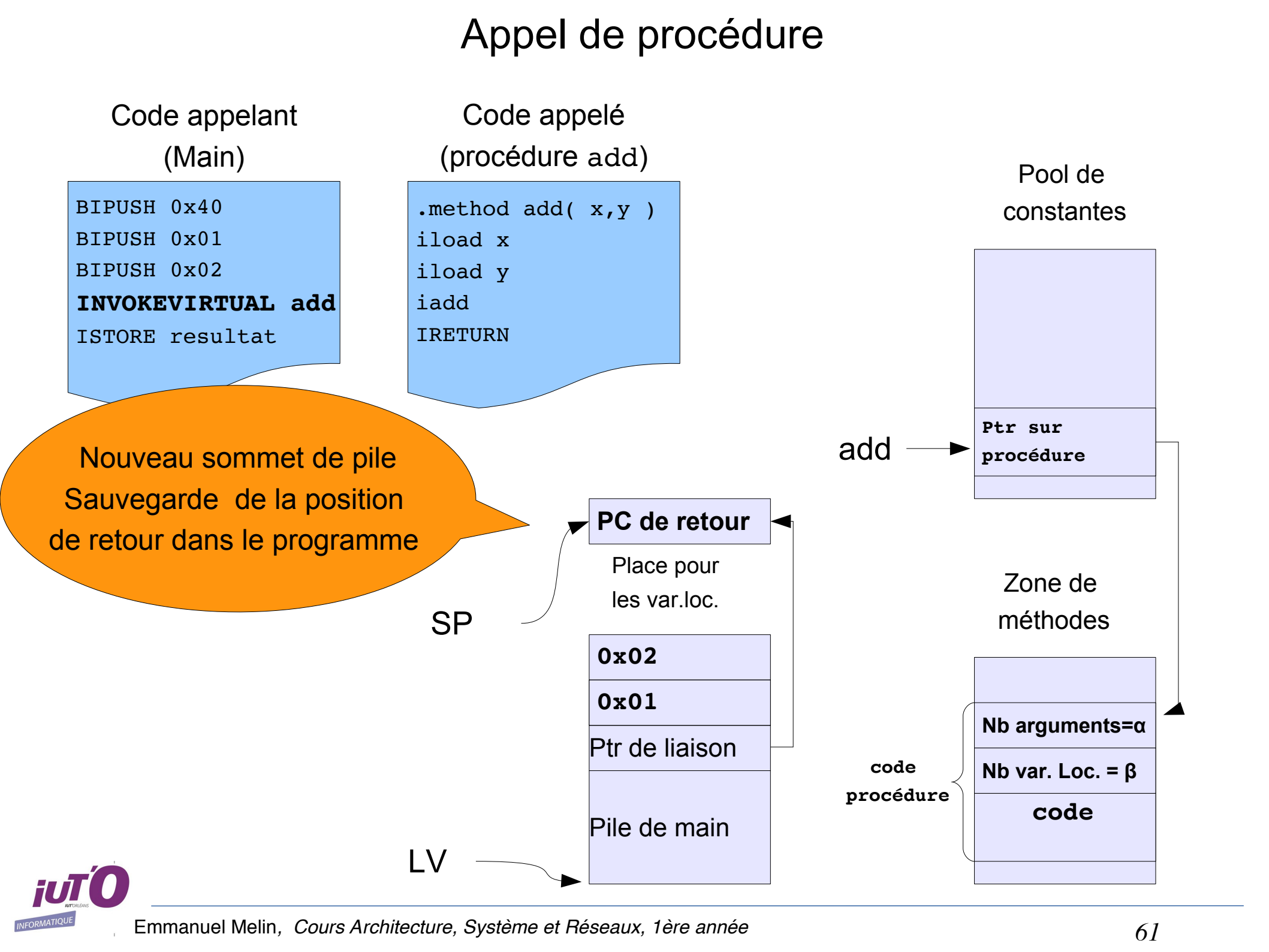
Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Nouveau sommet de pile
Sauvegarde de la position
de retour dans le programme



Appel de procédure

Code appelant (Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

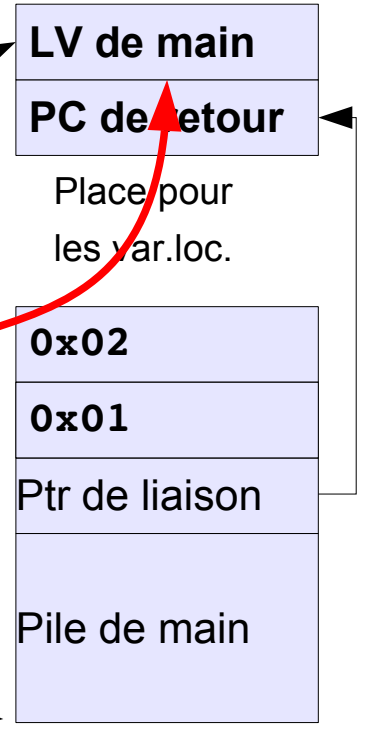
Code appelé (procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Nouveau sommet de pile
Sauvegarde de la position de référence des variables locales de main

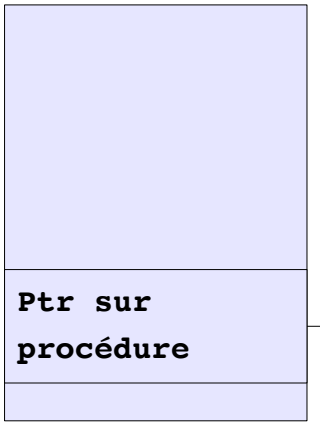
SP

LV



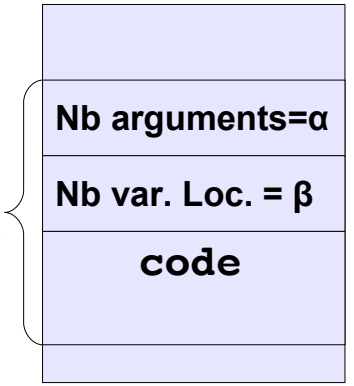
Pool de constantes

add



Zone de méthodes

code procédure



Appel de procédure

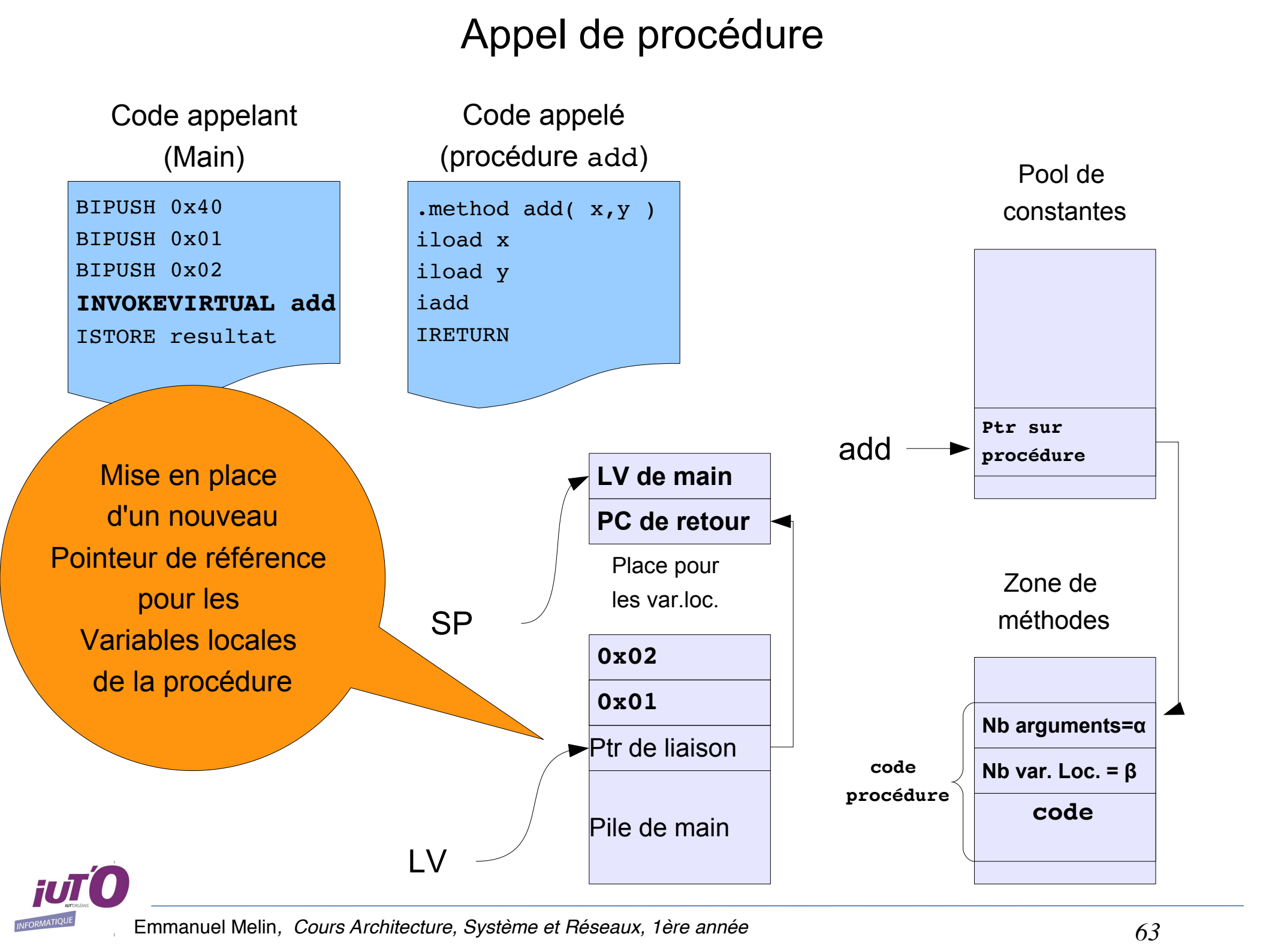
Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Mise en place
d'un nouveau
Pointeur de référence
pour les
Variables locales
de la procédure



Appel de procédure

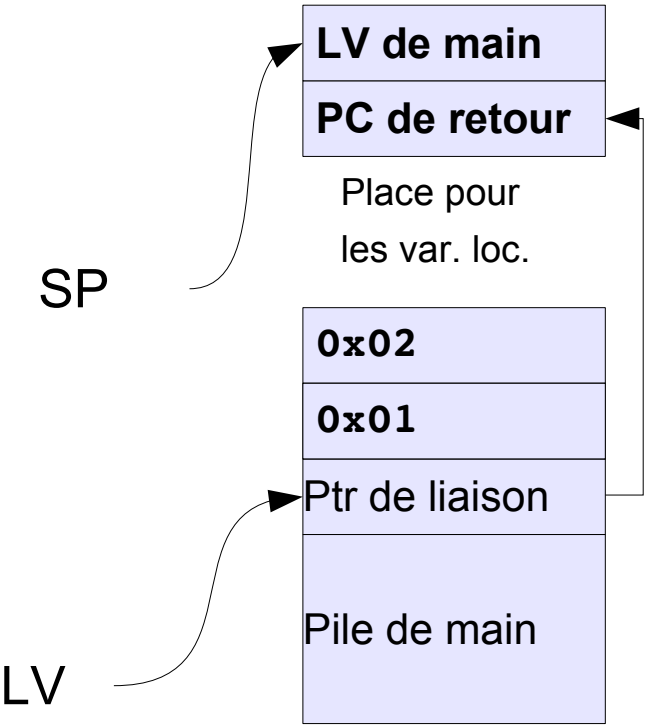
Code appelant (Main)

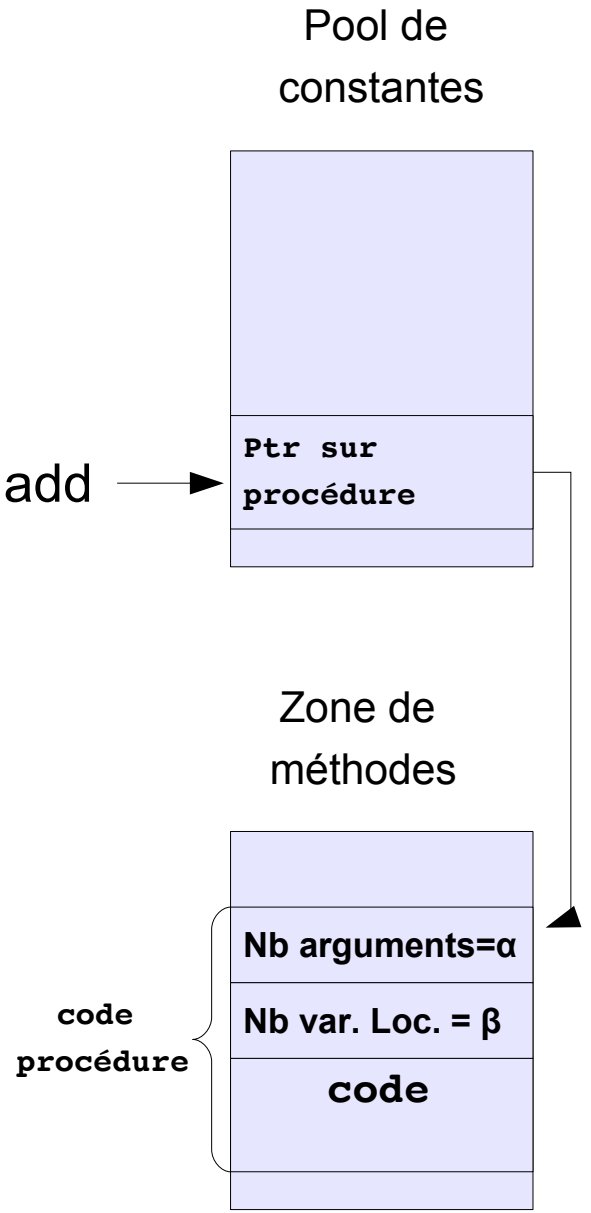
```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé (procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Exécution de la procédure





Appel de procédure

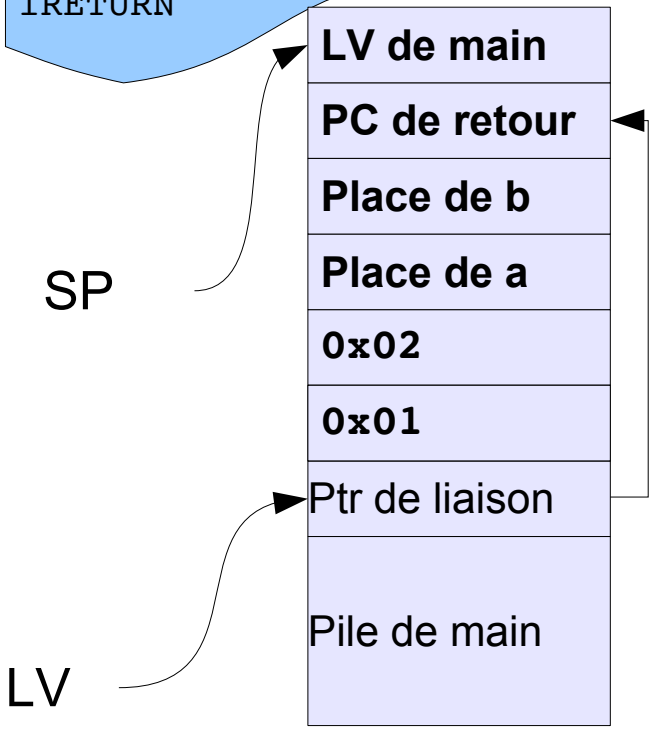
Code appelant (Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé (procédure add)

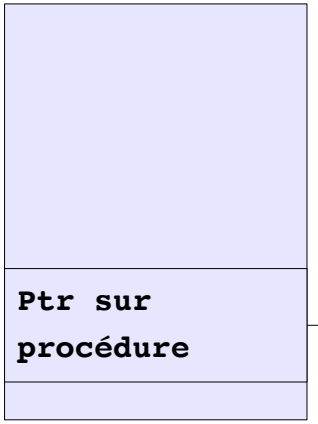
```
.method add( x,y )
.var
a
b
.end-var
iload x
iload y
iadd
IRETURN
```

Accès aux:
Param. : LV+1 ou LV+2
Var. Loc. a et b :
Via LV+**α-1+1 ou +2**



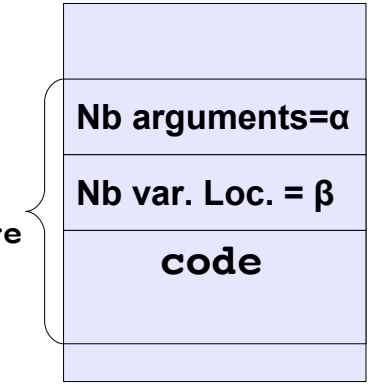
Pool de constantes

add



Zone de méthodes

code procédure



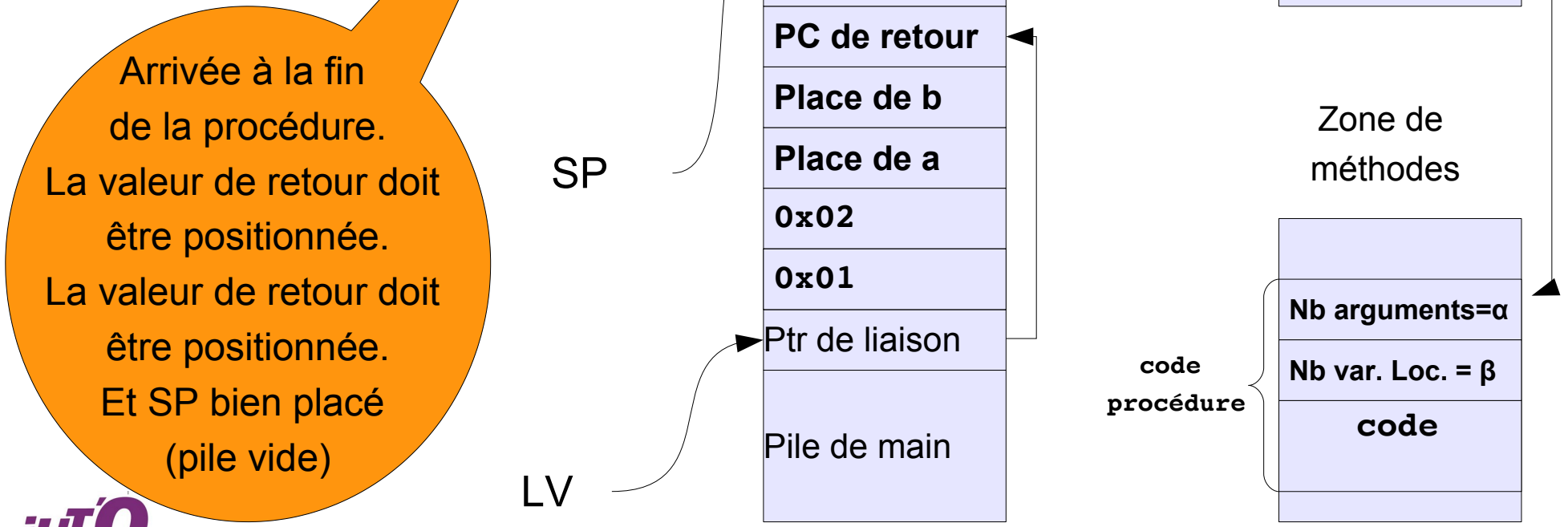
Appel de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```



Fin de procédure

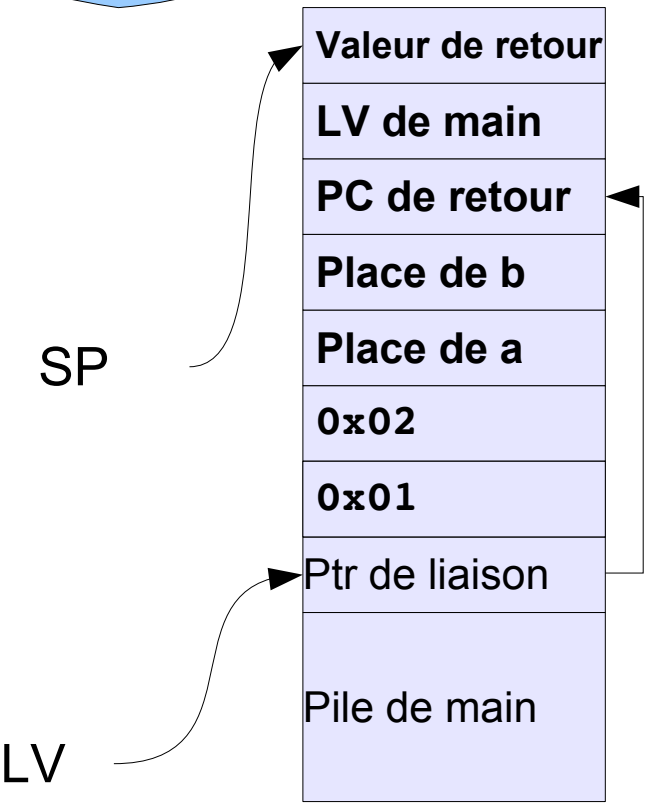
Code appelant (Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

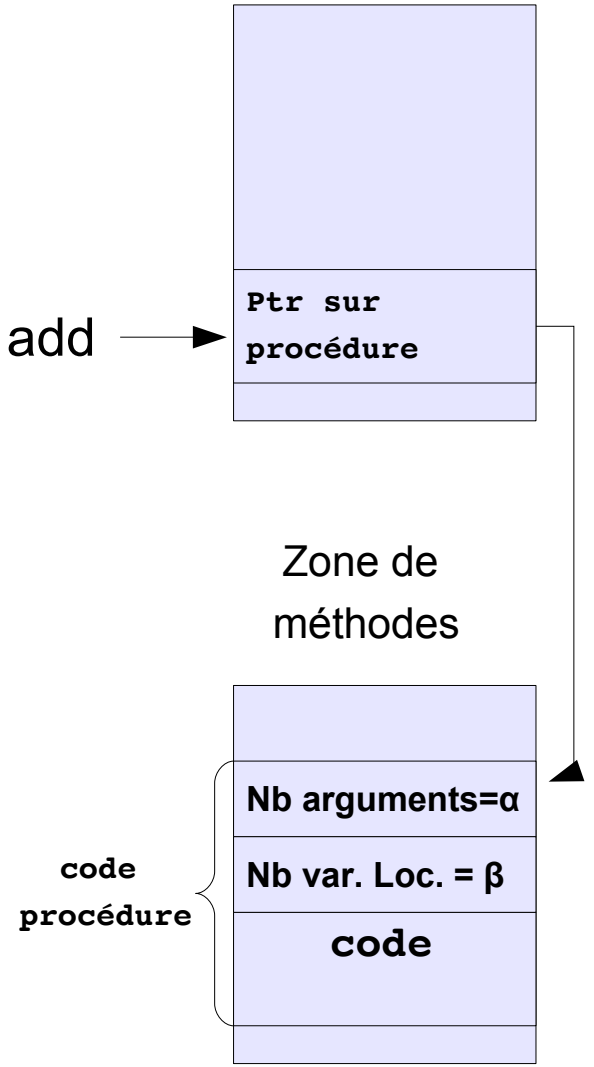
Code appelé (procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Le Return est exécuté



Pool de constantes



Fin de procédure

Code appelant
(Main)

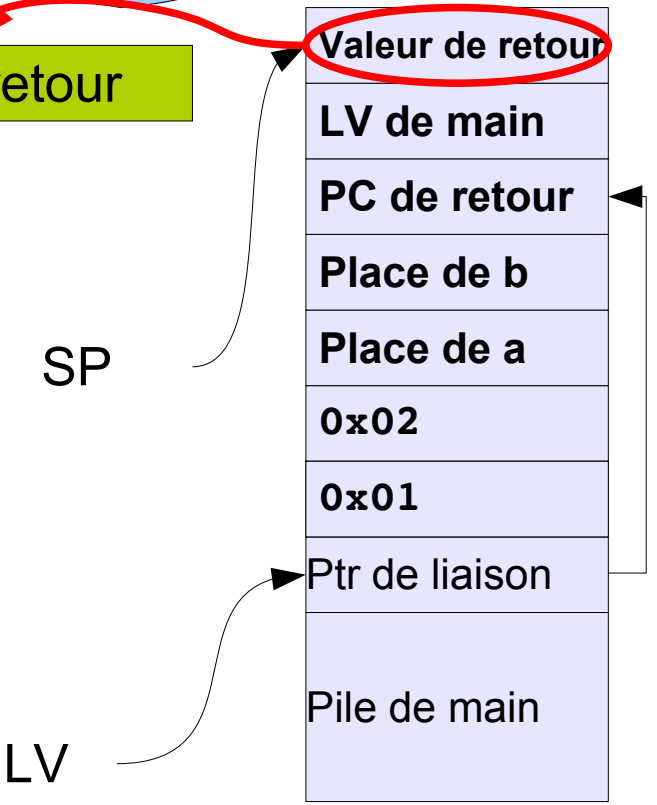
```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

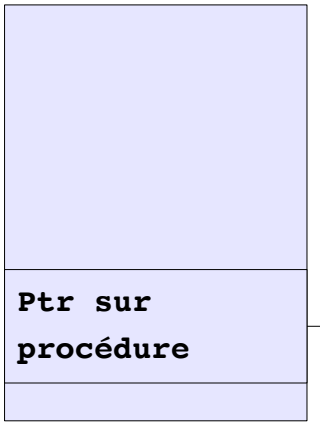
```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Valeur de retour

Sauvegarde de
la valeur de retour

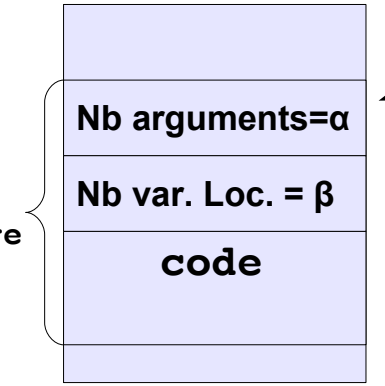


Pool de
constantes



add

Zone de
méthodes



code
procédure

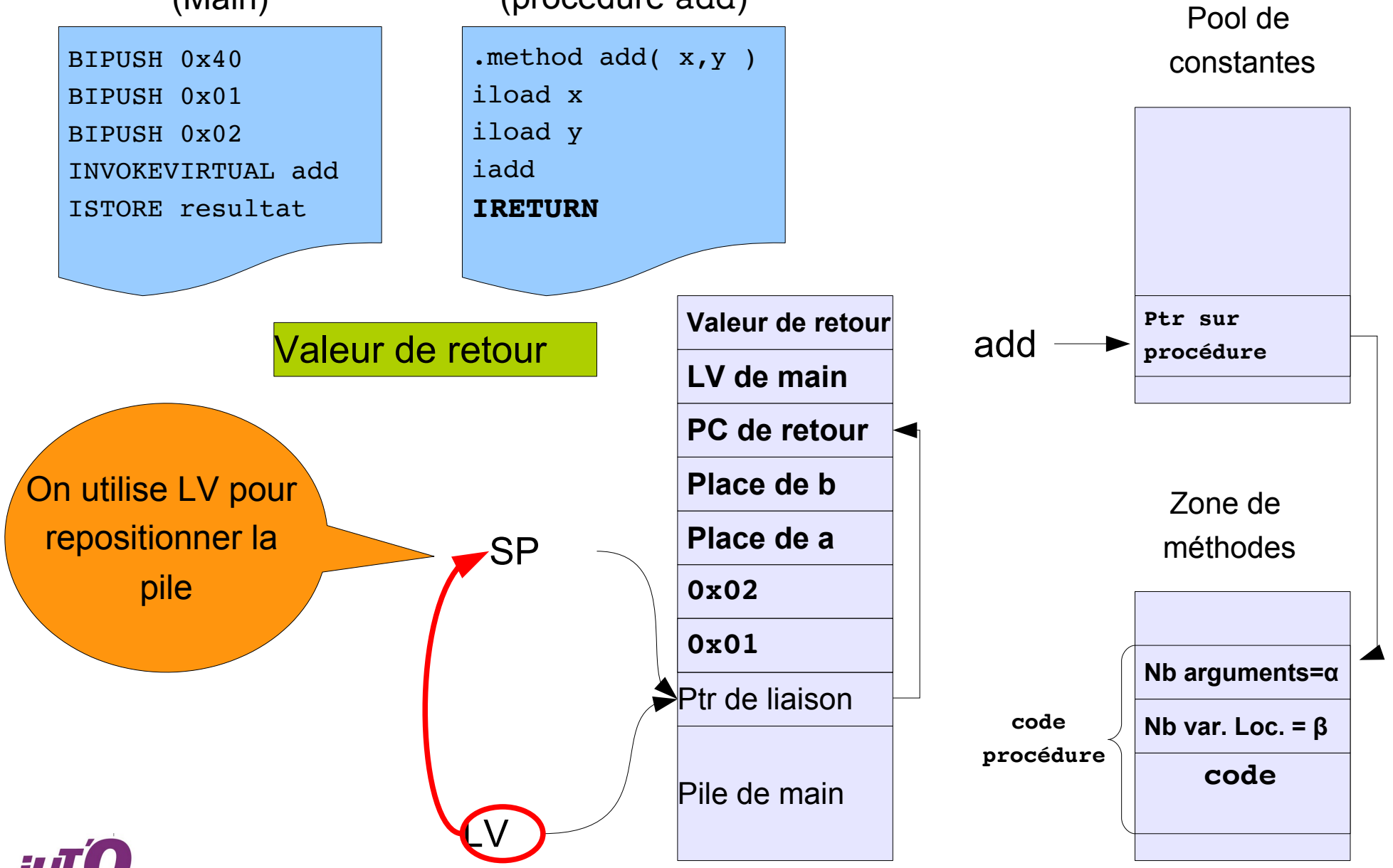
Fin de procédure

Code appelant (Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé (procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```



Fin de procédure

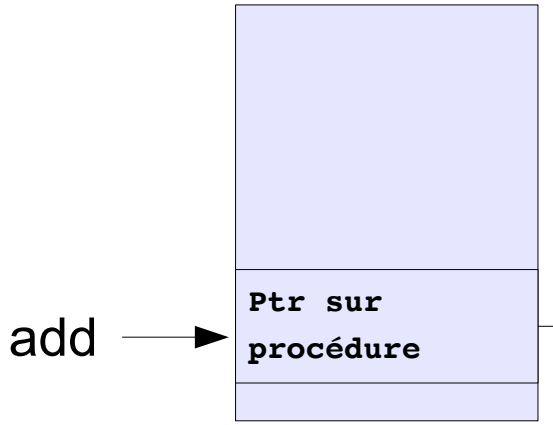
Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Pool de constantes



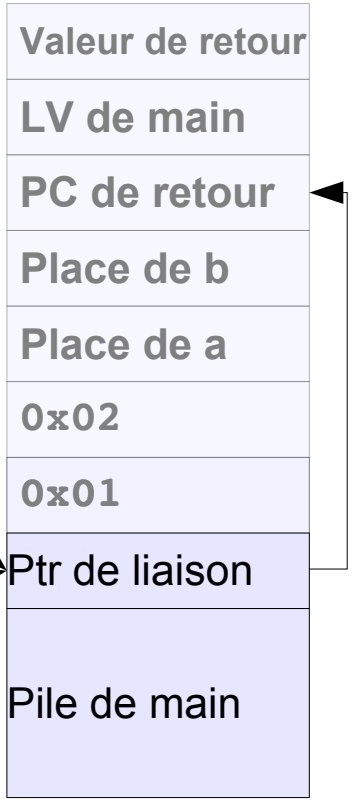
add

Valeur de retour

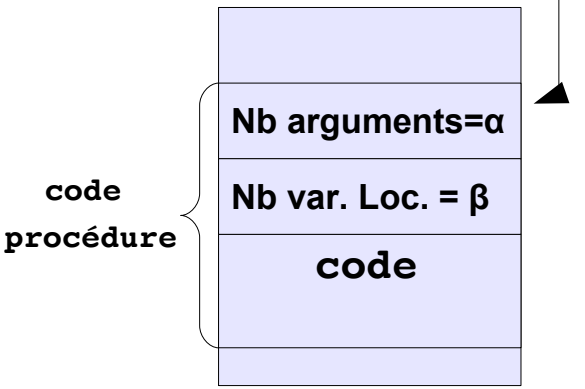
C'est comme si on Effaçait la pile

SP

LV



Zone de méthodes



Fin de procédure

Code appelant
(Main)

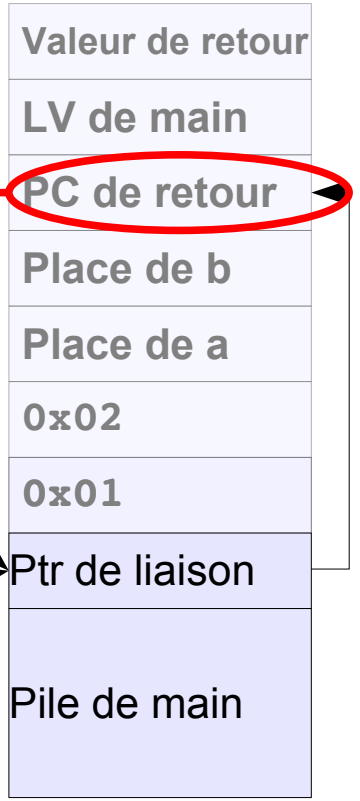
```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

PC

Valeur de retour



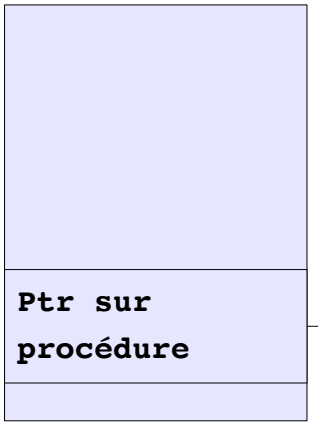
SP

LV

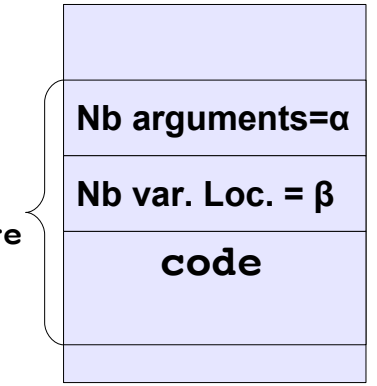
Mais elle n'est
pas encore effacée!
On récupère via le pointeur
de liaison l'adresse de retour
qui est restaurée

Pool de
constantes

add



Zone de
méthodes



code
procédure

Fin de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

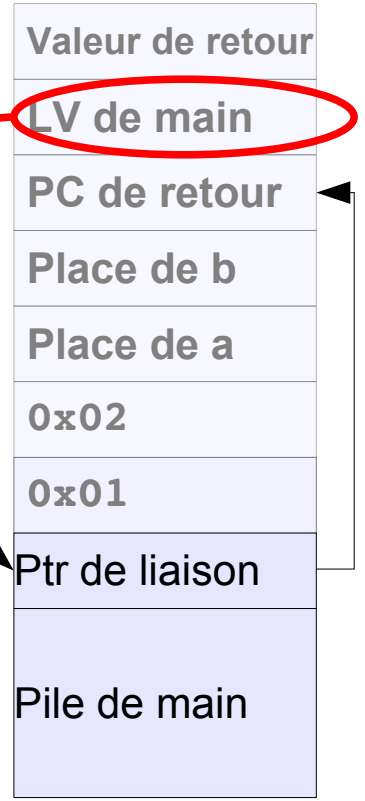
PC

Valeur de retour

On récupère via le pointeur de liaison l'adresse de LV initiale qui permet de trouver les variables locales de la procédure appelante

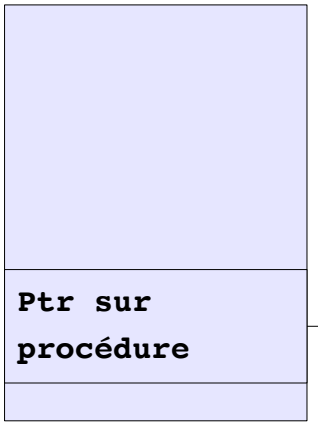
SP

LV



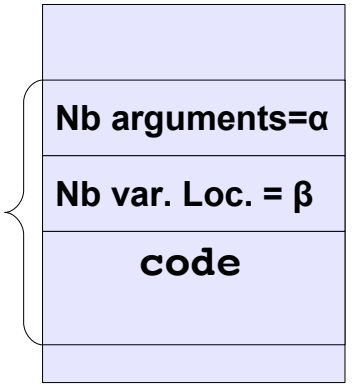
Pool de constantes

add



Zone de méthodes

code procédure



Fin de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

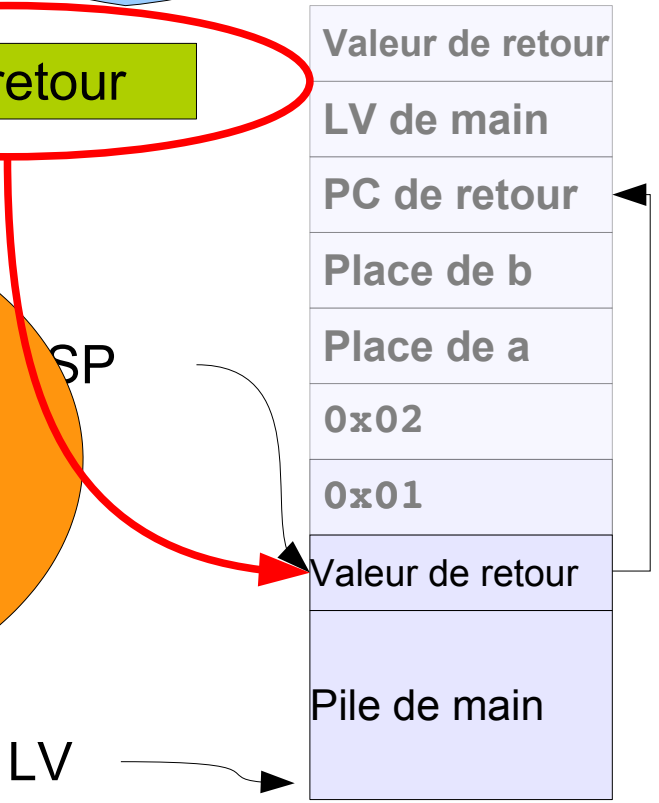
PC

Valeur de retour

On restaure la pile à l'état Initial en plaçant la valeur de retour au sommet, là où pointe SP

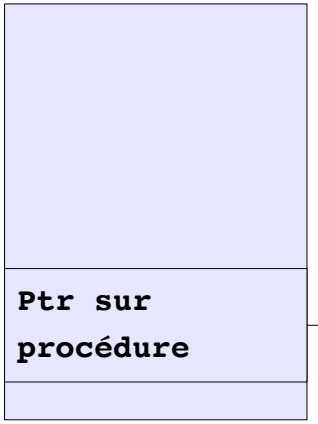
SP

LV



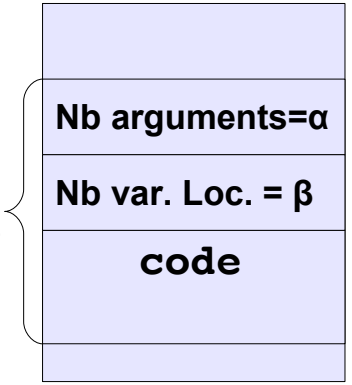
Pool de constantes

add



Zone de méthodes

code procédure



Fin de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

PC

Pool de constantes

add → **Ptr sur procédure**

La pile est (presque) restaurée dans l'état initial

SP

Valeur de retour

Pile de main

LV

Zone de méthodes

code procédure

Nb arguments=α

Nb var. Loc. = β

code

Fin de procédure

Code appelant
(Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

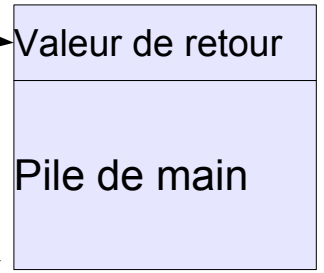
```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

PC

La procédure appelante
est exécutée
là où elle en était.
Elle trouve le résultat
de la procédure
sur le sommet de la pile

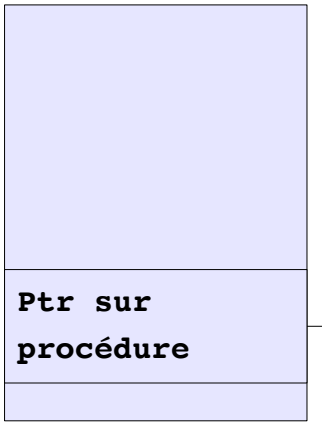
SP

LV



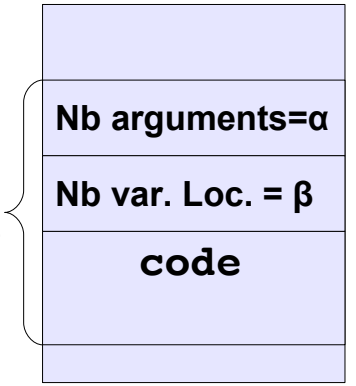
Pool de
constantes

add



Zone de
méthodes

code
procédure



Dans le microcode

```
invokevirtual1      PC = PC + 1 ; fetch      (MBR = le 1er 0ctet index) Extrait 2ème octet index
invokevirtual2      H = MBRU <<8             H = 1er octet non signé décalé de 8 bits
invokevirtual3      H = MBRU OR H            H = offset du pointeur de la méthode appelée
invokevirtual4      MAR = CPP + H; rd        Récupère pointeur vers méthode dans zone CPP
invokevirtual5      OPC = PC + 1             Préserve temporairement PC de retour dans OPC
invokevirtual6      PC = MDR; fetch          PC =adresse méthode; extrait 1er octet du nombre param;
invokevirtual7      PC = PC + 1; fetch       Extrait 2ème octet du nombre de parametres
invokevirtual8      H = MBRU <<8             H =1er octet non signe decale de 8bits
invokevirtual9      H = MBRU OR H            H =nombre de parametres
invokevirtual10     PC = PC + 1; fetch       Extrait ler octet du nombre de variables locales
invokevirtual11     TOS = SP – H             TOS = adresse de OBJREF – 1
invokevirtual12     TOS = MAR = TOS + 1      TOS = adresse de OBJREF (nouveau LV)
invokevirtual13     PC = PC + 1; fetch       Extrait 2ème octet du nombre de variables locales
invokevirtual14     H = MBRU <<8             H =1er octet non signe décalé de 8 bits
invokevirtual15     H = MBRU OR H            H =nombre de variables locales
invokevirtual16     MDR = SP + H + 1; wr     Ecrase OBJREF avec pointeur de liaison
invokevirtual17     MAR = SP = MDR           MAR et SP =adresse contenant ancien PC
invokevirtual18     MDR = OPC; wr            MDR =ancien PC ;ecrit au dessus variables locales
invokevirtual19     MAR = SP = SP + 1        Prépare MAR et SP pour stocker ancien LV
invokevirtual2O     MDR = LV; wr             Ecrit ancien LV au–dessus ancien PC préservé
invokevirtual21     PC = PC + 1; fetch       Extrait premier code d'opération de méthode invoquée
invokevirtual22     LV = TOS; goto Main1     Actualise LV avec nouvelle base variables locales


ireturn1            MAR = SP = LV; rd        Prépare SP et MAR pour récupération pointeur de liaison
ireturn2                                     Attend la disponibilité du pointeur
ireturn3            LV = MAR = MDR; rd       LV et MAR =pointeur de liaison; lit ancien PC
ireturn4            MAR = LV + 1             Prépare MAR pour lire ancien LV
ireturn5            PC = MDR; rd; fetch      Restauration ancien PC ;lit ancien LV;
                                             extrait prochain code d'opération
ireturn6            MAR = SP                 Prépare MAR pour écrire sommet de pile
ireturn7            LV = MDR                 Restaure ancien LV
ireturn8            MDR = TOS; wr; goto Main1    Ecrit valeur r
```

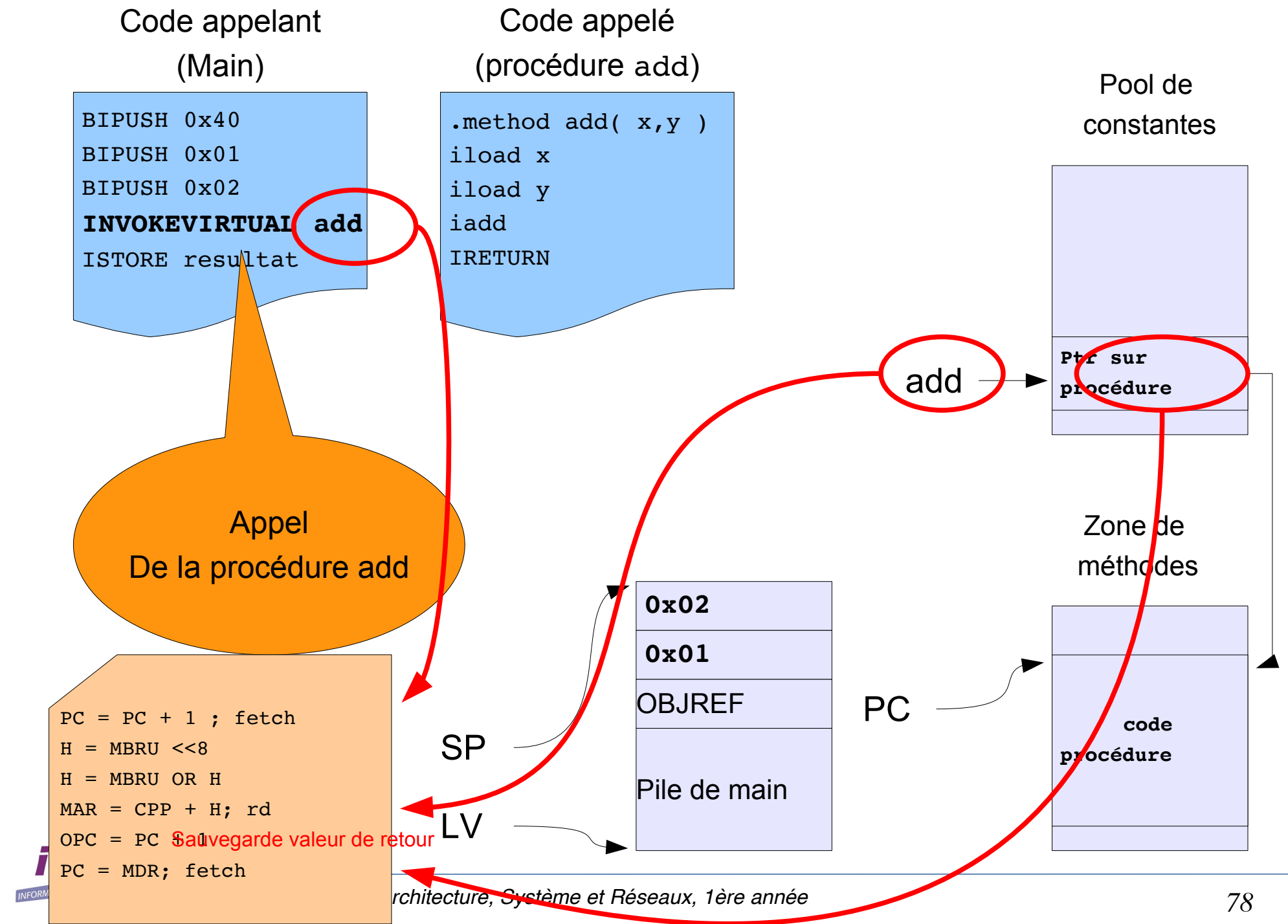
Exemple:

```
.main
BIPUSH 10
BIPUSH 00
BIPUSH 2
BIPUSH 3

INVOKEVIRTUAL add
BIPUSH 022
.end-main
.method add(x,y)
ILOAD x
IFEQ L1
BIPUSH 0
ILOAD x
BIPUSH 1
ISUB
ILOAD y
INVOKEVIRTUAL add
GOTO L2
L1: BIPUSH 0
L2: IRETURN
.end-method
```

Addr	Content
0x40000	0xb6 0x00 0x01 0x00
0x40004	0x01 0x00 0x00 0x10
0x40008	0x0a 0x10 0x00 0x10
0x4000c	0x02 0x10 0x03 0xb6
0x40010	0x00 0x02 0x10 0x16
0x40014	0x00 0x03 0x00 0x00
0x40018	0x15 0x01 0x99 0x00
0x4001c	0x12 0x10 0x00 0x15
0x40020	0x01 0x10 0x01 0x64
0x40024	0x15 0x02 0xb6 0x00
0x40028	0x02 0xa7 0x00 0x05
0x4002c	0x10 0x00 0xac 0x00

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

Appel de procédure: point vue du microprogramme



Appel de procédure: point vue du microprogramme

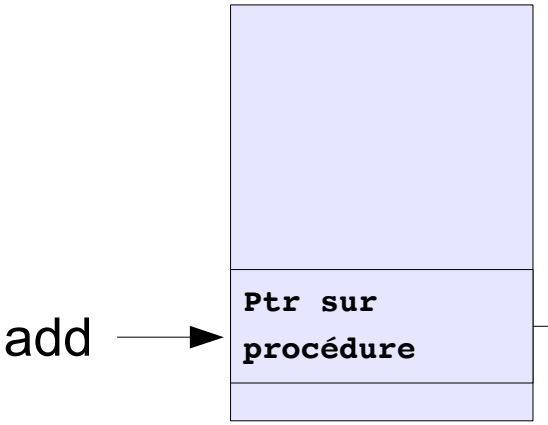
Code appelant (Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé (procédure add)

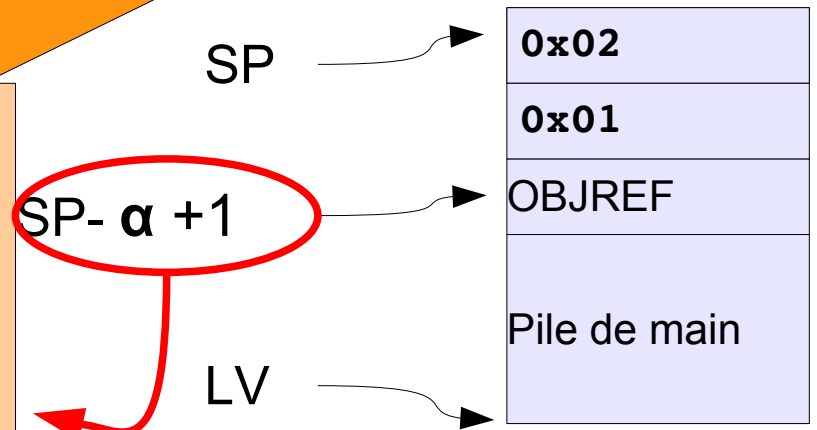
```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Pool de constantes

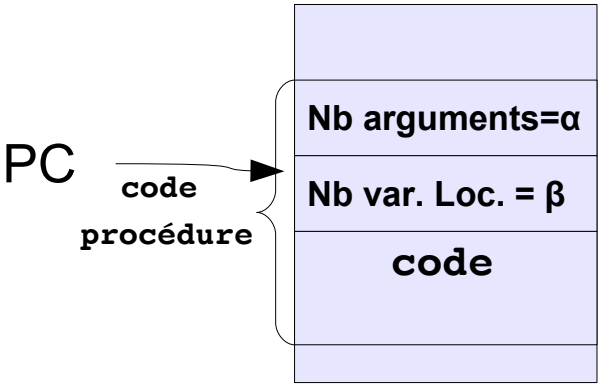


Récupération du nombre de paramètres **α (sur 2 octets)**

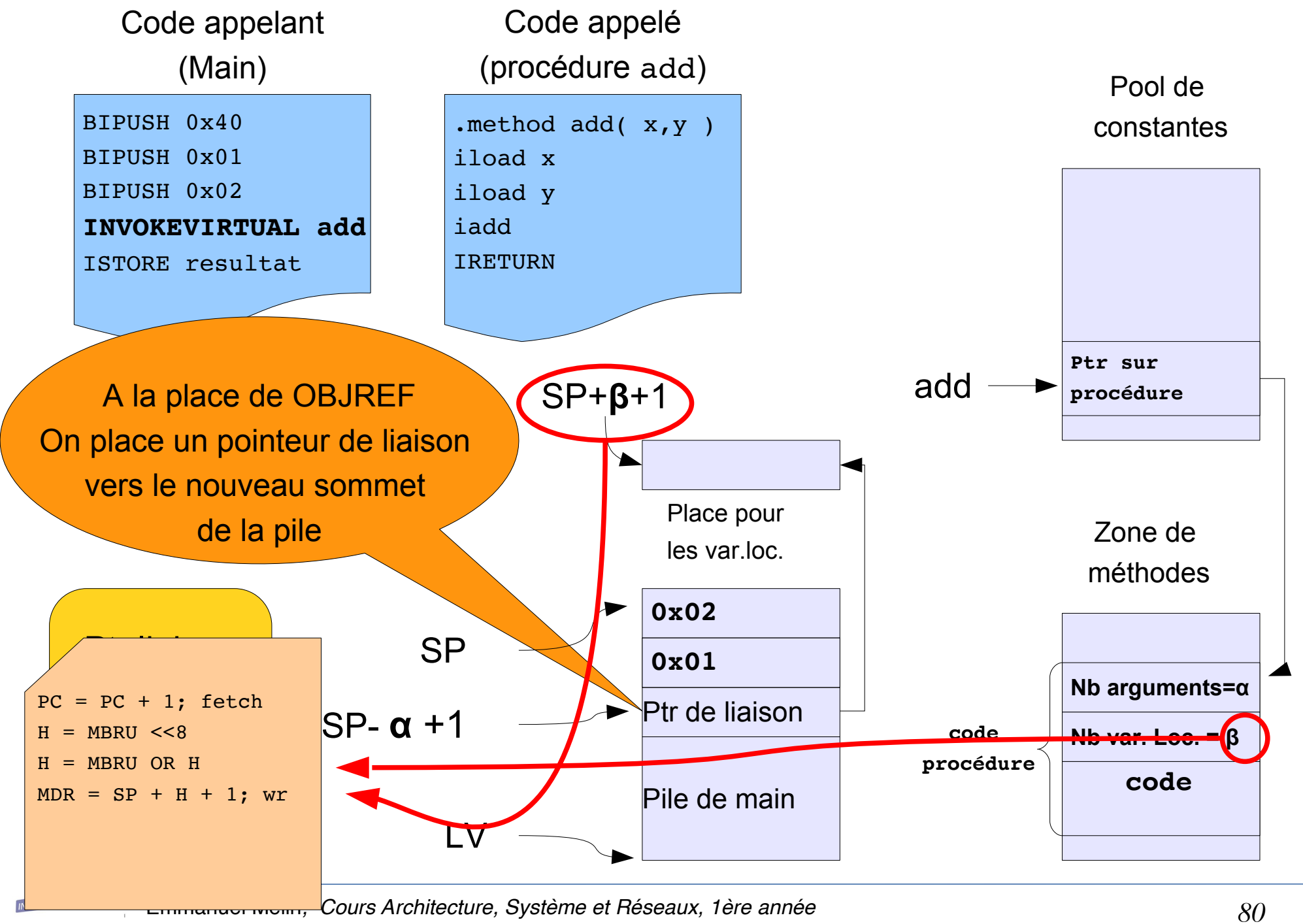
```
PC = PC + 1; fetch
H = MBRU <<8
H = MBRU OR H
PC = PC + 1; fetch
TOS = SP – H
TOS = MAR = TOS + 1
```



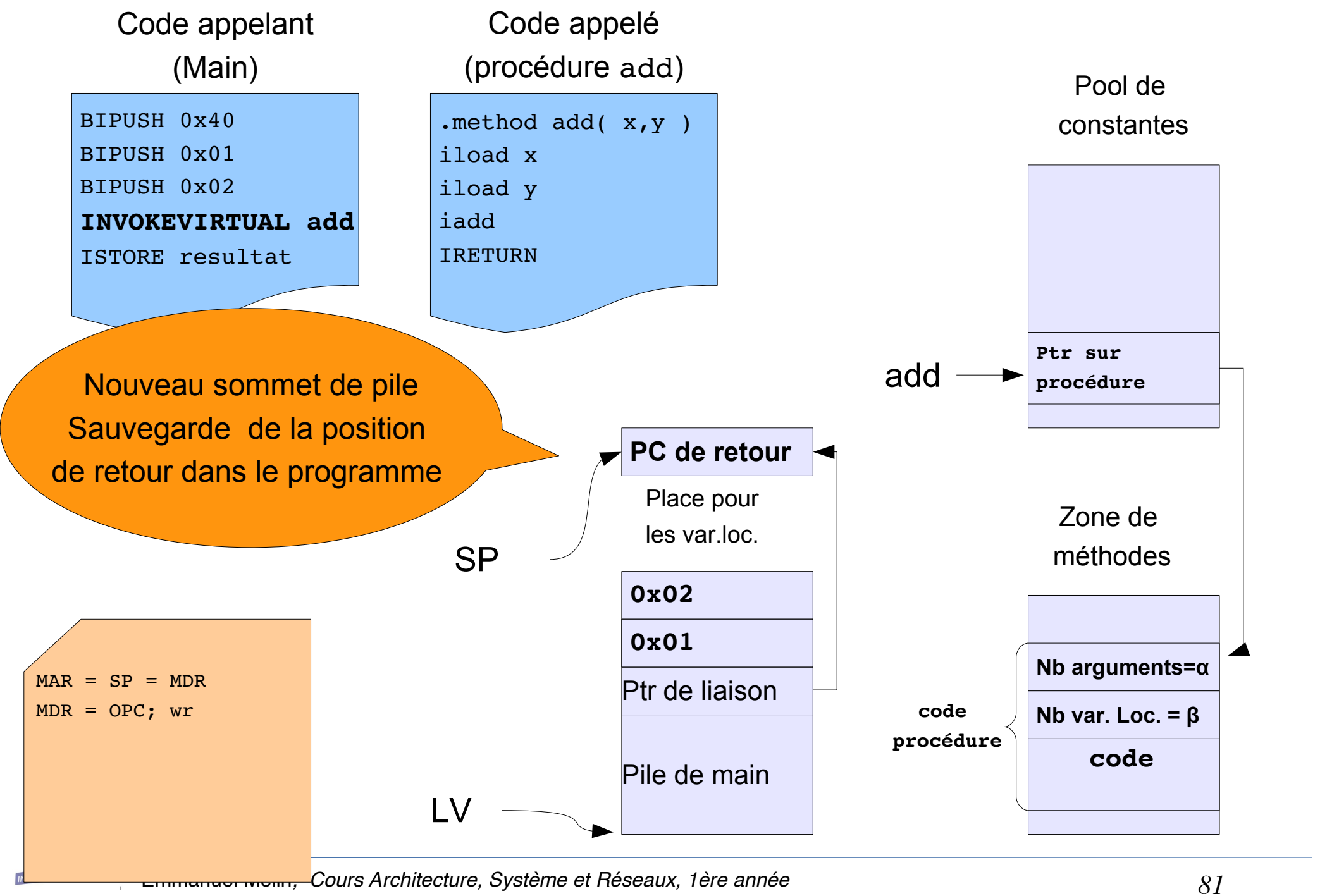
Zone de méthodes



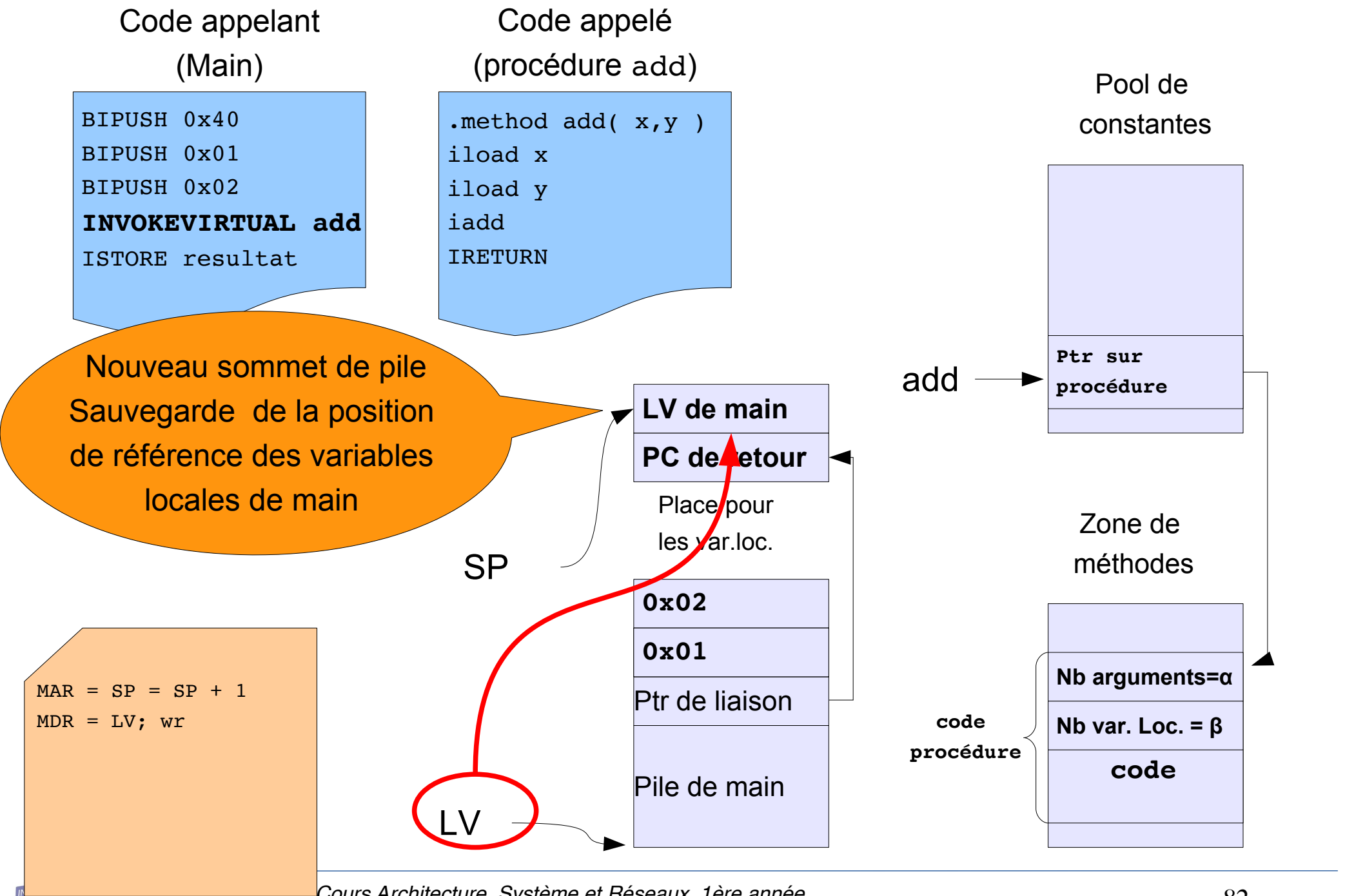
Appel de procédure: point vue du microprogramme



Appel de procédure: point vue du microprogramme



Appel de procédure: point vue du microprogramme



Appel de procédure: point vue du microprogramme

Code appelant
(Main)

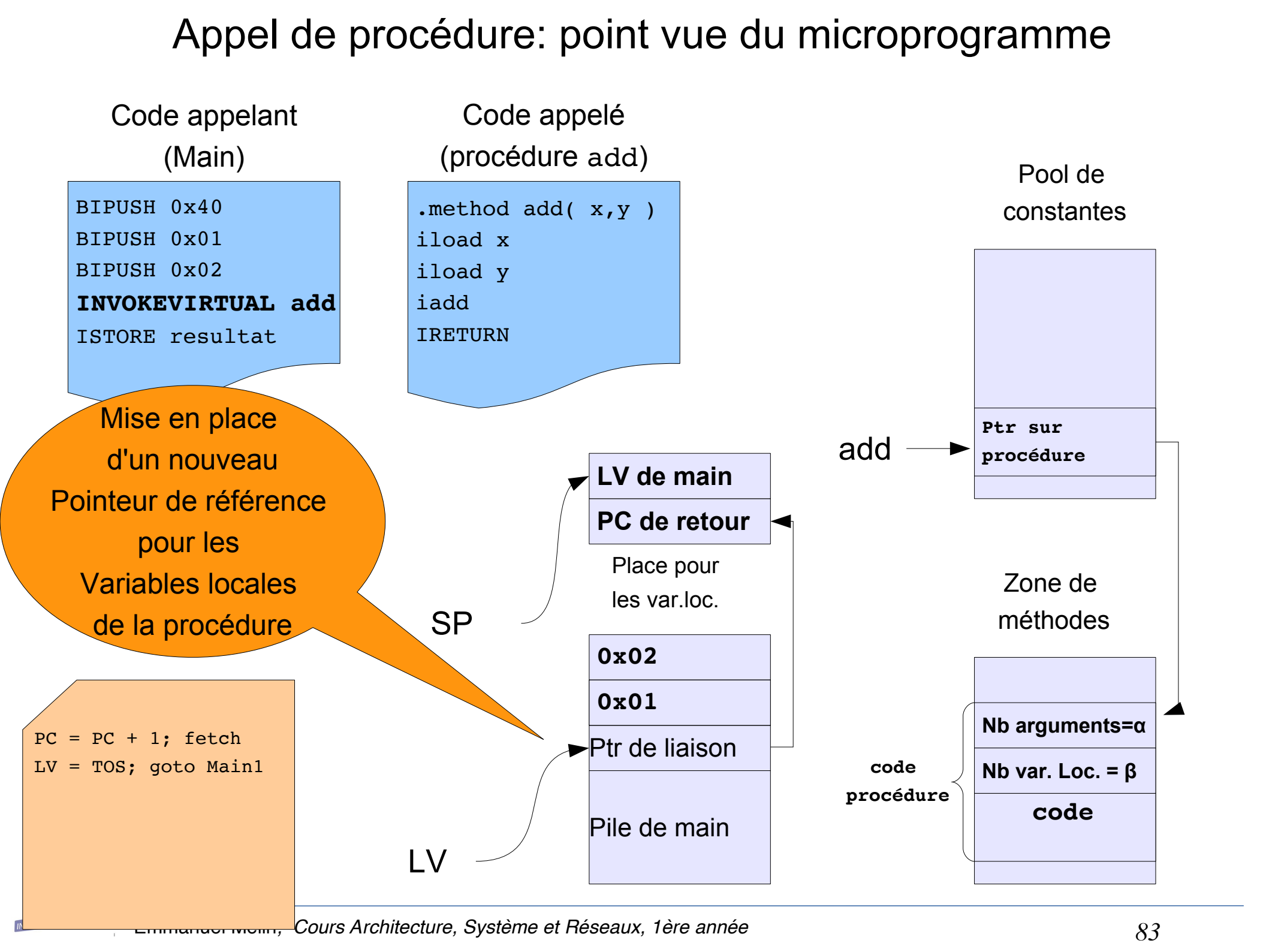
```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé
(procédure add)

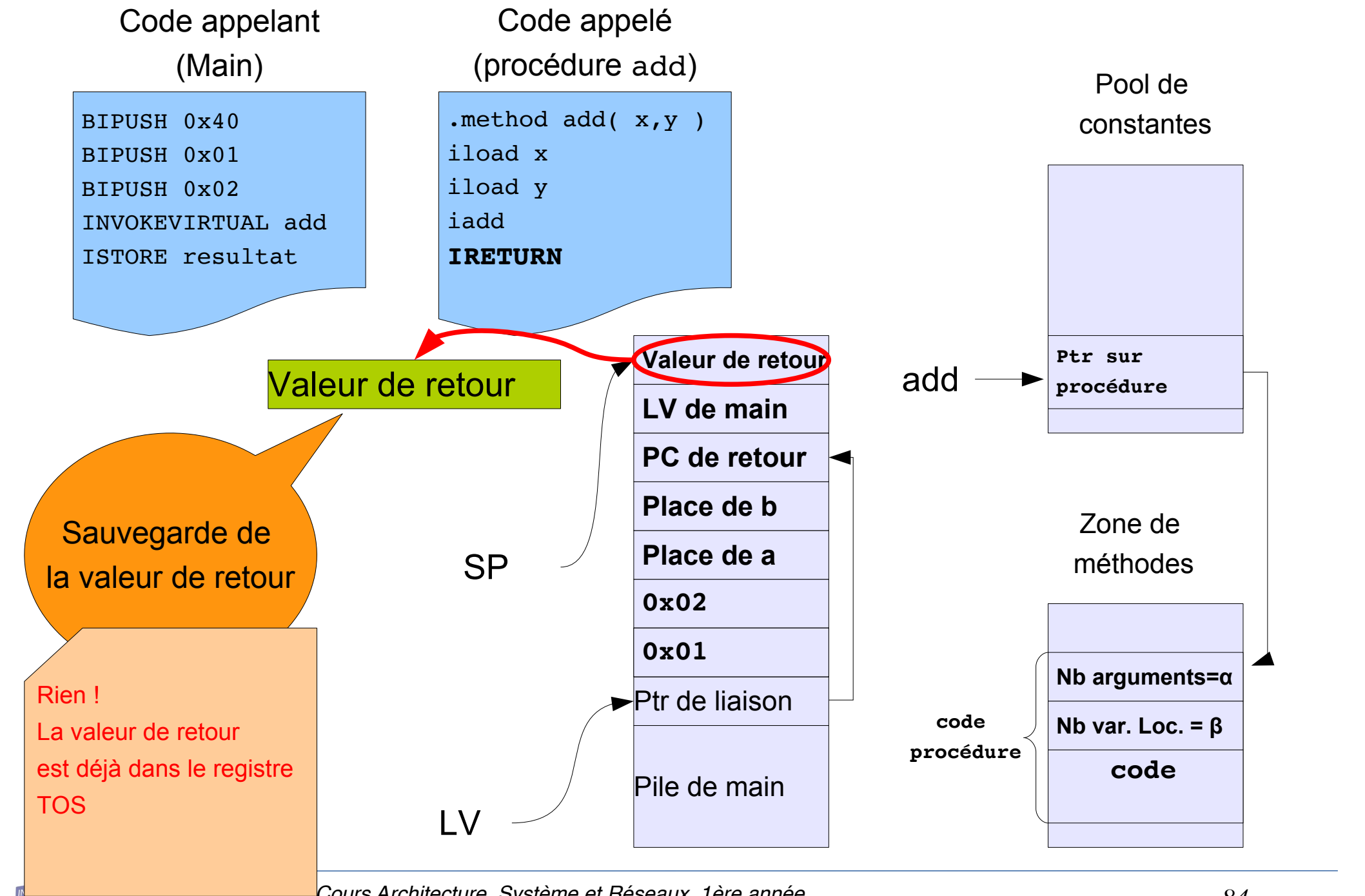
```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Mise en place
d'un nouveau
Pointeur de référence
pour les
Variables locales
de la procédure

```
PC = PC + 1; fetch
LV = TOS; goto Main1
```



Fin de procédure: point vue du microprogramme



Fin de procédure: point vue du microprogramme

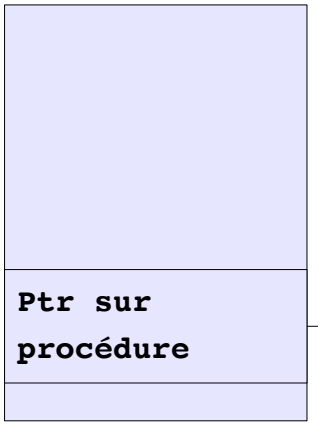
Code appelant (Main)

```
BIPUSH 0x40
BIPUSH 0x01
BIPUSH 0x02
INVOKEVIRTUAL add
ISTORE resultat
```

Code appelé (procédure add)

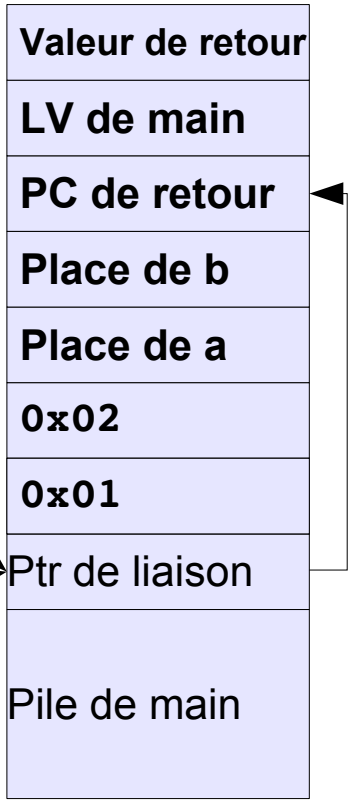
```
.method add( x,y )
iload x
iload y
iadd
IRETURN
```

Pool de constantes



add

Valeur de retour



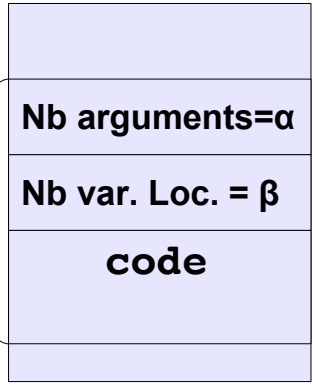
On utilise LV pour repositionner la

```
MAR = SP = LV; rd
Attente
LV = MAR = MDR; rd
MAR = LV + 1
PC = MDR; rd; fetch

MAR = SP
LV = MDR
MDR = TOS; wr; goto Main1
```

SP

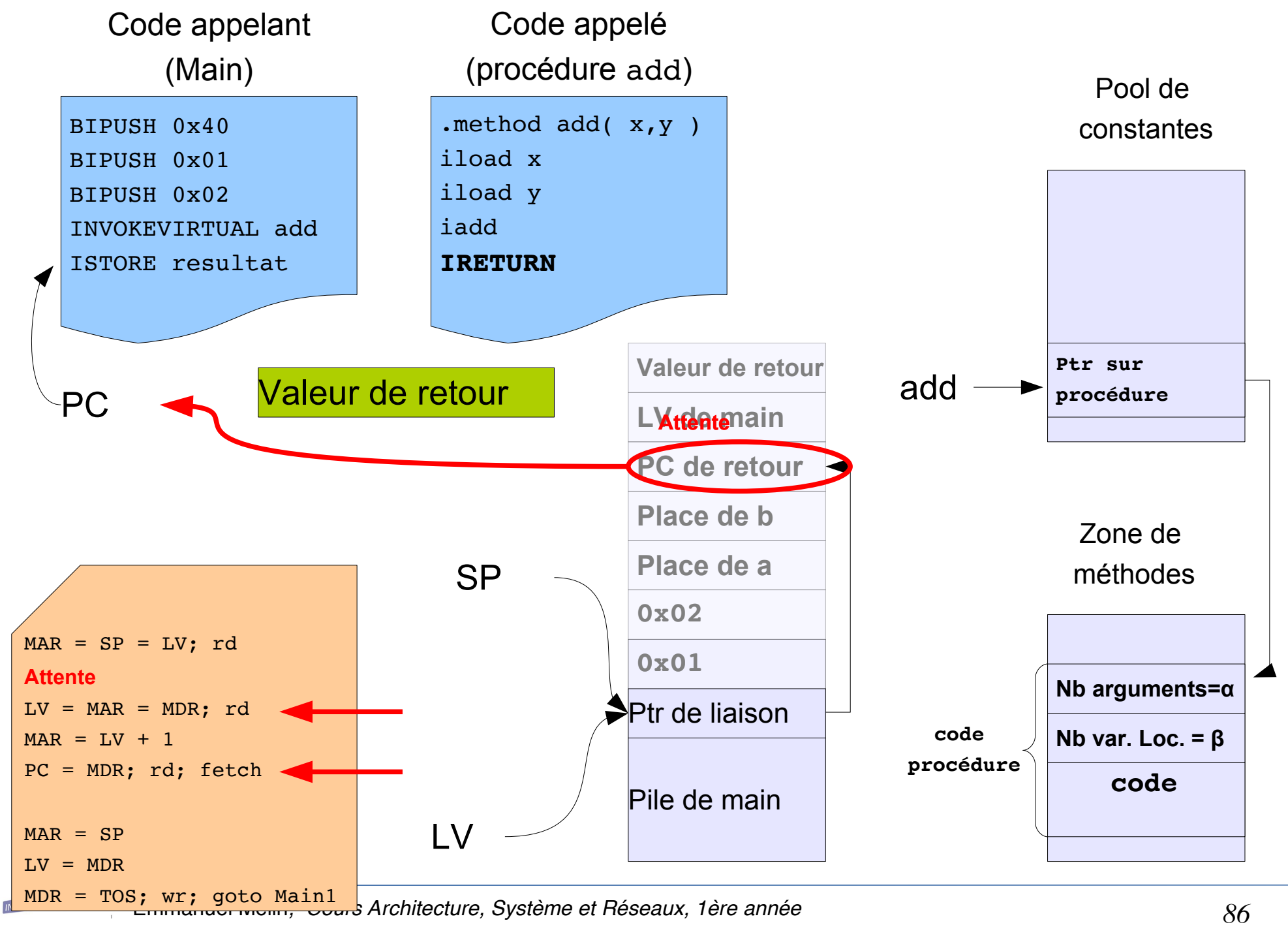
LV

Zone de méthodes

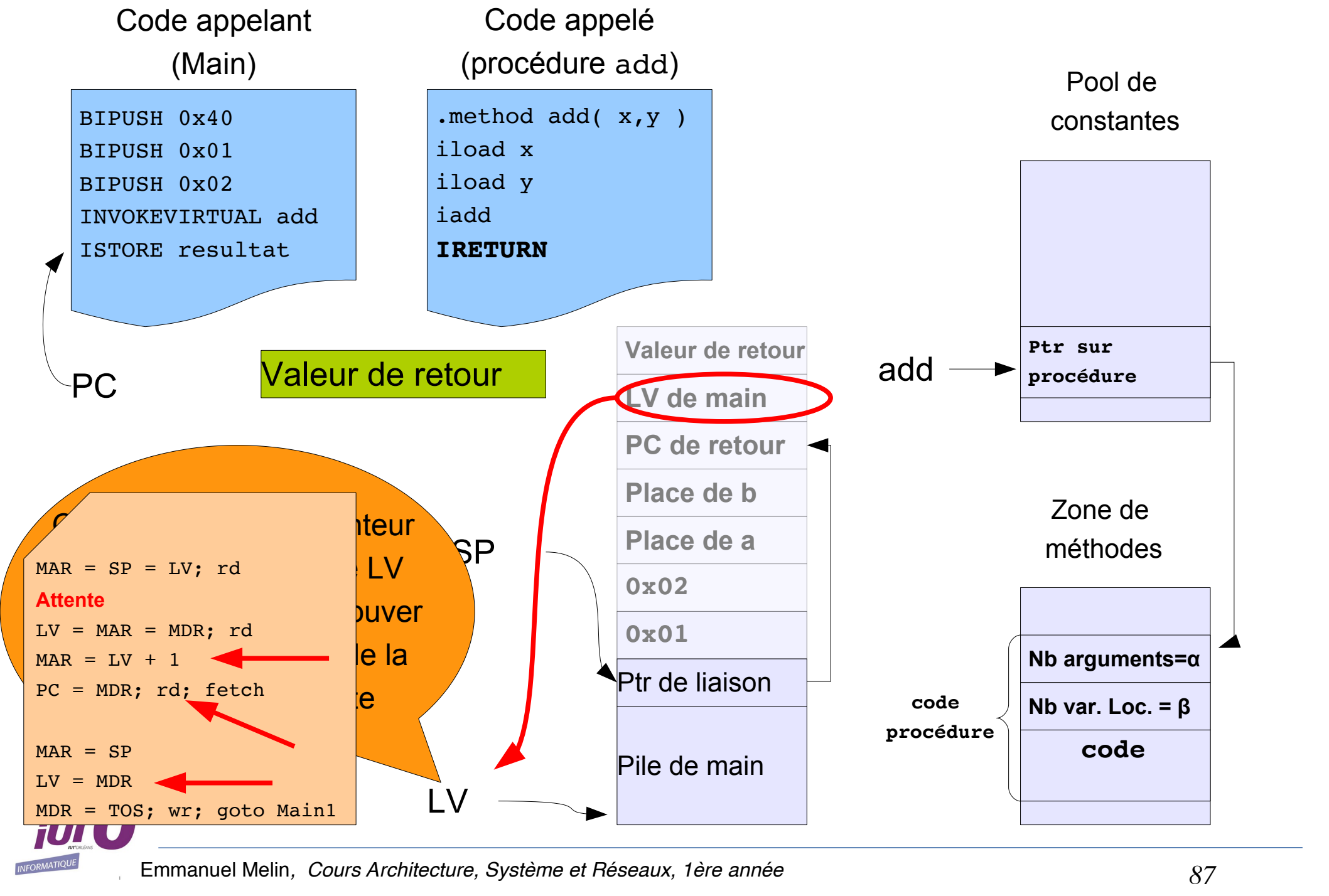


code procédure

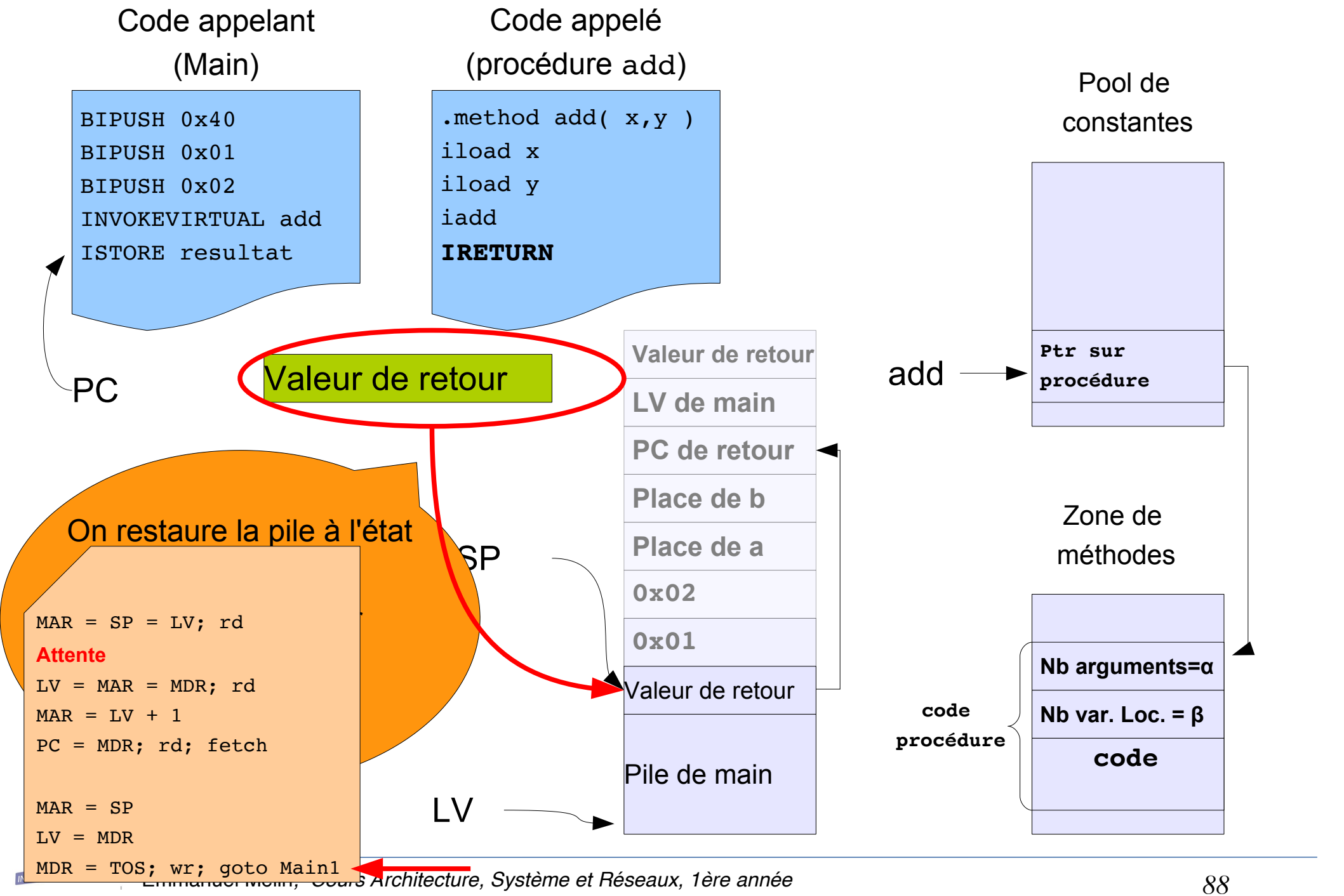
Fin de procédure: point vue du microprogramme



Fin de procédure: point vue du microprogramme



Fin de procédure: point vue du microprogramme



Exemple:

```
.main                          // start of program
.var                           // local variables for main
program
a
b
resultat
total
.end-var
start:
bipush 0x02
BIPUSH 0x40
BIPUSH 0x01
bipush 0x02
INVOKEVIRTUAL add
istore resultat
iload resultat
BIPUSH 0x30
IADD
OUT
.end-main
```

```
.constant
OBJREF 0x40
.end-constant

.method add( x,y )
.var
a
b
.end-var
iload x
iload y
iadd
IRETURN
.end-method
```