Architectures des microprocesseurs

Plan

Introduction générale : architecture en couches

Partie I : Structure générale d'une architecture

1. Structure générale d'une architecture
2. Organisation générale de l'unité centrale
3. Exécution d'une instruction
4. Exécution / Interprétation
5. Principes de conception RISC
6. Parallélisme d'instructions
7. Parallélisme du processeur
8. La mémoire principale
9. Principales architectures
    1. Pentium II
    2. UltraSparc II
    2. picoJava II

Partie II : La couche microarchitecture

1. Introduction
2. Exemple: Le modèle de l'IJVM
3. La micro-instruction
4. Le micro-programme
5. La micro-architecture MIC1

Partie III : La couche ISA

1. Propriétés de la couche ISA
2. Aperçu de la couche ISA du PENTIUM IV
3. Aperçu de la couche ISA de l'Ultra SPARC III
4. La couche ISA de l'IJVM

www.univ-rouen.fr/psi/paquet/

Introduction : Architecture en couches

1. Pour maîtriser la complexité d'un ordinateur…

- L'utilisateur souhaite faire Y mais l'ordinateur ne peut faire que X

- 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

- 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

Exemple:

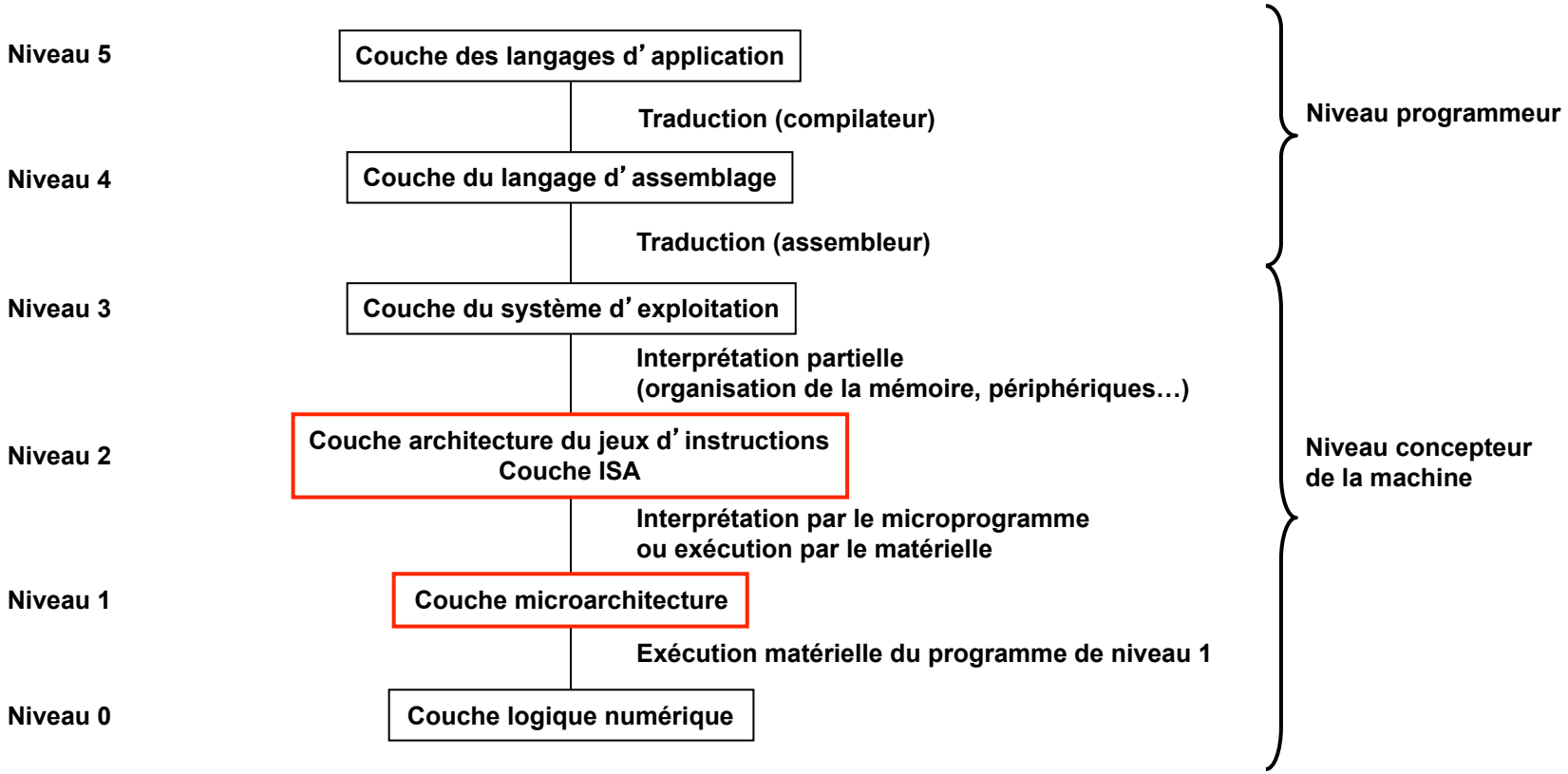
Pour réaliser des programmes exécutables sur toutes les plateformes, le langage java est compilé pour être exécuté sur une machine virtuelle: la JVM.

Il existe pour chaque plateforme des simulateurs de la JVM qui permettent d'exécuter un code binaire pour la JVM.

Il existe également des processeurs qui implémentent le jeux d'instruction de la JVM

Introduction : Architecture en couches

2. Architectures actuelles en six couches



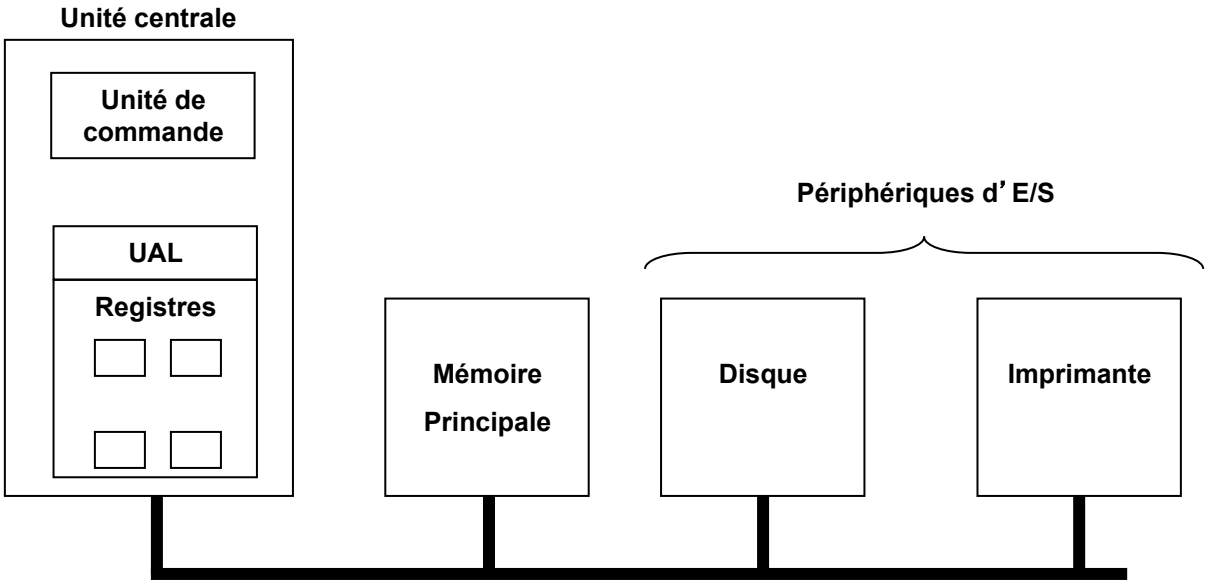
Partie I: Structure générale d'une architecture

1. Structure générale d'une architecture

Machine de Von Neuman

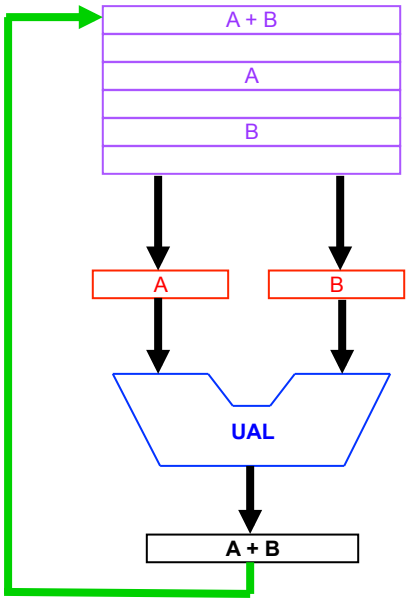
- arithmétique binaire
- programme numérique rangé en mémoire

première machine : EDSAC



2. 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

3. Exécution d'une instruction

Cycle CDE : Chargement Décodage Exécution

1. Charger la prochaine instruction à exécuter depuis la mémoire dans le registre instruction
2. Modifier le compteur ordinal pour qu'il pointe sur l'instruction suivante
3. Décoder l'instruction chargée
4. Localiser les données nécessaires pour l'exécution de l'instruction
5. Charger les données nécessaires dans les registres de l'UC
6. Exécuter l'instruction
7. Revenir à l'étape 1

4. Exécution / Interprétation

1. **Exécution du cycle CDE par le matériel : Exécution**
   - Rapide
   - Coûteux si on veut exécuter des instructions complexes
   - Architectures peu évolutives (pas de compatibilité)

2. **Exécution du cycle CDE par un programme : Interprétation**
   - Le matériel exécute l'interpréteur
   - Un interpréteur traite les instructions complexes de sa machine cible par petites étapes.
   - L'architecture sur laquelle tourne l'interpréteur est plus simple que l'architecture de la machine cible qui exécuterait matériellement les instructions du langage
   - compatibilité, évolutivité entre versions

4. Exécution / Interprétation

3. Un peu d'histoire

- 1951 Wilkes propose la technique de l'interprétation
- Architecture 360 d'IBM : famille d'ordinateurs tous compatibles entre eux mais aux performances et aux prix très différents.
- On peut ajouter des instructions complexes sans toucher au matériel (sans les contraintes du matériel)
- structure efficace de développement des instructions complexes
- au cours des années 70 le coût l'emporte sur la performance
- L'interprétation permet d'inonder le marcher avec des ordinateurs de faible coût
- les jeux d'instruction deviennent de plus en plus complexes
- les architectures matérielles restent simples mais ce sont les interpréteurs qui deviennent de plus en plus complexes (Motorola 68000)
- CISC = ***Complex Instruction Set Computer*** = 200 instructions
- durant cette période il existe des mémoires de micro-programme rapides (ROM). plus rapide que l'accès en RAM.
- 1982 David Patterson reprend les idées de Seymour Cray développées pour les calculateurs rapides et propose une architecture à jeux d'instructions réduit sans interprétation RISC = ***Reduced Instruction Set Computer*** = 50 instructions
- les mémoires RAM rapides accélèrent l'intérêt des architectures RISC
- pour des raisons de compatibilité on voit apparaître des architecture hybrides CISC + RISC (Intel 486)

5. Principes de conception RISC

RISC Design principles

- **Traitement par des composants matériels**
  - **Traiter directement les instructions simples au niveau du matériel (RISC).**
  - **Si nécessaire les instructions complexes sont exécutées par une deuxième unité fonctionnelle de type CISC**

- **Maximiser l'exécution des instructions**
  - **Maximiser le nombre d'instructions exécutées par seconde**
  - **Million d'Instructions Par Seconde = MIPS**
  - **Parallélisme d'instructions**
  - **Différer l'exécution de certaines instructions quand les ressources ne sont pas disponibles**

- **Décodage simple des instructions**
  - **Format d'instruction le plus régulier possible**
  - **Format fixe avec faible nombre de champs**

5. Principes de conception RISC

RISC Design principles

- **Limiter l'accès à la mémoire principale**
  - **L'accès à la mémoire principale reste plus lent**
  - **Augmenter le nombre de registres pour accélérer l'accès aux opérandes et la copie des résultats**
  - **Traiter parallèlement les échanges mémoire/registres par des instructions spécifiques *LOAD* et *STORE***

- **Nombre important de registres**
  - **Réduire le nombre de références à la mémoire principale**
  - **Un mot chargé dans un registre y reste aussi longtemps que nécessaire**
  - **32 registres semble un minimum**

6. Parallélisme d'instructions

6.1 Technique du pipeline

- Aller plus vite sans changer de technologie
- Diminuer le temps apparent de recherche d'instruction en mémoire (technique proposée dès 1959 chez IBM)
- Aller chercher les instructions en mémoire avant d'en avoir besoin
- Buffer d'instructions en file d'attente : *prefetch buffer*
- C'est un mécanisme d'anticipation en plusieurs phases
- Chaque phase est exécutée par une unité matérielle spécifique
- Chaque unité matérielle s'exécute simultanément aux autres
- Unité fonctionnelle = étage du pipeline
- TEMPS DE LATENCE

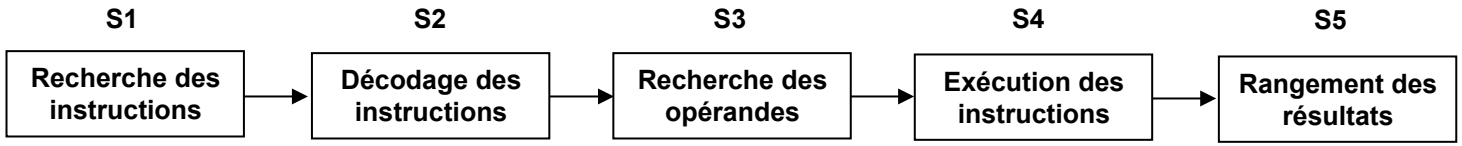
  = temps nécessaire pour exécuter une instruction

  = temps nécessaire pour amorcer le pipeline avant que tous les étages ne fonctionnent en parallèle

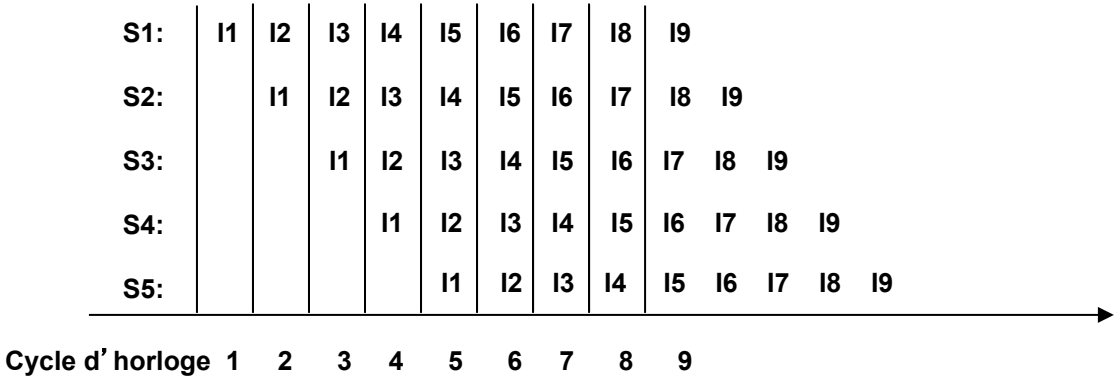
Le parallélisme d'instructions permet des gains d'un facteur 10

6. Parallélisme d'instructions

- **Technique du pipeline : exemple à 5 étages, Intel 486**



– Organisation dans le temps



– Analogie avec le montage à la chaîne

– Il faut 5 cycles pour que l'instruction soit exécutée complètement (temps de latence)

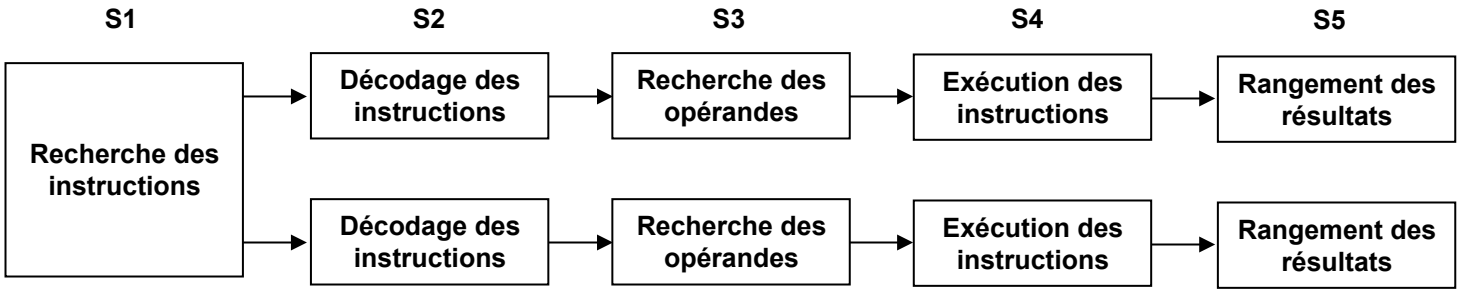
– Pour un temps de cycle de 2ns il faut donc 10ns pour exécuter une instruction

– On fonctionnerait en principe à 100 MIPS, mais le pipeline introduit 5 étages parallélisés.

– **Finalement on obtient 500 MIPS grâce au pipeline**

6. Parallélisme d'instructions

6.2. Architecture à double pipeline



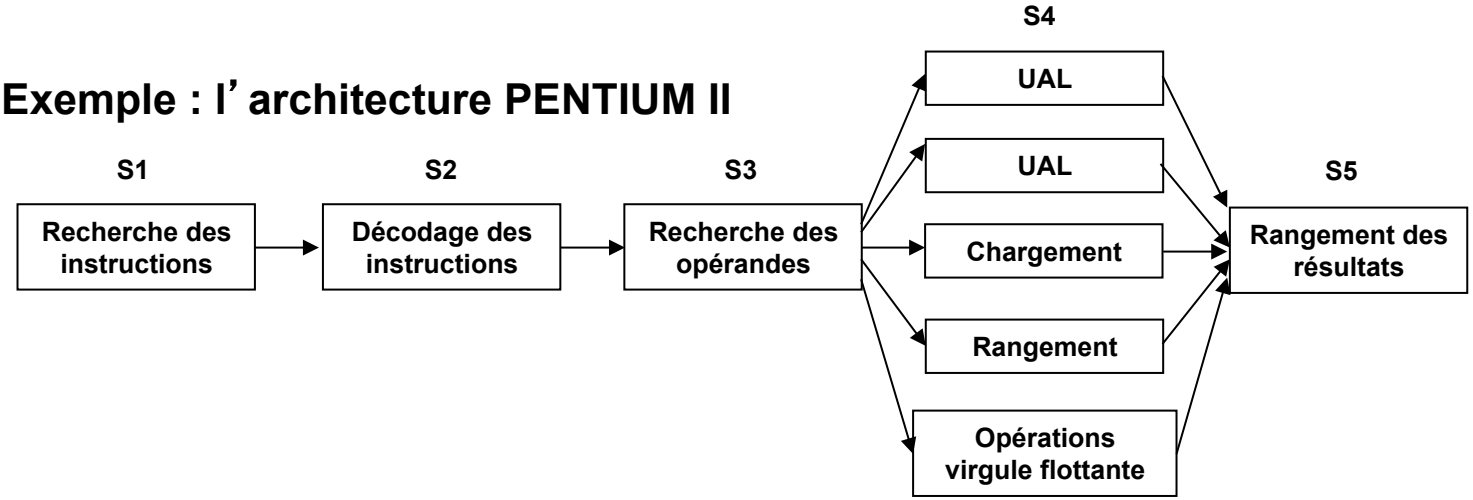
- L'unité S1 place les instructions exécutables en parallèle dans l'un ou l'autre des deux pipeline si elles ne sont pas dépendantes l'une de l'autre (rôle du compilateur)

- Pentium : double pipeline
  - Pipeline principal *u* exécute n'importe quelle instruction
  - Pipeline secondaire *v* n'exécute que des instructions simples
  - Le compilateur du PENTIUM effectue un ordonnancement dynamique complexe des instructions pour prendre en compte
    - les instructions incompatibles
    - Les instructions qui ne peuvent être placées sur le pipeline v et qui doivent être retardées
  - Gain de 2 entre 486 et PENTIUM

6. Parallélisme d'instructions

- **6.3. Architecture superscalaire**
  - **Problème de l'architecture en pipeline:**
    les temps d'exécution de l'étage 3 est parfois plus rapide que celui de l'étage 4.
  - **Solution: On met dans le pipeline des unités de traitement en parallèle au niveau de l'étage 4. C'est un début de parallélisme de traitement au niveau processeur**
  - **Mise en œuvre dans les années 70 pour la première fois**

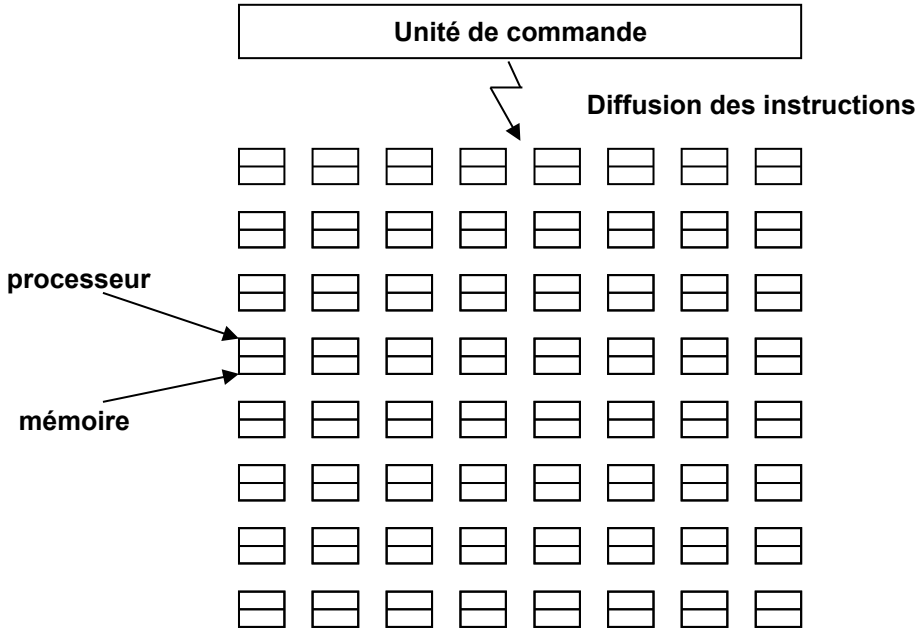
  - **Exemple : l'architecture PENTIUM II**



7. Parallélisme du processeur

Le parallélisme du processeur permet des gains d'un facteur > 50

7.1. Processeurs matriciels : ILLIAC IV 1972



Architecture dédiée aux calculs scientifiques réguliers pouvant être réalisés au même moment sur des données différentes

Les données vectorielles ou matricielles sont distribuées sur la **matrice de processeurs.** **Coût !!**

7. Parallélisme du processeur

7.2. Processeurs vectoriels : Cray I 1974

Une seule unité de calcul en pipeline travaille sur un tableau de données vectorielles ou matricielles.

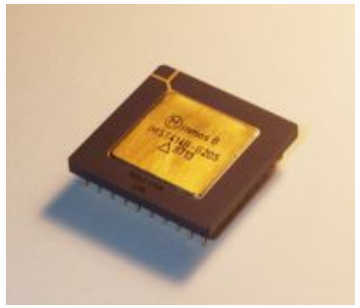
Registre vectoriel: Ensemble de registres simples chargés en une seule instruction

Une addition vectorielle réalise l'addition du contenu de deux registres vectoriels et sauvegarde le résultat dans un registre vectoriel

7. Parallélisme du processeur

7.3. Multiprocesseur: Transputer 1983

Contrairement aux architectures matricielles où les calculs sont sous le contrôle d'une seule unité de commande, les architectures multi-processeurs réalisent un **vrai réseau de processeurs autonomes communiquant entres eux** par **des liaisons rapides**



Mémoire privée à chaque processeur

Mémoire commune à l'ensemble

La communication devient rapidement un problème!!!

Langage de programmation spécifique: **Occam de la société Inmos**

7. Parallélisme du processeur

7.4. Parallélisme d'ordinateurs

- Mise en relation via le réseau d'un ensemble de machines échangeant des informations entres elles

- L'acheminement des messages entre machines n'est pas immédiat, il est pris en charge par des contrôleurs.

-On peut atteindre des capacités gigantesques de plusieurs milliers de machines

Exemple : PVM (Parallel Virtual Machine) 1989

http://www.csm.ornl.gov/pvm/

« une machine Unix gigantesque »

supportée par des plateformes différentes (Unix, windows…)

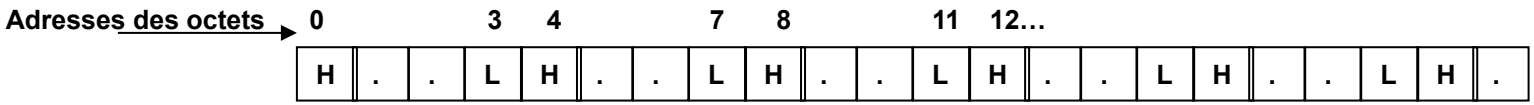
8. La mémoire principale

8.1. Organisation des octets

- L'octet est la plus petite quantité d'information adressable
- Les octets sont regroupés en mots
- le mot est l'unité d'information sur laquelle opèrent la plupart des instructions
- 2 organisations possibles des octets dans un mot

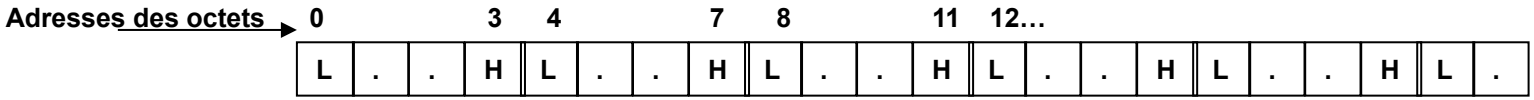
Gros-boutiste = *big endian* = Motorola, Sparc

Rangement en mémoire des octets de poids fort d'abord



Petit-boutiste = *little endian* = Intel

Rangement en mémoire des octets de poids faible d'abord



… l'absence d'une norme cause des problèmes de transferts

8. La mémoire principale

8.3. La mémoire cache

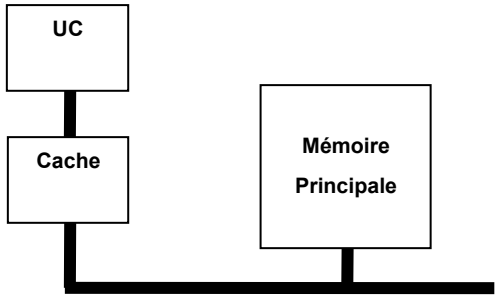
- Le temps d'accès à la mémoire principale reste très grand par rapport au temps de réaction du processeur
- La capacité de stockage a beaucoup évolué, mois le temps d'accès
- Temps de réaction mémoire = 2 à 3 cycles d'horloge

– 2 mauvaises solutions pour y remédier
  - Matérielle : bloquer le processeur après une instruction *load*
  - Logicielle : le compilateur insère des instruction *NOP* après une instruction *load*

– La solution retenue : ajout d'un cache
  - ajouter une mémoire rapide de plus faible capacité (coût)
  - elle doit contenir les références mémoire les plus récentes

8. La mémoire principale

8.3. La mémoire cache



- *c* est le temps d'accès au cache
- *m* le temps d'accès à la mémoire principale
- *h* le taux de présence ou de succès de lecture dans le cache

- alors le temps moyen d'accès aux données est: $t_{acc} = h\,c + (1-h)\,m$
  si toutes les données sont dans le cache alors $h=1$ et $t_{acc} = c$
- *m* le temps d'accès à la mémoire principale

– **Principe de gestion du cache**
  - principe de localité : pendant un court intervalle de temps on n'accède en générale qu'à une portion très localisée de la mémoire.
  - Organiser la mémoire cache en blocs : *lignes de cache*
  - On charge dans le cache des lignes entières plutôt que des mots : on réduit le nombre d'accès à la mémoire principale

– **Organisation du cache**
  - Cache de données et cache d'instructions : Architecture Harvard (parallélisme)
  - Cache unifié
  - Taille des lignes de cache
  - Cache principal sur la puce du processeur
  - Cache de niveau II à proximité immédiate de la puce, et de niveau III

9. Les principales architectures

9.1. Le Pentium II

- Processeur 32 bits, 7,5 Mo de transistors, Horloge > 233MHz, 50 Watt
- Entièrement compatible au niveau de la couche ISA avec
  I386, I486, Pentium, Pentium Pro

– *Une architecture physique différente :*

- Machine superscalaire
- 64 Go adressables par mots de 8 octets (33 bits d'adresse au lieu de 36)
- échange de mots de 64 bits en une seule instruction avec la mémoire
- Jeux d'instruction MMX

– *Existe en version bi-processeurs avec mémoire commune*

- technique d'espionnage pour la gestion des incohérences cache / mémoire

– *Décodage des instructions*

- les instructions ISA sont extraites de la mémoire par anticipation
- sont transformées en micro-opérations élémentaires de type RISC
- dès qu'une micro-opération dispose des ressources nécessaires elle est réalisée
- les micro-instructions sont exécutées avec un certain niveau de parallélisme

– *Organisation du cache*

- 16Ko de cache données et 16 Ko de cache d'instructions
- cache unifié de niveau 2 de 512 Ko
- Ligne de chache = 32 octets

– *Organisation des BUS externes*

- 1 BUS mémoire
- 1 BUS PCI pour les E/S

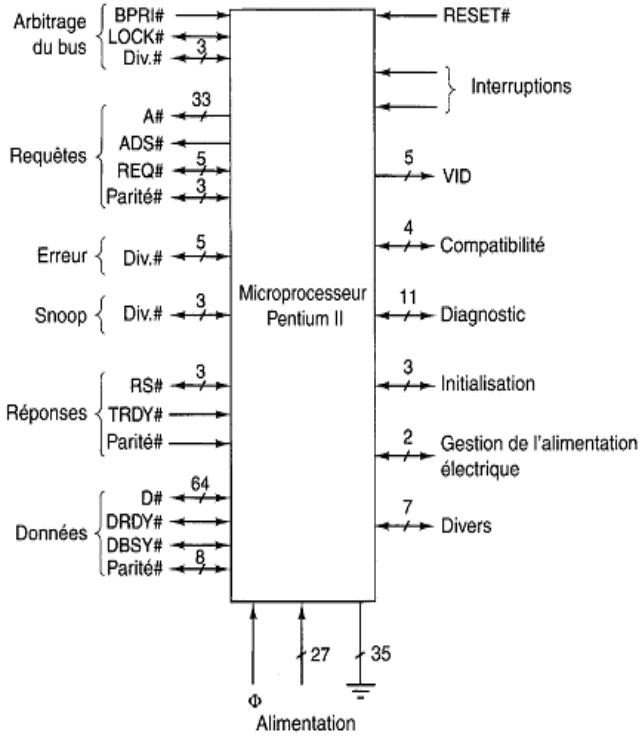
9. Les principales architectures

9.2. Technique de pipeline sur le BUS du Pentium II

- La cadence du processeur est nettement supérieure à la vitesse d'accès à la mémoire
- Nécessité d'optimiser le débit du BUS mémoire
- 8 transactions sont autorisées simultanément sur le BUS

• **Transaction** = demande d'accès à la mémoire
1 transaction se décompose en 6 phases

- **Arbitrage du BUS**
  Détermine lequel des maîtres potentiels sera le prochain utilisateur du BUS
- **Requête**
  autorise le positionnement d'un adresse sur le bus d'adresse et et la génération d'une requête à la mémoire
- **Compte rendu d'erreur**
  permet à un esclave d'indiquer l'apparition d'une erreur lors d'une transaction
- **Snooping (espionnage)**
  permet à un processeur de vérifier la cohérence d'une donnée en cache (multi-proc)
- **Réponse**
  permet au maître de savoir si l'esclave a accédé à la donnée demandée
- **Données**
  autorise l'envoi d'une donnée sur le BUS



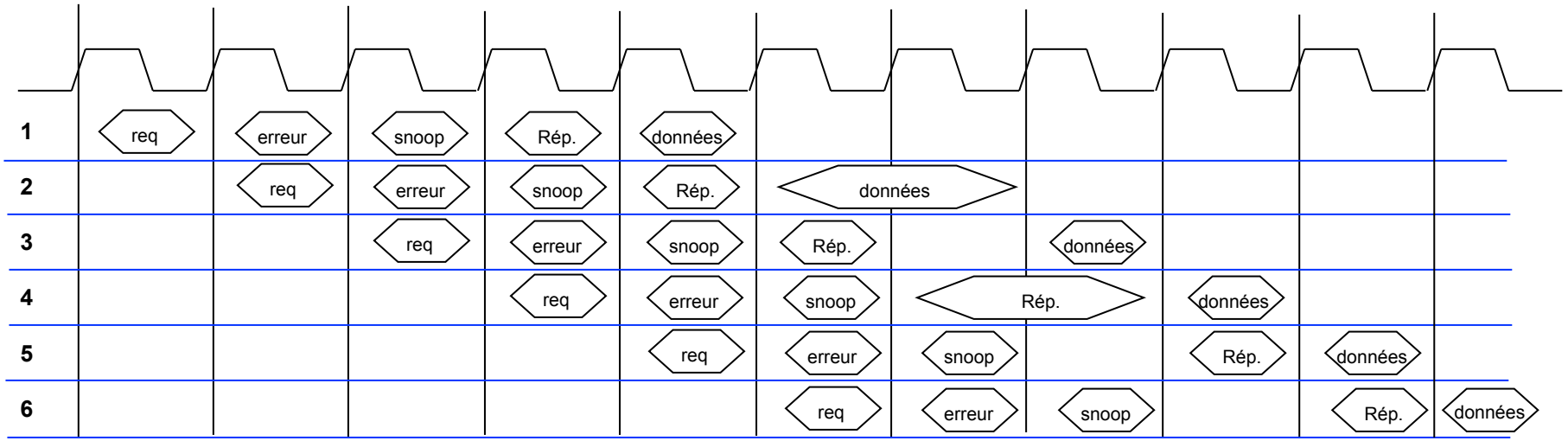
9. Les principales architectures

9.2. Technique de pipeline sur le BUS du Pentium II

– *6 groupes de signaux sont affectés aux 6 phase du pipeline*

- Les groupes de signaux sont indépendants les uns des autres ce qui autorise une parallélisation (pipeline)

Exemples de transactions



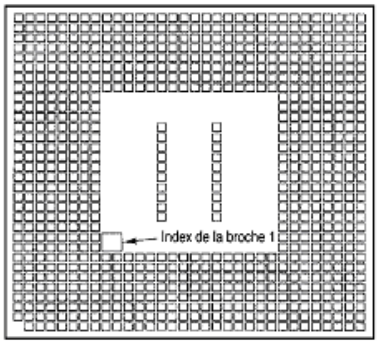
La phase d'arbitrage du BUS n'apparaît pas quand le processeur est seul et pas d'E/S

- La transaction 2 introduit un accès plus long aux données
- Donc la transaction 3 doit attendre la fin de la phase de données de la transaction 2
- Lors de la transaction 4 la réponse prend aussi plus temps
- Elle retarde la phase de réponse de la transaction 5
- Qui retarde elle-même l'entrée en phase de réponse de la transaction 6 (phénomène de propagation d'une « bulle »)

9. Les principales architectures

9.3. Le processeur UltraSPARC II

- Famille des processeurs RISC 64 bits SPARC de SUN microsystèmes
- UltraSPARC II a été conçu pour s'intégrer dans un système multi-processeur à mémoire partagée
- SUN conçoit les puces mais les fait réaliser contrairement à Intel
- 5,4 Mo de transistors



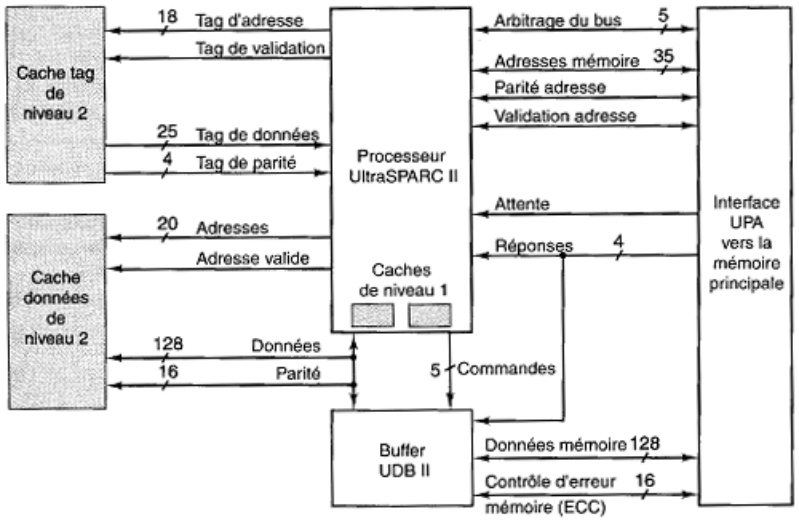
Principales caractéristiques

- BUS d'adresses de 64 bits
- BUS de données de 128 bits : le SBus à 25MHz (très lent)
- Contrôle de Bus rapide *Ultra Port Architecture* UPA
- cache d'instruction de 16Ko, cache de données de 16Ko
- Cache externe unifié entre 16 et 512Ko (plus facile à réaliser mais plus lent qu'un cache intégré de niveau II du Pentium)
- Un buffer UDB II (UltraSPARC Data Buffer II) qui désynchronise le processeur et l'accès à la mémoire principale
- La mémoire est organisée en lignes de 16 octets (comme le cache)
- Les 256 lignes de données les plus référencées sont dans le cache niveau 1
- Idem pour les 256 instructions les plus fréquemment utilisées

9. Les principales architectures

9.4. L'accès à la mémoire pour l'UltraSPARC II

- Gestion du cache de niveau II
  - Le cache tag contient les index des lignes mémoire présentent dans le cache de niveau II ainsi que leur type (donnée ou instruction)
  - Si la donnée est présente dans le cache de niveau II le tag renvoie au processeur l'adresse en cache de la donnée recherchée – Le processeur sollicite ensuite le cache de niveau II – Il faut 4 cycles d'horloge pour amener une ligne mémoire du cache II au cache I

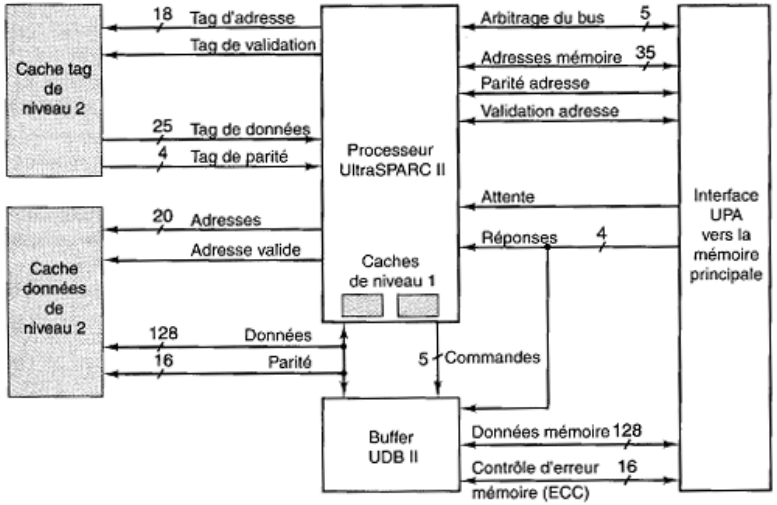


9. Les principales architectures

9.4. L'accès à la mémoire pour l'UltraSPARC II

– L'interface UPA

- **Si la donnée n'est pas présente dans le cache de niveau II le processeur s'adresse à l'interface UPA de la façon suivante:**
- **1 demande d'accès au bus (arbitrage)**
- **2 place l'adresse de ligne demandée (R ou W)**
- **3 les lignes venant de la mémoire sont placées dans le buffer UDB II**
- **4 L'écriture en mémoire est également gérée par le buffer UDB II**
- **L'asynchronisme de UDB II avec le processeur permet permet un gain de temps**
- **UDB II peut gérer plusieurs transactions avec le processeur**
- **UDB II gère deux flux simultanément : lecture et écriture**



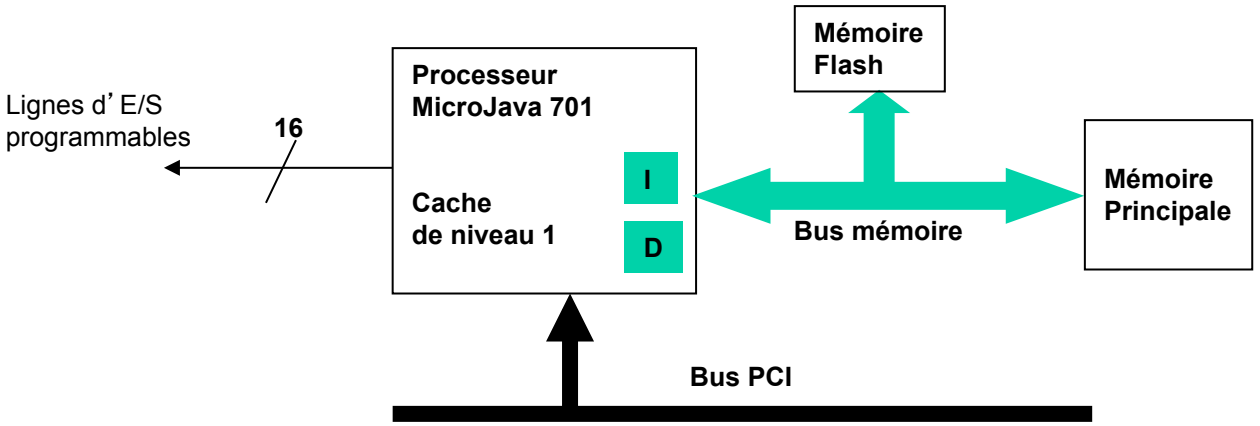
9. Les principales architectures

9.5. La machine PicoJava

- Conçu pour les systèmes embarqués ou enfouis
  - **Aller au-delà du simple micro-contrôleur : Notebook, PDA, Téléphone, imprimante, caméscope**
  - **Les appliances = systèmes à usages divers**
  - **Un problème de coût, pas de performance**
  - **Leur mise au point nécessite d'utiliser des langages de haut niveau (et pas l'assembleur)**
  - **Le langage Java est de plus en plus utilisé : code généré est réduit et relativement indépendant d'une machine cible**
  - **Mais nécessite un interpréteur Java assez volumineux. D'où un affaiblissement des performances**
  - **SUN et d'autres société ont conçu un processeur exécutant directement le code JVM**
  - **Les jeux d'instructions est le jeux des instructions JVM**

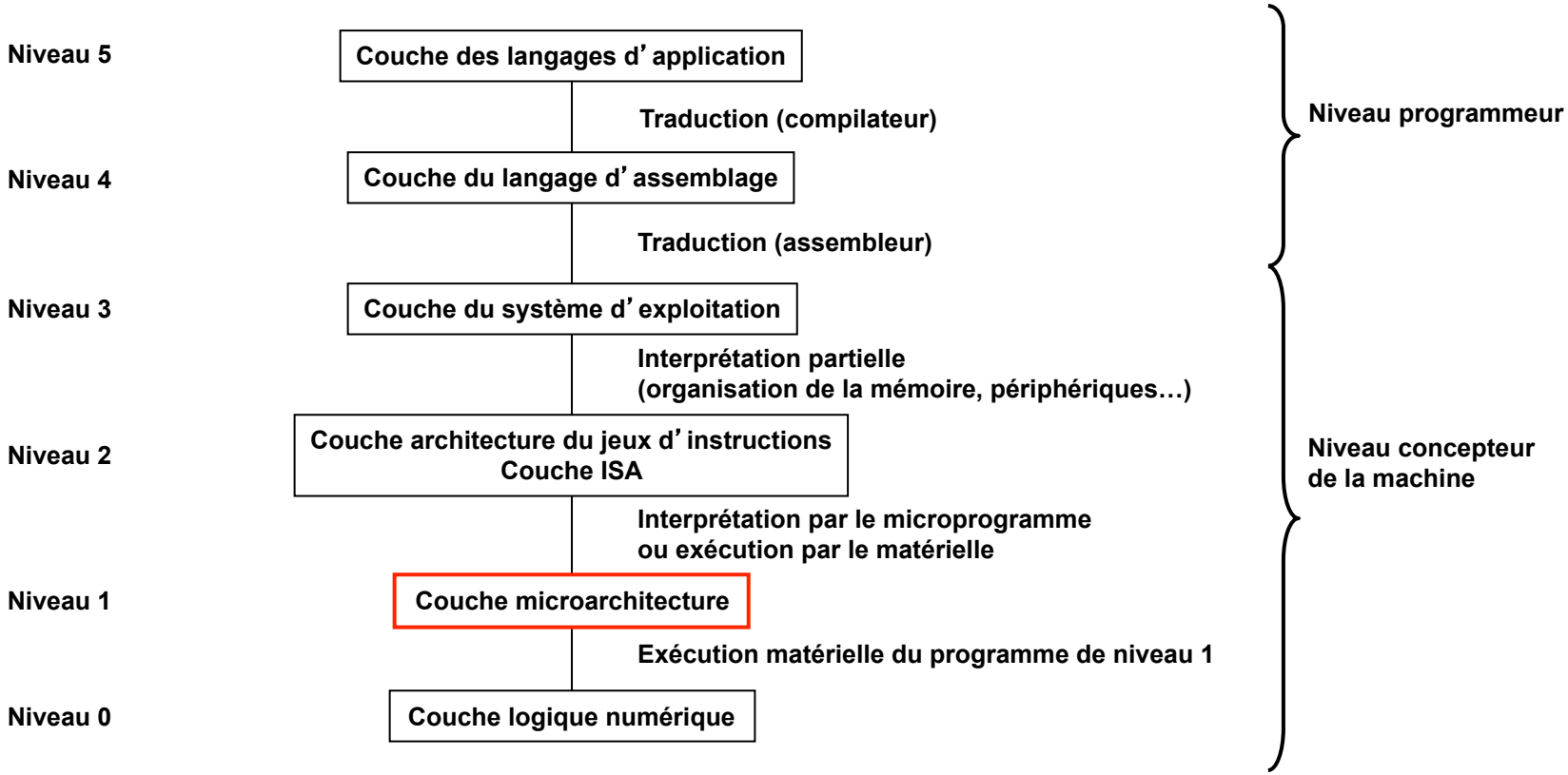
9. Les principales architectures

9.5. La machine MicroJava 701



- **Caractéristiques principales**
  - **Bus mémoire 64 ou 32 bits**
  - **Deux caches intégrés de 16 Ko chacun**
  - **Pas de cache externe de niveau II**
  - **2 millions de transitors pour le processeur**
  - **1,5 millions de transistors pour les deux caches en option**
  - **Bus PCI à 33 ou 66 MHz (Bus Intel reconnu comme standard par les industriels)**
  - **Mémoire Flash pour y placer le programme fonctionnel lié à l'appliance**
  - **16 lignes d'E/E reliées à des boutons, clavier, etc…**
  - **Intègre 3 compteurs programmable (temporisations)**
  - **Une architcture CISC**

Partie II: La couche microarchitecture



1. Introduction

1. 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

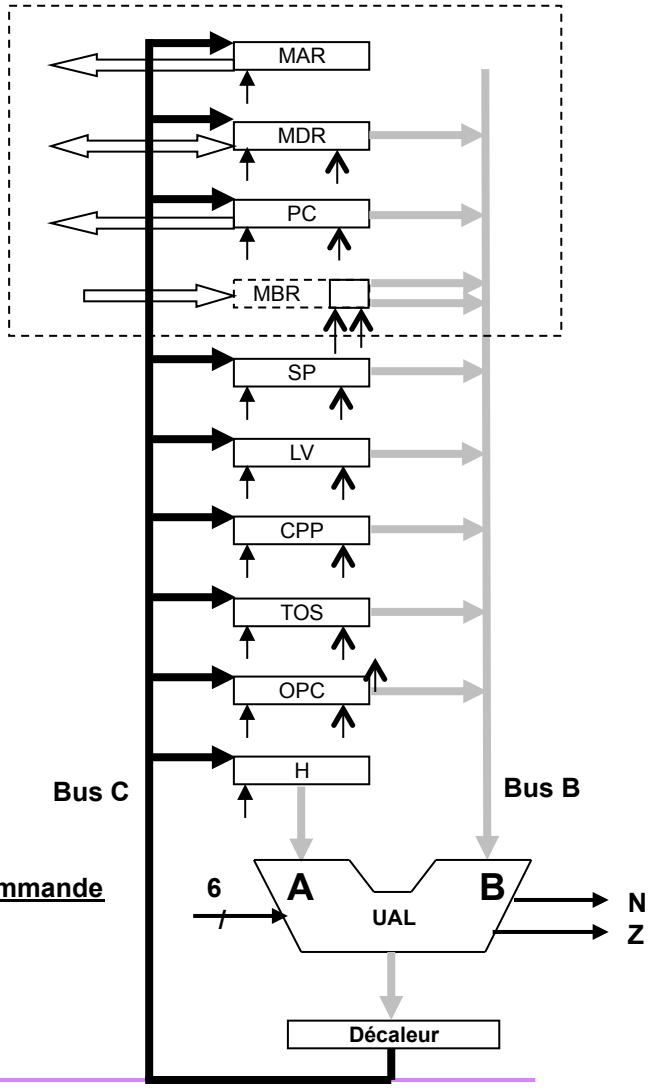
2. Approche

- 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)*

2. Modèle d'exécution de l'IJVM

2.1. 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 maintient



Signaux de commande des registres

Ecriture sur B
Lecture sur C

Signaux de Commande de l'UAL

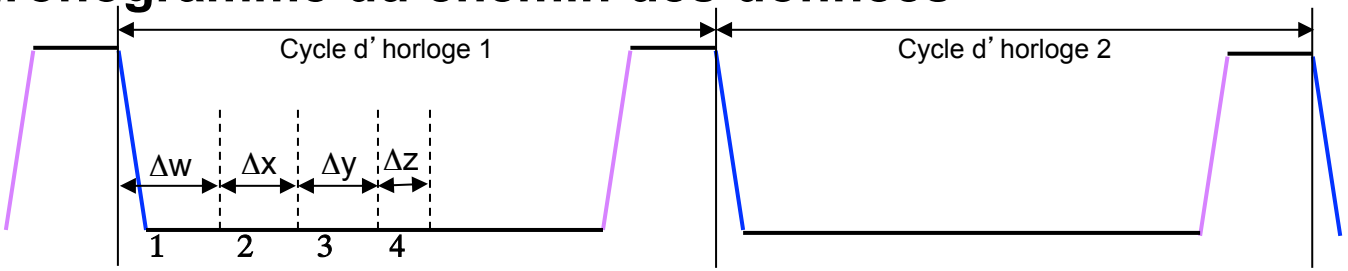
2. Modèle d'exécution de l'IJVM

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	1
- 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	0	1	0	0	0	1
- 1	0	1	0	0	1	0

2. Modèle d'exécution de l'IJVM

2.2. 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

- 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

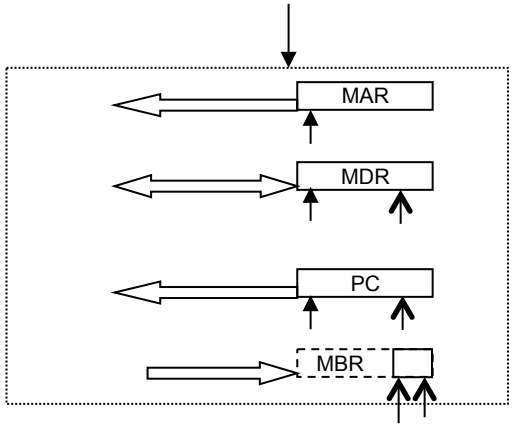
Les sous-cycles sont implicites ils ne représentent qu'un enchaînement d'actions et ne sont pas cadencés par une horloge. La difficulté est d'assurer que le temps global du cycle d'horloge soit supérieur à Δw + Δx + Δy + Δz

2. Modèle d'exécution de l'IJVM

2.3. 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**

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

2. Modèle d'exécution de l'IJVM

2.3. 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

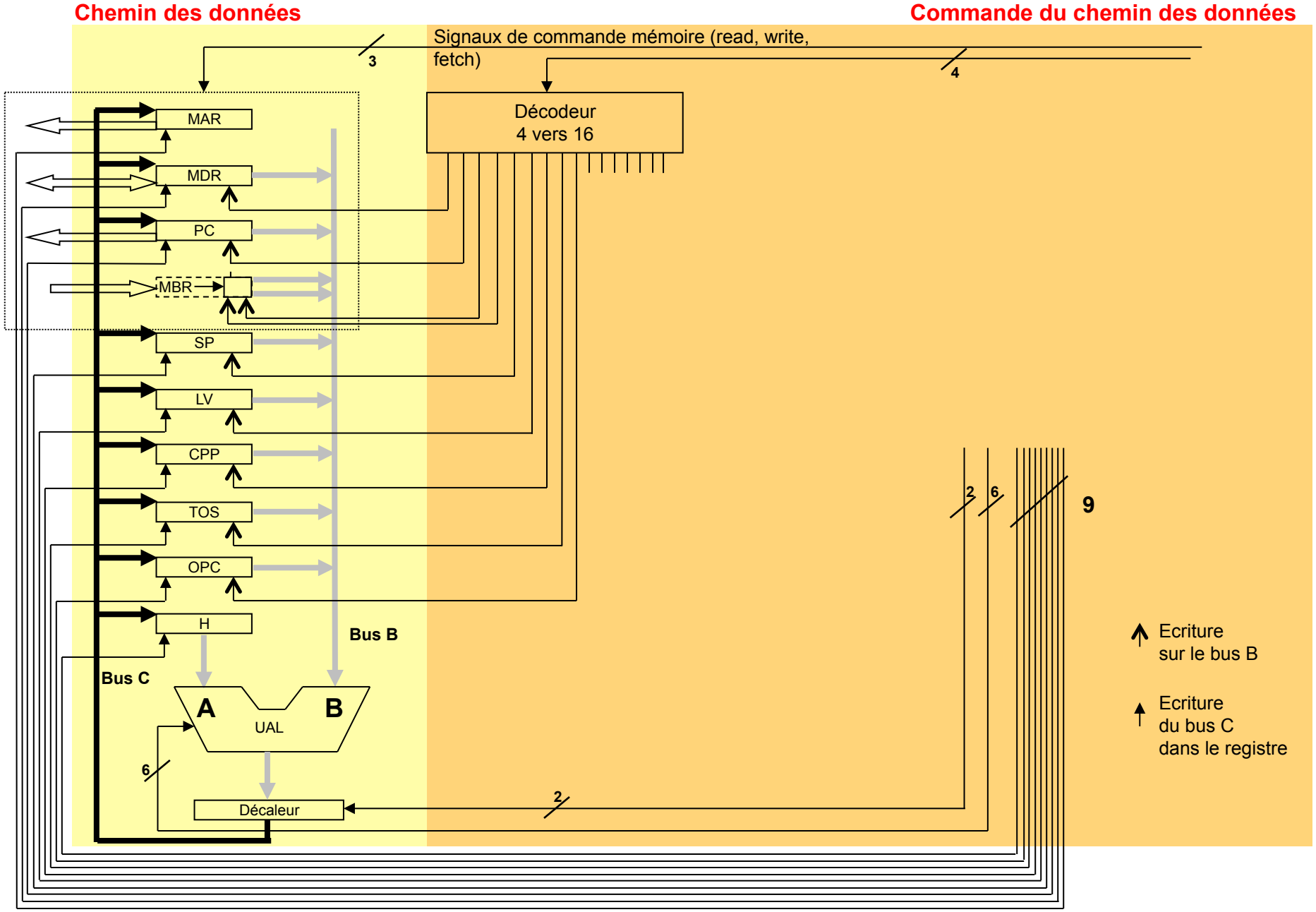
L'ensemble PC / MBR permet de lire le programme exécutable de la couche ISA

3. La micro-instruction

3.1. 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 (decallage gauche / droite)
  - 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!

5. La micro-architecture MIC1



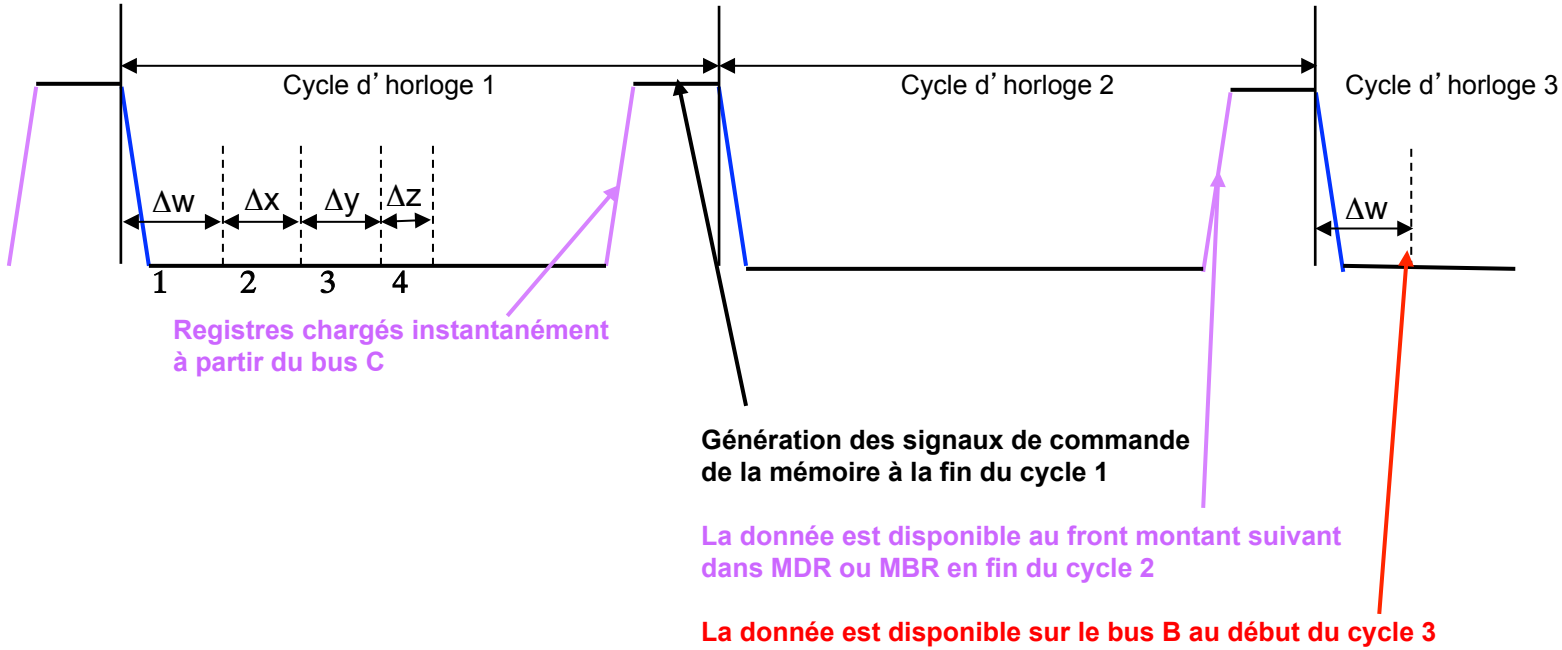
3. La micro-instruction

3.2. 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!!

3. La micro-instruction

3.2. 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!

3. La micro-instruction

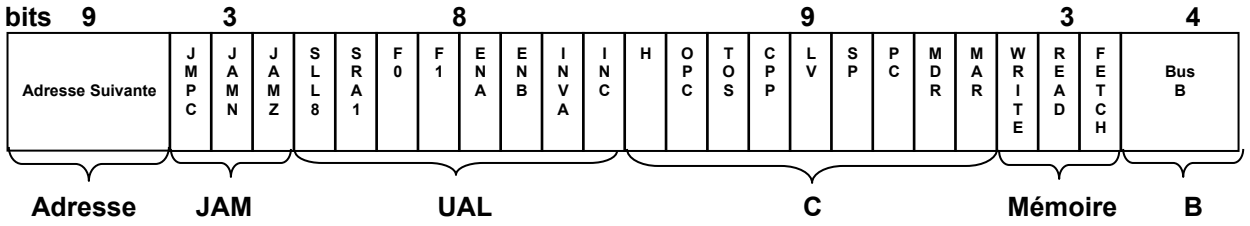
3.3. Format de la micro-instruction

- **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

4. Le micro-programme

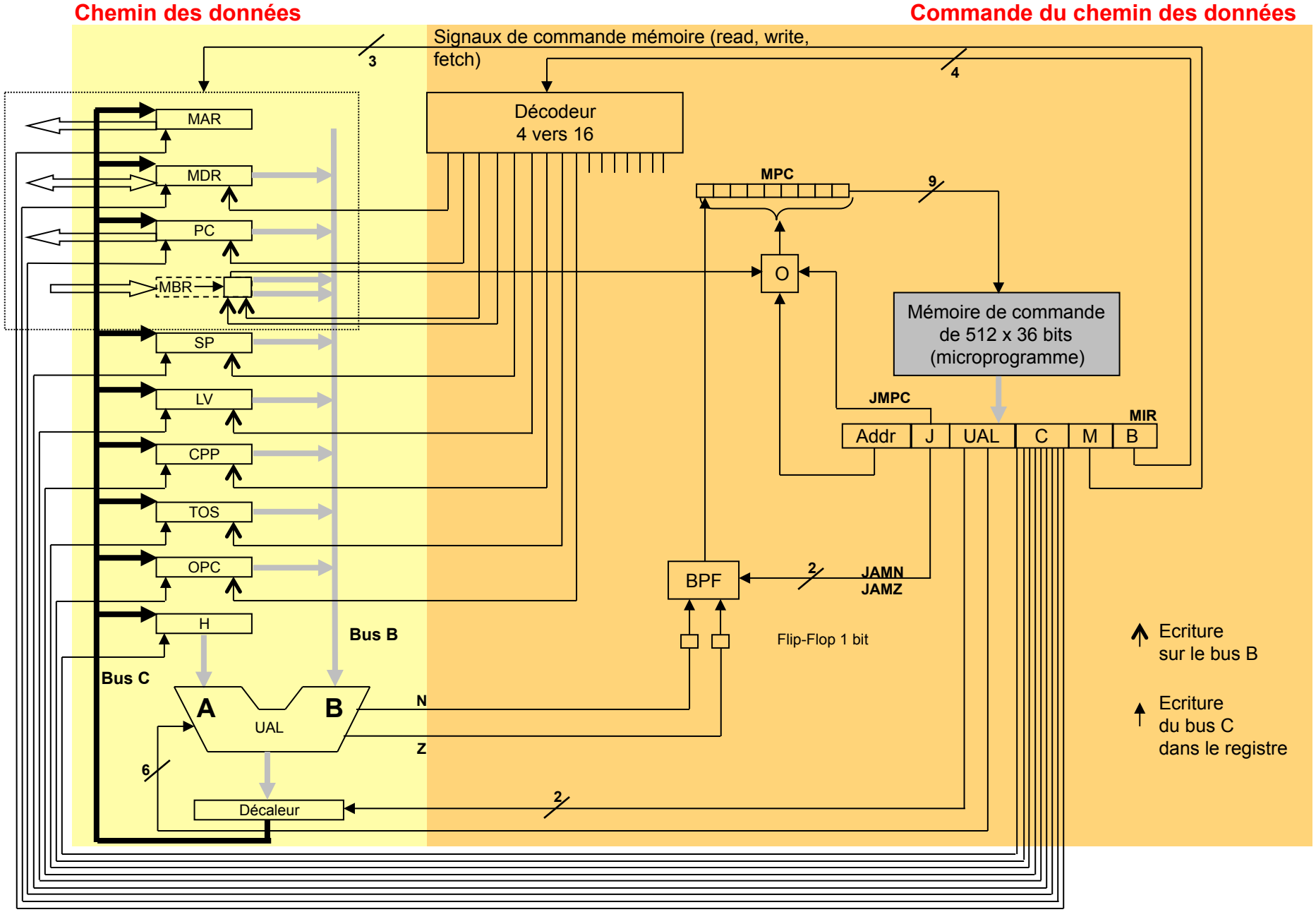
4.1 Rôle du micro-programme

- 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*).

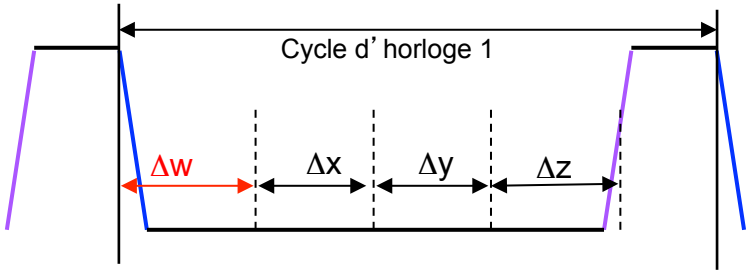
4.2 La mémoire de micro-programme

- 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

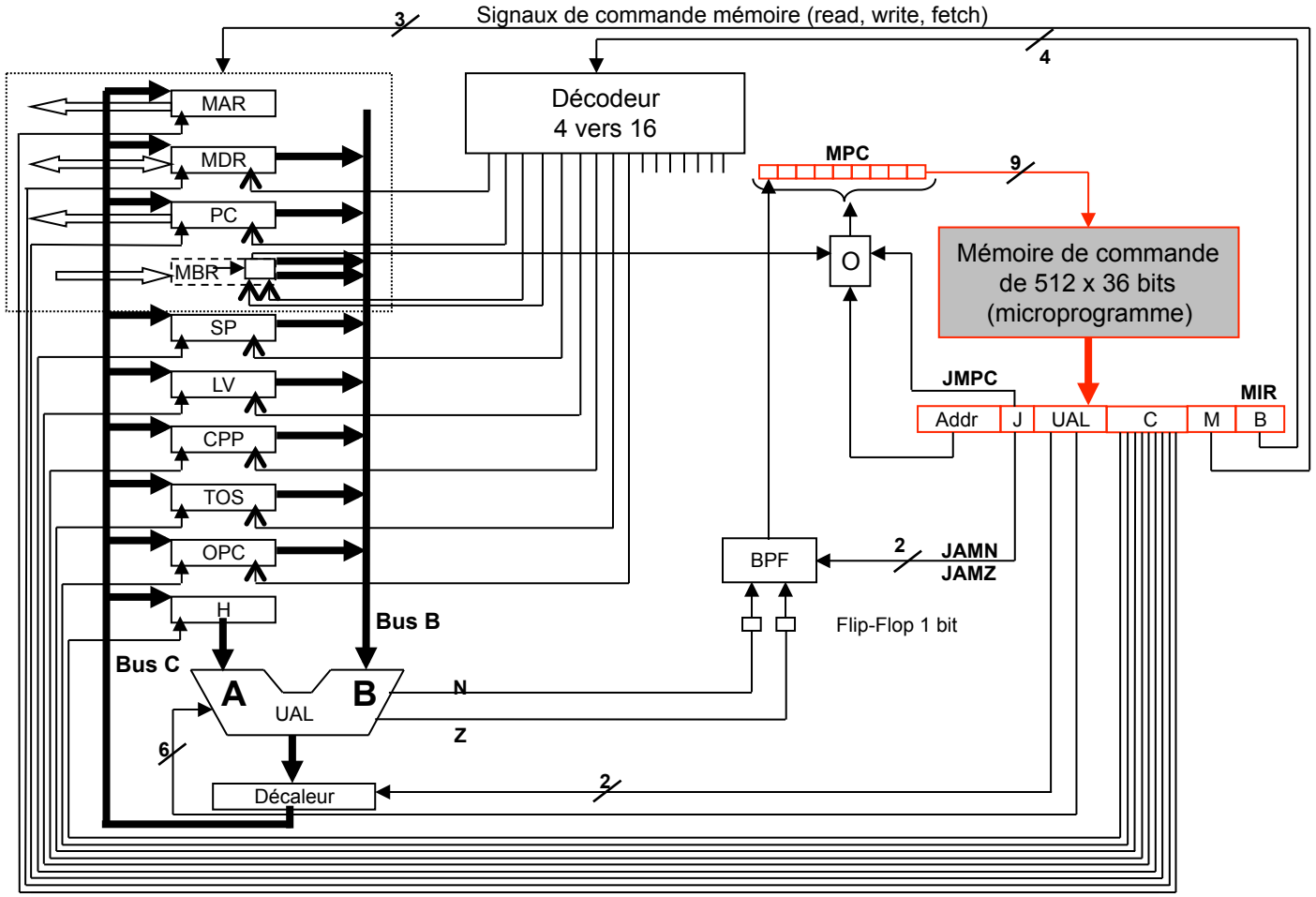
5. La micro-architecture MIC1



5.1 Fonctionnement de la micromachine



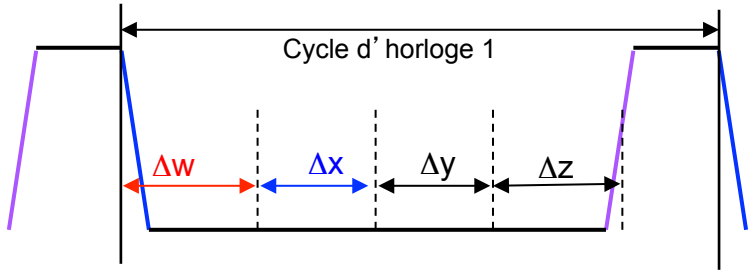
Sous-cycle 1
- MIR est chargé avec la micro instruction pointée par MPC



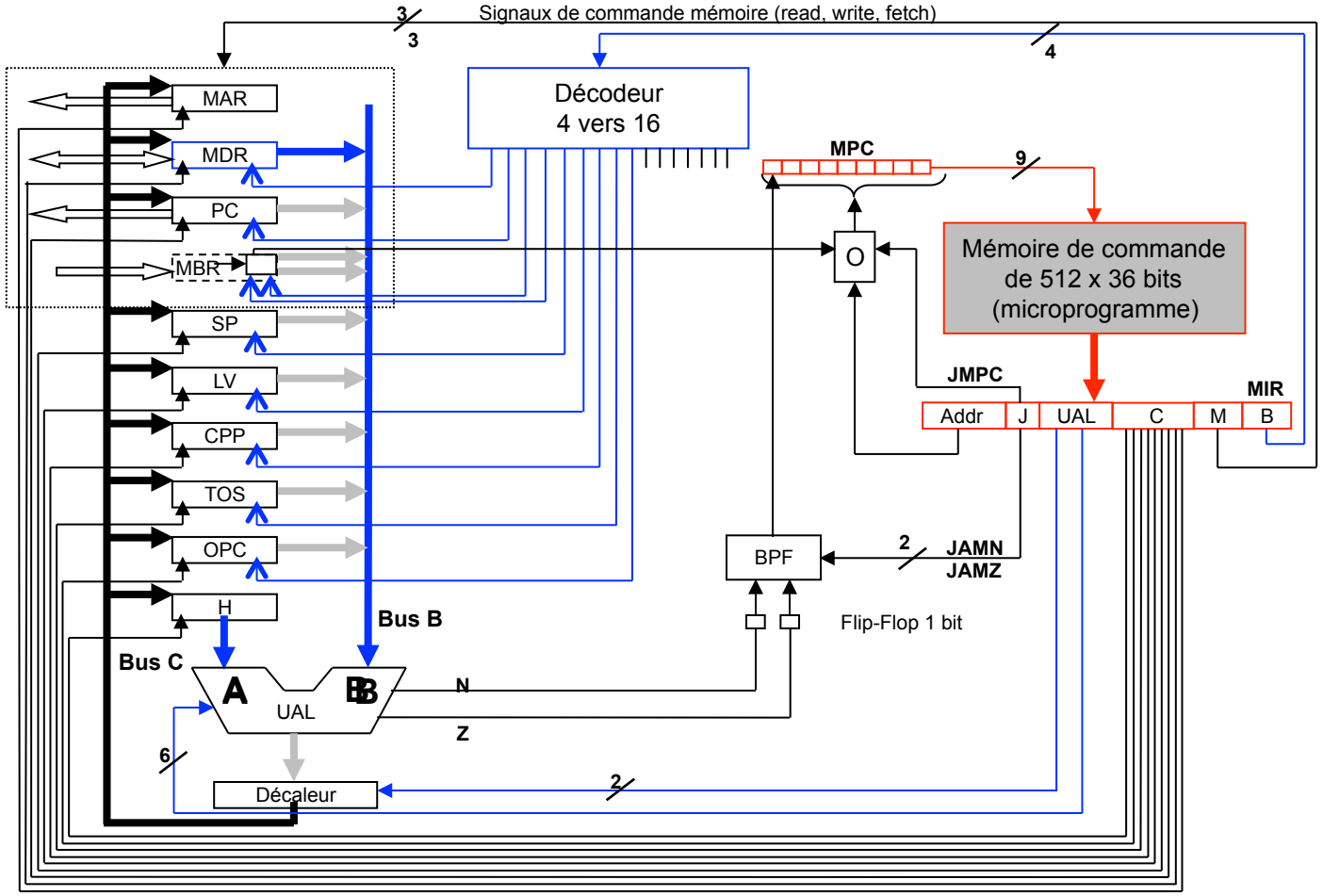
Ecriture sur le bus B

Ecriture du bus C dans le registre

5.1 Fonctionnement de la micromachine



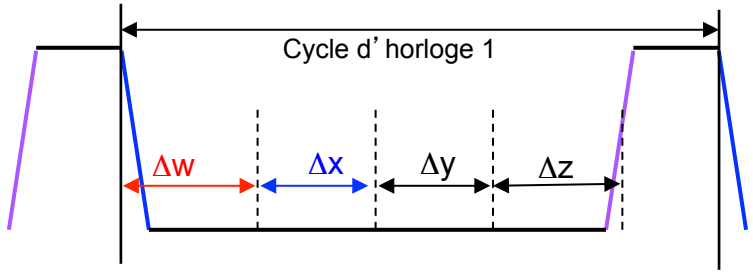
Sous-cycle 1
Sous cycle 2
Les divers signaux de commande se propagent
- Un registre est vidé sur le bus B
- les entrée de l' UAL sont stables
- la sélection de l' opération de l' UAL est réalisée



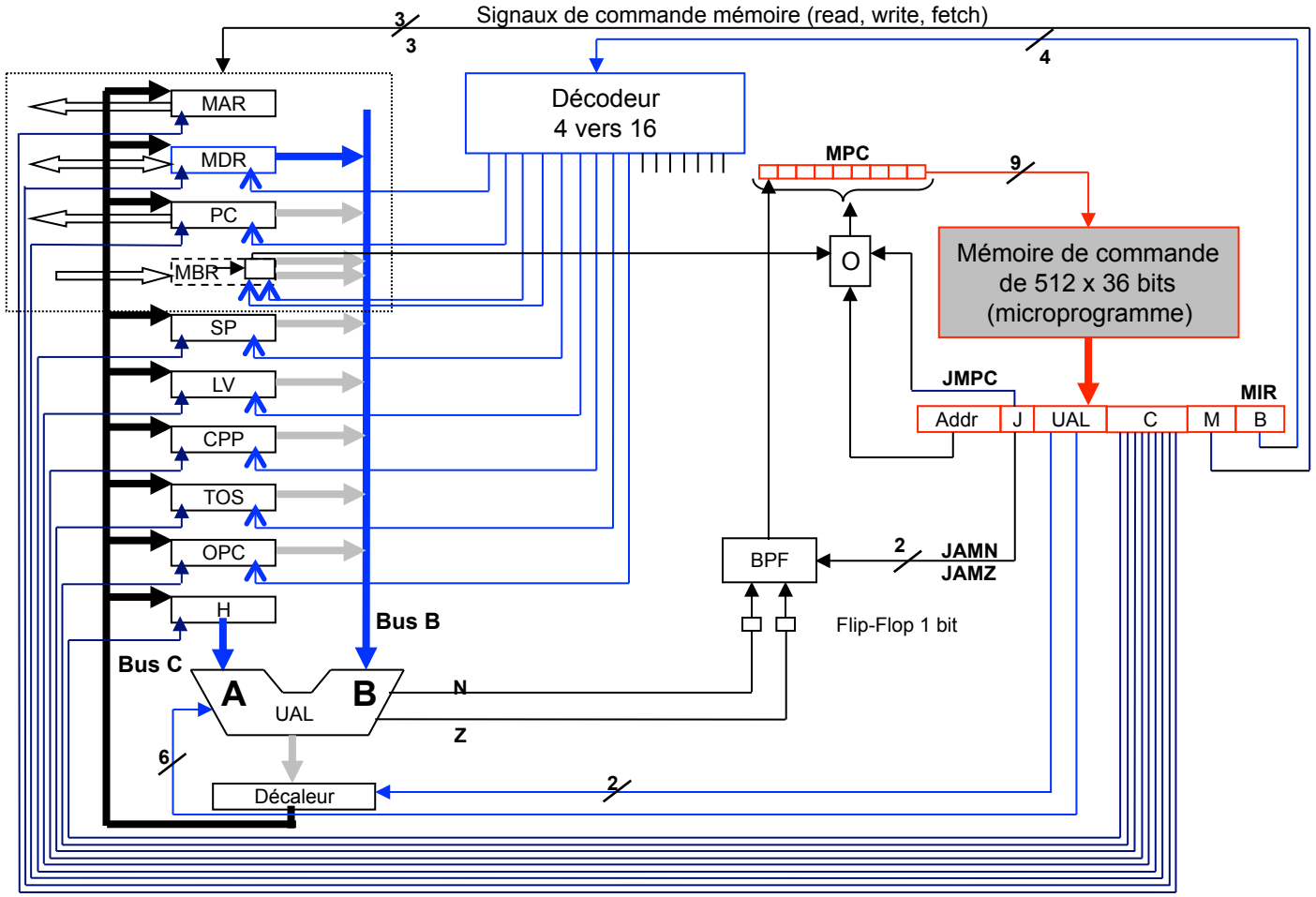
Ecriture
sur le bus B

Ecriture
du bus C
dans le registre

5.1 Fonctionnement de la micromachine



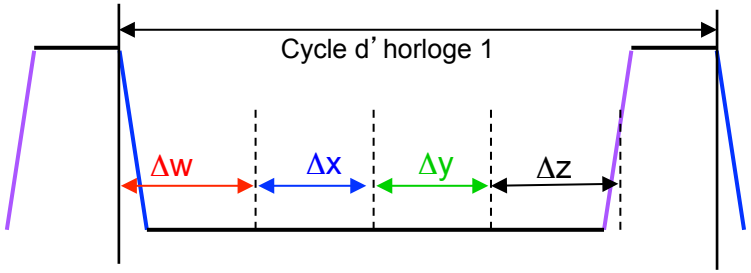
Sous-cycle 1
Sous cycle 2
Les divers signaux de commande se propagent
- Un registre est vidé sur le bus B
- les entrée de l' UAL sont stables
- la sélection de l' opération de l' UAL est réalisée
- le registre où sera sauvegardé le résultat est sélectionné
- la commande de la mémoire



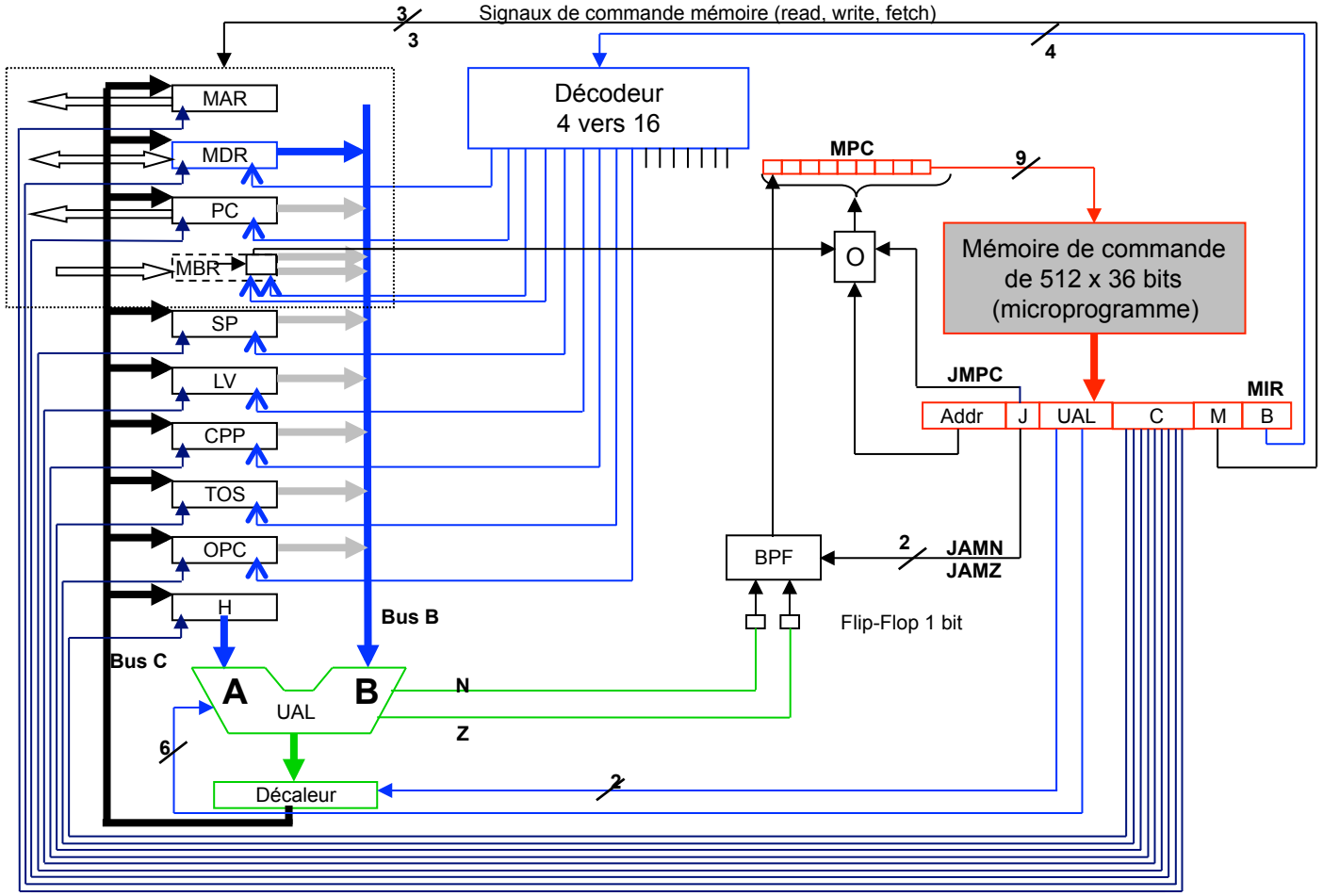
Ecriture sur le bus B

Ecriture du bus C dans le registre

5.1 Fonctionnement de la micromachine



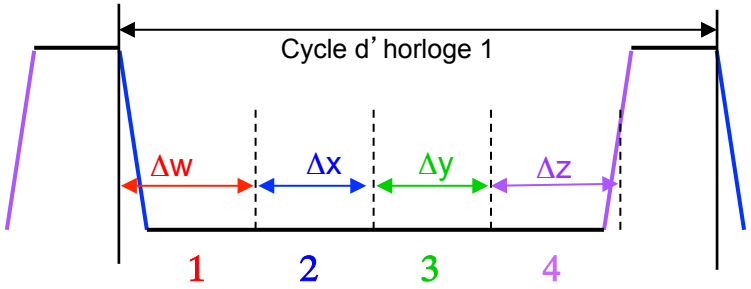
Sous-cycle 1
Sous cycle 2
Sous-cycle 3
- L' opération est réalisée par l' UAL
- Le résultat est disponible à la sortie du décaleur
- Les bits N et Z sont positionnés



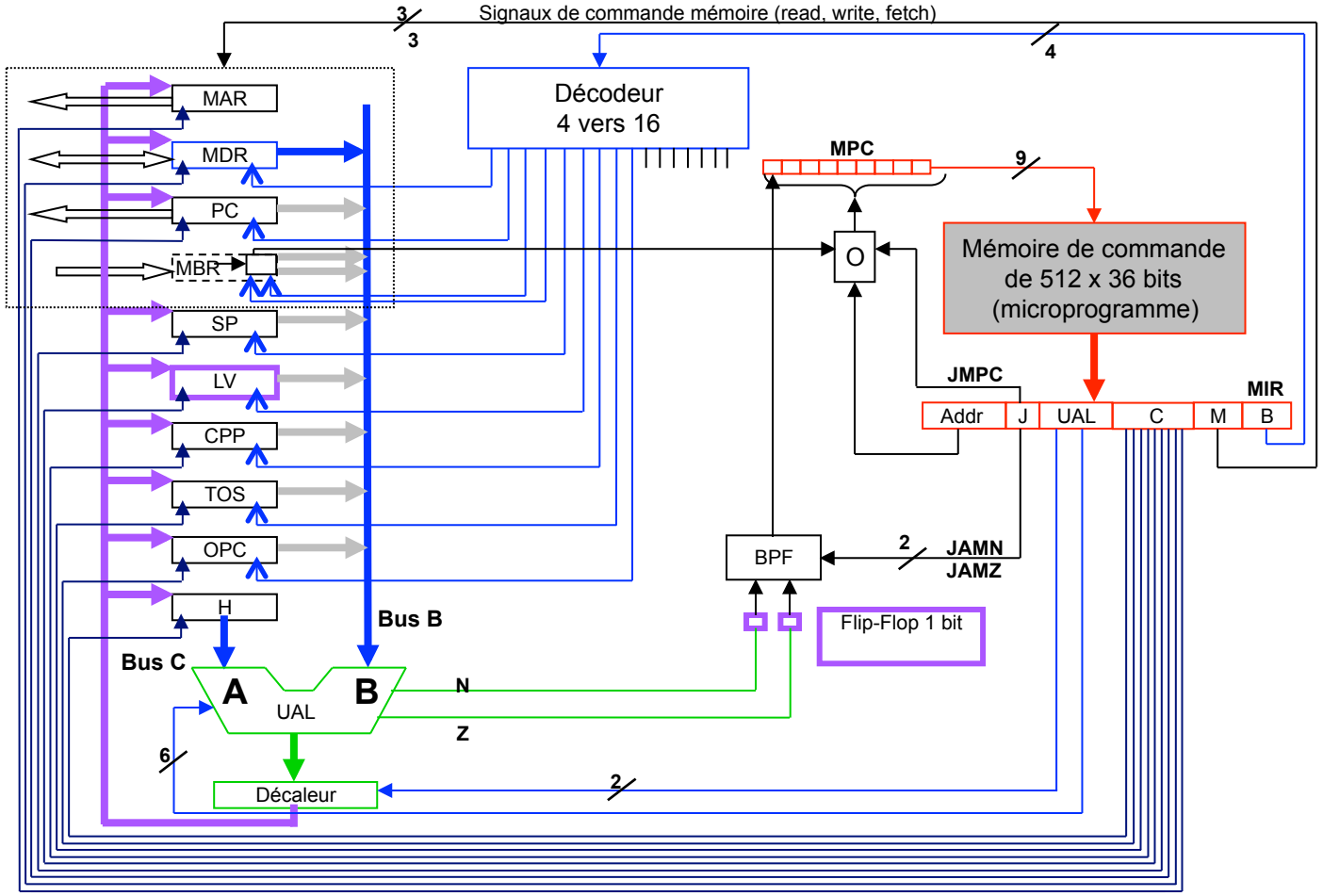
Ecriture
sur le bus B

Ecriture
du bus C
dans le registre

5.1 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

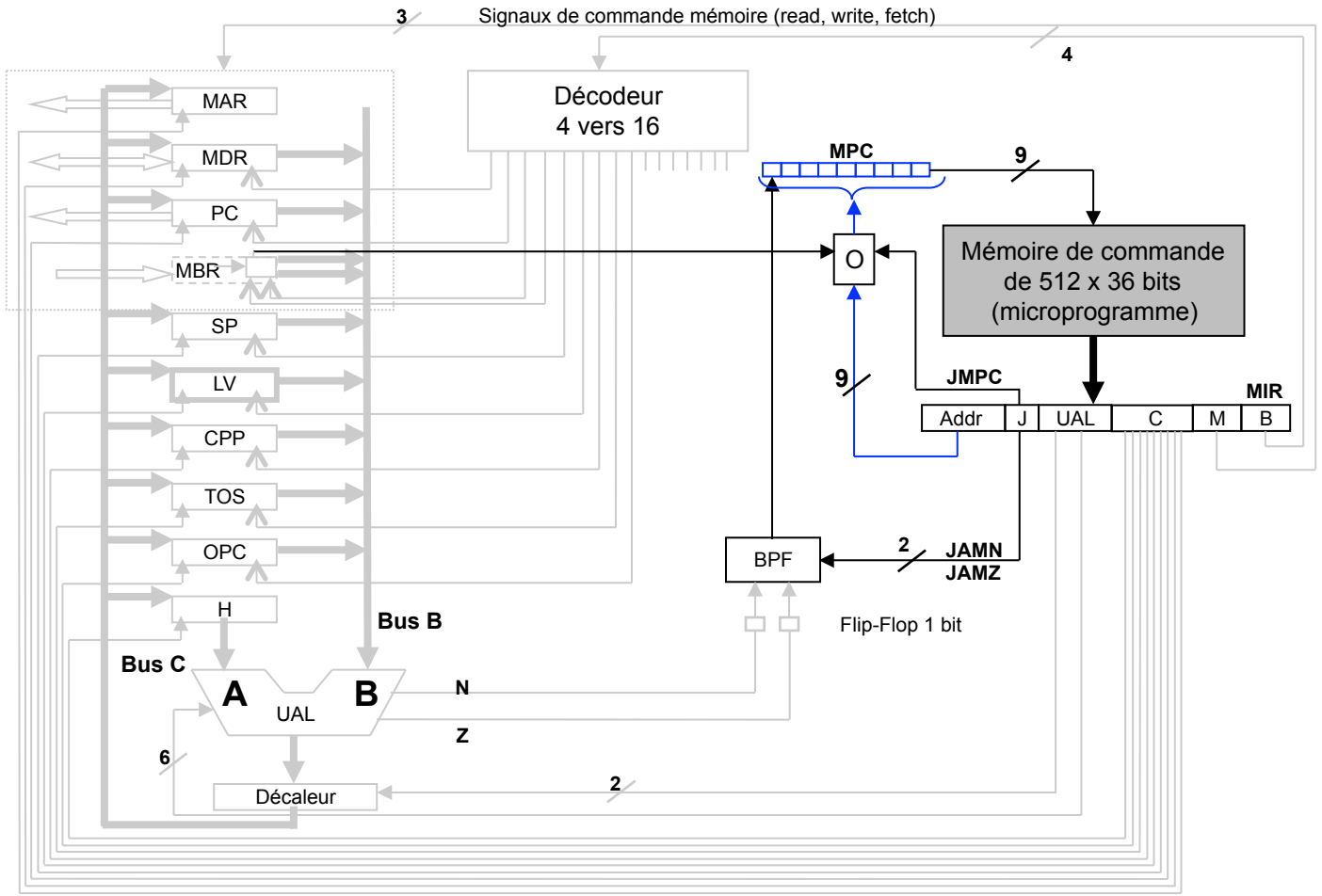
Ecriture
du bus C
dans le registre

5.2 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



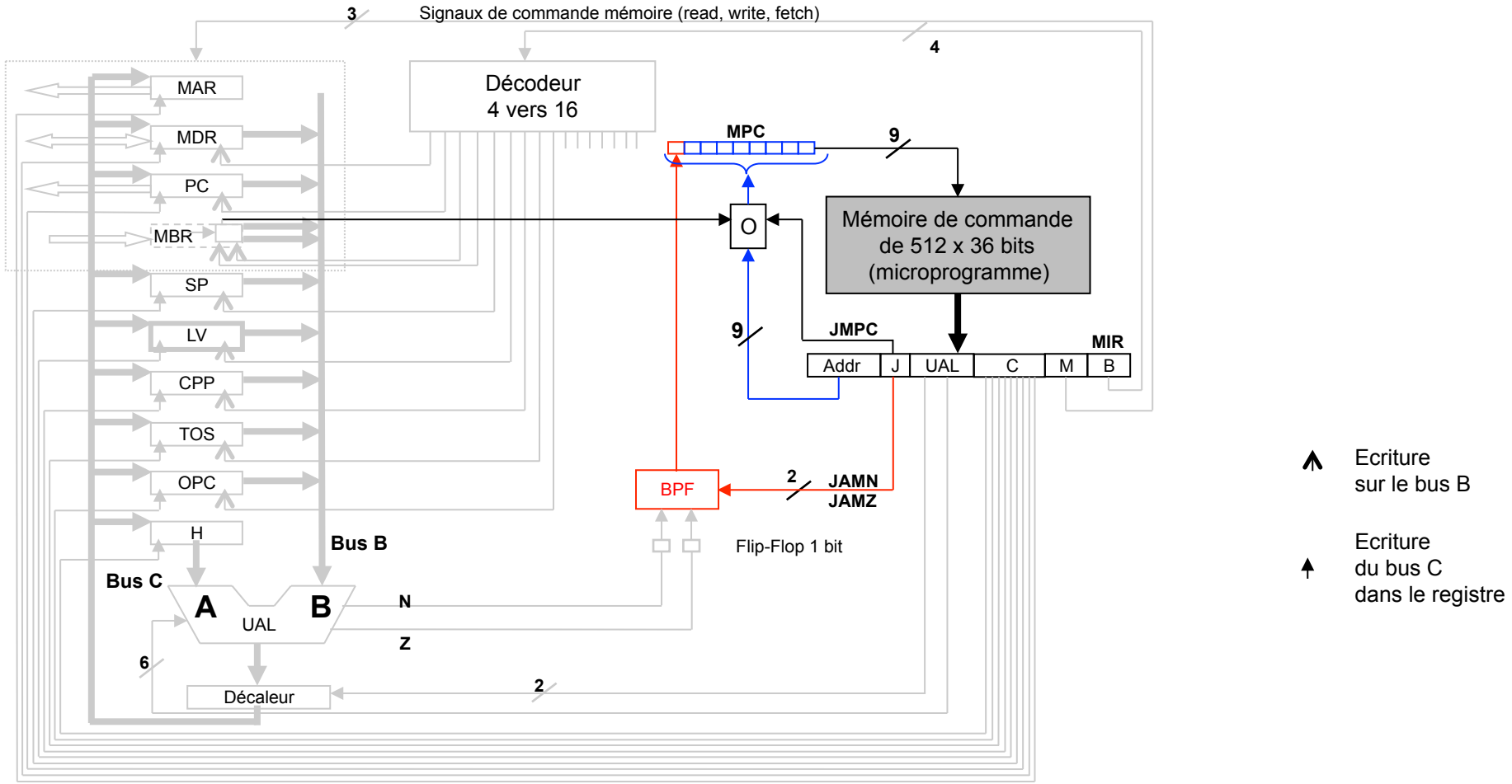
Ecriture
sur le bus B

Ecriture
du bus C
dans le registre

5.2 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

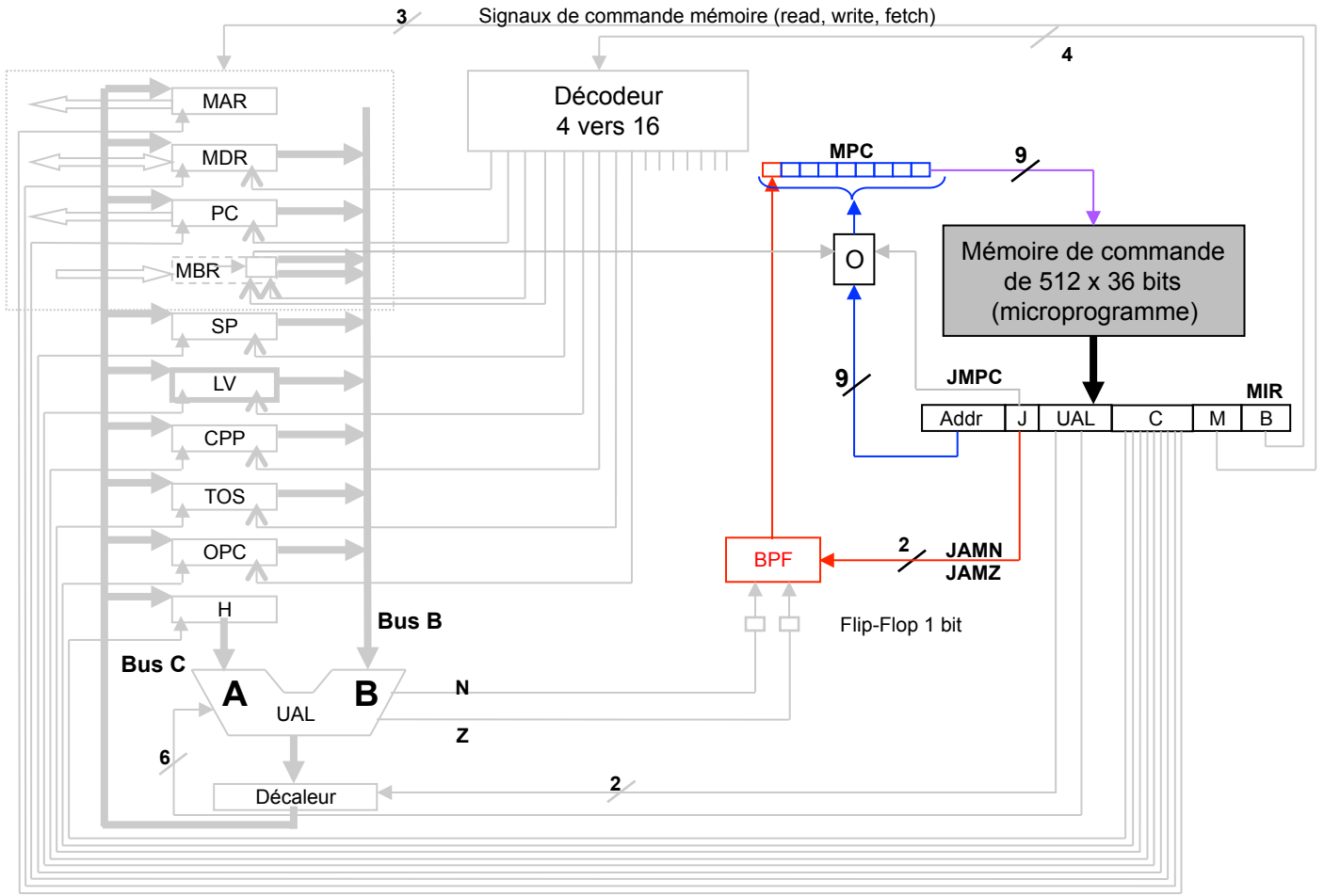


5.2 Séquencement des micro instructions

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 micro-instructions suivantes possibles selon le résultat du test qui détermine BPF



Ecriture sur le bus B

Ecriture du bus C dans le registre

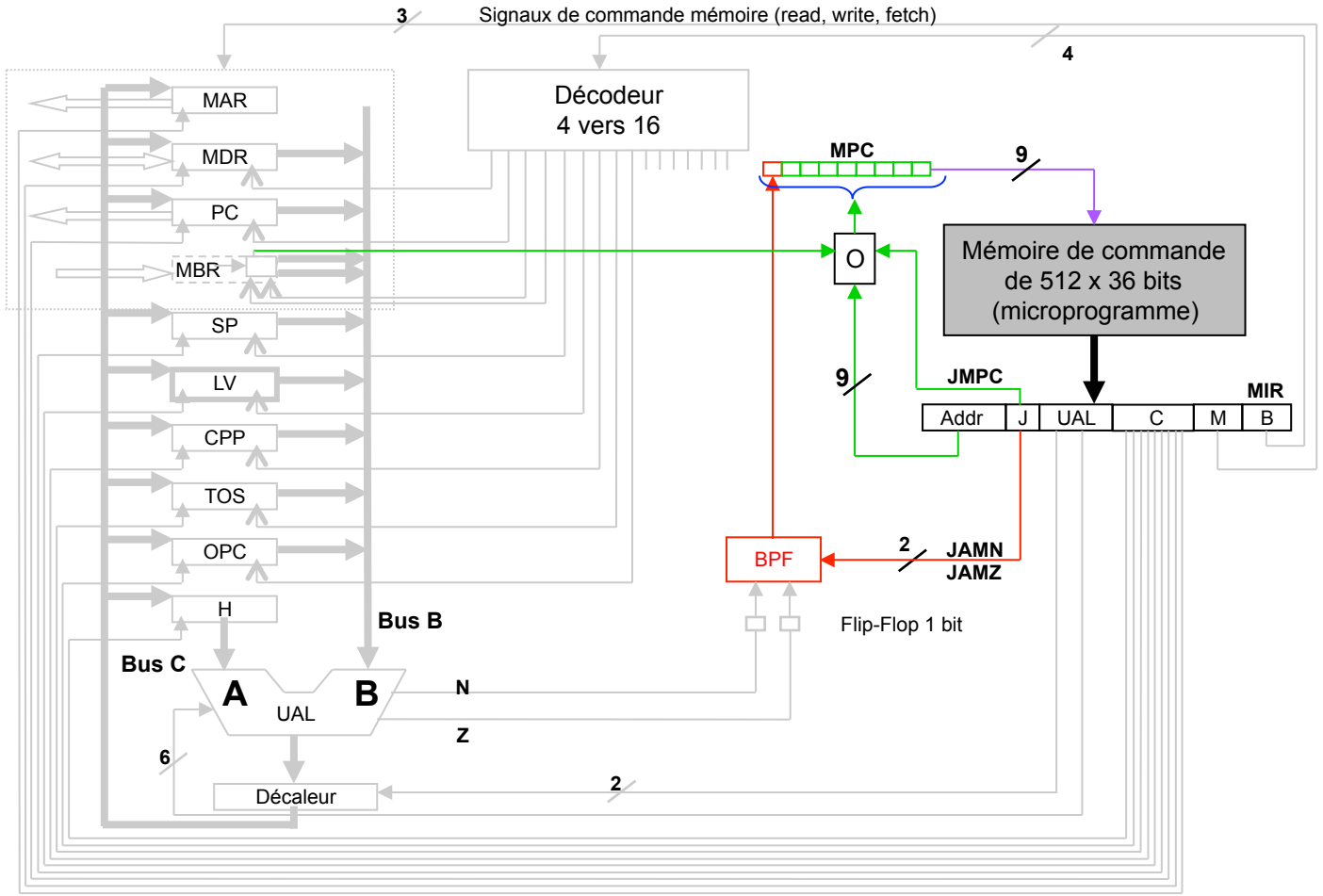
5.2 Séquencement des micro instructions

Mise à jour du MPC (micro program counter)

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

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

Lorsque JMPC=1 si Addr =0x00 alors l'adresse de l'instruction suivante vaut soit 0x100 soit 0x000
si Addr ≠ 0x00 alors si JAMN ET JAMZ =1 on peut adresser 512 adresses différentes
si JAMN=0 ET JAMZ =0 on peut adresser 256 adresses différentes



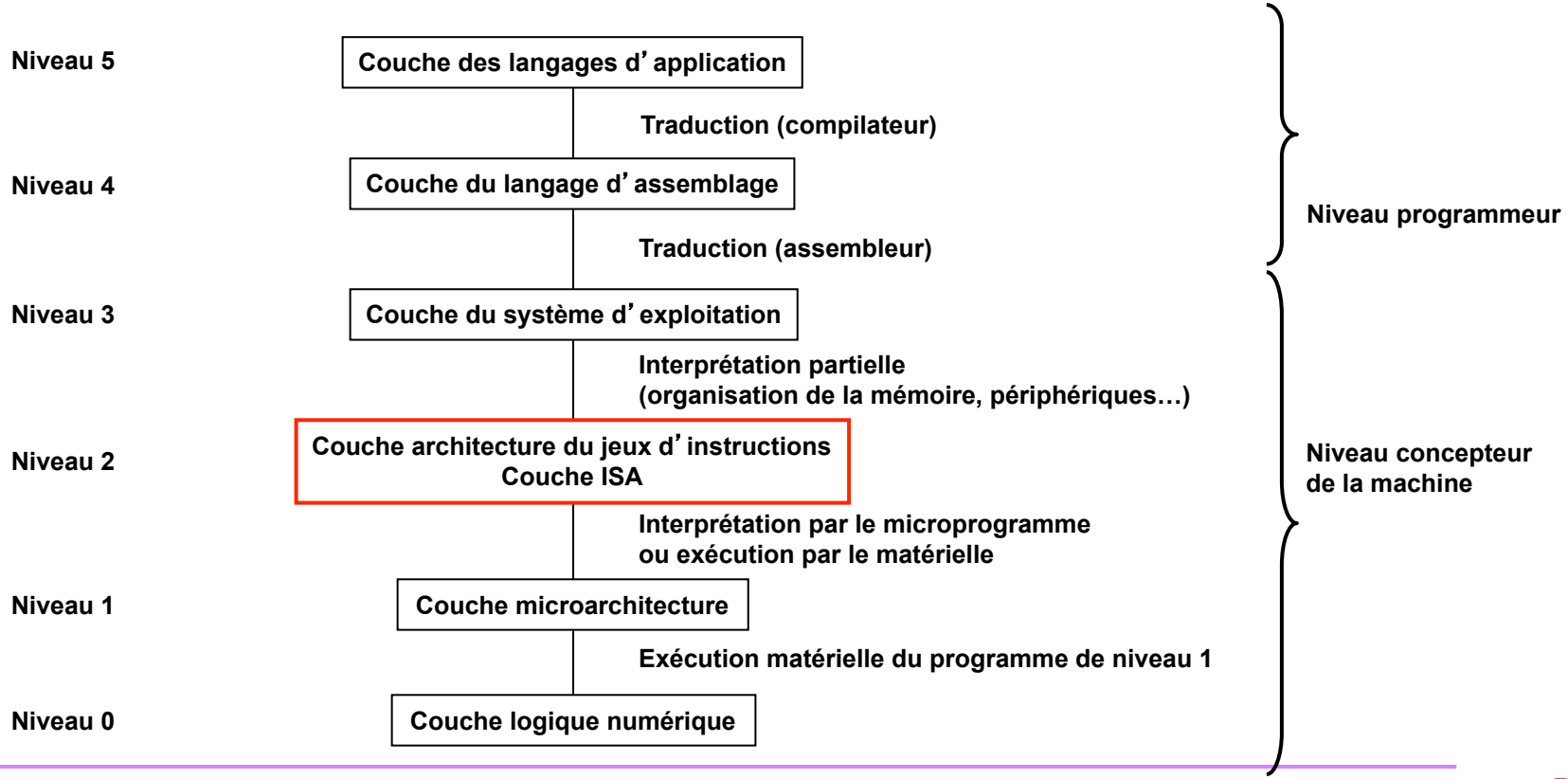
Ecriture sur le bus B

Ecriture du bus C dans le registre

Partie III: La couche ISA

Instruction Set Architecture

La couche ISA est la plus ancienne a 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



1. 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

1. Propriétés de la couche ISA

- 1.1 La mémoire
  - Alignement en mémoire
    - sur des adresses multiples de 4 (32 bits) ou de 8 (64 bits)
    - la compatibilité du PENTIUM avec des processeurs anciens (8 ou 16 bits: 8080…) oblige à gérer les adresses non alignées
  - Sérialisation des accès mémoire
    - l'optimisation des processeurs conduit parfois à réorganiser les instructions du programme (exécution dès que possible).
    - Il faut veiller dans certains cas à s'assurer que les informations transférées en mémoire le sont effectivement avant de continuer
  - Synchronisation
    - L'instruction SYNC : bloque l'exécution de toute nouvelle instruction devant accéder à la mémoire tant que la précédente n'est pas terminée

- 1.2. Les registres
  - Pas tous visibles de la couche ISA
    - MAR n'est pas visible, Registres réservés au système d'exploitation, au contrôle des caches
  - Registres spécialisés
    - compteur ordinal, pointeur de pile,
  - Registres généraux : stockent variables locales ou résultats de traitement
    - Sont interchangeables entre eux (pas de modification du comportement du programme)
    - Il y a parfois des conventions d'utilisation (passage de variables entre procédures)

- 1.3. 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

2. Couche ISA du PENTIUM IV

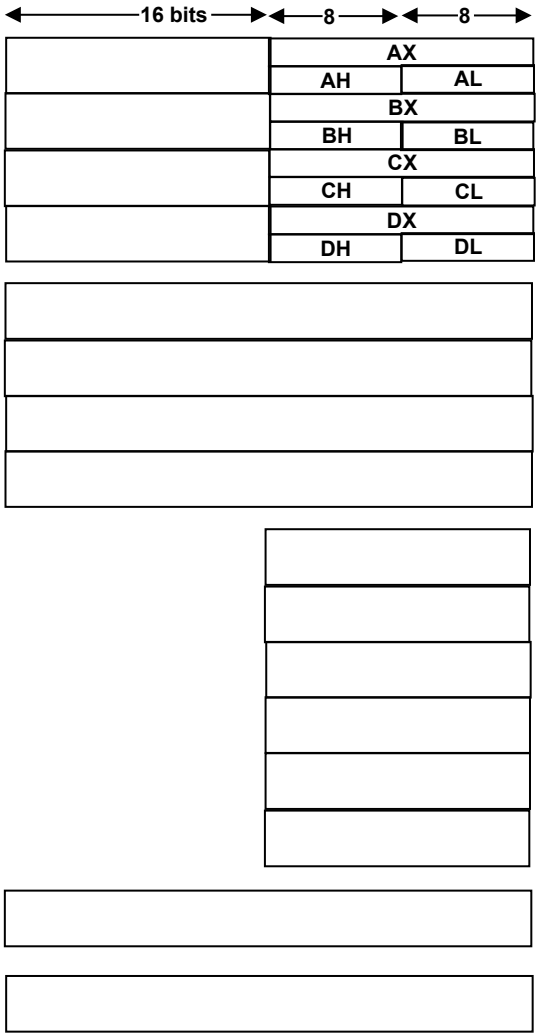
- **2.1. L'Héritage de l'architecture IA-32**
  - **Provient de la compatibilité ascendante:**
    - **Centrino, Pentium M, MPIV, PIII, PII, Pentium Pro, Pentium**
    - **I486 : instructions MMX**
    - **I386 : architecture 32 bits**
    - **I286 : 16 bits avec adressage de la mémoire par segments de 64 Ko**
    - **I8086, I8088 : puces 16 bits**
    - **I8080; I8008 : puces 8 bits**
    - **I4004 : puce 4 bits premier processeur d'Intel**

- **2.2. Modes de fonctionnement**
  - **Mode réel :**
    - **fonctionne comme un I8088, « Plante » si erreur du programme**
  - **Mode virtuel 8086 :**
    - **permet l'exécution en mode sécurisé des programmes compatibles 8086 (exemple ouverture d'une fenêtre MS-Dos)**
  - **Mode protégé :**
    - **comportement réel d'un Pentium IV**
    - **4 niveaux de privilèges selon l'état du registre PSW mot d'état**
      - **Niveau 0 : mode système, accès total, utilisé par le système d'exploitation**
      - **Niveau 3 : mode utilisateur, bloque certaines instructions et certains registres**
      - **Niveau 1 et 2 : rarement utilisé**

2. Couche ISA du PENTIUM IV

- **2.3. Organisation de la mémoire**
  - **Elle est divisée en segments adressant chacun 2^{32} octets**
  - **Il y a en théorie 16384 segments possibles! Mais seul un segment est utilisé par les OS actuels soit 2^{12} Mo= 4 Go adressables**
  - **Les mots de 32 bits sont stockés au format petit boutiste**

2.4 Les registres du PENTIUM IV



Registres généraux : sous registres 16 et 8 bits pour la compatibilité

Registre arithmétique principal

Registre de stockage des adresses

Utilisé dans les boucles

Joue un rôle particulier avec EAX dans les multiplications et divisions sur 64 bits

Pointeurs pour copie de chaînes de caractères : Origine et Destination

Pointeur sur la base du bloc de données

Pointeur de pile

Registres de segments pour compatibilité 8088

Utilisé pour le compteur ordinal

Registre mot d'état PSW

3. La couche ISA de l'Ultra SPARC III

- Première architecture RISC (couche ISA simple)
- 32 bits, la version 9 est 64 bits
- Les processeurs UltraSPARC sont tous compatibles sur leur couche ISA

- **3.1. La mémoire**
  - 2^{64} octets adressables en théorie, limite actuelle 2^{44} octets
  - Organisation petit boutiste par défaut mais modifiable par le registre PSW

3. La couche ISA de l'Ultra SPARC III

3.2. Les registres

- 32 registres à virgule flottante de 64 bits
  - Utilisable par paire pour une précision sur 128 bits
- 32 registres généraux de 64 bits : R0-R31

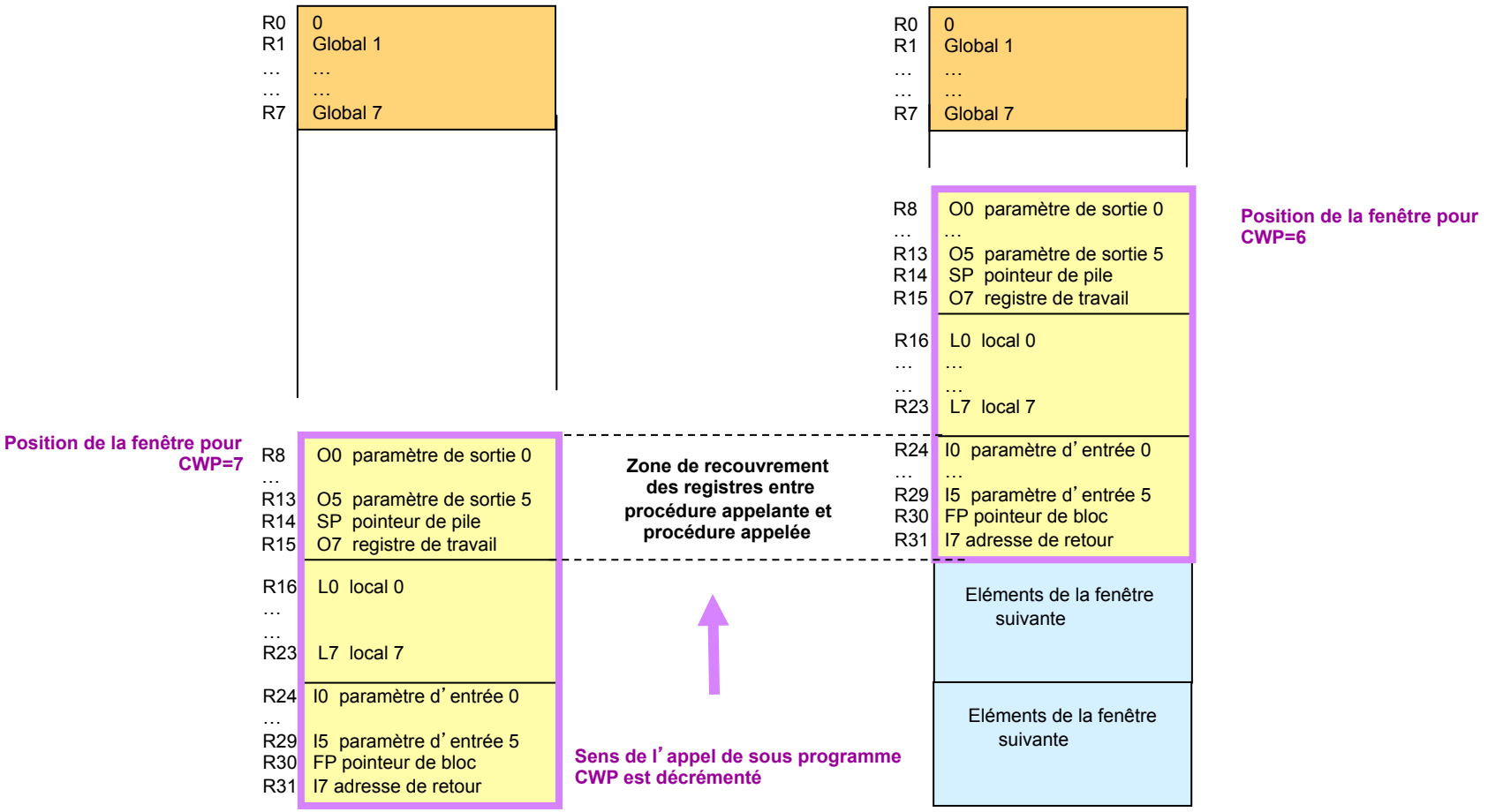
•	R0	G0	Contient toujours 0
•	R1-R7	G1-G7	Contiennent des variables globales
•	R8-R13	O0-O5	Contiennent les paramètres de la procédure en cours d'appel
•	R14	SP	Pointeur de pile
•	R15	O7	Registre de travail
•	R16-R23		L0-L7 Variables locales d'une procédure
•	R24-R29		I0-I5 Contiennent les paramètres entrants
•	R30	FP	Pointeur sur la base du bloc de données en cours
•	R31	I7	Adresse de retour de la procédure en cours

3. La couche ISA de l'Ultra SPARC III

3.3. Fenêtre de registres

- **Il y a en fait beaucoup plus de 32 registres mais seuls 32 sont visibles depuis le programme à un instant donné. Les 32 registres visibles à un instant donné constituent la fenêtre de registres**
- **Permet d'émuler une pile de registres pour optimiser les performances lors des appels de procédures**
- **La pile est gérée par le registre CWP (*Current Window Pointer*)**
  - **Lors d'un appel de procédure**
    - » le CWP est décrémenté et l'ensemble des 32 registres est remplacé par un nouvel ensemble.
    - » Les registres de sortie R8-R15 deviennent les registre d'entrée visibles dans R24-R31
    - » Les registres globaux R0-R7 restent les mêmes
  - **Les appels imbriqués de procédures peuvent épuiser les ensembles de registres utilisables. Il y a alors une copie en mémoire principale des ensembles de registres les plus anciens**

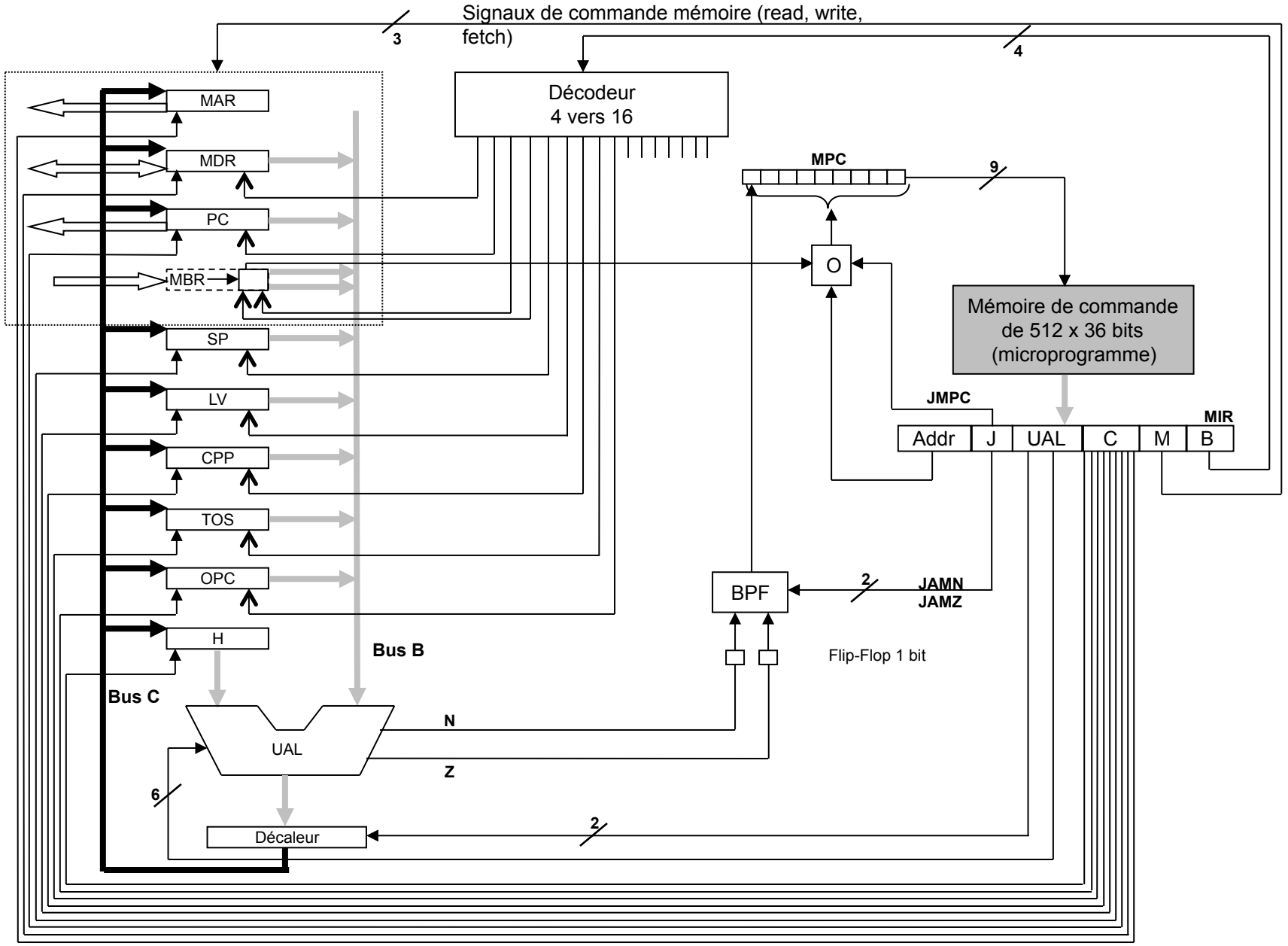
3.3 Fenêtre de registres



4. La couche ISA de l'IJVM

Il s'agit de décrire la couche instruction de la machine IJVM qui est interprétée par le microprogramme s'exécutant sur la microarchitecture

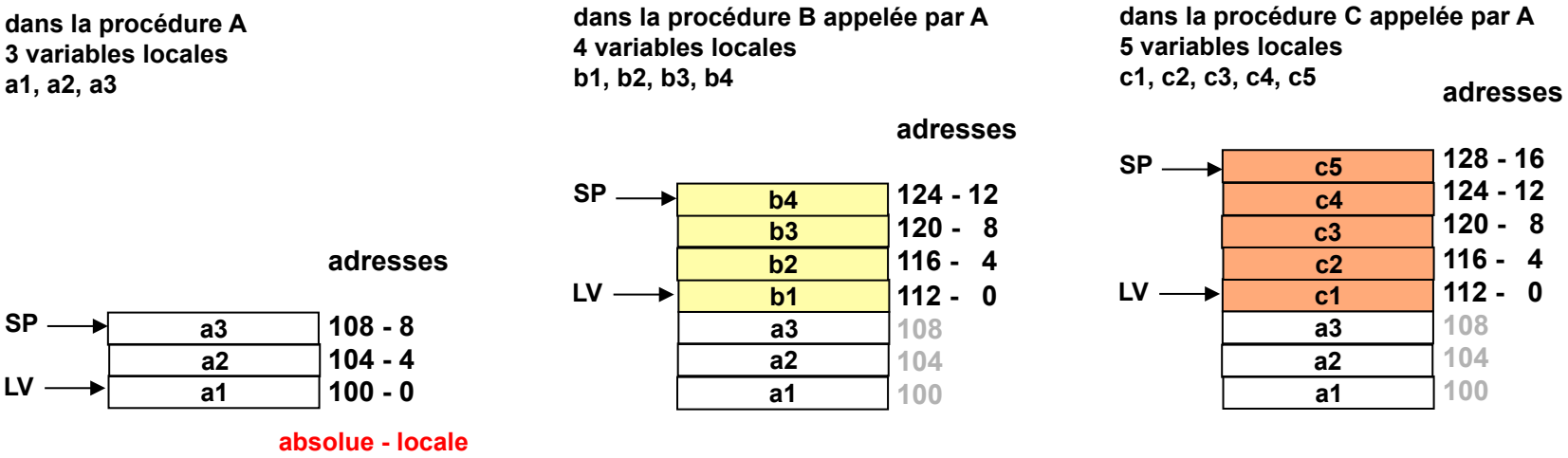
4. La couche ISA de l'IJVM



4.1 La notion de pile

- 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 de 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)

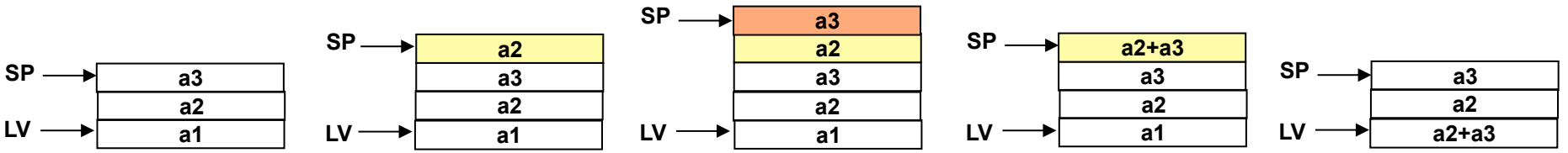


4.1 La pile d'opérandes

- On peut aussi 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 au sommet de la pile dans la variable a1



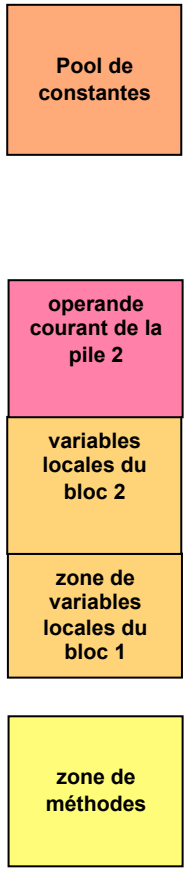
- Pratiquement toutes les machines utilisent des piles pour stocker les variables locales
- Très peu utilisent une pile d'opérandes: la JVM en fait partie

4.2. 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

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



4.3. Les instructions de l'IJVM

- **Chaque instruction comprend un code opération et parfois un code opérande**

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	permute les 2 mots au sommet de la pile
0xC4	WIDE	préfixe d'instruction

4.4. 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
	1 ILOAD j
	2 ILOAD k
	3 IADD
	4 ISTORE i
	5 ILOAD i
	6 BIPUSH 3
	7 IF_ICMPEQ L1
	8 ILOAD j
	9 BIPUSH 1
	10 ISUB
	11 ISTORE j
	12 GOTO L2
	13 L1: BIPUSH 0
	14 LISTORE k
	15 L2:
		0x15 0x02
		0x15 0x03
		0x60
		0x36 0x01
		0x15 0x01
		0x10 0x03
		0x9F 0x00 0x0D
		0x15 0x02
		0x10 0x01
		0x64
		0x36 0x02
		0xA7 0x00 0x07
		0x10 0x00
		0x36 0x03

4.5. Implémentation du microprogramme

- Il s'agit de définir les micro-instructions qui réalisent le microprogramme de chaque instruction de la macro-archirecture
- On définit un langage de micro-instructions plus facile à manipuler que des codes 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 en un seul cycle d'horloge

4.5. 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 – H
DEST = SOURCE	DEST = SOURCE – 1
DEST = $\overline{H}$	DEST = –H
DEST = $\overline{SOURCE}$	DEST = H AND SOURCE
DEST = H + SOURCE	DEST = H OR SOURCE
DEST = H + SOURCE + 1	DEST = 0
DEST = H + 1	DEST = 1
DEST = SOURCE + 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

4.5. 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 est 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 de la seconde micro-instruction

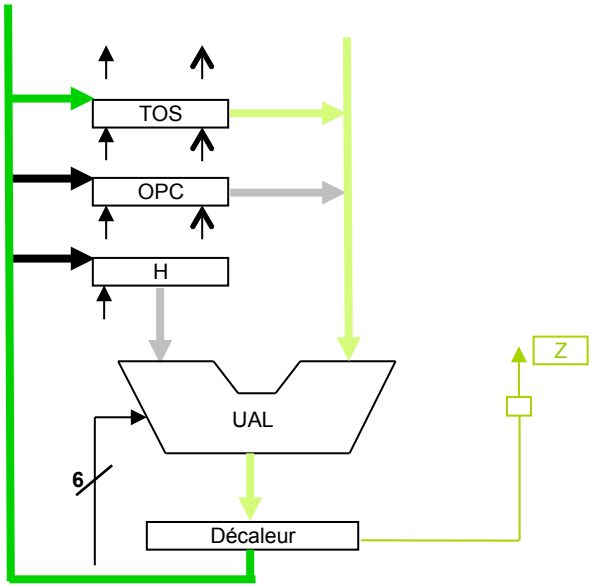
4.5. Le langage de micro-assemblage

- **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)**



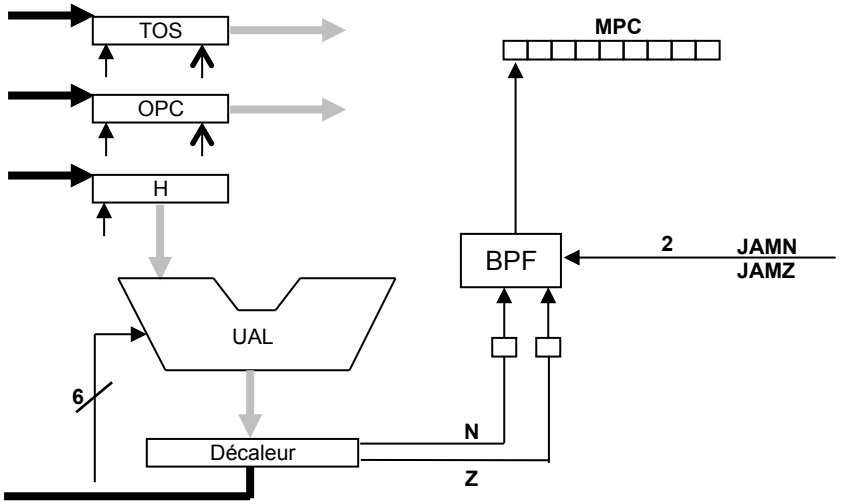
4.5. Le langage de micro-assemblage

- **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

4.5. Le langage de micro-assemblage

- **Branchement à une micro-instruction spécifiée par MBR**

  goto (MBR OR *addr*)

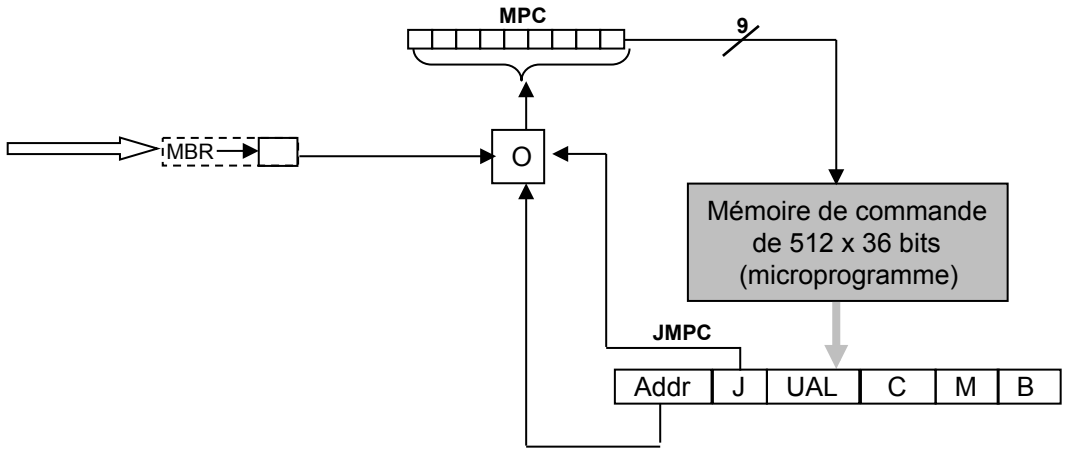
  lorsque le bit JMPC est positionné

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

  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



4.5. 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 groupe de 4 octets provenant de la mémoire**

  - **TOS et POC 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**

4.5. 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

4.5. 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

4.5. 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; goto Main1	La valeur du sommet de la pile est copiée dans MDR qui sera écrit dans le nouveau sommet de pile

4.5. 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

4.5. 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

4.5. Implémentation du microprogramme

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

Label	Opérations	Commentaires
goto1 goto2 goto3 goto4 goto5 goto6	OPC = PC – 1 PC = PC + 1; fetch H = MBR << 8 H = MBRU OR H PC = OPC + H; fetch goto Main1	Sauve addresse opcode MBR = premier octet offset; fetch 2nd octet Décal. et sauv. premier octet signé dans H H = offset brancht de 16 bits Additionne offset à OPC Attente fetch opcode suivant
iflt1 iflt2 iflt3 iflt4	MAR = SP = SP – 1; rd OPC = TOS TOS = MDR N = OPC; if (N) goto T; else goto F	Lit et stocke mot sommet de pile suivant Sauve TOS dans OPC temporairement Range nouv. som. pile ds TOS Brancht sur bit N
ifeq1 ifeq2 ifeq3 ifeq4	MAR = SP = SP – 1; rd OPC = TOS TOS = MDR Z = OPC; if (Z) goto T; else goto F	Lit et stocke mot sommet de pile suivant Sauve TOS dans OPC temporairement Range nouv. som. pile ds TOS Brancht sur bit Z
if_icmpeq1 if_icmpeq2 if_icmpeq3 if_icmpeq4 if_icmpeq5 if_icmpeq6	MAR = SP = SP – 1; rd MAR = SP = SP – 1 H = MDR; rd OPC = TOS TOS = MDR Z = OPC – H; if (Z) goto T; else goto F	Lit et stocke mot sommet pile suivant Set MAR pour lire et écrire nouv. som. pile Copie 2e mot de pile dans H Sauve TOS dans OPC temporairement Range nouv. som. pile ds TOS If 2 mots égaux, goto T, else goto F
T	OPC = PC – 1; fetch; goto goto2	Idem goto1
F F2 F3	PC = PC + 1 PC = PC + 1; fetch goto Main1	Saute 1er octet offset PC contient opcode suivant Attente fetch opcode

invokevirtual1	PC = PC + 1; fetch	MBR = index octet 1; inc. PC, get 2e octet
invokevirtual2	H = MBRU << 8	Décal. et sauv. 1er octet dans H
invokevirtual3	H = MBRU OR H	H = offset pointeur méthode de CPP
invokevirtual4	MAR = CPP + H; rd	Prend pointeur méthode ds zone CPP
invokevirtual5	OPC = PC + 1	Sauve retour PC dans OPC temporairement
invokevirtual6	PC = MDR; fetch	PC pointe nouv. méthode; get compt. param.
invokevirtual7	PC = PC + 1; fetch	Fetch 2e octet compteur paramètres
invokevirtual8	H = MBRU << 8	Décal. et sauv. 1er octet dans H
invokevirtual9	H = MBRU OR H	H = nombre de paramètres
invokevirtual10	PC = PC + 1; fetch	Fetch 1er octet de # locales
invokevirtual11	TOS = SP - H	TOS = addresse de OBJREF - 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = addresse de OBJREF (nouv. LV)
invokevirtual13	PC = PC + 1; fetch	Fetch 2e octet de # locales
invokevirtual14	H = MBRU << 8	Décal. et sauv. 1er octet dans H
invokevirtual15	H = MBRU OR H	H = # locales
invokevirtual16	MDR = SP + H + 1; wr	Superposer OBJREF avec pointeur lien
invokevirtual17	MAR = SP = MDR;	Set SP, MAR à endroit maintien vieux PC
invokevirtual18	MDR = OPC; wr	Sauv. vieux PC au-dessus variables locales
invokevirtual19	MAR = SP = SP + 1	SP pointe à position maintien vieux LV
invokevirtual20	MDR = LV; wr	Sauv. vieux LV au dessus de sauv. PC
invokevirtual21	PC = PC + 1; fetch	Fetch 1er opcode de nouvelle méthode
invokevirtual22	LV = TOS; goto Main1	Set LV pour pointer bloc variables LV
ireturn1	MAR = SP = LV; rd	Reset SP, MAR reçoit pointeur lien
ireturn2		Attente pour lire
ireturn3	LV = MAR = MDR; rd	Set LV à ptr lien; prendre vieux PC
ireturn4	MAR = LV + 1	Set MAR pour lire ancien LV
ireturn5	PC = MDR; rd; fetch	Restaure PC; fetch opcode suivant
ireturn6	MAR = SP	Set MAR pour écrire TOS
ireturn7	LV = MDR	Restaure LV
ireturn8	MDR = TOS; wr; goto Main1	Sauve val. retour au sommet pile originale

Label	Opérations	Commentaires
Main1	PC = PC + 1; fetch; goto (MBR)	MBR = opcode; fetch octet suiv.; brancht.
nop1	goto Main1	Ne rien faire
iadd1 iadd2 iadd3	MAR = SP = SP – 1; rd H = TOS MDR = TOS = MDR + H; wr; goto Main1	Lit et stocke mot sommet de pile suivant H = sommet pile (*top of stack*) Add. deux mots; écrit résul. ds TOS
isub1 isub2 isub3	MAR = SP = SP – 1; rd H = TOS MDR = TOS = MDR – H; wr; goto Main1	Lit et stocke mot sommet de pile suivant H = sommet pile (*top of stack*) Fait soustr.; écrit résul. ds TOS
iand1 iand2 iand3	MAR = SP = SP – 1; rd H = TOS MDR = TOS = MDR AND H; wr; goto Main1	Lit et stocke mot sommet de pile suivant H = sommet pile (*top of stack*) Fait ET; écrit à som. pile
ior1 ior2 ior3	MAR = SP = SP – 1; rd H = TOS MDR = TOS = MDR OR H; wr; goto Main1	Lit et stocke mot sommet de pile suivant H = sommet pile (*top of stack*) Fait OU; écrit à nouv. som. pile
dup1 dup2	MAR = SP = SP + 1 MDR = TOS; wr; goto Main1	Incrémente SP et copie ds MAR Ecrit nouv. mot ds pile
pop1 pop2 pop3	MAR = SP = SP – 1; rd TOS = MDR; goto Main1	Lit et stocke mot sommet de pile suivant Attente nouv. TOS de mémoire Copie nouv. mot dans TOS
swap1 swap2 swap3 swap4 swap5 swap6	MAR = SP – 1; rd MAR = SP H = MDR; wr MDR = TOS MAR = SP – 1; wr TOS = H; goto Main1	Set MAR à SP – 1; lit 2e mot de pile Set MAR à sommet de pile Sauv. TOS ds H; écrit 2e mot au som. pile Copie ancien TOS dans MDR Set MAR à SP – 1; écrit 2e mot dans pile Màj TOS

bipush1 bipush2 bipush3	SP = MAR = SP + 1 PC = PC + 1; fetch MDR = TOS = MBR; wr; goto Main1	MBR = octet à ranger dans pile Incrémente PC, fetch opcode suivant Ext. signe const. et push dans pile
iload1 iload2 iload3 iload4 iload5	H = LV MAR = MBRU + H; rd MAR = SP = SP + 1 PC = PC + 1; fetch; wr TOS = MDR; goto Main1	MBR contient index; copie LV dans H MAR = ad. variable locale à ranger ds pile SP pointe nouv. som. pile; prépare écriture Inc. PC; get opcode suivant; écrit som. pile Màj TOS
istore1 istore2 istore3 istore4 istore5 istore6	H = LV MAR = MBRU + H MDR = TOS; wr SP = MAR = SP - 1; rd PC = PC + 1; fetch TOS = MDR; goto Main1	MBR contient index; copie LV dans H MAR = ad. variable locale à ranger ds pile Copie TOS ds MDR; écrit mot Lit et stocke mot sommet de pile suivant Incrémente PC; fetch opcode suivant Màj TOS
wide1	PC = PC + 1; fetch; goto (MBR OR 0x100)	Brancht multivoie
wide_iload1 wide_iload2 wide_iload3 wide_iload4	PC = PC + 1; fetch H = MBRU << 8 H = MBRU OR H MAR = LV + H; rd; goto iload3	MBR contient 1er octet index; fetch 2e H = 1er octet index décalé à gauche de 8 bits H = index de 16 bits MAR = ad. variable locale à ranger ds pile
wide_istore1 wide_istore2 wide_istore3 wide_istore4	PC = PC + 1; fetch H = MBRU << 8 H = MBRU OR H MAR = LV + H; goto istore3	MBR contient 1er octet index; fetch 2e H = 1er octet index décalé à gauche de 8 bits H = index de 16 bits MAR = addr. variable locale à ranger ds pile
ldc_w1 ldc_w2 ldc_w3 ldc_w4	PC = PC + 1; fetch H = MBRU << 8 H = MBRU OR H MAR = H + CPP; rd; goto iload3	MBR contient 1er octet index; fetch 2e H = 1er octet index << 8 H = index 16 bits MAR = addresse constante ds pool
iinc1 iinc2 iinc3 iinc4 iinc5 iinc6	H = LV MAR = MBRU + H; rd PC = PC + 1; fetch H = MDR PC = PC + 1; fetch MDR = MBR + H; wr; goto Main1	MBR contient index; copie LV dans H Copie LV + index ds MAR; lit variable Fetch constante Copie variable dans H Fetch opcode suivant Range somme ds MDR; màj variable

5. Optimisation de l'architecture Mic-1

On veut diminuer le nombre de micro-instructions par instruction ISA

1. **intégrer la boucle de l'interpréteur dans les micro-instructions**
   la boucle principale Main1 peut parfois être intégrée à l'instruction en cours. On anticipe ainsi la recherche de l'instruction suivante.

2. **architecture à 3 BUS : 2 BUS pour alimenter l'UAL (A & B)**
   Éviter les cycles de chargement du registre H pour faire une addition

3. **ajout d'une unité de recherche d'instructions**
   la recherche de l'instruction suivante occupe l'UAL inutilement (pour l'incrémentation de PC). On va introduire un circuit spécialisé

5. Optimisation de l'architecture Mic-1

1. intégrer la boucle de l'interpréteur dans les micro-instructions

Exemple sur l'instruction POP: « dépile »

Label	Opérations	Commentaires
pop1	MAR = SP = SP – 1; rd	Lit le 2ème mot dans la pile, prépare MAR et SP
pop2		Attend la disponibilité du mot lu en mémoire
pop3	TOS = MDR; goto Main1	Actualise TOS et lance la boucle principale

La boucle principale est donc systématiquement exécutée ensuite

Main1	PC = PC + 1; fetch, goto MBR	

5. Optimisation de l'architecture Mic-1

1. intégrer la boucle de l'interpréteur dans les micro-instructions

On profite de l'attente au cours de pop 2 pour lancer la recherche de l'instruction suivante et éviter ainsi le lancement de la boucle principale ensuite

Label	Opérations	Commentaires
pop1	MAR = SP = SP – 1; rd	Lit le 2ème mot dans la pile, prépare MAR et SP
pop2	PC = PC + 1; fetch	Attend la disponibilité du mot lu en mémoire
pop3	TOS = MDR; goto MBR	Actualise TOS et lance la prochaine instruction

L'instruction pop prend toujours 3 cycles d'horloge mais la prochaine instruction s'exécutera immédiatement après. On évite le cycle du au passage par Main 1

5. Optimisation de l'architecture Mic-1

2. architecture à 3 BUS : 2 BUS pour alimenter l'UAL (A & B)

Pour aller plus vite il faut réduire le chemin des données
L'ajout d'un second bus A permet d'éviter le passage par H d'un des opérandes lors d'une opération sur 2 opérandes

Rappel : iload avec une architecture à 1 bus

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

5. Optimisation de l'architecture Mic-1

2. architecture à 3 BUS : 2 BUS pour alimenter l'UAL (A & B)

Instruction iload avec une architecture à 3 bus

on peut éviter le cycle de chargement de H et faire directement l'addition du pointeur de début de zone de mémoire locale avec le numéro de la variable locale (varnum) que l'on veut charger en haut de la pile

Label	opérations	commentaires
Iload1	MAR = MBRU + LV; rd	MAR est chargé avec l'adresse de la variable locale à charger en haut de la pile; l'addition est effectuée La lecture en mémoire est lancée
Iload2	MAR = SP = SP + 1	Le pointeur de sommet de la pile est incrémenté et copié dans MAR
iload3	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 en haut de la pile Parallèlement on incrémente le compteur de programme pour aller chercher l'instruction suivante en mémoire
iload4	TOS = MDR;	Le registre TOS est actualisé avec le contenu du nouveau sommet de la pile. On saute à la boucle principale
iload5	PC = PC + 1 ; fetch; goto MBR	Le cycle gagné permet d'intègrer la boucle principale dans l'instruction

5. Optimisation de l'architecture Mic-1

3. ajout d'une unité de recherche d'instructions

Les étapes nécessaires à la recherche de l'instruction suivante

- Incrémenter PC (en passant par l'UAL)
- Extraire l'instruction grace à PC
- Lire les opérandes en mémoire
- L'UAL effectue un calcul dont les résultats sont enregistrés

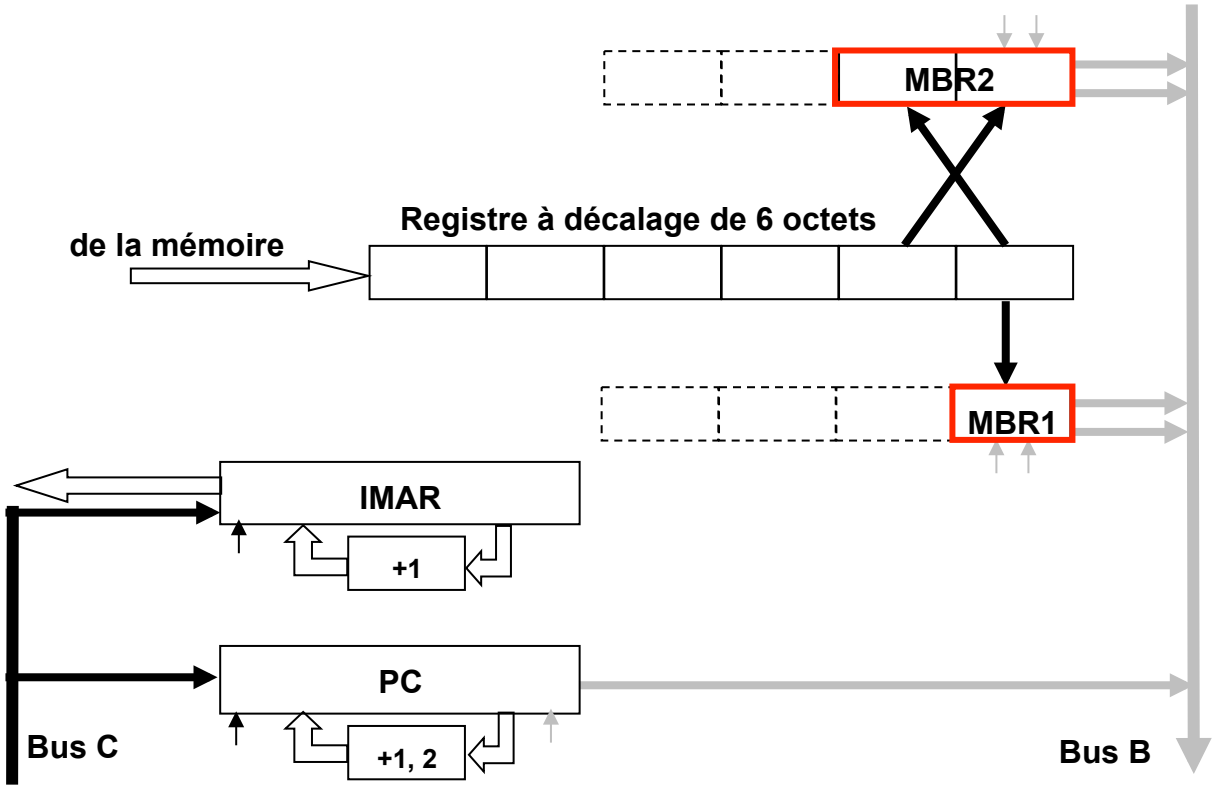
Les instructions sont lues octets par octets,
Les opérandes, qui sont codés sur deux octets, sont lus octets par octets de la même façon et doivent être assemblés pour constituer des mots de 2 octets.

L'UAL est beaucoup utilisée pour réaliser le décodage des instructions

On conçoit donc un circuit spécifique pour décharger l'UAL de l'ensemble de cette tâche
C'est l'unité de recherche d'instruction

5. Optimisation de l'architecture Mic-1

3. Schéma de l'Unité de Recherche d'Instruction

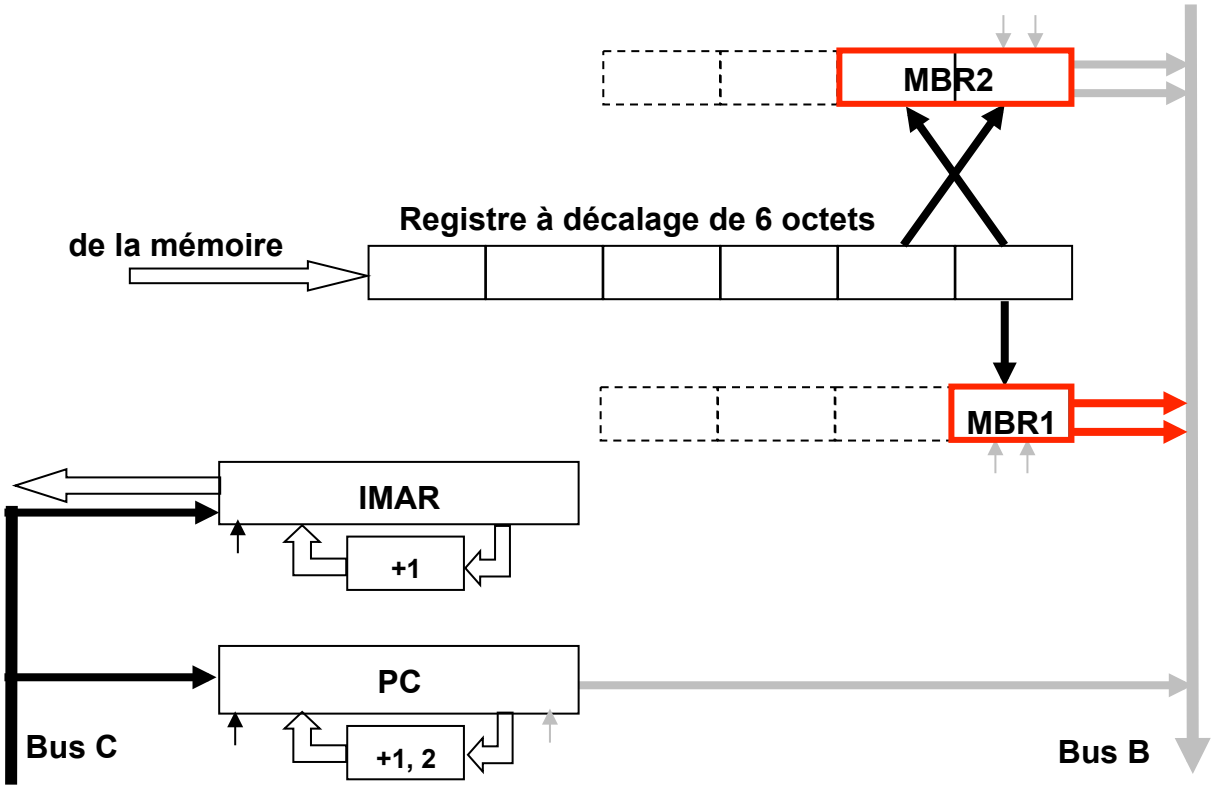


L'unité peut
- interpréter chaque code opération
- déterminer le nombre de champs supplémentaires à lire, les extraire puis les assembler et les maintenir prêts dans des registres (MBR1 et MBR2, ils sont disponibles lorsque l'unité principale en a besoin

- L'unité prépare de façon systématique les futures portions de 8 et 16 bits dont l'unité principale pourrait avoir besoin et les place dans MBR1 et MBR2

5. Optimisation de l'architecture Mic-1

3. Schéma de l'Unité de Recherche d'Instruction



Le registre MBR2 est de 16 bits il contient toujours les 2 prochains octets

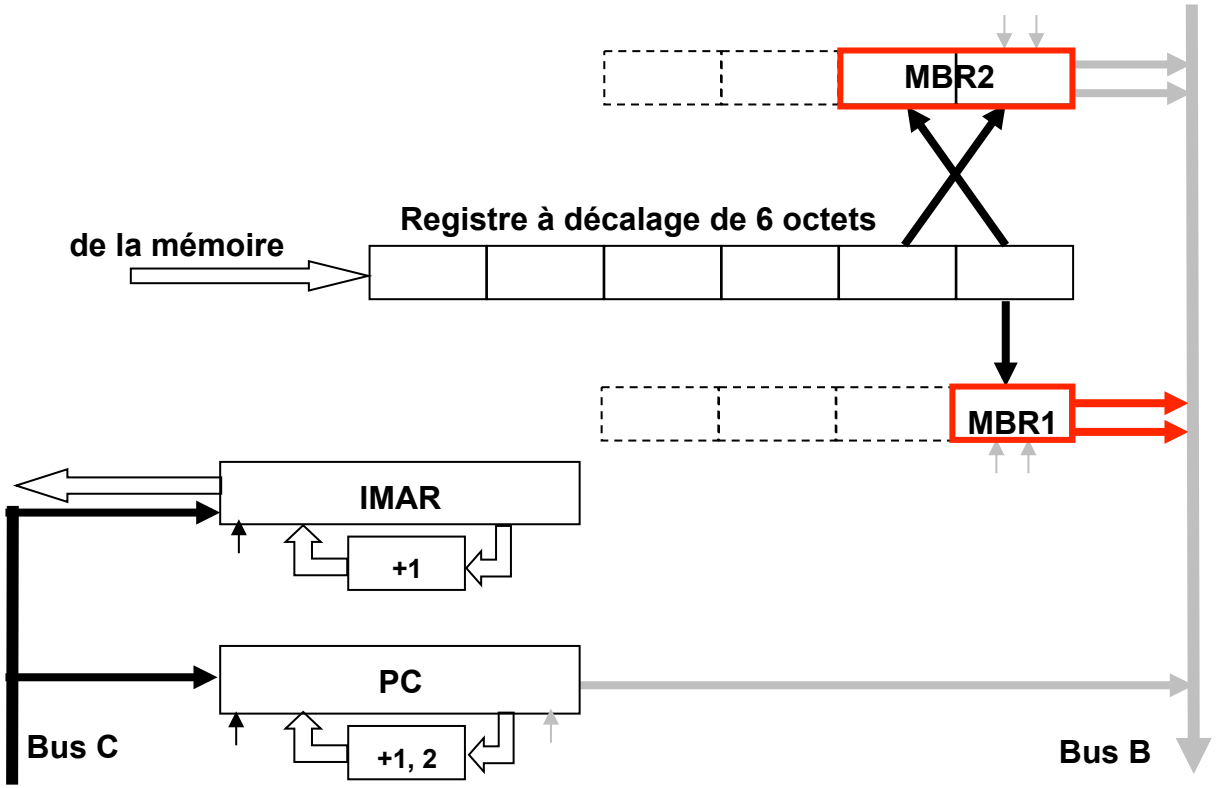
Le prochain octet est toujours disponible dans MBR1

L'unité possède deux interfaces vers le bus B MBR1 et MBRU1: Produisant une extension de l'octet sur 32 bits par extension de signe (MBR1) et remplissage à zéro (MBRU1)

Même chose pour le registre 16 bits MBR2

5. Optimisation de l'architecture Mic-1

3. Schéma de l'Unité de Recherche d'Instruction

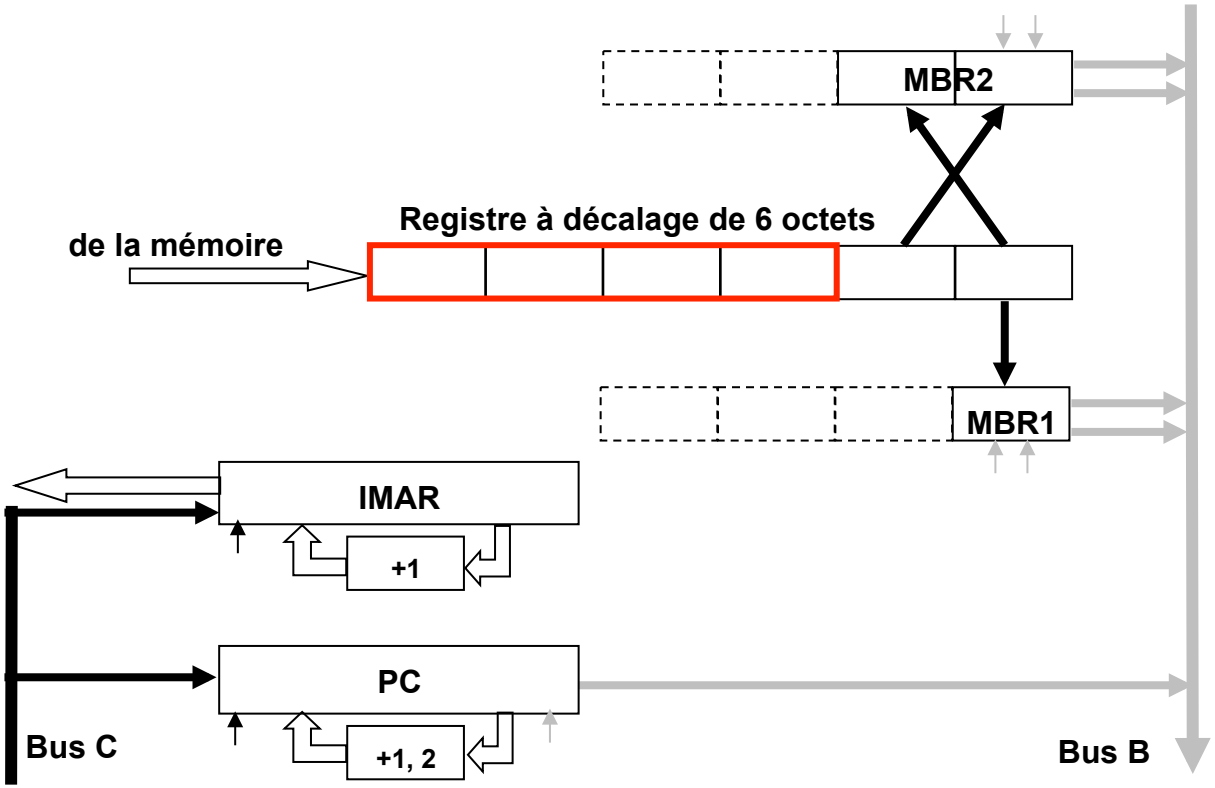


MBR1 contient l'octet le plus ancien du registre à décalage

MBR2 contient les deux octets les plus anciens du registre à décalage le plus ancien étant placé à gauche pour former un entier de 16 bits

5. Optimisation de l'architecture Mic-1

3. Schéma de l'Unité de Recherche d'Instruction

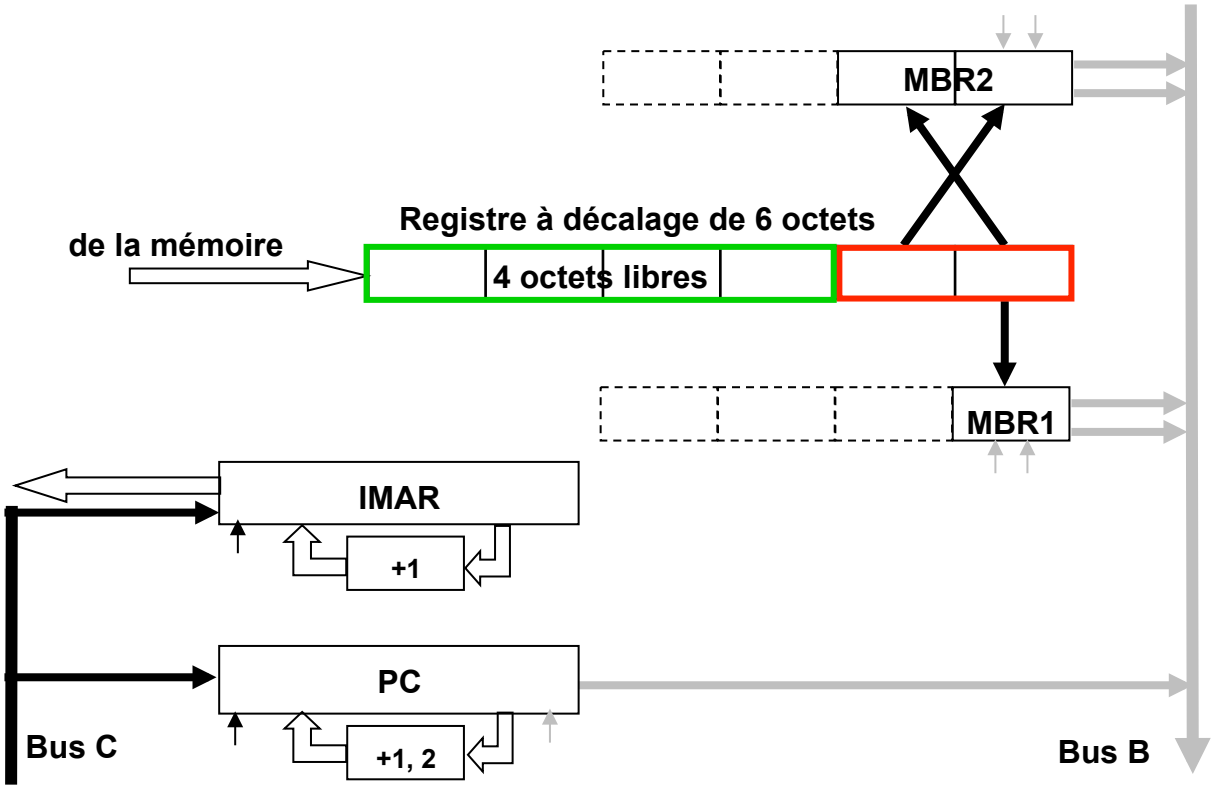


Le registre à décalage est chargé par lecture en mémoire par l'unité, de mots de 32 bits

Les deux octets les plus anciens du registre à décalage servent à alimenter MBR1 et MBR2

5. Optimisation de l'architecture Mic-1

3. Schéma de l'Unité de Recherche d'Instruction



Lorsque c'est MBR1 qui est lu
Le registre est décalé d'un octet

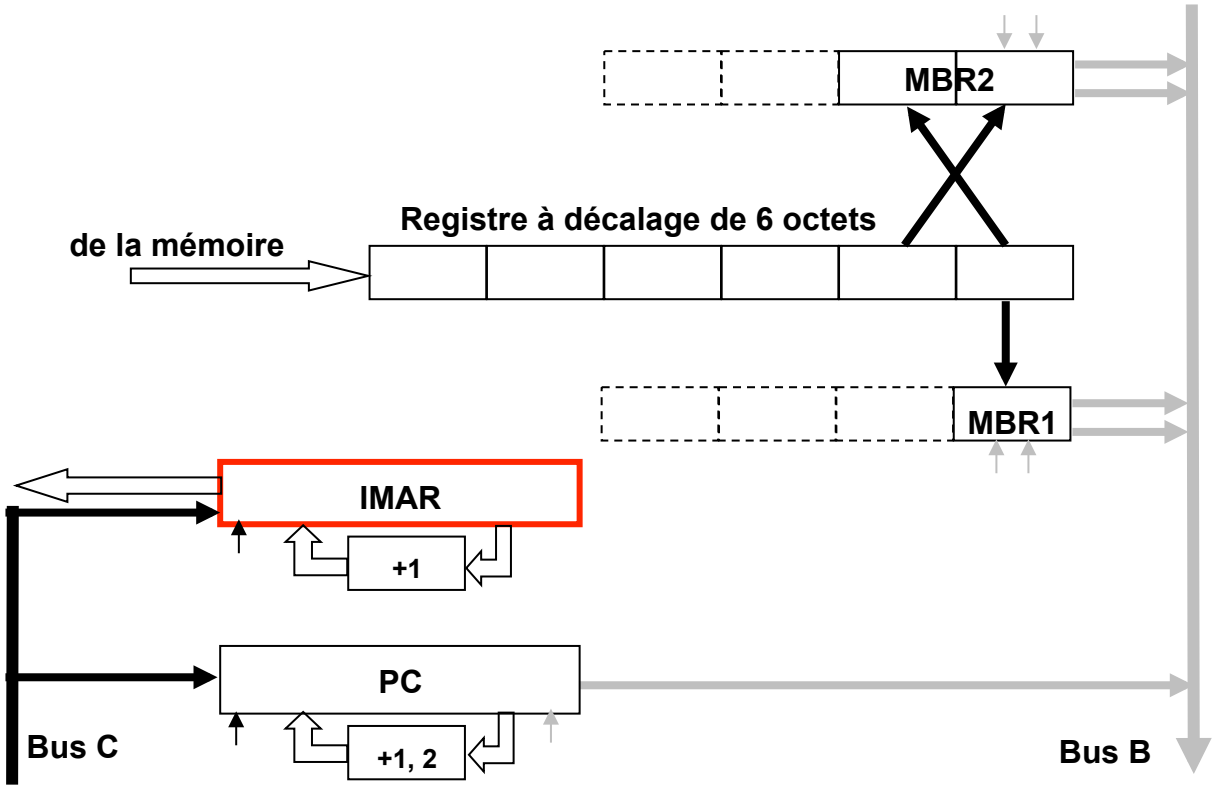
Lorsque c'est MBR2 qui est lu, le registre est décalé de deux octets

Lorsque l'un des registres est lu, il est systématiquement rempli au départ du cycle d'horloge suivant

Une lecture d'un mot mémoire est demandée par l'unité dès que le registre à décalage possède 4 octets de libres

5. Optimisation de l'architecture Mic-1

3. Schéma de l'Unité de Recherche d'Instruction



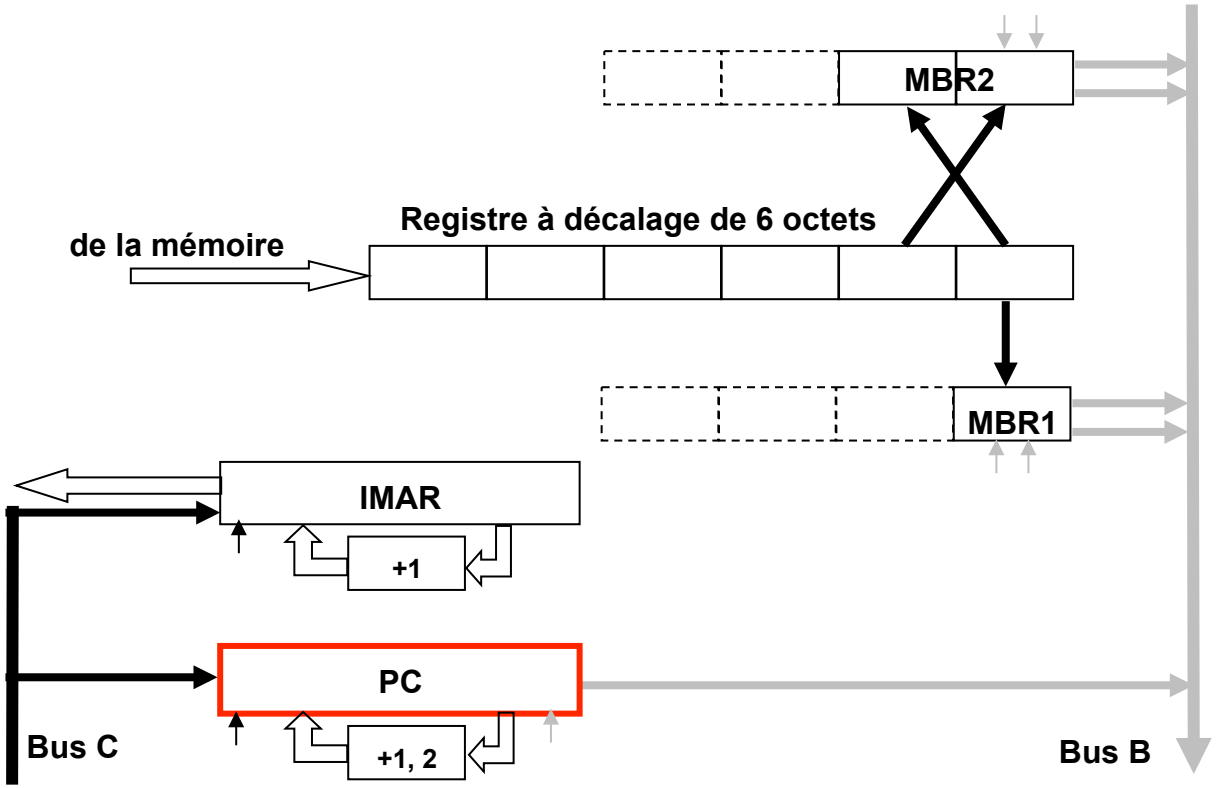
Le registre IMAR contient l'adresse du mot à charger dans l'unité
Il possède son propre incrémenteur
Pour délester l'UAL.

Dès que le PC est chargé, l'unité doit mettre à jour IMAR
Le PC peut pointer un octet qui n'est pas à la frontière d'un mot de 32 bits.

Dans ce cas l'unité doit rechercher le mot de 32 bits à adresser et ajuster le registre à décalage comme il faut lors du chargement de ce registre

5. Optimisation de l'architecture Mic-1

3. Schéma de l'Unité de Recherche d'Instruction



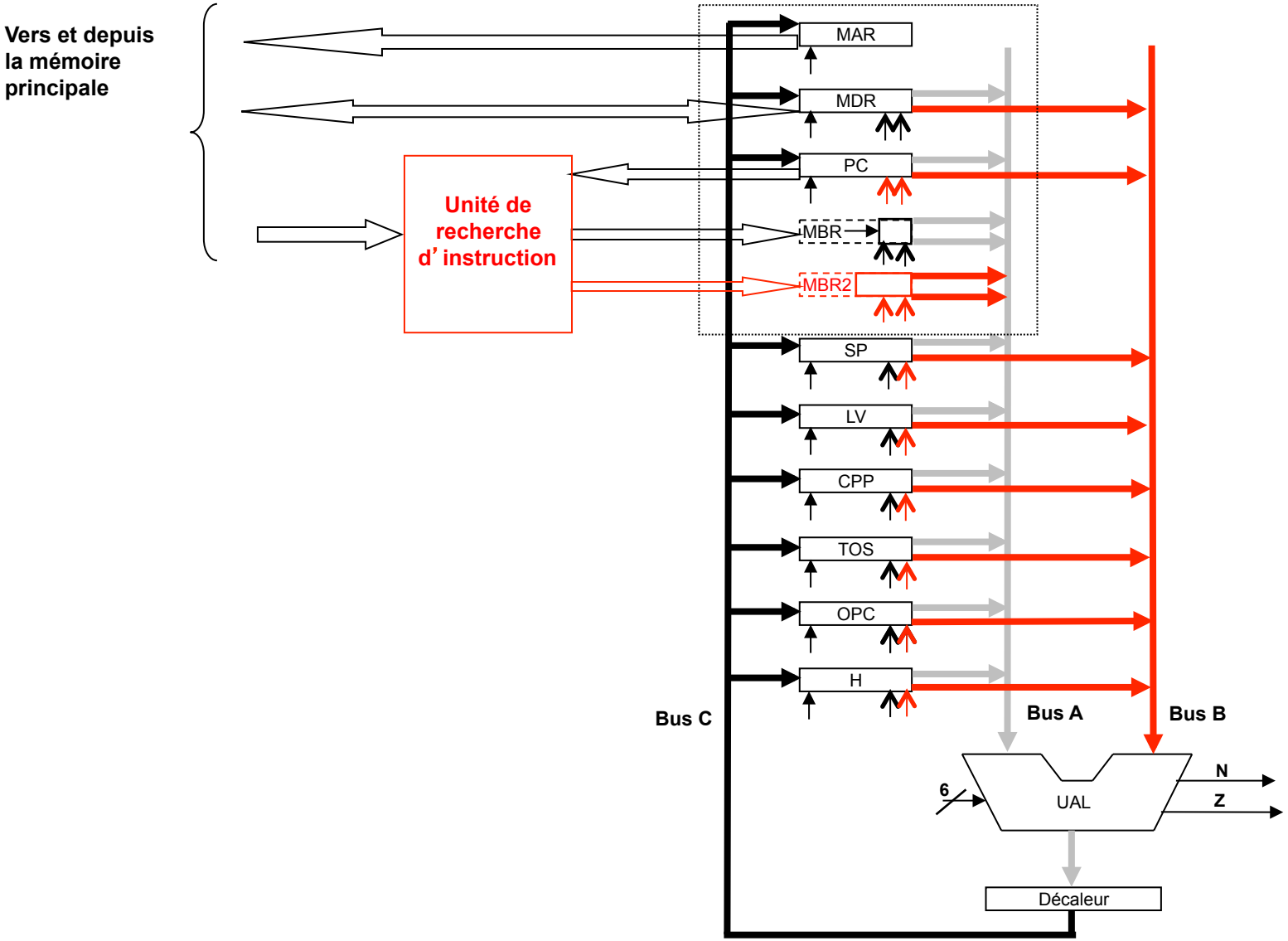
Le registre PC est incrémenté de 1 ou 2 dès la lecture de MBR1 ou MBR2 respectivement
Il possède son propre incrémenteur

PC contient toujours l'adresse du premier octet non encore utilisé, donc de la prochaine instruction

MBR comporte l'adresse du code opération de cette même prochaine instruction

Remarque:
PC compte en octet les instruction
IMAR compte en mots de 32 bits

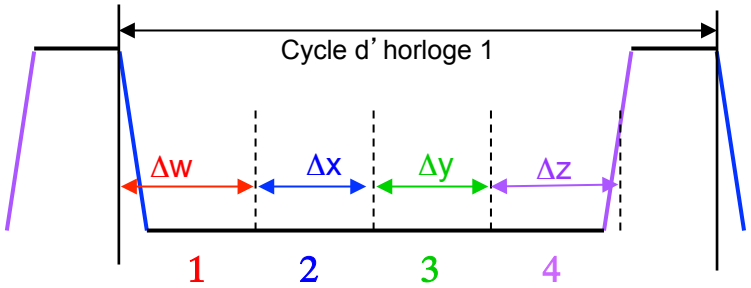
5. Optimisation de l'architecture Mic-1: Mic2



6. Optimisation de Mic2

Un second niveau d'optimisation va consister à introduire plus de parallélisme dans l'architecture

Rappel du cycle du chemin des donnée:



Sous-cycle 1: activation des signaux de commande
Sous cycle 2 : écriture sur Bus
Sous-cycle 3 : opération sur l'UAL
Sous-cycle 4 : résultat disponible sur Bus C

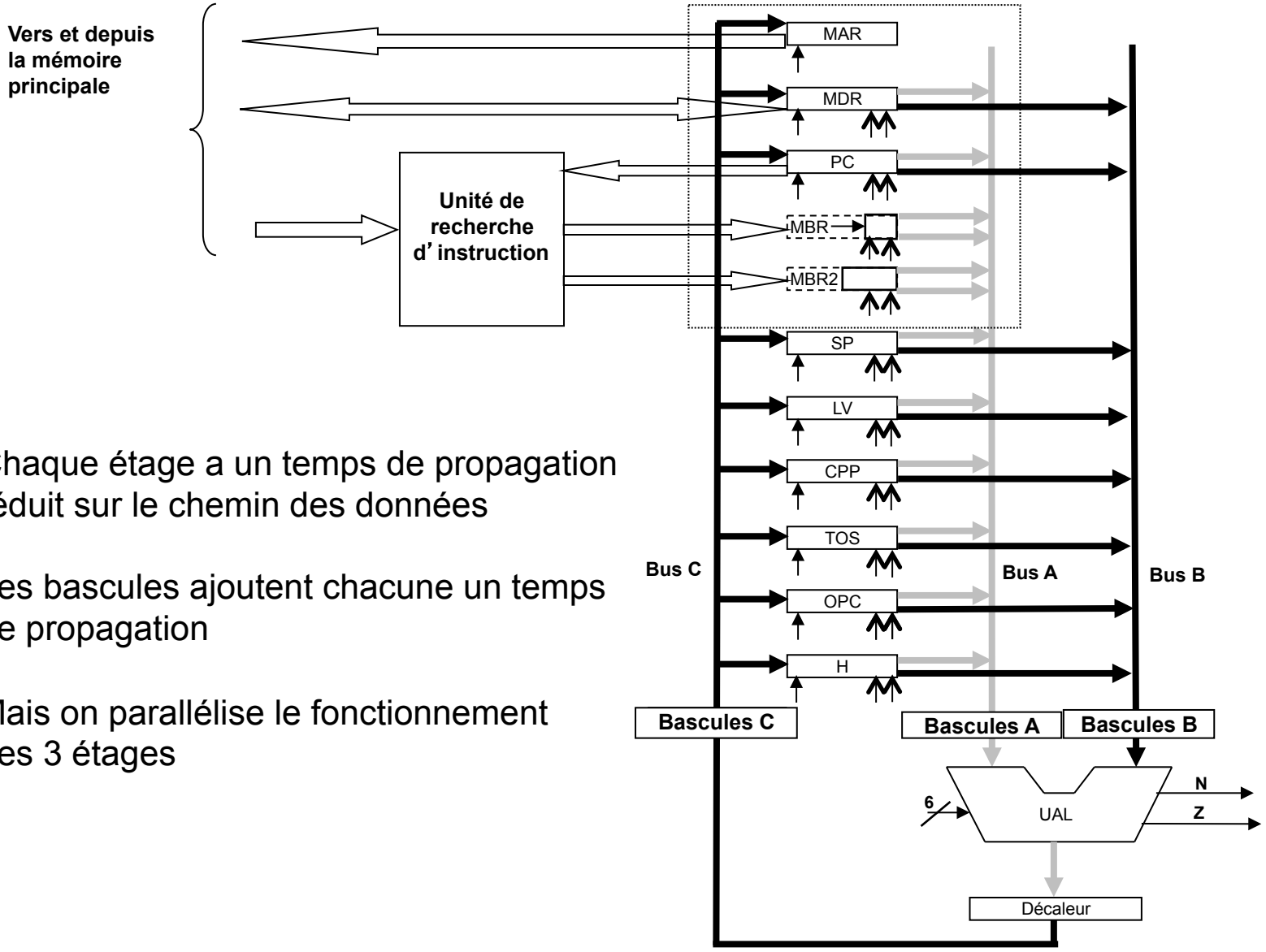
L'enchaînement des sous-cycles en séquences permet de distinguer trois étages sur le chemin des données

Etage 1: activation des signaux de commande & écriture sur Bus
Etage 2:Sous-cycle 3 : opération sur l'UAL
Etage 4 : résultat disponible sur Bus C

Afin de paralléliser ces trois étages on introduit
- **des bascules sur les Bus A & B**
- **des bascules sur le Bus C**

ces registres, commandés par l'horloge, vont avoir pour effet de cloisonner les étages et permettre de les faire fonctionner en parallèle

6. Optimisation de Mic 2

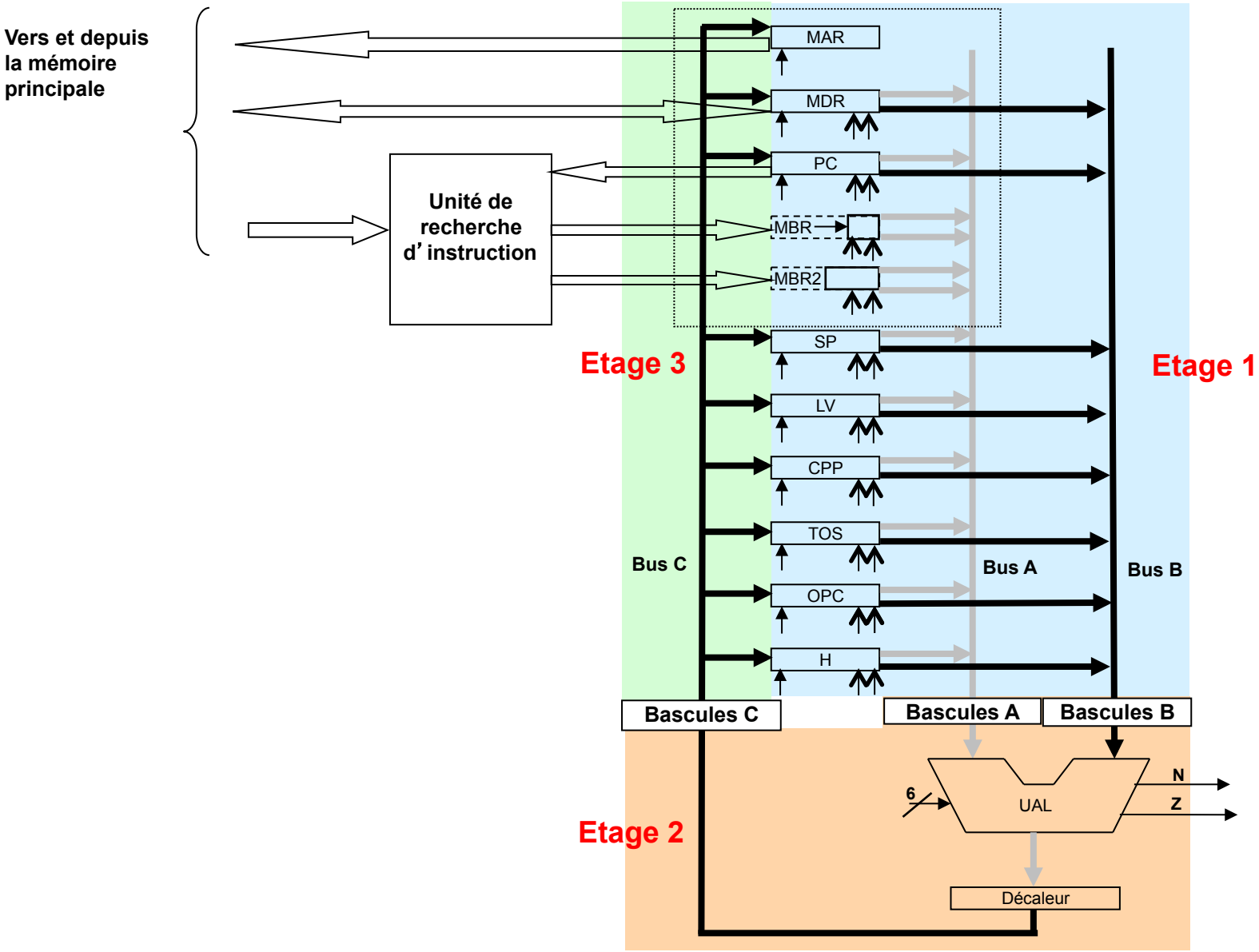


Chaque étage a un temps de propagation réduit sur le chemin des données

Les bascules ajoutent chacune un temps de propagation

Mais on parallélise le fonctionnement des 3 étages

6. Optimisation de Mic 2



6. Fonctionnement du PipeLine sur Mic 3

l'instruction swap sur Mic 1 ou Mic 2

Label	Micro-instructions	Description
swap1	MAR = SP-1; rd	Charge MAR et demande une lecture du second mot dans la pile
swap2	MAR = SP	Prépare MAR pour écrire en haut de la pile
swap3	H = MDR; wr	Copie dans H le second mot lu; et l'écrit en haut de la pile
swap4	MDR = TOS	Récupère le contenu du haut de pile précédent
swap5	MAR = SP-1; wr	Demande son écriture en second dans la pile
swap6	TOS = H; goto MBR1	Le registre TOS est actualisé avec la bonne donnée; branchement à l'instruction suivante

6. Fonctionnement du PipeLine sur Mic 3

l'instruction swap sur Mic 3

	swap1	swap2	swap3	swap4	swap5	swap6
Cycles de Mic 1	MAR=SP-1;rd	MAR=SP	H=MDR;wr	MDR=TOS	MAR=SP-1;wr	TOS=H;goto MBR
Cycles de Mic 2						
1	B=SP					
2	C=B-1	B=SP				
3	MAR=C;rd	C=B				
4		MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C;wr	C=B	B=SP	
8				MDR=C	C=B-1	B=H
9					MAR=C;wr	C=B
10						TOS=C
11						Goto MBR

swap3 est dépendante de la lecture en mémoire lancée dans swap1 au cycle 3
elle ne peut donc débuter qu'au cycle 5 au lieu du cycle 3 (normalement

6. Fonctionnement du PipeLine sur Mic 3

Comparaison des temps d'exécution sur Mic2 et Mic3

Sur Mic3 il faut 11 cycles d'horloge **11 Δt**

Sur Mic2 chaque cycle d'horloge est globalement trois fois plus long que sur Mic 3
il faut 6 cycles d'horloge pour exécuter l'instruction swap, c'est-à-dire 6 X 3 Δt **= 18 Δt**

On a gagné environ 1/3 du temps d'exécution de l'instruction swap grâce au pipeline

Bibliographie

Andrew Tanenbaum, Architecture de l'ordinateur, 4eme édition, Dunod, 2001.

Andrew Tanenbaum, Architecture de l'ordinateur, 5eme édition, PearsonEducation, 2005.

David Patterson, John Hennessy, Organisation et conception des ordinateurs, l'interface matériel/logiciel, Dunod, 1994.