Architecture des ordinateurs
Cours 4

Responsable de l'UE : Emmanuelle Encrenaz
Supports de cours : Karine Heydemann

Contact : emmanuelle.encrenaz@lip6.fr

Plan du cours 4

Architecture générale d'un ordinateur

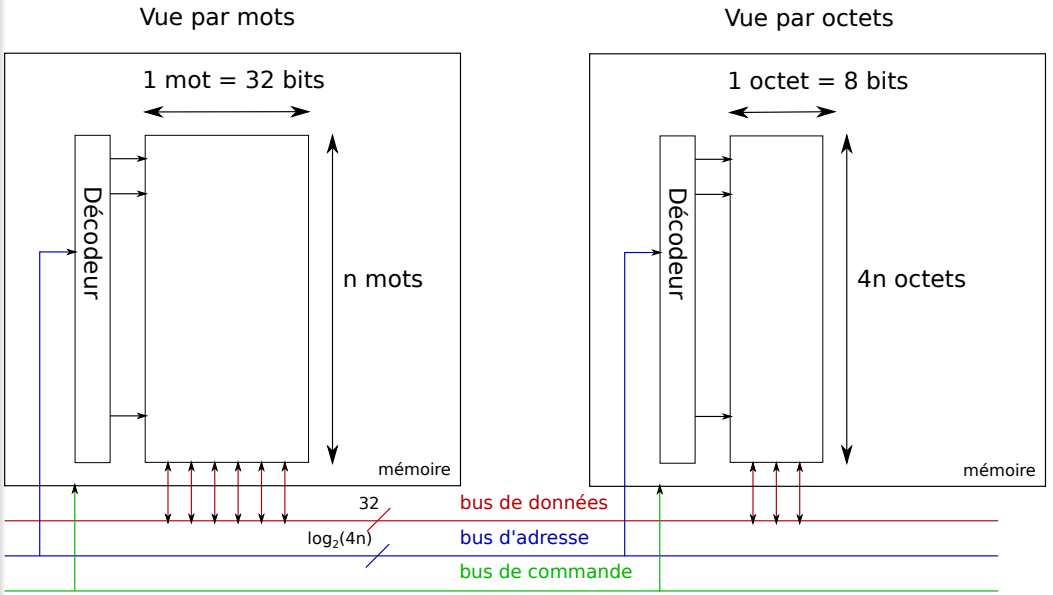
- Le processeur (ou CPU) est l'unité de traitement de l'information (instructions et données). Il exécute des programmes (suite d'instructions qui définissent un traitement à appliquer à des données).
- La mémoire centrale (ou RAM ou mémoire vive) est une unité de stockage temporaire des informations nécessaires à l'exécution d'un programme. Externe au processeur, elle stocke en particulier les instructions du programme en cours d'exécution ou à exécuter, et les données du programme (nombres, caractères alphanumériques, adresses mémoire, ...).
- Le bus est le support physique des transferts d'information entre les différentes unités.
- Les périphériques sont des unités connexes permettant de communiquer avec l'ensemble processeur-mémoire : clavier, écran, disque dur, réseau, imprimante/scanner, ...

La mémoire et son fonctionnement

Architecture de la mémoire

Généralité

- La mémoire peut être vue comme un tableau de n mots de 32 bits ou de 4n octets de 8 bits.
- La position d'un octet définit son adresse :
  adresse du 1er octet = 0,
  adresse du 2ème = 1,
  ...
  adresse du dernier = 4n-1
- On dit que l'unité adressable est l'octet : on peut lire ou écrire au minimum un octet



- En Mips, on peut écrire ou lire au plus 4 octets à la fois ; dans ce cas, l'adresse du mot (de 4 octets) est l'adresse la plus petite des 4 octets du mot, et cette adresse doit être un multiple de 4

Capacité mémoire

Définition

- La capacité mémoire correspond au nombre d'octets qu'elle peut stocker, l'unité de la taille d'une mémoire est donc un nombre d'octets

Capacité de stockage

- 1 kilo-octet ou 1 Kio = 2^{10} = 1 024 octets
- 1 mega-octet ou 1 Mio = 2^{20} = 1 048 576 octets
- 1 giga-octet ou 1 Gio = 2^{30} octets
- 1 tera-octet ou 1 Tio = 2^{40} octets

- Lire la page wikipédia "*Préfixes binaires*"

Les différents types de mémoire

Il existe différentes mémoires (usages différents ou mêmes usages mais caractéristiques différentes)

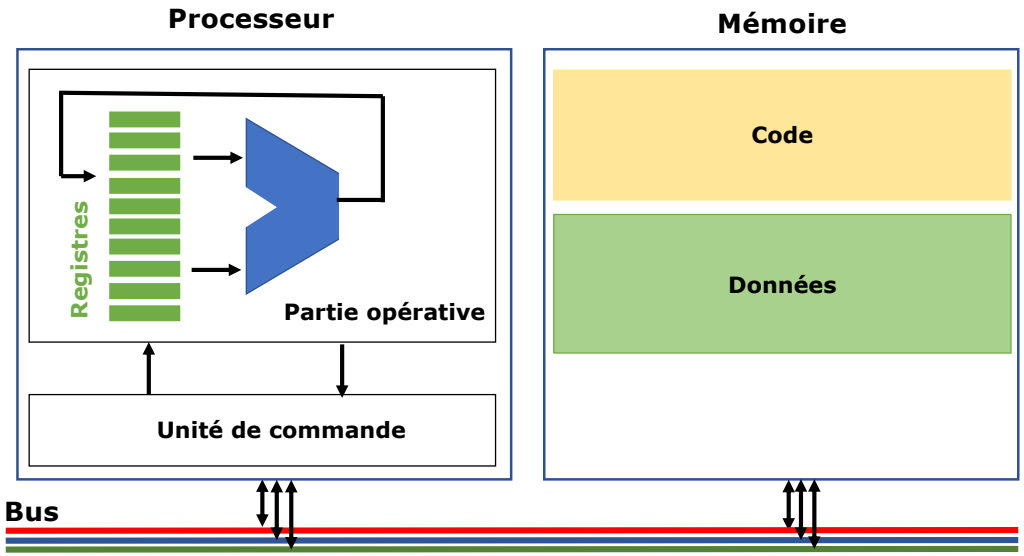
- Mémoire non réinscriptible ou ROM, assez lente d'accès : contient le code de démarrage, ou l'ensemble du code dans certains petits systèmes embarqués
- Mémoire vive ou RAM : mémoire volatile et rapide d'accès, contient le code et les données manipulées par le processeur pendant l'exécution d'un programme
- Mémoire Flash : mémoire non volatile, réinscriptible (mais nombre d'écritures limitées) et rapide d'accès : utilisée pour stocker code ou données non modifiables d'un programme, dans les clés USB ou certains appareils numériques pour du stockage long terme.

On s'intéresse dans ce cours à la mémoire RAM contenant le code et les données

Transfert entre la mémoire et le processeur

Le processeur initie les transferts de données entre le processeur et la mémoire et indique/donne à la mémoire :

- L'**adresse** du mot à transférer
- La **taille** du mot à transférer
- Le **sens du transfert** :
  - si processeur → mémoire alors c'est une **écriture** ou un *store*,
  - si mémoire → processeur alors c'est une **lecture** ou un *load* (chargement).
- La donnée à écrire si écriture

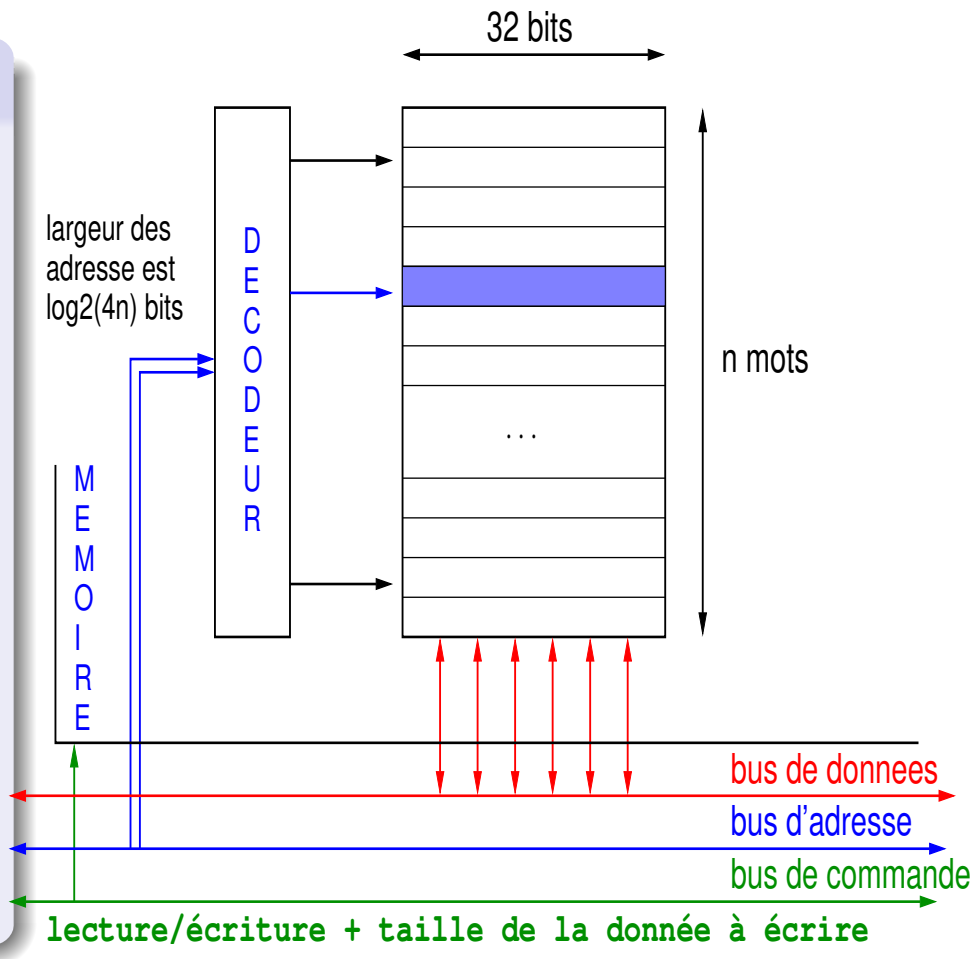


- Effet d'une écriture : jusqu'à la prochaine écriture à la même adresse (sinon jusqu'à la fin du programme)
- Une lecture n'est pas destructrice

Fonctionnement de la mémoire

Lors d'un transfert

- le décodeur d'adresse sélectionne la ligne correspondant à l'adresse demandée sur le bus d'adresse
- la commande indique l'opération : lecture ou écriture
- Si lecture : les données lues sont mises sur le bus de données
- Si écriture : les données à écrire sont présentes sur le bus de données



Rangement de mots de plusieurs octets

- Une donnée de plusieurs octets est rangée à des adresses contiguës

- Soit $M = 0x o_3 o_2 o_1 o_0$ un mot de 4 octets rangé à l'adresse A
- Il y a 2 rangements possibles :
  - Big Endian (grand boutien) : l'octet de poids fort est rangé à l'adresse la plus petite
  - Little Endian (petit boutien) : l'octet de poids faible est rangé à l'adresse la plus petite $\Rightarrow$ rangement utilisé en Mips
- Illustration avec une mémoire vue par octet

Adresse	A	A + 1	A + 2	A + 3
Little Endian	o_0	o_1	o_2	o_3
Big Endian	o_3	o_2	o_1	o_0

- Illustration avec une mémoire vue par mot (4 octets)

Adresse	A
Little Endian	$o_3 o_2 o_1 o_0$
Big Endian	$o_0 o_1 o_2 o_3$

Exemple d'écriture d'un mot

État initial de la mémoire

- Vue par octet :

Adresse	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	...
Contenu	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0x??	...

- Vue par mot :

Adresse	0x0	0x4	0x8	...
Contenu	0x????????	0x????????	0x????????	...

Effet de l'écriture du mot 0xAABBCCDD à l'adresse 0x4

- Vue par octet :

Adresse	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	...
Contenu	0x??	0x??	0x??	0x??	**0xDD**	**0xCC**	**0xBB**	**0xAA**	0x??	0x??	0x??	0x??	...

- Vue par mot :

Adresse	0x0	0x4	0x8	...
Contenu	0x????????	**0xAABBCCDD**	0x????????	...

Exemple d'écriture d'un demi-mot

État de la mémoire avant l'écriture

- Vue par octet :

Adresse	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	...
Contenu	0x??	0x??	0x??	0x??	0xDD	0xCC	0xBB	0xAA	0x??	0x??	0x??	0x??	...

- Vue par mot :

Adresse	0x0	0x4	0x8	...
Contenu	0x????????	0xAABBCCDD	0x????????	...

Effet de l'écriture du demi-mot 0x1234 à l'adresse 2

- Vue par octet :

Adresse	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	...
Contenu	0x??	0x??	**0x34**	**0x12**	0xDD	0xCC	0xBB	0xAA	0x??	0x??	0x??	0x??	...

- Vue par mot :

Adresse	0x0	0x4	0x8	...
Contenu	**0x1234**????	0xAABBCCDD	0x????????	...

Exemple d'écriture d'un octet

État de la mémoire avant l'écriture

- Vue par octet :

Adresse	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	...
Contenu	0x??	0x??	0x34	0x12	0xDD	0xCC	0xBB	0xAA	0x??	0x??	0x??	0x??	...

- Vue par mot :

Adresse	0x0	0x4	0x8	...
Contenu	0x1234????	0xAABBCCDD	0x????????	...

Effet de l'écriture de l'octet 0xFF à l'adresse 1

- Vue par octet :

Adresse	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	...
Contenu	0x??	**0xFF**	0x34	0x12	0xDD	0xCC	0xBB	0xAA	0x??	0x??	0x??	0x??	...

- Vue par mot :

Adresse	0x0	0x4	0x8	...
Contenu	0x1234**FF**??	0xAABBCCDD	0x????????	...

Uniformité et structuration de la mémoire

Uniformité de la mémoire

Mémoire

- Elle est modélisée comme un tableau de mots de 4 octets, est adressable en octets
- Elle stocke sous forme de suite d'octets tout ce qui est nécessaire l'exécution d'un programme : les données et instructions notamment.

Uniformité $\Rightarrow$ universalité

Cette uniformité confère à l'ordinateur (machine de Von Neumann) son universalité : les traitements exécutables (instructions) par un processeur sont précablés/finis mais l'agencement de traitements possibles est infini.

Les données et les instructions sont stockées sur le même support : ce qui est interprété à un instant donné comme une donnée peut être interprété comme une instruction en langage machine plus tard.

Uniformité de la mémoire : exemple

Un programme peut être utilisé pour obtenir de nouveaux programmes !

- Rédaction de programme source
  - donnée = programme texte (codage ASCII)
  - programme = éditeur de texte.

- Compilation d'un programme source en un binaire exécutable :
  - donnée d'entrée = programme source (texte codage ASCII)
  - programme = compilateur
  - donnée de sortie = programme binaire exécutable

- Exécution d'un programme binaire : pour exécuter un programme binaire, il faut d'abord le charger en mémoire. C'est le travaille du *loader*. Une fois le programme chargé, le loader donne la main au programme binaire exécutable.
  1. programme = loader, donnée = programme binaire.
  2. programme = le programme binaire exécutable, données = celles du programme correspondant.

Structuration de la mémoire

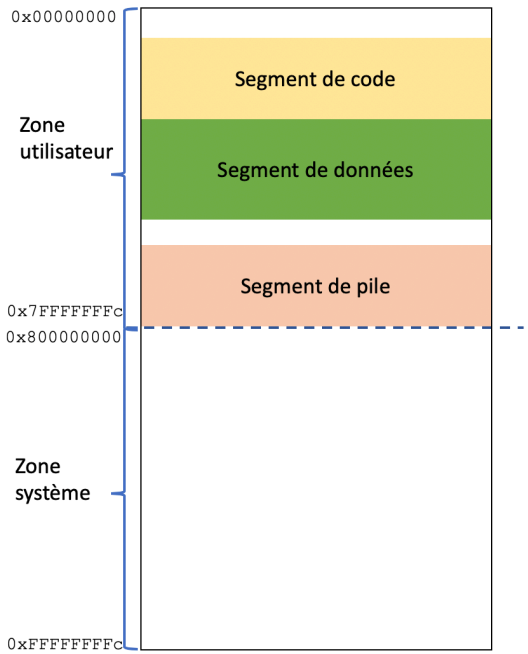
Besoin de structuration

Un programme n'est pas le seul objet présent en mémoire : il y a les données et programmes de l'OS, éventuellement d'autres utilisateurs de la machine.

Protections nécessaires :

- des informations du système vis-à-vis des programmes utilisateurs,
- des différents types d'informations au sein d'un même programme,
- des programmes vis à vis d'autres programmes.

Structuration de l'espace d'adressage



L'espage d'adressage (ensemble des adresses possibles) est découpé en deux parties distinctes :

1. une partie réservée au système (OS) : adresses accessibles uniquement en mode superviseur/super utilisateur,
2. une partie réservée aux utilisateurs : accès non restreint.

Les informations de même nature d'un programme sont regroupés en dans des zones distinctes appelées **segments** (espace d'adressage contigu muni d'une taille maximale). Au moins 3 segments par programme :

1. **segment de code** : pour les instructions codées en binaire
2. **segment de données** : pour les données et variables globales
3. **segment de pile** : mémoire temporaire allouée à la demande au cours de l'exécution pour les variables locales, les contextes d'exécution des sous-programmes et leurs paramètres d'appel.

Déclaration de données globales et implantation mémoire des données

Structure d'un programme assembleur Mips

En MIPS, tout programme assembleur est constitué d'au moins 2 sections : la section de données et la section de code (déjà vu)

Section de code : directive `.text`

- La directive `.text` désigne la section de code. Les instructions doivent se trouver dans cette section
- Elle spécifie que ce qui suit sera implanté dans le segment de code

Section de données : directive `.data`

- La directive `.data` désigne la section de données
- Les données globales doivent être allouées, et initialisées si besoin, dans cette section du programme
- Elle spécifie que ce qui suit correspond aux données globales : celles-ci seront implantées dans le segment de données

Déclaration des variables globales

Directives d'allocation et initialisation

- La syntaxe générale pour déclarer une variable globale est :
  `[nom:] directive [init]`
- Toutes les variables globales sont déclarées dans la section `.data`
- Les directives que l'on utilisera sont :
  `.word`, `.byte`, `.half`, `.space`, `.ascii`, `.asciiz`
- `nom:` est une déclaration d'*étiquette* de nom `nom`.

  Une étiquette désigne l'adresse de la première case allouée.

  Pour des raisons de lisibilité du code on utilise souvent le nom des variables globales correspondant à ces allocations.
- Les étiquettes de la section de données peuvent être utilisées dans des pseudo-instructions : nous ne les utilisons pas dans cette partie du cours.

Allocation et initialisation de données globales

Directives d'allocation utilisées dans ce cours :

- `.word VAL1,[VAL2, VAL3,...]` : alloue un mot mémoire (4 octets) initialisé avec la valeur `VAL1` (donnée en décimal ou hexadécimal si préfixée par `0x`), ou plusieurs mots consécutifs initialisés avec les valeurs `VAL1`, `VAL2`, `VAL3`,...
- `.half VAL1,[VAL2, VAL3,...]` : alloue un demi-mot initialisé avec la valeur `VAL1` (donnée en décimal ou hexadécimal si préfixée par `0x`), ou plusieurs demi-mots consécutifs initialisés avec les valeurs `VAL1`, `VAL2`, `VAL3`,...
- `.byte VAL1,[VAL2, VAL3,...]` : alloue un octet initialisé avec la valeur `VAL1` (donnée en décimal ou hexadécimal si préfixée par `0x`), ou plusieurs octets consécutifs initialisés avec les valeurs `VAL1`, `VAL2`, `VAL3`,...
- `.space NB` alloue `NB` octets (initialisés à 0 par défaut)
- `.asciiz CH` alloue le nombre d'octets nécessaire pour l'implantation consécutive de chaque caractère de la chaîne de caractères `CH` (doit être donnée entre "") en incluant le caractère de fin de chaîne (dont la valeur est 0x00).
- `.ascii CH` alloue le nombre d'octets nécessaire pour l'implantation consécutive de chaque caractère de la chaîne de caractères `CH` (doit être donnée entre ""). N'inclut PAS le caractère de fin de chaîne.

Exemples de déclaration de données globales

Utilisation des directives

- `n: .word 5` : réserve un mot et l'initialise à la valeur 5
- `n: .byte 1` : réserve un octet et l'initialise à la valeur 1
- `n: .half 12` : réserve un demi-mot et l'initialise à la valeur 12
- `buf: .space 20` : réserve 20 octets non initialisés (mis à 0 par défaut)
- `s: .asciiz "abc"` : alloue et initialise 4 octets pour la chaîne de caractères "abc", y compris le caractère de fin de chaîne
- `s: .ascii "abc"` : alloue et initialise 3 octets pour la chaîne de caractères "abc", sans le caractère de fin de chaîne (pas très utile pour nous)

Contrainte d'alignement des accès mémoire

- En MIPS, les accès à des données ne peuvent pas avoir lieu à n'importe quelle adresse
- Principe : les adresses accédées doivent être multiples de la taille de l'accès
- Les mots (4 octets) accédés (lecture ou écriture) doivent avoir une adresse qui est multiple de 4
- Les demi-mots (2 octets) accédés (lecture ou écriture) doivent avoir une adresse qui est multiple de 2

Directive d'alignement des données globales

- On peut forcer l'alignement des données en mémoire lors des déclarations de données en utilisant la directive `.align`
- `.align n` force l'adresse de la déclaration suivante à avoir une adresse multiple de 2^n
- MARS aligne automatiquement les données déclarées avec des directives `.half` ou `.word`. L'alignement est implicite.
- Par contre l'utilisation de la direction `.space` pour déclarer des données non initialisées (mais initialisées à 0) nécessite l'utilisation de la directive `.align n` lorsqu'il s'agit de mots ou demi-mots
- On précédera toujours les déclarations de mots (resp. de demi-mots) avec la directive `.space` de la directive `.align 2` (resp. `.align 1`) pour expliciter les contraintes d'alignement mémoire des données qu'on alloue.

Alignement explicite

```
       .align 1  # alignement sur des adresses de demi-mots
buf1:  .space 20 # réserve 10 demi-mots (initialisés à 0)

       .align 2  # alignement sur des adresses de mots
buf2:  .space 20 # réserve 5 mots (initialisés à 0)
```

Exemple de code avec des données globales

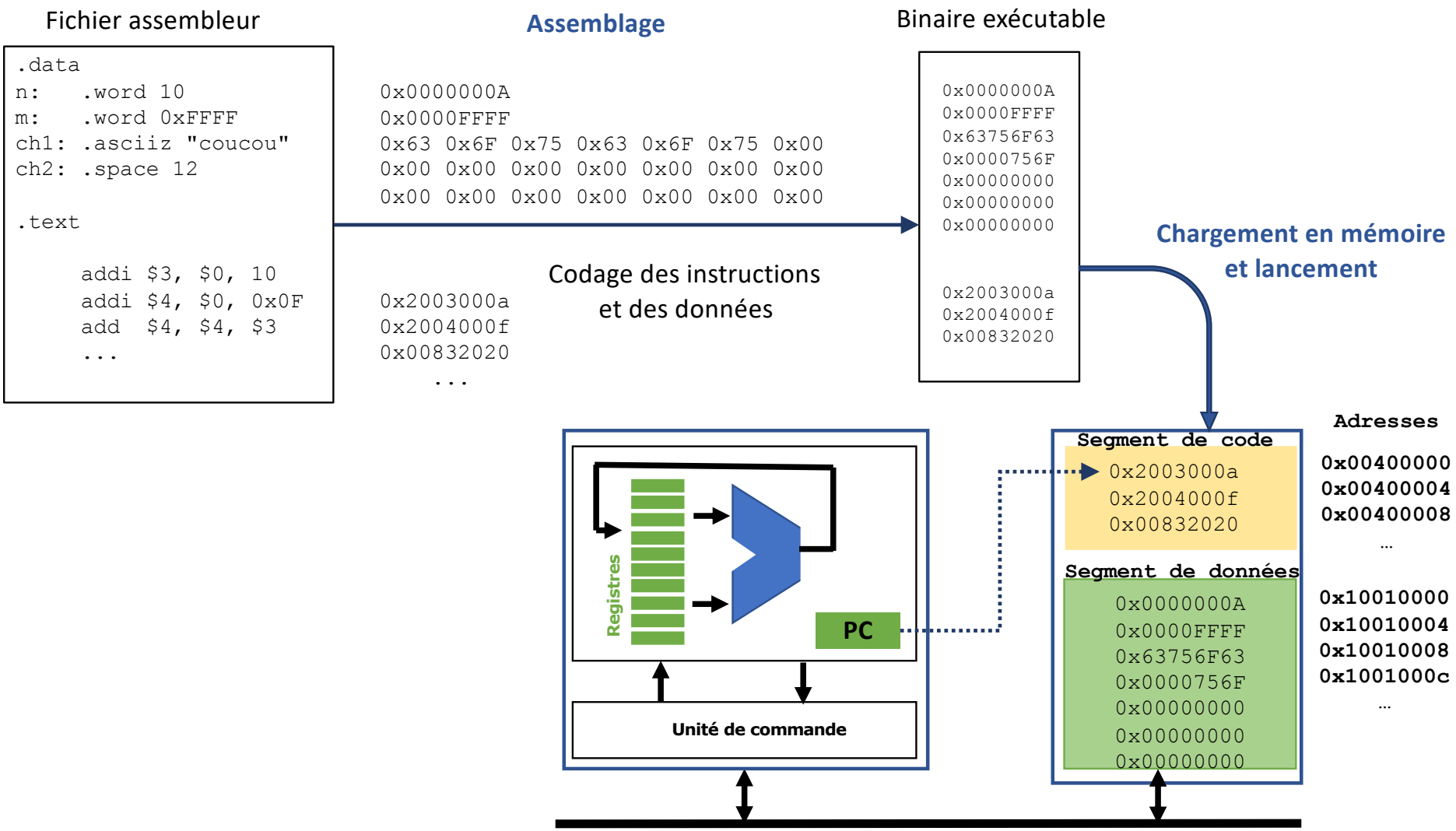
Allocation et initialisation de données + quelques instructions (qui n'utilisent pas les données) :

```
.data
n:  .word 10        # entier 10 codé sur 4 octets
m:  .word 0xFFFF    # entier 0xFFFF sur 4 octets

ch1: .asciiz "coucou" # allocation et initialisation d'une chaine de caractères

ch2: .space 12      # allocation d'un espace de 12 octets
                    # non initialisés (mis à0)

.text
   addi $3, $0, 10
   addi $4, $0, 0x0F
   add $4, $4, $3
   ori $2, $0, 10
   syscall
```

Assemblage et lancement d'un programme

- Une fois un programme assembleur écrit, il faut l'assembler afin de créer un programme binaire exécutable (déjà vu au cours 3)
- Pour exécuter un programme, il faut d'abord le charger en mémoire $\Rightarrow$ Fait par un programme du système appelé *loader* (déjà vu au cours 3)
- La section de code est rangée dans le segment de code et la section de données est rangée dans le segment de données
- La première instruction de la section `.text` est implantée à l'adresse `0x00400000` et les suivantes sont rangées consécutivement en mémoire selon leur ordre d'apparition dans le fichier assembleur
- La première donnée de la section `.data` est implantée à l'adresse `0x10010000` et les données déclarées ensuite sont allouées en séquence dans le segment de données
- L'adresse du point d'entrée du programme est mise (par le loader) dans le registre PC, ce qui engendre l'exécution de la première instruction... et donc du programme (déjà vu au cours 3)

Assemblage et lancement d'un programme



Adresse des données et instructions

- Lors du chargement du programme les données et instructions sont rangées dans leur segment mémoire respectif. Elles sont rangées consécutivement dans leur ordre d'apparition dans le fichier assembleur dans leur segment mémoire respectif.
- La section `.text` est implantée à l'adresse `0x00400000`, la section `.data` à l'adresse `0x10010000`

⇒ On peut calculer les adresses d'implantation (et leur représentation) :

Calcul de l'adresse d'une variable globale

- **Hypothèse** : la section `.data` est implantée au début du segment data, à l'adresse `0x10010000`
- **Contrainte d'alignement** : l'adresse d'une donnée (simple) doit être un multiple de sa taille
  - Par exemple, l'adresse d'un mot doit être un multiple de 4, donc se terminer par 0x0, 0x4, 0x8 ou 0xc
- L'adresse de la première variable `v1` est toujours l'adresse `0x10010000`
- L'adresse de la deuxième variable `v2` est `@v1 + taille(v1) + alignement`
- L'adresse de la n-ième variable `vn` est `@vn-1 + taille(vn-1) + alignement`
- `alignement` est la plus petite valeur (positive ou nulle) permettant de respecter la contrainte d'alignement explicite ou implicite

Adresses des données globales

Exemple

```
.data
n:   .word 0x20   # @ =
c0:  .byte 0x31   # @ =
tab: .word 0x0, 0x1 # @ =
str: .asciiz "Entrez un nombre :" # @ =
p:   .word 0x10   # @ =
c1:  .byte 0xFF   # @ =
c2:  .byte 0xFE   # @ =
     .align 2
m1:  .space 4     # @ =
```

Adresses des données globales

Exemple

```
.data
n:   .word 0x20       # @ = 0x10010000
c0:  .byte 0x31       # @ = 0x10010000 + 4_d           = 0x10010004
tab: .word 0x0, 0x1   # @ = 0x10010004 + 1_d + align(3) = 0x10010008
str: .asciiz "Entrez un nombre :" # @ = 0x10010008 + 8_d = 0x10010010
p:   .word 0x10       # @ = 0x10010010 + 19_d + align(1) = 0x10010024
c1:  .byte 0xFF       # @ = 0x10010024 + 4_d           = 0x10010028
c2:  .byte 0xFE       # @ = 0x10010028 + 1_d           = 0x10010029
     .align 2
m1:  .space 4         # @ = 0x10010029 + 1_d + align(2) = 0x1001002c
```

Adresse des données et des instructions

```
.data                    # début du segment de données 0x10010000

n:  .word 10             # 0x0000000A adresse 0x10010000
m:  .word 0xFFFF         # 0x0000FFFF adresse 0x10010000 + 4 = 0x10010004

ch1: .asciiz "coucou" # adresse de ch1 = 0x10010004 + 4 = 0x10010008
                      # représentation 0x63 0x6F 0x75 0x63 0x6F 0x75 0x00
ch2: .space 12        # adresse ch2 = adresse ch1 + taille (ch1)
                      # adresse ch2 = 0x10010008 + 7 = 0x1001000F

.text                 # début du segment de code 0x00400000

    addi $3, $0, 10   # inst 0x2003000a adresse 0x00400000
    addi $4, $0, 0x0F # inst 0x2004000f adresse 0x00400004
    add $4, $4, $3    # inst 0x00832020 adresse 0x00400008
    ori $2, $0, 10    # inst 0x3402000a adresse 0x0040000c
    syscall           # inst 0x0000000c adresse 0x00400010
```

Manipulation de données globales en assembleur : instruction de transfert mémoire

Manipulation de variables globales

- On sait déclarer des variables globales : données globales
- On sait déterminer où elles se trouvent en mémoire (leur adresse d'implantation)
- Comment les utiliser ? Par exemple, comment réaliser :
  - `b = a + 5`

  $\Rightarrow$ signifie
  1. lire la variable `a`
  2. ajouter 5
  3. modifier la variable `b` avec le résultat
- Lire une variable : la valeur de la variable est dans la mémoire, il faut donc l'amener dans le processeur
  $\Rightarrow$ transfert mémoire, sens mémoire $\rightarrow$ processeur / lecture mémoire
- Écrire une variable : la variable correspond à un emplacement mémoire qui doit être mis à jour avec la nouvelle valeur pour la variable
  $\Rightarrow$ transfert mémoire, sens processeur $\rightarrow$ mémoire / écriture mémoire
- Il faut utiliser des instructions de transfert mémoire

Instruction de transferts mémoire (1)

- Ces instructions lisent (*load*) ou écrivent (*store*) des données en mémoire
- Elles utilisent les bus d'adresse, de données et de commande pour réaliser le transfert mémoire
- Syntaxe des instructions d'accès mémoire :

  `Codop Rt, Imm`$_{16}$`(Rs)`
- Sens du transfert du transfert et taille du mot transféré indiquée dans l'opération :

  `lw`, `lh`, `lb`, `sw`, `sh`, `sb`
- L'adressage est **indexé** : adresse accédée = contenu `Rs` + immédiat signé `Imm`$_{16}$ sur 16 bits
  $\Rightarrow$ 2 opérandes sources pour le calcul d'adresse : `Rs` et l'immédiat `Imm`$_{16}$
  $\Rightarrow$ Utilisation de l'ALU pour ce calcul, extension signée de l'immédiat `Imm`$_{16}$
- `Rt` contient la valeur à écrire en mémoire (si écriture) ou recevra la valeur lue en mémoire (si lecture)

Exemples

- `lh $4,4($3)` : lecture d'1 demi-mot à partir de l'adresse `4+$3` et le ranger dans `$4`
- `sw $5,-2($8)` : écriture à l'adresse `-2+$8` du mot (4 octets) contenu dans `$5`

Instructions de lecture mémoire

- Une lecture de n octets (n = 1,2 ou 4) réalise le transfert
  `Rt ← MEM[Rs+Imm:Rs+Imm+(n-1)]`

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	0xBB	0xAA	0x11	0x22	0x33	0x44	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	0xAABBCCDD	0x44332211	...

- Lecture d'un mot `lw $4,0($3)` :
  `$4←MEM[$3+0:$3+3]`
  si `$3` contient `0x10010004` alors `$4` contiendra `0x44332211`

Instructions de lecture mémoire

- Une lecture de n octets (n = 1,2 ou 4) réalise le transfert
  `Rt ← MEM[Rs+Imm:Rs+Imm+(n-1)]`

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	0xBB	0xAA	0x11	0x22	0x33	0x44	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	0xAABBCCDD	0x44332211	...

- Lecture signée d'un demi-mot `lh $4,2($3)` :
  `$4←SignedExt32(MEM[$3+2:$3+3])`
  si `$3` contient `0x10010000` alors `$4` contiendra `0xFFFFAABB`

- Lecture non signée d'un demi-mot `lhu $4,2($3)` :
  `$4←UnsignedExt32(MEM[$3+2:$3+3])`
  si `$3` contient `0x10010000` alors `$4` contiendra `0x0000AABB`

Instructions de lecture mémoire

- Une lecture de n octets (n = 1,2 ou 4) réalise le transfert
  `Rt ← MEM[Rs+Imm:Rs+Imm+(n-1)]`

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	0xBB	0xAA	0x11	0x22	0x33	0x44	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	0xAABBCCDD	0x44332211	...

- Lecture signée d'un octet `lb $4,3($3)` :
  `$4←SignedExt32(MEM[$3+3])`
  si `$3` contient `0x10010000` alors `$4` contiendra `0xFFFFFFAA`

- Lecture non signée d'un octet `lbu $4,3($3)` :
  `$4←UnsignedExt32(MEM[$3+3])`
  si `$3` contient `0x10010000` alors `$4` contiendra `0x000000AA`

Instructions d'écriture mémoire

- Une écriture de n octets (n = 1,2 ou 4) réalise le transfert
  `Rt`$\rightarrow_{n}$`MEM[Rs+Imm:Rs+Imm+(n-1)]` :

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	0xBB	0xAA	0x11	0x22	0x33	0x44	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	0xAABBCCDD	0x44332211	...

- Écriture d'un mot `sw $4,0($3)` :

  `$4`$\rightarrow_{4}$`MEM[$3+0:$3+3]`

  si `$3` contient `0x10010004` et `$4` contient `0x55667788` alors la mémoire contiendra le mot `0x55667788` à cette adresse après l'écriture

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	0xBB	0xAA	**0x88**	**0x77**	**0x66**	**0x55**	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	0xAABBCCDD	**0x55667788**	...

Instructions d'écriture mémoire

- Une écriture de n octets (n = 1,2 ou 4) réalise le transfert
  `Rt`$\rightarrow_n$`MEM[Rs+Imm:Rs+Imm+(n-1)]` :

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	0xBB	0xAA	0x11	0x22	0x33	0x44	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	0xAABBCCDD	0x44332211	...

- Écriture d'un demi-mot `sh $4,2($3)` :

  `$4`$\rightarrow_2$`MEM[$3+2:$3+3]`

  si `$3` contient `0x10010000` et `$4` contient `0x5566`**`7788`** alors la mémoire contiendra le demi-mot **`0x7788`** à l'adresse `0x10010002` après l'écriture

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	**0x88**	**0x77**	0x11	0x22	0x33	0x44	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	**0x7788**CCDD	0x44332211	...

Instructions d'écriture mémoire

- Une écriture de n octets (n = 1,2 ou 4) réalise le transfert
  `Rt`$\rightarrow_n$`MEM[Rs+Imm:Rs+Imm+(n-1)]` :

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	0xBB	0xAA	0x11	0x22	0x33	0x44	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	0xAABBCCDD	0x44332211	...

- Écriture d'un octet `sb $4,3($3)` :

  `$4`$\rightarrow_1$`MEM[$3+3]`

  si `$3` contient `0x10010000` et `$4` contient `0x556677`**`88`** alors la mémoire contiendra l'octet **`0x88`** à l'adresse `0x10010003` après l'écriture

Adresse 0x10010000	+ 0	+1	+2	+3	+4	+5	+6	+7	...
Contenu	0xDD	0xCC	0xBB	**0x88**	0x11	0x22	0x33	0x44	...

Adresse par mot	0x10010000	0x10010004	...
Contenu	0x**88**BBCCDD	0x44332211	...

Suite d'instructions pour un accès à la mémoire

Si la variable `a` (de type entier) est implantée à l'adresse `0x10010004` et initialisée, si la variable `b` (de type entier) est implantée à l'adresse `0x10010008`, quelle suite d'instructions pour réaliser `b = a + 5` ?

- On doit disposer d'un registre contenant l'adresse à accéder ou permettant de réaliser un accès via l'adressage indexé
  ⇒ Instructions pour charger une adresse de 32 bits dans un registre
  ⇒ Instruction d'accès à la mémoire utilisant ce registre et un immédiat pour que l'adresse accédée soit celle voulue

```
# on est dans la section de code
lui $3, 0x1001
ori $3, $3, 4   # chargement de l'adresse de a
lw $4, 0($3)    # lecture mémoire àl'adresse de a
addi $4, $4, 5  # calcul de a + 5
sw $4, 4($3)    # écriture de a + 5 en mémoire à
                # l'adresse de b
```

Suite d'instructions pour un accès à la mémoire

Si la variable `a` (de type entier) est implantée à l'adresse `0x10010004` et initialisée, si la variable `b` (de type entier) est implantée à l'adresse `0x10010008`, quelle suite d'instructions pour réaliser `b = a + 5` ?

- On doit disposer d'un registre contenant l'adresse à accéder ou permettant de réaliser un accès via l'adressage indexé
  ⇒ Instructions pour charger une adresse de 32 bits dans un registre
  ⇒ Instruction d'accès à la mémoire utilisant ce registre et un immédiat pour que l'adresse accédée soit celle voulue

```
# on est dans la section de code
lui $3, 0x1001
ori $3, $3, 4  # chargement de l'adresse de a
lw $4, 0($3)   # lecture mémoire àl'adresse de a
addi $4, $4, 5 # calcul de a + 5
sw $4, 4($3)   # écriture de a + 5 en mémoire à
               # l'adresse de b
```

Suite d'instructions pour un accès à la mémoire

Si la variable `a` (de type entier) est implantée à l'adresse `0x10010004` et initialisée, si la variable `b` (de type entier) est implantée à l'adresse `0x10010008`, quelle suite d'instructions pour réaliser `b = a + 5` ?

- On doit disposer d'un registre contenant l'adresse à accéder ou permettant de réaliser un accès via l'adressage indexé
  ⇒ Instructions pour charger une adresse de 32 bits dans un registre
  ⇒ Instruction d'accès à la mémoire utilisant ce registre et un immédiat pour que l'adresse accédée soit celle voulue

```
# on est dans la section de code
lui $3, 0x1001
ori $3, $3, 4   # chargement de l'adresse de a
lw $4, 0($3)    # lecture mémoire àl'adresse de a
addi $4, $4, 5 # calcul de a + 5
sw $4, 4($3)    # écriture de a + 5 en mémoire à
                # l'adresse de b
```

Suite d'instructions pour un accès à la mémoire

Si la variable `a` (de type entier) est implantée à l'adresse `0x10010004` et initialisée, si la variable `b` (de type entier) est implantée à l'adresse `0x10010008`, quelle suite d'instructions pour réaliser `b = a + 5` ?

Programme complet

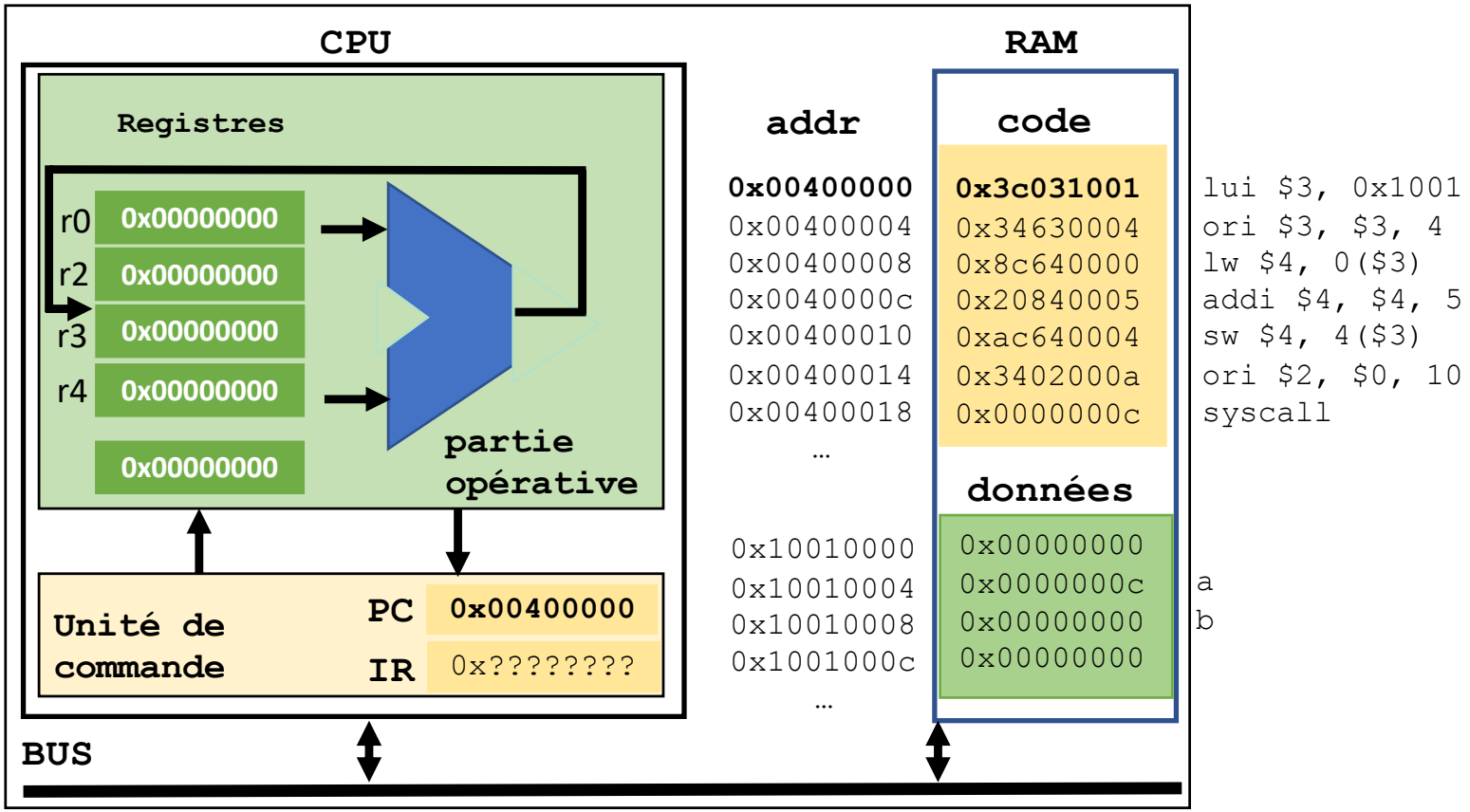
```
.data
    .space 4  # pour que a soit àla bonne adresse
a:  .word 12
    .align 2  # contrainte alignement d'un entier
b:  .space 4  # non initialisée

.text
  lui $3, 0x1001
  ori $3, $3, 4   # chargement de l'adresse de a
  lw $4, 0($3)    # lecture mémoire àl'adresse de a
  addi $4, $4, 5 # calcul de a + 5
  sw $4, 4($3)    # écriture de a + 5 en mémoire à
                  # l'adresse de b
  ori $2, $0, 10 # terminaison
  syscall
```

Programme complet

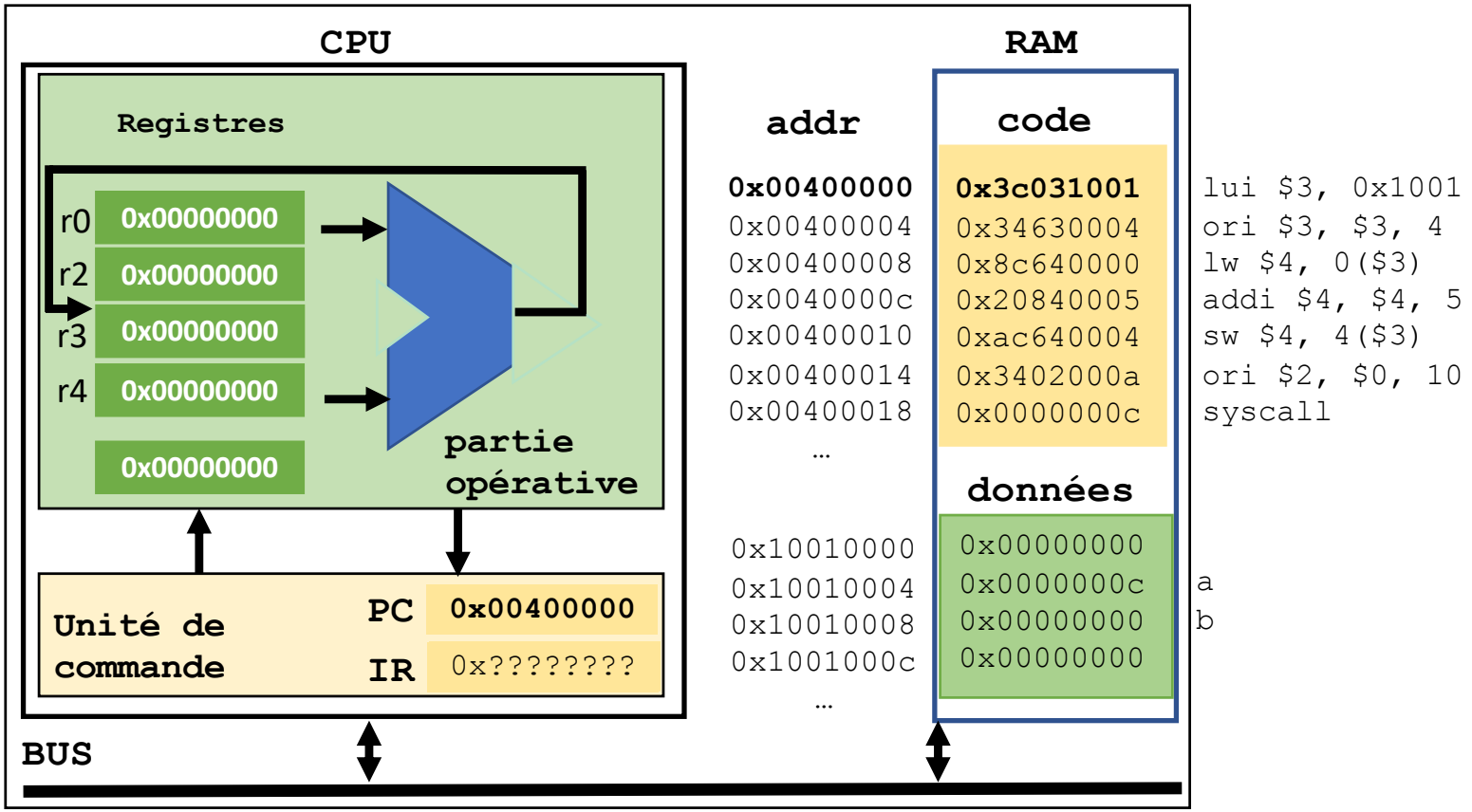
Programme complet avec le codage des données et instructions, ainsi que leur adresse mémoire

```
.data
   .space 4  # adresse 0x10010000 valeur 0x00000000
a: .word 12  # adresse 0x10010004 valeur 0x0000000c
   .align 2  # contrainte d'alignement d'un entier
b: .space 4  # adresse 0x10010008 valeur 0x00000000

.text
  lui $3, 0x1001 # adresse 0x00400000 inst 0x3c031001
  ori $3, $3, 4  # adresse 0x00400004 inst 0x34630004
  lw $4, 0($3)   # adresse 0x00400008 inst 0x8c640000
  addi $4, $4, 5 # adresse 0x0040000c inst 0x20840005
  sw $4, 4($3)   # adresse 0x00400010 inst 0xac640004

  ori $2, $0, 10 # adresse 0x00400014 inst 0x3402000a
  syscall        # adresse 0x00400018 inst 0x0000000c
```

Chargement du programme

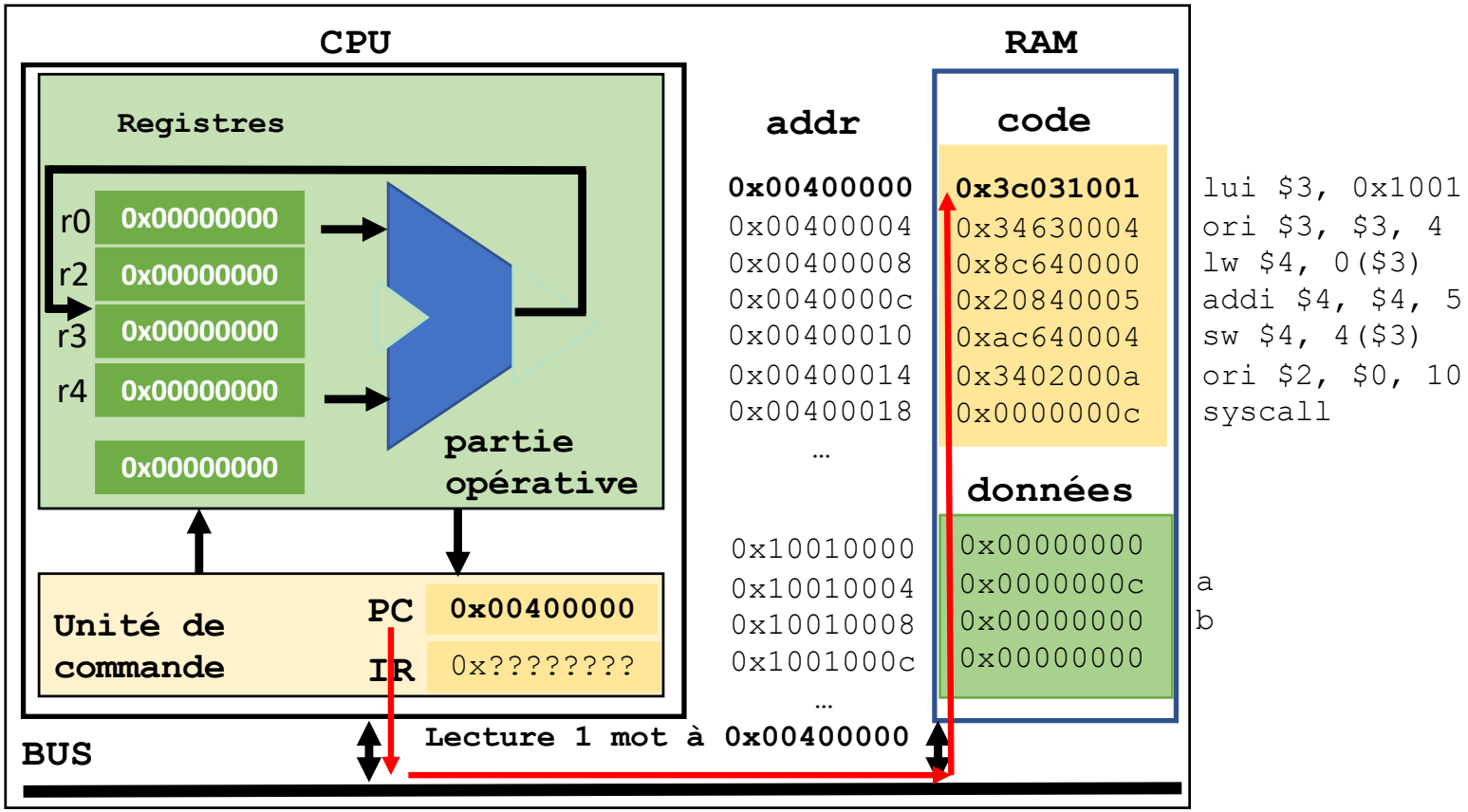


Exécution pas à pas du programme



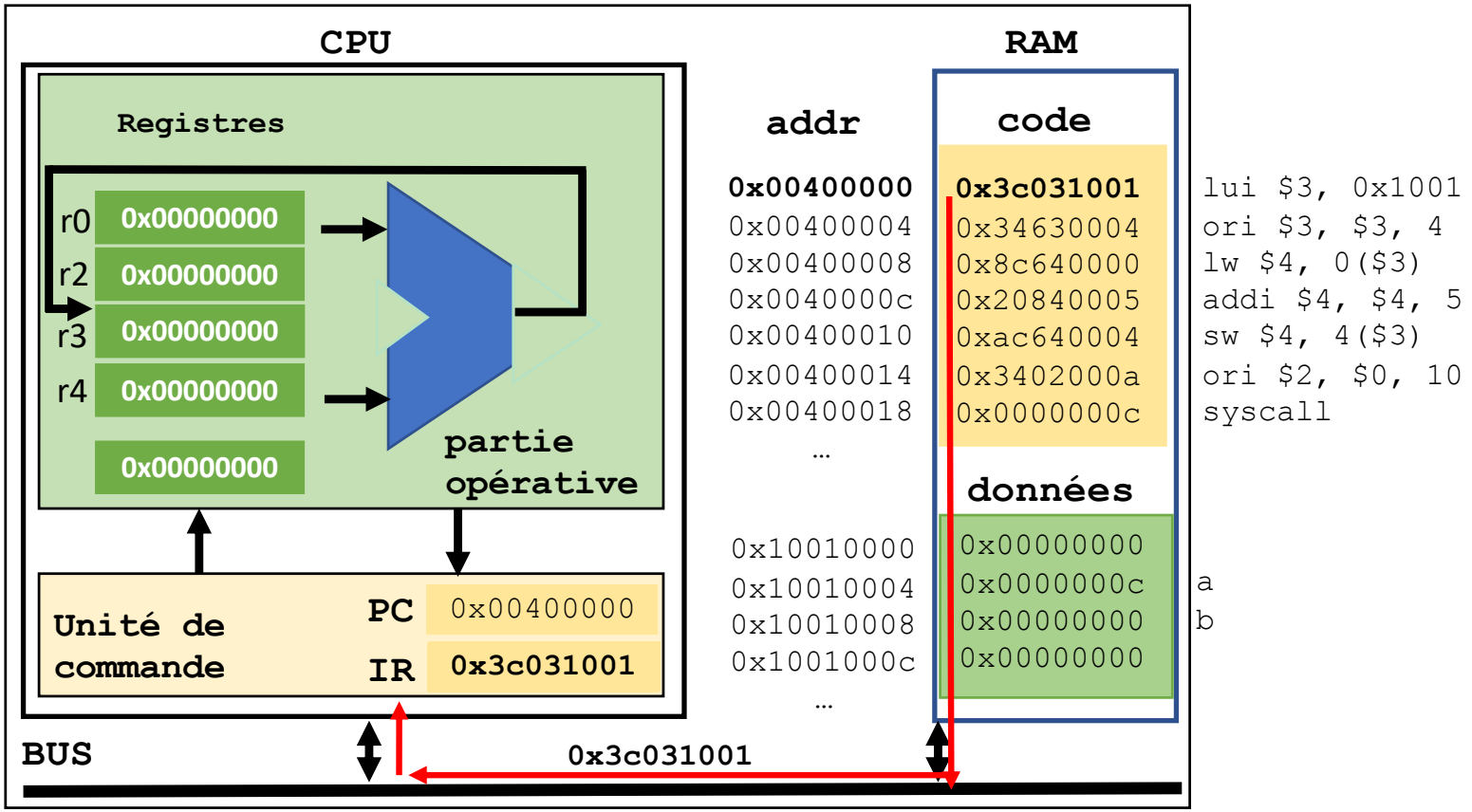
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



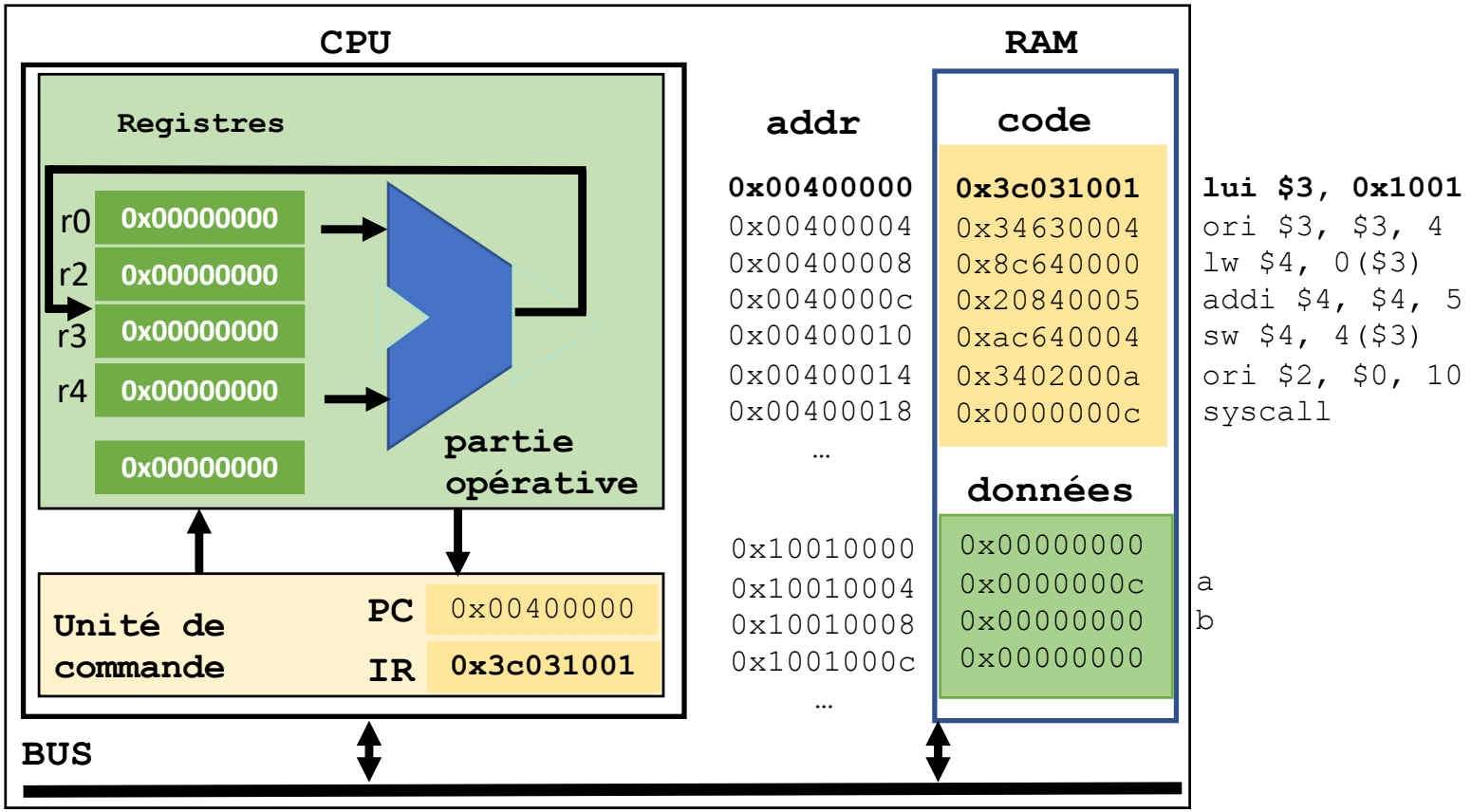
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



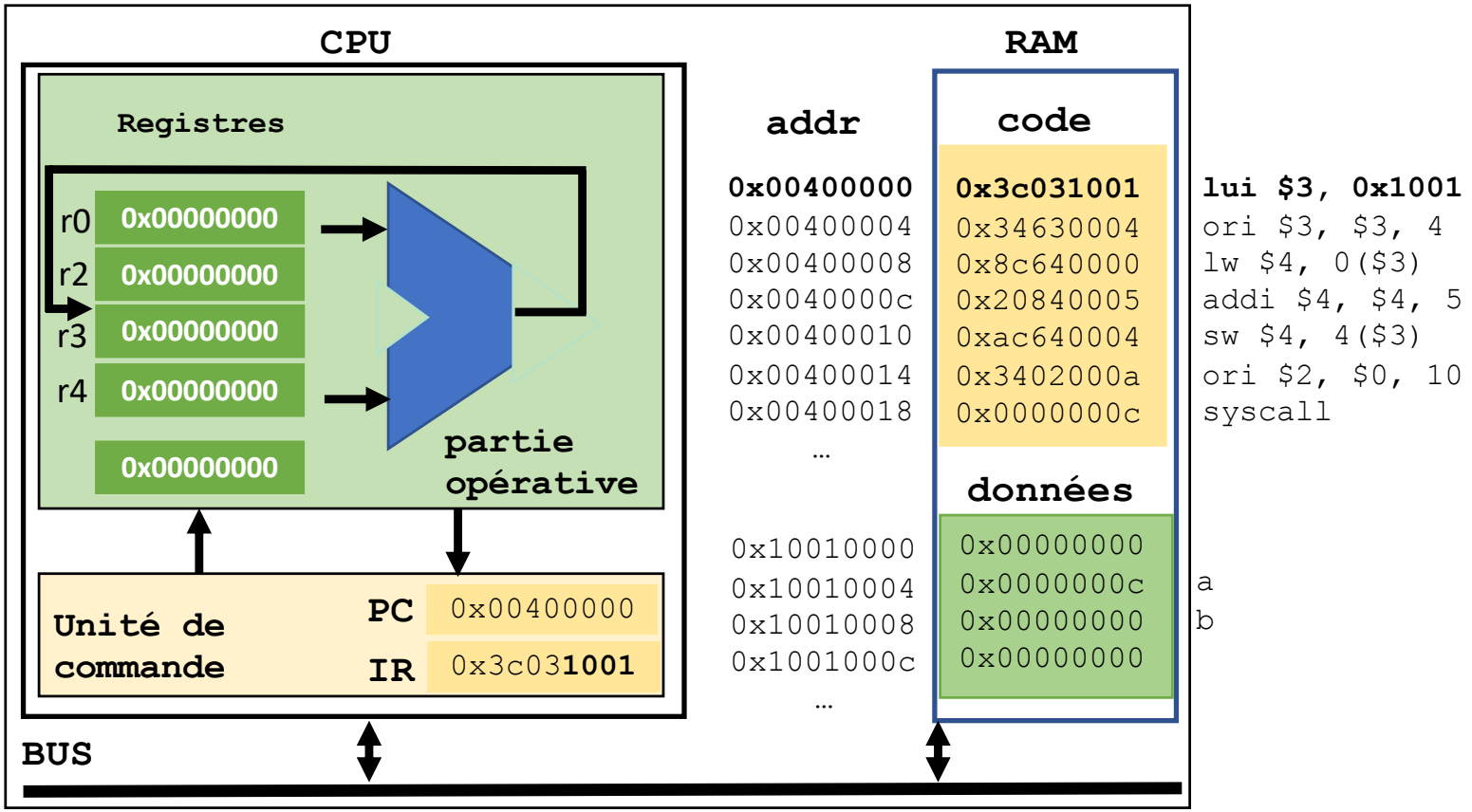
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



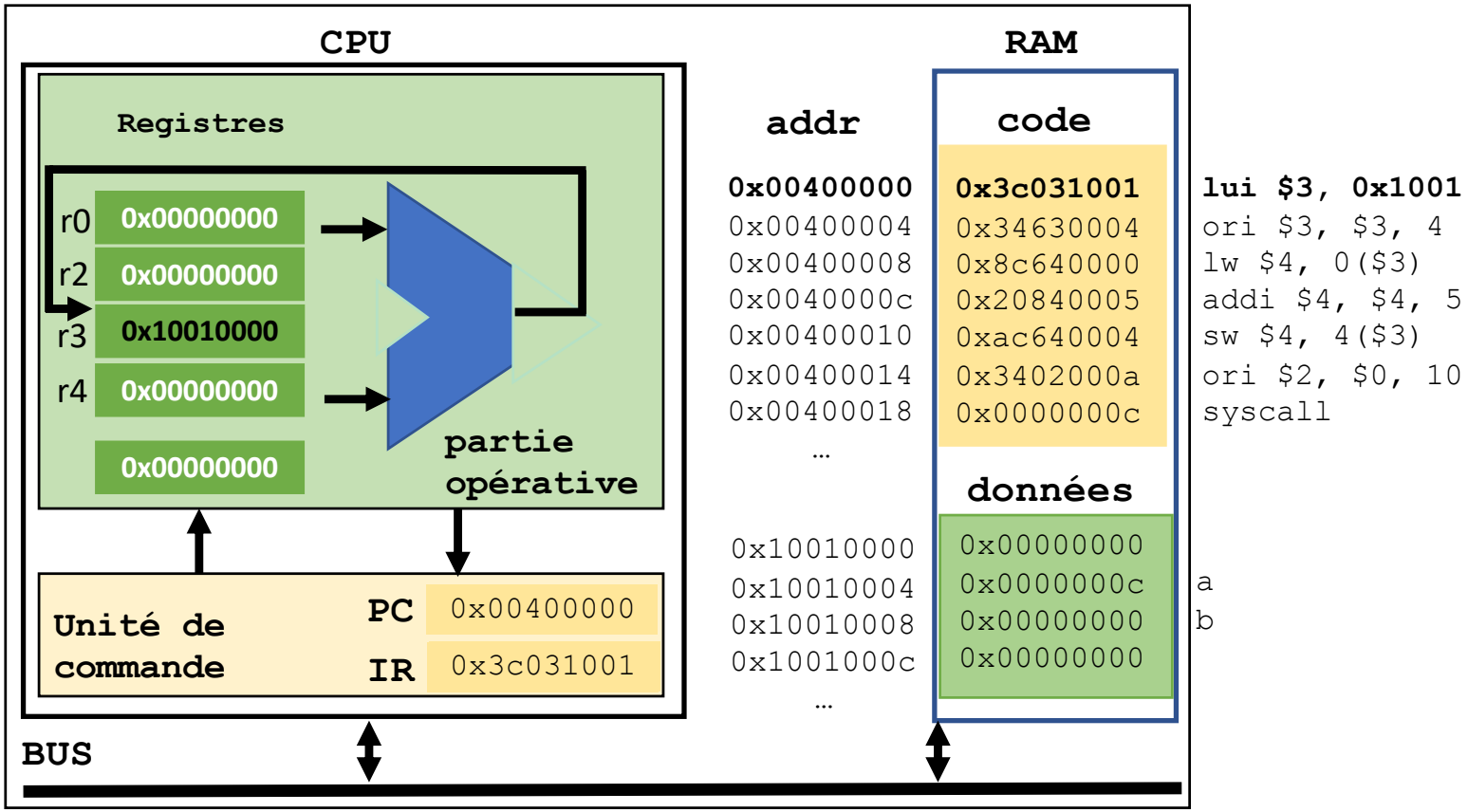
2 – Décoder l'instruction : affectation partie haute du registre $3 avec un immédiat

Exécution pas à pas du programme



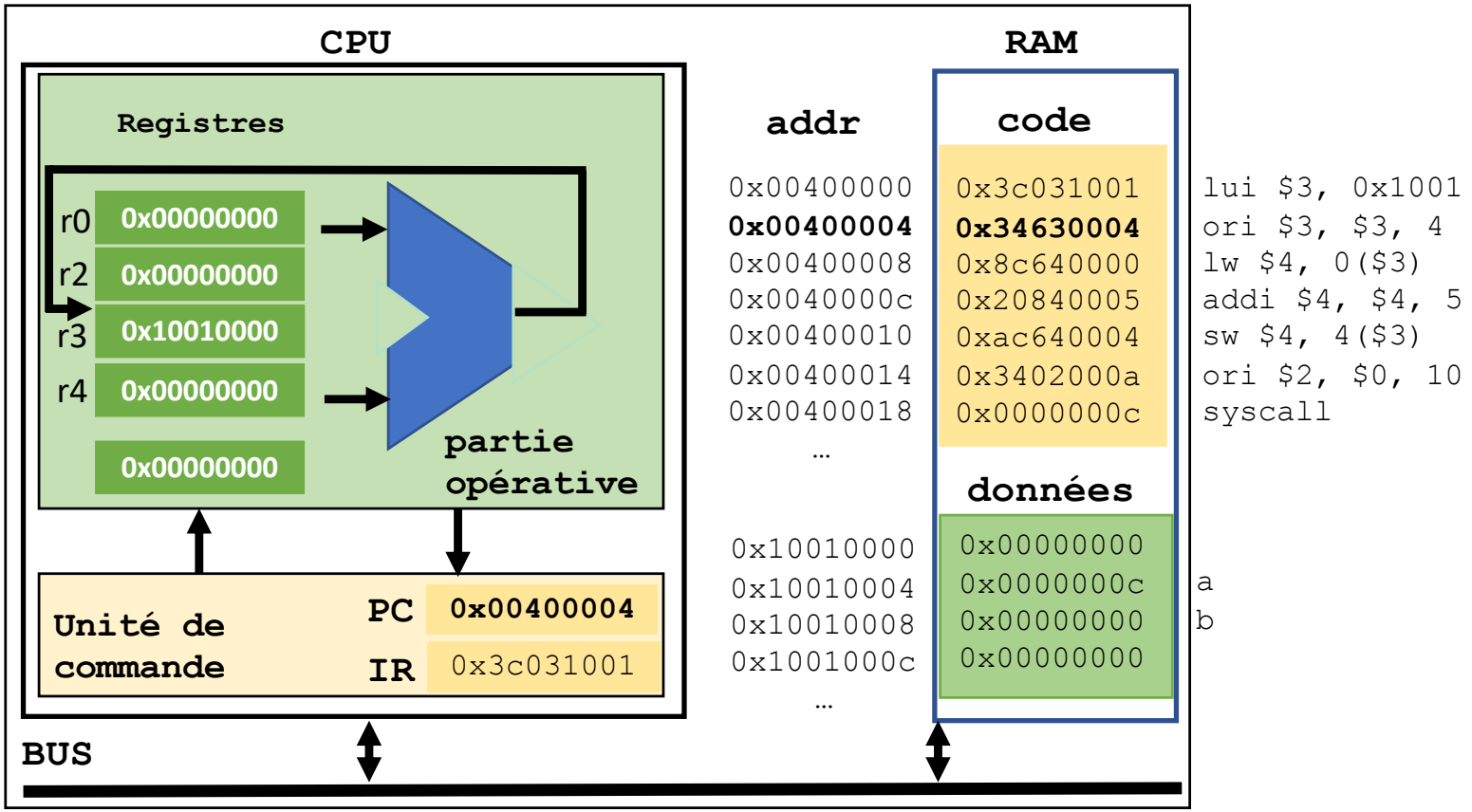
3 – Exécuter l'instruction : affectation partie haute du registre $3 avec immédiat 0x1001

Exécution pas à pas du programme



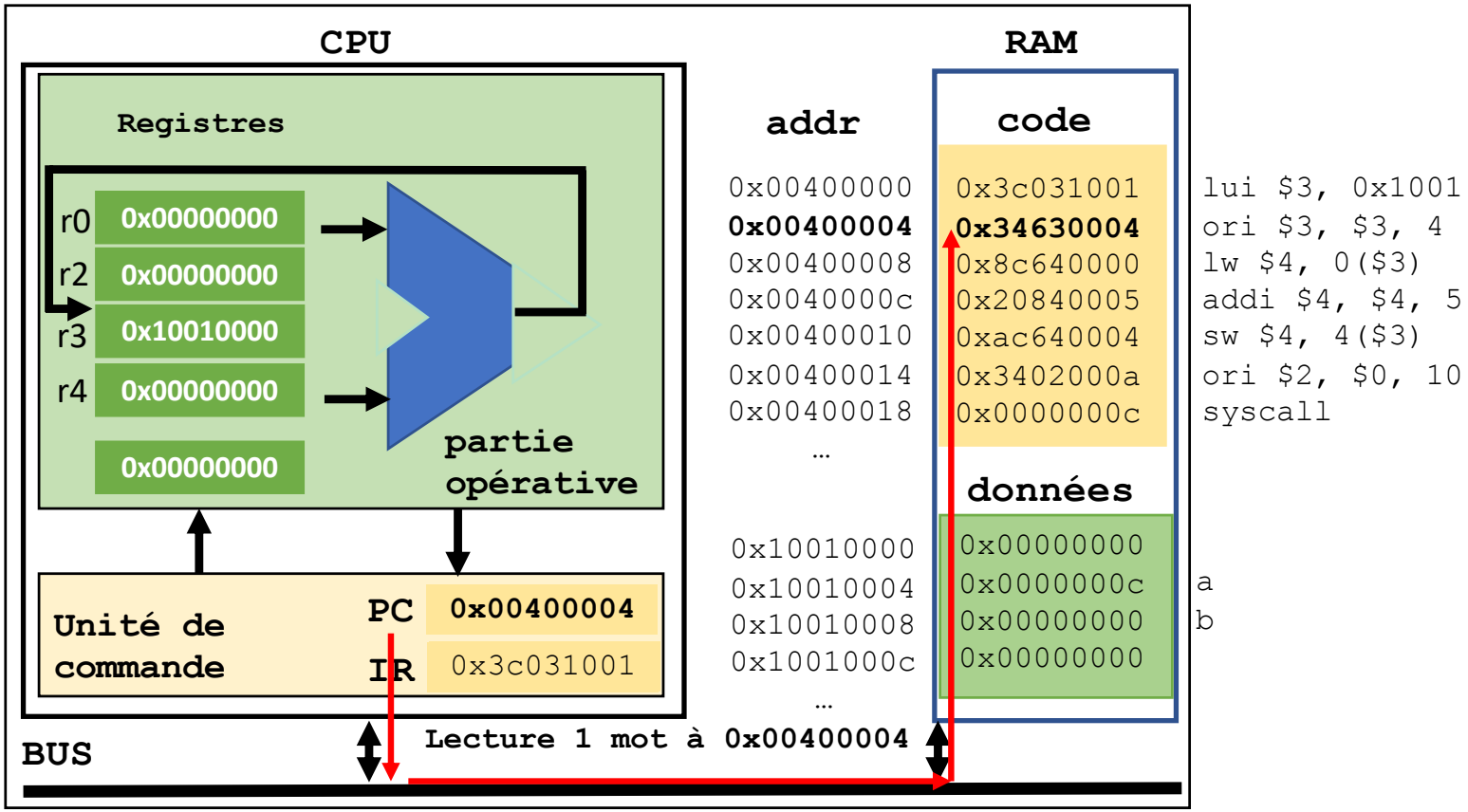
3 – Exécuter l'instruction : affectation partie haute du registre $3 avec immédiat 0x1001

Exécution pas à pas du programme



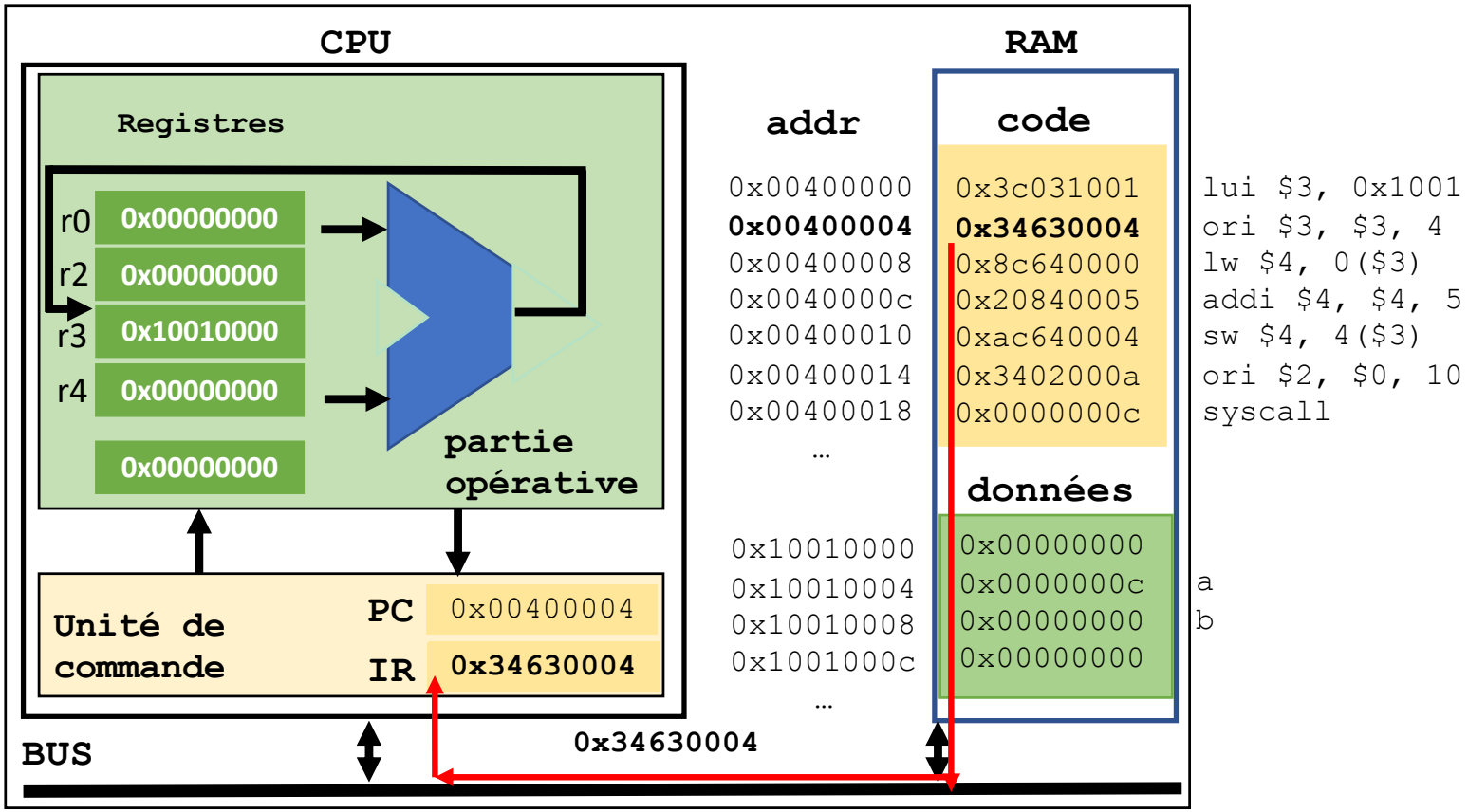
4 – Calculer l'adresse de l'instruction suivante : PC += 4

Exécution pas à pas du programme



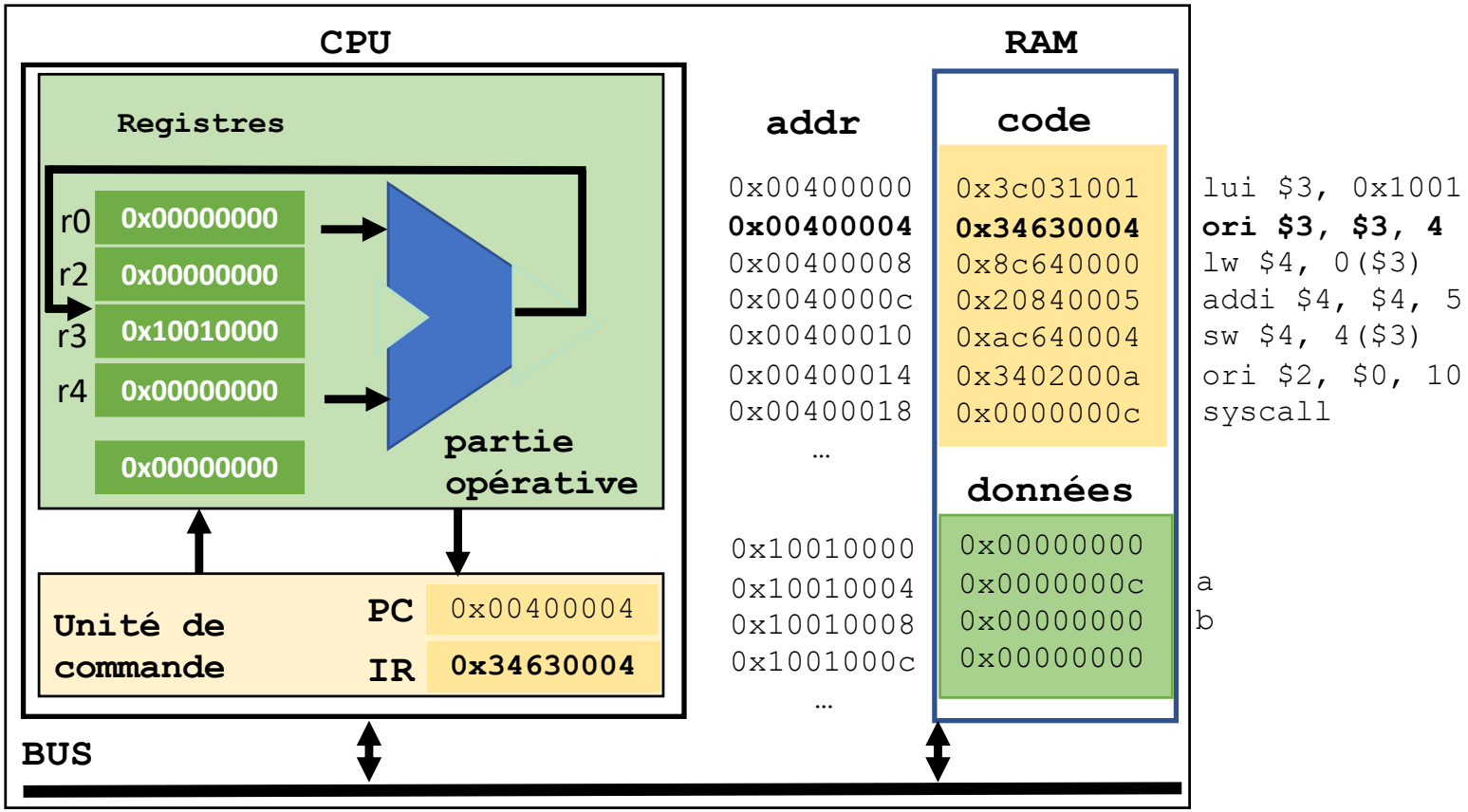
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



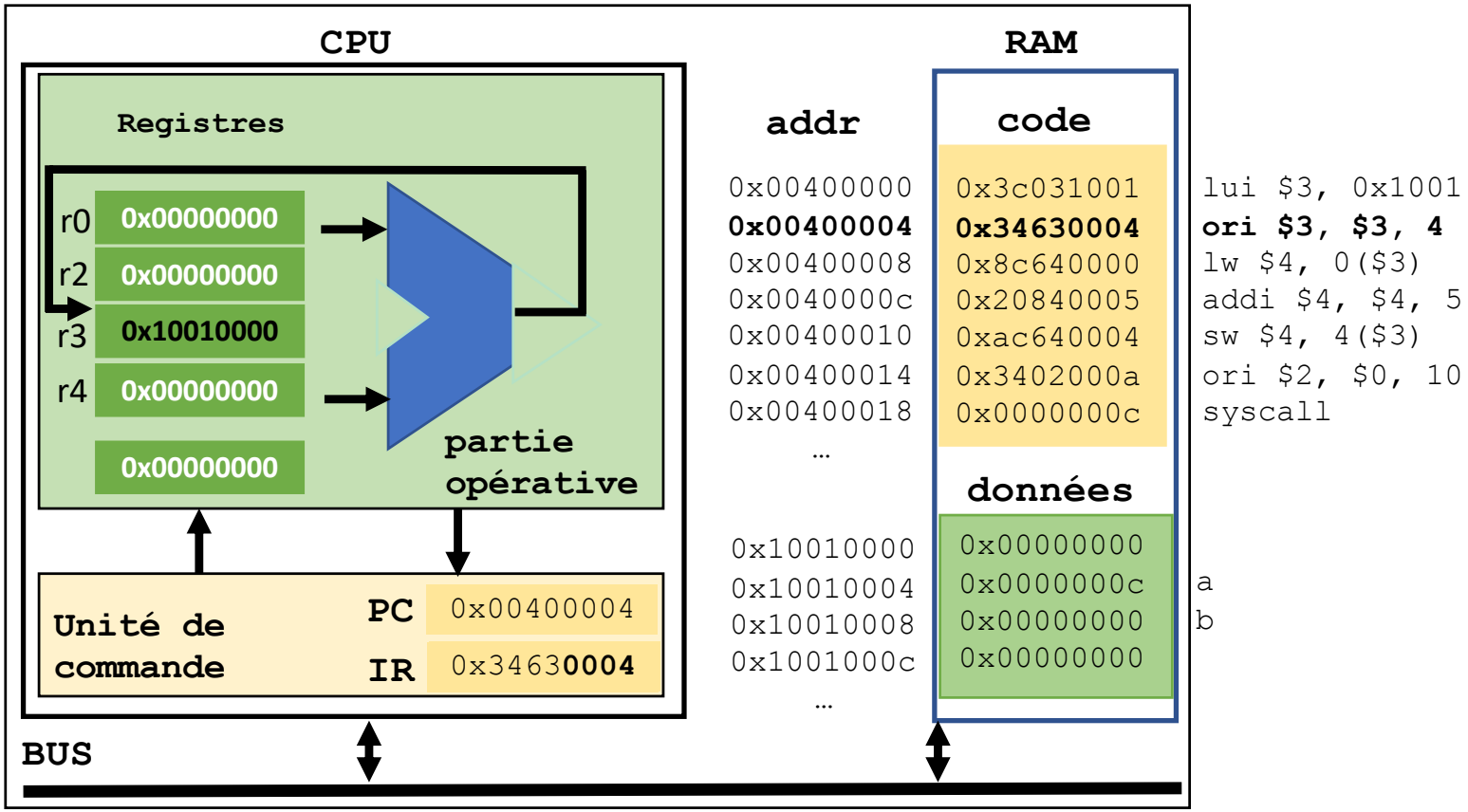
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



2 – Décoder l'instruction : OR entre $3 et un immédiat vers $3

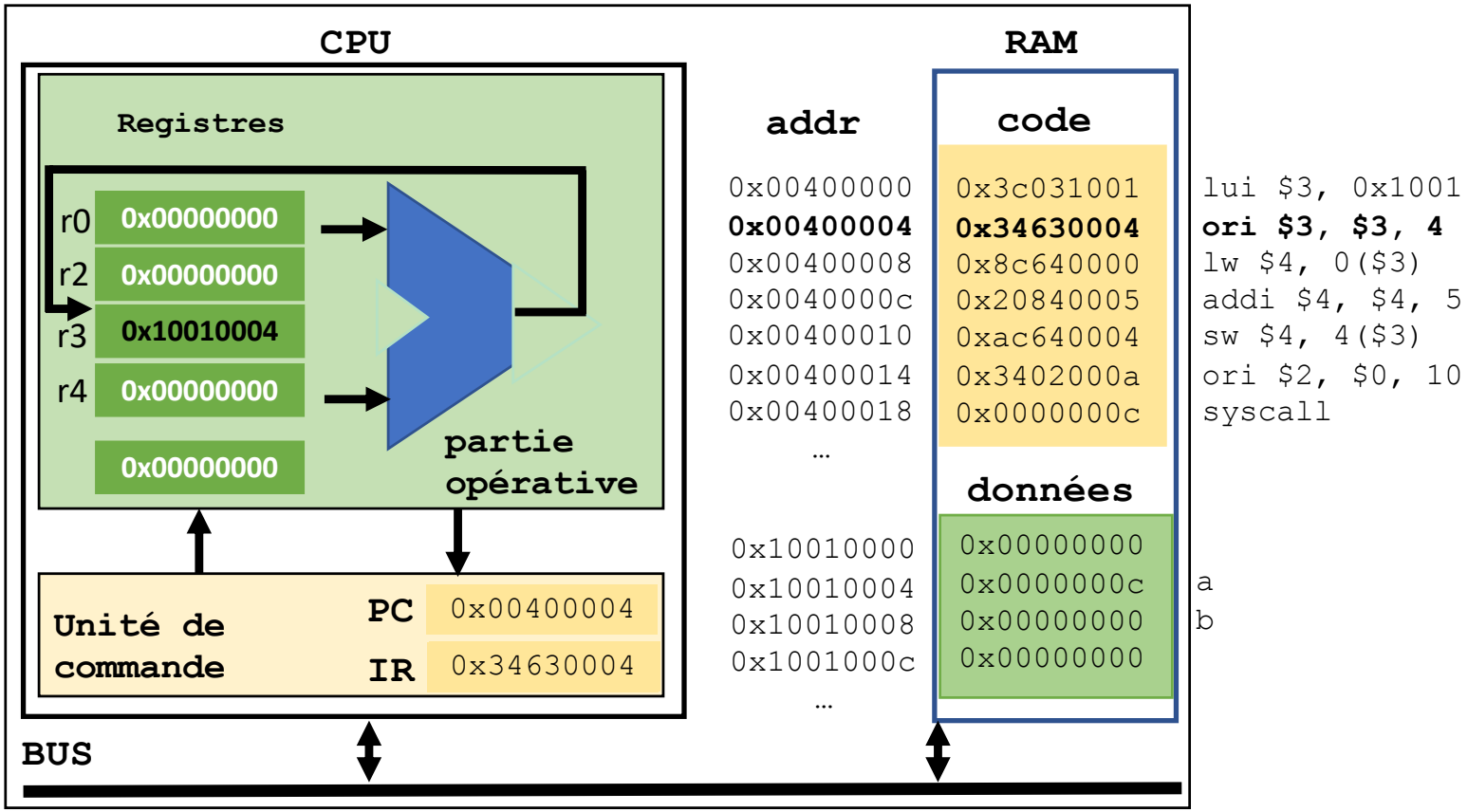
Exécution pas à pas du programme



3 – Exécuter l'instruction : OR entre $3 et un immédiat vers $3

a) OU entre $3 et immediat 0x0004

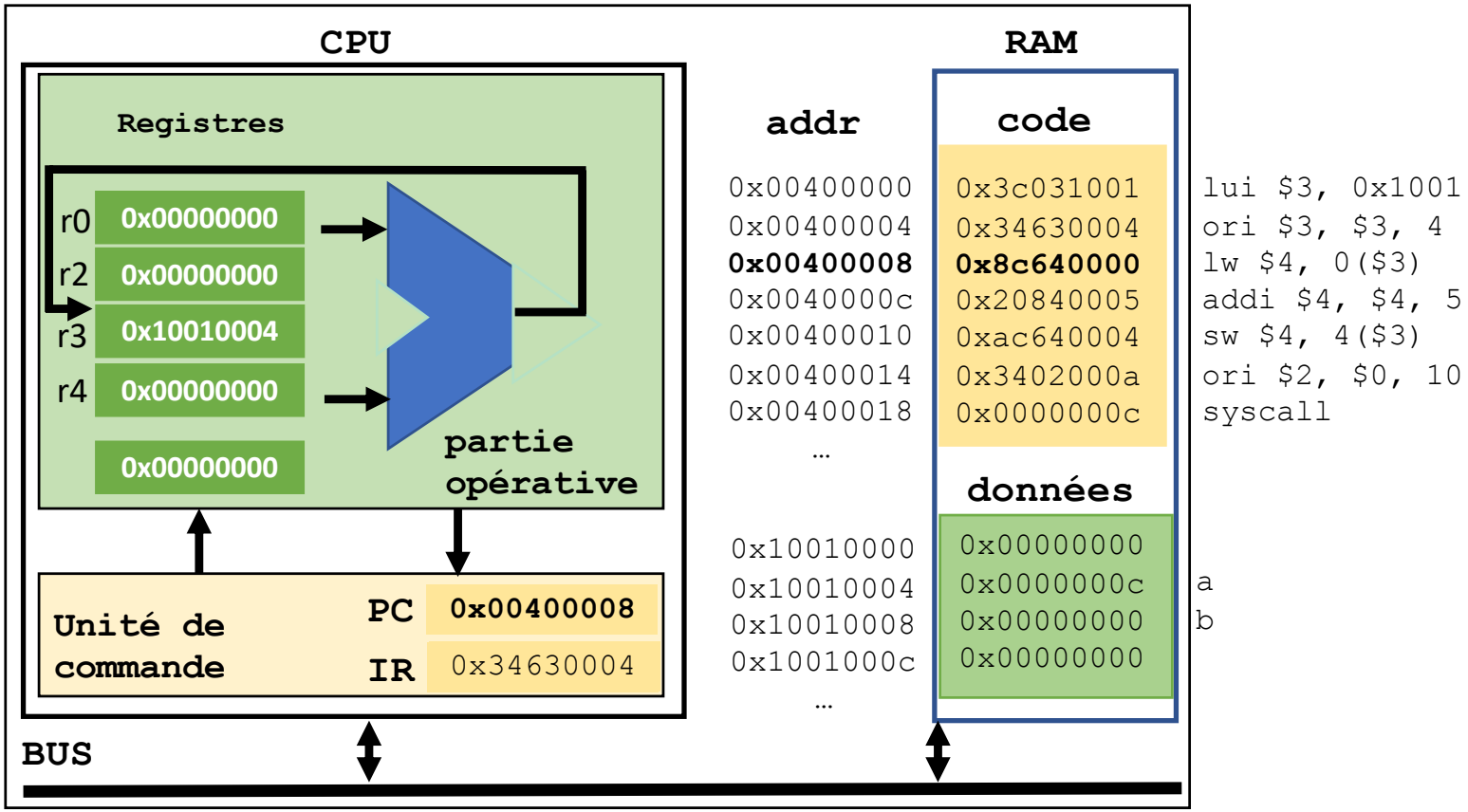
Exécution pas à pas du programme



3 – Exécuter l'instruction : OR entre $3 et un immédiat vers $3

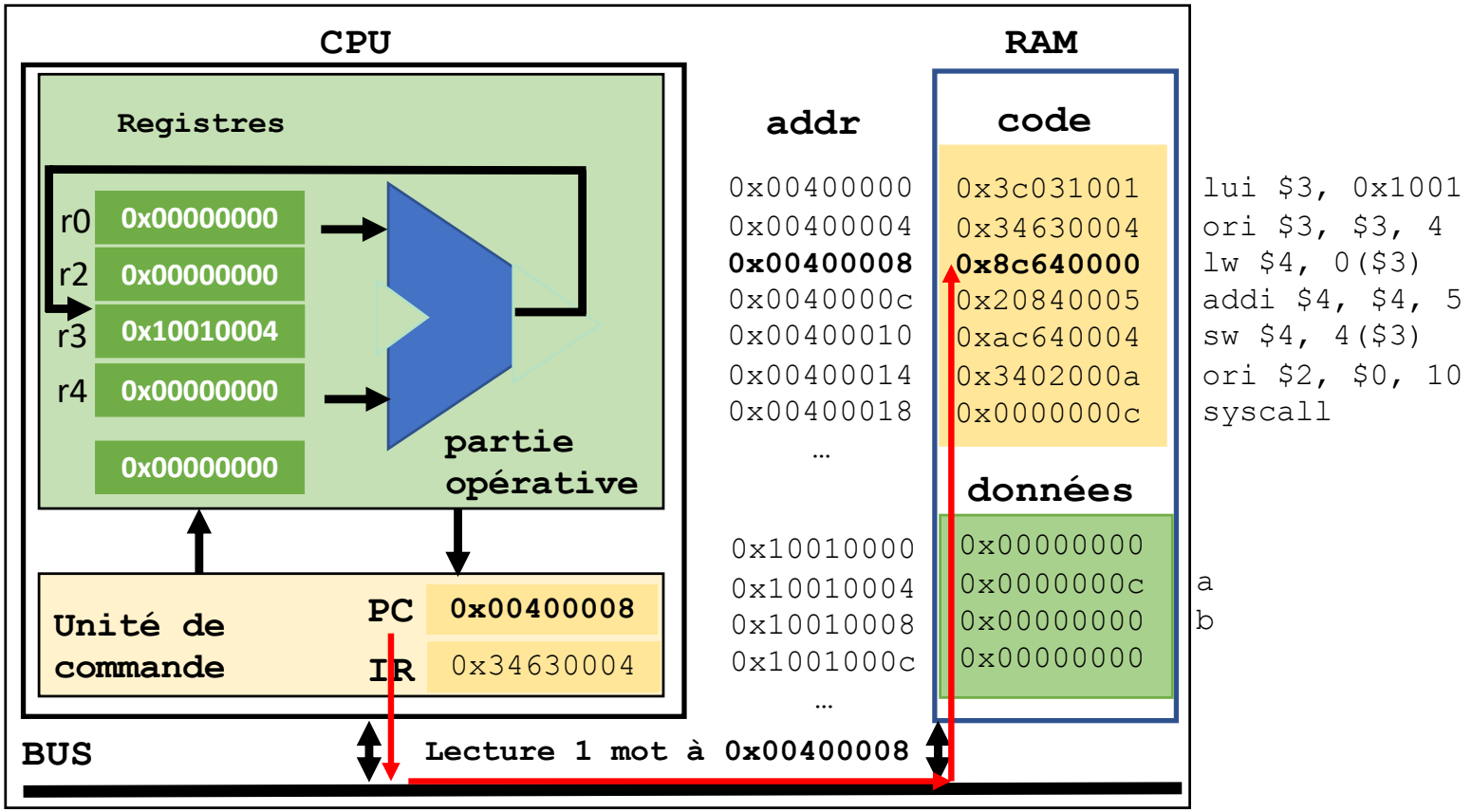
b) résultat dans $3

Exécution pas à pas du programme



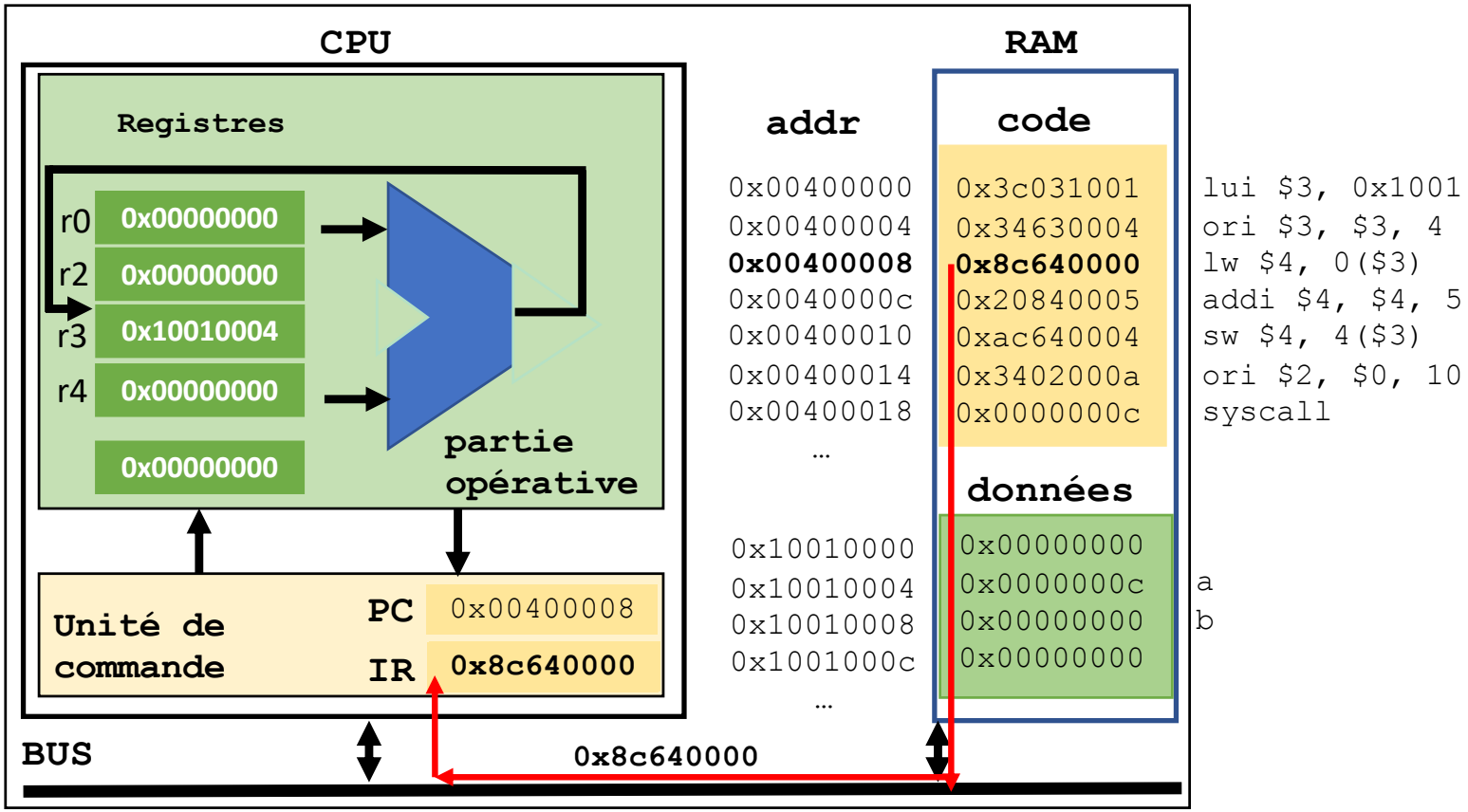
4 – Calculer l'adresse de l'instruction suivante : PC += 4

Exécution pas à pas du programme



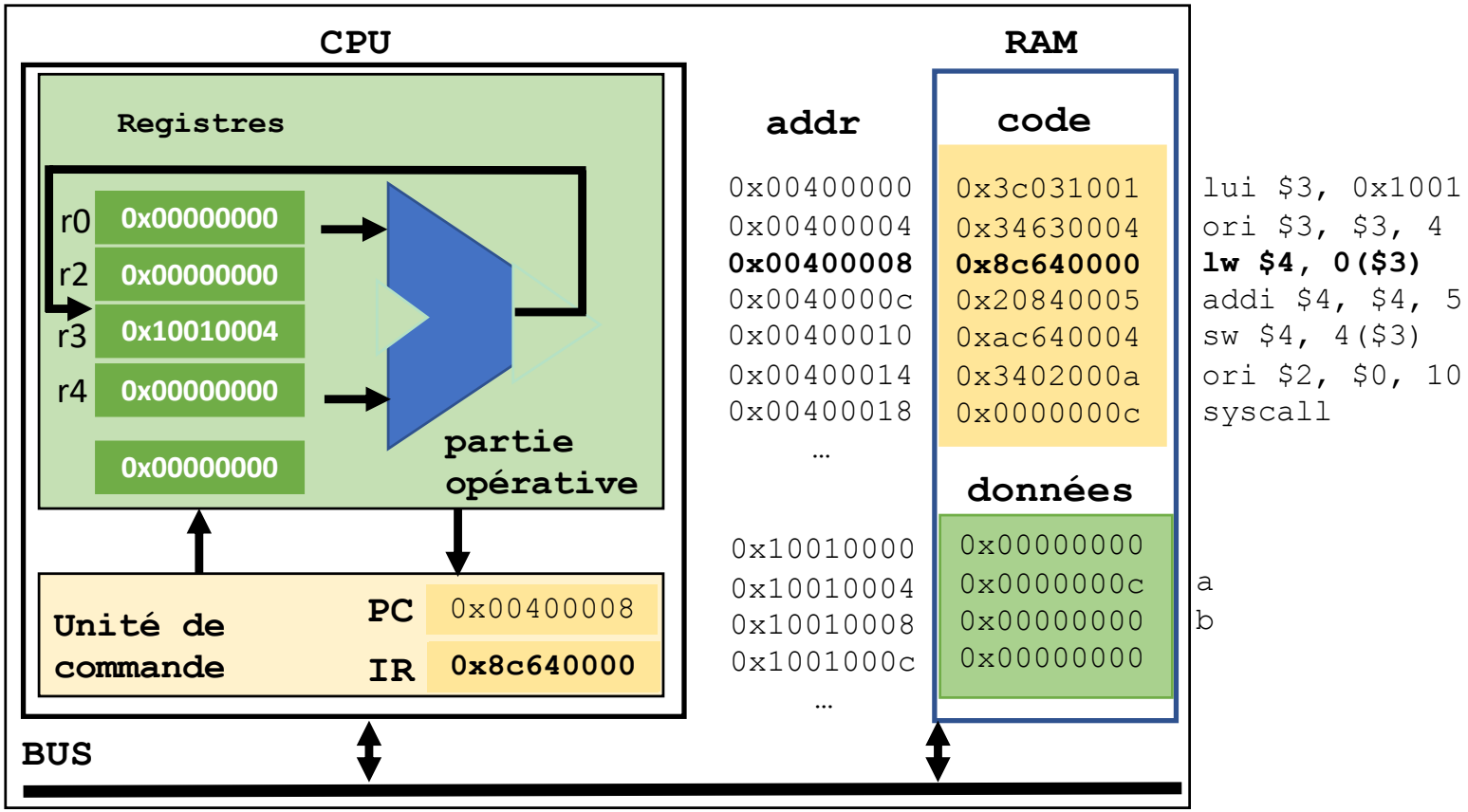
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



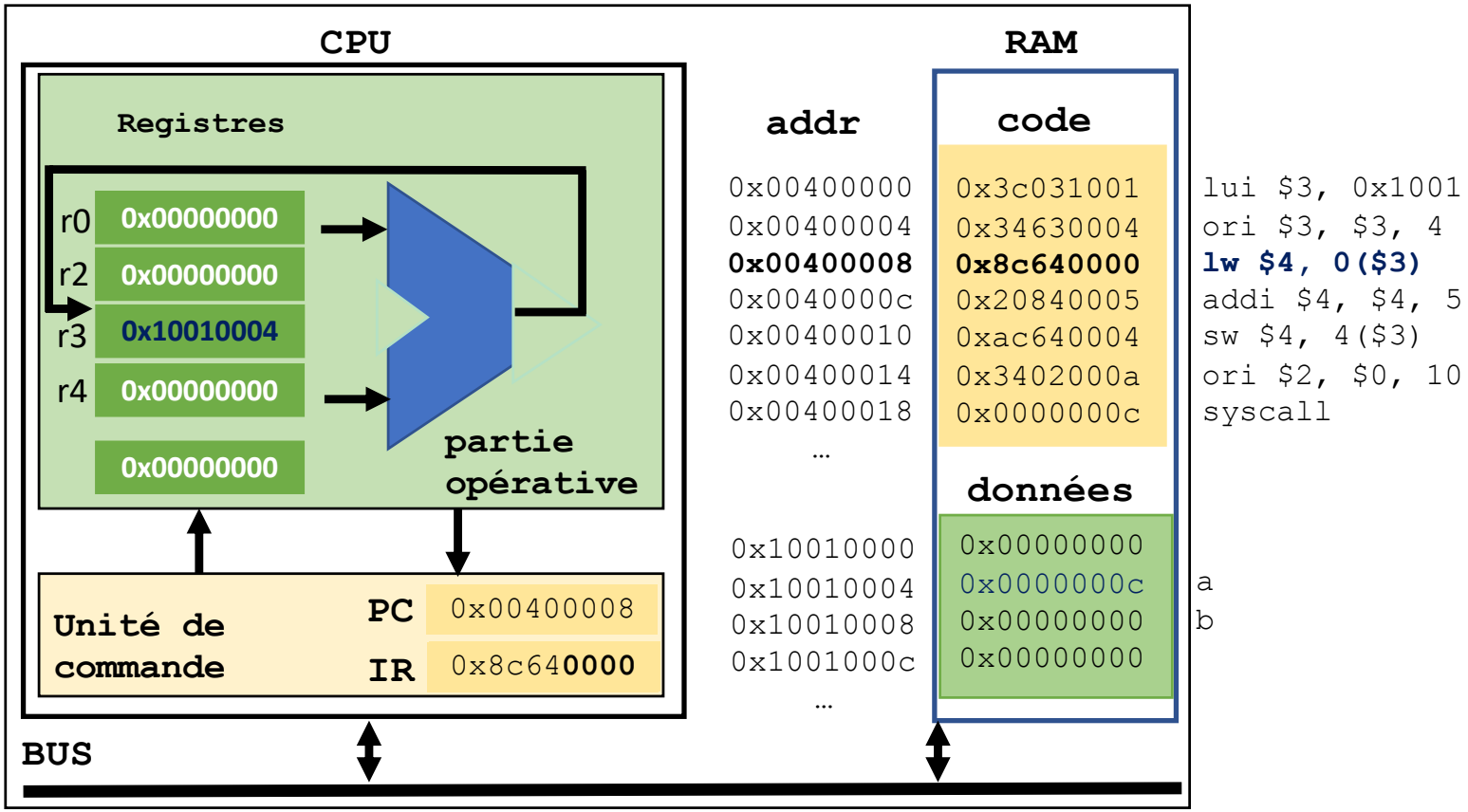
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



2 – Décoder l'instruction : lecture mémoire d'1 mot à l'adresse $3 + 0

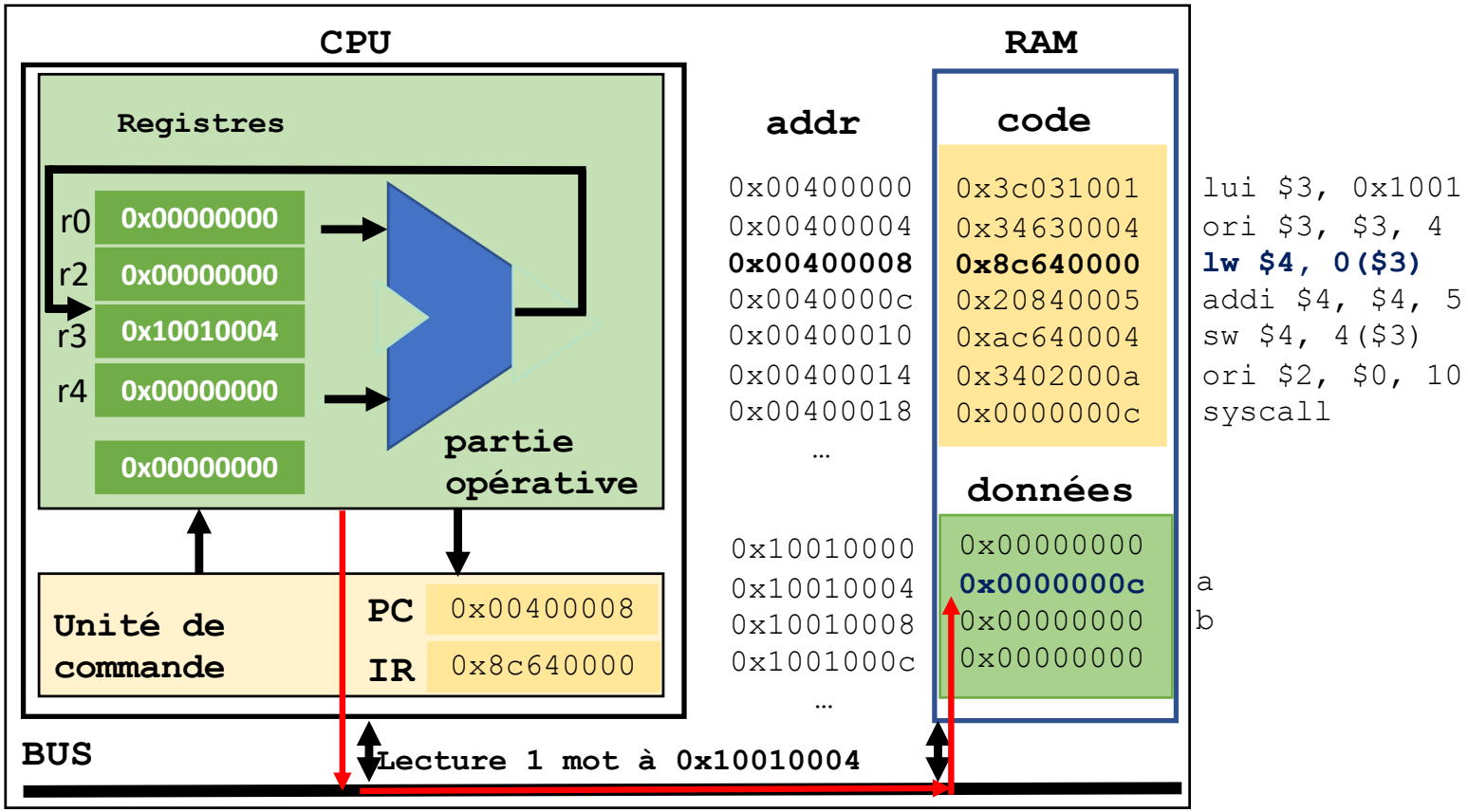
Exécution pas à pas du programme



3 – Exécuter l'instruction : lecture mémoire d'1 mot à l'adresse $3 + 0

a) calcul de l'adresse ($3 + 0)

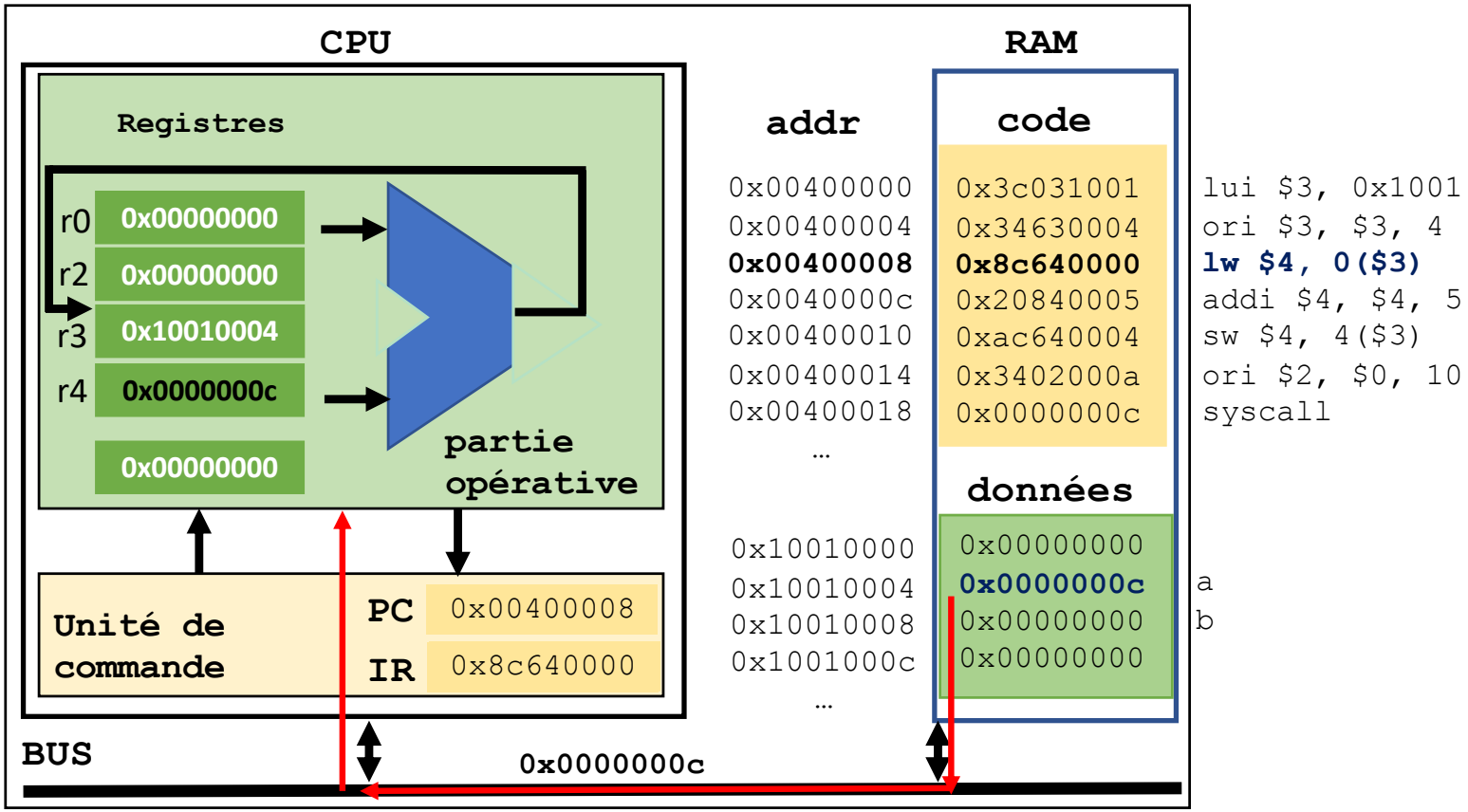
Exécution pas à pas du programme



3 – Exécuter l'instruction : lecture mémoire d'1 mot à l'adresse $3 + 0

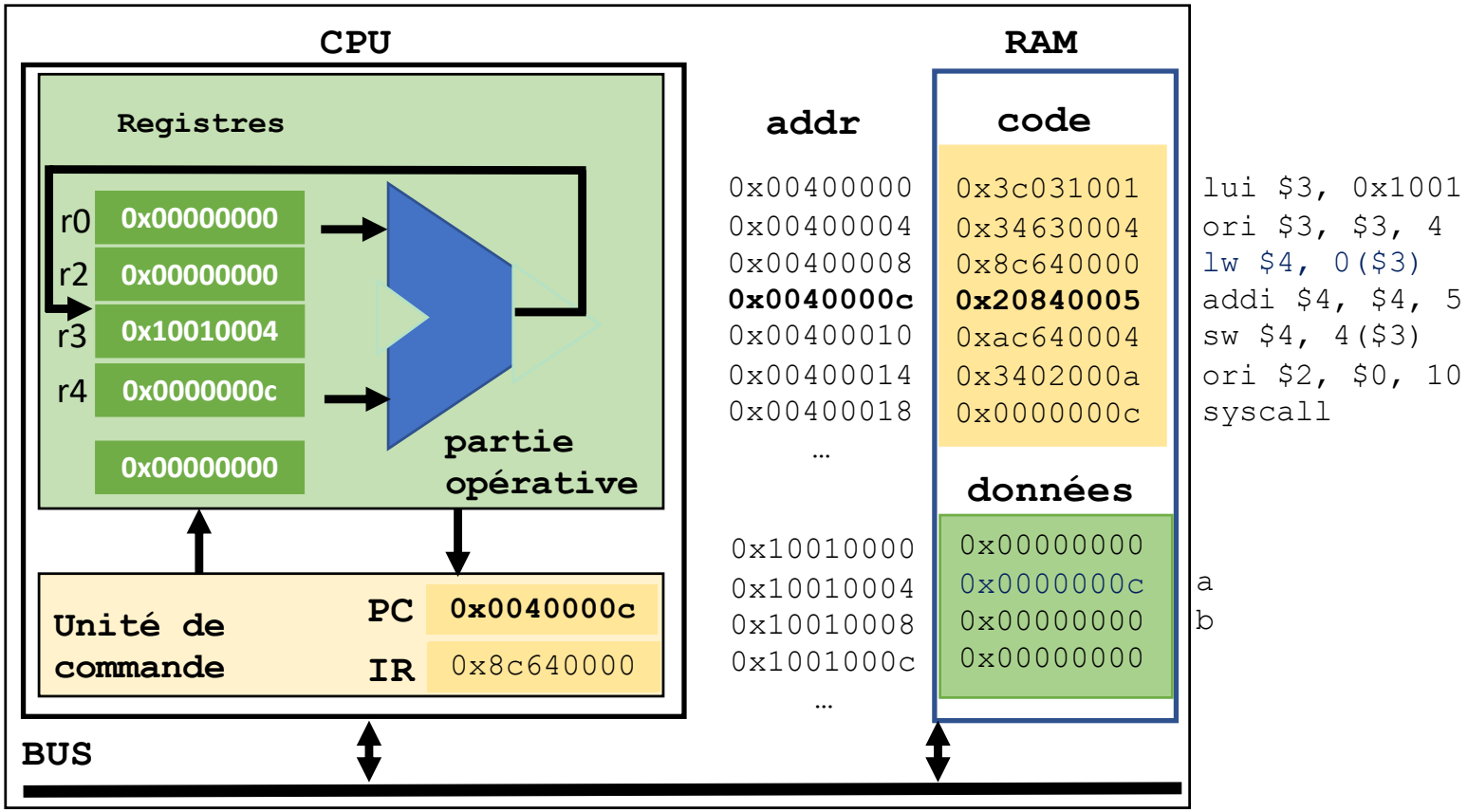
b) transfert mémoire en lecture

Exécution pas à pas du programme



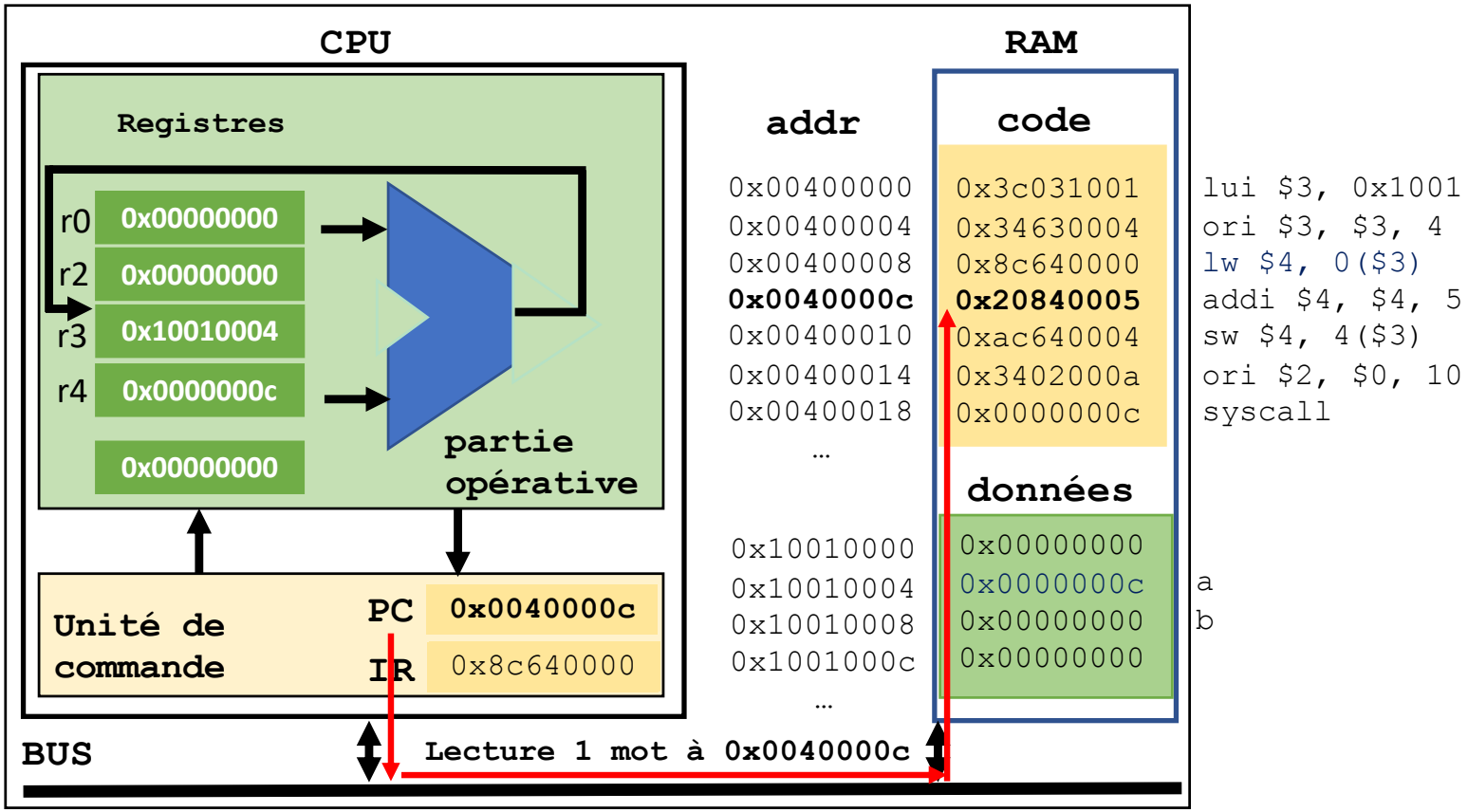
3 – Exécuter l'instruction : lecture mémoire d'1 mot à l'adresse $3 + 0
c) résultat dans $4

Exécution pas à pas du programme



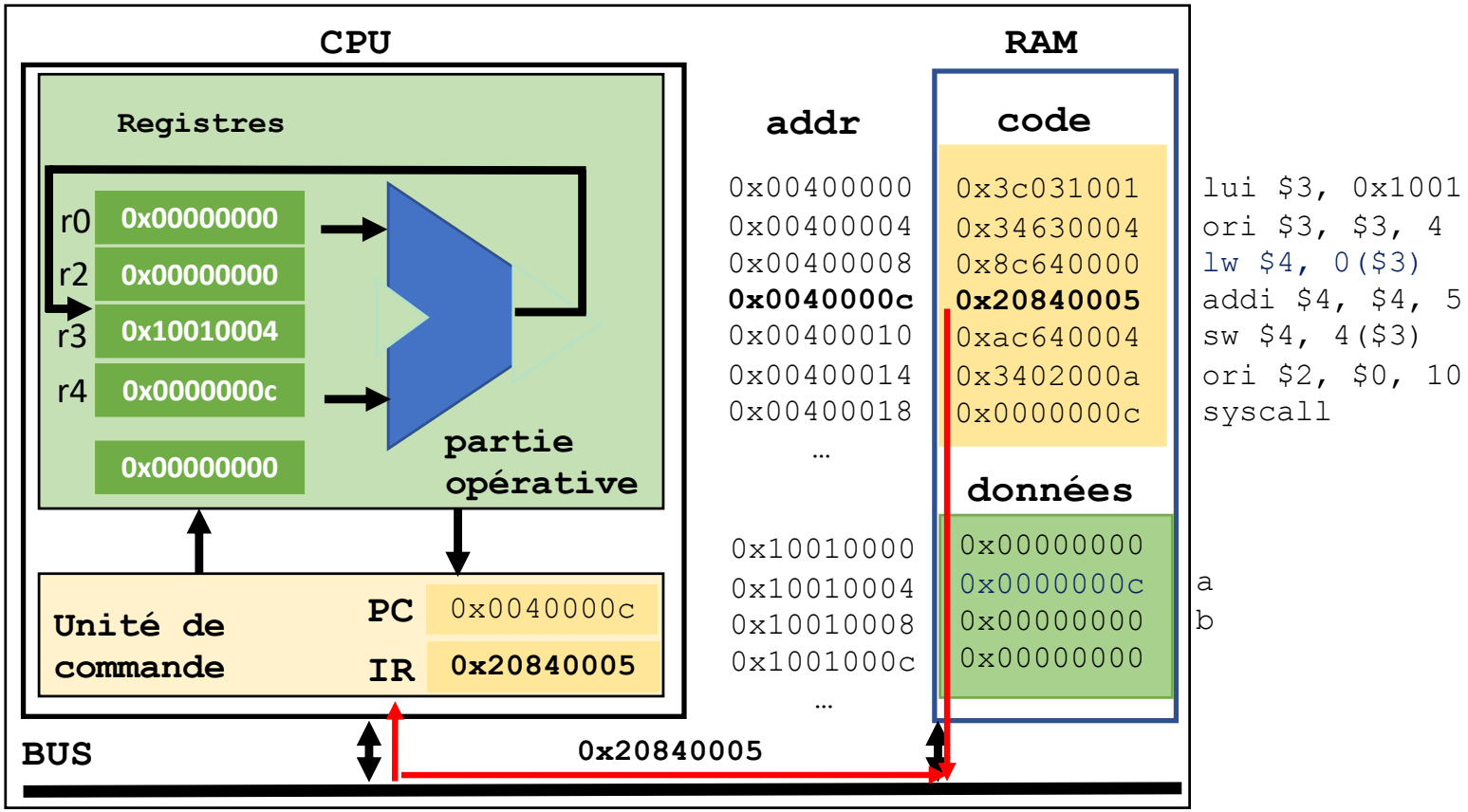
4 – Calculer l'adresse de l'instruction suivante : PC += 4

Exécution pas à pas du programme



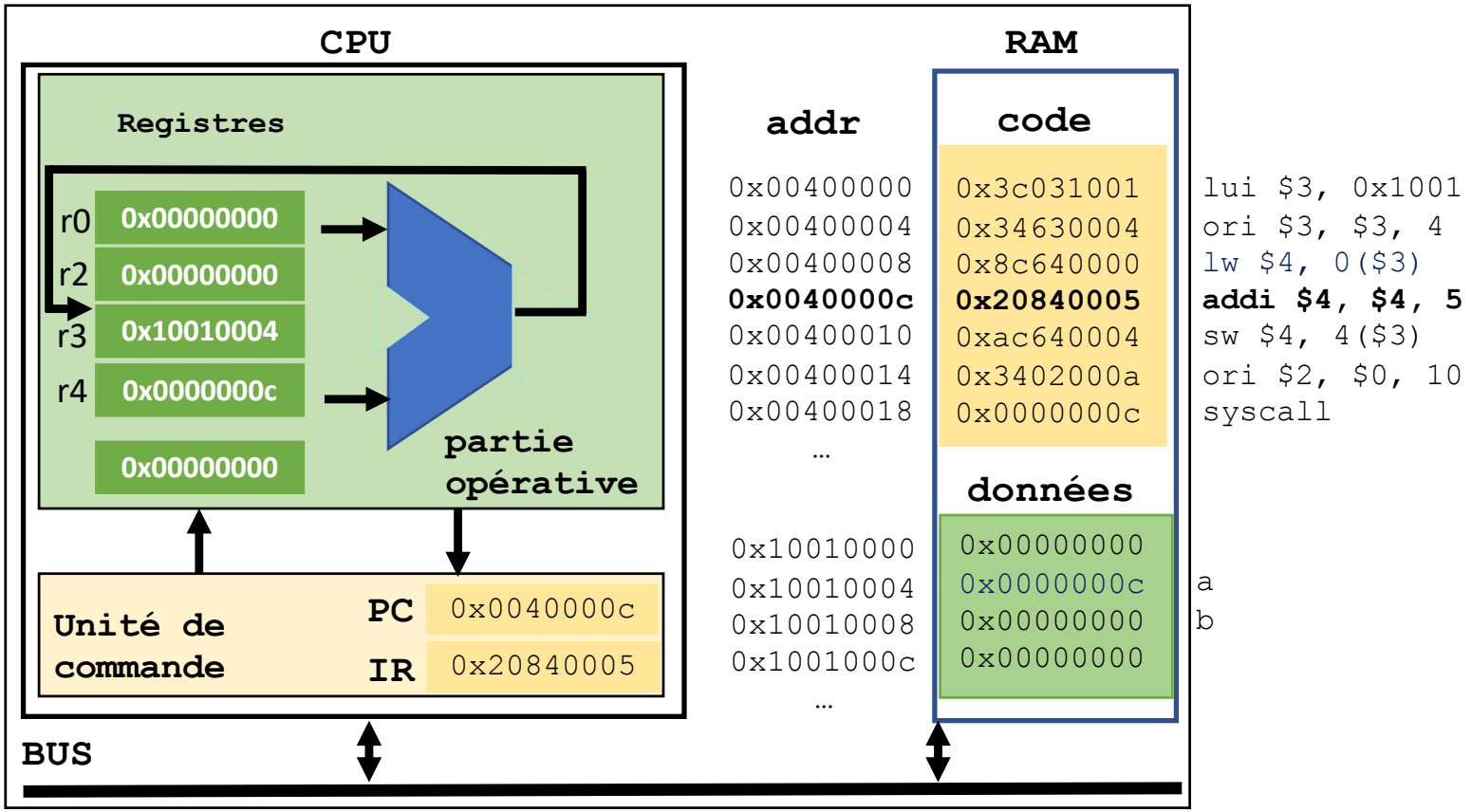
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



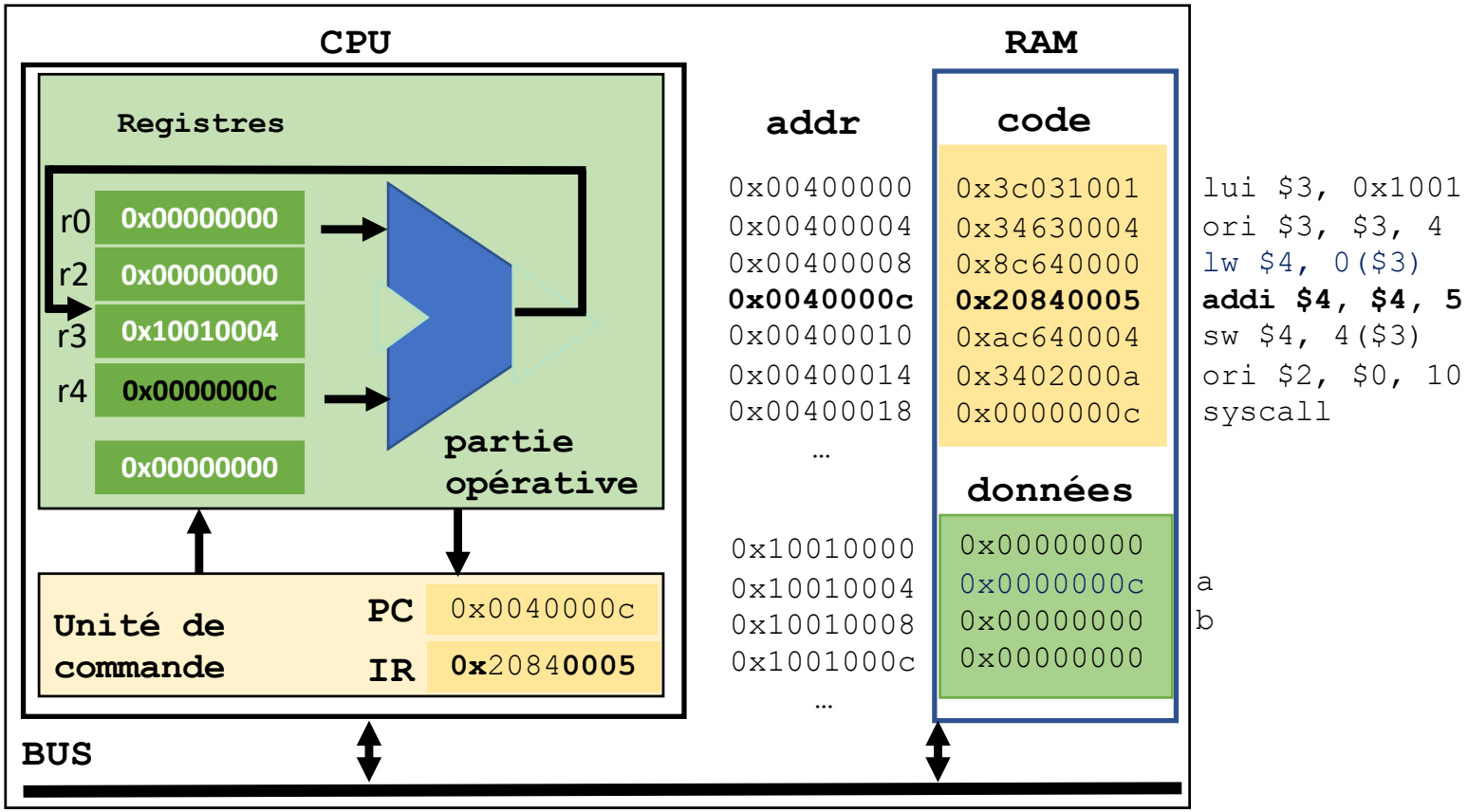
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



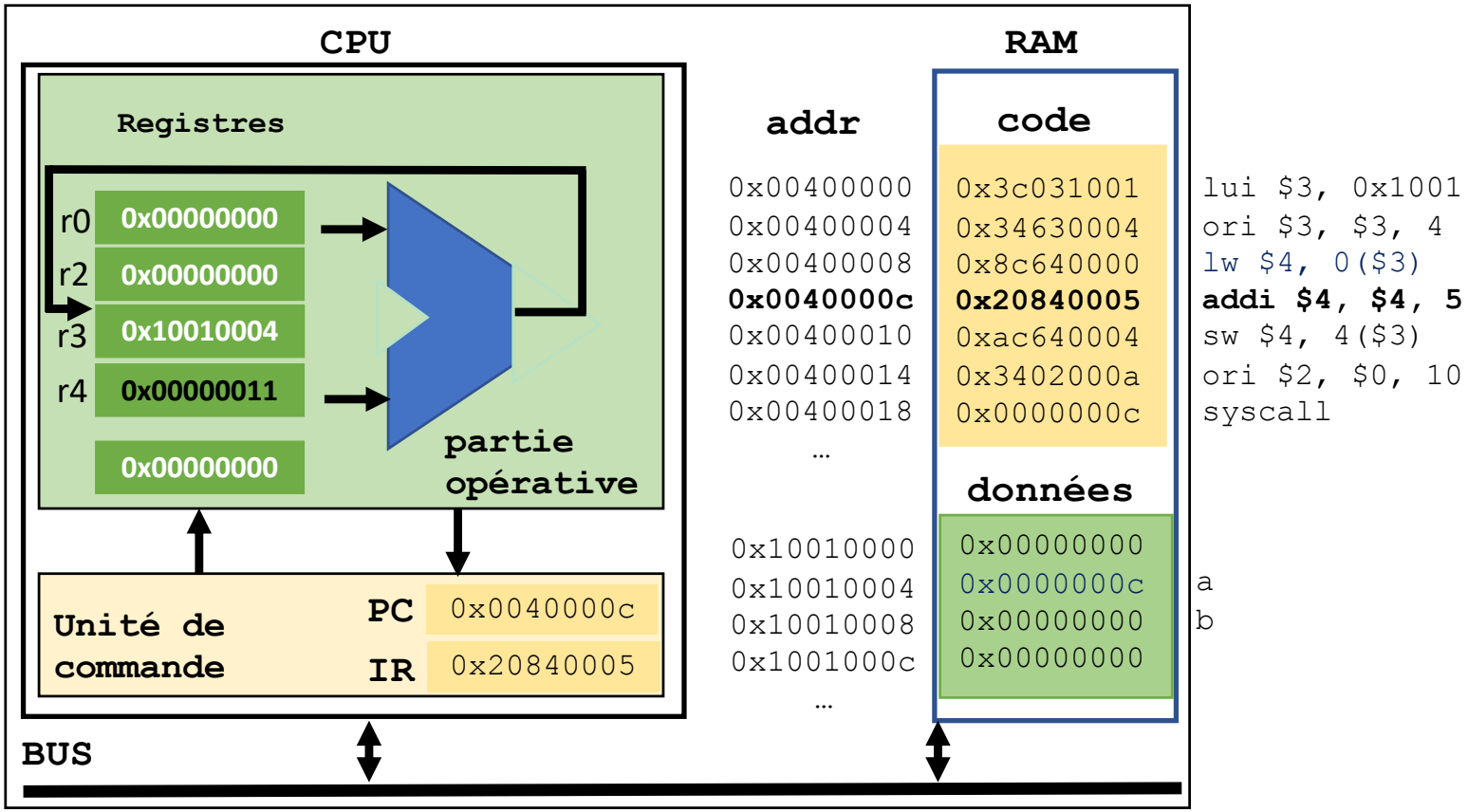
2 – Décoder l'instruction : addition de $4 avec un immédiat (0x0005)

Exécution pas à pas du programme



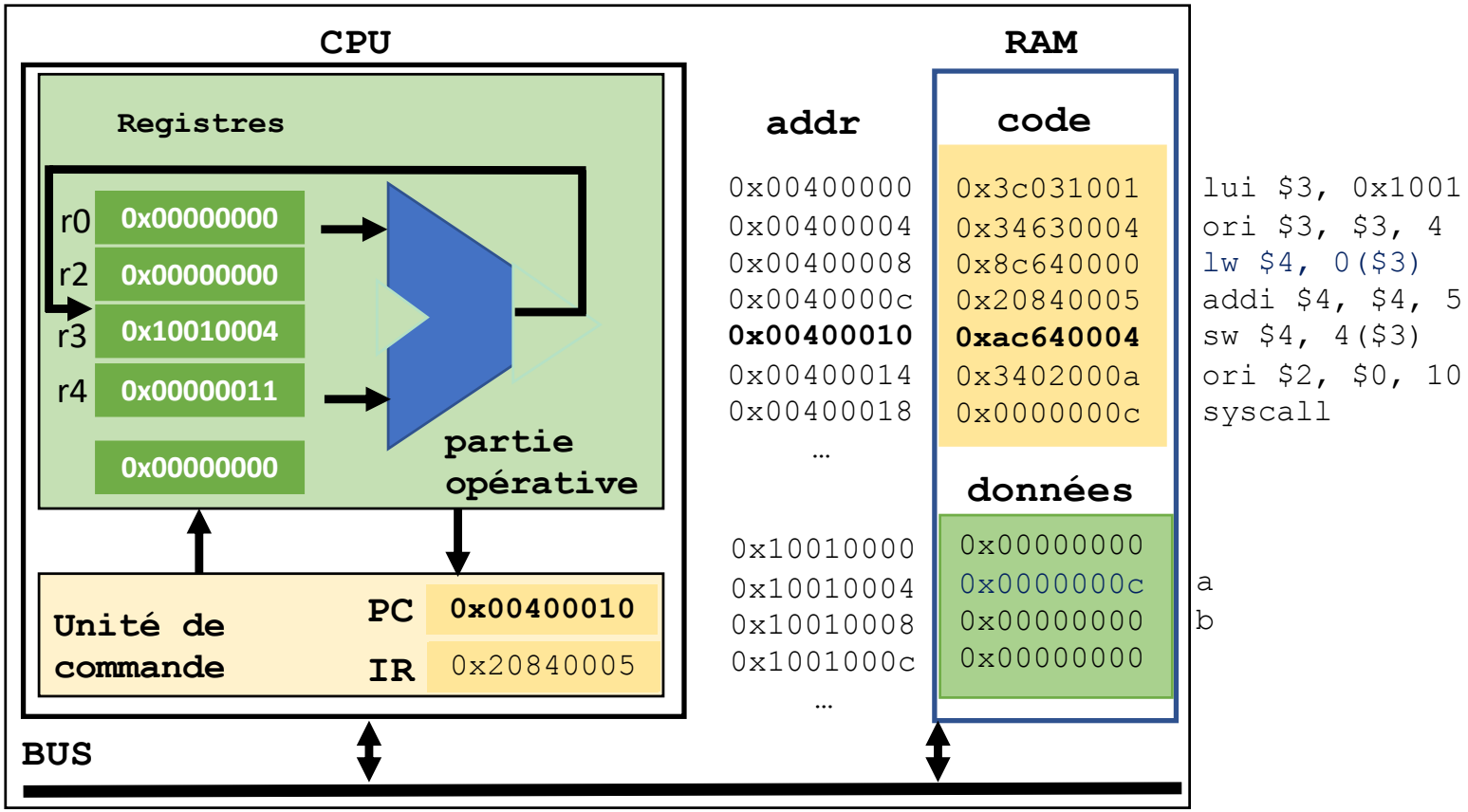
3 – Executer l'instruction : addition de $4 avec un immédiat (0x0005)
a) faire l'addition

Exécution pas à pas du programme



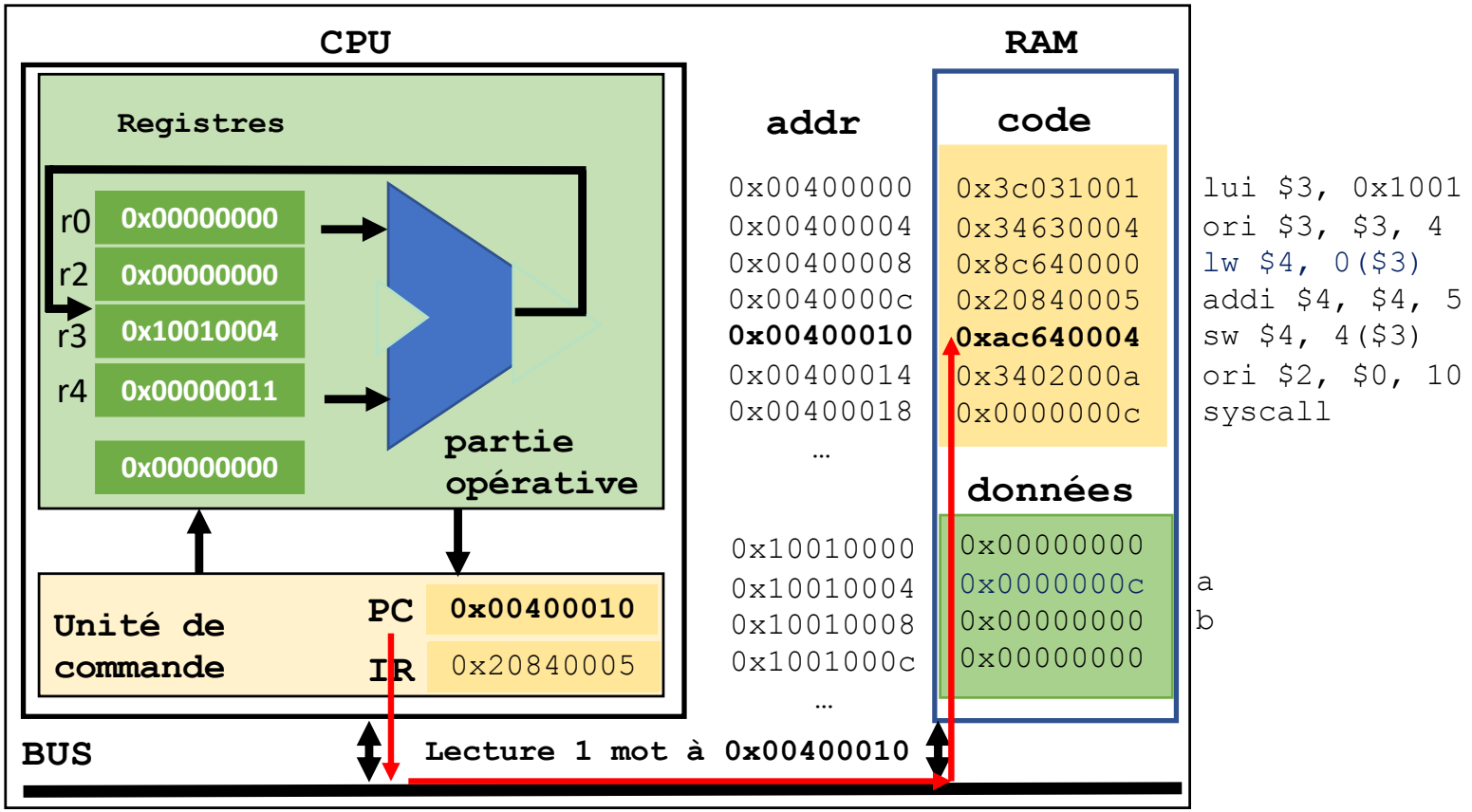
3 – Executer l'instruction : addition de $4 avec un immédiat (0x0005)
b) résultat dans $4

Exécution pas à pas du programme



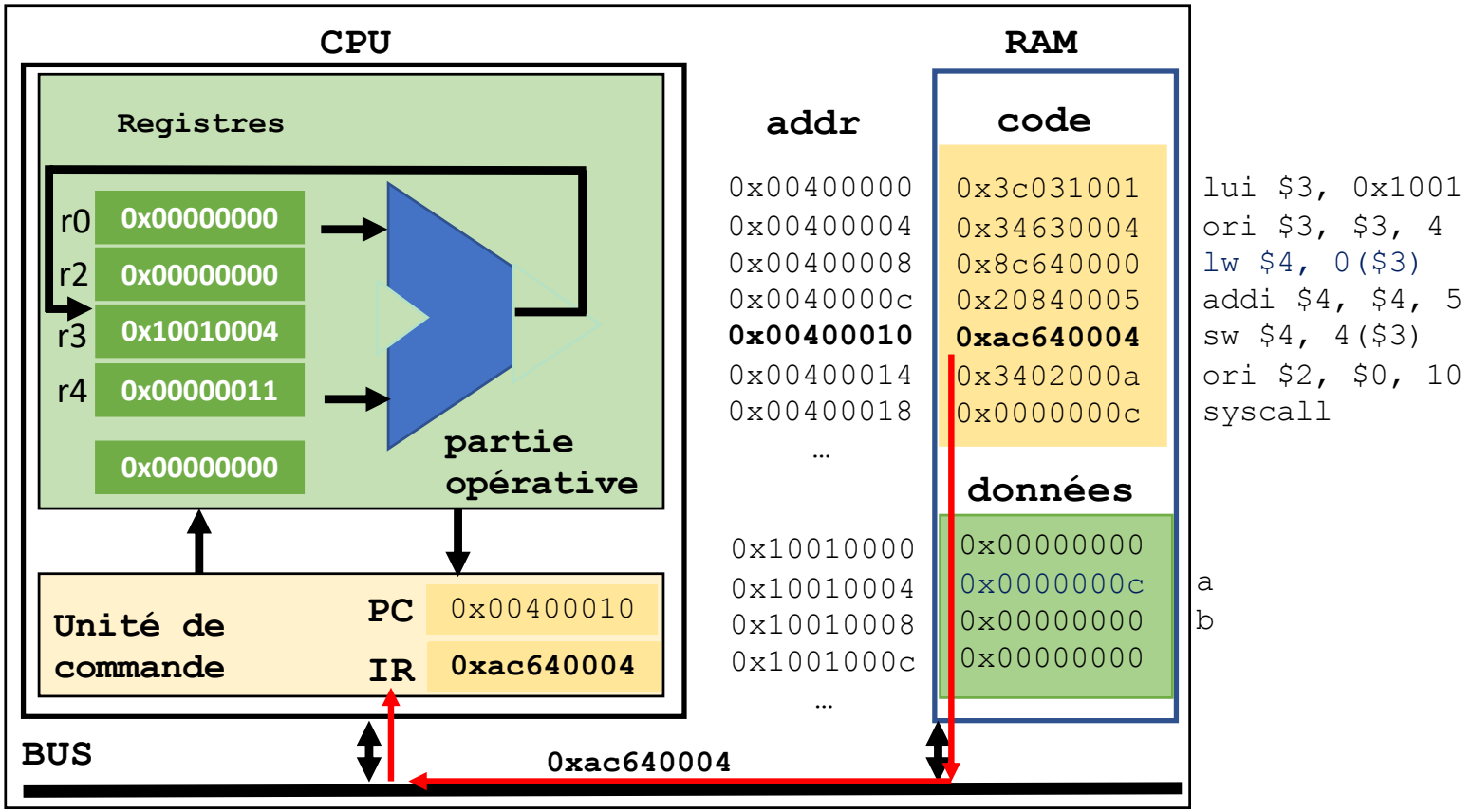
4 – Calculer l'adresse de l'instruction suivante : PC += 4

Exécution pas à pas du programme



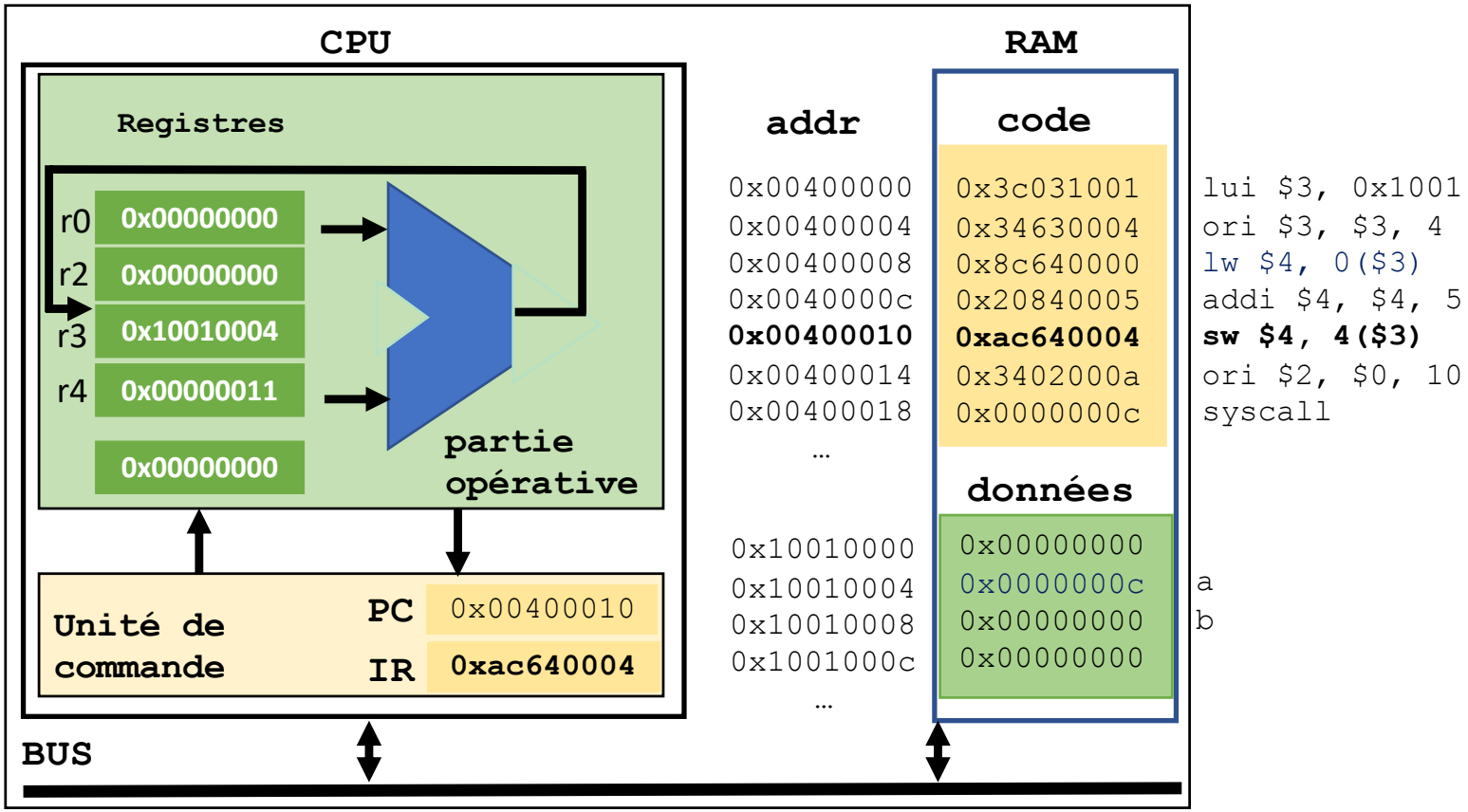
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



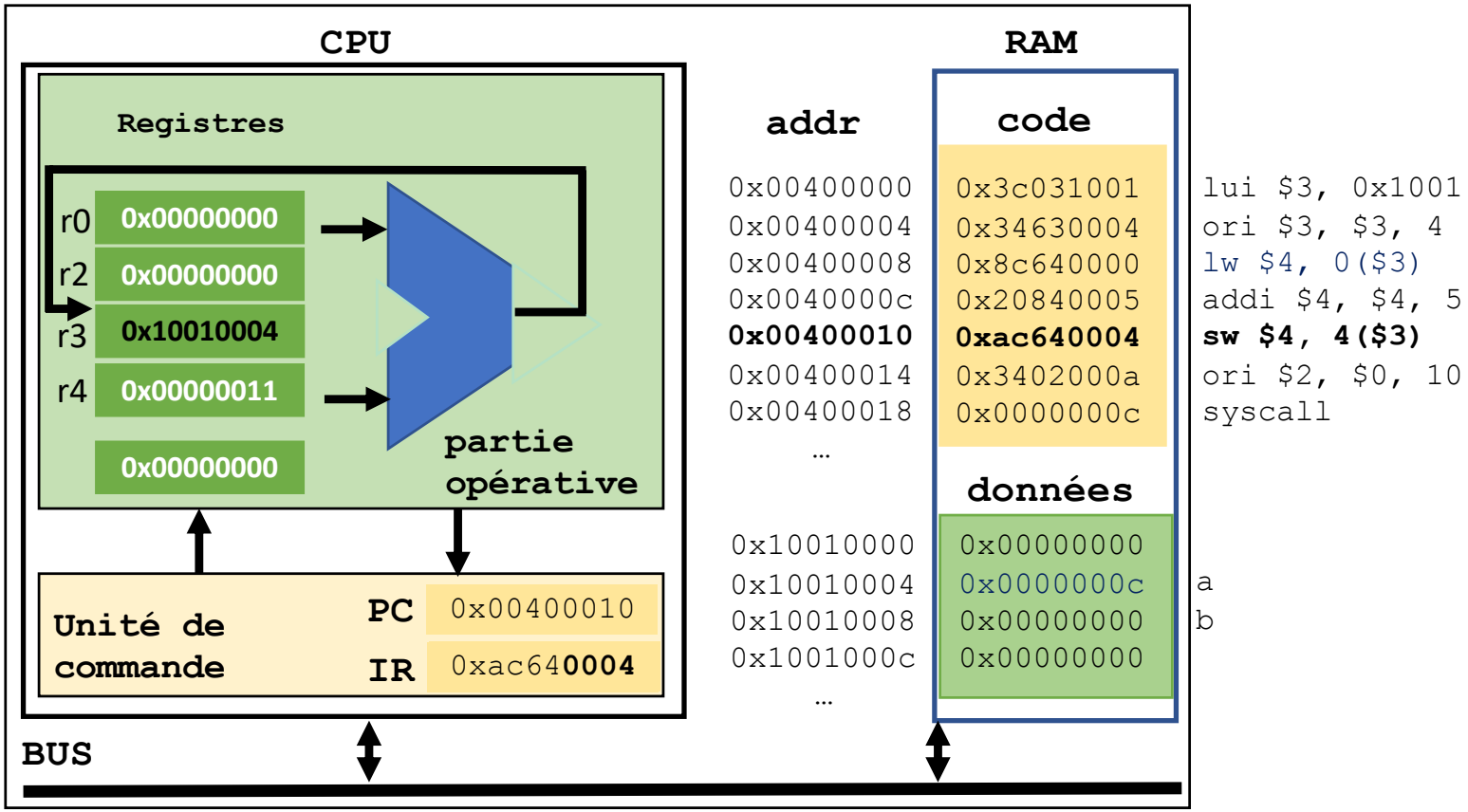
1 - Lire l'instruction à exécuter (adresse contenue PC)

Exécution pas à pas du programme



2 – Décoder l'instruction : écriture mémoire du mot contenu dans $4 à l'adresse 4 + $3

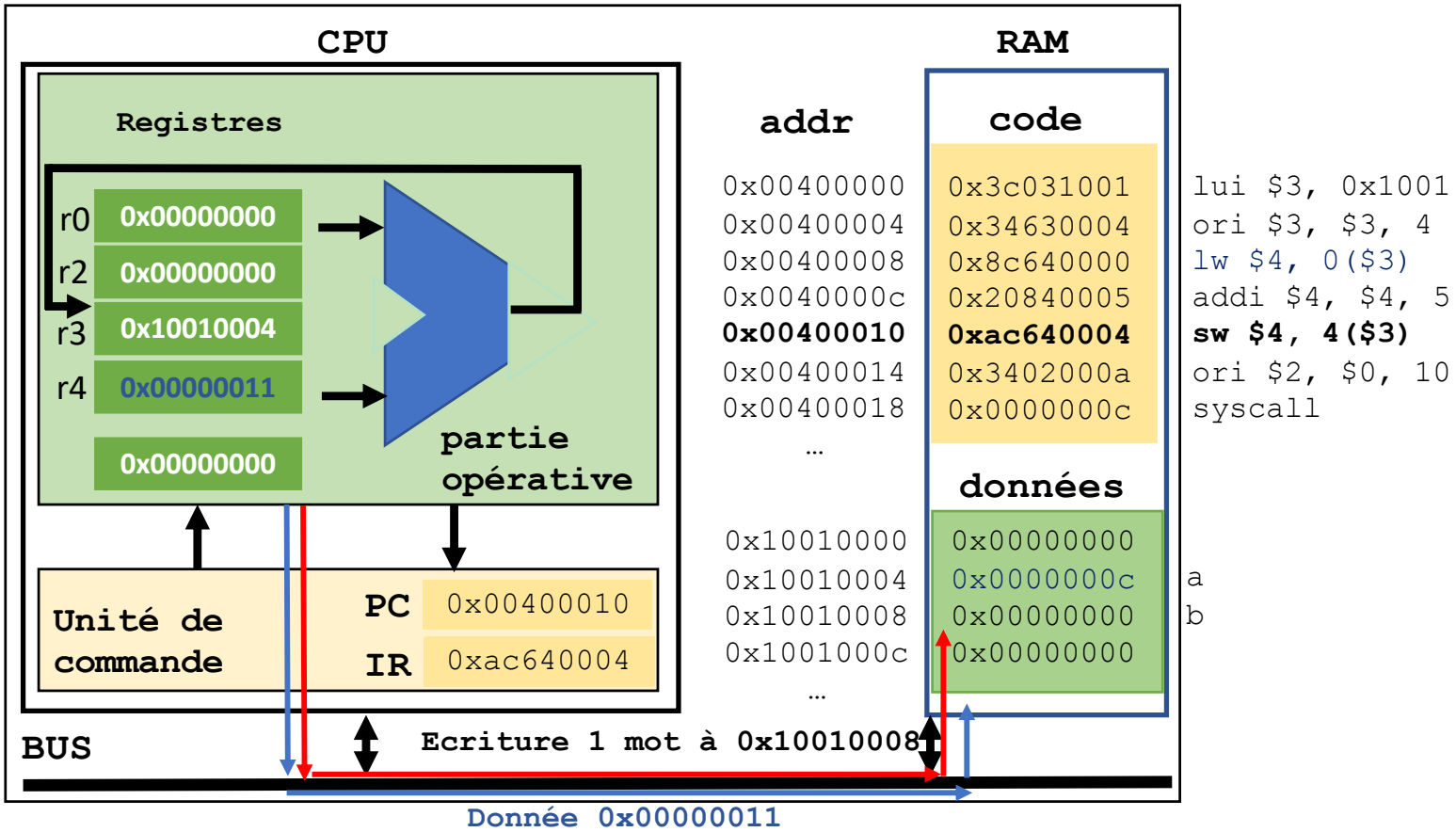
Exécution pas à pas du programme



3 – Exécuter l'instruction : écriture mémoire du mot contenu dans $4 à l'adresse 4 + $3

a) calcul d'adresse 4 + $3

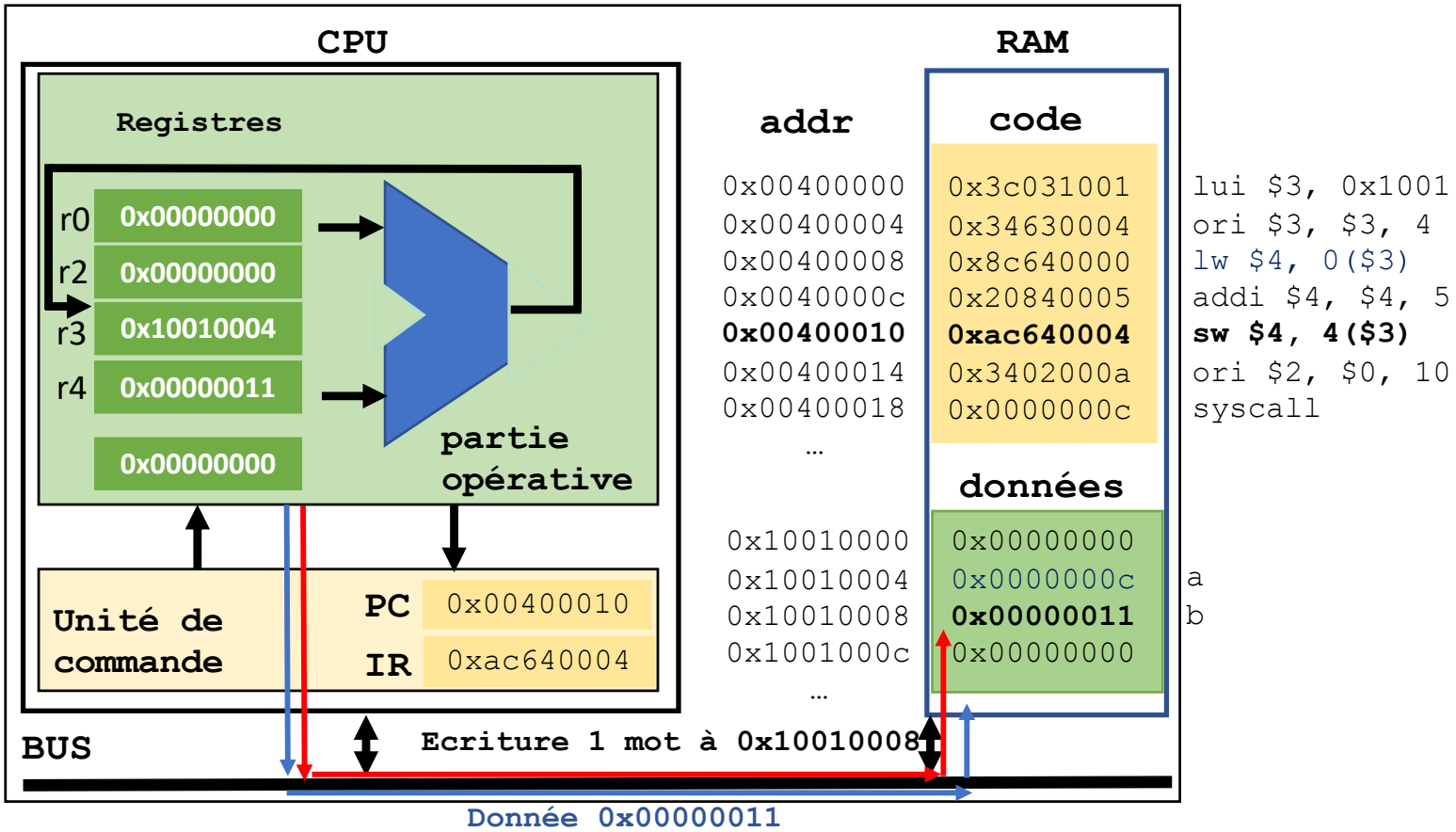
Exécution pas à pas du programme



3 – Exécuter l'instruction : écriture mémoire du mot contenu dans $4 à l'adresse 4 + $3

a) transfert mémoire en écriture

Exécution pas à pas du programme



3 – Exécuter l'instruction : écriture mémoire du mot contenu dans $4 à l'adresse 4 + $3

a) transfert mémoire en écriture

À retenir

Mémoire versus registre

- 1 accès en lecture par instruction : pour lire l'instruction en mémoire et la ranger dans le registre IR
- Accès en lecture pour les données : instruction de type *load*)
  $\Rightarrow$ permet de ranger dans un registre une valeur (de données) présente en mémoire
- Les instructions qui utilisent l'ALU NE changent PAS le contenu de la mémoire. Elles modifient uniquement le contenu de registre(s).
- Seule une écriture mémoire, soit une instruction de type *store*, peut changer le contenu de la mémoire (données)

Exemple de programmes

Exemple : Afficher un entier stocké en mémoire

- Allouer et inititaliser un entier dans la section de données, avec la bonne directive
- Affichage d'un entier : appel système numéro 1 + entier à afficher dans `$4`
  L'entier à afficher se trouver en mémoire ⇒ transfert mémoire en lecture pour récupérer sa valeur
- Terminaison du programme : appel système numéro 10

```
.data
n:   .word 0x18   # allocation d'un entier initialisé à24

.text
   lui $3, 0x1001 # chargement de l'adresse de l'entier dans $3
   lw $4, 0($3)   # lecture de la valeur dans $4 en mémoire
   ori $2, $0, 1  # affichage de l'entier
   syscall

   ori $2, $0, 10 # fin de programme
   syscall
```

Exemple : Afficher un entier stocké en mémoire

- Allouer et inititaliser un entier dans la section de données, avec la bonne directive
- Affichage d'un entier : appel système numéro 1 + entier à afficher dans `$4`
  L'entier à afficher se trouver en mémoire ⇒ transfert mémoire en lecture pour récupérer sa valeur
- Terminaison du programme : appel système numéro 10

```
.data
n:   .word 0x18   # allocation d'un entier initialisé à24

.text
   lui $3, 0x1001 # chargement de l'adresse de l'entier dans $3
   lw $4, 0($3)   # lecture de la valeur dans $4 en mémoire
   ori $2, $0, 1  # affichage de l'entier
   syscall

   ori $2, $0, 10 # fin de programme
   syscall
```

Exemple : Modifier un entier stocké en mémoire

Modifier un entier stocké en mémoire après l'avoir affiché et lu sa nouvelle valeur au clavier

- Allouer et initialiser un entier, l'afficher : programme précédent
- Lecture d'un entier au clavier : appel système numéro 5 + valeur lue dans `$2` après l'appel système
- Écriture de la valeur lue dans la mémoire
- Terminaison du programme : appel système numéro 10

```
.data
n:   .word 0x18   # allocation d'un entier initialisé à24
.text
   lui $3, 0x1001 # chargement de l'adresse de l'entier dans $3
   lw $4, 0($3)   # lecture de la valeur dans $4 en mémoire
   ori $2, $0, 1  # affichage de l'entier
   syscall
   ori $2, $0, 5 # service lire un entier
   syscall
   # entier lu est dans $2
   lui $3, 0x1001 # rechargement de l'adresse, $3 non persistant
   sw $2, 0($3)   # écriture nouvelle valeur en mémoire

   ori $2, $0, 10 # fin de programme
   syscall
```

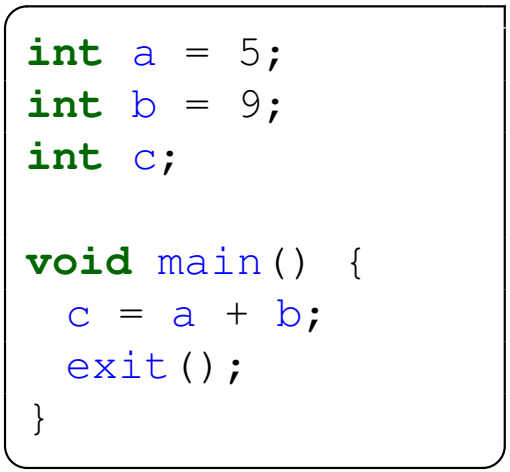
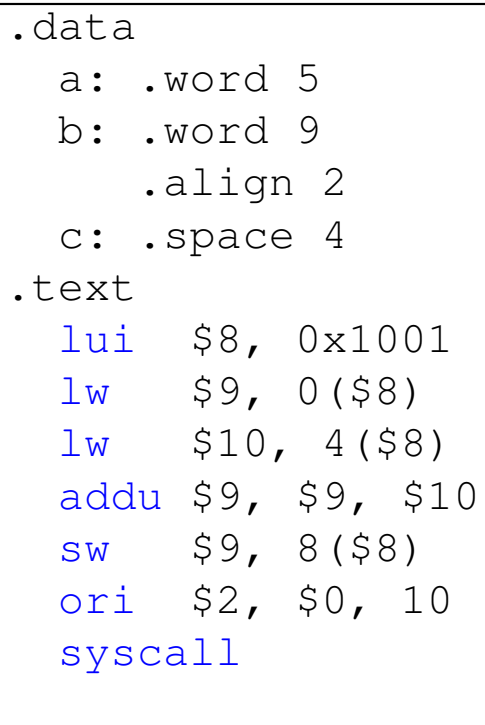
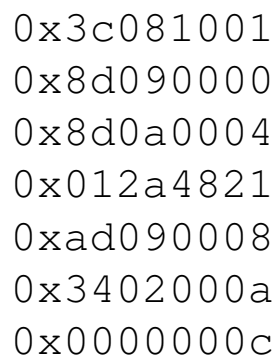
Exemple : Modifier un entier stocké en mémoire

Modifier un entier stocké en mémoire après l'avoir affiché et lu sa nouvelle valeur au clavier

- Allouer et initialiser un entier, l'afficher : programme précédent
- Lecture d'un entier au clavier : appel système numéro 5 + valeur lue dans `$2` après l'appel système
- Écriture de la valeur lue dans la mémoire
- Terminaison du programme : appel système numéro 10

```
.data
n:   .word 0x18   # allocation d'un entier initialisé à24
.text
   lui $3, 0x1001 # chargement de l'adresse de l'entier dans $3
   lw $4, 0($3)   # lecture de la valeur dans $4 en mémoire
   ori $2, $0, 1  # affichage de l'entier
   syscall
   ori $2, $0, 5 # service lire un entier
   syscall
   # entier lu est dans $2
   lui $3, 0x1001 # rechargement de l'adresse, $3 non persistant
   sw $2, 0($3)   # écriture nouvelle valeur en mémoire

   ori $2, $0, 10 # fin de programme
   syscall
```

Exemple : programme C, assembleur et binaire

Programme C	ASM	Binaire
		

```
int a = 5;
int b = 9;
int c;

void main() {
  c = a + b;
  exit();
}
```

```
.data
  a: .word 5
  b: .word 9
     .align 2
  c: .space 4
.text
  lui  $8, 0x1001
  lw   $9, 0($8)
  lw   $10, 4($8)
  addu $9, $9, $10
  sw   $9, 8($8)
  ori  $2, $0, 10
  syscall
```

```
0x3c081001
0x8d090000
0x8d0a0004
0x012a4821
0xad090008
0x3402000a
0x0000000c
```

Remarque sur le code assembleur

- On retrouve les 3 phases : 1) Lecture mémoire → registre 2) Opération registre → registre 3) Écriture registre → mémoire

Conclusion

On a vu :

- Mémoire : architecture, fonctionnement et structuration
- La déclaration de données globales, leur alignement et le calcul de leur adresse
- Les instructions de transfert mémoire (load et store)
- La manipulation de données globales en assembleur

Vous devez :

- Bien faire la différence entre les registres du processeur et la mémoire
- Comprendre la vue de la mémoire par mot et par octet (rangement little endian)
- Savoir utiliser les instructions de transfert mémoire (load et store)
- Savoir écrire des programmes avec des variables globales (les déclarer, calculer leur adresse, les utiliser dans le programme)

Prochain cours : instructions de saut et programme avec flot de contrôle non séquentiel