

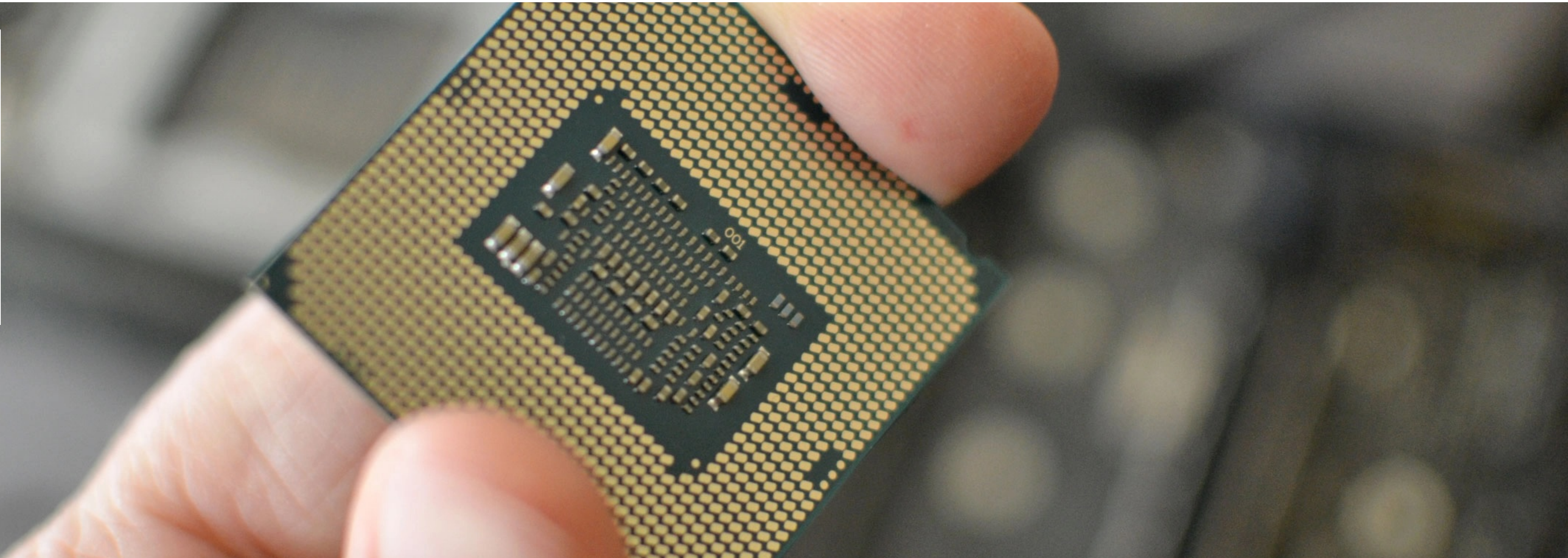
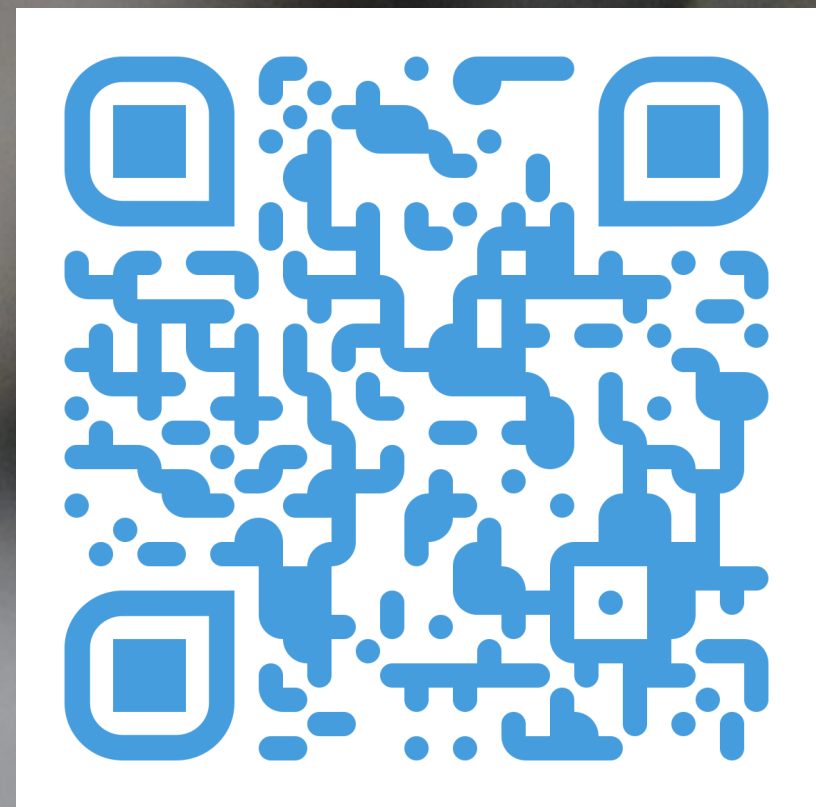
Architecture interne des ordinateurs



Pr. Dominique Ginhac
dginhac@u-bourgogne.fr



<https://ginhac.com/archi/04-processeur.pdf>

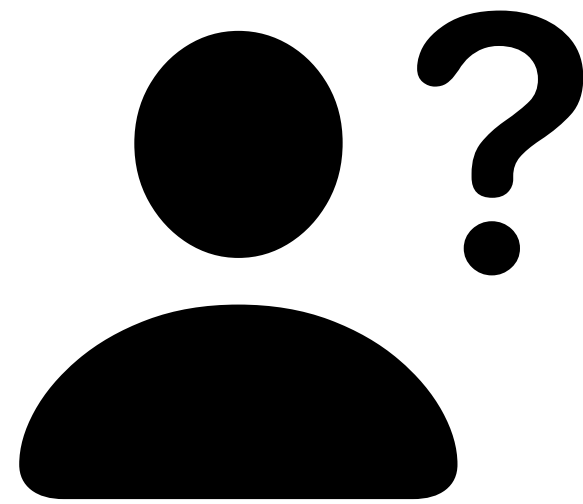


Etudier un processeur basique avec son jeu d'instructions minimal pour comprendre comment il exécute ses instructions et accomplit ses calculs.

Enjoy! 😊

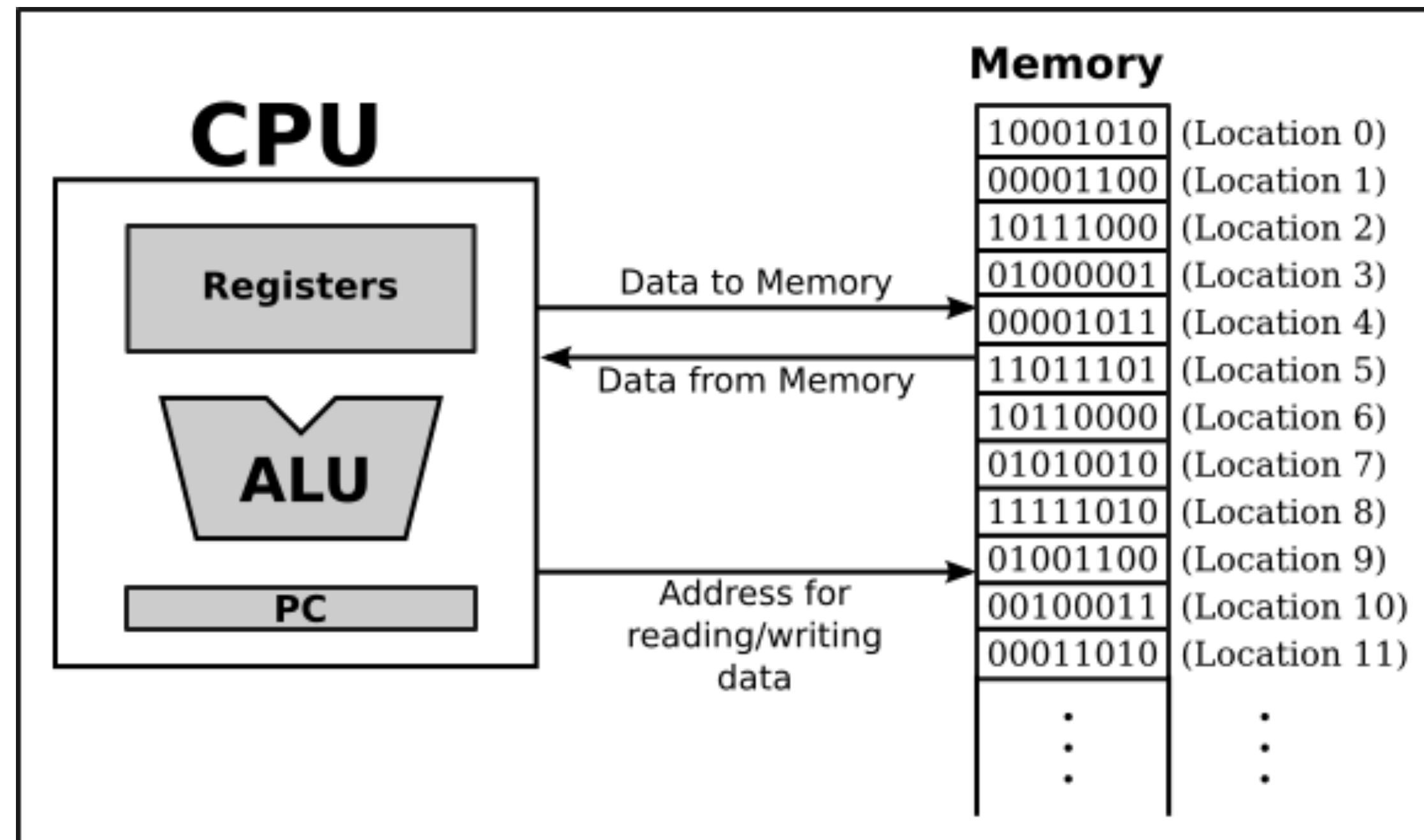
PROCESSEUR

CPU



Comment ça “marche” ?

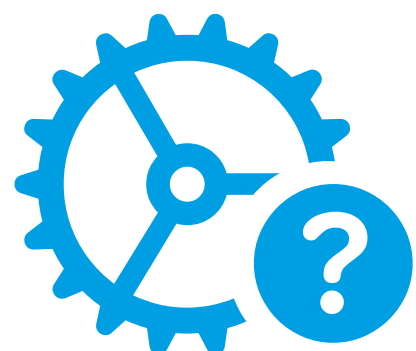
Processeur



Le CPU (Central Point Unit) est un **circuit intégré** constitué de milliers / millions / milliards de transistors.

Le CPU est l'**Unité de contrôle** (chef d'orchestre) qui est chargé de :

1. Gérer les **transferts de données** avec la RAM (en utilisant des registres)
2. **Interpréter** et **séquencer les instructions** (en utilisant l'ALU pour faire des calculs)

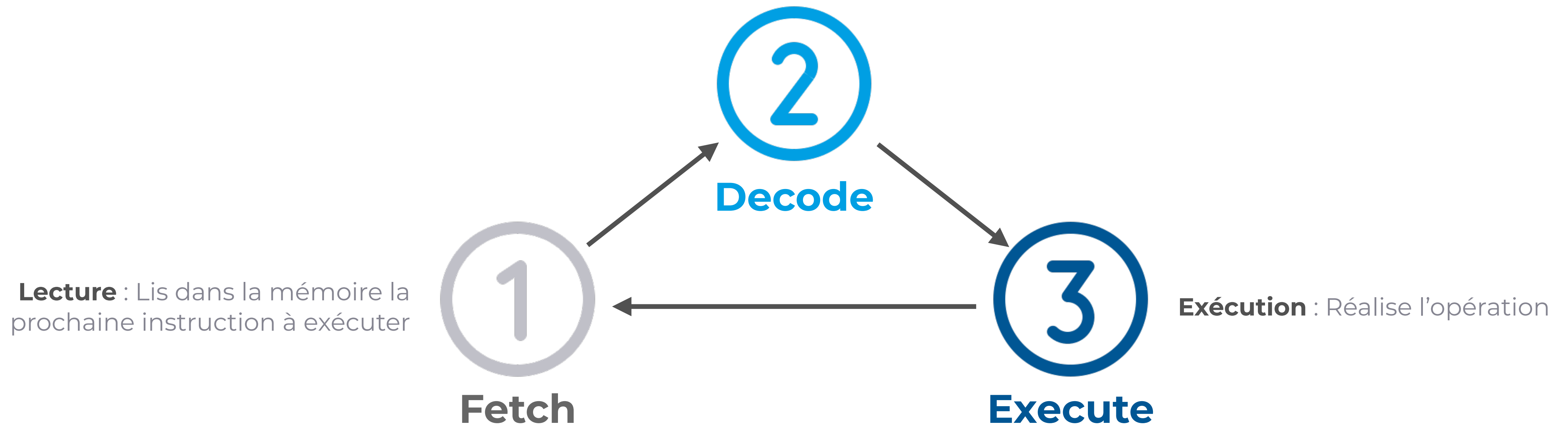


Comment le CPU exécute les **instructions** des programmes et traite les **données** ?

Comment fonctionne un CPU ?

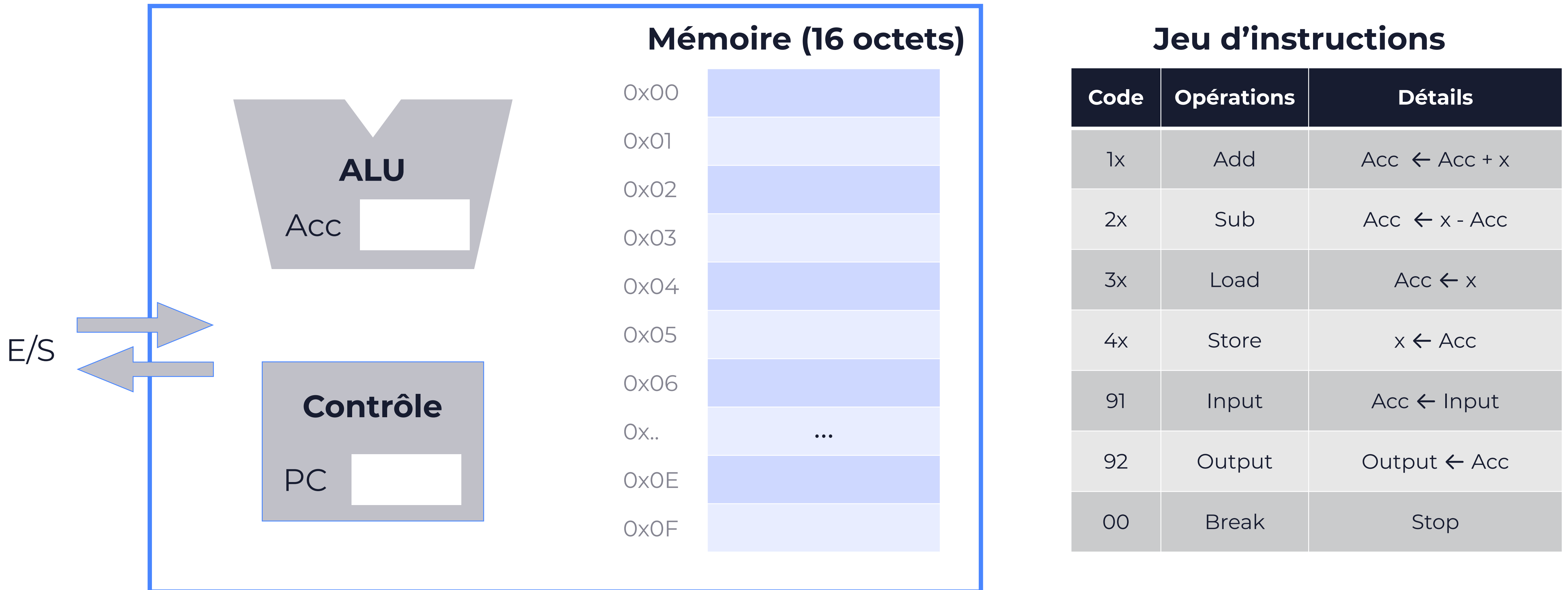
Selon un mécanisme “simple” de successions de cycles de 3 opérations de base

Décodage : Identifie l'opération à réaliser et configure le CPU pour réaliser cette opération



Exemple de CPU trivial

capable uniquement de faire des additions / soustractions de 2 nombres.



Opération : 4 + 9 - 5

Instructions possibles

Code	Opérations	Détails
1x	Add	$Acc \leftarrow Acc + x$
2x	Sub	$Acc \leftarrow x - Acc$
3x	Load	$Acc \leftarrow x$
4x	Store	$x \leftarrow Acc$
91	Input	$Acc \leftarrow Input$
92	Output	$Output \leftarrow Acc$
00	Break	Fin du programme

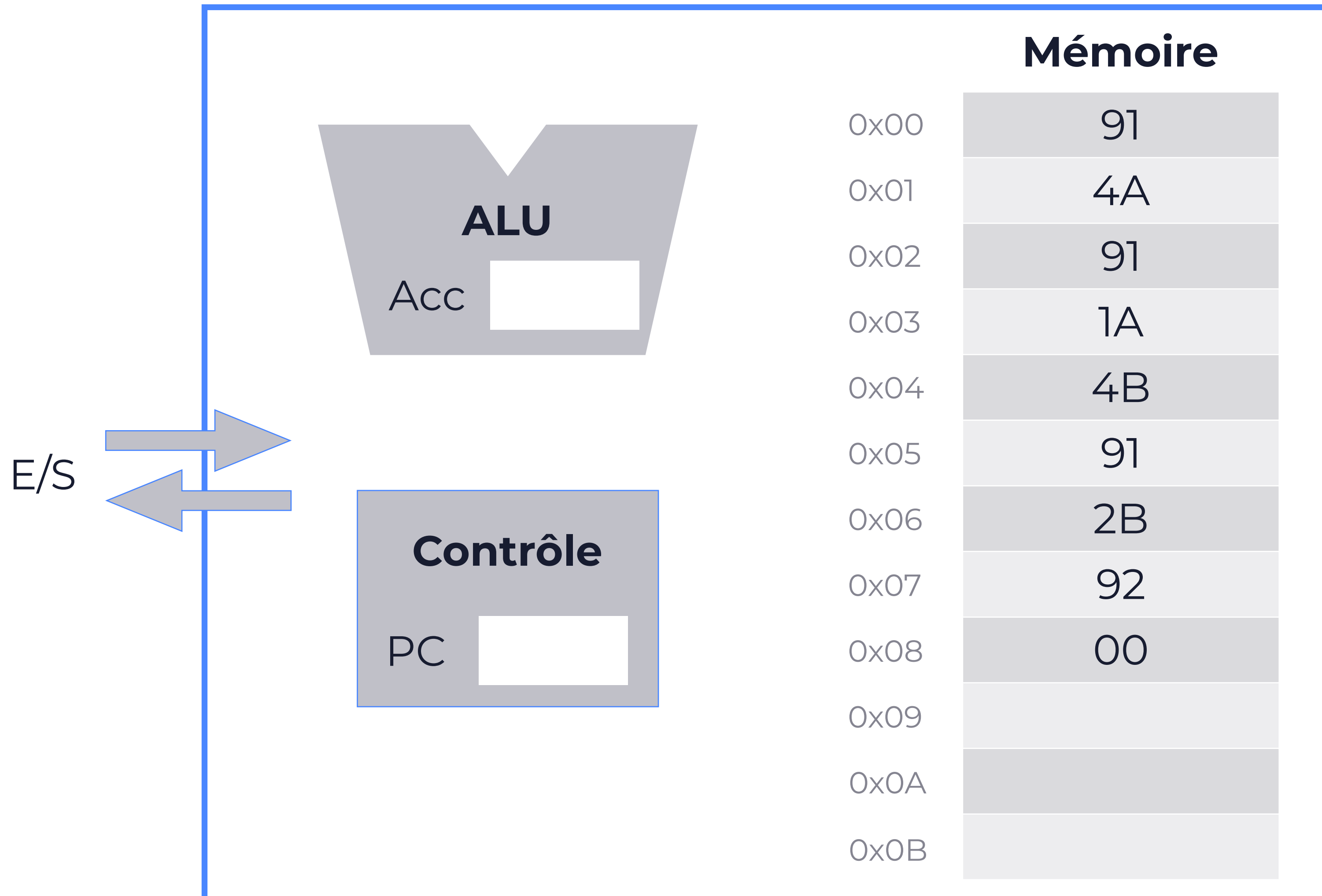
Algorithme

Demander valeur 1
Stocker valeur 1 dans mémoire A
Demander valeur 2
Add valeur 2 avec mémoire A
Stocker résultat dans mémoire B
Demander valeur 3
Soustraire mémoire B avec valeur 3
Afficher résultat
Fin

Code

91
4A
91
1A
4B
91
2B
92
00

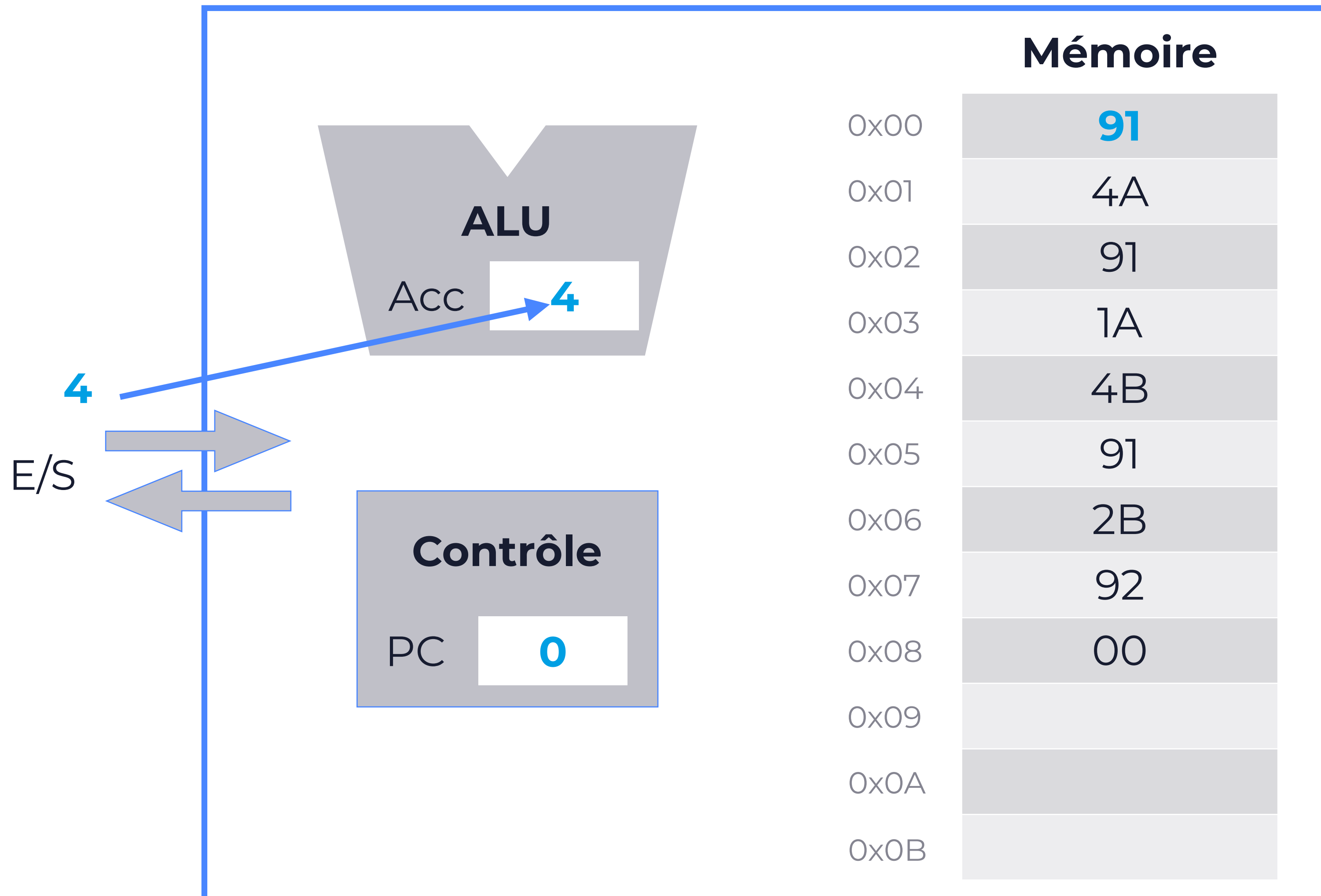
Opération : 4 + 9 - 5



Exécution du programme

1. Ecriture du code dans la mémoire
2. Exécution du code :
 1. Initialisation PC à 0
 2. Lecture mémoire indiquée par PC
 3. Décodage de l'instruction
 4. Exécution de l'instruction
 5. Incrémentation PC
 6. Retour au point 2 tant que l'instruction n'est pas 000

Opération : 4 + 9 - 5



Description pas à pas

PC = 0

Lecture / Décodage 91

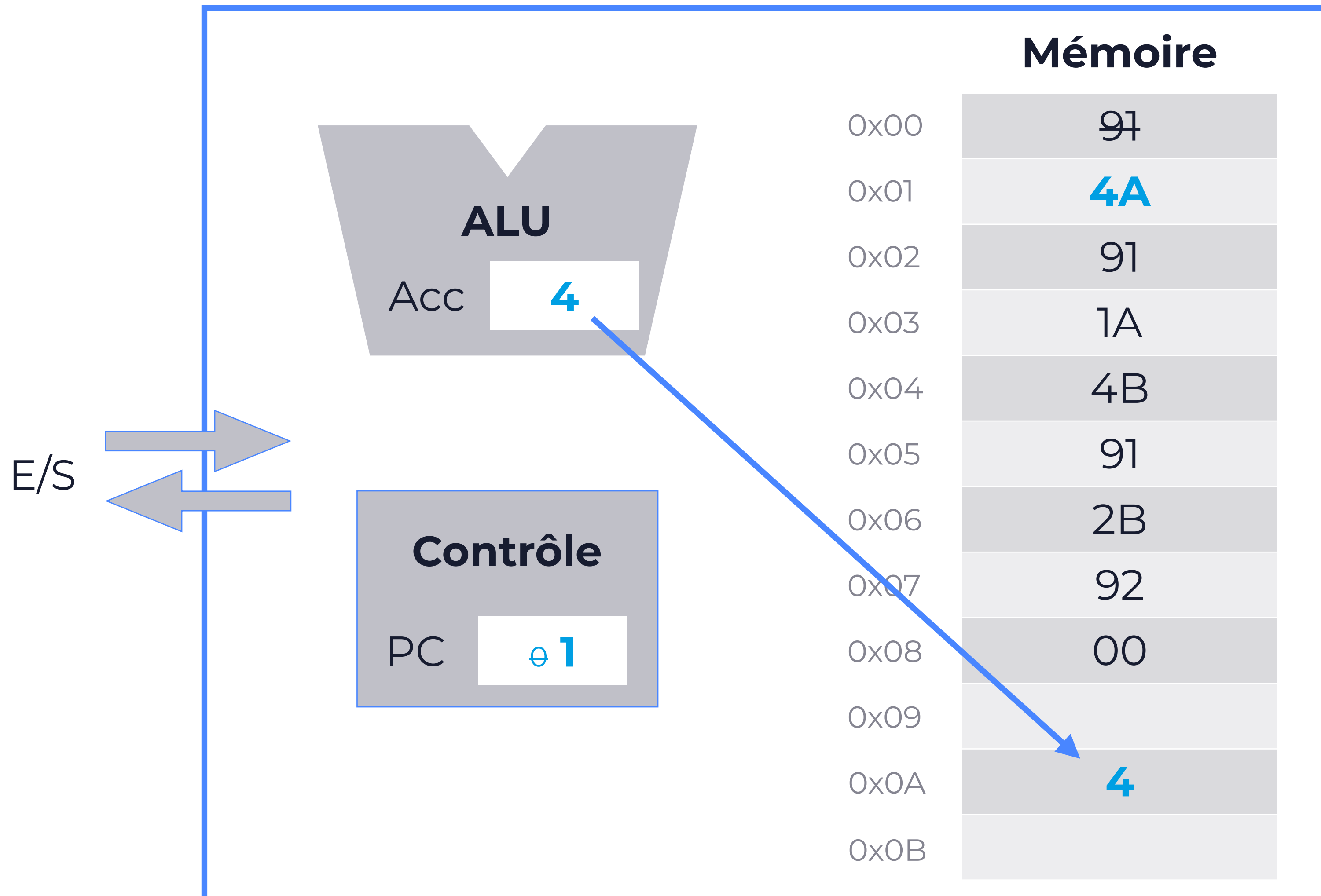
Demande valeur (Ex: 4)

Acc ← 4

Incrémentation PC

Code	Opérations	Détails
1x	Add	Acc ← Acc + x
2x	Sub	Acc ← x - Acc
3x	Load	Acc ← x
4x	Store	x ← Acc
91	Input	Acc ← Input
92	Output	Output ← Acc
00	Break	Stop

Opération : 4 + 9 - 5



Description pas à pas

PC = 1

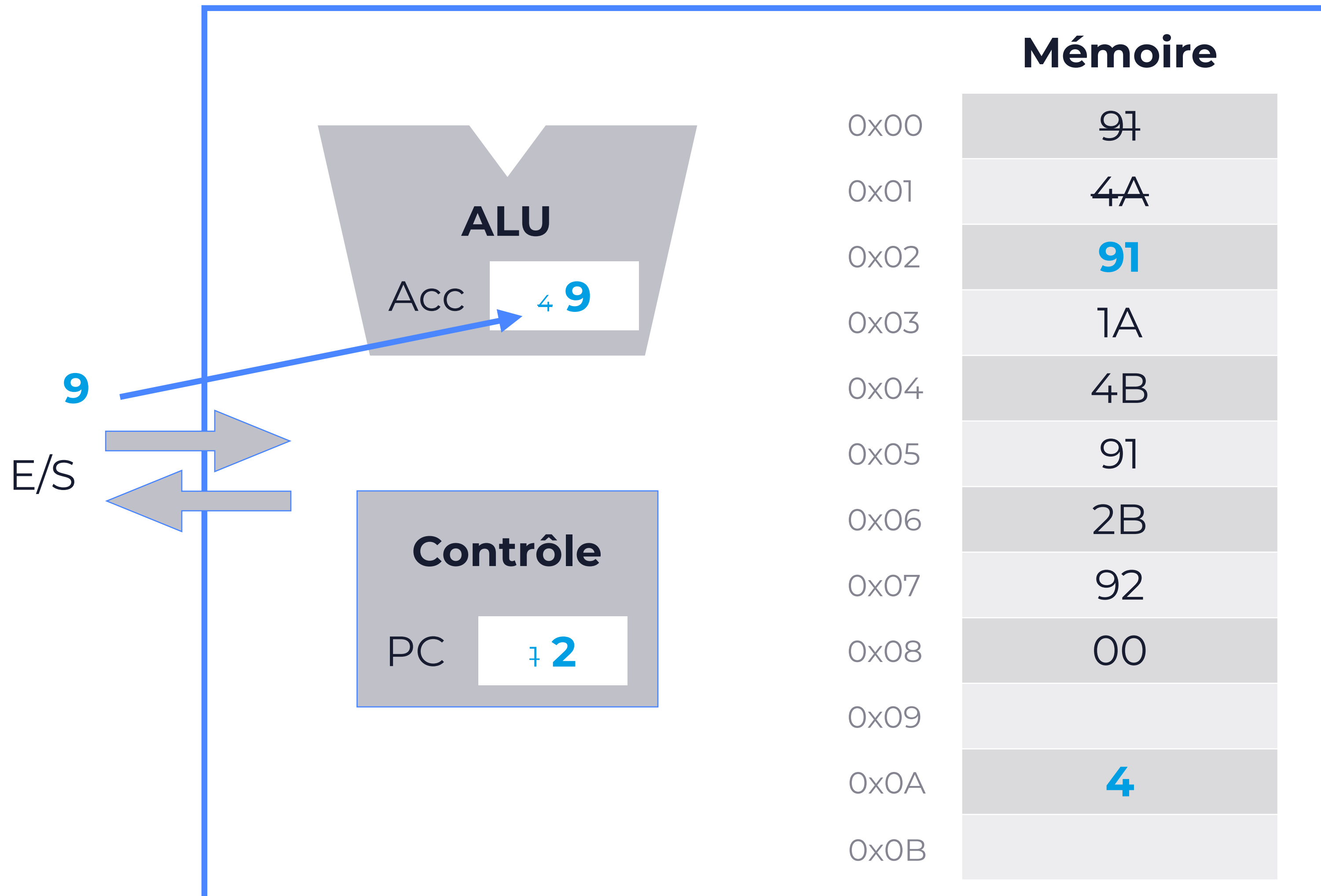
Lecture / Décodage 4A

MemA ← Acc

Incrémentation PC

Code	Opérations	Détails
1x	Add	Acc ← Acc + x
2x	Sub	Acc ← x - Acc
3x	Load	Acc ← x
4x	Store	x ← Acc
91	Input	Acc ← Input
92	Output	Output ← Acc
00	Break	Stop

Opération : 4 + 9 - 5



Description pas à pas

PC = 2

Lecture / Décodage 91

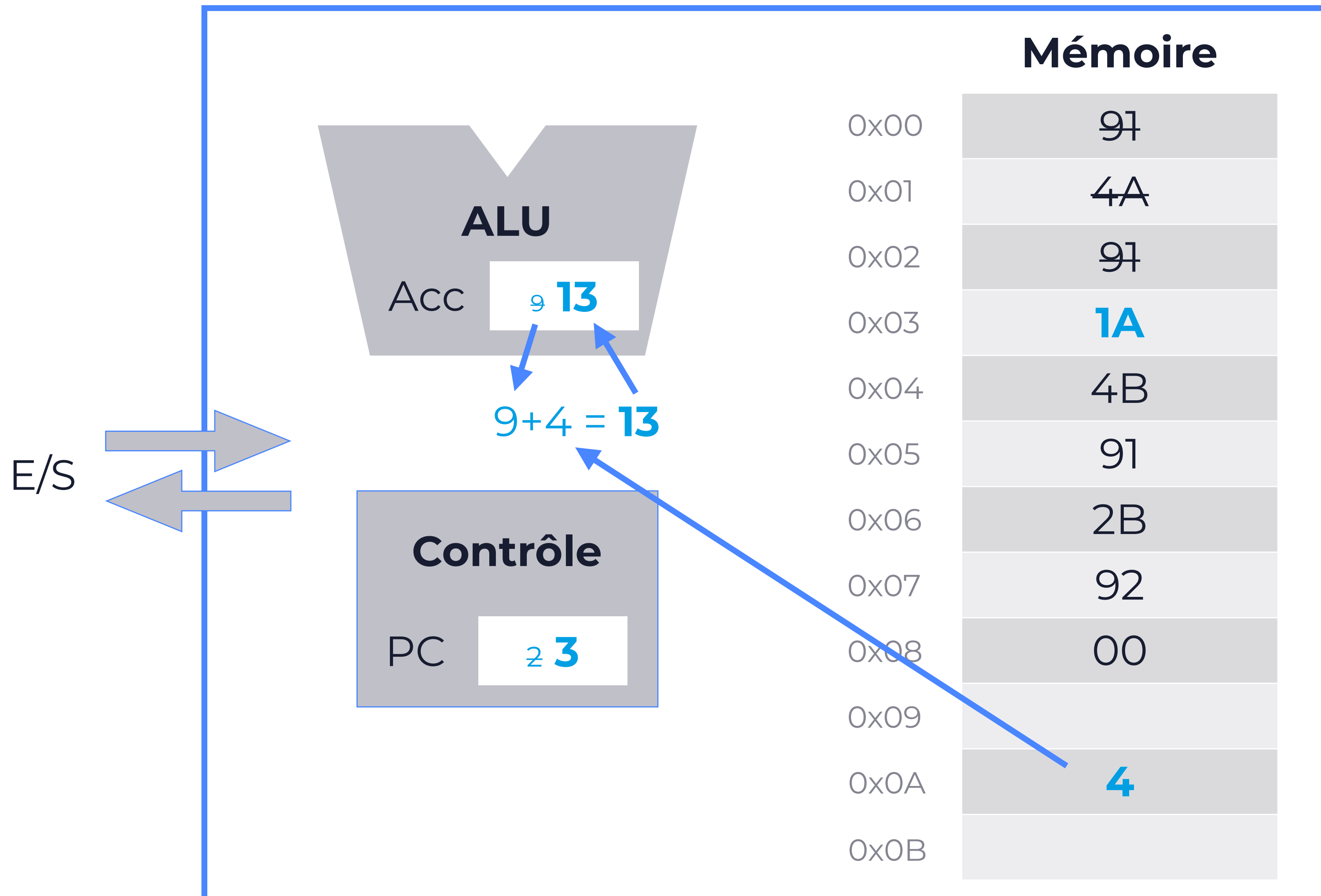
Demande valeur (Ex: 9)

Acc ← 9

Incrémentation PC

Code	Opérations	Détails
1x	Add	Acc ← Acc + x
2x	Sub	Acc ← x - Acc
3x	Load	Acc ← x
4x	Store	x ← Acc
91	Input	Acc ← Input
92	Output	Output ← Acc
00	Break	Stop

Opération : 4 + 9 - 5



Description pas à pas

PC = 3

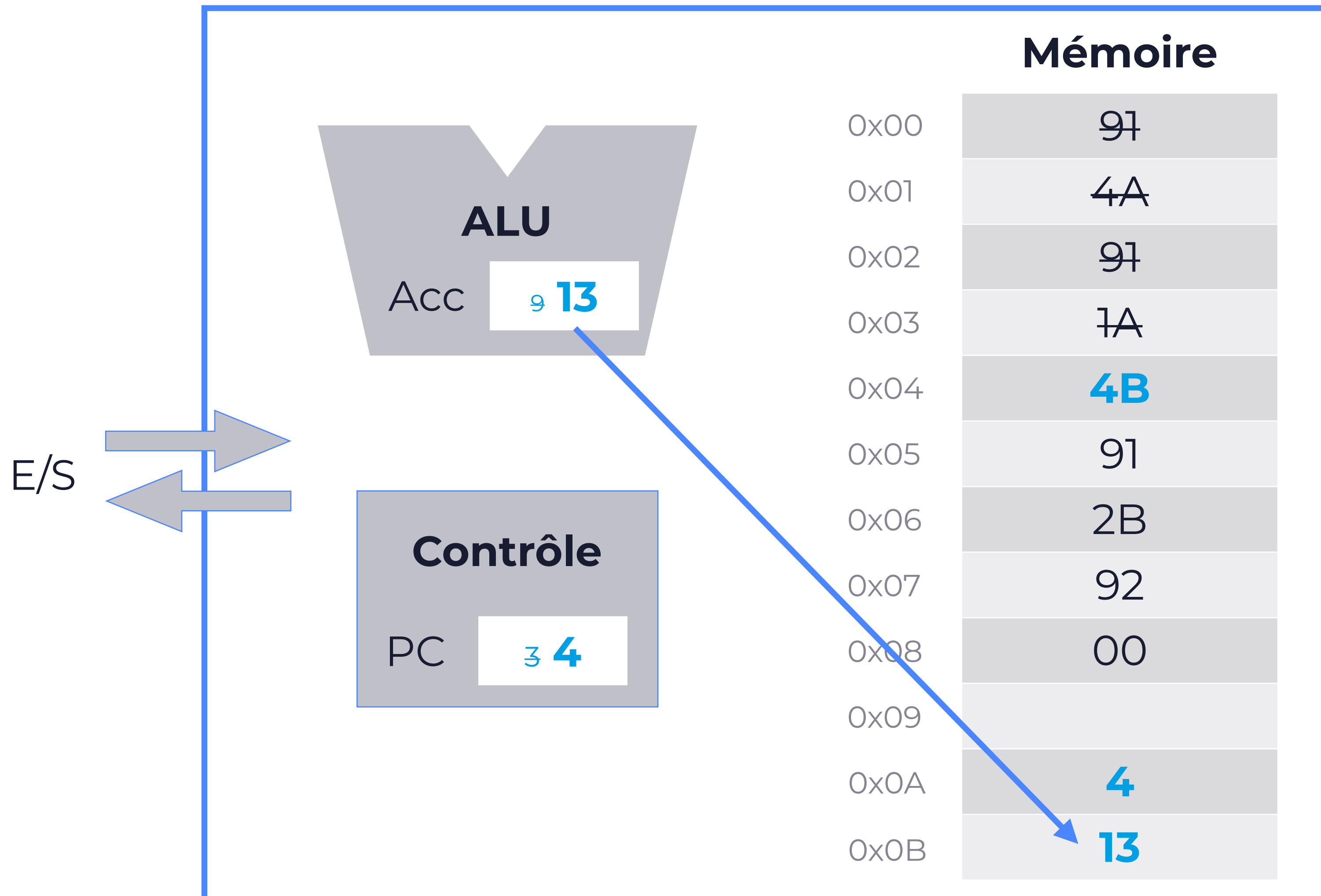
Lecture / Décodage 1A

Acc \leftarrow Acc + Mémoire A

Incrémentation PC

Code	Opérations	Détails
1x	Add	Acc \leftarrow Acc + x
2x	Sub	Acc \leftarrow x - Acc
3x	Load	Acc \leftarrow x
4x	Store	x \leftarrow Acc
91	Input	Acc \leftarrow Input
92	Output	Output \leftarrow Acc
00	Break	Stop

Opération : 4 + 9 - 5



Description pas à pas

PC = 4

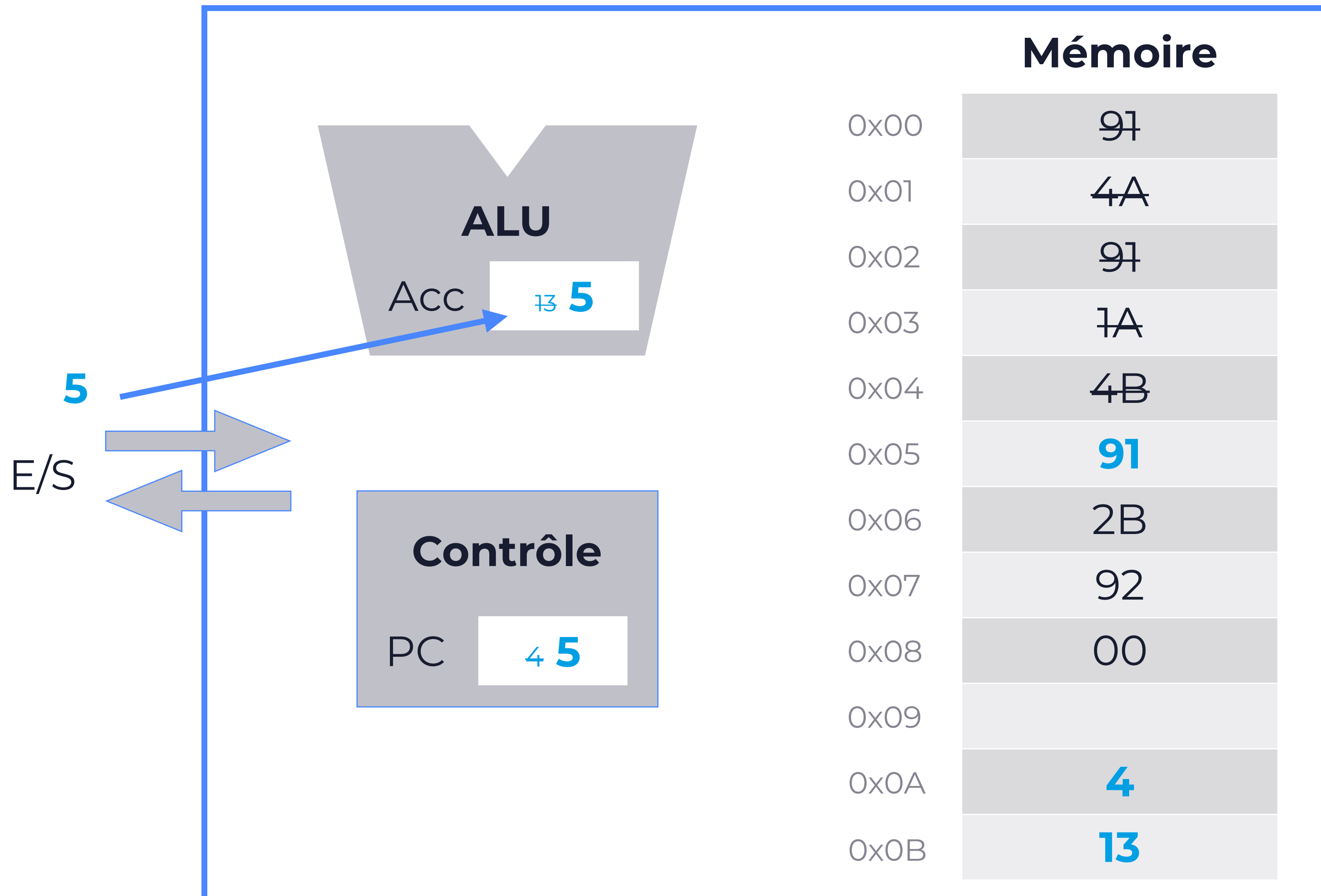
Lecture / Décodage 4B

MemB ← Acc

Incrémentation PC

Code	Opérations	Détails
1x	Add	Acc ← Acc + x
2x	Sub	Acc ← x - Acc
3x	Load	Acc ← x
4x	Store	x ← Acc
91	Input	Acc ← Input
92	Output	Output ← Acc
00	Break	Stop

Opération : 4 + 9 - 5



Description pas à pas

PC = 5

Lecture / Décodage 91

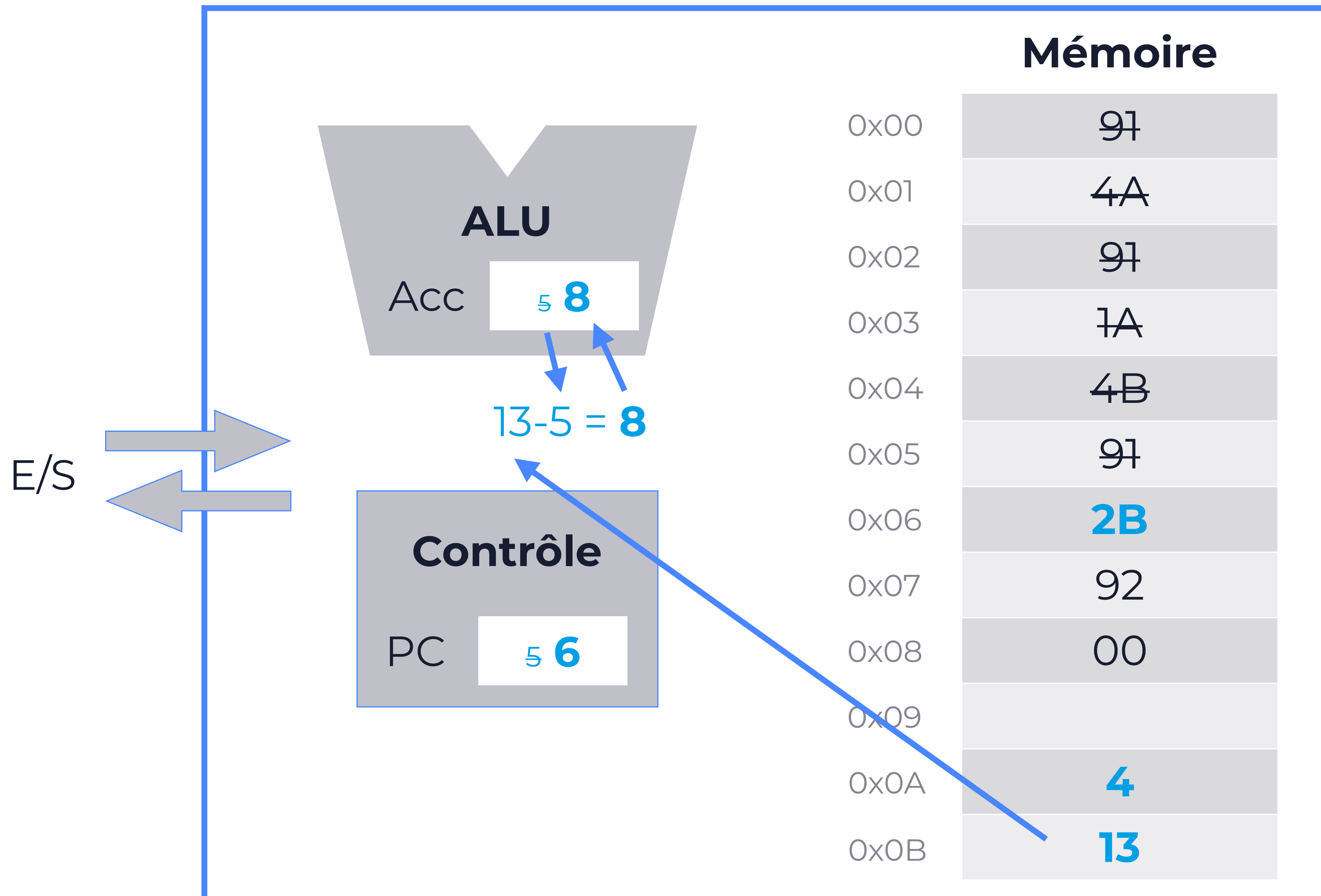
Demande valeur (Ex: 5)

Acc ← 5

Incrémentation PC

Code	Opérations	Détails
1x	Add	Acc ← Acc + x
2x	Sub	Acc ← x - Acc
3x	Load	Acc ← x
4x	Store	x ← Acc
91	Input	Acc ← Input
92	Output	Output ← Acc
00	Break	Stop

Opération : 4 + 9 - 5



Description pas à pas

PC = 6

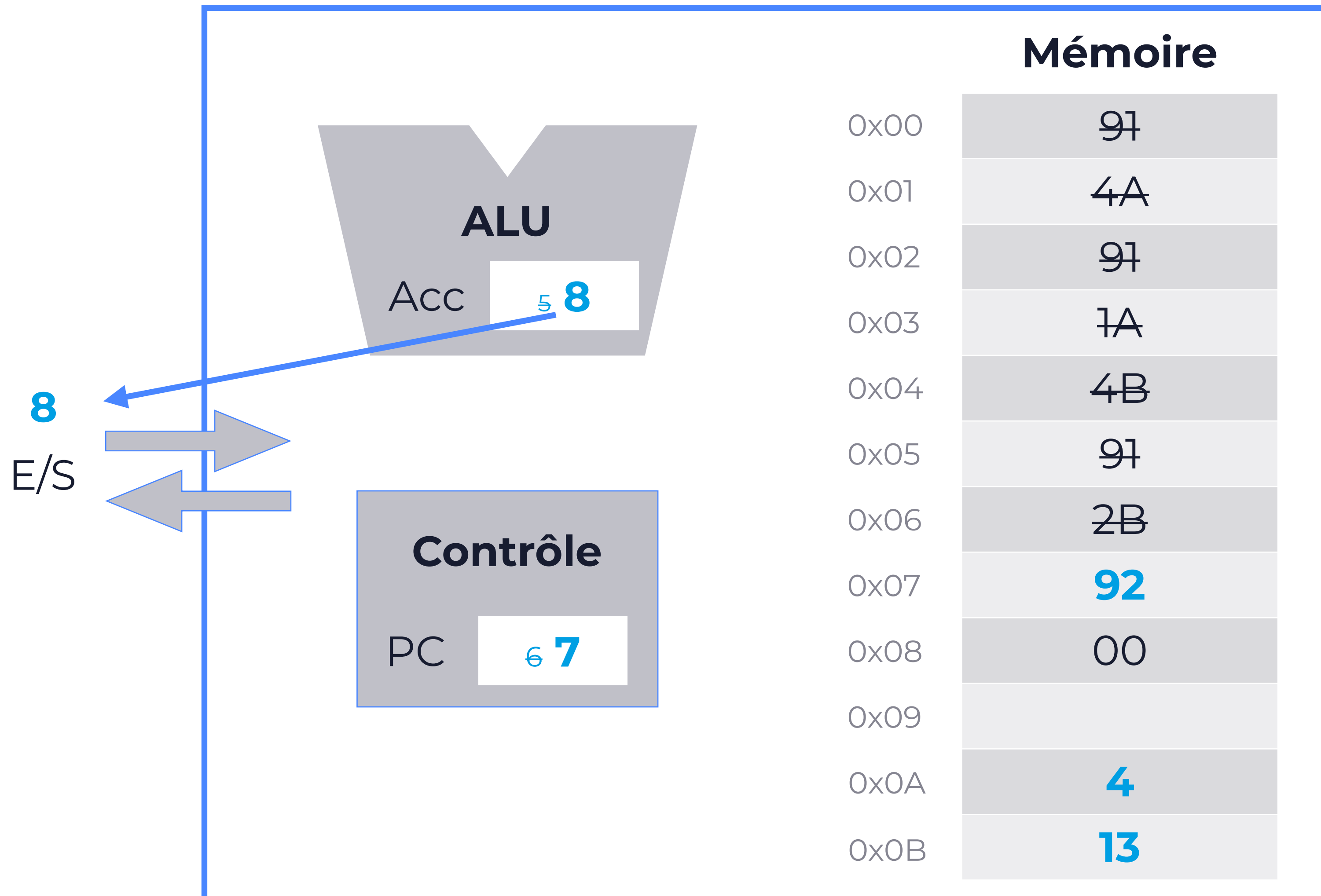
Lecture / Décodage 2B

Acc ← Mémoire B - Acc

Incrémentation PC

Code	Opérations	Détails
1x	Add	Acc ← Acc + x
2x	Sub	Acc ← x - Acc
3x	Load	Acc ← x
4x	Store	x ← Acc
91	Input	Acc ← Input
92	Output	Output ← Acc
00	Break	Stop

Opération : 4 + 9 - 5



Description pas à pas

PC = 7

Lecture / Décodage 92

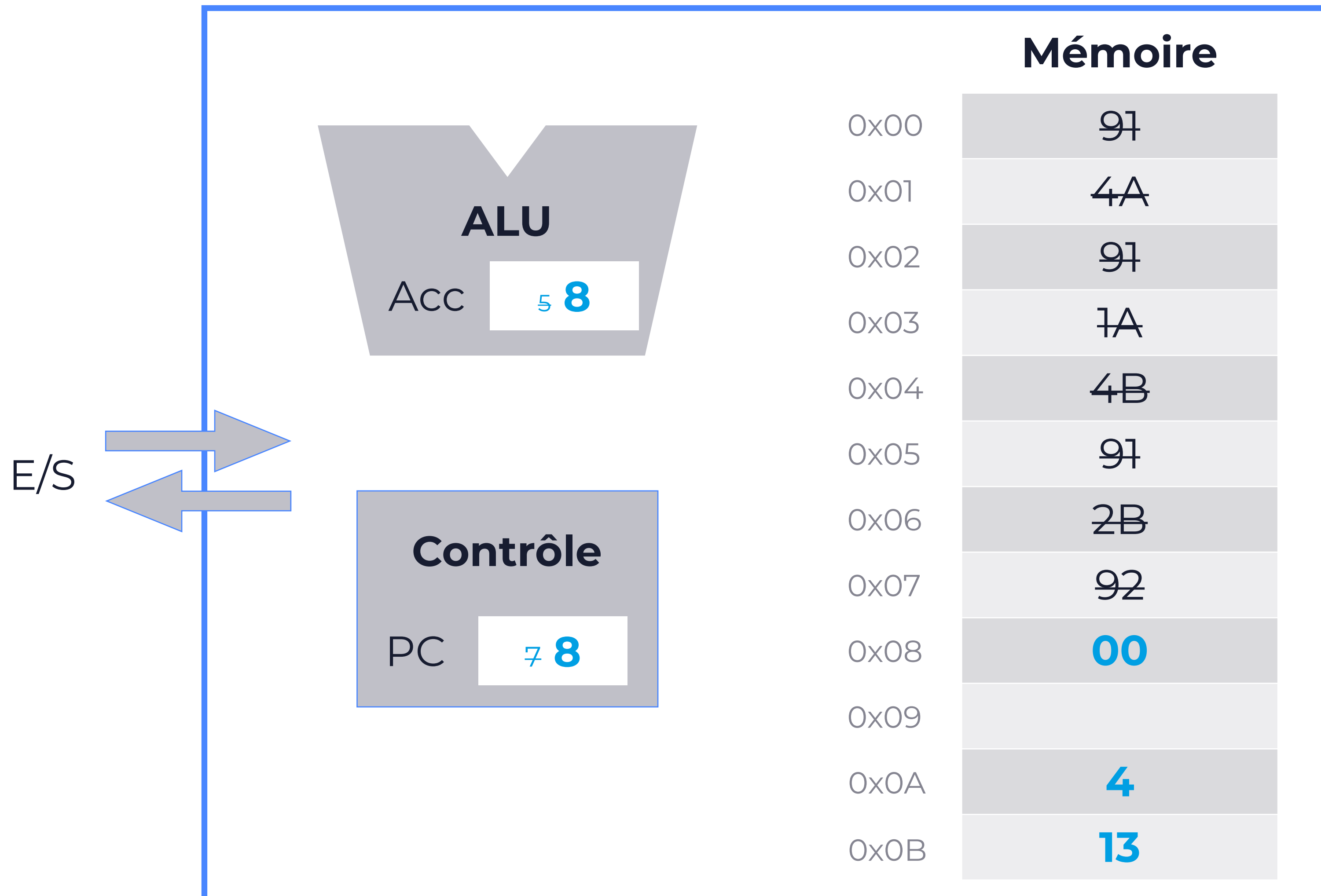
Affiche résultat

Output ← 8

Incrémentation PC

Code	Opérations	Détails
1x	Add	Acc ← Acc + x
2x	Sub	Acc ← x - Acc
3x	Load	Acc ← x
4x	Store	x ← Acc
91	Input	Acc ← Input
92	Output	Output ← Acc
00	Break	Stop

Opération : 4 + 9 - 5



Description pas à pas

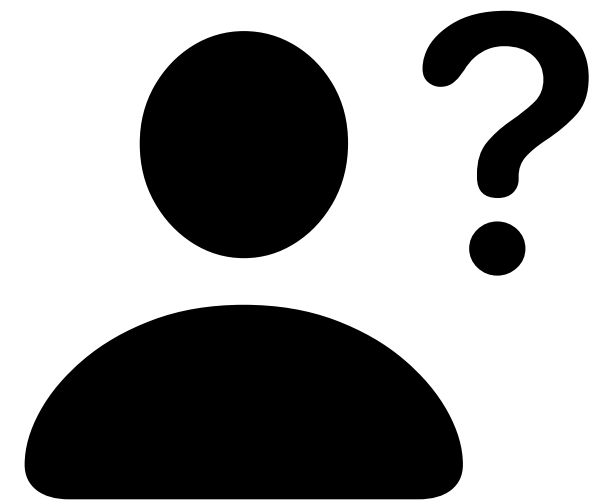
PC = 8

Arrêt du programme



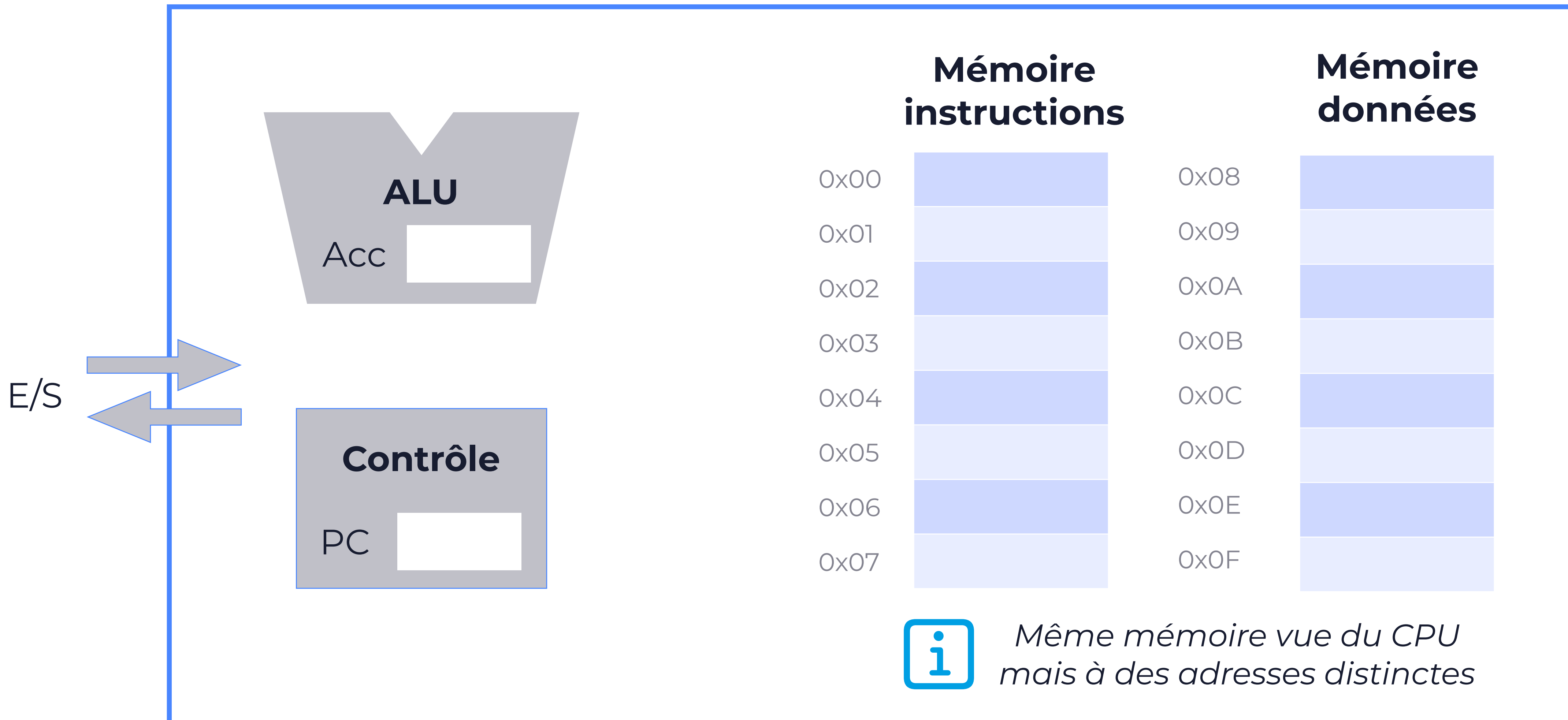
Code	Opérations	Détails
1x	Add	Acc ← Acc + x
2x	Sub	Acc ← x - Acc
3x	Load	Acc ← x
4x	Store	x ← Acc
91	Input	Acc ← Input
92	Output	Output ← Acc
00	Break	Stop

Que manque t'il à notre processeur trivial pour
DEVENIR UN VRAI PROCESSEUR ?

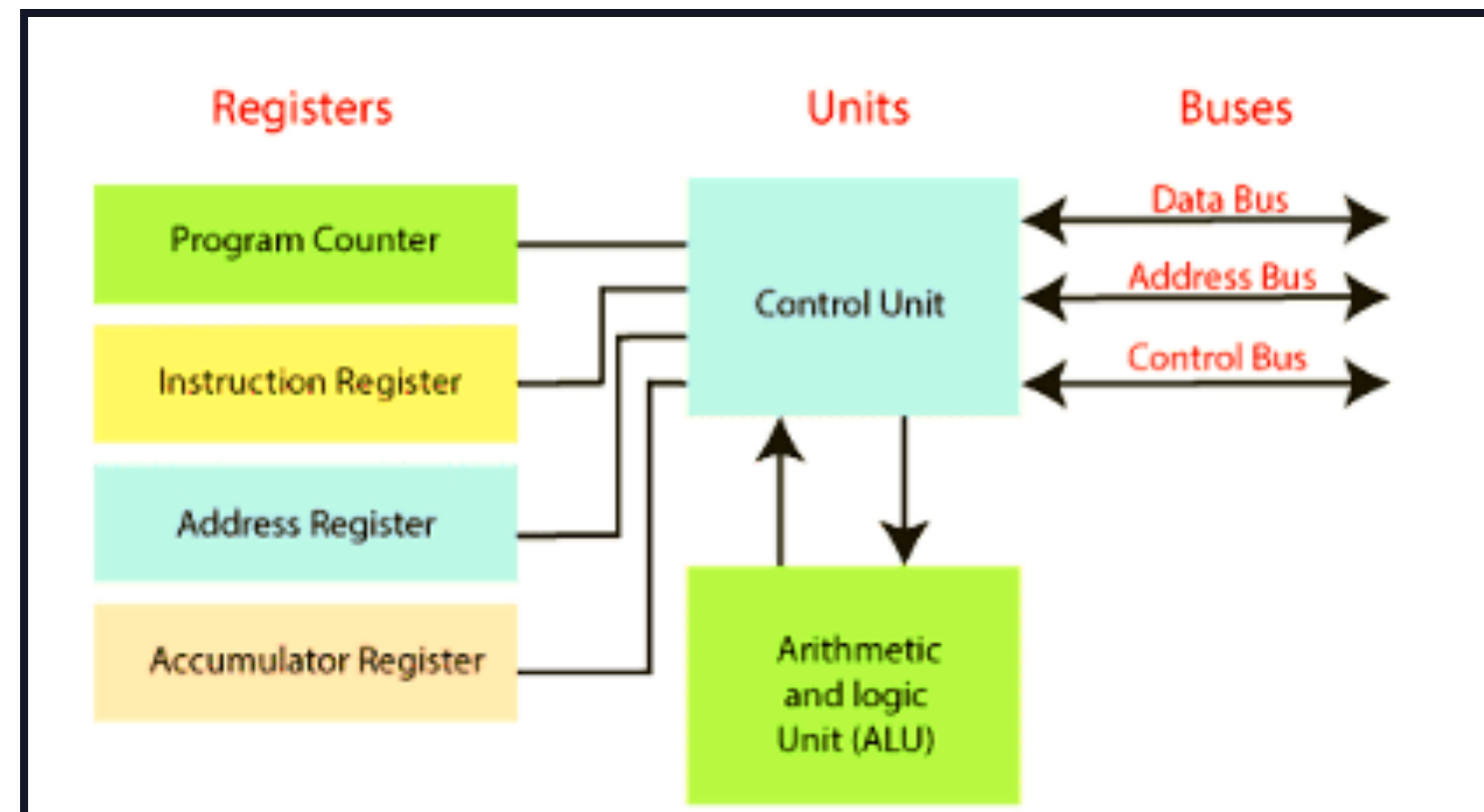


Modification du CPU trivial

Etape 1 : Séparation mémoire d'instructions et mémoire de données



Ajout de Registres



Le CPU ne contient plus un seul Accumulateur mais plusieurs éléments de **stockage temporaires** (nommés **registres**).

Les registres en **quantité limitée** permettent des **manipulations de données** internes au CPU.

Le **temps d'accès** aux registres est bien **plus court** que pour la mémoire RAM externe au CPU

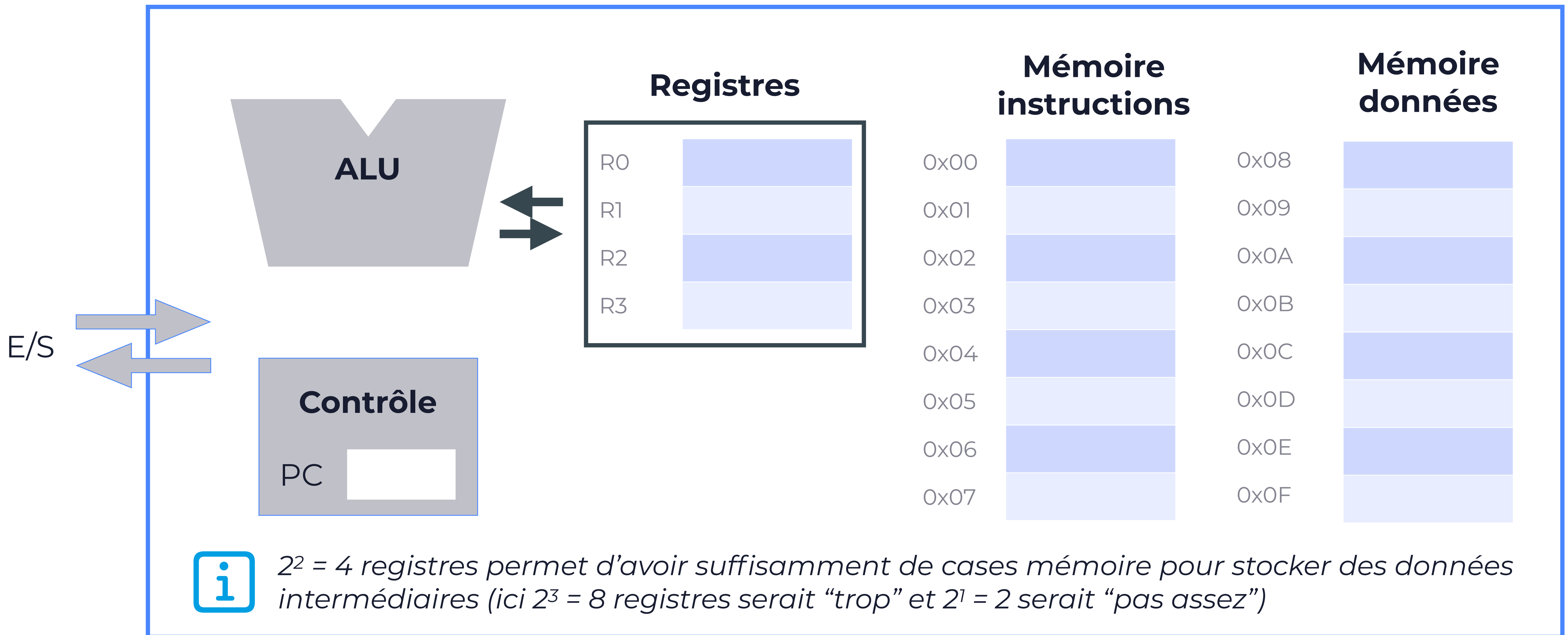
Les registres **généraux** permettent de stocker temporairement des **données** issues de l'exécution des programmes

Les registres **particuliers** ont des rôles spécifiques :

- ➔ PC : "Program Counter" qui stocke l'adresse de la prochaine instruction à exécuter
- ➔ ACC : Accumulateur qui stocke les résultats temporaires des calculs faits par l'ALU
- ➔ MAR : "Memory Address Register" qui contient la prochaine adresse à lire
- ➔ MDR : "Memory Data Register" qui stocke les informations à envoyer dans la RAM
- ➔ Registres de Statuts qui stockent des drapeaux : Carry (C), Overflow (OV), Zero (Z), Negative (N), ..²⁰

Modification du CPU trivial

Etape 2 : Remplacement de ACC par 4 registres généraux



Instruction et Jeu d'instructions

Une **instruction** est une “**action**” pouvant être exécutée par le processeur (Ex : lire / écrire données, additionner, multiplier, ...)

Le **jeu d'instructions** (ISA: Instruction Set Architecture) représente **toutes les instructions** pouvant être exécutées par un CPU donné.

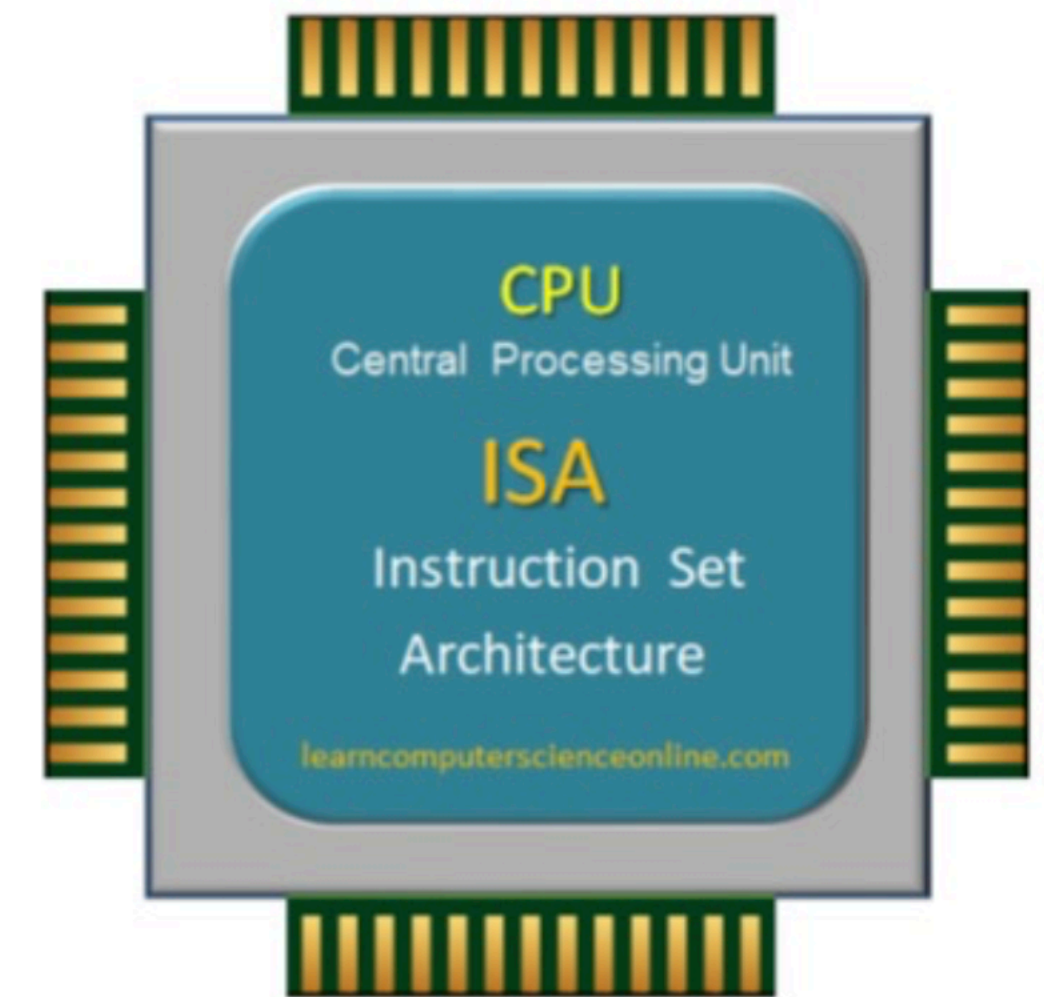
Il existe de nombreux types d'instructions communes à tous les CPU :

Mouvement de données	Lecture (Load), Ecriture (Store) mémoire
Arithmétique	Add, Subtract, Multiply, Divide, Shift, ...
Logique	Or, And, Not, Xor, ...
Contrôle	Saut et branchement, Contrôle du système, appels de fonctions
Entrées / Sortie	Récupération (In), Envoi (Out) de données d'un périphérique
Autres	Floating Point Arithmetic, SIMD, Vector operations, ...

Anatomie d'un Jeu d'instructions

Le jeu d'instructions constitue l'ensemble des **opérations de base** disponibles dans la micro-architecture du CPU, c'est-à-dire l'ensemble des opérations qui peuvent être décodées et exécutées directement par le CPU, soit matériellement, soit par exécution d'un micro-code.

L'ISA fournit une "**interface software**" sous la forme d'un **langage assembleur** permettant au développeur d'écrire des programmes et d'interagir "directement" avec le hardware.



Chaque **famille de CPUs** possède son ISA.

La rétro-compatibilité de l'ISA permet de faire exécuter des "vieux programmes" sur des machines récentes. Ex: L'ISA x86 des Intel Core d'aujourd'hui est héritée du processeur Intel 8088 de 1981 qui a été utilisé pour le premier IBM PC.

Anatomie d'un Jeu d'instructions

RISC vs CISC

Reduced Instruction Set Computer

- ➔ Instructions simples “primitives” hardware en faible nombre
- ➔ 1 instruction = 1 seule opération élémentaire
- ➔ Décodage facile et exécution en un cycle d'horloge
- ➔ Fréquence d'horloge élevée et vitesse d'exécution plus grande
- ➔ Code plus long (x 2 par rapport à CISC) avec nombreuses instructions
- ➔ Ex: ARM, RISC-V, MIPS

Complex Instruction Set Computer

- ➔ Instructions complexes et nombreuses utilisant du micro-code (plus lent)
- ➔ 1 instruction = plusieurs opérations élémentaires
- ➔ Décodage complexe et exécution en un nombre variable de cycles d'horloge
- ➔ Fréquence d'horloge plus basse et vitesse d'exécution limitée
- ➔ Code plus court avec instructions proches des langages de haut niveau,
- ➔ Ex: Intel, AMD

RISC vs CISC

Code C d'une addition

```
int main() {  
    int a = 42;  
    int b = 1337;  
    int c = a + b;  
    return 0;  
}
```

Compilation : gcc add.c -S

Code **Assembleur ARM** (Apple M1 Max - RISC)

```
mov    w8, #42  
str    w8, [sp, #8]  
mov    w8, #1337  
str    w8, [sp, #4]  
ldr    w8, [sp, #8]  
ldr    w9, [sp, #4]  
add    w8, w8, w9  
str    w8, [sp]  
add    sp, sp, #16
```

Code **Assembleur X86** (Intel Core I3 - CISC)

```
movl   $42, -4(%rbp)  
movl   $1337, -8(%rbp)  
movl   -4(%rbp), %edx  
movl   -8(%rbp), %eax  
addl   %edx, %eax  
movl   %eax, -12(%rbp)  
movl   $0, %eax  
popq   %rbp
```



Un **même code source** de haut niveau → des **codes assembleurs différents**

Code assembleur lié à l'architecture du processeur : CISC pour X86, RISC pour M1

Propriétés d'un jeu d'instructions RISC

→ Codage uniforme des instructions

Toutes les instructions sont codées avec un même nombre de bits, ce qui facilite le décodage des instructions.

→ Registres indifférenciés et nombreux

Tous les registres peuvent être utilisés dans tous les contextes.

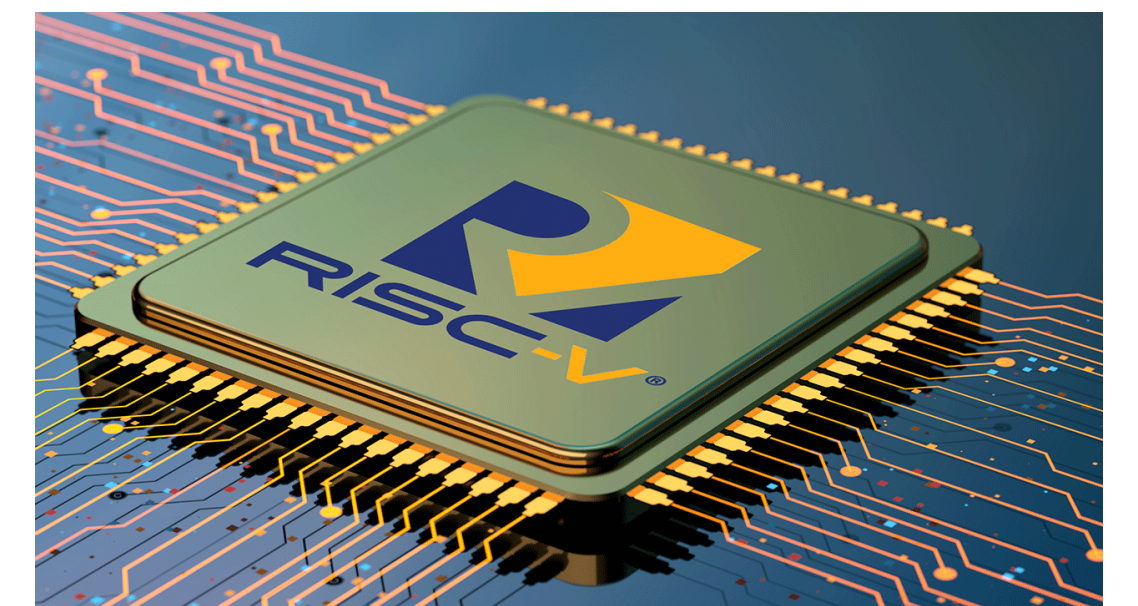
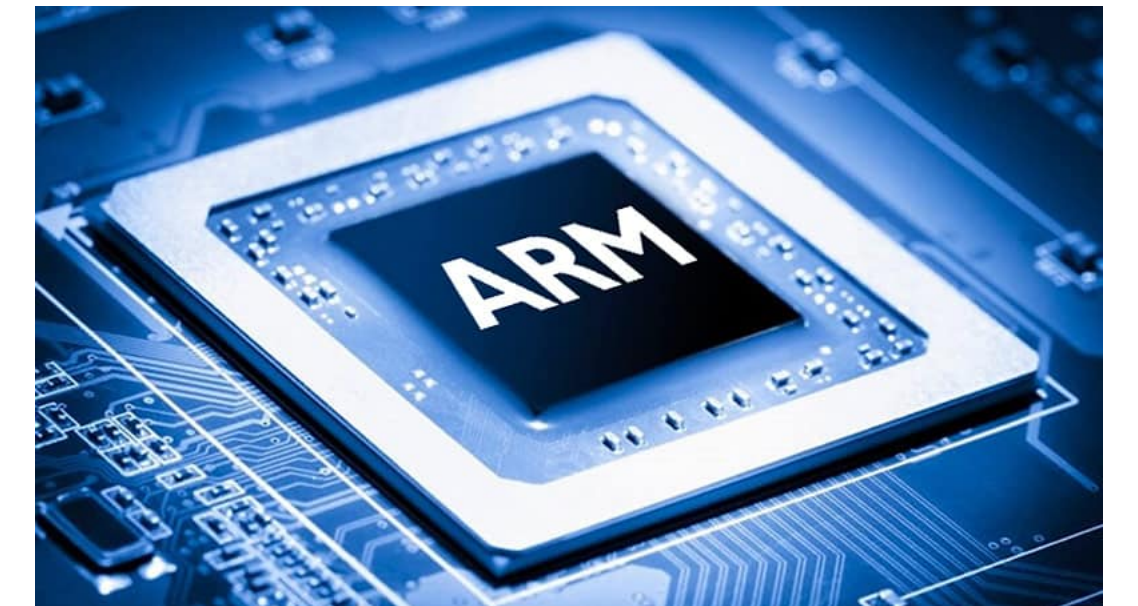
→ Limitation des accès mémoire

Les seules instructions ayant accès à la mémoire sont les instructions de chargement et de stockage.

Toutes les autres instructions travaillent uniquement sur les registres, ce qui garantit des performances maximales (plus aucun transfert mémoire).

→ Nombre réduit de types de données

Les seuls types de données supportés sont les entiers de différentes tailles (8, 16, 32 et 64 bits) et les nombres réels en simple et double précision.



Instruction et Jeu d'instructions

Code	Opérations	Détails
1x	Add	$Acc \leftarrow Acc + x$
2x	Sub	$Acc \leftarrow x - Acc$
3x	Load	$Acc \leftarrow x$
4x	Store	$x \leftarrow Acc$
91	Input	$Acc \leftarrow Input$
92	Output	$Output \leftarrow Acc$
00	Break	Fin du programme

Jeu d'instructions de notre
processeur trivial

Données manipulées : entiers non
signés stockés sur 8 bits ([0, 255])

Opérations arithmétiques (Add, Sub)

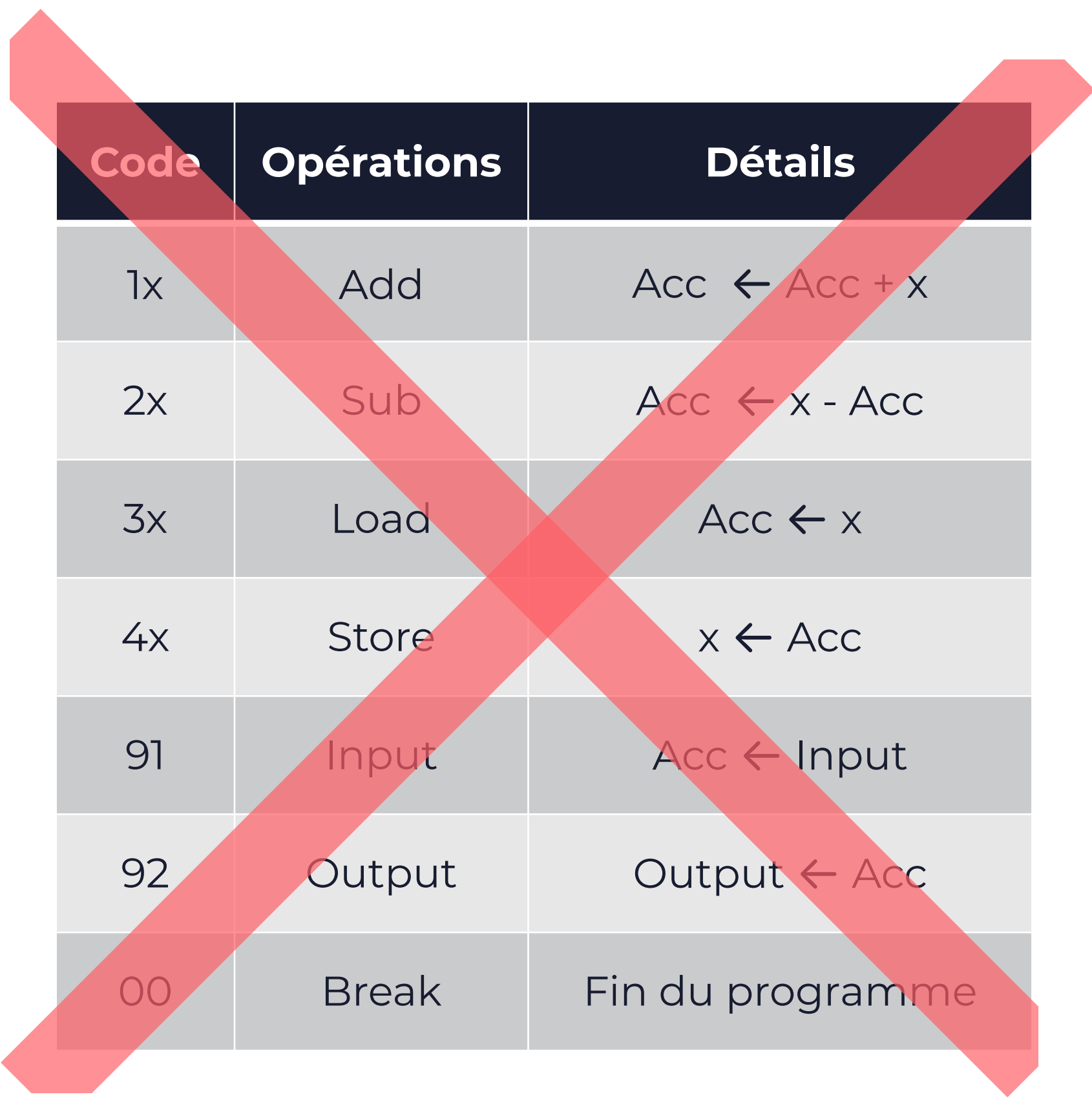
Mouvement de données (Load, Store)

Entrées / Sorties (Input, Output)

Contrôle (Break)

Modification du CPU trivial

Etape 3 : Modification du jeu d'instructions pour prendre en compte les registres



Code	Opérations	Détails
1x	Add	$Acc \leftarrow Acc + x$
2x	Sub	$Acc \leftarrow x - Acc$
3x	Load	$Acc \leftarrow x$
4x	Store	$x \leftarrow Acc$
91	Input	$Acc \leftarrow Input$
92	Output	$Output \leftarrow Acc$
00	Break	Fin du programme

Nouvelles opérations d'entrées-sorties :

- ➔ Input charge une donnée dans le registre x
- ➔ Output envoie le registre x sur la sortie

Nouvelles opérations de mouvements de données :

- ➔ Load charge la mémoire x dans le registre y
- ➔ Store écrit le registre x dans la mémoire y
- ➔ Nécessité de créer une instruction Mov pour faire des transferts entre registres

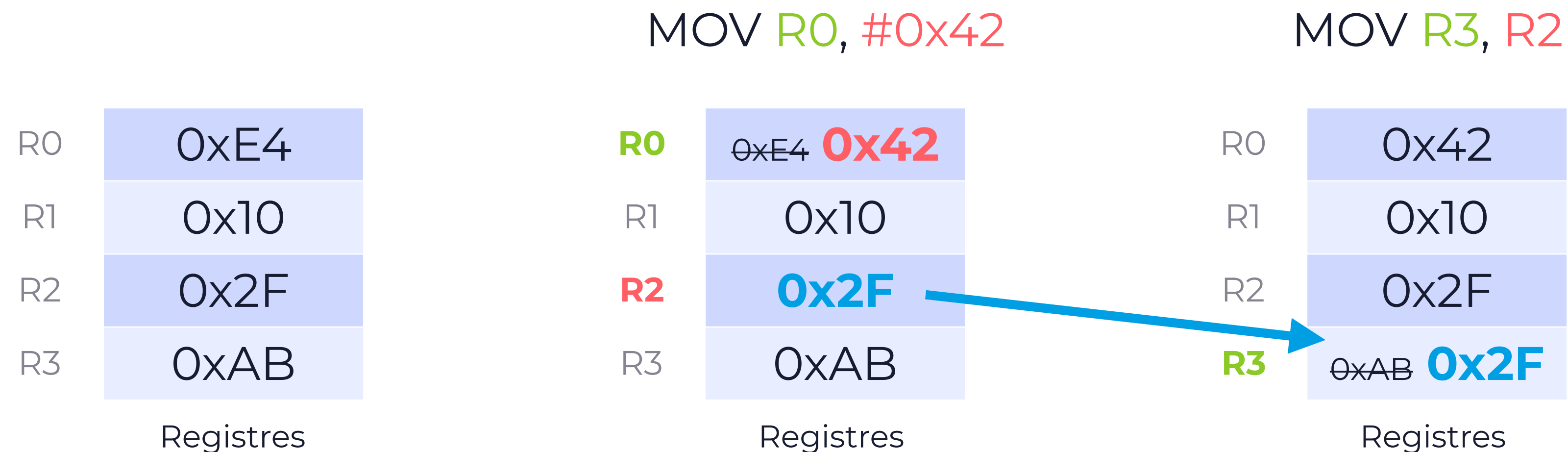
Nouvelles opérations arithmétiques :

- ➔ Travaillent uniquement sur les registres. Par ex, Add additionne le registre x avec le registre y et met le résultat dans le registre z

Instructions de **mouvement de données**

Proposition d'une instruction de **déplacement de données**

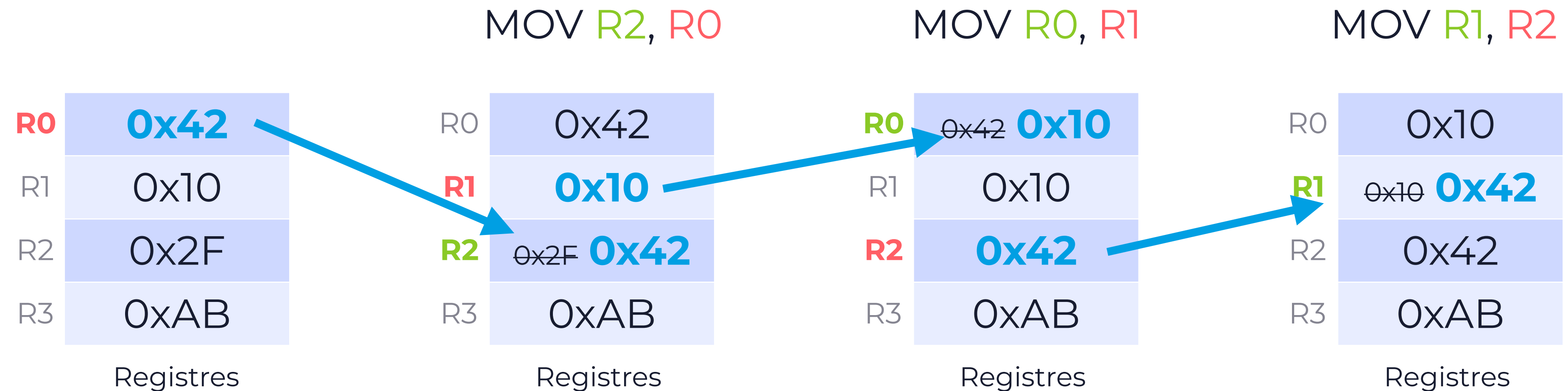
MOV **Destination** **Source**



Le chargement du registre se fait par **adressage direct** d'une constante ou d'un registre.

Exemple de **mouvement de données**

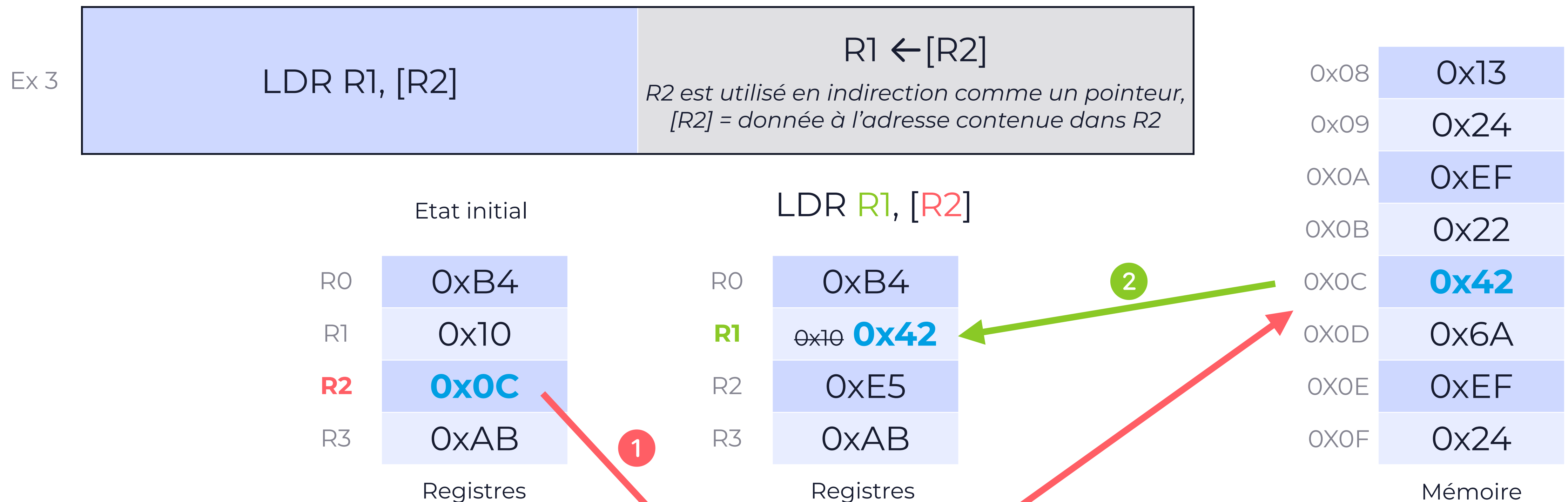
Permutation de données avec enchainement de
MOV Destination Source



Instructions de **mouvement de données**

Proposition d'une instruction **Lecture de la mémoire vers un registre**

LDR Destination Source

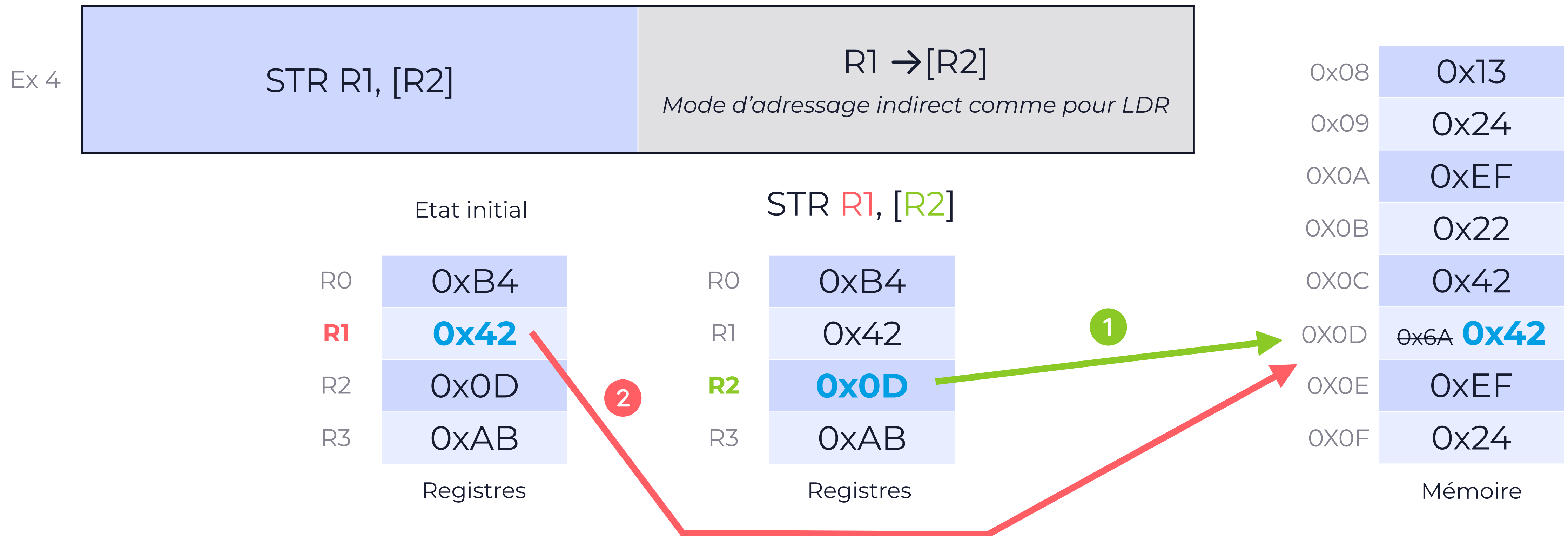


La lecture de la mémoire se fait par **adressage indirect** en utilisant un autre registre (base register) car LDR ne peut pas accéder directement à la mémoire.

Instructions de **mouvement de données**

Proposition d'une instruction **Stockage d'un registre vers la mémoire**

STR **Source** **Destination**

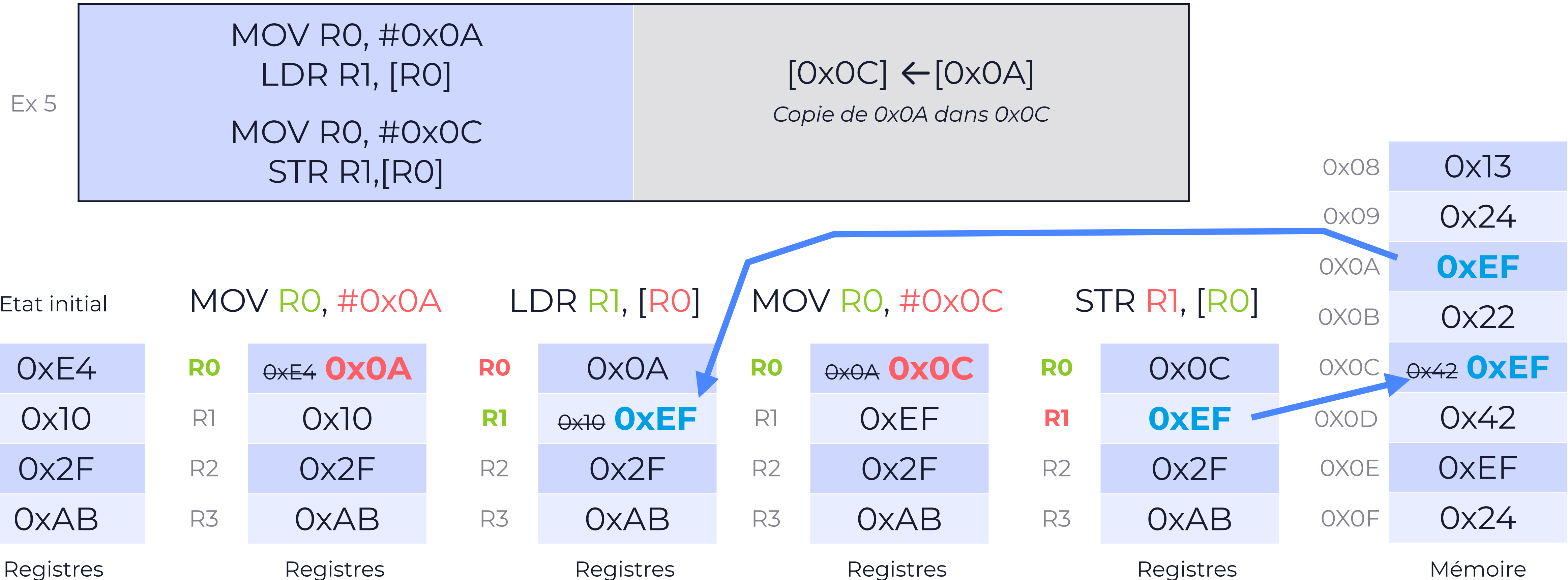


LDR copie la case mémoire stockée dans le 2^{ème} registre dans le 1^{er} registre : $R1 \leftarrow [R2]$

STR copie le 1^{er} registre dans la case mémoire représentée par le 2^{ème} registre : $R1 \rightarrow [R2]$

Exemple de **mouvements de données**

Recopie de données dans la mémoire avec MOV, LDR et STR



*Impossible de faire une copie directe. Obligation de passer par des registres.
Nécessite 2 sous opérations (lecture [0x0A] puis copie dans [0x0C])*

Instructions arithmétiques

Proposition d'une instruction **Addition de données**

ADD **Destination** **Source**

Ex 6

ADD R3, #0x11	R3 ← R3+0x11
ADD R1, R2	R1 ← R1 + R2

Etat initial

R0	0xE4
R1	0x12
R2	0x2F
R3	0xAB

Registres

ADD **R3**, #0x11

R0	0xE4
R1	0x12
R2	0x2F
R3	0xAB 0xBC

Registres

$$0xAB + 0x11 = 0xBC$$

ADD **R1**, **R2**

R0	0xE4
R1	0x12 0x41
R2	0x2F
R3	0xBC

Registres

$$0x2F + 0x12 = 0x41$$

Instructions arithmétiques

Proposition d'une instruction Soustraction de données

SUB Destination Source

Ex 7

SUB R3, #0x11	$R3 \leftarrow R3 - 0x11$
SUB R2, R1	$R2 \leftarrow R2 - R1$

Etat initial

R0	0xE4
R1	0x12
R2	0x2F
R3	0xAB

Registres

SUB R3, #0x11

R0	0xE4
R1	0x12
R2	0x2F
R3	0xAB 0x9A

Registres

$$0xAB - 0x11 = 0x9A$$

SUB R2, R1

R0	0xE4
R1	0x12
R2	0x12 0x1D
R3	0xBC

Registres

$$0x2F - 0x12 = 0x1D$$

Exemple d'instructions arithmétiques

Addition de données dans la mémoire avec MOV, LDR, ADD et STR

Ex 8

<pre>MOV R0, #0x0A LDR R1, [R0] MOV R0, #0x0B ; ou ADD R0, #0x01 LDR R2, [R0] ADD R2, R1 MOV R0, #0x0C ; ou ADD R0, #0x01 STR R2,[R0]</pre>	<p>$[0x0C] \leftarrow [0x0A] + [0x0B]$ <i>(Addition de 0x0A et 0x0B dans 0x0C)</i></p> <p><i>Pas possible de faire d'addition directe. Obligation de passer par des registres.</i></p> <p><i>4 sous opérations à réaliser : 2 lectures de 0x0A est 0x0B, Addition et Copie dans 0x0C</i></p> <p><i>Les instructions MOV peuvent être remplacées par des ADD si on connaît l'offset entre les cases mémoires</i></p>
--	---

0x08	0x13
0x09	0x24
0x0A	0xCF
0x0B	0x22
0x0C	0x08 0xF1
0x0D	0x6A
0x0E	0xEF
0x0F	0x24

Mémoire

MOV R0, #0x0A

LDR R1, [R0]

MOV R0, #0x0B

LDR R2, [R0]

ADD R2, R1

MOV R0, #0x0C

STR R2, [R0]

R0	0xE4 0x0A	0x0A	0x0A 0x0B	0x0B	0x0B	0x0B 0x0C	0x0C
R1	0x10	0x10 0xCF	0xCF	0xCF	0xCF	0xCF	0xCF
R2	0x2F	0x2F	0x2F	0x2F 0x22	0x22 0xF1	0xF1	0xF1
R3	0xAB	0xAB	0xAB	0xAB	0xAB	0xAB	0xAB

Registres

Registres

Registres

Registres

Registres

Registres

Registres

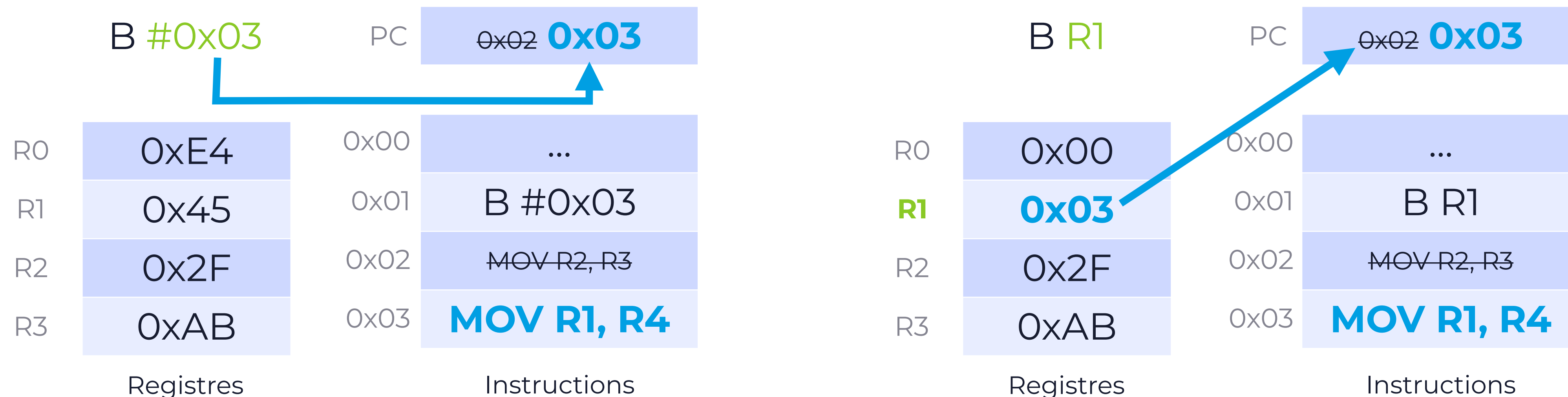
Instructions de **contrôle de programme**

Proposition d'une instruction **Branchement non conditionnel**
ou **Saut d'une adresse mémoire à une autre**

B **Destination**

Ex 9	B #0x03	PC ← 0x03
Ex 10	B R1	PC ← R1

PC est le registre "Program Counter" qui stocke l'adresse de la prochaine instruction à exécuter.



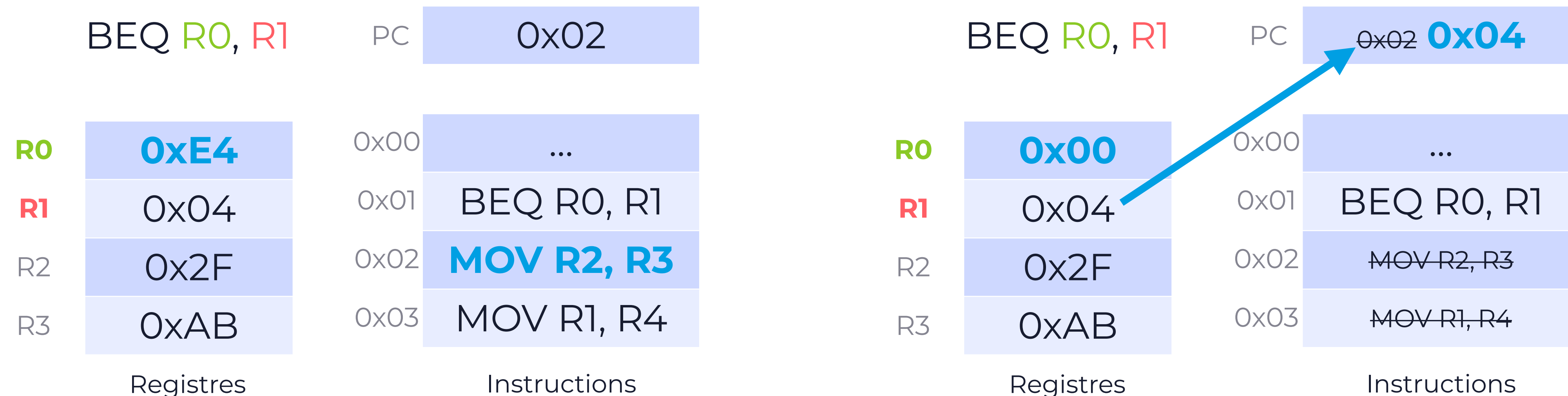
Dans les deux cas, lors de l'instruction B, PC passe de 0x02 à 0x03 et l'instruction MOV R2, R3 à l'adresse 0x02 n'est jamais exécutée.

Instructions de **contrôle de programme**

Proposition d'une instruction **Branchement conditionnel**
ou **Saut d'une adresse mémoire à une autre sous condition**

BEQ **Test Destination** (*instruction Branch if Equal to ZERO*)

Ex 11	BEQ R0, #0x04	SI R0 == 0, alors PC ← 0x04
Ex 12	BEQ R0, R1	SI R0 == 0, alors PC ← R1



R0 != 0 alors PC ← PC+1 et
MOV R2, R3 est exécuté

R0 = 0 alors PC ← 0x04 et les 2 instructions
suivantes ne sont pas exécutées

Exemple de contrôle de programme

Si le contenu de l'adresse mémoire 0x0A n'est pas égal à 0, alors placer la valeur 0x10 dans R1, sinon ne pas modifier R1.

Adresses	Instructions	Description
0x01	MOV R0, #0x0A	R0 ← 0x0A
0x02	LDR R1, [R0]	R1 ← [R0]
0x03	BEQ R1, #0x05	Si R1 == 0, alors on saute l'instruction suivante sinon on continue
0x04	MOV R1, #0x10	R1 ← 0x10
0x05	MOV R2, #0x20	R2 ← 0x20

Cas 1
0x00 dans 0x0A

MOV R0, #0x0A

R0	0xE4 0x0A
R1	0x23
R2	0x2F
R3	0xAB
PC	0x02

Registres

LDR R1, [R0]

R0	0x0A
R1	0x23 0x00
R2	0x2F
R3	0xAB
PC	0x03

Registres

BEQ R1, 0x05

R0	0x0A
R1	0x00
R2	0x2F
R3	0xAB
PC	0x04 0x05

Registres

MOV R2, #0x20

R0	0x0A
R1	0x00
R2	0x2F 0x20
R3	0xAB
PC	0x06

Registres

0x08	0x13
0x09	0x24
0x0A	0x00
0x0B	0x22
0x0C	0x42
0x0D	0x6A
0x0E	0xEF
0x0F	0x24

Mémoire

Exemple de contrôle de programme

Si le contenu de l'adresse mémoire 0x0A n'est pas égal à 0, alors placer la valeur 0x10 dans R1, sinon ne pas modifier R1.

Adresses	Instructions	Description
0x01	MOV R0, #0x0A	R0 ← 0x0A
0x02	LDR R1, [R0]	R1 ← [R0]
0x03	BEQ R1, #0x05	Si R1 == 0, alors on saute l'instruction suivante sinon on continue
0x04	MOV R1, #0x10	R1 ← 0x10
0x05	MOV R2, #0x20	R2 ← 0x20

Ex 14

Cas 2
0x45 dans 0x0A

MOV R0, #0x0A

R0	0xE4 0x0A
R1	0x23
R2	0x2F
R3	0xAB
PC	0x02

Registres

LDR R1, [R0]

R0	0x0A
R1	0x23 0x45
R2	0x2F
R3	0xAB
PC	0x03

Registres

BEQ R1, 0x05

R0	0x0A
R1	0x00
R2	0x2F
R3	0xAB
PC	0x04

Registres

MOV R1, #0x10

R0	0x0A
R1	0x00 0x10
R2	0x2F
R3	0xAB
PC	0x05

Registres

0x08	0x13
0x09	0x24
0x0A	0x45
0x0B	0x22
0x0C	0x42
0x0D	0x6A
0x0E	0xEF
0x0F	0x24

Mémoire

Codage des instructions

Une instruction est composée d'un ou plusieurs champs :

- ➔ Le premier nommé **code d'opération** ou “**opcode**” représente le type d'opération réalisée par l'instruction. Ex: LDR, STR, MOV, ADD, ...
- ➔ Les autres champs optionnels nommées “**adresses**” représentent les **paramètres** ou **opérandes** de l'instruction. Ex: registres, valeur constante, label, ...
- ➔ Les champs optionnels dépendent de l'opcode.

Les instructions en **code assembleur** (opcode + opérandes) sont la représentation exacte du langage machine dans un format compréhensible par le développeur.

La transformation du code assembleur en **code machine** pour un CPU donné est accomplie par un **programme assembleur**. Cela consiste à traduire les instructions assembleur en une **représentation binaire** codée sur un certain nombre de bits.



Ne pas confondre Code assembleur, Programme assembleur et Code machine.

Code assembleur
LDR, STR, MOV



Code machine
010001111100011

Codage des **instructions** du processeur trivial

Code	Opérations	Détails	Opcode	Operand
1x	Add	$\text{Acc} \leftarrow \text{Acc} + x$	0001	[1000, 1111]
2x	Sub	$\text{Acc} \leftarrow x - \text{Acc}$	0010	[1000, 1111]
3x	Load	$\text{Acc} \leftarrow x$	0011	[1000, 1111]
4x	Store	$x \leftarrow \text{Acc}$	0100	[1000, 1111]
91	Input	$\text{Acc} \leftarrow \text{Input}$	1001	0000
92	Output	$\text{Output} \leftarrow \text{Acc}$	1010	0000
00	Break	Fin du programme	0000	0000

Codage des instructions sur un **octet**

- ➔ Opcode sur les **4 bits de poids fort**
- ➔ Opérande sur les **4 bits de poids faible** correspondant à la mémoire données entre l'adresse 0x08 (0b1000) et 0x0F (0b1111)

Opcode	Opérande
4 bits	4 bits

Ex: Add 0xE → 0x1E → 0b0001 1110



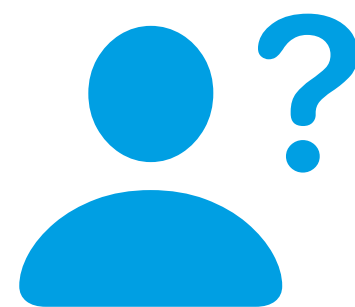
Avec un jeu d'instructions limité à 7 instructions, on pourrait coder l'opcode sur 3 bits seulement. Cela nécessiterait juste de changer l'opcode de Input et Output (71 et 72 par exemple au lieu de 91 et 92)

Codage des **instructions** du processeur amélioré avec 4 **registres**

Opérations	Détails
IN Rx	$Rx \leftarrow \text{Input}$
OUT Rx	$\text{Output} \leftarrow Rx$
MOV RX, #cst	$Rx \leftarrow \#cst$
MOV RX, RY	$Rx \leftarrow Ry$
LDR Rx, [Ry]	$Rx \leftarrow [Ry]$
STR Rx, [Ry]	$Rx \rightarrow [Ry]$
ADD Rx, #cst	$Rx \leftarrow Rx + \#cst$
ADD Rx, Ry	$Rx \leftarrow Rx + Ry$
SUB Rx, #cst	$Rx \leftarrow Rx - \#cst$
SUB Rx, Ry	$Rx \leftarrow Rx - Ry$
B #cst	$PC \leftarrow cst$
B Rx	$PC \leftarrow Rx$
BEQ Rx, #cst	SI $Rx == 0$, alors $PC \leftarrow cst$
BEQ Rx, Ry	SI $Rx == 0$, alors $PC \leftarrow Ry$
Break	Fin du programme

Jeu d'instructions comprenant **15 instructions** avec différentes combinaisons d'opérandes :

- ➔ Aucune opérande pour Break
- ➔ 1 registre pour IN, OUT, B
- ➔ 1 constante pour B
- ➔ 2 registres pour MOV, LDR, STR, ADD, SUB et BEQ
- ➔ 1 registre et une constante pour certaines versions de MOV, ADD, SUB et BEQ



Peut on conserver le **codage des instructions sur **un seul octet** ?**

Codage des **instructions** du processeur amélioré avec 4 **registres**

15 instructions différentes

- ➔ Codage de l'opcode sur **4 bits** au minimum (ISA à 16 instructions au maximum), voir **5 bits** pour anticiper l'ajout de nouvelles instructions

2 types différents d'opérandes

- ➔ Registre R0 à R3 : possibilité de codage de chaque registre sur **2 bits**
- ➔ Une valeur constante codée sur **x bits** (x étant à déterminer)

4 combinaisons d'opérandes :

- ➔ Aucune opérande : nécessite **0 bit**
- ➔ 1 registre : nécessite **2 bits**
- ➔ 1 constante : nécessite **x bits**
- ➔ 2 registres : nécessite **2+2 = 4 bits**
- ➔ 1 registre et 1 constante : nécessite **2+x bits**

$$4 \leq \#bits \leq 6+x$$



*Pour conserver le **codage** des instructions sur **un seul octet**, il faudrait choisir **x = 2 bits** ce qui est trop limitant.*

Codage des **instructions** du processeur amélioré avec 4 **registres**

①

Le CPU utilise des mots de 8 bits, donc $x = 8$ bits

②

$$4 \leq \#bits \leq 6+x$$
$$4 \leq \#bits \leq 6+x = 14$$

③

Choix d'un codage des instructions sur 16 bits

Opcode	Param 1.	Param 2.
4, 5 ou 6 bits	4, 3 ou 2 bits	8 bits



Param 1 sur 2, 3 ou 4 bits = Registre (nécessite en réalité 2 bits si 4 registres)

Param 2 sur 8 bits = Registre (avec bits de poids fort à 0) ou constante (sur 8 bits)

Exemples de Codage d'instructions sur 16 bits du processeur amélioré

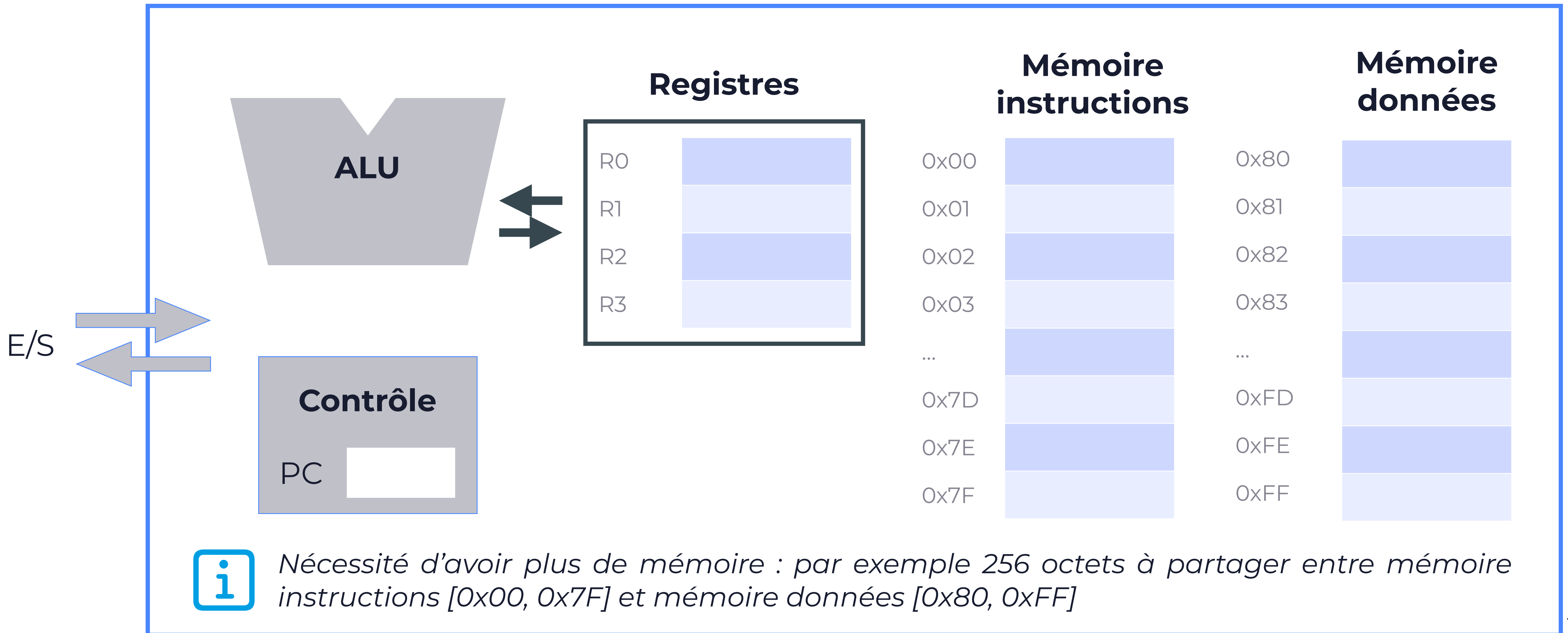
Instruction	Opcode (5 bits)	Param 1 (3 bits)	Param2 (8 bits)	Code
MOV R1, #0x42	00010	001	0100 0010	0b0001000101000010 = 0x1142
MOV R3, R2	00011	011	0000 0010	0b0001101100000010 = 0x1b02
LDR R3, [R2]	00100	011	0000 0010	0b0010001100000010 = 0x2302
ADD R3, #0x02	01000	011	0000 0010	0b0100001100000010 = 0x4302
ADD R3, R2	01001	011	0000 0010	0b0100101100000010 = 0x4b02
B #0x23	01100	111	0010 0011	0b0110011100100011 = 0x6723



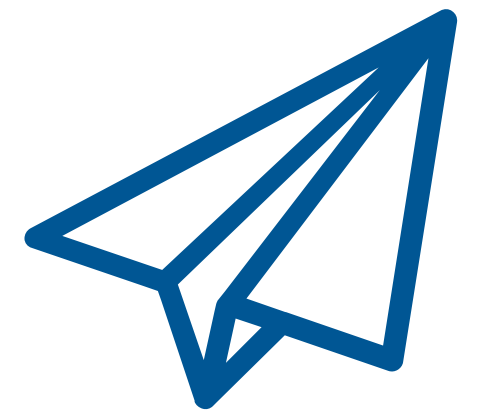
C'est l'opcode qui détermine le sens des valeurs stockées dans les différents paramètres, en particulier pour le param 2 qui peut contenir un registre ou une constante.

Modification du CPU trivial

Etape 4 : Ajout de mémoire pour prendre en compte les instructions 16 bits



TAKE HOME MESSAGE



Le CPU est piloté par une **unité de contrôle** chargée d'enchaîner des cycles **Lecture, Décodage, Exécution** des instructions.

Les instructions possibles constituent le **jeu d'instructions** de type RISC (Reduced) ou CISC (Complex).

Les **calculs** (MAC) sont réalisés par une **ALU**.

Questions





Contacts

Pr. Dominique Ginhac

@ dginhac@u-bourgogne.fr

Retrouvez toutes les infos sur :

 <https://github.com/dginhac/esirem-archi>



This work is **licensed** under a
Creative Commons Attribution-NonCommercial 4.0 International License.

<https://creativecommons.org/licenses/by-nc/4.0/>

