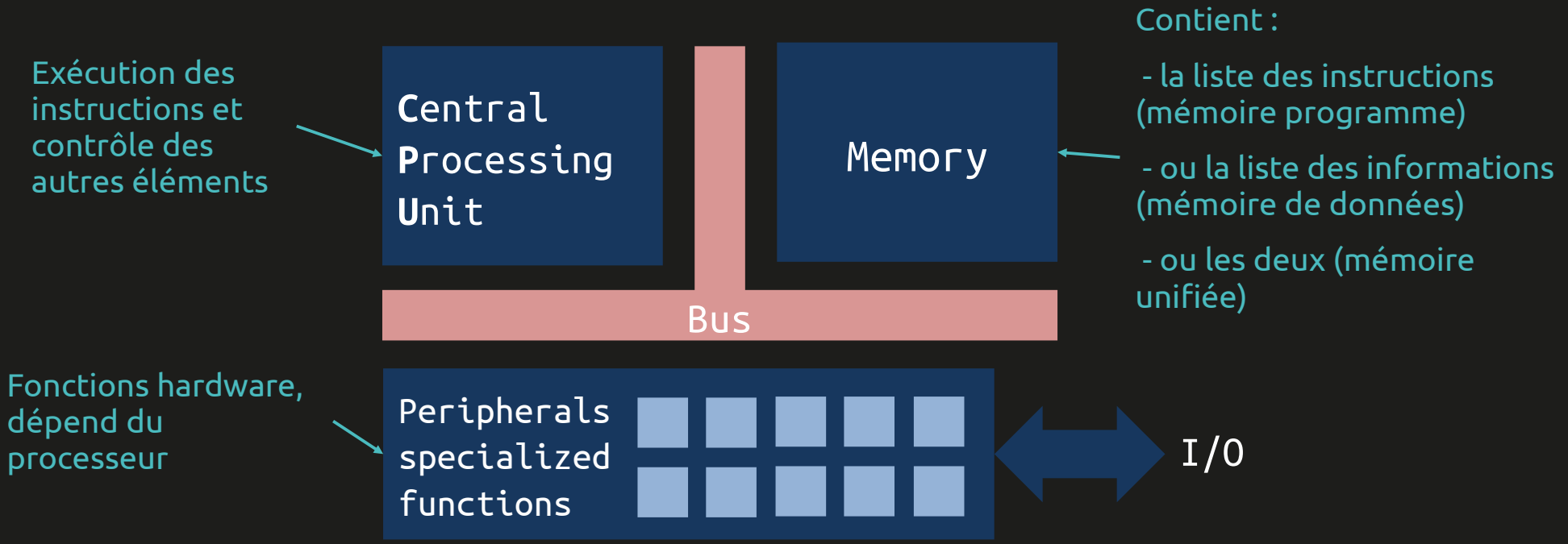
Chapitre 2

Le Processeur

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Quel que soit son type (MCU, DSP, ...), un processeur à CPU peut être décrit avec le schéma suivant.



Tous les processeurs modernes utilisent un CPU ou un ensemble de CPU, dont les fonctionnalités dépendent de la famille du CPU.

La mémoire peut être interne (dans le processeur) ou externe (CI séparé). Différents usages de la mémoire existent : travail direct avec le CPU (*main memory*) ou stockage des informations sur le long terme (*mass storage*).

Les périphériques dépendent aussi de l'architecture du processeur. Pour l'instant, disons simplement que les périphériques participent à l'interaction du processeur avec son environnement.

Différentes associations CPU/mémoire/périphériques mènent à des architectures différentes. Les architectures les plus courantes sont décrites dans le chapitre suivant.

CENTRAL PROCESSING UNIT

Control Unit

Processing Unit (ALU)

Register file

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Le **CPU (*Central Processing Unit*, Unité Centrale de Traitement)** est le cerveau des processeurs modernes, des MCU basse-consommation aux GPU hautes performances.

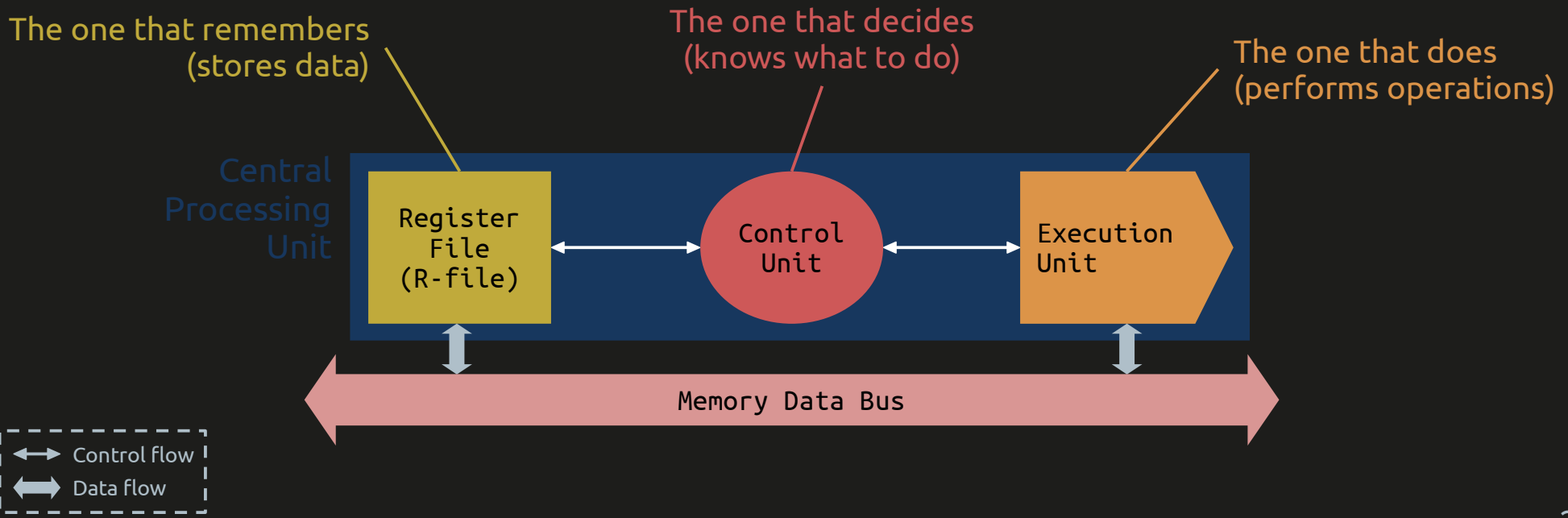
Le rôle du CPU est de **contrôler le flux d'informations dans le processeur**.

Pour cela, il contrôle les bus de données et bus d'adresses.

Il contrôle aussi de nombreux fils/signaux, ce qui lui donne un contrôle indirect vers toutes les autres fonctionnalités matérielles.

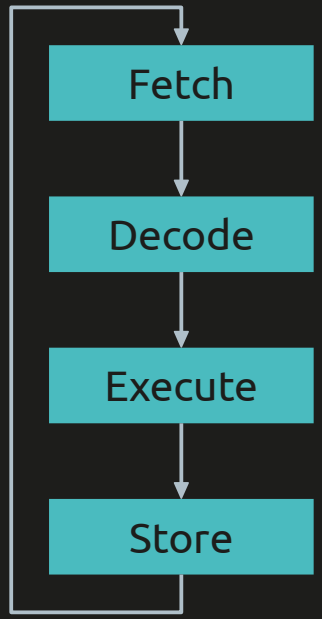
Ce sont les instructions, stockées sous forme binaire, qui indiquent au CPU les opérations à effectuer et les données à manipuler.

Un CPU possède au moins une unité de contrôle (*Control Unit*), une unité d'éxécution (Execution Unit) et une file/banque de registres (Register file).



Le CPU lit les instructions du programme de manière séquentielle.

L'**unité de contrôle** se charge d'exécuter ce flux d'instructions, selon ce cycle :



Va chercher (lire) la prochaine instruction depuis la mémoire programme

Détecte l'opération à réaliser grâce au code binaire opératoire (opcode)

Réalise l'opération, généralement via l'une des unités d'exécution

Stocke le résultat dans les registres CPU ou directement en mémoire data

Les **unités d'exécution** du CPU se doivent de réaliser la plupart des opérations.

Selon la famille du CPU, les différentes unités de traitement peuvent être :

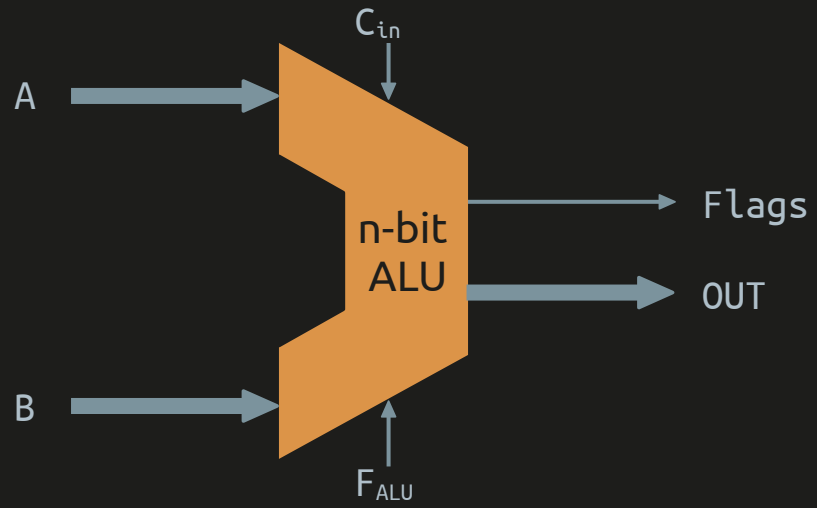
- Une unité arithmétique et logique (ALU, *Arithmetic and Logic Unit*)
  - Opérations logiques et arithmétiques élémentaires
- Une unité de calcul en virgule flottante (FPU, *Floating-Point Unit*)
  - Pour les processeurs performants
- Un multiplieur
- Un registre à décalage
- …

L' ***Arithmetic and Logic Unit*** **(ALU)** est l'unité de traitement universelle.

Sur cet exemple, les données à traiter sont les entrées A et B.

Le choix de l'opération est fourni par l'unité de contrôle sur les bits F_{ALU} (grâce à l'étage DECODE).

Le résultat arrive sur la sortie OUT, tandis que les flags (S, Z, C, O, ...) sont mis à jour en fonction du résultat. L'unité de contrôle lira ces flags pour adapter les instructions à exécuter (instructions conditionnelles : if, while, for, ...)




```
Opérations :
 AND, OR, XOR, NOT,
 ADD, SUB, INC,
 CMP, ...

Flags :
 S - Signed
 Z - Zero
 C - Carry
 O - Overflow
 ...
```

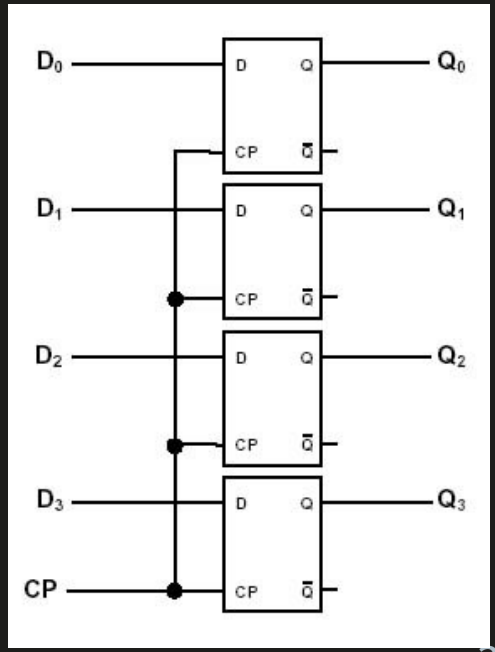
La **banque de registres (*Register file*)** contient, vous l'aurez deviné, des registres.

Les registres sont de petits emplacements mémoire situés au coeur du CPU : ils sont extrêmement rapides, mais ne contiennent que très peu de données.

Certains sont des *general purpose registers* (ou registres de travail), et peuvent contenir n'importe quelle donnée.

D'autres sont des registres spécifiques, utilisés uniquement dans un but précis.

Par exemple le *Status Register* contient des *flags*, le *Program Counter register* contient l'adresse de la prochaine instruction à exécuter, ...



Banque de registres

Exemple :

Banque de registres du MCU SMP430 de Texas Instruments

R0/PC – Program Counter

Contient l'adresse de la prochaine instruction

R1/SP – Stack Pointer

Utilisé pour le contexte de la fonction courante

R2/SR – Status Register

Contient les *flags* issus de l'ALU, il est lu par l'unité de contrôle

R4 to R15 – General Purpose registers

Peut contenir n'importe quoi, utilisé pour stocker les données à traiter

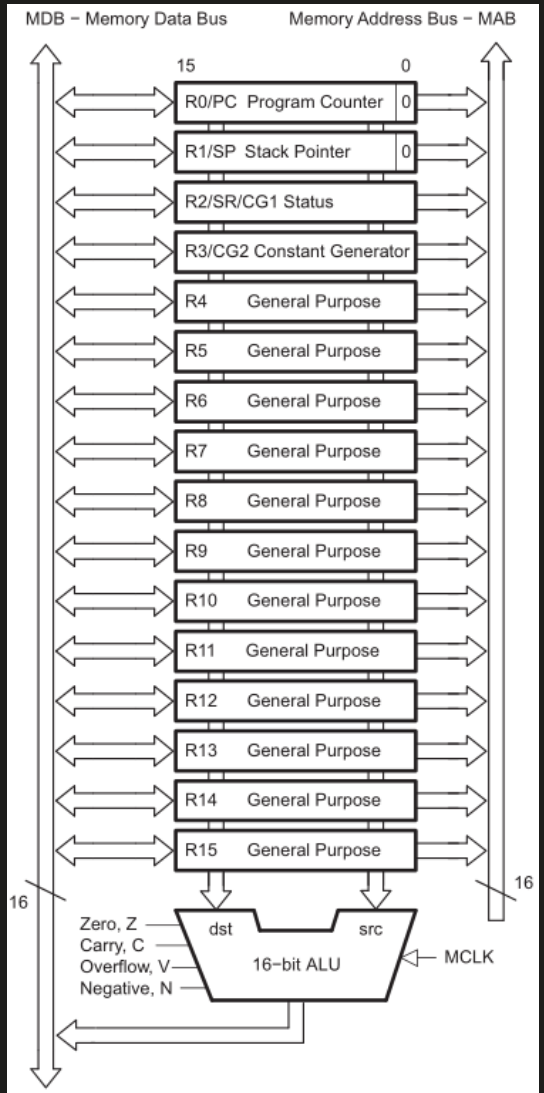


Figure 3-1. CPU Block Diagram

Exemple :

Registre de statut du MSP430 de Texas Instruments

Figure 3-6. Status Register Bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C
rw-0							rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

V: Overflow bit — Passe à ‘1’ quand le résultat d’une opération arithmétique déborde, passe à ‘0’ sinon.

N: Negative bit — Passe à ‘1’ quand le résultat d’une opération est négatif.

Z: Zero bit — Passe à ‘1’ quand le résultat d’une opération est nul.

C: Carry bit — Passe à ‘1’ quand le résultat d’une opération provoque une retenue.

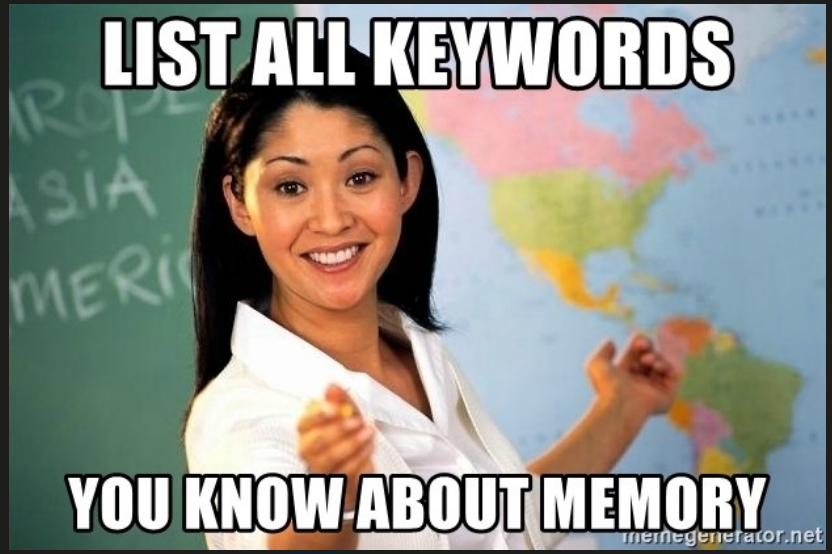
MÉMOIRE

Mémoire volatile
Mémoire rémanente

Mémoire de travail
Stockage de masse

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Brainstorming



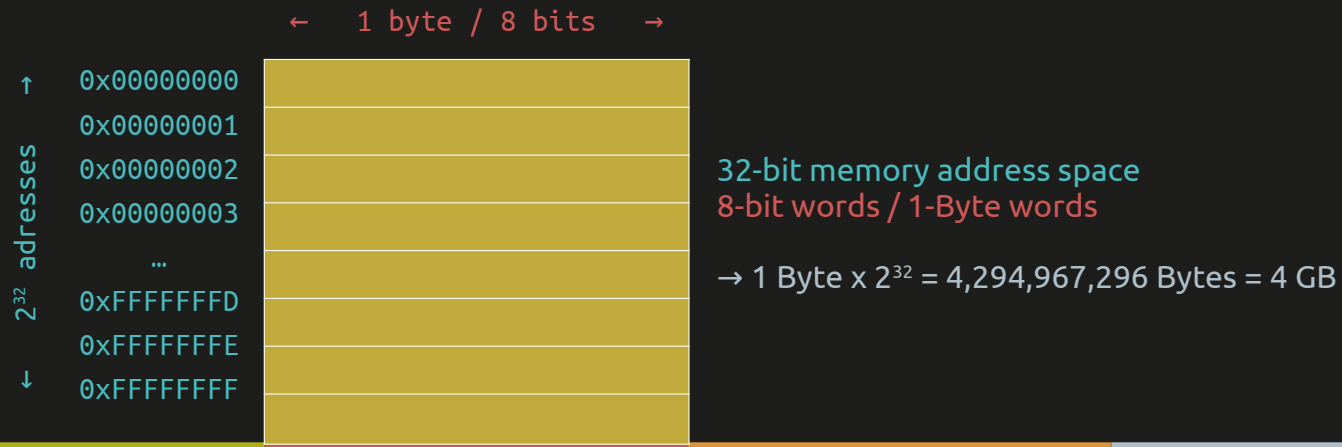
Mémoire adressable par octet

La mémoire est un composant stockant de l'information (données et/ou instructions).

Les usages les plus courants sont la mémoire volatile (qui travaille avec le processeur) et la mémoire de stockage de masse (stocke les informations qui ne sont pas traitées).

Les mémoires sont adressables par octet (qui est donc l'unité de taille).

NB : Ce n'est pas vrai pour la mémoire cache (construire dans le CPU) ou les stockages de masse qui emploient des systèmes de fichiers (ext4, FAT32, NTFS, …)



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Soyons clairs !

Une fois éteinte, une mémoire volatile perdra ses données tandis qu'une mémoire rémanente les conservera.

ROM (*Read-Only Memory*) est une technologie obsolète pour laquelle la mémoire ne pouvait être écrite qu'une fois. Elle a été remplacée par PROM (*Programmable ROM*), plus particulièrement UVPROM (*Ultra-Violet PROM*, obsolète) et EEPROM (*Electrically Erasable PROM*).

Quand on parle de ROM aujourd'hui, on désigne en réalité une EEPROM, et plus précisément une Flash.

RAM (*Random Access Memory*) est une technologie de mémoire volatile. "*Random Access*" signifie qu'on peut accéder à n'importe quelle adresse (*random*) en un temps constant..

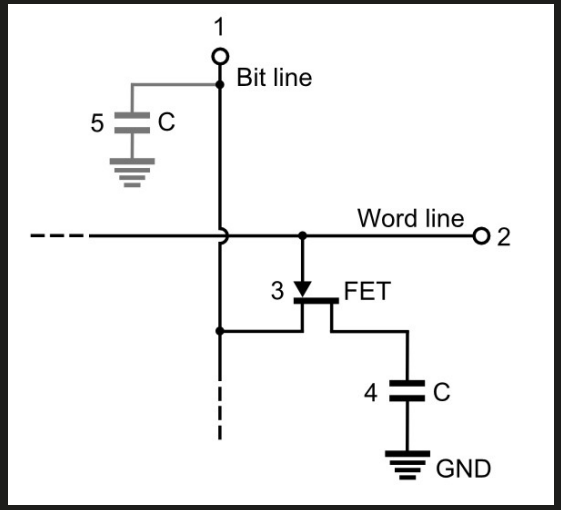
Une mémoire de stockage de masse est une mémoire rémanente qui conserve les données même hors tension.

La mémoire de travail (*main memory*) est volatile mais très rapide. Le processeur l'utilise pour stocker les données en cours de traitement.

Deux technos de mémoire volatile : DRAM (*Dynamic RAM*) et SRAM (*Static RAM*).

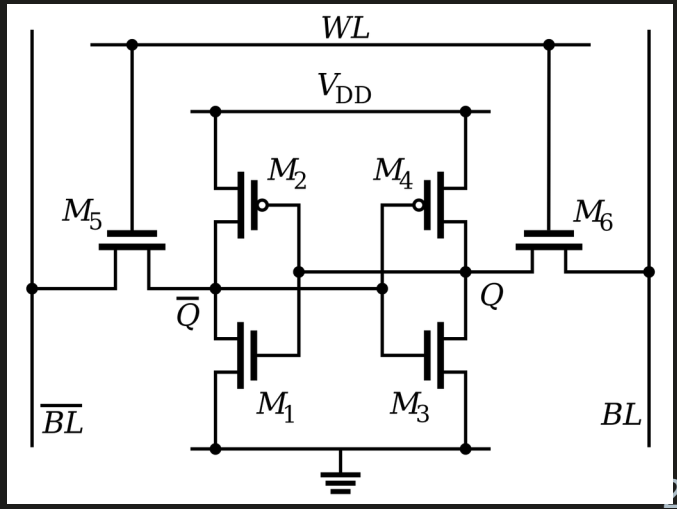
La DRAM nécessite d'être régulièrement rafraîchie à cause de ses pico-condensateurs. Utilisée pour la mémoire d'ordinateur. Faible empreinte silicium mais plus lente que la SRAM. Les technologies actuelles sont la DDR4 SDRAM (*4th generation of Double Data Rate Synchronous DRAM*).

La SRAM se base sur des bascules. Utilisée pour les registres et mémoires cache L1/L2/L3. Bien plus rapide mais très grande empreinte silicium.



DRAM
1 bit requires 1 transistor
and 1 pico-capacitor

SRAM
1 bit requires 6
CMOS transistors

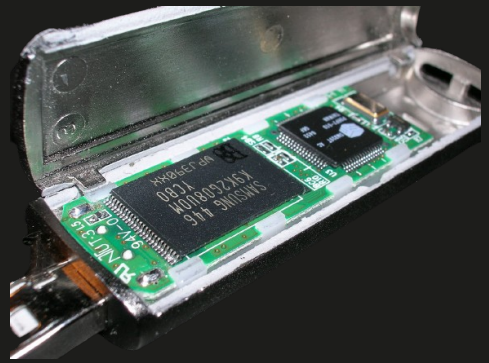


Le stockage de masse a été effectué par plusieurs technologies :

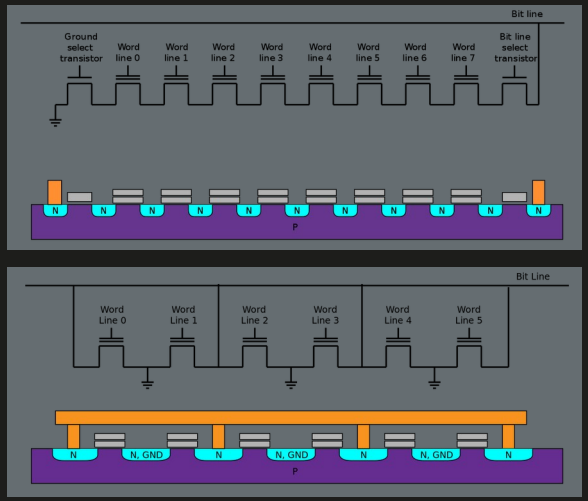
Les disquettes et disques durs (HDD, *Hard Drive Disk*) utilisent le stockage magnétique.

Les EEPROM utilisent de la circuiterie basique pour stocker des charges électriques. La technologie EEPROM la plus commune est la mémoire Flash (NAND et NOR), qui a un temps constant d'accès à l'information. Les SSD (*Solid-State Drive*) utilisent aussi une technologie Flash.

Hard Drive Disk (HDD)

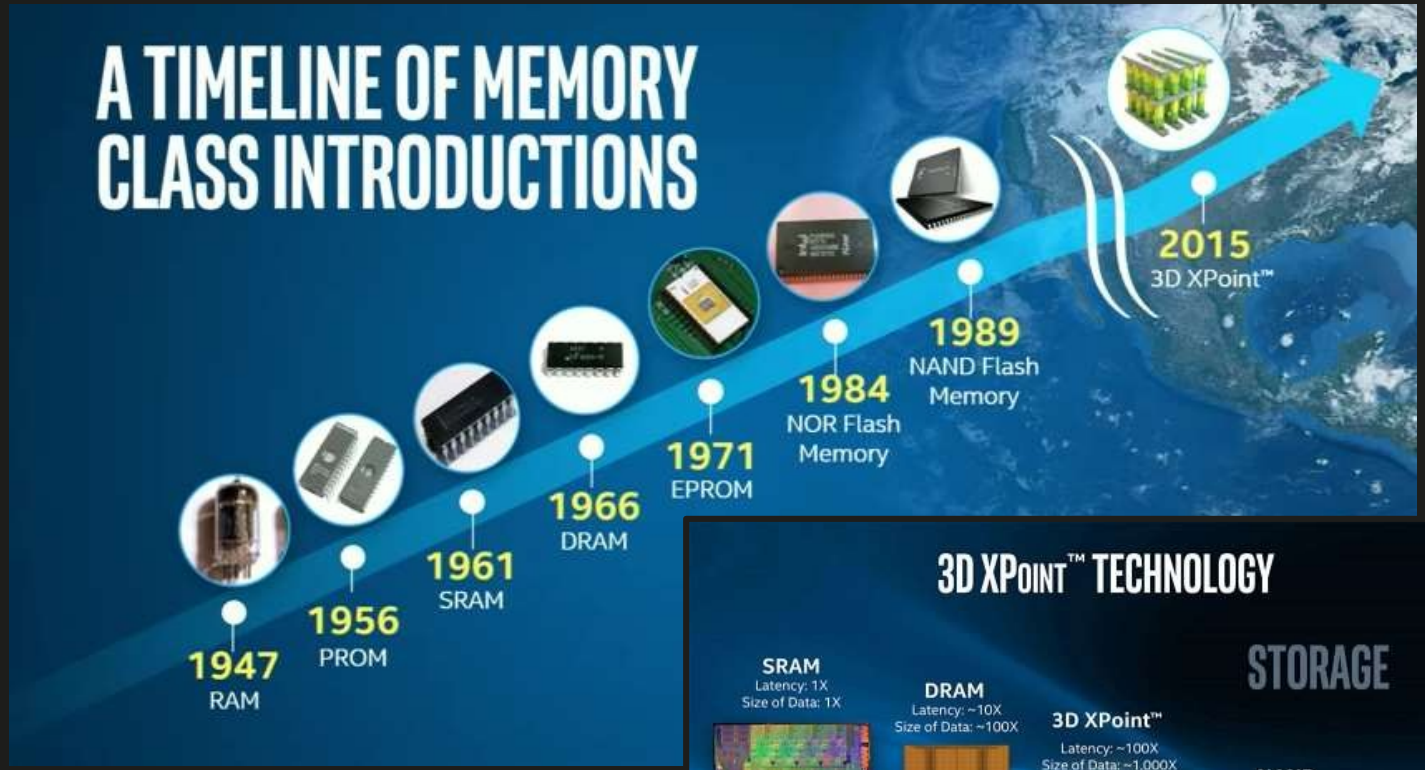


Flash drive / clé USB
(Mémoire à gauche, MCU à droite)



NAND (top) and NOR (bottom)
Flash memory structures

Évolution


A TIMELINE OF MEMORY CLASS INTRODUCTIONS
1947
RAM
1956
PROM
1961
SRAM
1966
DRAM
1971
EPROM
1984
NOR Flash Memory
1989
NAND Flash Memory
2015
3D XPoint™
3D XPOINT™ TECHNOLOGY
STORAGE
MEMORY
SRAM
Latency: 1X
Size of Data: 1X
DRAM
Latency: ~10X
Size of Data: ~100X
3D XPoint™
Latency: ~100X
Size of Data: ~1,000X
NAND
Latency: ~100,000X
Size of Data: ~1,000X
HDD
Latency: ~10 MillionX
Size of Data: ~10,000 X

EXÉCUTION D'UN PROGRAMME

Du fichier C à l'exécutable binaire

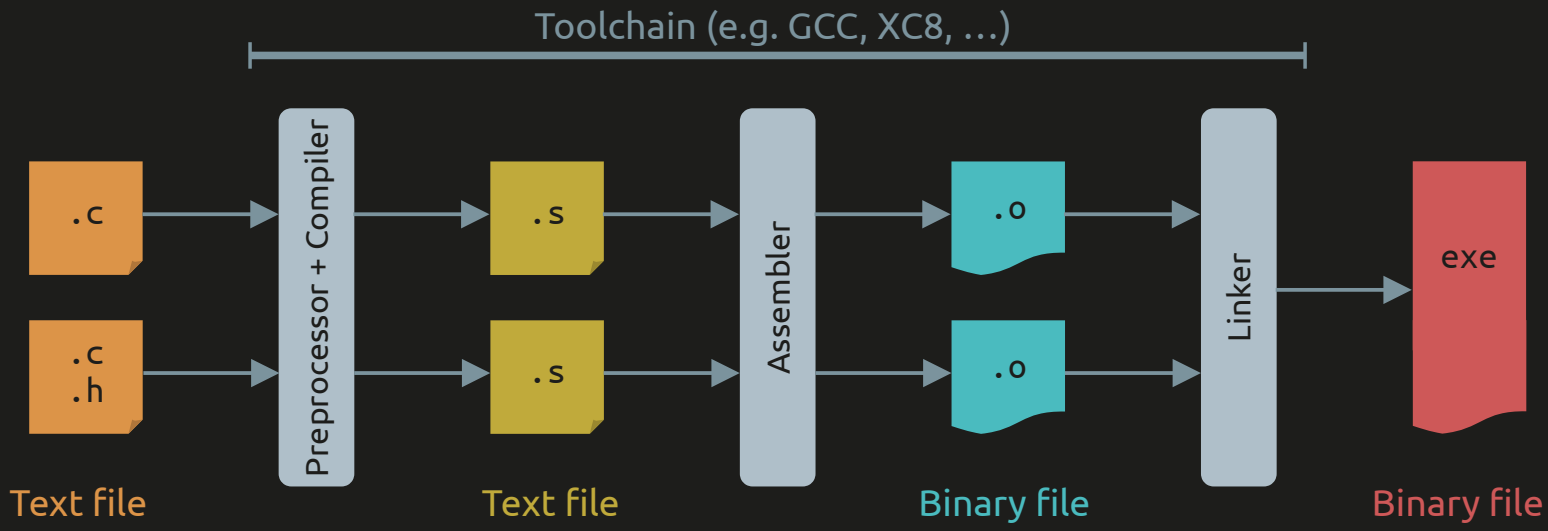
Exécution sur un processeur maison

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Faisons simple, vous verrez la *toolchain* en détails en « Architecture des ordinateurs ».

La **toolchain (*chaîne de compilation*)** est la suite logicielle qui « convertit » votre ficher source C en un fichier binaire exécutable.

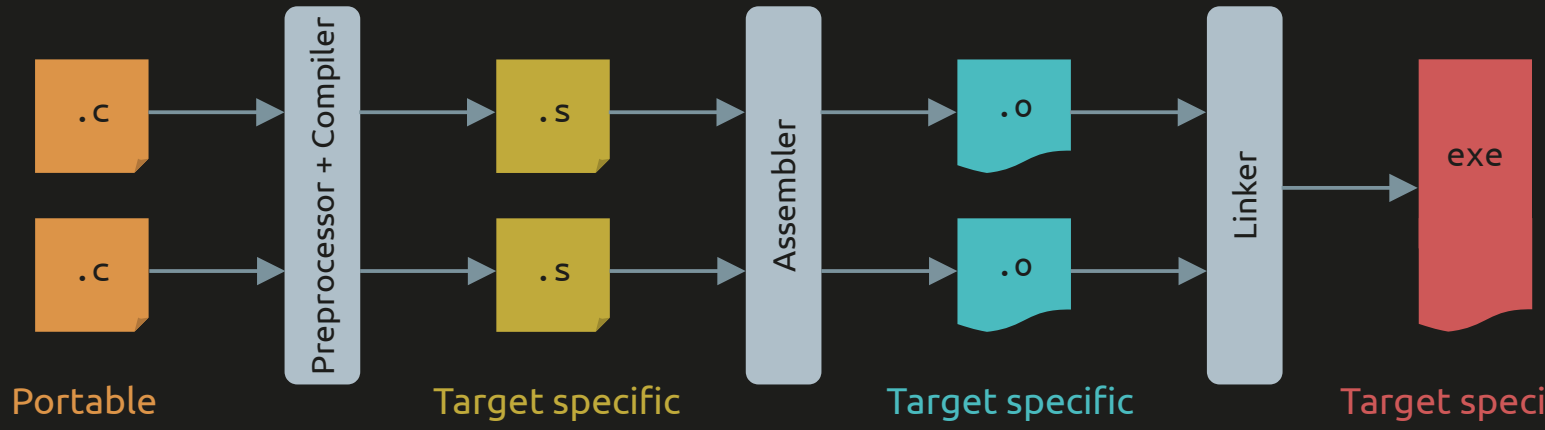


Pourquoi utiliser une *toolchain* ?

Le langage C est portable : il peut être exécuté sur n'importe quel processeur cible.

Mais en réalité le processeur ne comprend que son propre jeu d'instructions. C'est le contraire même de la portabilité : seul le processeur peut exécuter ses instructions.

La toolchain permet d'écrire un programme unique dans une langue universelle (langage portable) pour ensuite créer un exécutable pour chaque processeur cible.



From the C file to an executable program

Example of an executable program for a x64-architecture processor, from C to binary.

C language program

```
char inc(char bar);

int main(void){
    char foo;
    foo = inc(1);
    return 0;
}

char inc(char bar) {
    return bar+1;
}
```

Assembly language program

```
Instructions    Operands

main:
push    %rbp
mov     %rsp,  %rbp
sub     $0x10, %rsp
mov     $0x1,  %edi
call    4004f2 <inc>
mov     %al,   %-0x1(%rbp)
mov     %0x0,  %eax
leave
ret

inc:
push    %rbp
mov     %rsp,  %rbp
mov     %edi,  %eax
mov     %al,   -0x4(%rbp)
movzbl  -0x4(%rbp), %eax
add     $0x1,  %eax
pop     %rbp
ret
```

Binary program

```
Program memory address    Binary instructions

00000000004004d6 <main>:
4004d6:     55
4004d7:     48 89 e5
4004da:     48 83 ec 10
4004de:     bf 01 00 00 00
4004e3:     e8 0a 00 00 00
4004e8:     88 45 ff
4004eb:     b8 00 00 00 00
4004f0:     c9
4004f1:     c3

00000000004004f2 <inc>:
4004f2:     55
4004f3:     48 89 e5
4004f6:     89 f8
4004f8:     88 45 fc
4004fb:     0f b6 45 fc
4004ff:     83 c0 01
400502:     5d
400503:     c3
```

Processeur maison

Créons notre processeur maison.

C'est un CPU élémentaire, RISC-like (*Reduced Instruction Set Computer*).

Son jeu d'instruction (ISA, Instruction Set Architecture) est purement fictif.

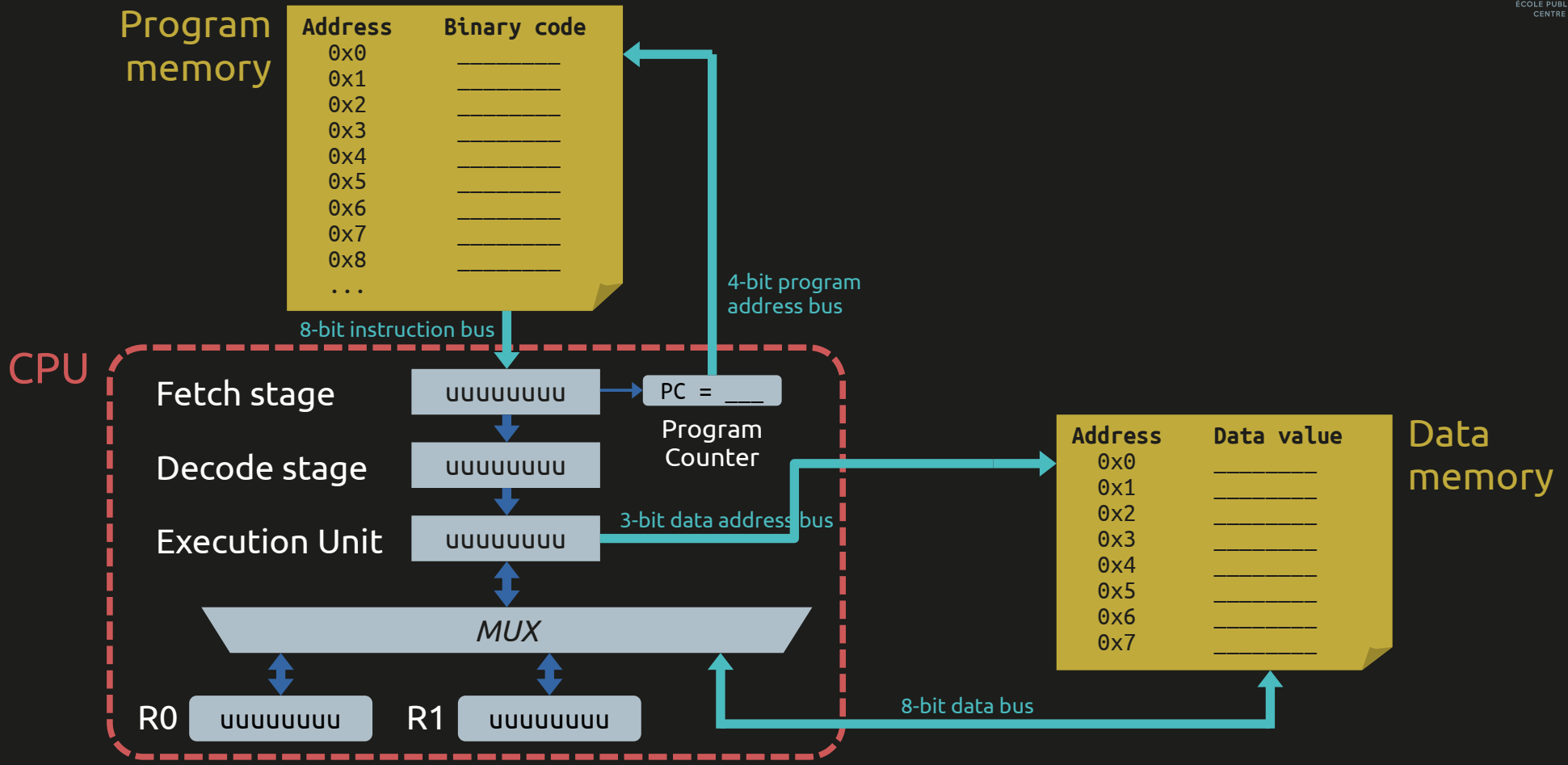
Operation	Syntax	Description	Example	Binary Opcode
ADD	`ADD srcReg, srcReg, dstReg`	Add two register values	`ADD R0, R1, R0`	`000 r r r uu`
JMP	`JMP label`	Program memory jump	`JMP main`	`001 aaaa u`
LOAD	`LOAD address, dstReg`	Load data memory value to register	`LOAD var1, R1`	`010 aaa r u`
MOV	`MOV srcReg, dstReg`	Copy register value to another register	`MOV R1, R0`	`011 r r uuu`
MOVK	`MOVK constant, dstReg`	Copy 3-bit constant value into register	`MOV 5,  R1`	`100 kkk r u`
STR	`STR srcReg, address`	Store register value to data memory	`STR R1, var1`	`101 r aaa u`

r = register bit
r=0 → Select R0
r=1 → Select R1

a = address bits
k = constant value
u = bit unused

EXÉCUTION D'UN PROGRAMME

Processeur maison



Processeur maison

Maintenant :

1. traduisons ce programme C en assembleur,
2. puis obtenons le contenu de sa mémoire programme.

```c
char value = 3; // Stored at 0x0
char saveValue; // Stored at 0x1

void main(void) {
    while(1) {
        value += 2;
        saveValue = value;
    }
}
```

} Data memory map

Operation	Syntax	Description	Example	Binary Opcode
ADD	ADD srcReg, srcReg, dstReg	Add two register values	ADD R0, R1, R0	000 r r r uu
JMP	JMP label	Program memory jump	JMP main	001 aaaa u
LOAD	LOAD address, dstReg	Load data memory value to register	LOAD var1, R1	010 aaa r u
MOV	MOV srcReg, dstReg	Copy register value to another register	MOV R1, R0	011 r r uuu
MOVK	MOVK constant, dstReg	Copy 3-bit constant value into register	MOV 5, R1	100 kkk r u
STR	STR srcReg, address	Store register value to data memory	STR R1, var1	101 r aaa u

r = register bit
r = 0 → Select R0
r = 1 → Select R1

a = address bits
k = constant value
u = bit unused

Processeur maison

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Solution

C language program

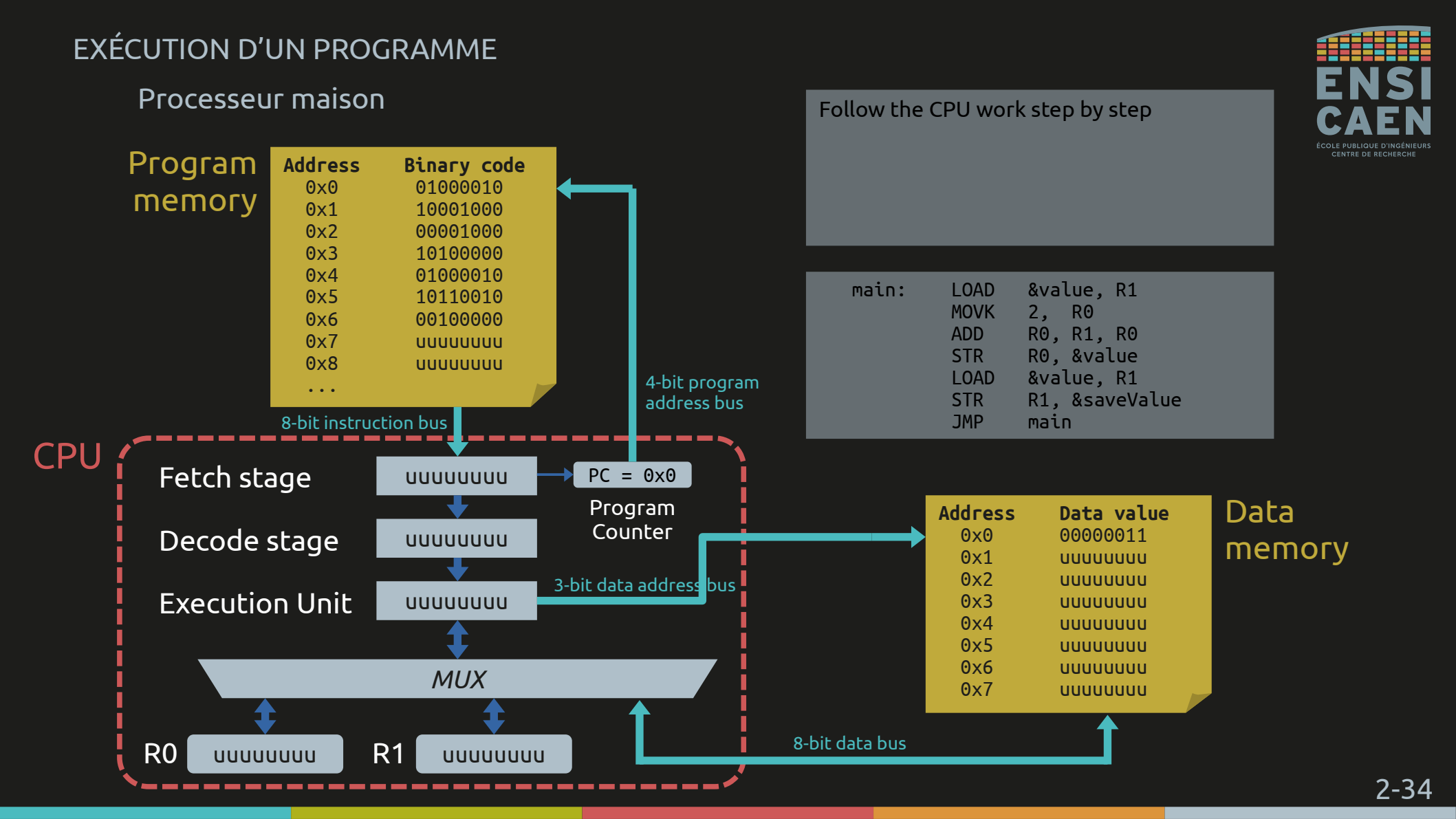
```c
char value = 3; // Stored at 0x0
char saveValue; // Stored at 0x1

void main(void) {
    while(1) {
        value += 2;
        saveValue = value;
    }
}
```

Assembly language program

Instruction address	Instruction = Operation + Operands		Binary
0x0 main:	LOAD	&value, R1	01000010
0x1	MOVK	2, R0	10001000
0x2	ADD	R0, R1, R0	00001000
0x3	STR	R0, &value	10100000
0x4	LOAD	&value, R1	01000010
0x5	STR	R1, &saveValue	10110010
0x6	JMP	main	00100000
0x7	undef		uuuuuuuu
0x8	undef		uuuuuuuu
0x...	...		...
0xF	undef		uuuuuuuu

EXÉCUTION D'UN PROGRAMME

Processeur maison

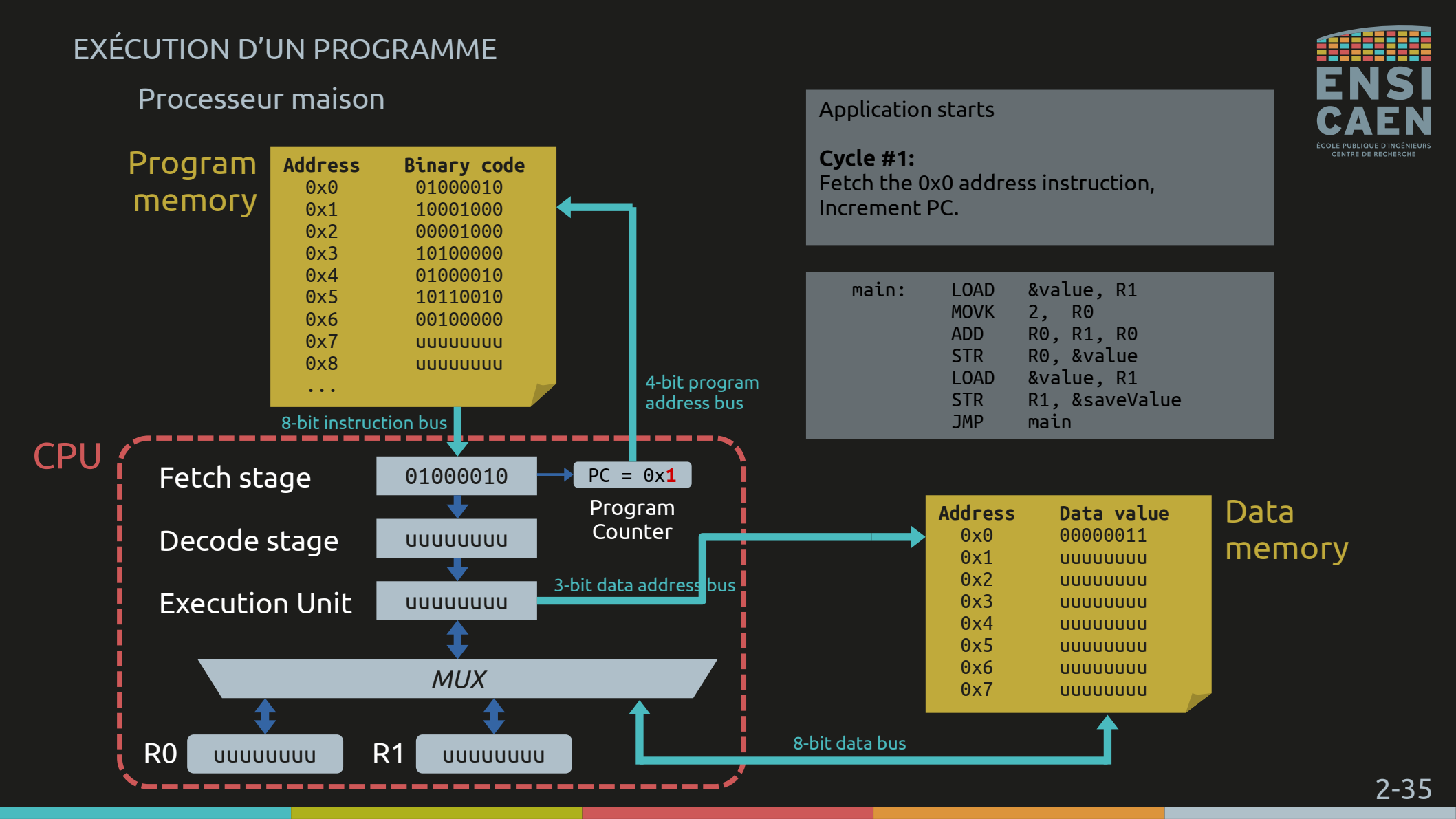


Follow the CPU work step by step

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison



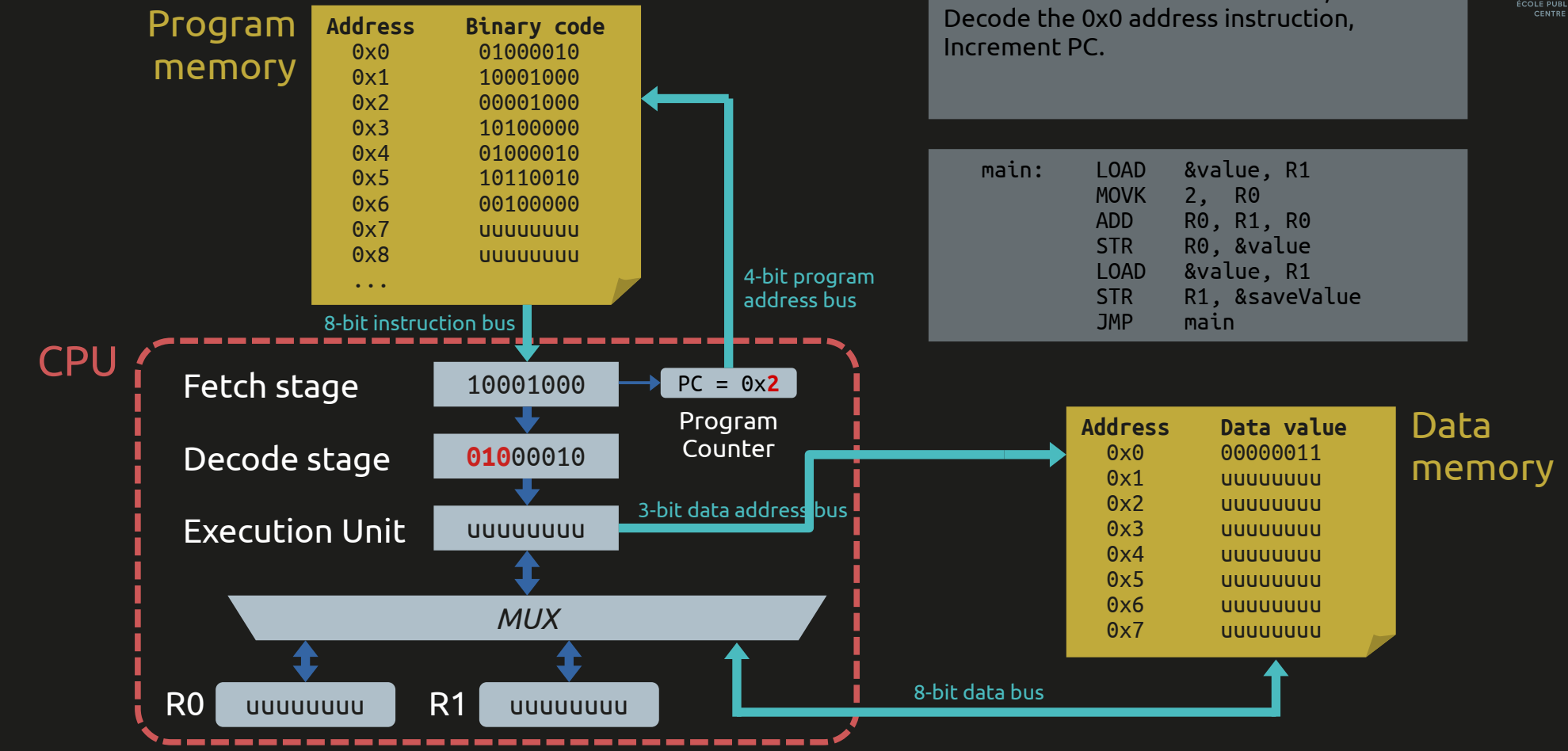
Application starts

Cycle #1:
Fetch the 0x0 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

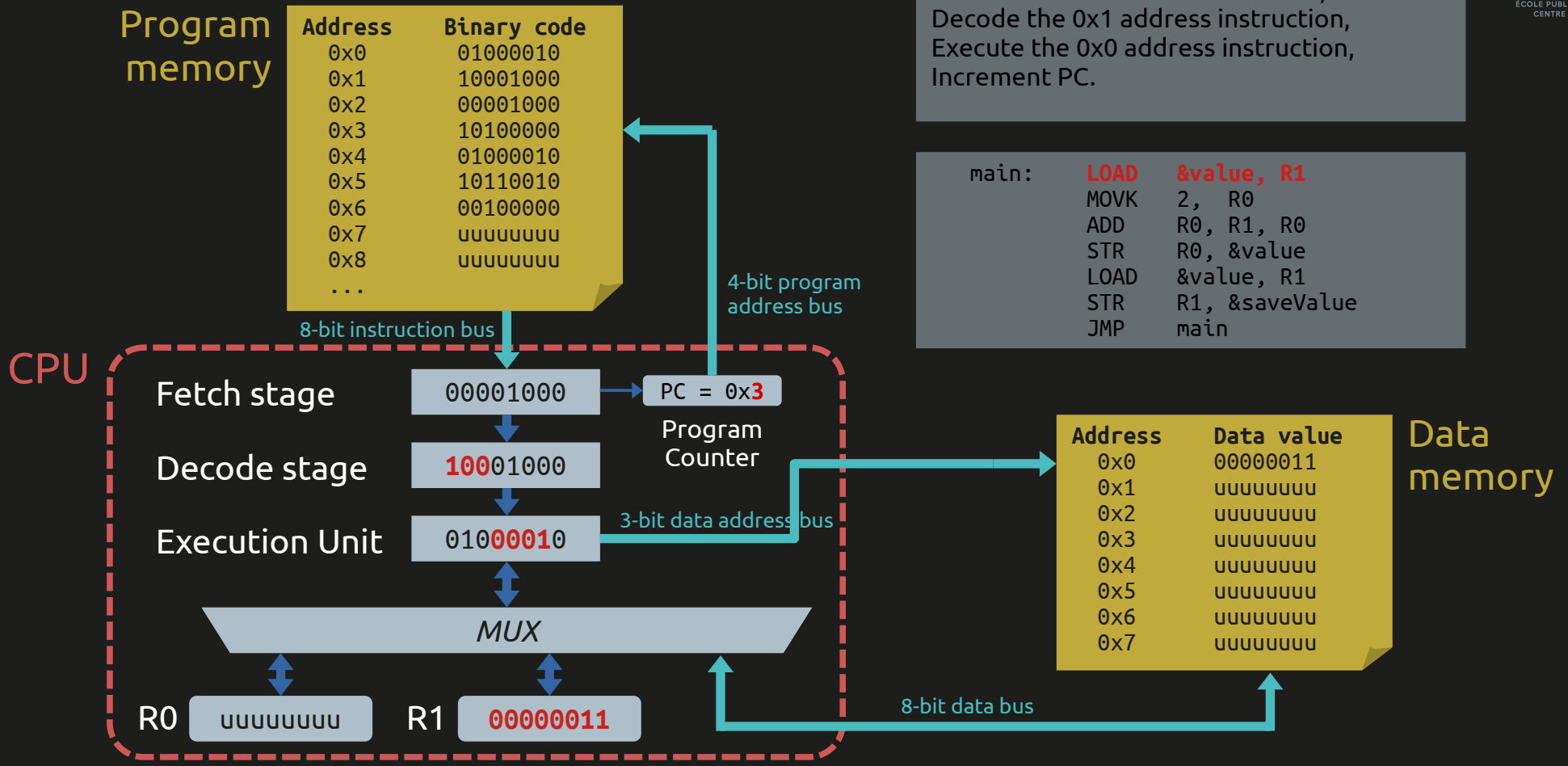


Cycle #2:
Fetch the 0x1 address instruction,
Decode the 0x0 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

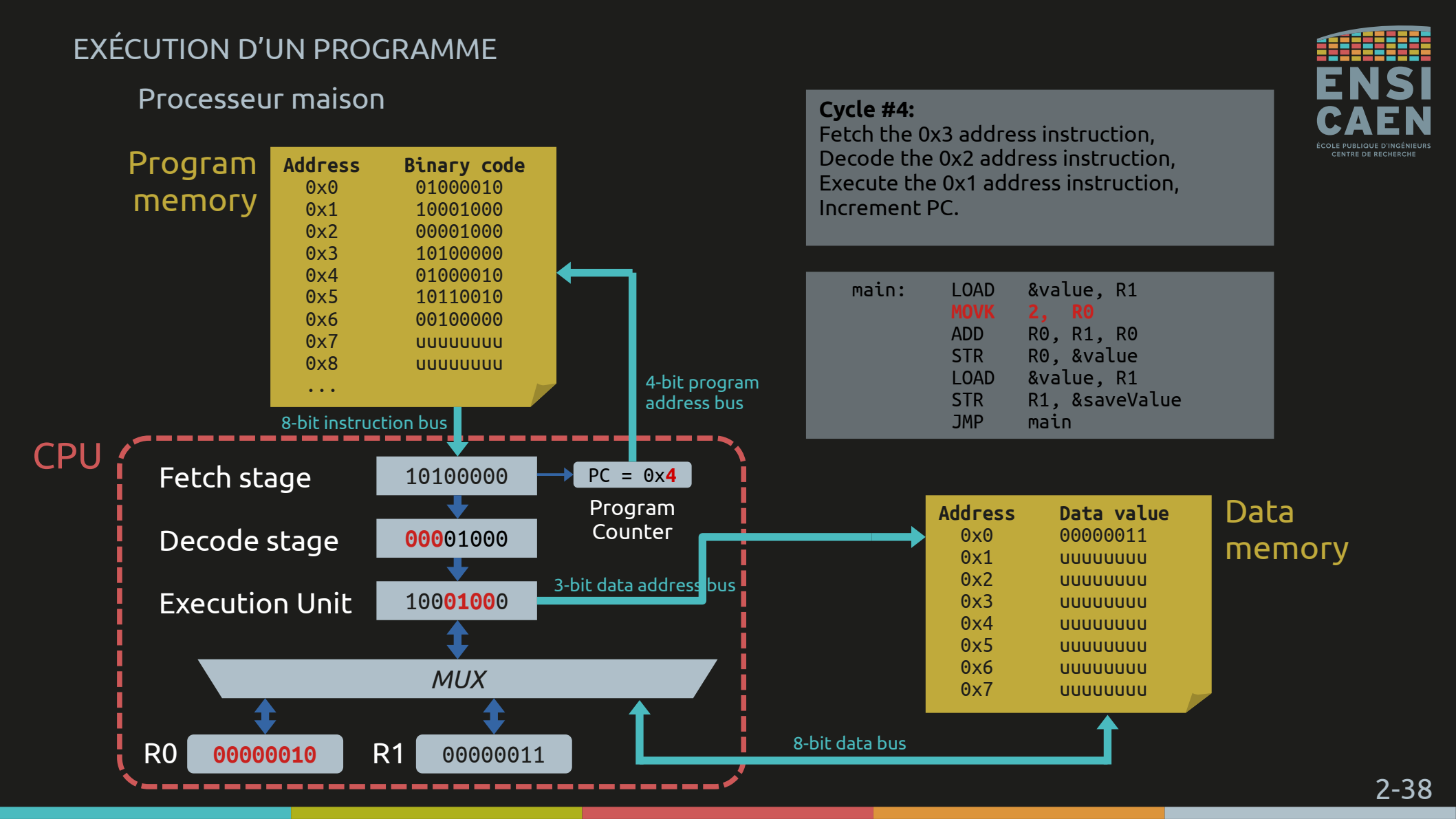


Cycle #3:
Fetch the 0x2 address instruction,
Decode the 0x1 address instruction,
Execute the 0x0 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

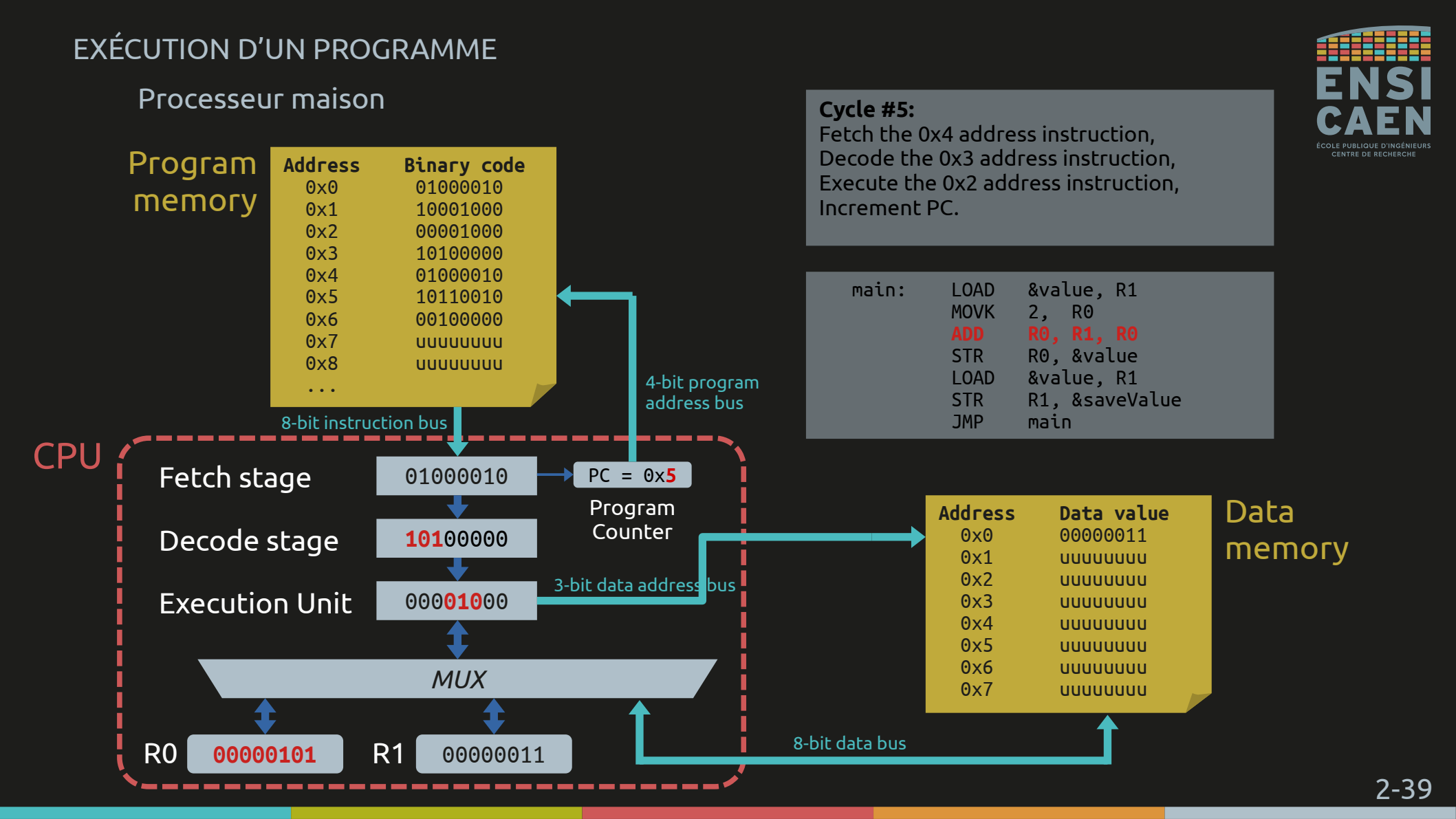


Cycle #4:
Fetch the 0x3 address instruction,
Decode the 0x2 address instruction,
Execute the 0x1 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

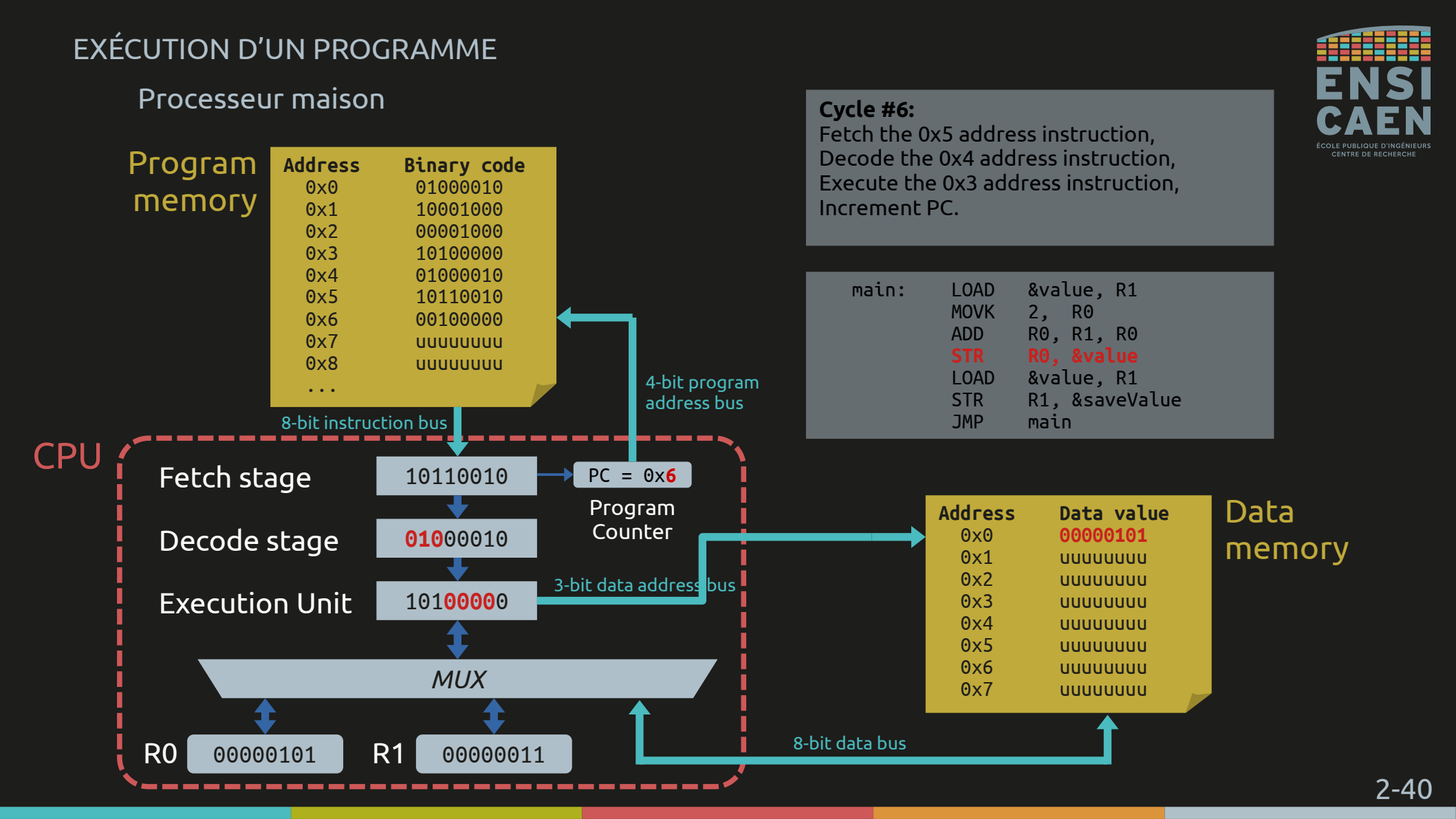


Cycle #5:
Fetch the 0x4 address instruction,
Decode the 0x3 address instruction,
Execute the 0x2 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

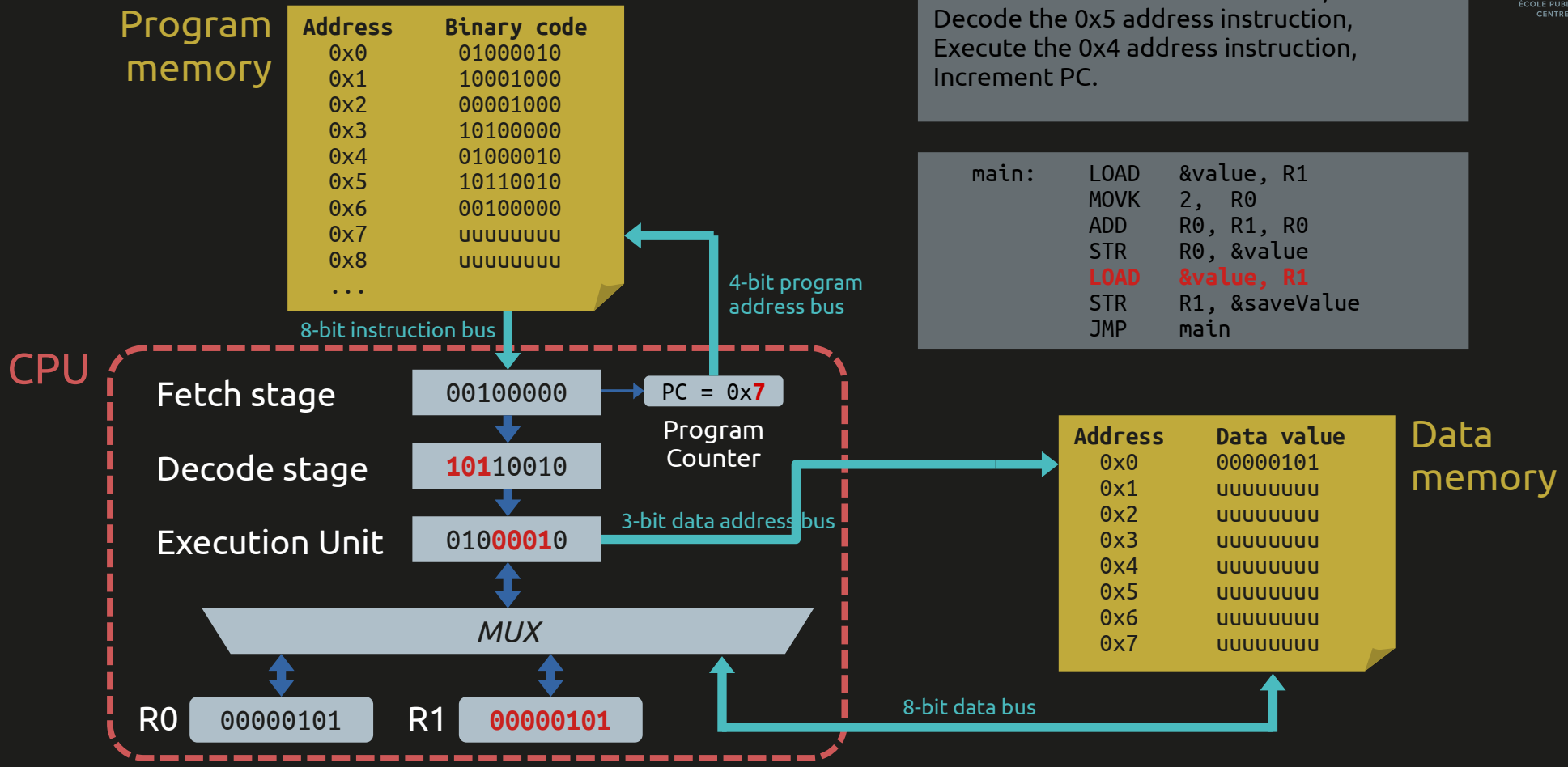


Cycle #6:
Fetch the 0x5 address instruction,
Decode the 0x4 address instruction,
Execute the 0x3 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

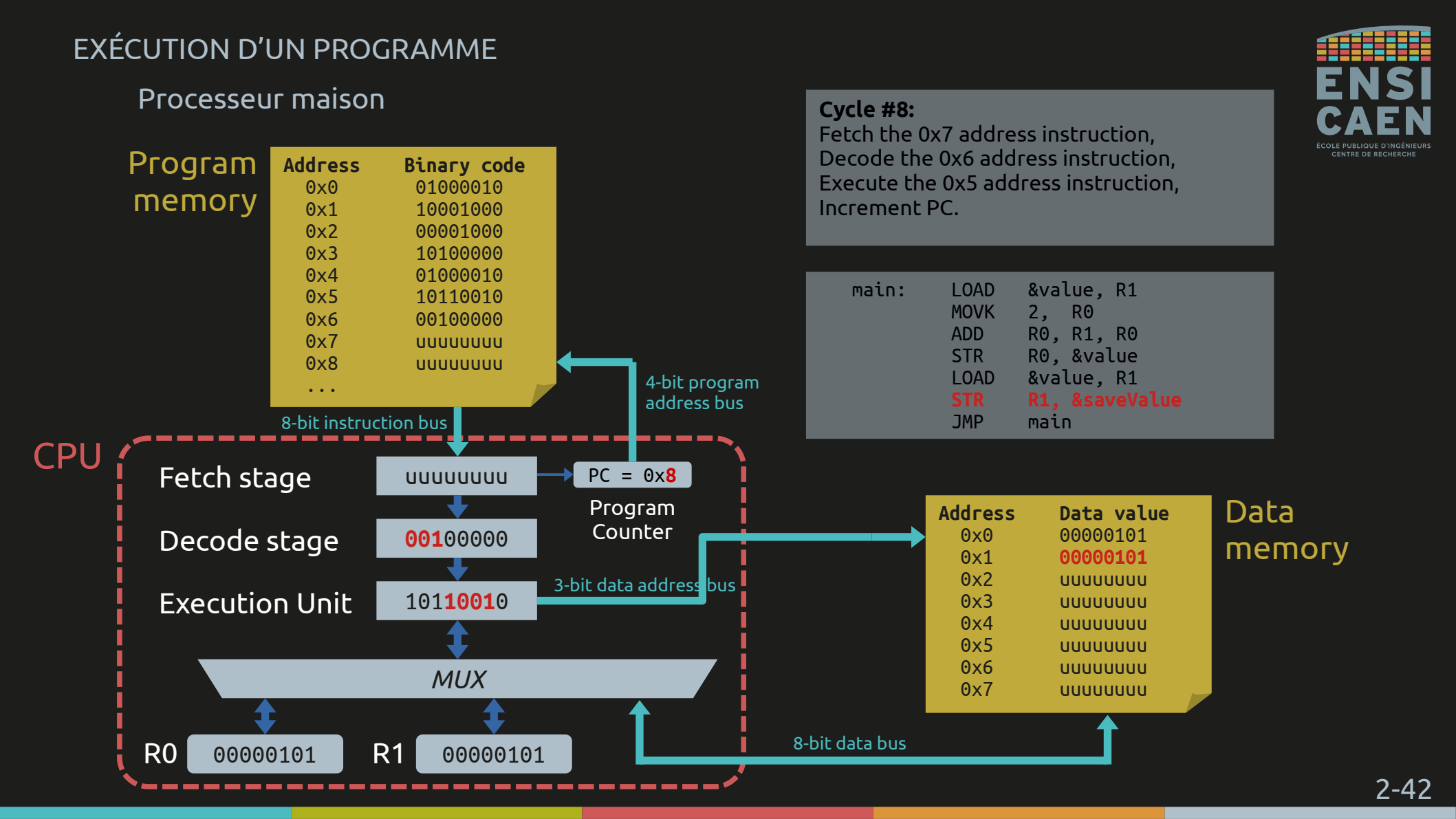


Cycle #7:
Fetch the 0x6 address instruction,
Decode the 0x5 address instruction,
Execute the 0x4 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

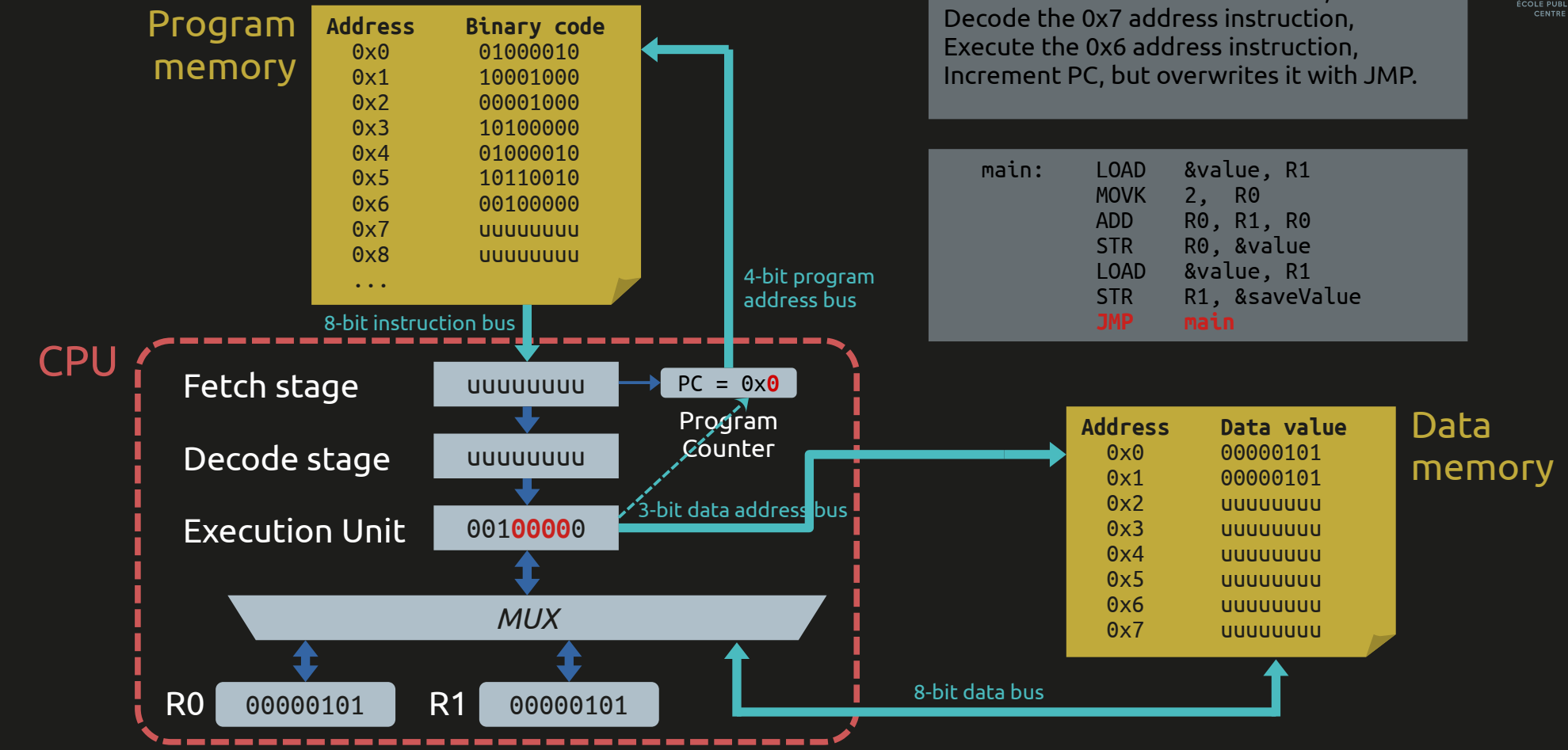


Cycle #8:
Fetch the 0x7 address instruction,
Decode the 0x6 address instruction,
Execute the 0x5 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

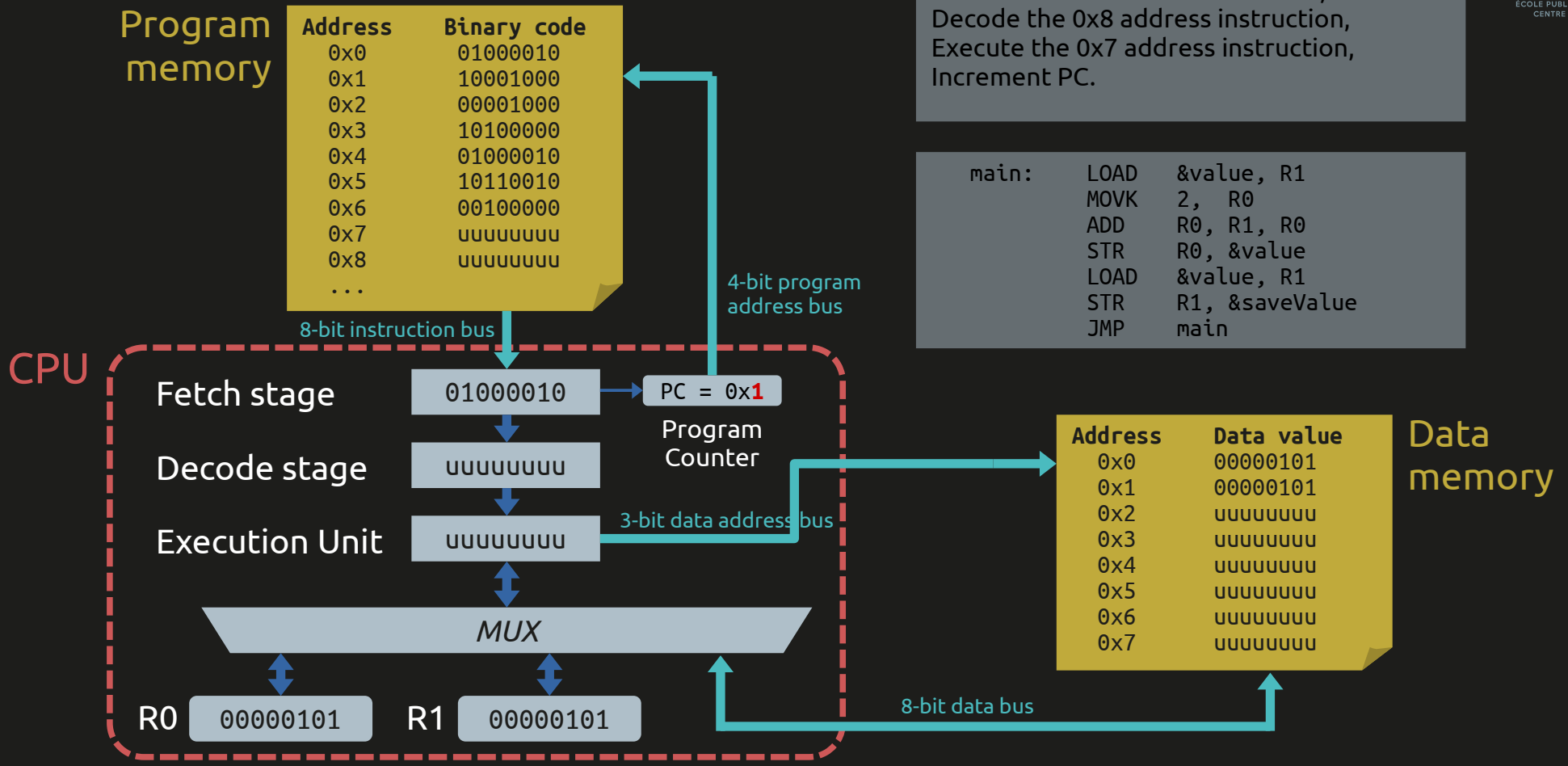


Cycle #9:
Fetch the 0x8 address instruction,
Decode the 0x7 address instruction,
Execute the 0x6 address instruction,
Increment PC, but overwrites it with JMP.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

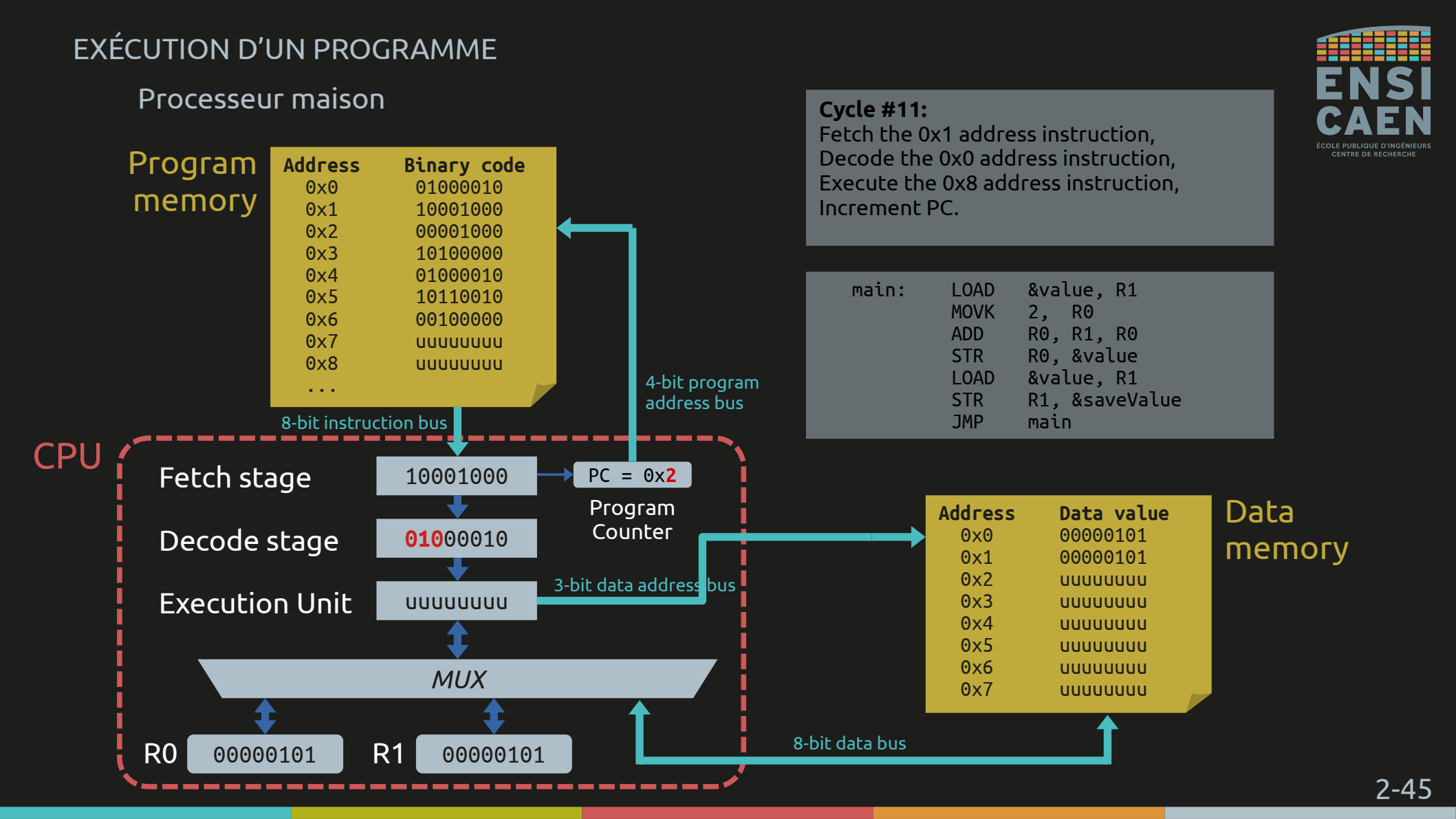


Cycle #10:
Fetch the 0x0 address instruction,
Decode the 0x8 address instruction,
Execute the 0x7 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison

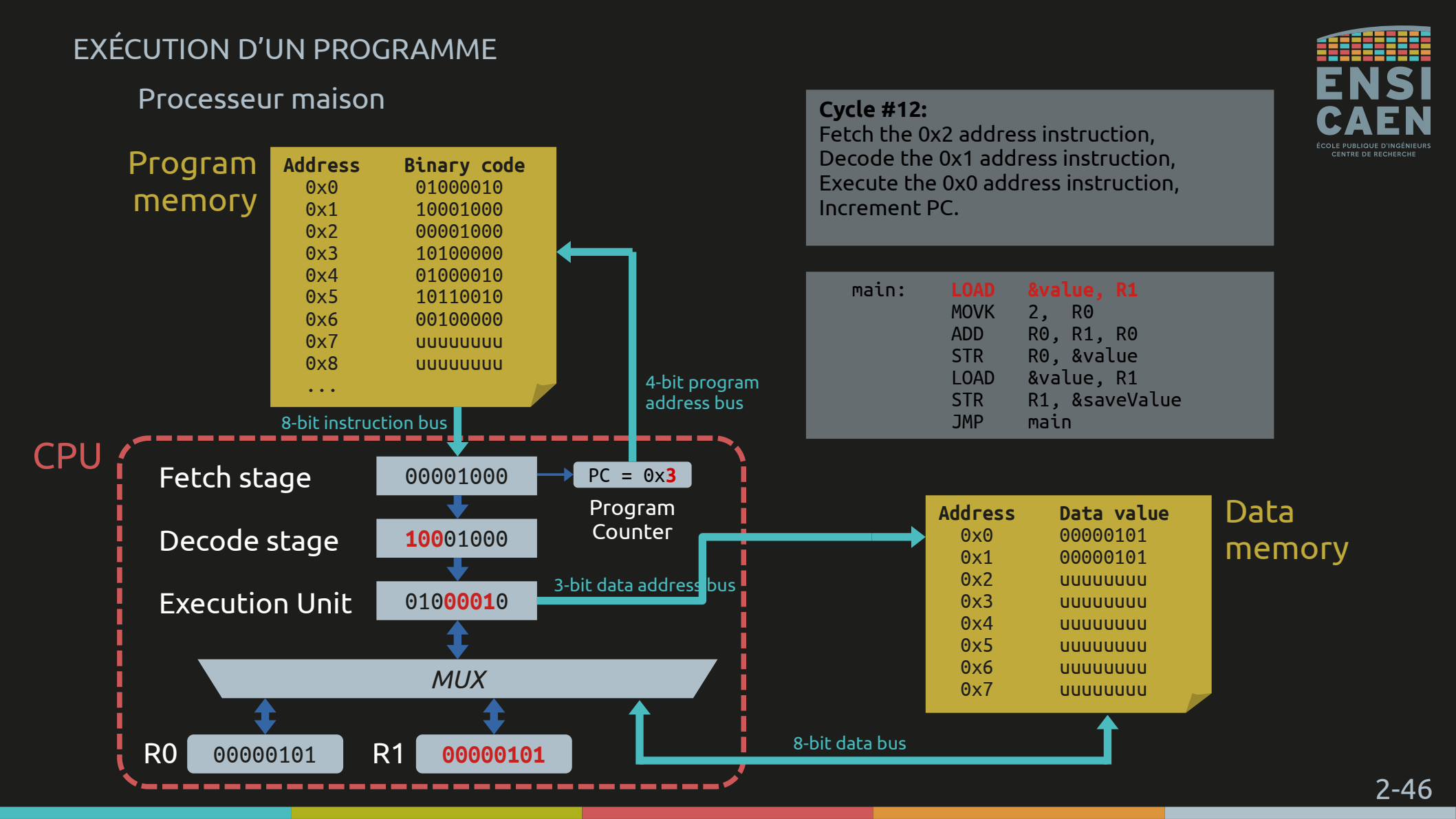


Cycle #11:
Fetch the 0x1 address instruction,
Decode the 0x0 address instruction,
Execute the 0x8 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXÉCUTION D'UN PROGRAMME

Processeur maison



Cycle #12:
Fetch the 0x2 address instruction,
Decode the 0x1 address instruction,
Execute the 0x0 address instruction,
Increment PC.

```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

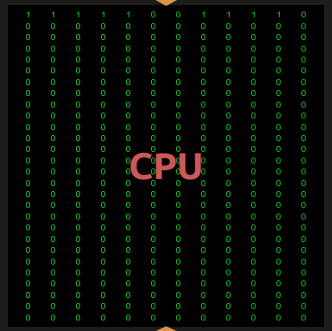
Et ça continue, encore et encore ...

EXÉCUTION D'UN PROGRAMME

Processeur

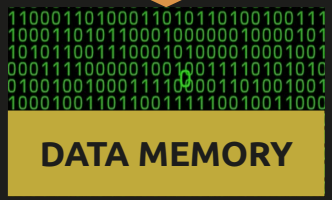
Persistent
EEPROM

INSTRUCTION BUS (Read only)



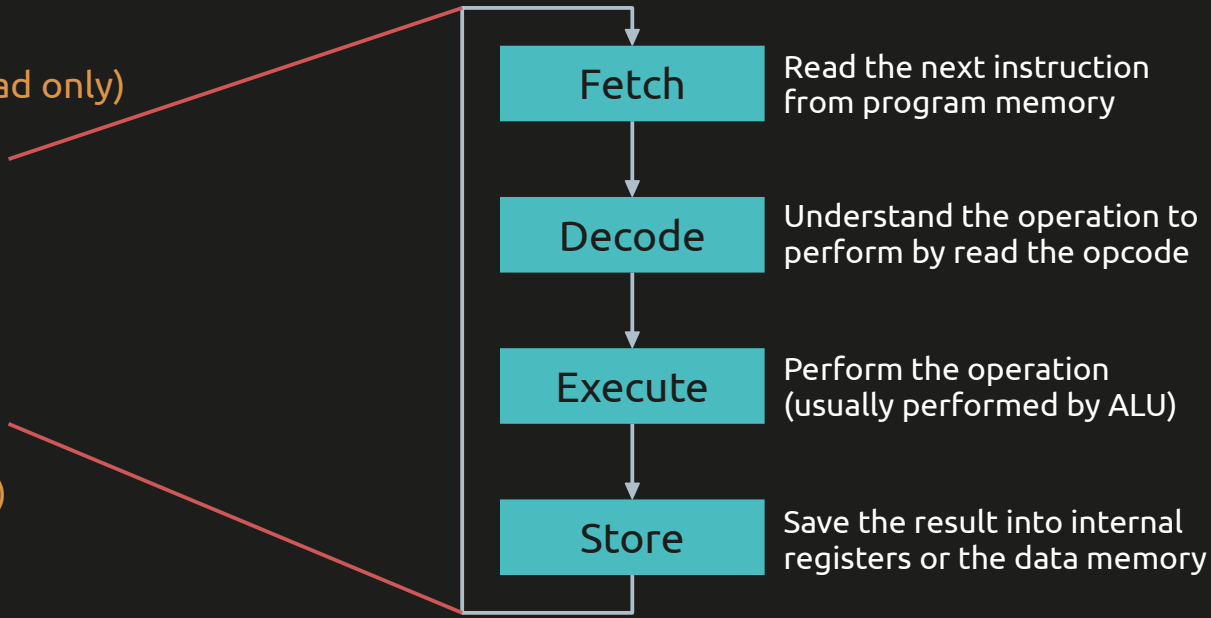
FETCH
DECODE
EXECUTE
WRITEBACK

DATA BUS (Read/Write)



Volatile
RAM

CPU CYCLE



PÉRIPHÉRIQUES

Exemples

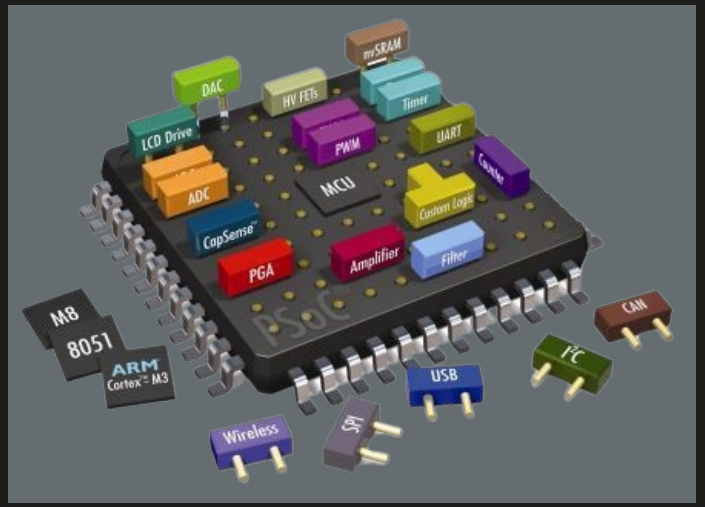
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Les périphériques sont des fonctions matérielles dédiées à une tâche spécifique.

Le CPU peut déléguer quelques tâches aux périphériques (compter, FFT, …) afin d'alléger sa charge et de rester concentré sur la supervision.

Mais la plupart des périphériques sont des interfaces d'entrées/sorties (*General Purpose I/O*, *analogue I/O*, communication…).

Les périphériques forment un ensemble de services (GPIO, ADC, timers, SPI/IœC/UART/USB/Eth, …) qui varient d'un processeur à un autre.r.



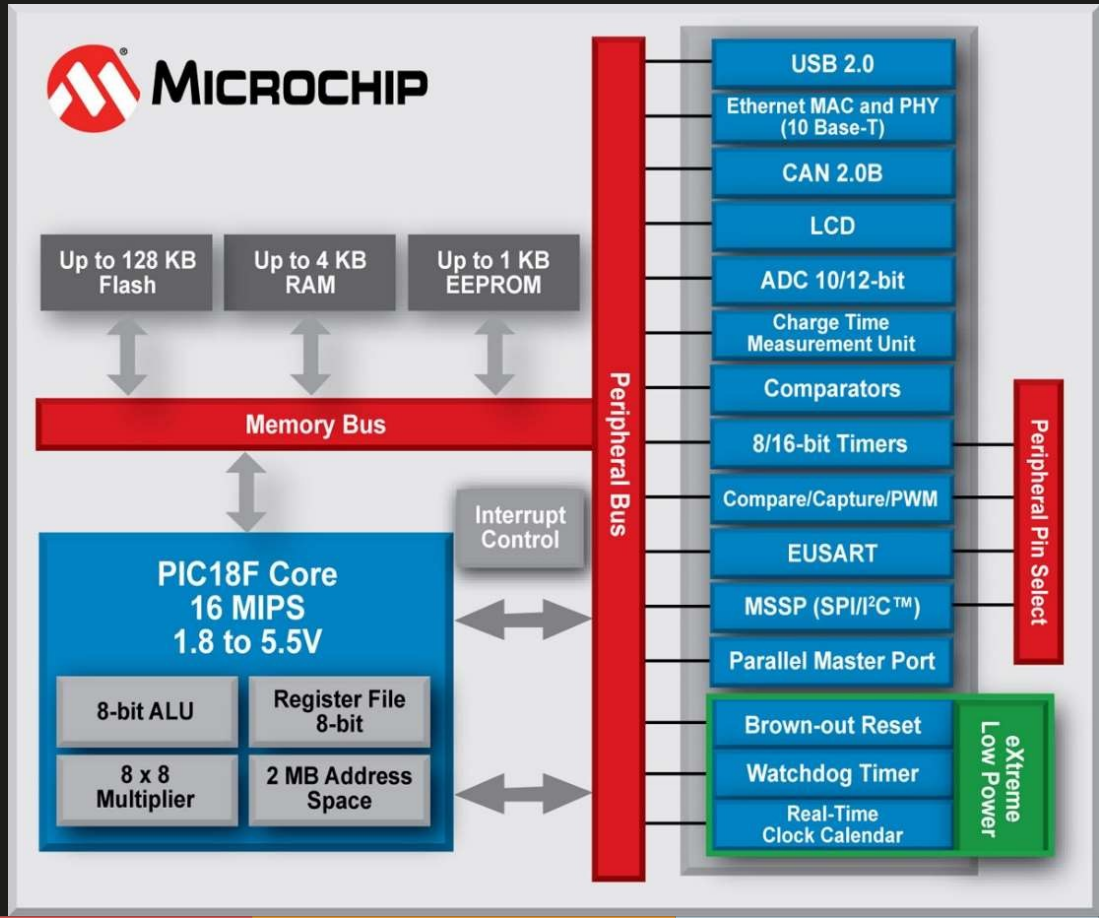
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Exemple

PIC18 de Microchip (8-bit MCU)

Ce micro-contrôleur sera utilisé comme exemple pendant le cours et utilisé pendant les TP.

Les périphériques seront donc abordés en temps voulu.



CONTACT

Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)