Chapitre 4

Central Processing Unit

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

CC BY NC SA

CONSTITUTION D'UN CPU

Unité d'exécution

Unité de contrôle

Banque de registres

Fetch-Decode-Execute-Writeback

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.

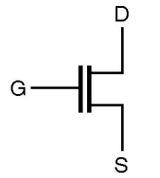
Il contrôle aussi de nombreux fils/signaux, ce qui lui donne un contrôle indirect vers toutes les autres fonctionnalités matérielles.

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

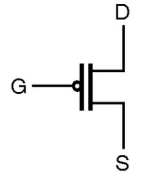
D'un point de vue couche d'abstraction (ou complexité si vous préférez), le CPU se situe au dessus de la couche circuits logiques.

Il est donc construit à partir de nombreux circuits logiques intégrés sur un seul CI.

Transistor layer



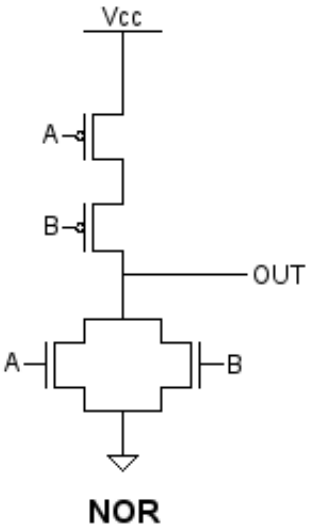
(e)
NMOS used in Digital circuits



(f)
PMOS used in Digital circuits

Logic gate layer

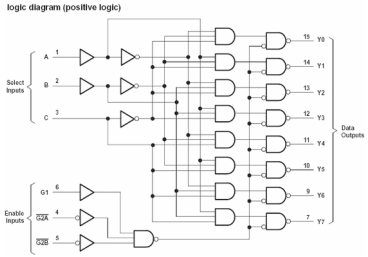
(mostly NOT, NOR and NAND)

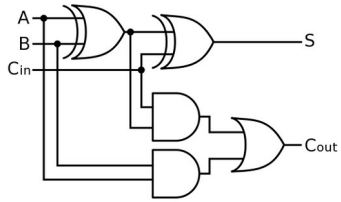


NOR

Logic circuit layer

(Multplixer, adder, … and D-latches)





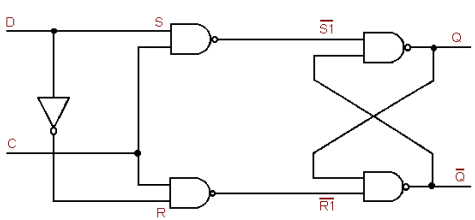
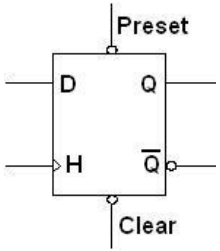


Fig. 7. - Structure d'une bascule D latch.



CPU layer

(ALU, registers)

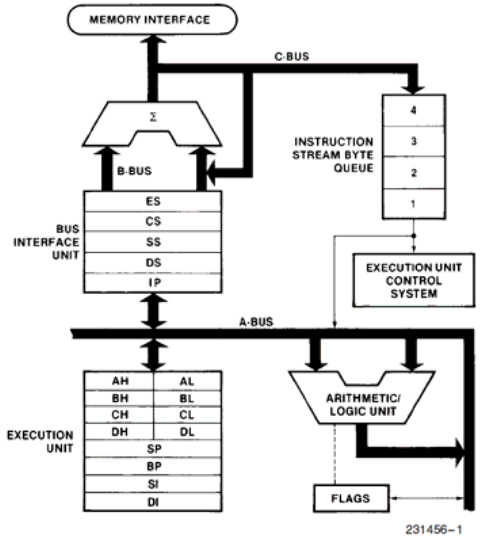
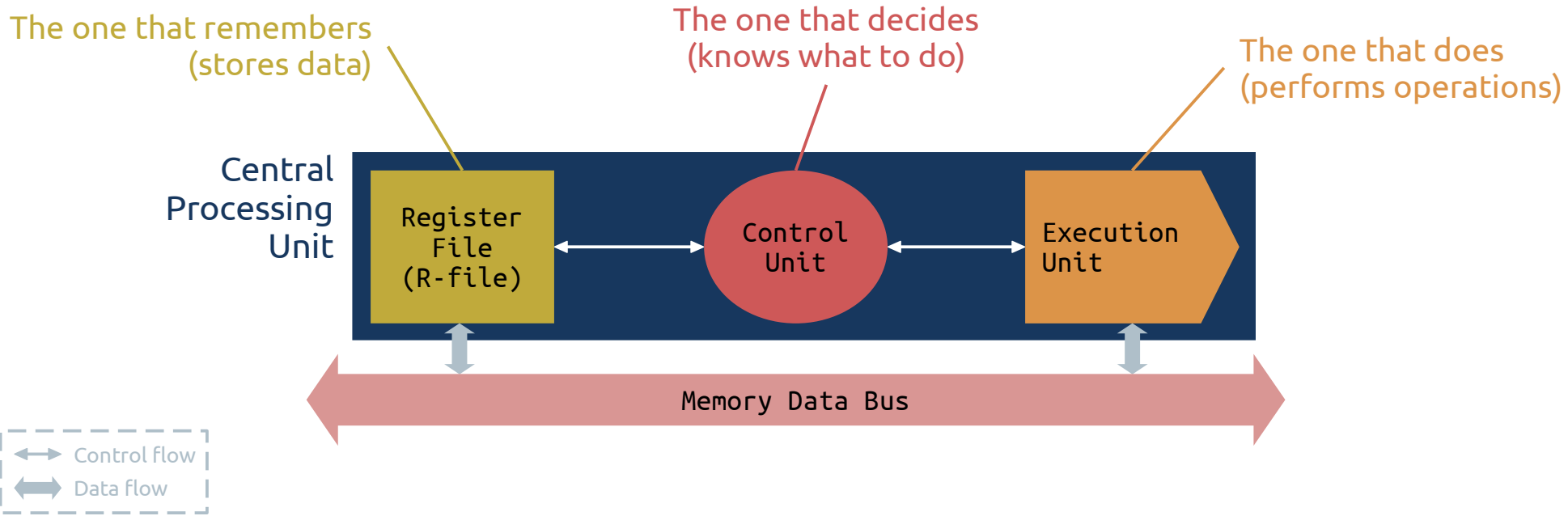


Figure 1. 8088 CPU Functional Block Diagram

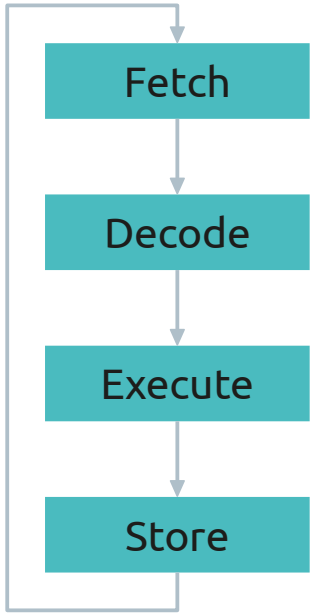
Schéma fonctionnel

Un CPU possède au moins une unité de contrôle (*Control Unit*), une unité d'exécution (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.
Beaucoup de CPU modernes sont capables de récupérer plusieurs instructions en une seule phase de fetch (CPU superscalaires, VLIW, …).

Détecte l'opération à réaliser grâce au code binaire opératoire (opcode).
Cela permet de savoir quelles ressources (registres, unité d'exécution, …) sont nécessaires.

Réalise l'opération, généralement via l'une des unités d'exécution (ALU, FPU, multiplier, …).

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 :

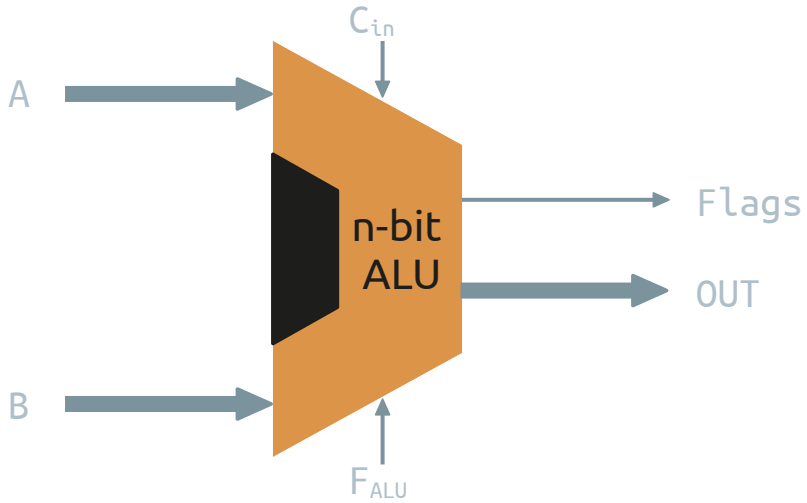
- Une unité arithmétique et logique (ALU, *Arithmetic and Logic Unit*)
  - Opérations logiques et arithmétiques élémentaires
- Une unité de calcul en virgule flottante (FPU, *Floating-Point Unit*)
  - Pour les processeurs performants
- Un multiplieur
- Un registre à décalage
- …

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
 ...
```

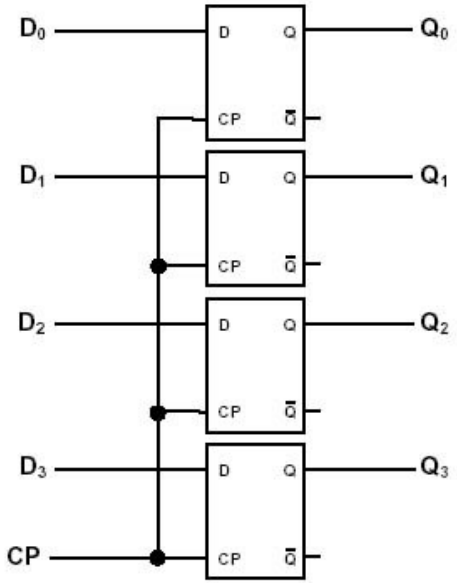
La **banque de registres (*Register file*)** contient, vous l'aurez deviné, des registres.

Les registres sont de petits emplacements mémoire situés au cœur 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*, générés par l'ALU,
Le *Program Counter (PC) ou Instruction Pointer (IP)* contient l'adresse de la prochaine instruction à exécuter,
Le *Instruction Register (IR)* contient 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

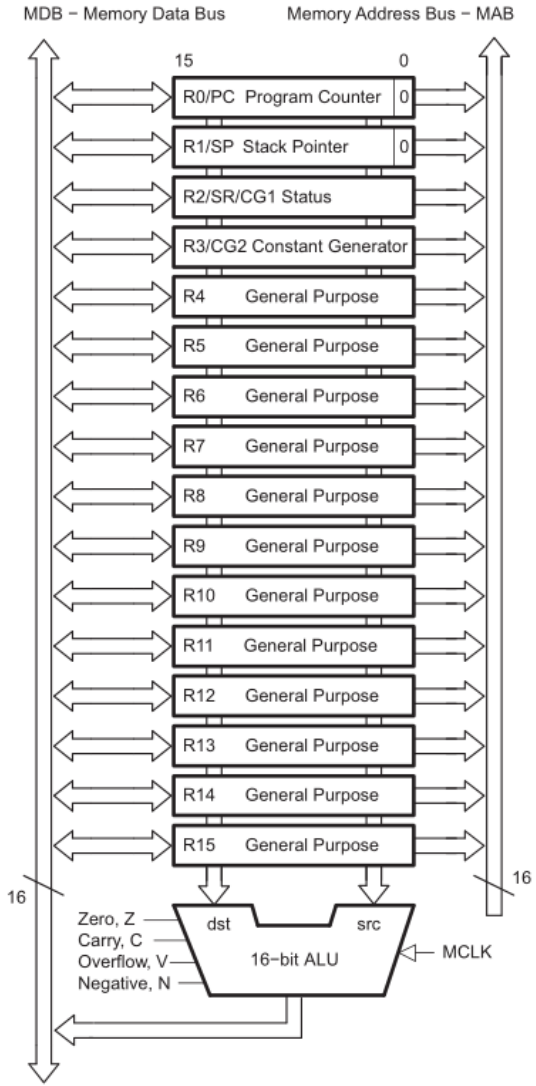


Figure 3-1. CPU Block Diagram

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.

Exemple :

Registre EFLAGS d'un Intel 64 et IA-32.

Faisant notamment office de registre de statut.

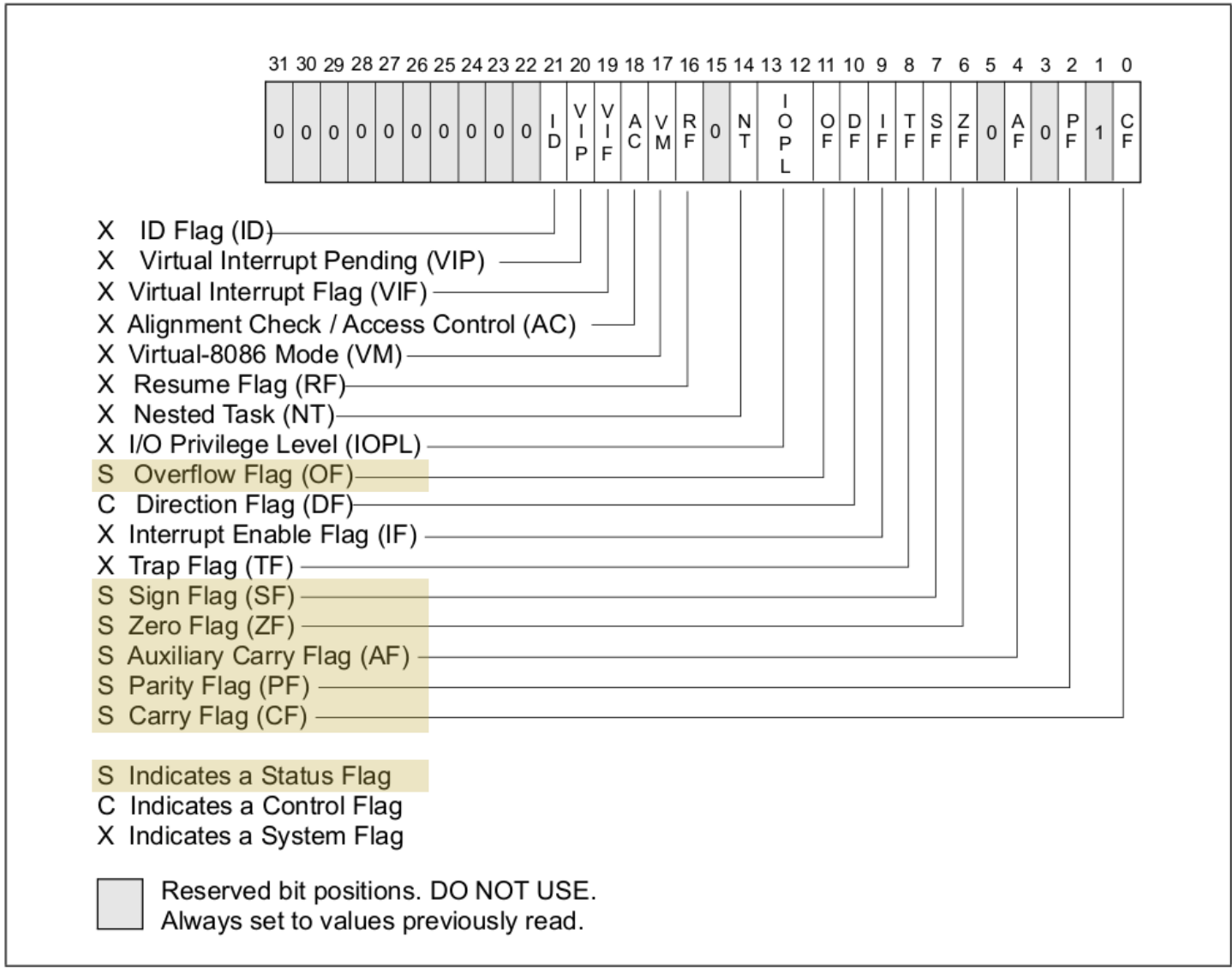


Figure 3-8. EFLAGS Register

« Intel 64 and IA-32 Architectures Software Developer's Manual »
Volume 1 – Basic Architecture
Intel, June 2023

HARDWARE PIPELINE

Principes

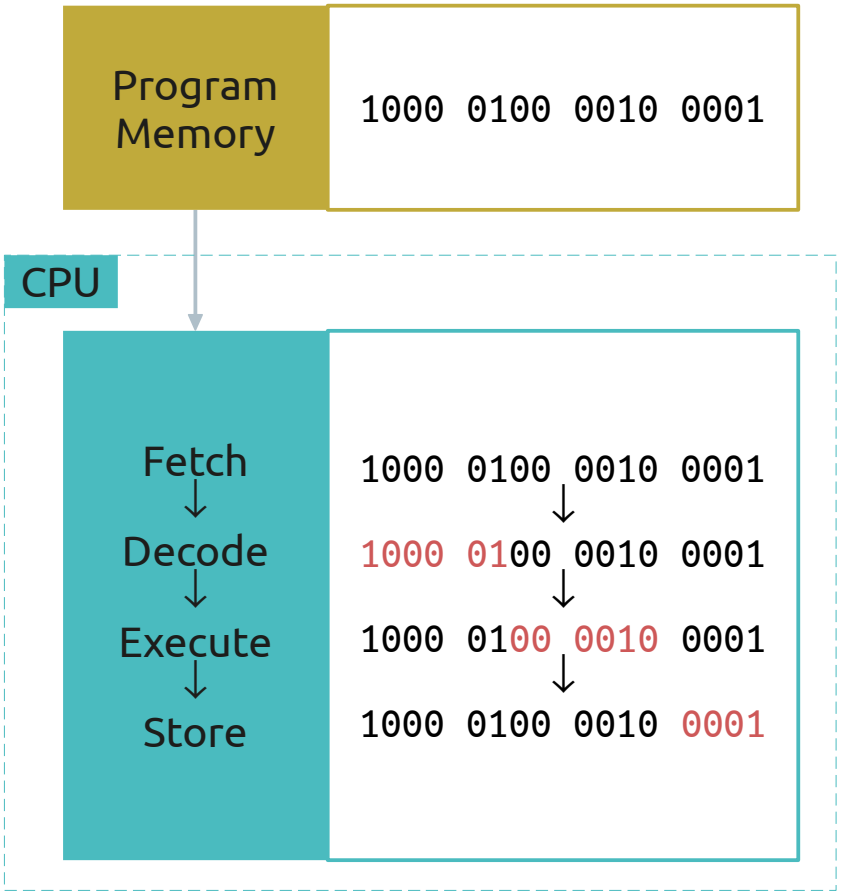
Aléas

Prédiction de branchement

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

1-stage hardware pipeline



Chaque instruction passe par ces étapes.

Supposons que chaque étape prend un cycle, il faut donc 4 cycles pour exécuter chaque instruction.

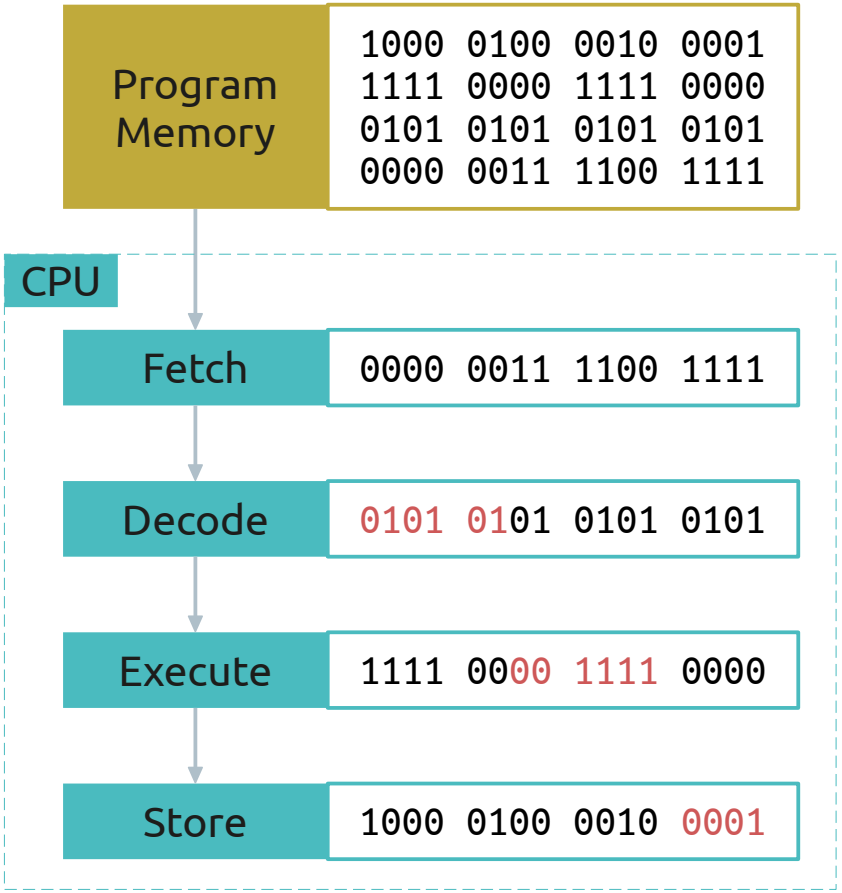
C'est la latence : nombre de cycle nécessaires pour effectuer un traitement complet sur une instruction.

Ensuite l'instruction suivante est chargée et son résultat est produit 4 cycles plus tard.

C'est la cadence : nombre de cycles entre le résultat de deux instructions successives.

On note quand même un défaut : seuls 25 % du CPU sont utilisé en moyenne.

4-stage hardware pipeline



Aujourd'hui la plupart des architectures peut effectuer une partie de ces étapes en simultané.

Ici, la latence ne change pas : il faut toujours 4 cycles pour effectuer un traitement complet sur l'instruction.

En revanche, dès qu'une instruction passe à l'étage i+1, l'instruction suivante en mémoire programme vient occuper l'étage i. Ainsi tous les étages sont utilisés à un instant donné.

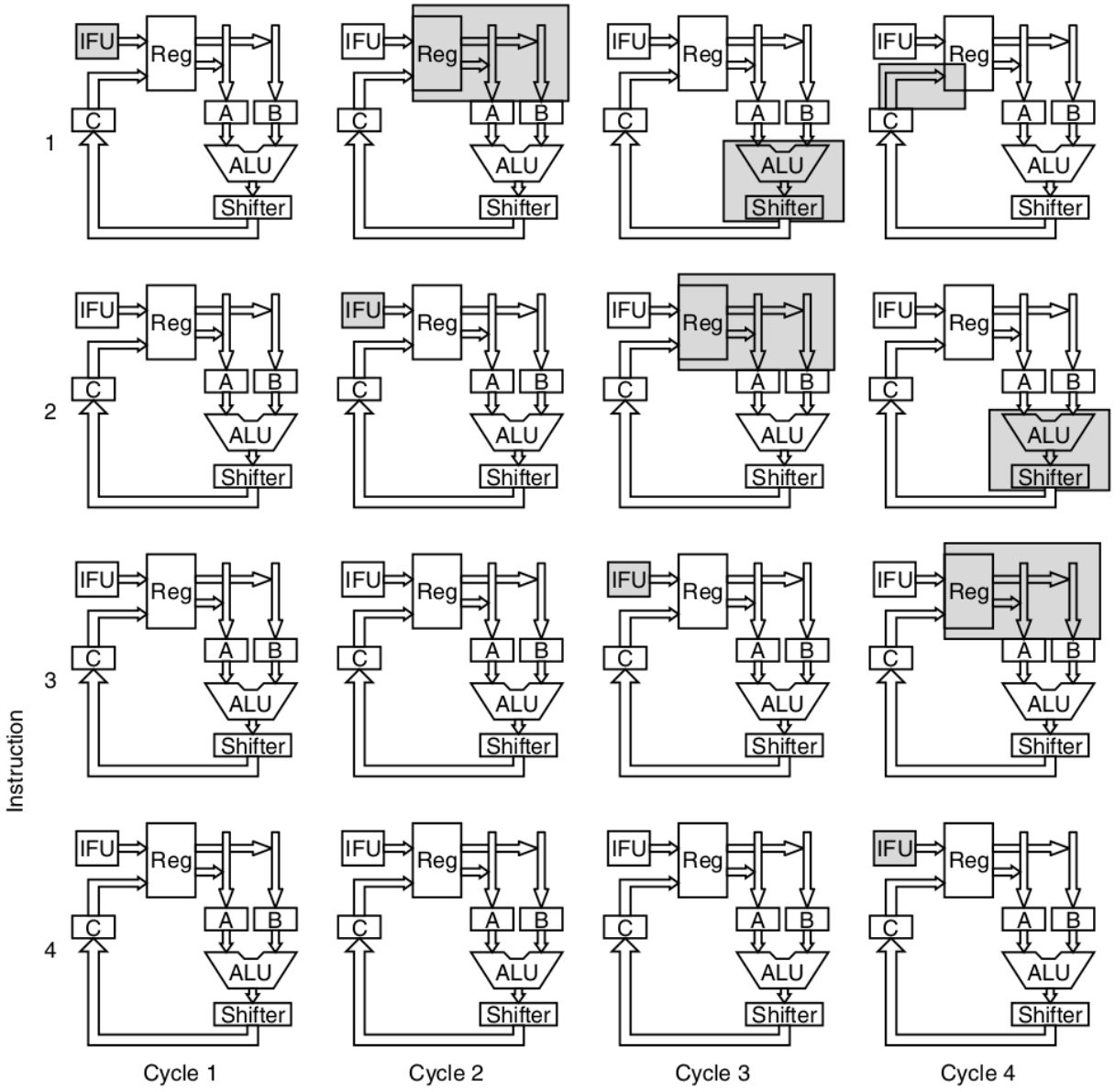
Ainsi en régime permanent, le résultat d'une instruction est produit à chaque cycle. La cadence a donc été réduite à 1 cycle.

Le **hardware pipeline** est une technique matérielle d'optimisation du temps d'exécution.

HARDWARE PIPELINE

4-stage hardware pipeline

Illustration graphique du fonctionnement d'un pipeline à 4 étages.



« Structured Computer Organization »
6th edition, 2013.
Andrew Tanenbaum and Todd Austin.

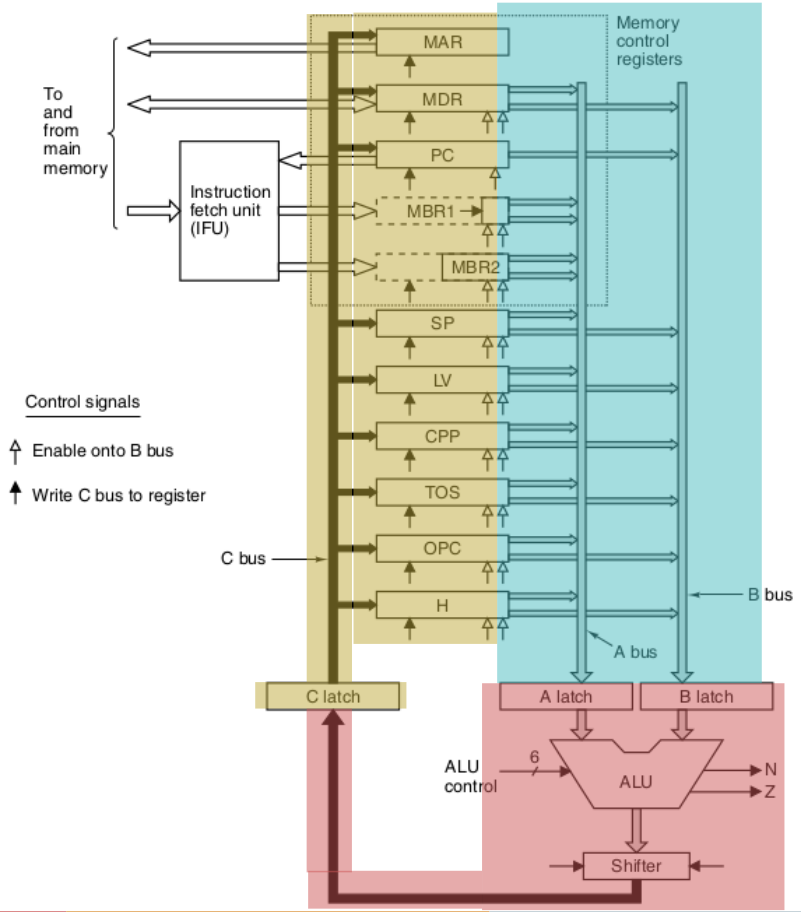
Implémentation

Afin de permettre au CPU de traiter plusieurs instructions à la fois, les différents étages du pipeline doivent être « isolés » les uns des autres.

Des bascules D (*flip-flops* ou *latches*) sont employées pour cela. Elles délimitent matériellement chaque étage en ne mettant à jour leur sortie qu'au coup d'horloge du CPU.

Les bascules sont synchronisées entre elles. Et en considérant le temps de propagation du signal entre l'entrée et la sortie d'un étage (il n'est pas négligeable), chaque étage contient bien des informations liées à une instruction différente des autres étages.

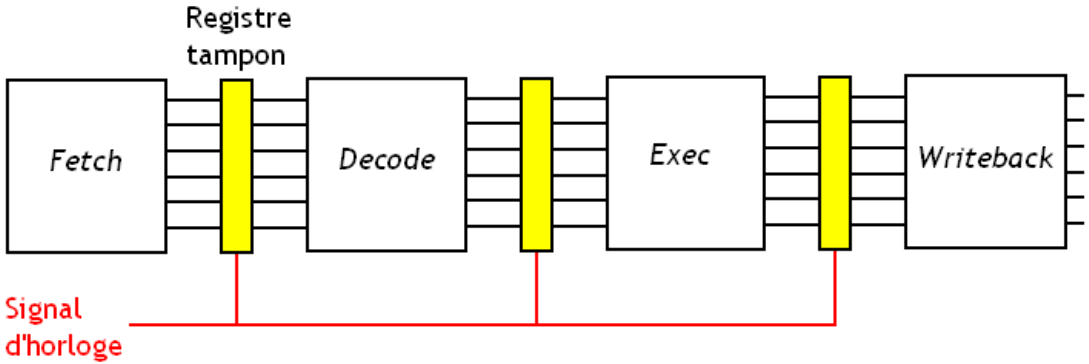
Ci-contre, un exemple de CPU décomposé en trois étages de pipeline. Notez que les sorties de la file de registre sont aussi activées par front d'horloge, ce qui signifie qu'elles constituent aussi des bascules d'étage de pipeline.



Nombre d'étages

Théoriquement, soit un temps de latence t_l donné, le temps de cadence t_c est égal à la latence divisée par le nombre n d'étages du pipeline : $t_c = t_l / n$.

En pratique, c'est plus compliqué car le CPU ne peut pas être décomposé en n étages égaux en temps de traitement. Cependant, plus le nombre d'étages est grand, plus la distance entre deux étages est courte. Ainsi, en plus de réduire le temps de cadence pour une fréquence d'horloge donnée, il est aussi possible d'augmenter cette fréquence d'horloge : c'est du *win-win*.



Les petits processeurs RISC (ex : MCU) ne contiennent que quelques étages de pipeline. Les GPP (processeurs CISC) en contiennent entre 15 et 20. Ces architectures sont dites *super-pipelined*. Le gain apporté par un pipeline matériel tourne aujourd'hui entre x5 et x10.

Nombre d'étages

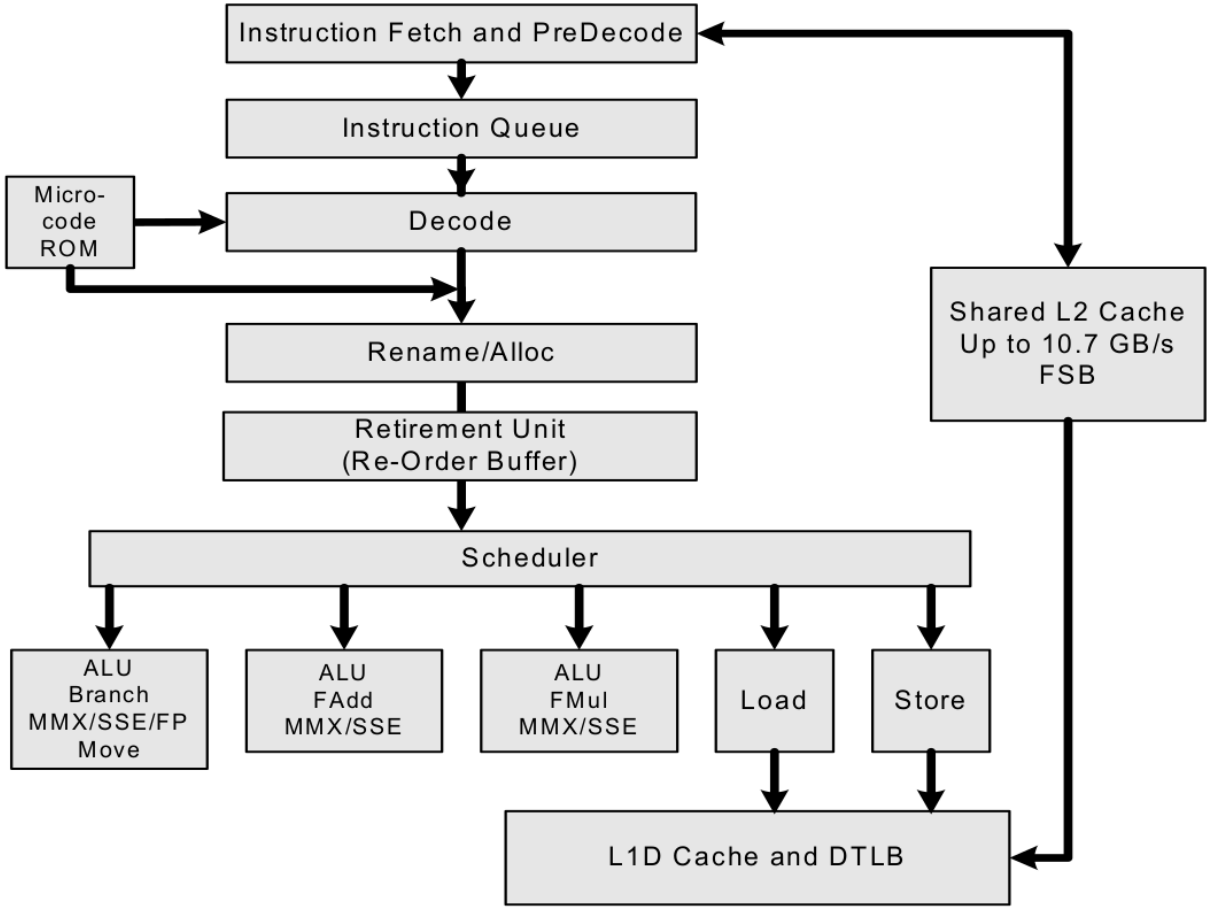


Figure 2-3.
The Intel® Core™ Microarchitecture Pipeline Functionality

Le pipeline matériel est apparu chez Intel avec le 80486 (1989), utilisant 5 étages.

Il est monté jusqu'à 31 étages avec le Pentium 4 Prescott (2004).

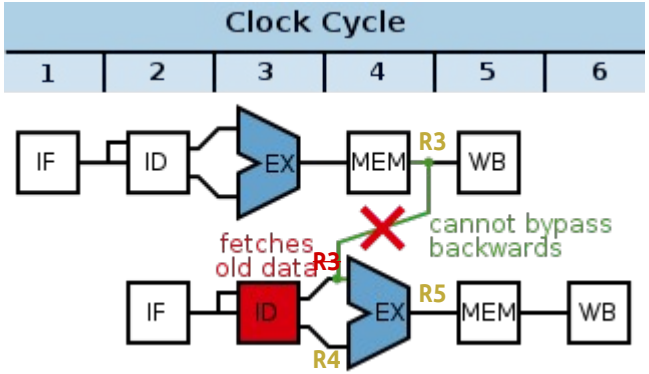
Aujourd'hui (2023) le pipeline matériel d'un Intel Core contient 14 étages.

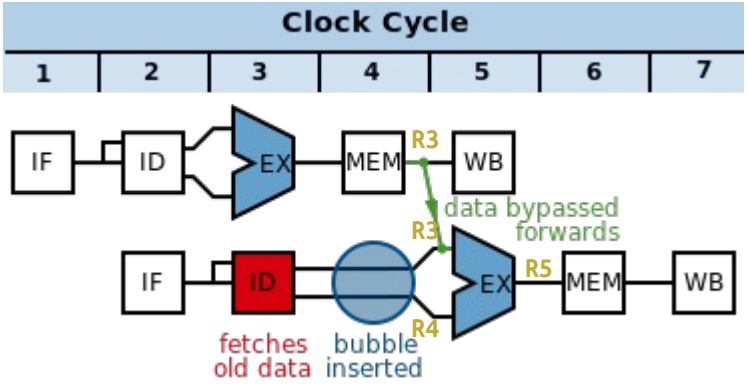
La dépendance de données est un premier problème avec les super-pipelines.

Considérons ces deux pseudo-instructions : (I1) `LOAD addr → R3`
(I2) `ADD R3, R4 → R5`

Il y a dans ce cas un dépendance RAW (*Read After Write*) puisque l'instruction I2 lit un registre R3 qui vient d'être écrit par l'instruction I1. Selon le découpage du pipeline, il se peut que l'instruction I2 récupère ses opérandes d'entrée avant que l'instruction I1 n'ait eu le temps de mettre le registre R3 à jour.

Dans ce cas une stratégie possible consiste à insérer une bulle (on attend un cycle) afin de garantir la validité des données. D'un point de vue fonctionnel cela induit un ralentissement de la cadence, mais il ne faut pas oublier que cette logique de « rattrapage » des aléas implique également une plus grande complexité matérielle.





Le principal problème du pipeline est la gestion des branchements

Considérons un test banal en C : `if(…) { /* corps_du_if */ } else { /* corps_du_else */ }`

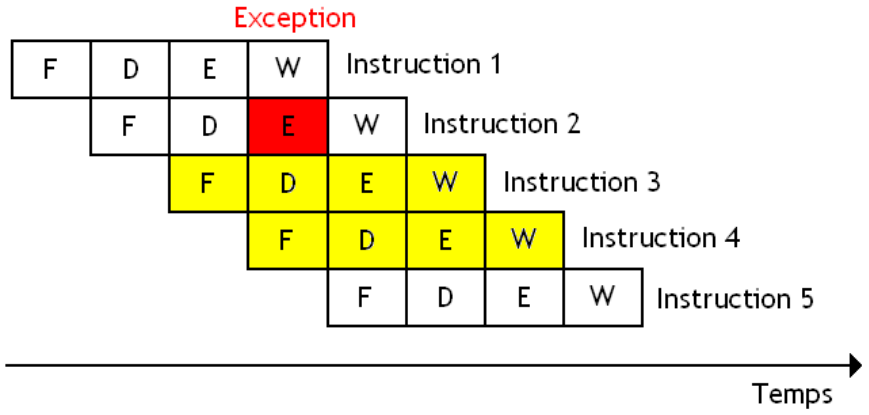
D'un point de vue fonctionnel si la condition est vraie, alors les instructions `/* corps_du_if */` seront exécutées, sinon ce seront les instructions `/* corps_du_else */` .

Au moment où la condition est testée, il n'y a aucun moyen de savoir dans quel corps on rentrera. Or le pipeline matériel a pour rôle de charger les instructions suivantes alors qu'il ne connaît pas encore le résultat du test.

On ne connaît le résultat du test qu'après un certain nombre de cycles (en rouge).

Si les mauvaises instructions ont été chargées (en jaune), il faut annuler leur exécution : c'est la purge du pipeline. Évidemment cela entraîne une baisse drastique des perfs !

Notons que cela se produit aussi lors des branchements inconditionnels, car l'adresse de branchement n'est connue qu'après l'étage d'exécution.



Le code exécuté sur des processeurs équipés de pipeline profond est généralement du code de contrôle, rempli d'appel de fonctions, de tests, d'interruptions et d'exceptions. Or c'est justement ce qui rend inefficace un super-pipeline, loi de Murphy oblige !

Backward/Forward

Une solution simple est de toujours effectuer des sauts arrières, mais jamais des sauts avants.

Un saut arrière (*backward branch*) correspond à sauter à une instruction d'adresse plus faible que l'instruction courante. Cela est typique des boucles (`for`, `while`, `do-while`) pour lesquelles les sauts se font très souvent.

Un saut avant (forward branch) signifie sauter à une instruction qui n'a pas encore rencontrée. C'est plutôt le cas des structures de test (`if-else`, `switch-case`).

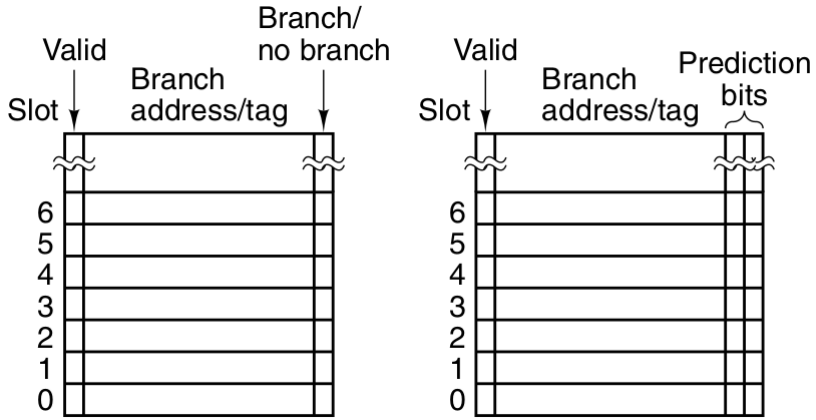
Globalement, même si la stratégie de ne pas effectuer les sauts avants est discutable, celle de toujours effectuer les sauts arrière s'avère payante.

Prédiction de branchement dynamique

Une autre solution est de disposer de matériel dédié dans le CPU pour prédire dynamiquement le résultat des branchements, sous forme d'une table de prédiction (zone mémoire).

Chaque entrée de la table correspond à un test (référencé par l'adresse de l'instruction) auquel s'ajoute un bit qui contient le résultat du dernier branchement ('0' = *do not branch* ; '1' = *branch*). Lorsqu'une instruction de test est rencontrée, la table est consultée, le saut est fait en fonction du bit et ce dernier est mis à jour si besoin.

Cela fonctionne bien, mais il y aura toujours un faux « do-not-branch » en sortie de boucle. En guise d'amélioration pour ce cas précis, le bit ne change d'état que lorsqu'il y a eu deux échecs consécutifs.



Profilage

Une dernière solution purement logicielle existe.

Le programme est simulé lors de sa compilation. Les comportements des tests et branchements sont capturés et analysés, ce qui permet au compilateur de générer des instructions particulières afin de donner des consignes de branchements au matériel qui exécutera le programme.

EXÉCUTION D'UN PROGRAMME

Sur un processeur maison

ISA

Assembleur et binaire

Pipeline matériel

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Processeur maison

Créons notre processeur maison.

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

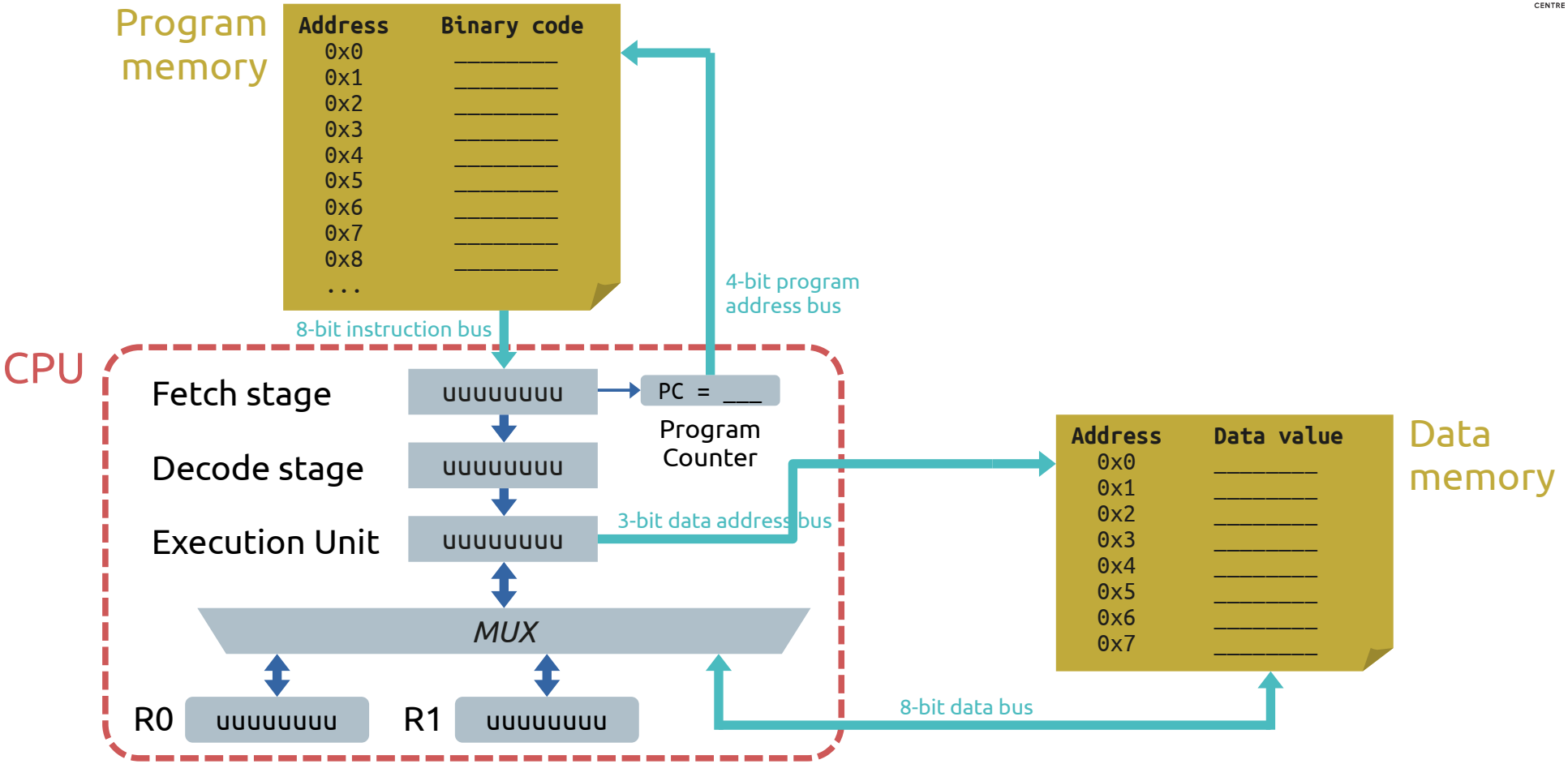
Son jeu d'instruction (*ISA, Instruction Set Architecture*) est purement fictif.

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

Processeur maison



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Solution

C language program

```c
// To keep it simple, we assume that variables
// are already stored and initialised.

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

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

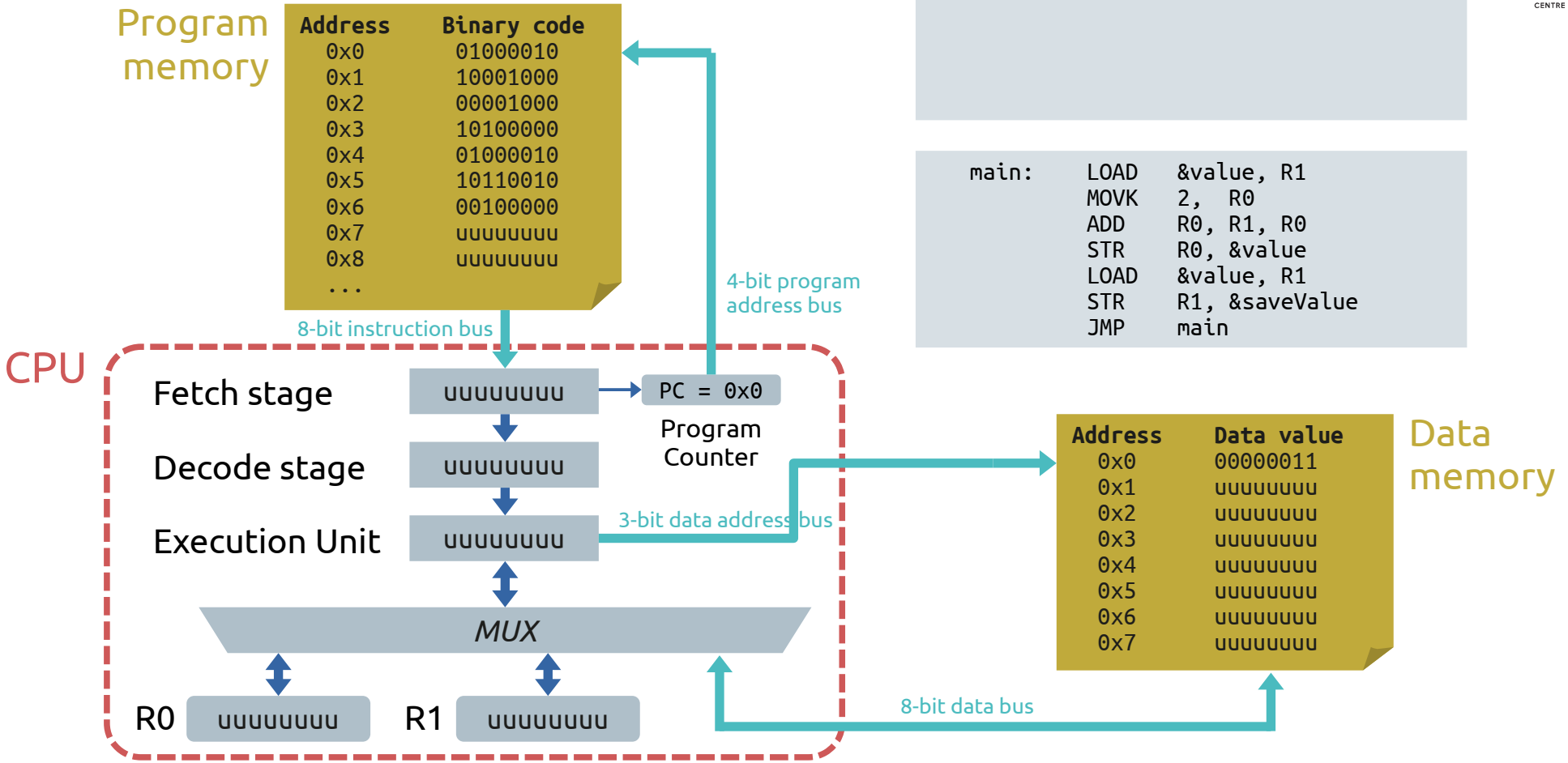
Assembly language program

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

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
```

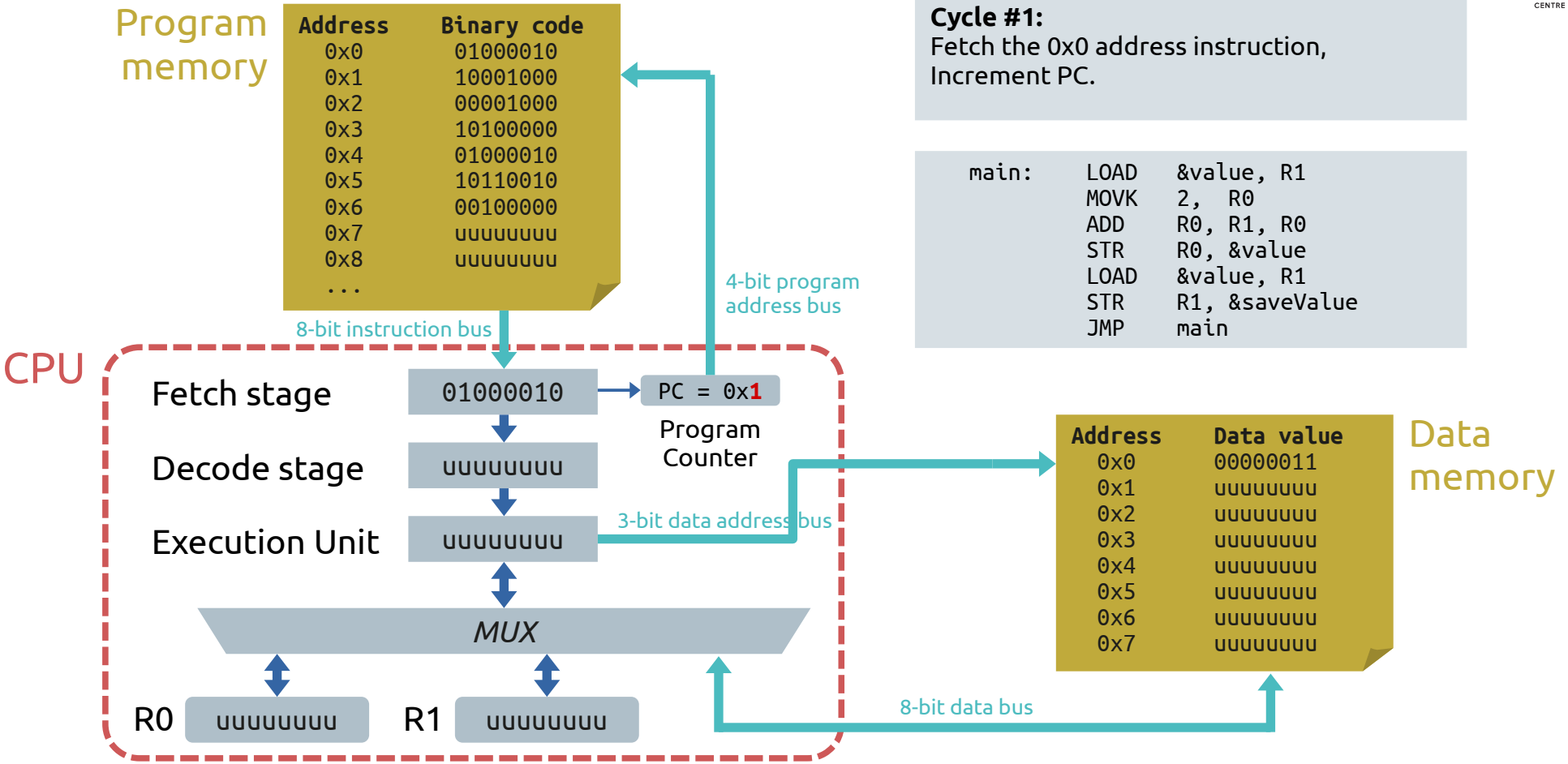


Processeur maison

Application starts

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

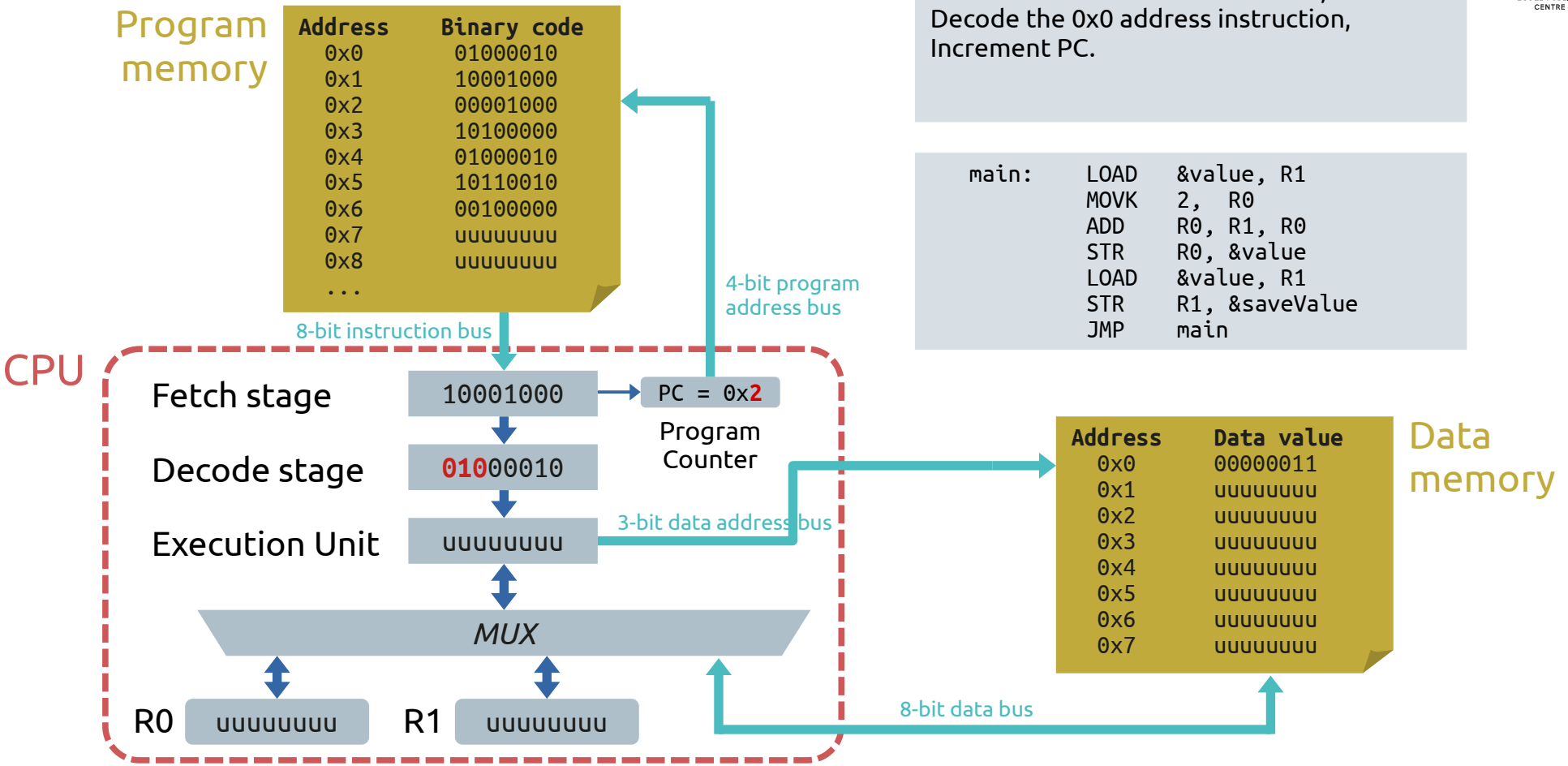
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



Processeur maison

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

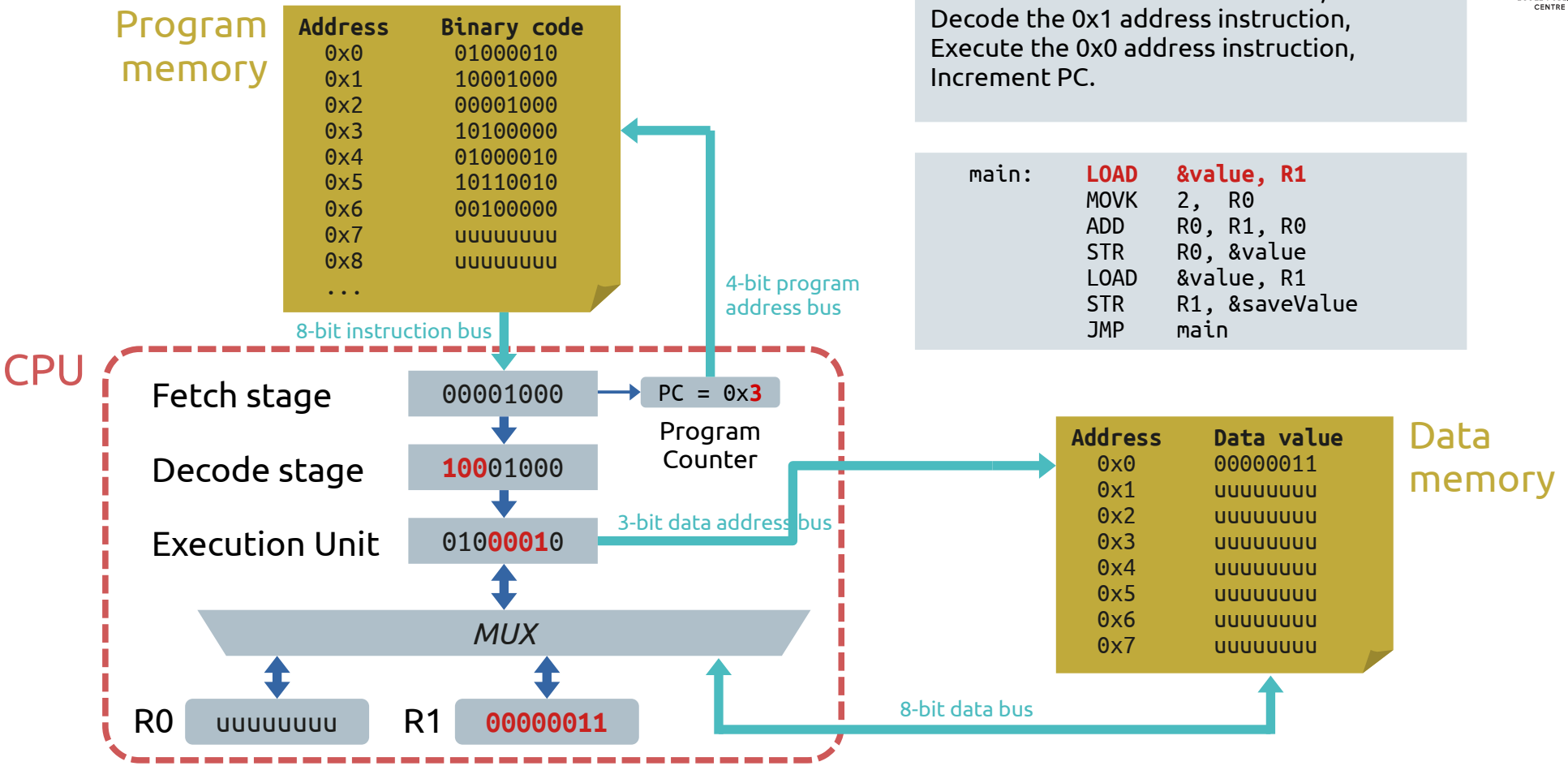
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



Processeur maison

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

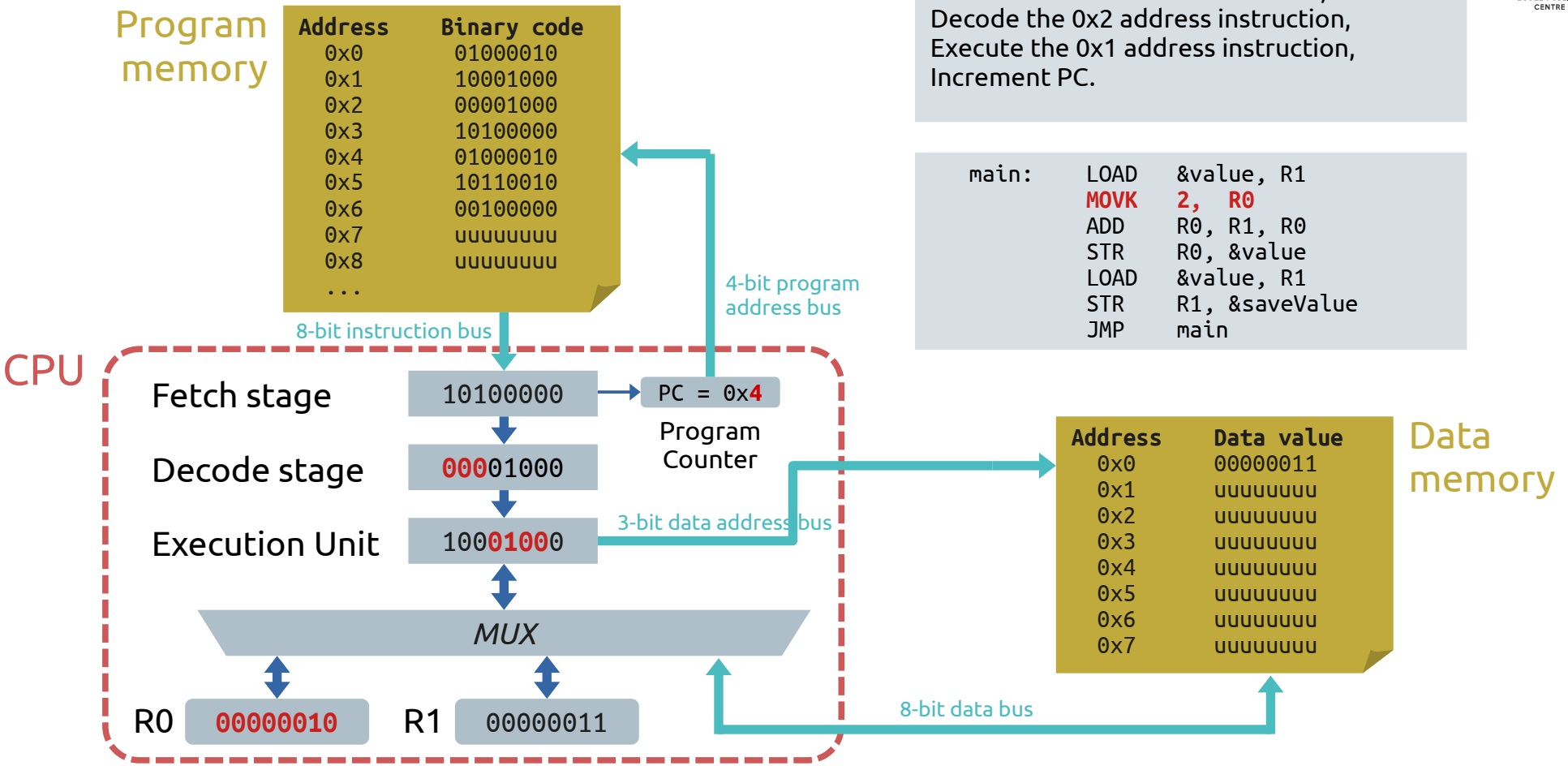
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



Processeur maison

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

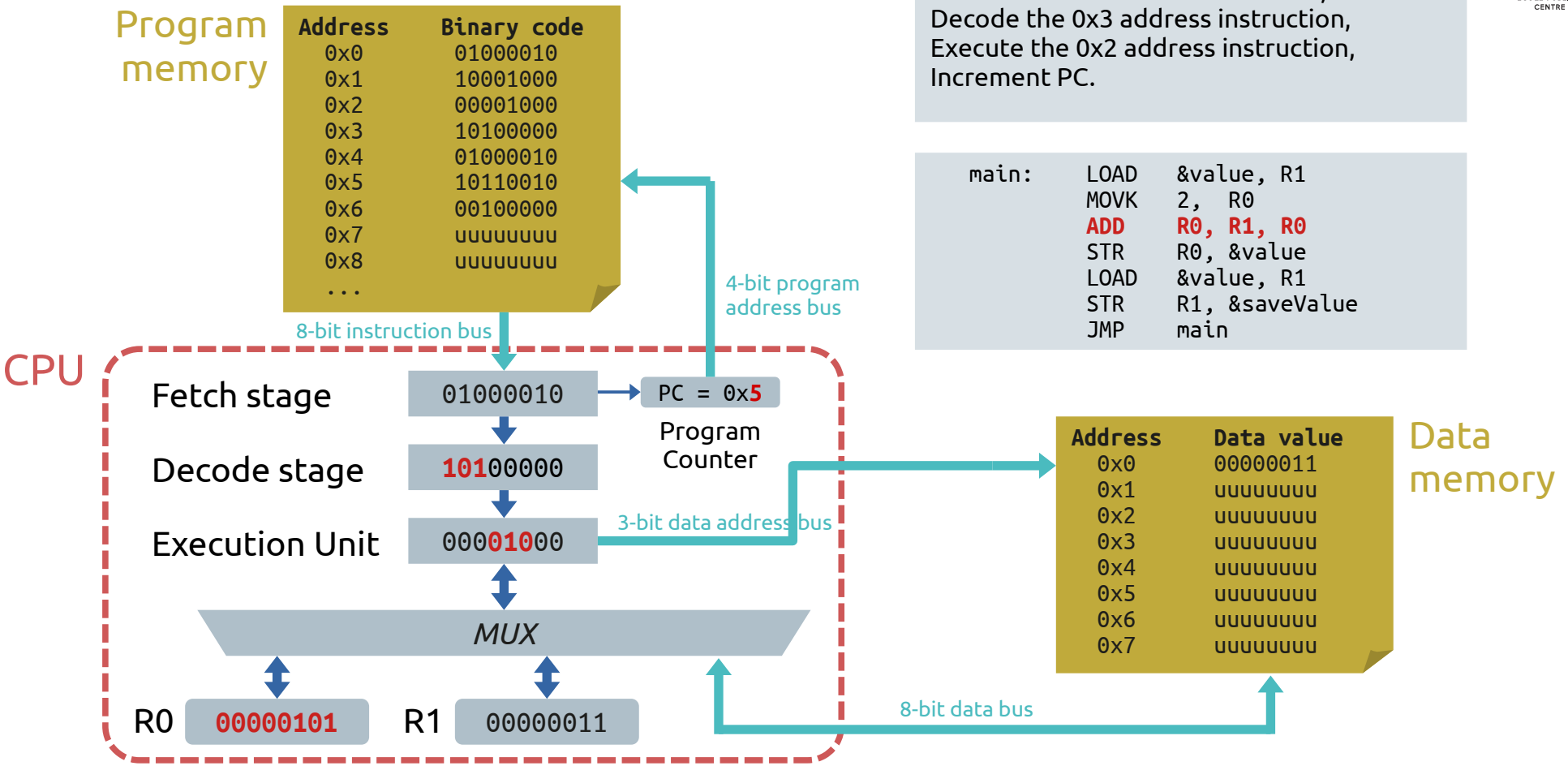
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



Processeur maison

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

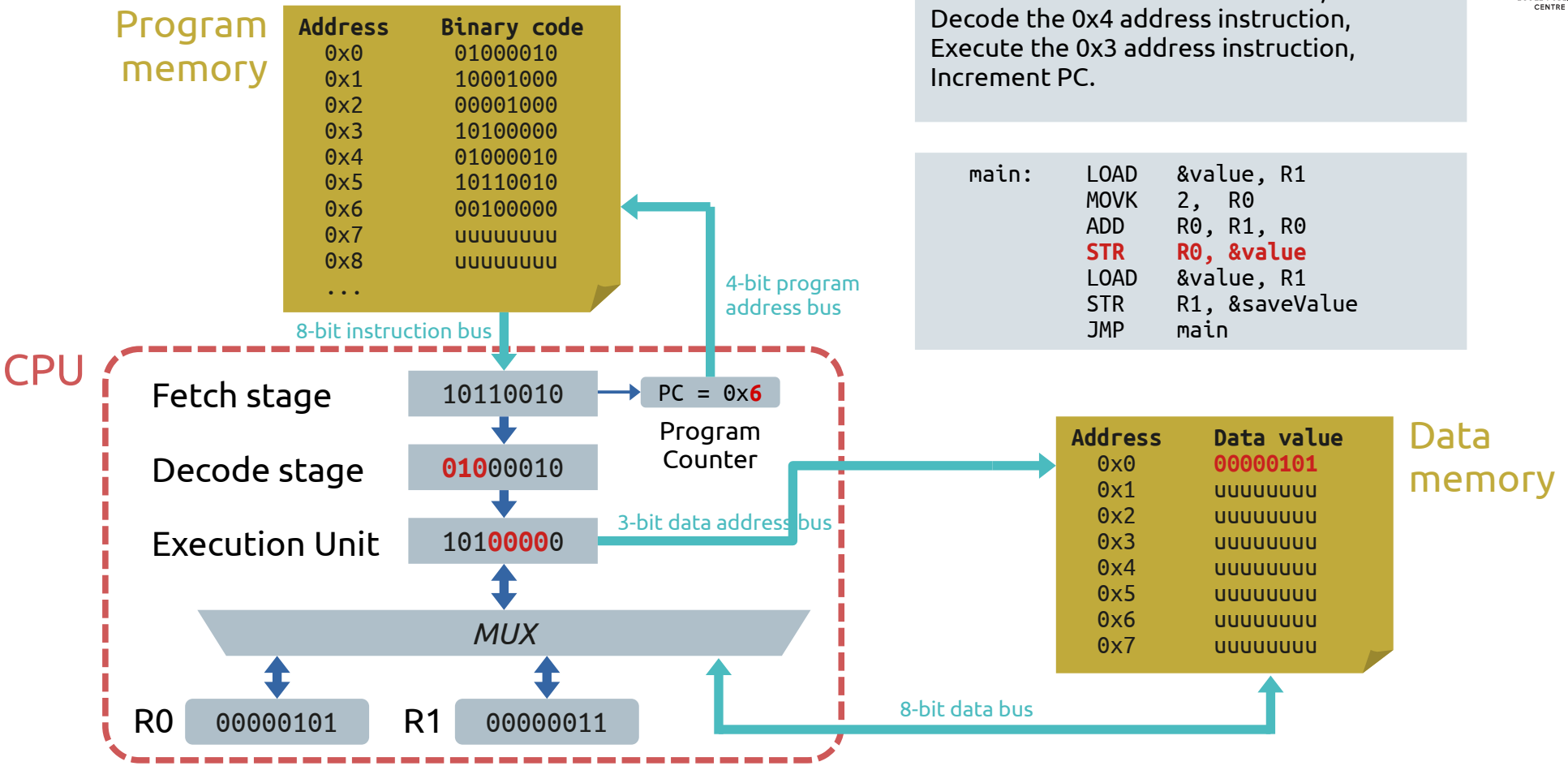
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



Processeur maison

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

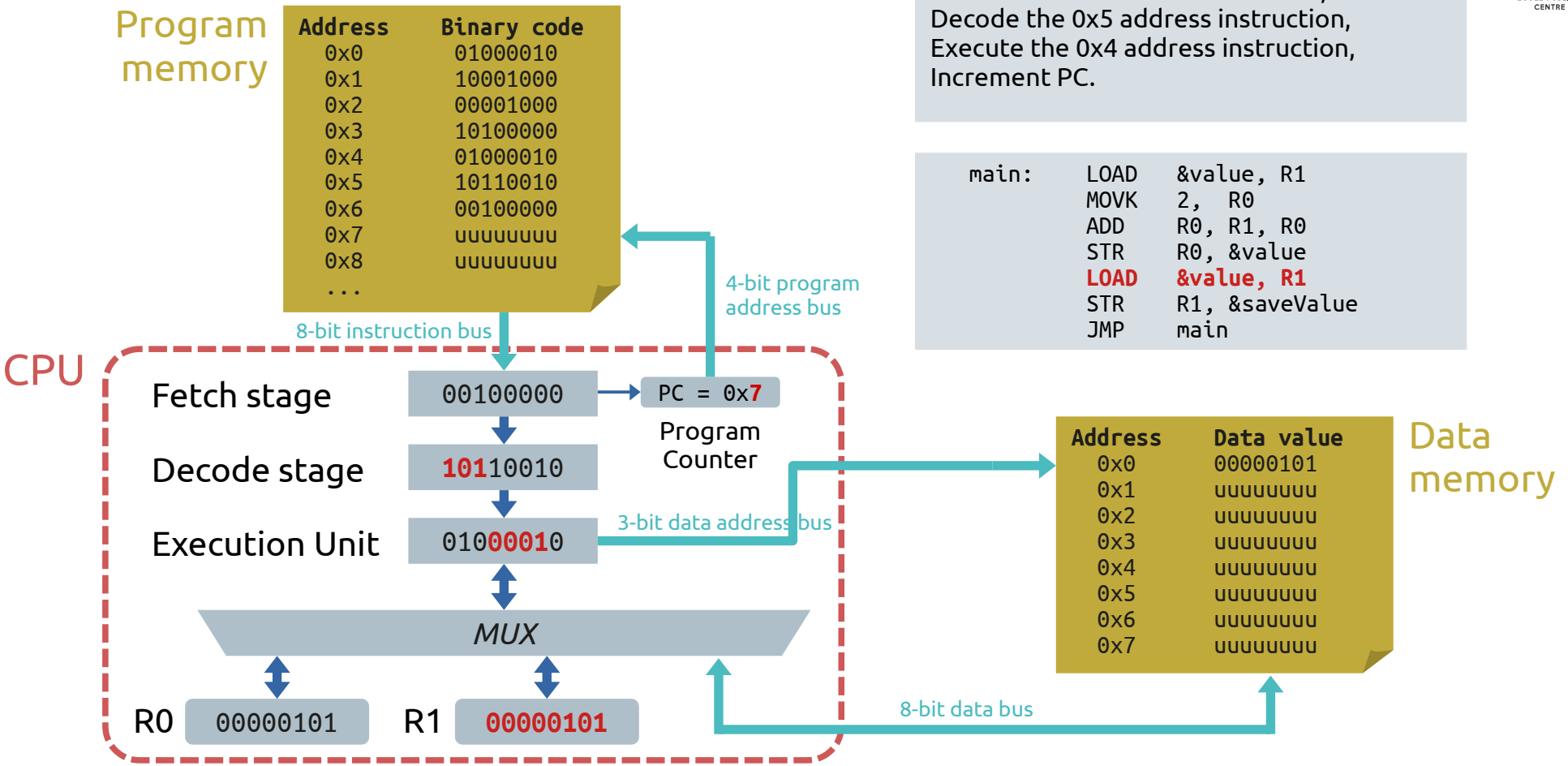
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



Processeur maison

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

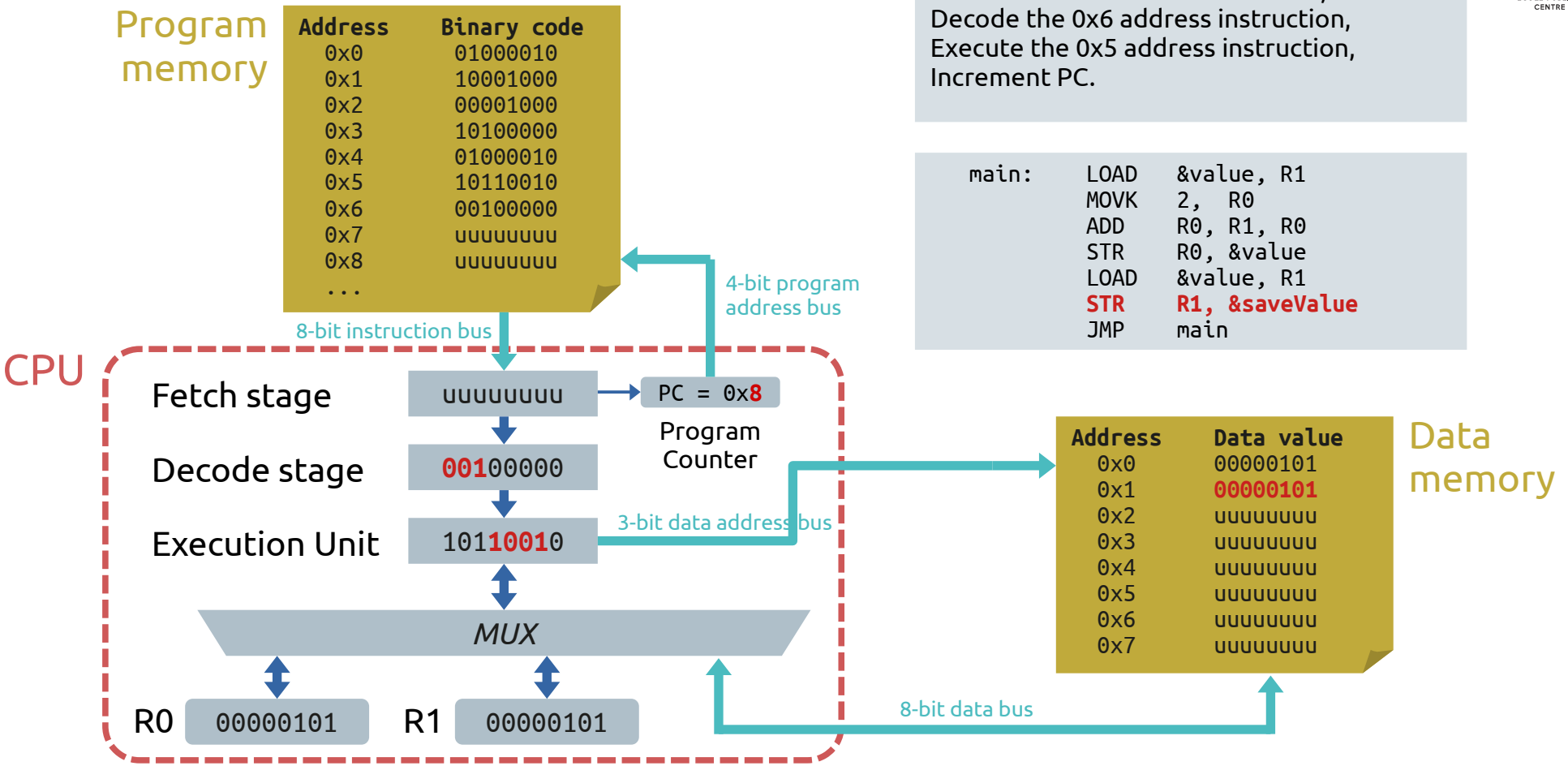
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



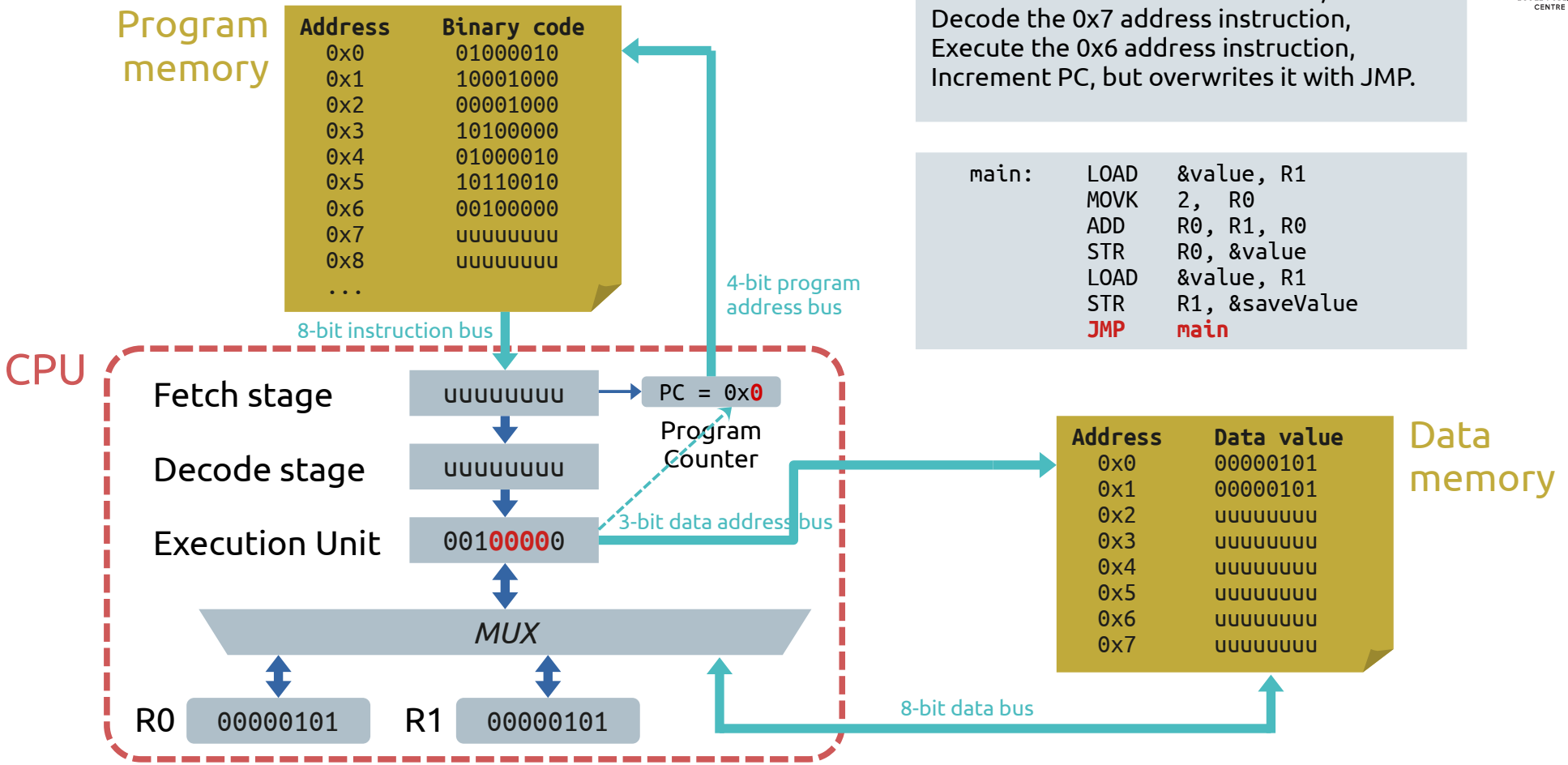
Processeur maison

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
```



Processeur maison



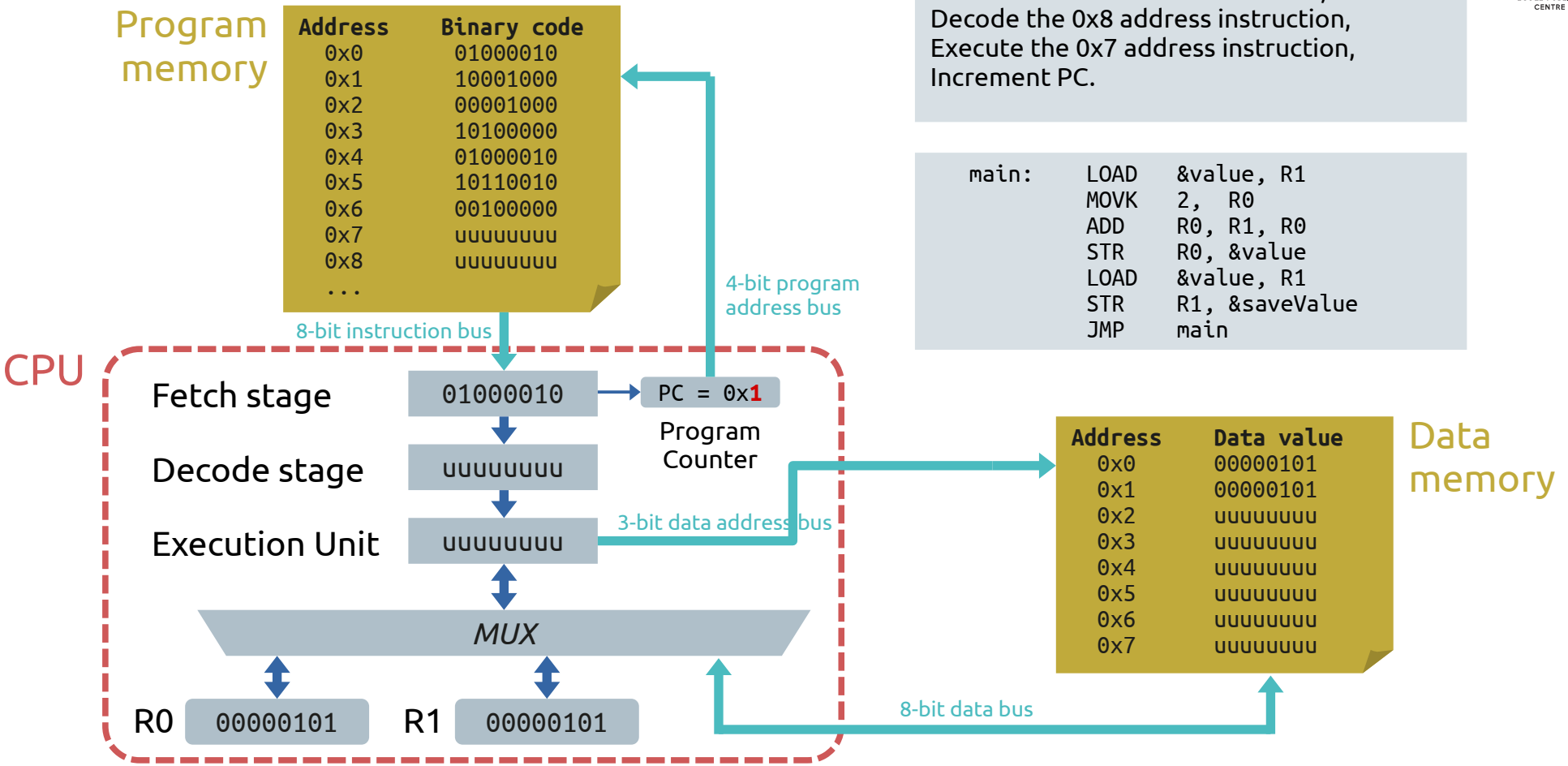
Cycle #9:
Fetch the 0x8 address instruction,
Decode the 0x7 address instruction,
Execute the 0x6 address instruction,
Increment PC, but overwrites it with JMP.

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

Processeur maison

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

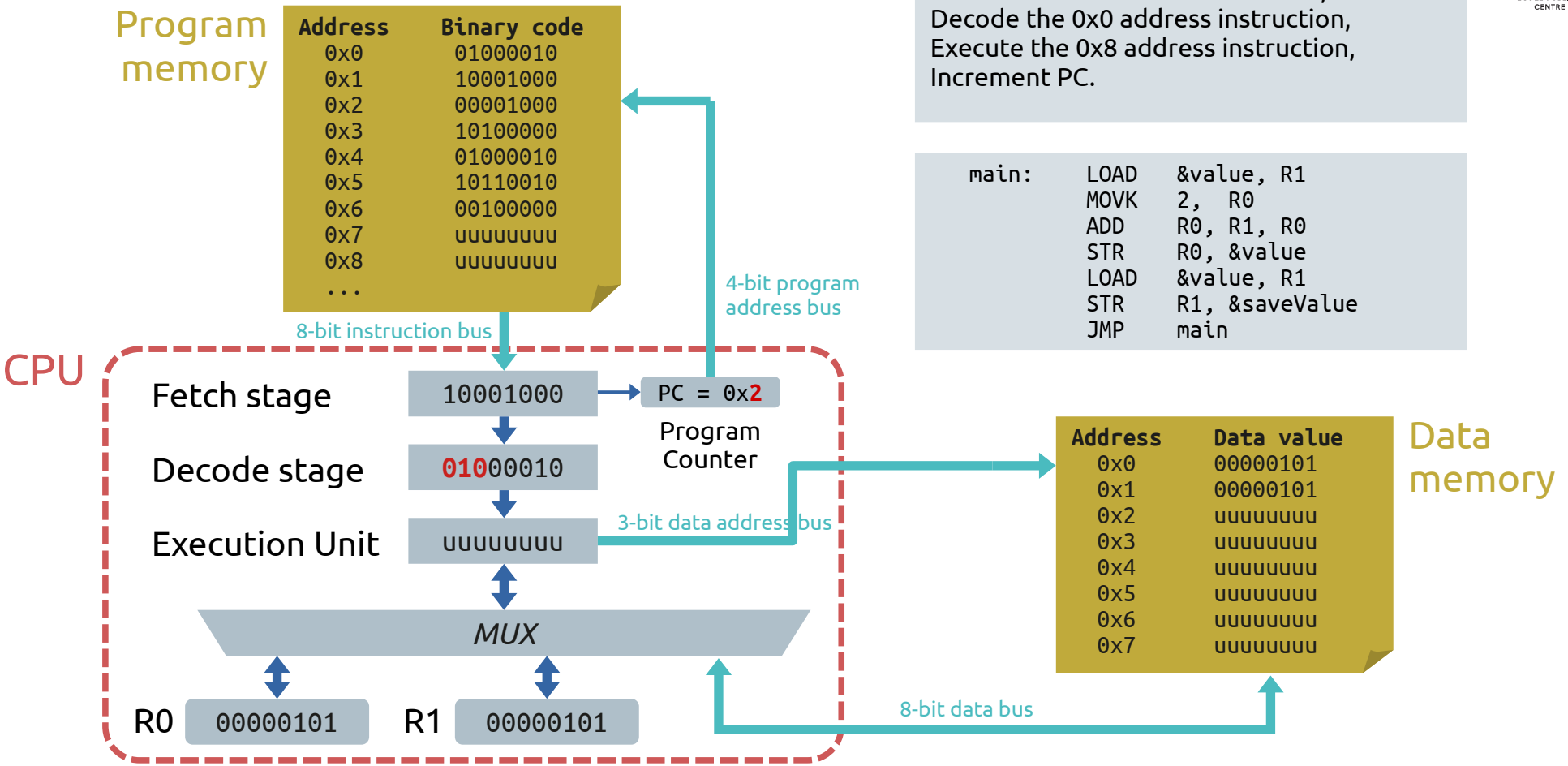
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



Processeur maison

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

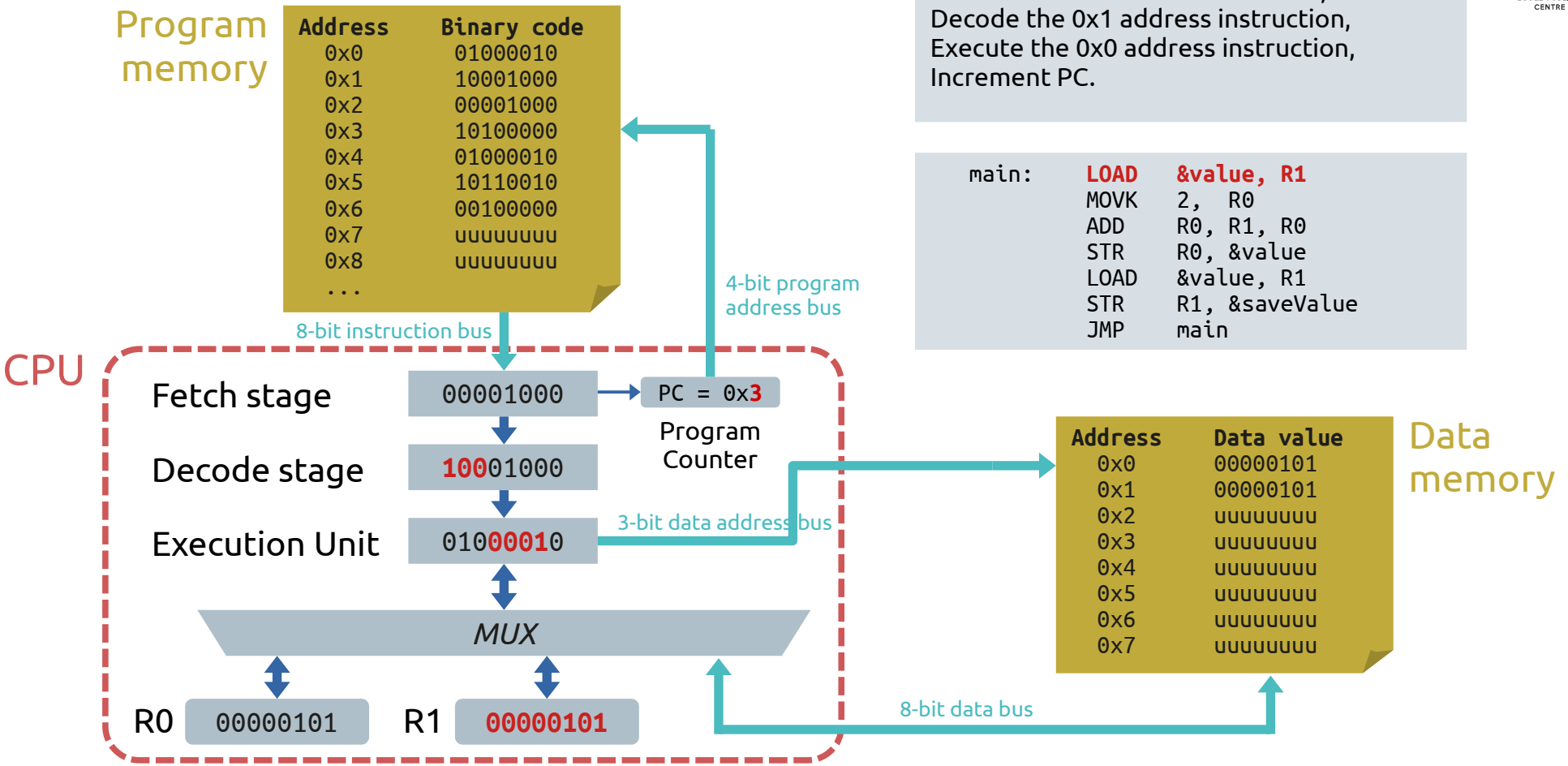
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```



Processeur maison

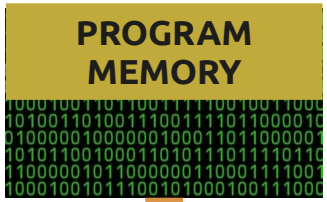
Cycle #12:
Fetch the 0x2 address instruction,
Decode the 0x1 address instruction,
Execute the 0x0 address instruction,
Increment PC.

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



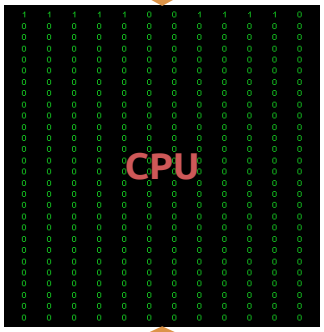
Et ça continue, encore et encore ...

Processeur



Persistent
EEPROM

INSTRUCTION BUS (Read only)

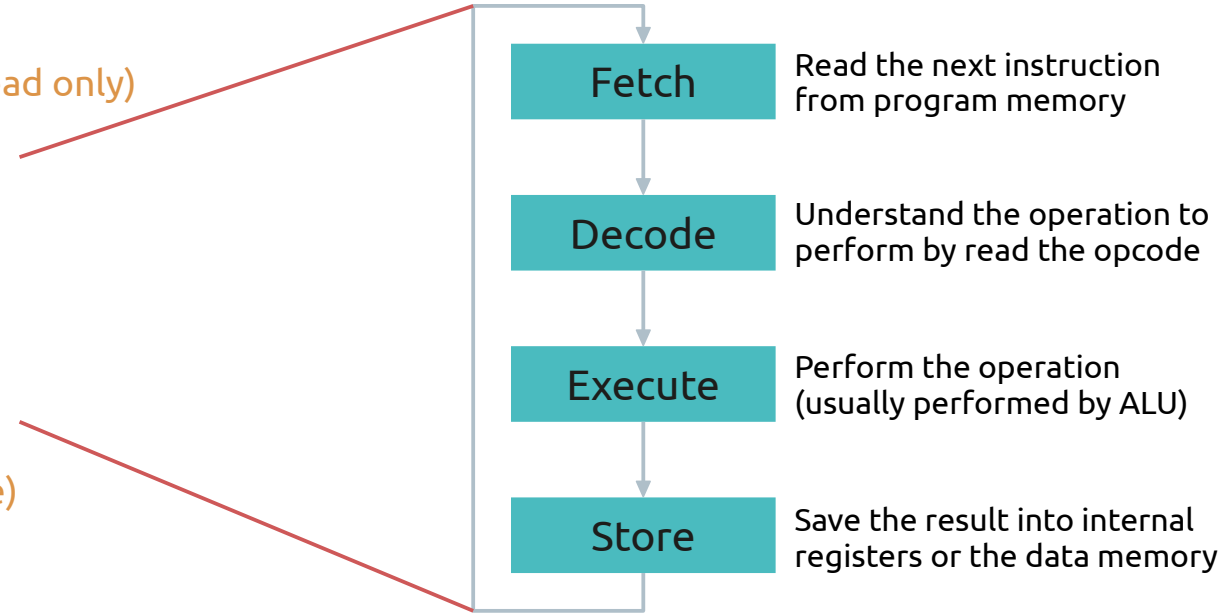


FETCH
DECODE
EXECUTE
WRITEBACK

DATA BUS (Read/Write)

Volatile
RAM

CPU CYCLE



ARCHITECTURE CPU

Harvard

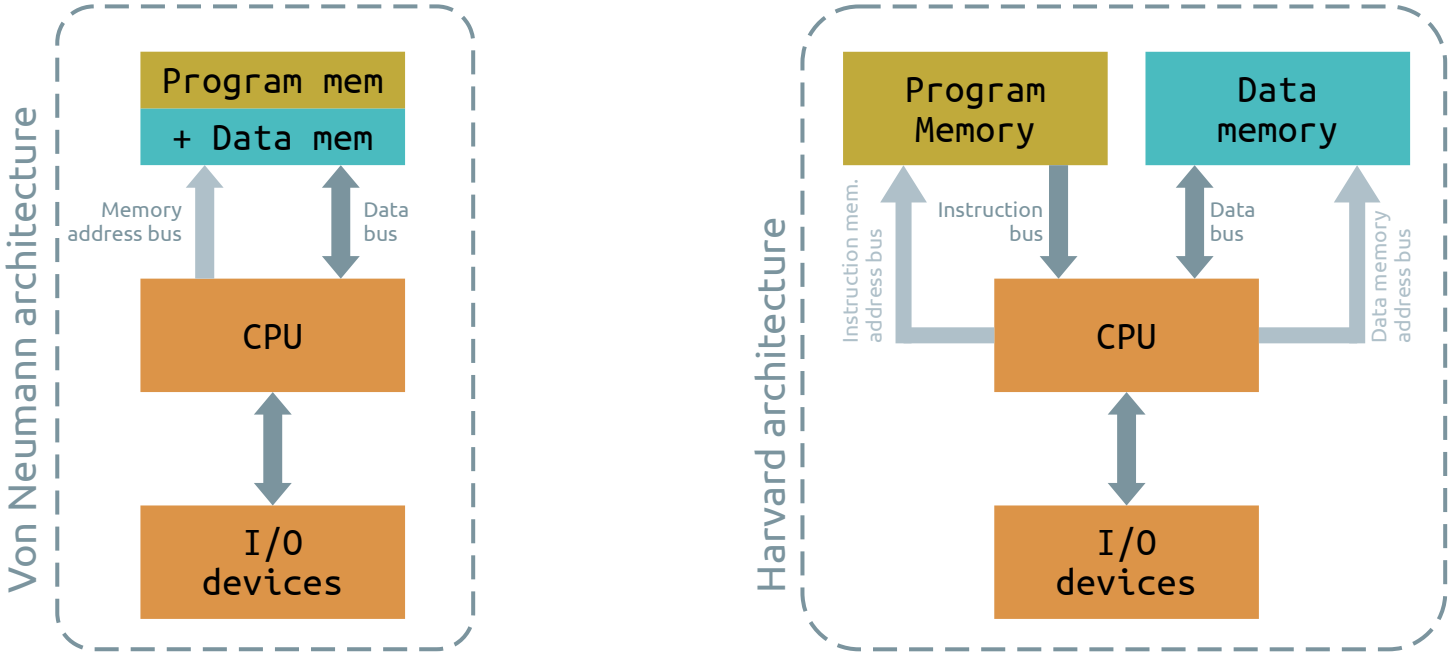
Von Neumann

Hybride

ENSI CAEN

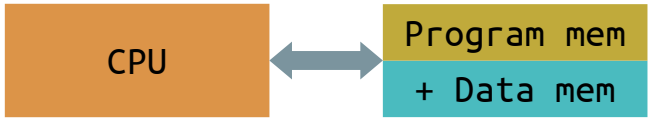
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

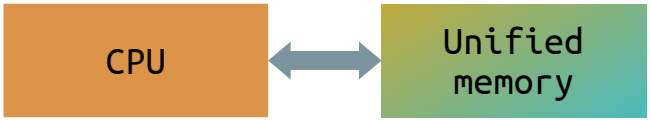
Un CPU peut posséder différents modèles d'interconnexion avec les mémoires.
Évidemment chaque modèle amène son lot d'avantages et d'inconvénients.



L'une des premières architectures rencontrées est celle de **Von Neumann** (1945).

Elle est définie par un plan d'adressage mémoire unifié (mêmes bus d'adresse et de données pour différentes mémoires), voire d'une mémoire réellement unifiée (programme et données stockées dans la même mémoire).





Ce modèle mémoire a été utilisé dans le Intel 8086. Aujourd'hui il a quasiment disparu dans sa forme « pure », mais il est encore très répandu dans sa forme « hybride » (vue plus loin).

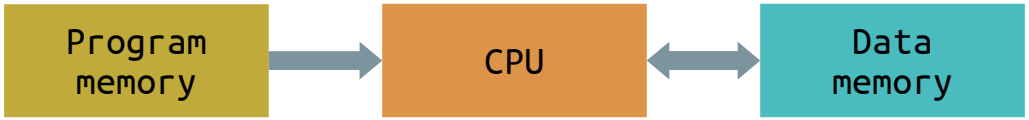
✓ Mapping mémoire unique (moins de matériel)

✓ Polyvalent si mémoire unifiée (avec un code large et peu de données, ou *vice versa*)

✗ Pipeline matériel difficile (fetch-decode-execute compliqué car les instructions sont mélangées aux données).

L'architecture de **Harvard** (1944) dispose de deux mémoires séparées.

Chaque mémoire (programme et données) dispose de son propre bus d'adresse et de son propre bus de données/instructions. La technologie, la taille des mémoires et la taille des bus sont donc distincts.



La forme pure du Harvard est encore utilisée, principalement dans les CPU low-power. On peut citer par exemple les Microchip PIC18, Atmel AVR, certains DSP, …

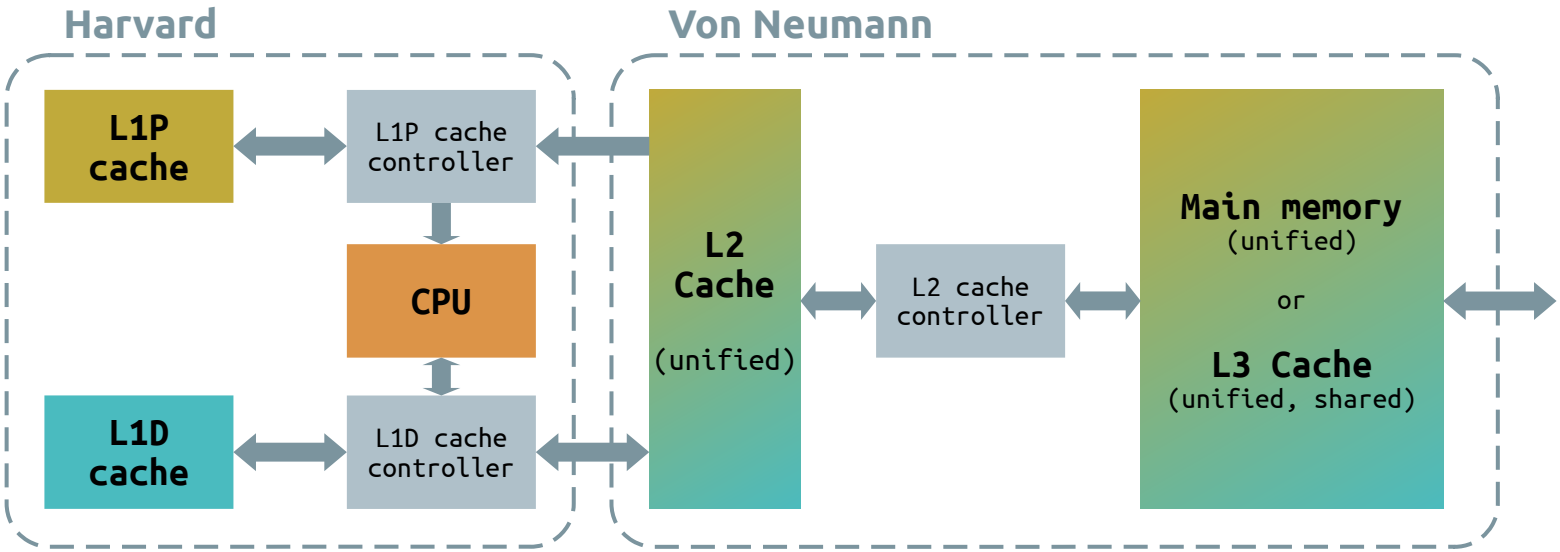
✓ Pipeline possible (Fetch en *program mem.*, Decode en CPU, Execute et Writeback en CPU ou *data mem.*)

Donc plus rapide qu'une architecture Von Neumann

✗ Peu polyvalent (choix de la référence de processeur correspondant aux besoins mémoires de l'application)

✗ Complexité matérielle (plus de bus, plus cher)

Le modèle de **Harvard modifié** cherche à allier les avantages des deux précédents.



Cette architecture se retrouve sur de nombreux processeurs modernes. On peut notamment citer les GPP Intel Core-i et AMD Ryzen, mais aussi les AP ARM Cortex-A, ou les DSP TI C6000.

Harvard modifié

L'architecture Harvard modifiée parvient à associer les avantages des modèles Harvard et Von Neumann grâce à l'utilisation de mémoires caches.

Le CPU n'a pas « conscience » de la présence de caches, et un développeur haut-niveau n'a pas besoin d'en connaître davantage. Toutefois un développeur bas-niveau adepte de l'optimisation devra assurer avec rigueur la cohérence des données (voir plus loin).

✓ Pipeline possible

Grâce à l'existence de la L1P (Layer 1 Program cache memory)

✓ Très polyvalent

Le code peut être bien plus lourd que les données (ou *vice-versa*) sans que cela ne pose de problème.

✗ Très grande complexité matérielle (plus de bus, plus de composants, plus cher)

Les différentes caches ont une très forte empreinte spatiale sur le silicium

Nécessité d'ajouter de la logique pour assurer les différents mécanismes liés à la cache (contrôleurs de cache)

MÉMOIRE CACHE

Ligne de cache

Avantages

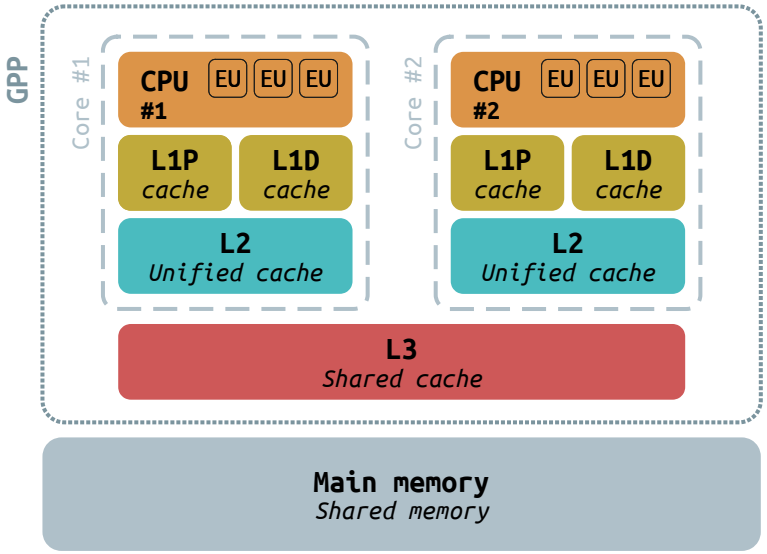
Cohérence des données

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

En informatique au sens large, une **mémoire cache** est chargée d'enregistrer temporairement des copies d'informations (données ou code) provenant d'une autre source (par opposition à une mémoire tampon qui ne réalise pas de copie).

Le cache peut être matériel (ce qu'on verra ici) ou logiciel (cache DNS, cache ARP, cache du navigateur, …). Sur processeur, le cache est matériel est hiérarchisé en différents niveaux (*layers*).



For 13th generation Intel Core:

Level	*Cache size*	*Line size*
L1D	32-48 KiB	64 B
L1P	32-64 KiB	64 B
L2	2-4 MiB	64 B
L3	36 MiB	64 B

https://www.intel.com/content/www/us/en/products/docs/processors/core/core-technical-resources.html

Mémoire principale et mémoire cache

Sur le principe, la mémoire principale (RAM) est divisée en blocs de taille fixe appelées **lignes de cache**. La taille d'une ligne dépend évidemment des technologies, par exemple 64 octets pour la 13e génération de Intel Core.

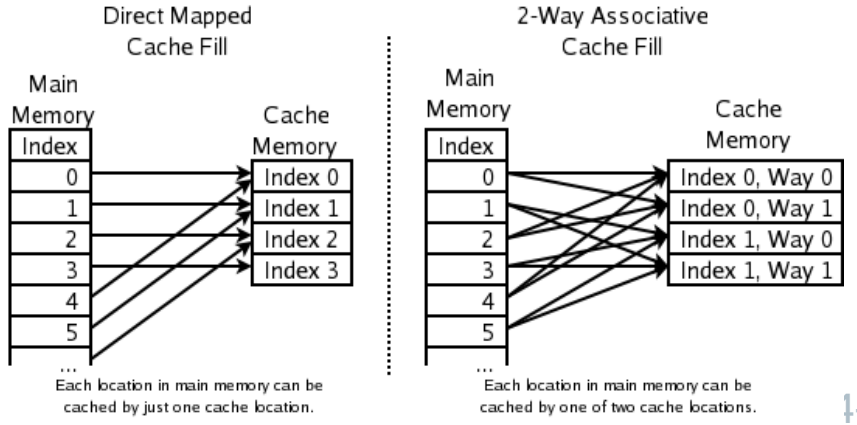
Lorsque le CPU cherche une information (une donnée ou une instruction), il consulte d'abord la mémoire cache :

✓ Si l'information est présente en cache (*cache hit*), elle est immédiatement disponible pour le CPU.

✗ Si l'information n'est pas présente en cache (*cache miss*), la mémoire principale est consultée : la ligne contenant l'information est rapatriée en cache et écrase une ligne existante.

Chaque ligne de la mémoire principale peut être associée (*mapped*) à une ligne de la cache.

Il existe différents modes d'association :
direct cache, *associative cache*,
n-way set associative cache (le plus courant).



Temps d'accès moyen à l'information

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

L'utilité de mémoires cache se justifie par deux principes :

- Principe de localité spatiale : soit une donnée récemment manipulée, les données dont l'adresse est proche de la première sont susceptibles d'être utilisées (ex : tableau)
- Principe de localité temporelle : soit une donnée récemment référencée, elle est susceptible d'être à nouveau référence dans un cours laps de temps (ex : variable)

Ainsi, en raison des temps d'accès à la mémoire principale (en comparaison à la vitesse de traitement de l'information par le CPU), il est judicieux de disposer de mémoires cache car leur temps d'accès est nettement plus faible.

Temps moyen d'accès à l'information : $t_{access} = c + (1 - h) \cdot m$

c = temps d'accès à la cache (~1 ns pour L1, ~5 ns pour L2)
m = temps d'accès à la mémoire (~100 ns)
h = taux de succès (cache hit) ou taux de présence en cache

On voit que le taux de succès (taux de *cache hits*) impacte très fortement les performances. Il est donc primordial de disposer d'une bonne stratégie de gestion de la cache.

https://colin-scott.github.io/personal_website/research/interactive_latency.html

Avant d'écraser une ligne de la cache pour en récupérer une nouvelle depuis la mémoire principale, il faut gérer deux problèmes :

Quelle ligne remplacer ?

Il serait dommage d'écraser une ligne qui est susceptible d'être réutilisée, et donc qu'il faudra à nouveau récupérer. Différents algorithmes implémentés matériellement existent. Le plus simple est le FIFO (la ligne qui a été chargée le plus longtemps est remplacée), ou plus efficace le RLU (*Least Recently Used*) qui supprime la ligne de cache qui a été utilisée le moins récemment.

La ligne contient-elle des données qui ont été modifiées ?

Le CPU travaille avec la cache, il va donc consulter des données mais aussi les modifier. Avant d'écraser une ligne de cache, il faut assurer la cohérence des données entre la cache et la mémoire principale. Encore une fois, il existe plusieurs stratégies, dont le *write-through* (écriture immédiate) et le *write-differed* (écriture différée).

Cohérence des données entre caches

Si on prend un peu de recul, on notera que les caches sont inclusifs :

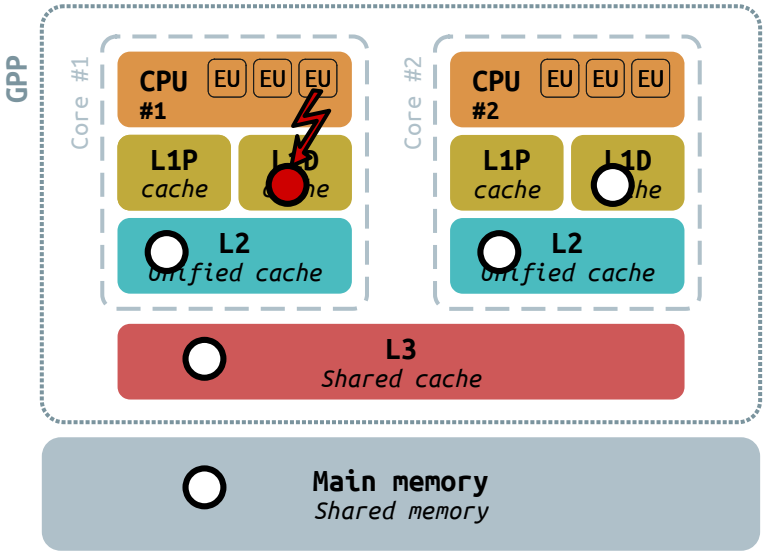
Tout le contenu de la L1 se trouve dans la L2, dont tout le contenu se retrouve dans la L3, dont le contenu total se retrouve en mémoire principale. Les informations existent donc en plusieurs exemplaires.

Dans le cas d'un GPP multi-cœurs, une même donnée va aussi se retrouver dans plusieurs caches L2 à la fois.

Si les cœurs #1 et #2 ont tous les deux besoins d'une même donnée, elle sera copiée de la mémoire principale vers la L3, puis vers les deux L2 puis vers les deux L1D.

Si le cœur #1 modifie cette donnée dans sa L1D et que le cœur #2 a besoin de la valeur à jour juste après, il faut s'assurer que la bonne valeur sera effectivement lue.

Le matériel doit être capable de gérer ces situations. Mais c'est aussi dans ce cas qu'un développeur bas-niveau (par exemple spécialisé dans la parallélisation d'algorithmes) doit être extrêmement rigoureux pour s'assurer de la cohérence des données afin de ne pas dégrader les performances normalement apportées par les caches.



Analyse des performances d'un algorithme

Certains outils (comme valgrind) peuvent analyser l'utilisation de la cache d'un programme, par exemple dans l'optique de comparer plusieurs algorithmes et choisir le plus efficace.

Ce même outil est également capable d'analyser le nombre de branchement réussis ou ratés.

```
dboudier:Correction$ valgrind --tool=cachegrind ./bin/Release/bench_vs_omp_for_i
==17307== Cachegrind, a cache and branch-prediction profiler
==17307== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==17307== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==17307== Command: ./bin/Release/bench_vs_omp_for_i
==17307==
--17307-- warning: L3 cache found, using its data for the LL simulation.
Seq. time (s):  58.542865
Par. time (s):  122.559042
Thread(s)     :  16
Speedup       :  0.477671
Efficiency    :  0.029854
==17307==
==17307== I   refs:      28,217,946,558
==17307== I1  misses:             1,871
==17307== LLi misses:             1,845
==17307== I1  miss rate:           0.00%
==17307== LLi miss rate:           0.00%
==17307==
==17307== D   refs:       7,391,805,031  (7,387,508,249 rd   + 4,296,782 wr)
==17307== D1  misses:       227,045,846  (  224,755,979 rd   + 2,289,867 wr)
==17307== LLd misses:       210,967,720  (  209,858,245 rd   + 1,109,475 wr)
==17307== D1  miss rate:            3.1% (          3.0%     +       53.3%  )
==17307== LLd miss rate:            2.9% (          2.8%     +       25.8%  )
==17307==
==17307== LL refs:          227,047,717  (  224,757,850 rd   + 2,289,867 wr)
==17307== LL misses:        210,969,565  (  209,860,090 rd   + 1,109,475 wr)
==17307== LL miss rate:             0.6% (          0.6%     +       25.8%  )
```

Analyse d'un algorithme de calcul condensé (très peu d'instructions) mais beaucoup de données. De plus l'algorithme est parallélisé (exécuté sur 16 cœurs logiques).

$28.2 \cdot 10^9$ instructions demandées.
Parmi celles-ci, 1871 n'étaient pas en L1i (L1 Instructions).
Parmi les 1871 absentes de la L1i,
1845 n'étaient pas en LL (Last Level Cache = L3).

$7.39 \cdot 10^9$ données demandées (rd = lecture + wr = écriture).
Parmi celles-ci, $227 \cdot 10^6$ n'étaient pas en L1d (L1 Data).
Parmi les $227 \cdot 10^6$ données absentes de la L1d,
$210 \cdot 10^6$ n'étaient pas en LL (Last Level Cache = L3).

Notes :
- la L1i et la L1d sont propres à chaque cœur physique
- la L2 n'est pas simulée ici
- la LL (= L3) est partagée entre tous les cœurs

EXCEPTIONS MATÉRIELLES

Interruptions

Exceptions

Signaux

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Vous avez sûrement déjà rencontré l'erreur « `Segmentation fault (core dumped)` ».

```
dboudier:tp1$ ./seg_fault
Segmentation fault (core dumped)
```

Plus précisément, il s'agit d'une **exception**.

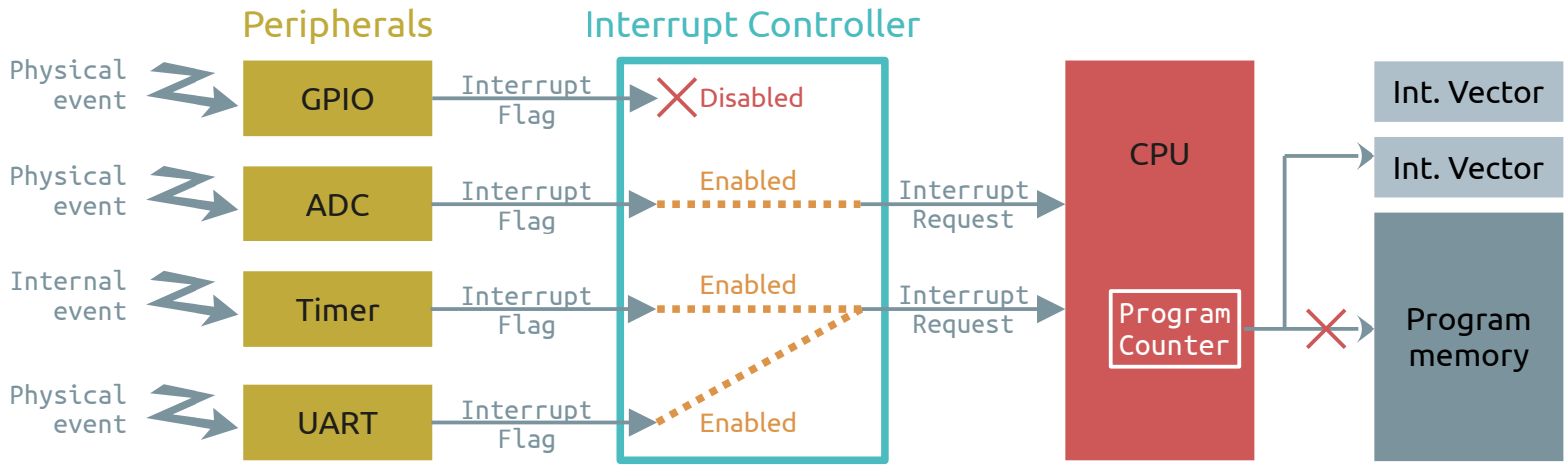
L'exception est un mécanisme matériel du CPU, assez proches des interruptions.

Mais là où les interruptions servent à la communication des périphériques vers le CPU, les exceptions sont plutôt associées à la sûreté d'exécution du code.

Lorsqu'une exception est levée, c'est au système d'exploitation de la gérer.

Pour rappel, une **interruption** est un mécanisme matériel. Elle survient suite à un évènement asynchrone (décorrélé de l'horloge du CPU) détecté par un **périphérique**.

Suite à la requête d'interruption (*Interrupt Request*, IRQ), le CPU stoppe l'exécution de son programme et traite à la place une routine d'interruption (*Interrupt Service Routine*, ISR). La routine d'interruption est une portion du programme située dans une zone spécifique de la mémoire programme (le vecteur d'interruption).

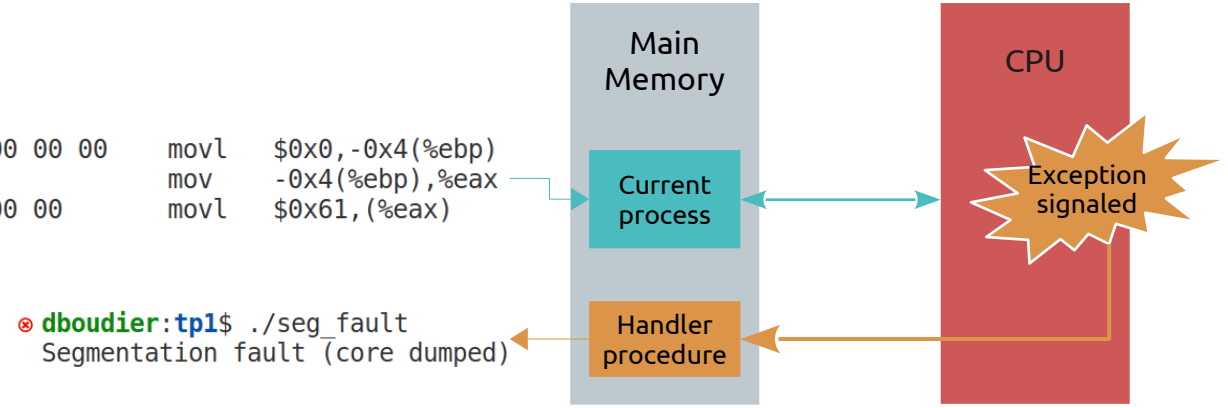


Une **exception** est aussi un mécanisme matériel. Elle survient de manière **synchrone** suite à la détection d'une anomalie par le **CPU**.

De la même manière que pour les interruptions, le CPU arrête le processus en cours d'exécution lorsqu'une exception est levée pour exécuter à la place une procédure dédiée au traitement de cette exception. L'adresse des procédures est fournie par le système d'exploitation au CPU grâce à une table. L'exception est donc gérée par l'OS dans un premier temps, mais il peut redonner la main au processus interrompu pour qu'il gère lui-même l'exception.

```
1  #include "stdio.h"
2
3  void main() {
4      int* a = NULL;
5      *a = 97;
6  }
```

```
c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
8b 45 fc                mov    -0x4(%ebp),%eax
c7 00 61 00 00 00       movl   $0x61,(%eax)
```




```
dboudier:tp1$ ./seg_fault
Segmentation fault (core dumped)
```

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Liste des interruptions et exceptions gérées par un Intel Core.

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[2]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[3]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[4]
20	#VE	Virtualization Exception	Fault	No	EPT violations[5]
21	#CP	Control Protection Exception	Fault	Yes	RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump.
22-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

« Intel 64 and IA-32 Architectures Software Developer's Manual »
Volume 3 – System Programming Guide
Chapter 6, Interrupt and Exception Handling
Intel, June 2023

Exception : Fault, Abort et Trap

Sur architecture Intel, les exceptions sont répertoriées en 3 grandes classes :

Fault : Lorsqu'une exception de ce type arrive, elle peut en général être corrigée et peut potentiellement autoriser la poursuite d'exécution du programme ayant causé le défaut (dépend de la stratégie de l'OS). Sous Linux, si un défaut de ce type est détecté, le kernel prend l'initiative d'envoyer un **signal** (`SIGSEGV`, `SIGFPE`, `SIGILL`, `SIGBUS`) au processus à la cause du défaut. Sinon par défaut, le processus est alors mis à mort.

(la notion de signal est abordée par la suite).

Abort : Défaut critique pour le système, le processus en cause n'est pas autorisé à reprendre la main

Trap : Ce type d'exception n'est pas un défaut matériel, prenons l'exemple de l'exception `#BP` ou `Break Point`. Il s'agit, dans le cas présent, d'un opcode de 1 octet (instruction breakpoint = `INT3`) remplaçant le premier octet de l'opcode de chaque instruction du programme sous test. Ce type d'exception peut être utilisé comme alternative par les outils de debug (signal `SIGTRAP`). En effet, le debugger sera alors appelé à l'exécution de chaque instruction.

```c
#include <stdio.h>

int main(void)
{
    __asm("int3");
    return 0;
}
```

```
000011ad <main>:
    11ad:       f3 0f 1e fb             endbr32
    11b1:       55                      push   %ebp
    11b2:       89 e5                   mov    %esp,%ebp
    11b4:       e8 0d 00 00 00          call   11c6 <__x86.get_pc_thunk.ax>
    11b9:       05 23 2e 00 00          add    $0x2e23,%eax
    11be:       cc                      int3
    11bf:       b8 00 00 00 00          mov    $0x0,%eax
    11c4:       5d                      pop    %ebp
    11c5:       c3                      ret
```

```
dboudier:tp1$ ./sig_trap
Trace/breakpoint trap (core dumped)
```

Suite à une exception matérielle, l'OS reprend la main à la place du processus en cause.

Par exemple, un défaut de page sous Linux (#PF) est traité par la procédure `handle_page_fault()` écrite dans le fichier `arch/<cpu>/mm/fault.c` (https://kernel.org/).

Toutefois l'OS peut proposer au processus coupable de traiter le problème en lui indiquant à l'aide d'un **signal**. C'est une notification logicielle asynchrone, envoyée par le kernel à un processus (ou à un thread) suite à un évènement matériel ou logiciel.

Si le processus ne gère pas le problème, l'OS applique alors une procédure par défaut.

Les appels système `kill` et `signal` permettent respectivement d'envoyer un signal (évènement logiciel) et de capturer un signal (évènement matériel ou logiciel) depuis un processus.

Le noyau Linux supporte les signaux suivants :

```
dboudier:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

SIGFPE

Détection d'opérations arithmétiques erronées, par exemple division par zéro, valeur dé-normalisée, *overflow* ou *underflow* arithmétique … (`#XM SIMD Floating-Point Exception`, `#MF x87 FPU Floating-Point Error`, …).

Exemple ici d'une exception `#MF Math Fault` suite à une erreur arithmétique (division par 0).

```
void main(void) {
    int value = 3;
    int zero = 0;
    value /= zero;
}
```

```
dboudier:example_code$ ./math_fault
Floating point exception (core dumped)
```

Interrupt 16—x87 FPU Floating-Point Error (#MF)

Exception Class **Fault.**

Description

Indicates that the x87 FPU has detected a floating-point error. The NE flag in the register CR0 must be set for an interrupt 16 (floating-point error exception) to be generated. (See Section 2.5, "Control Registers," for a detailed description of the NE flag.)

NOTE

SIMD floating-point exceptions (#XM) are signaled through interrupt 19.

While executing x87 FPU instructions, the x87 FPU detects and reports six types of floating-point error conditions:

- Invalid operation (#I)
  - Stack overflow or underflow (#IS)
  - Invalid arithmetic operation (#IA)
- Divide-by-zero (#Z)
- Denormalized operand (#D)
- Numeric overflow (#O)
- Numeric underflow (#U)
- Inexact result (precision) (#P)

« Intel 64 and IA-32 Architectures Software Developer's Manual »
Volume 3 – System Programming Guide
Chapter 6, Interrupt and Exception Handling
Intel, June 2023

SIGBUS

Erreur sur le bus physique.

Par exemple détection de défauts d'alignement (exception `#AC Alignement Check Exception`) ou d'adresses physiques invalides (exception `#MC Machine-Check`, dépendant de l'architecture CPU).

Ci-contre un exemple de défaut d'alignement (exception `#AC Alignement Check Exception`).

(activation matérielle nécessaire côté processeur)

```
dboudier:example_code$ ./sigbus
Bus error (core dumped)
```

```c
#include <stdlib.h>

void main(void) {
    int *pInt;
    char *pArea;

    // Enable alignement checking
    #if defined(__GNUC__)
    #if defined(__i386__)
        __asm__("pushf\n orl $0x40000, (%esp)\n popf");
    #elif defined(__x86_64__)
        __asm__("pushf\n orl $0x40000, (%rsp)\n popf");
    #endif
    #endif

    // malloc() always provides aligned memory
    pArea = (char*) malloc(sizeof(int)+1);

    // Making the pointer misaligned (not modulo sizeof(*pInt))
    pInt = (int*) (pArea += 3);                 // NOK
    //pInt = (int*) (pArea += sizeof(*pInt));   // OK

    // Unaligned access (dereferenced pointer)
    *pInt = 51;
}
```

SIGSEGV

Signal générant le fameux « `Segmentation fault (core dumped)` ».

Parmi les quelques exceptions matérielles pouvant en être à la source, on illustre ici le `#PF Page fault` : page non présente en mémoire physique, exécution d'une page non-exécutable, …

Ici, erreur de segmentation, probablement `#PF Page Fault`.

```c
#include <stdio.h>

void main(void) {
    char* ptr = 0x12345678;     // Random address

    printf("Pointer is located there: %p\n", &ptr);
    printf("Pointer points to there:  %p\n", ptr);

    *ptr = 'H';
}
```

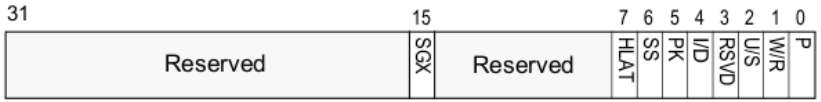
```
dboudier:example_code$ ./page_fault
Pointer is located there: 0xffa041b8
Pointer points to there:  0x12345678
Segmentation fault (core dumped)
```

SIGSEGV

Avec le mécanisme de pagination (voir chapitre sur la mémoire), une translation d'adresse linéaire invalide peut lever une exception matérielle côté CPU (`Page Fault`) dans deux cas de figure :

Translation d'adresse invalide ou Droits d'accès à la page cible invalides .

Ci-contre se trouvent les différents codes d'erreur pour l'exception matérielle `#PF Page Fault`.



P	0	The fault was caused by a non-present page.
	1	The fault was caused by a page-level protection violation.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
U/S	0	A supervisor-mode access caused the fault.
	1	A user-mode access caused the fault.
RSVD	0	The fault was not caused by reserved bit violation.
	1	The fault was caused by a reserved bit set to 1 in some paging-structure entry.
I/D	0	The fault was not caused by an instruction fetch.
	1	The fault was caused by an instruction fetch.
PK	0	The fault was not caused by protection keys.
	1	There was a protection-key violation.
SS	0	The fault was not caused by a shadow-stack access.
	1	The fault was caused by a shadow-stack access.
HLAT	0	The fault occurred during ordinary paging or due to access rights.
	1	The fault occurred during HLAT paging.
SGX	0	The fault is not related to SGX.
	1	The fault resulted from violation of SGX-specific access-control requirements.

« Intel 64 and IA-32 Architectures Software Developer's Manual »
Volume 3 – System Programming Guide
Chapter 6, Interrupt and Exception Handling
Intel, June 2023

SIGSEGV

Signal générant le fameux « `Segmentation fault (core dumped)` ».

Ce signal peut aussi être amené par des exceptions `#GP General Protection`, qui relèvent de nombreux défauts principalement associés à des accès mémoire illégaux : écriture sur segment *read-only*, lecture d'un segment *execute-only*, dépassement taille limite de segment, violation de privilège, *null segment selector*, …

Ici, une exception se lève suite à une tentative d'écriture en zone lecture-seule (`#GP General Protection`).

```
void main(void) {
    char *str = "Donkey";

    *str = 'M';
}
```

```
⊗ dboudier:example_code$ ./generalprotection_fault
  Segmentation fault (core dumped)

● dboudier:example_code$ objdump -s generalprotection_fault
Contents of section .rodata:
 2000 01000200 446f6e6b 657900                    ....Donkey.
```

SIGSEGV

Signal générant le fameux « `Segmentation fault (core dumped)` ».

Cependant l'exception la plus connue reste le `Stack Overflow`. En mode réel (un mode particulier de fonctionnement des CPU Intel), celui-ci est notamment détecté par l'exécution d'instructions des familles `PUSH` et `POP` capables de lever l'exception `#SS (Stack Fault)` en cas de dépassement de limite du *Stack Segment*. Dans les autres modes mémoire, il est en général détecté par exception matérielle `#PF (Page Fault)`, la pile étant de taille multiple de la taille d'une page mémoire. L'instruction `PUSH` est également capable de lever l'exception `#PF` (hors mode réel).

Erreur *stack overflow* suite à l'appel d'une fonction récursive.

```
void main(void) {
    main();
}
```

```
dboudier:example_code$ ./stackoverflow
Segmentation fault (core dumped)
```

ARCHITECTURES x86

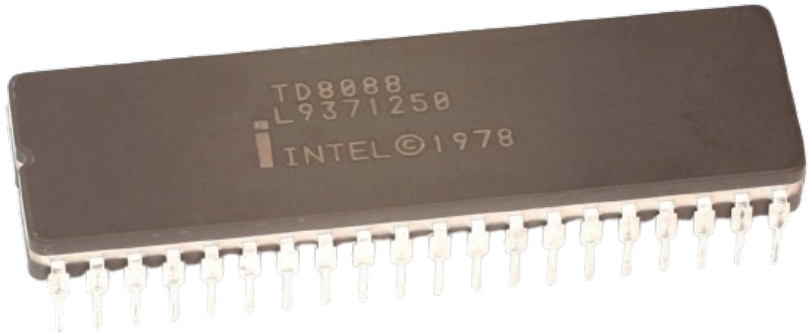
Intel 8086

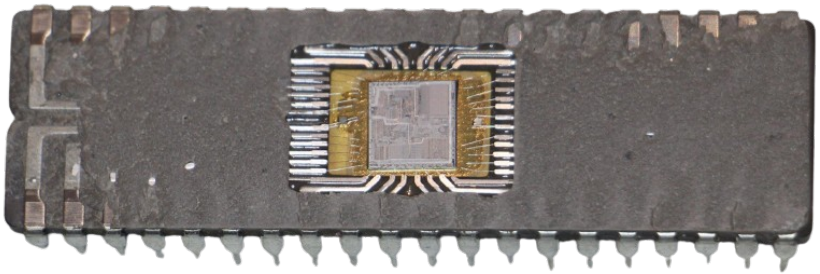
Intel 64 and IA-32

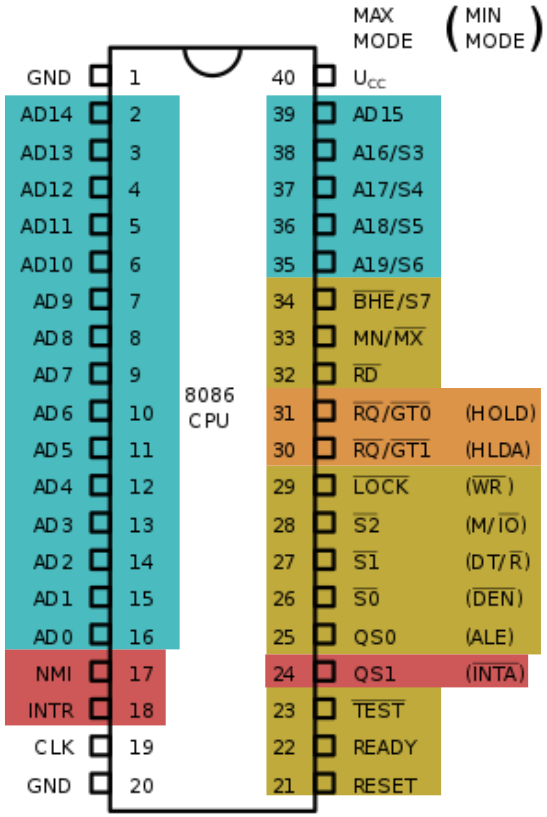
ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Observons le processeur Intel 8086 (1978), ancêtre commun des architectures x86.







Bus d'addresse

Bus de contrôle

Interruptions

Direct Memory Access

Unité d'exécution

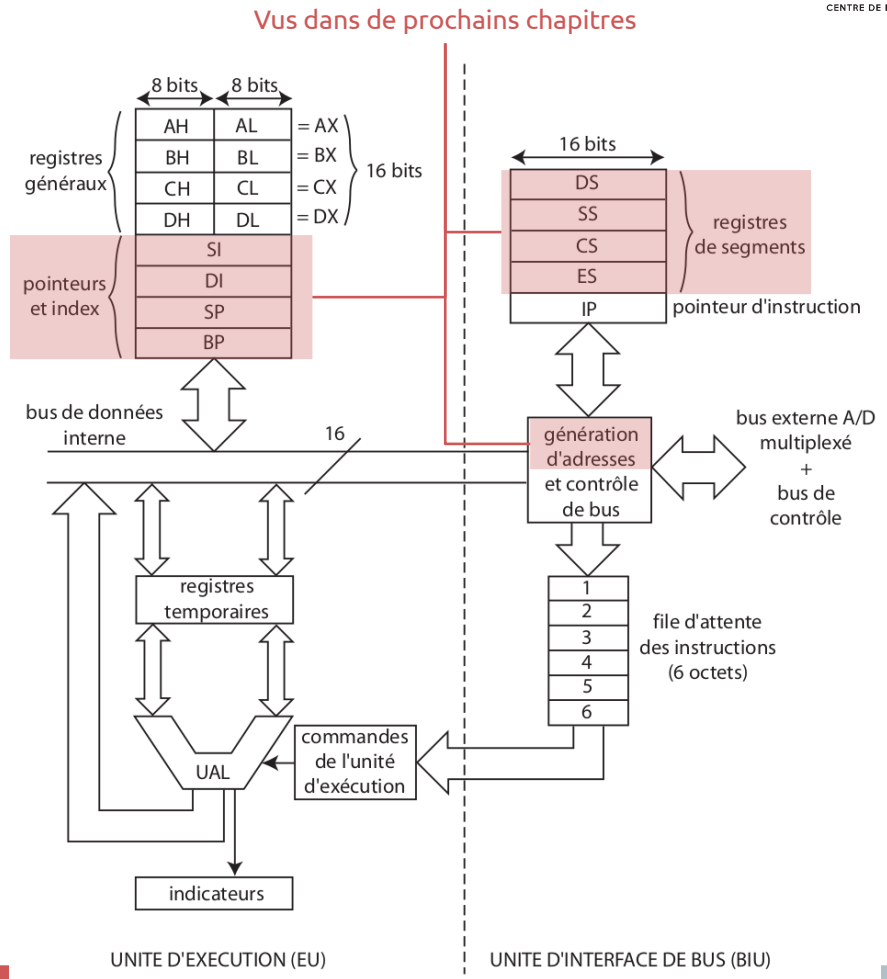
Décode puis exécute les instructions présentes dans la file d'attente

Unité d'interface de bus

Contrôle des bus pour les accès mémoire.

Calcul des adresses physiques (segmentation).

Gestion de la phase de Fetch via le registre pointeur d'instruction IP.



16-bit general-purpose registers

Registres généralistes 16-bit, mais il est possible de référencer un octet uniquement (par ex : AX = AH:AL)

Instruction pointer

Contient l'adresse de prochaine instruction à Fetch.

Arithmetic Logic Unit

Unité de calcul, chargée des opérations élémentaires

Flags

Indicateurs associés à l'unité de calcul

				O	D	I	T	S	Z		A		P		C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

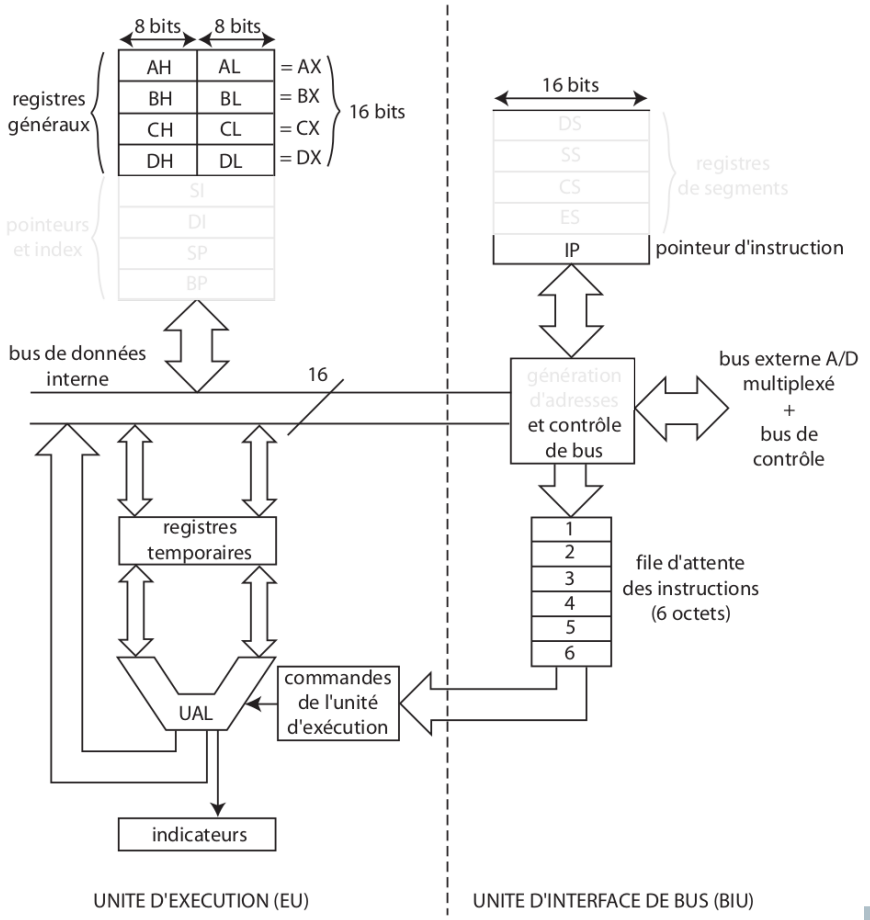
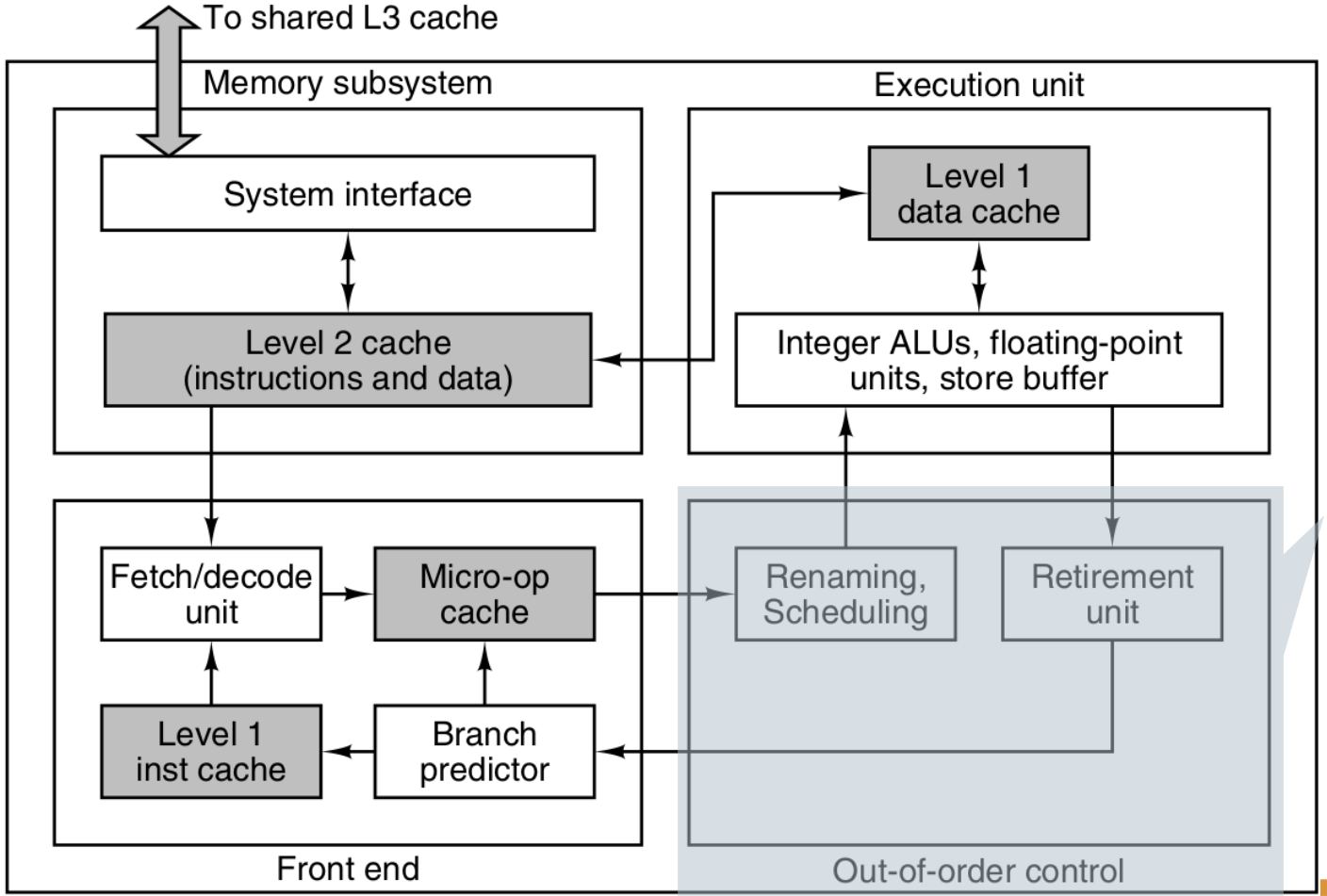


Schéma-bloc de l'Intel Core-i7 Sandy Bridge



Intel, toujours leader du marché des GPP, a bâti ses nouveaux processeurs tout en conservant l'héritage du 8086.

Voici l'exemple d'un Intel Core-i7 utilisant l'architecture x86-64 (ou AMD64, ou encore x64). On y voit notamment les évolutions liées aux caches et prédiction de branchement.

Partie non-vue ici mais typique d'un processeur superscalaire :

À cause des dépendances de données entre instructions exécutées en parallèle, une partie du matériel œuvre à la détection des dépendances et si besoin exécute une autre instruction à la place en attendant que la donnée soit enfin disponible et à jour.

« Structured Computer Organization »
6th edition, 2013.
Andrew Tanenbaum and Todd Austin.

Schéma fonctionnel du pipeline matériel des Intel Core

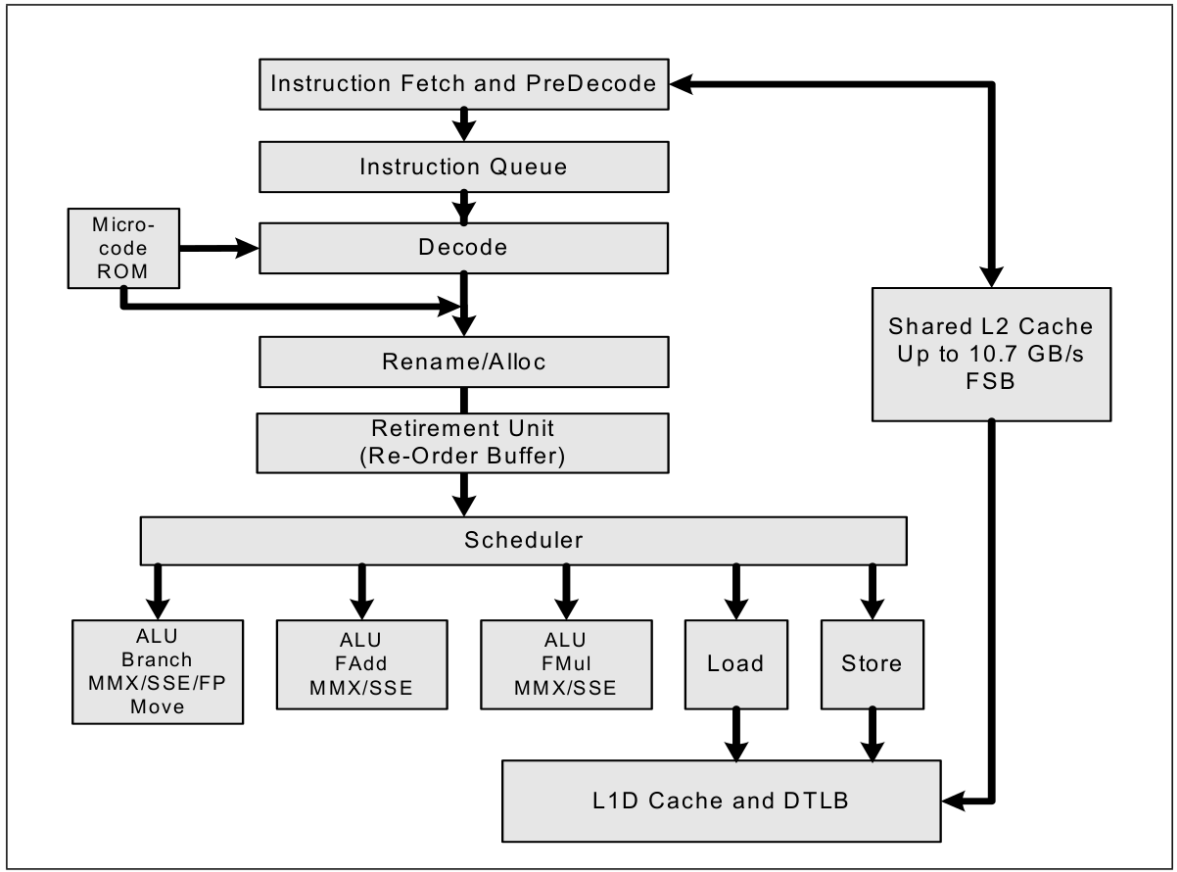


Figure 2-3. The Intel® Core™ Microarchitecture Pipeline Functionality

« Intel 64 and IA-32 Architectures Software Developer's Manual »
Volume 1 – Basic Architecture
Intel, June 2023

Registres des architectures 64-bit

`16 General-purpose registers`

`Segment registers` = abordés dans le chapitre mémoire

`RFLAGS register` = registre de statut

`RIP` = pointeur d'instruction (ou compteur programme)

`FPU registers`

Floating-Point Unit, registres contenant des floats

`MMX, XMM and YMM registers`

Registres SIMD pouvant contenir plusieurs données

(YMM est une extension de XMM, qui est une extension de MMX)

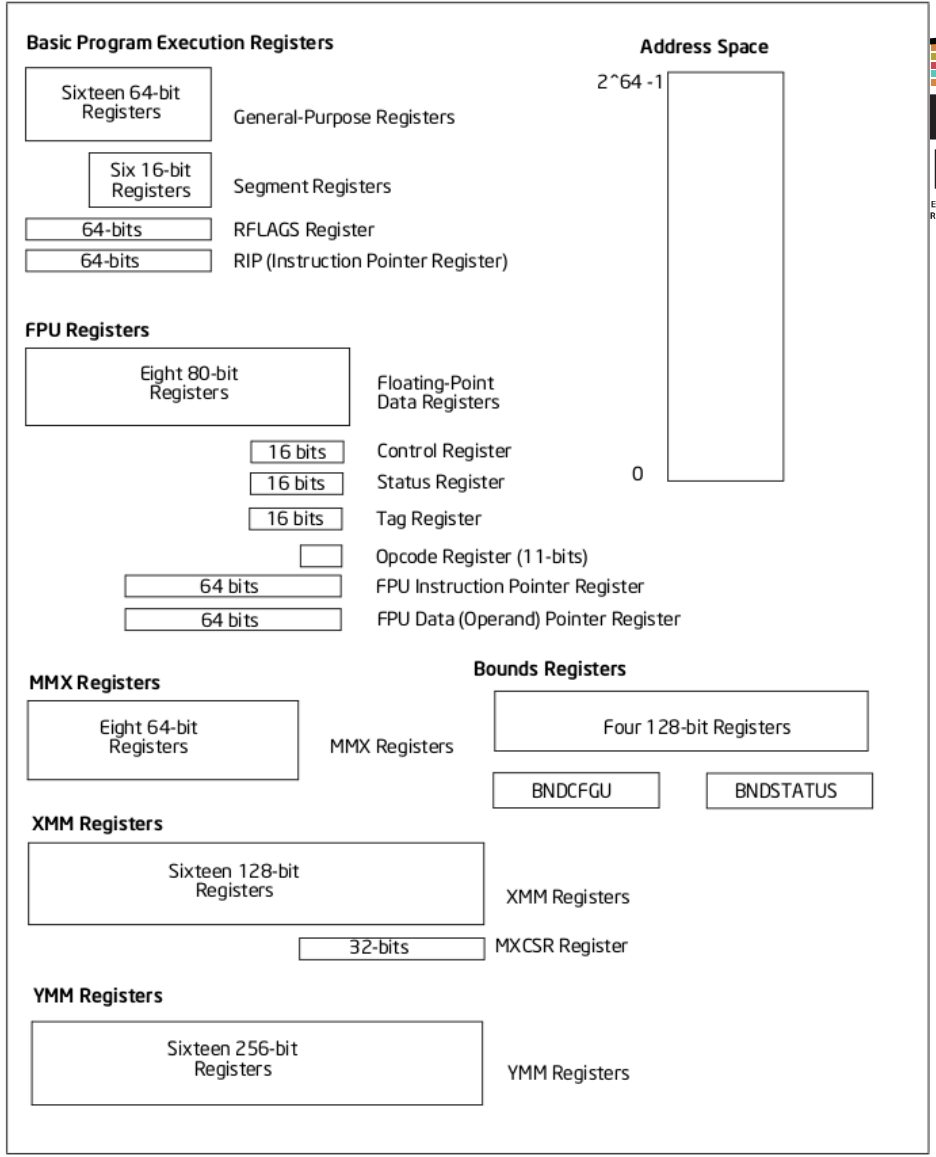


Figure 3-2. 64-Bit Mode Execution Environment

Registres génériques

Afin d'assurer la rétro-compatibilité depuis le 8086, tous les registres sont accessibles avec leur nom d'origine, dans leur taille d'origine.

Les « nouveaux » registres, plus grands (32 bits, puis 64 bits), ne sont que des extensions de leurs ancêtres. Cela signifie que RAX, EAX, AX, … désignent un seul et même registre, seuls les octets manipulés changent.

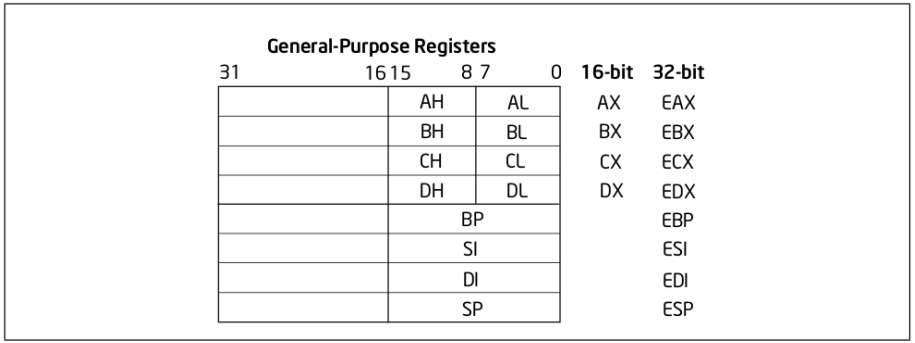
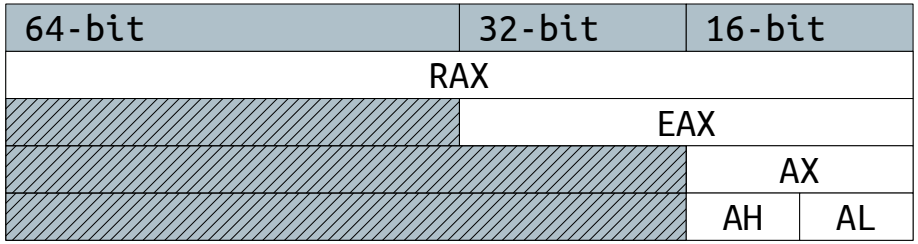


Figure 3-5. Alternate General-Purpose Register Names



Note : depuis l'architecture 64-bit, il existe 8 nouveaux registres 64-bit nommés R8 à R15.

3.4.1.1 General-Purpose Registers in 64-Bit Mode

« In 64-bit mode, there are 16 general purpose registers and **the default operand size is 32 bits**. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. »

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

SIMD registers

Les registres SIMD (Single Instruction Multiple Data) permettent de stocker plusieurs données de même type afin d'y opérer un traitement identique.

MMX est un jeu d'instructions orienté SIMD apparu en 1997, utilisant huit registres 64-bit.

Mal conçu chez Intel (il désactivait la FPU), MMX a vite été remplacé par SSE (1999) qui a apporté avec lui huit nouveaux registres 128-bit et la possibilité de faire du SIMD sur des nombres flottants.

Plus récemment (en 2008) est apparu AVX. Sur architecture 32-bit, les huit registres YMM sont des extensions (sur 256 bits) des registres XMM existants. Sur architecture 64-bit, il y a seize registres 256-bit.

Enfin, AVX-512 (2017) utilise 32 registres 512-bit.

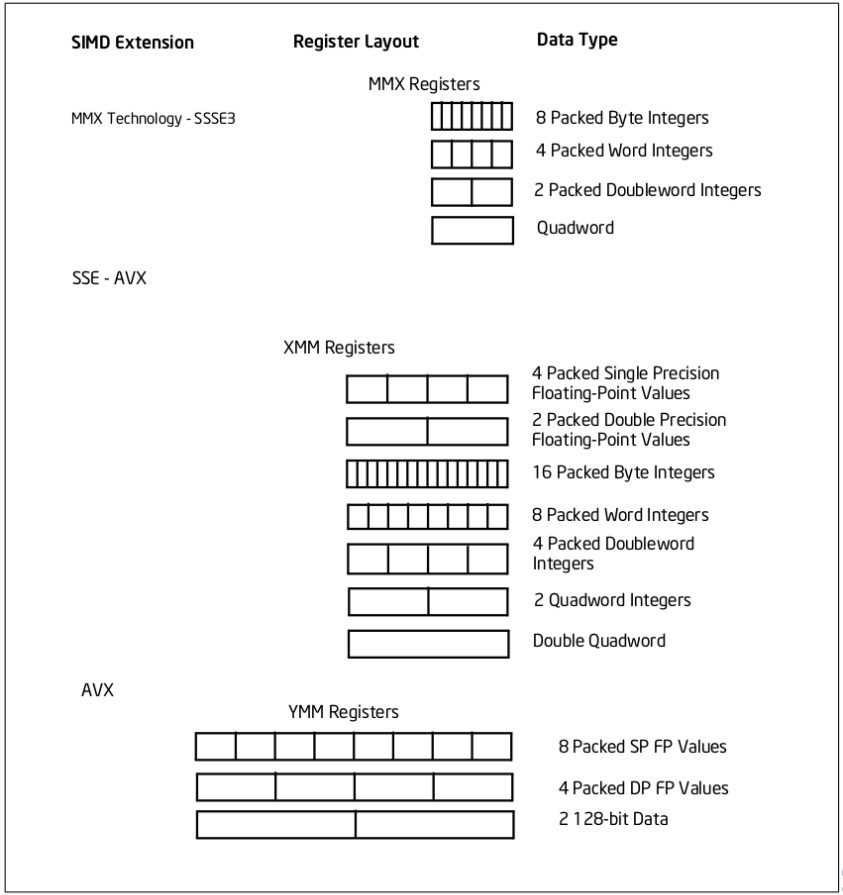
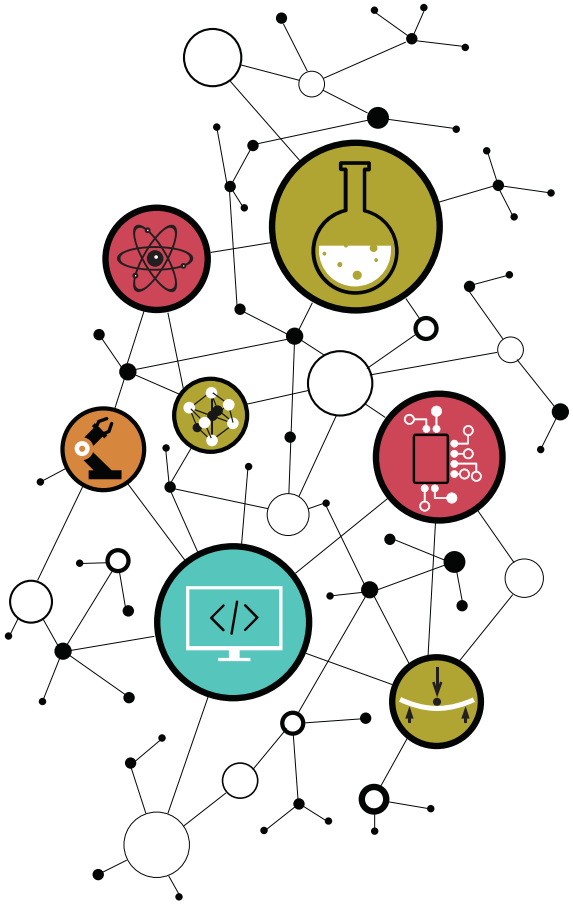


Figure 2-4. SIMD Extensions, Register Layouts, and Data Types

CONTACT



Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)

Except where otherwise noted, this work is licensed under
https://creativecommons.org/licenses/by-nc-sa/4.0/