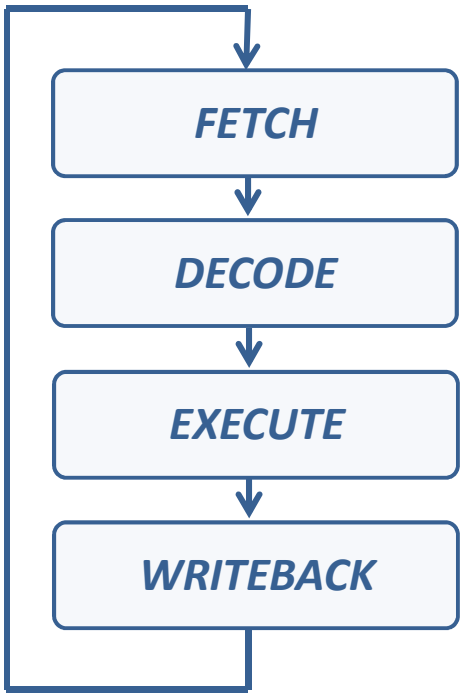
CENTRAL PROCESSING UNIT

Architecture et Technologie des Ordinateurs

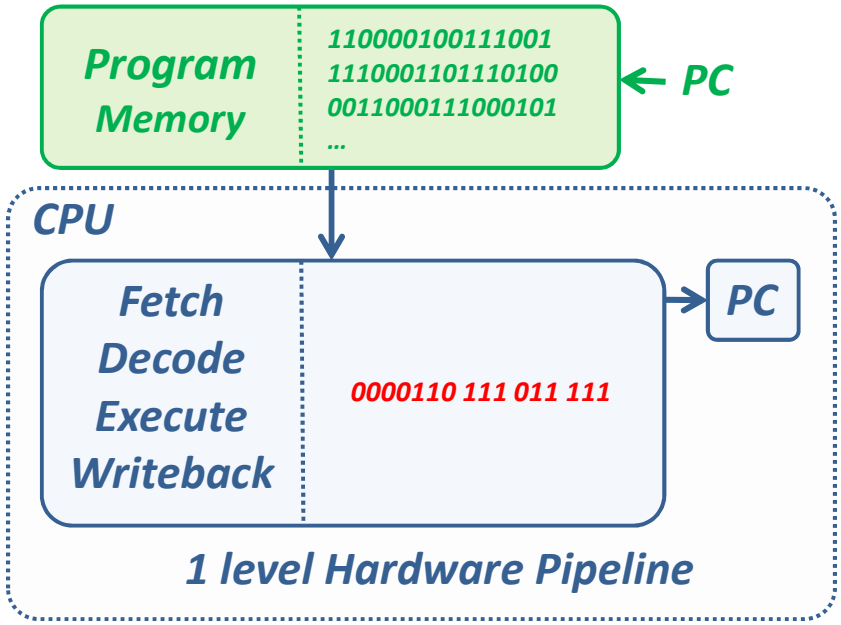
Hugo Descoubes - Juin 2013

Tout CPU effectue séquentiellement les traitements présentés ci-dessous :



- ***FETCH :*** *Aller chercher le code binaire d'une instruction en mémoire programme. Beaucoup de CPU récents sont capables d'aller chercher plusieurs instructions durant la phase de fetch (superscalar, VLIW …).*
- ***DECODE :*** *décodage du ou des opcodes des instructions précédemment récupérées.*
- ***EXECUTION :*** *Exécution de ou des instructions précédemment décodées. Cette opération est réalisée par les unités d'exécution (EU ou Execution Unit).*
- ***WRITEBACK :*** *Ecriture du résultat en mémoire ou dans les registres internes au CPU.*

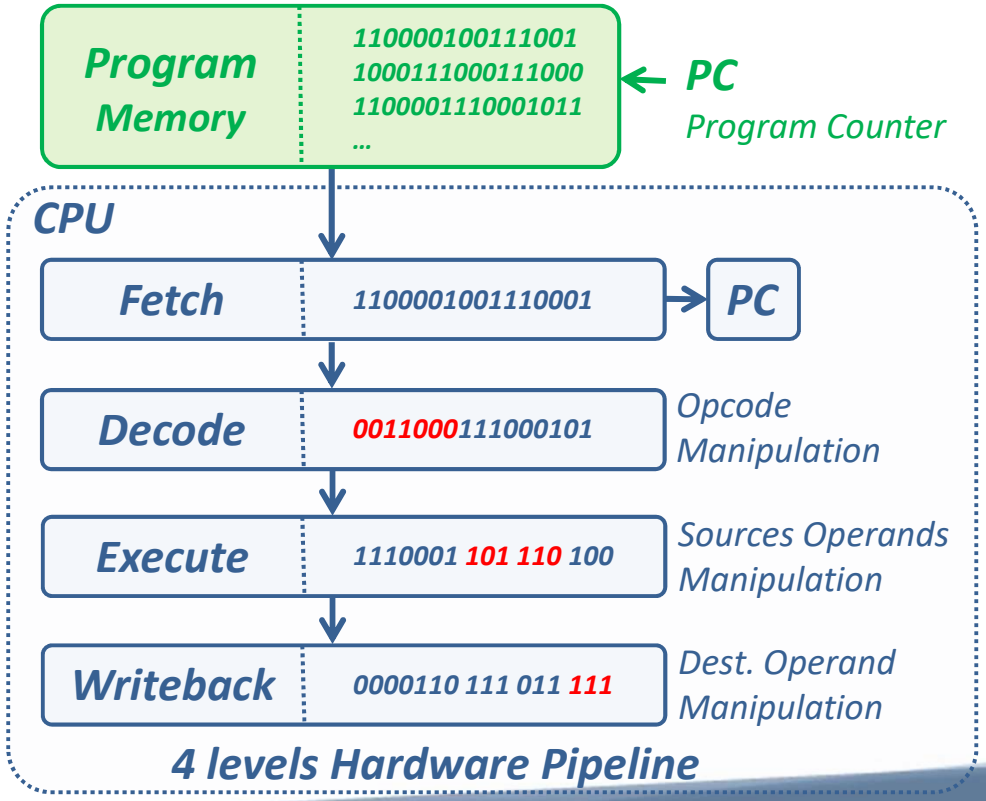
La très grande majorité des architectures modernes sont capables de réaliser une partie voire toutes ces étapes en parallèle. Nous parlerons de pipelining hardware.



Prenons un exemple et supposons que chaque étape prend un cycle CPU (fetch, decode, execute et writeback).

Il faudrait donc 4cy pour exécuter chaque instruction.

La très grande majorité des architectures modernes sont capables de réaliser une partie voir toutes ces étapes en parallèle. Nous parlerons de pipelining hardware.

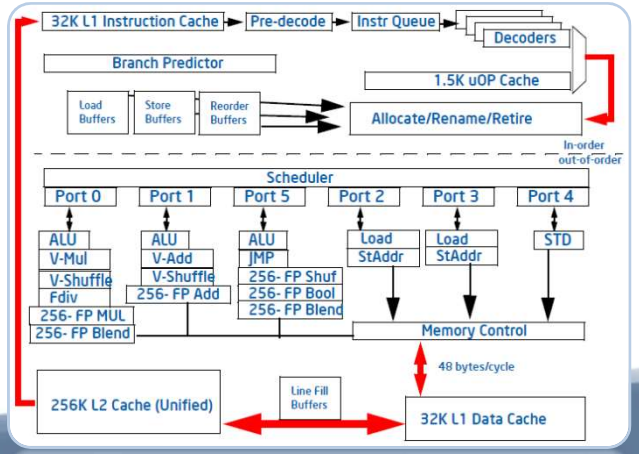


Prenons un exemple et supposons que chaque étape prend un cycle CPU (fetch, decode, execute et writeback).

Il faudrait donc 4cy pour la première instruction et 1cy (théoriquement) pour les suivantes.

Pour un CPU, posséder un pipeline hardware est donc intéressant. Cependant, un pipeline trop profond peut entraîner des ralentissement (souvent lié aux instructions de saut). Il devient alors très difficile d'accélérer l'architecture (mécanismes d'accélération).

A titre d'exemple, les architectures Penryn's de Intel possèdent un pipeline Hardware de 14 niveaux et Nehalem 20-24 étages. Pipeline matériel de la famille sandy bridge :



http://www.intel.com

Etudions un CPU élémentaire RISC-like n'étant rattaché à aucune architecture connue. Observons le jeu d'instruction très très très réduit associé :

Mnemonic	Syntax	Description	Example	Binary (bits)
ADD	ADD regSrc, regSrc, regDst	Addition contenu de 2 registres	ADD R1, R2, R1	000 r r r uu
JMP	JMP label	Saut en mémoire programme	JMP addInst	001 aaaa u
LOAD	LOAD address, regDst	Chargement d'une donnée depuis la mémoire vers le CPU	LOAD addData, R2	010 aaa r u
MOV	MOV regSrc, regDst	Copie le contenu d'un registre vers un autre registre	MOV R2, R1	011 r r uuu
MOVK	MOVK constant, regDst	Charge une constante dans un registre	MOVK cst3bits, R1	100 kkk r u
STR	STR regSrc, address	Sauvegarde une donnée contenu dans un regsitre vers la mémoire	STR R1, addData	101 r aaa u
Glossary : r=registre a=address u=unused k=constant R1=register R2=register addInst=Program memory address addData=Data memory address				

Implémentation assembleur du langage C ci-dessous :

Programme en C

```
char value=3, saveValue;

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

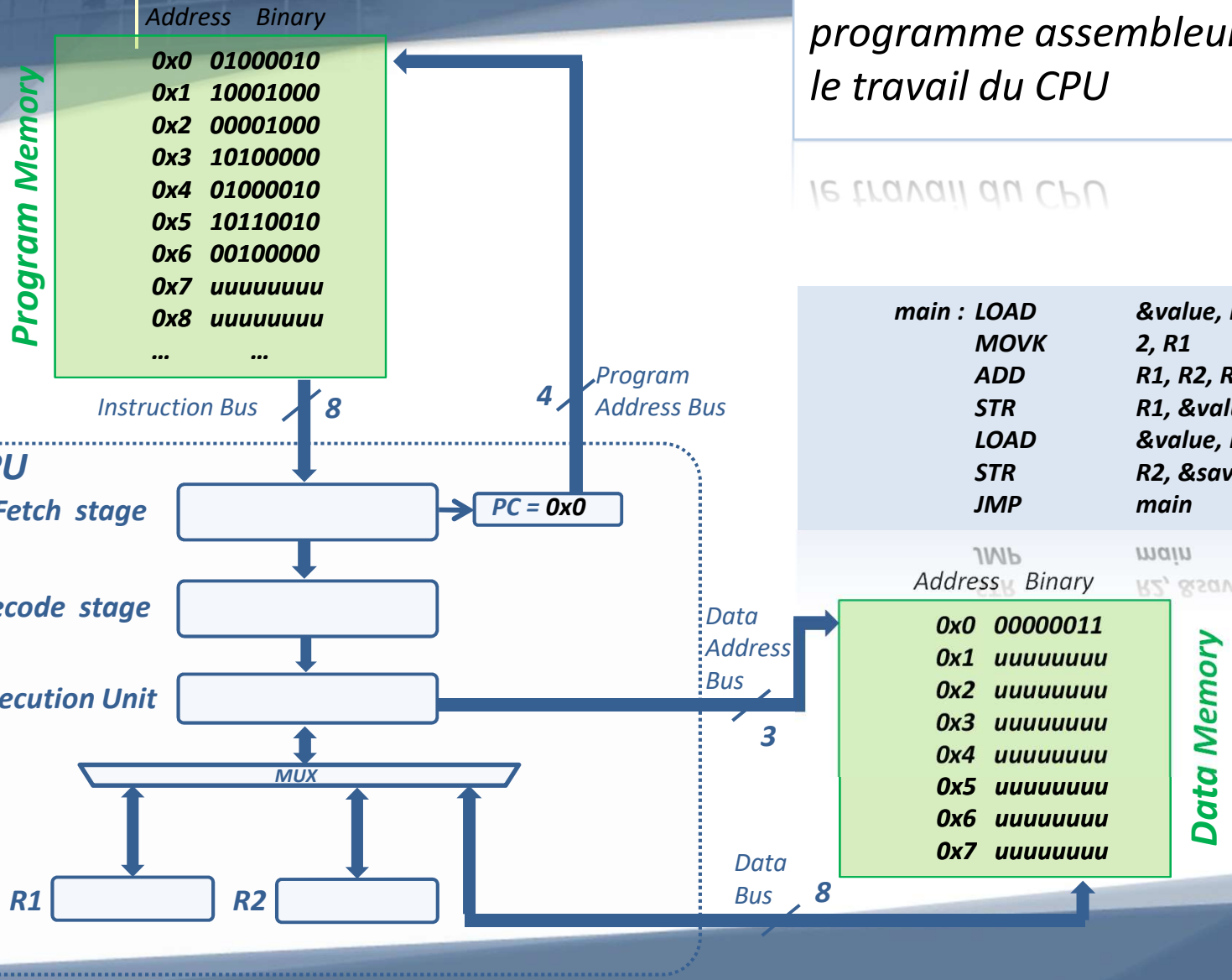
Programme assembleur

Program Address	Mnemonic	operands	Binary
0x0 main :	LOAD	&value, R2	01000010
0x1	MOVK	2, R1	10001000
0x2	ADD	R1, R2, R1	00001000
0x3	STR	R1, &value	10100000
0x4	LOAD	&value, R2	01000010
0x5	STR	R2, &saveValue	10110010
0x6	JMP	main	00100000
0x7	undefined		uuuuuuuu
0x8	undefined		uuuuuuuu
…	…		…
0xF	undefined		uuuuuuuu

Glossary :

R1=0
R2=1
&value=0x0
&saveValue=0x1

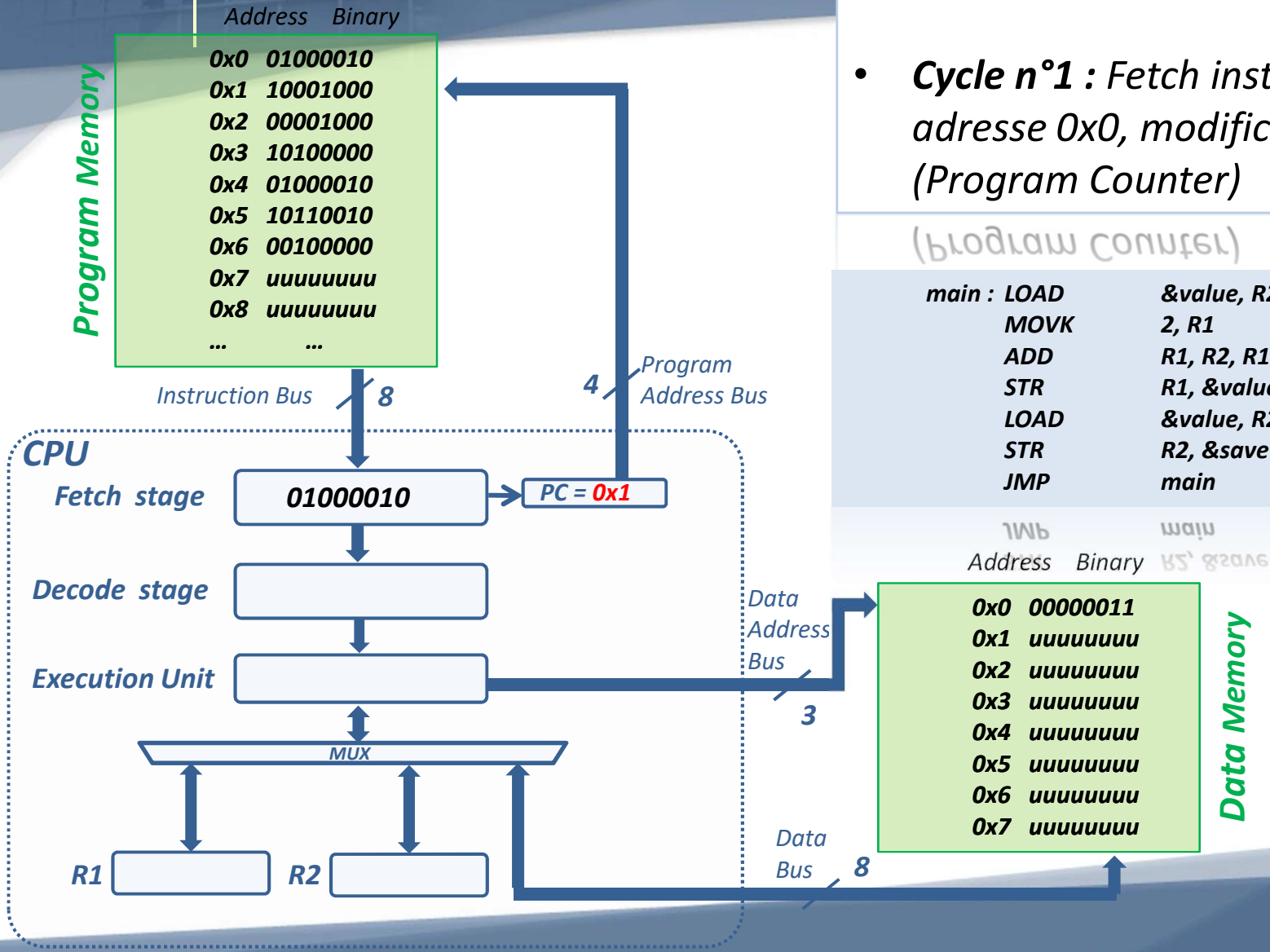
Etudions l'évolution de notre programme assembleur ainsi que le travail du CPU



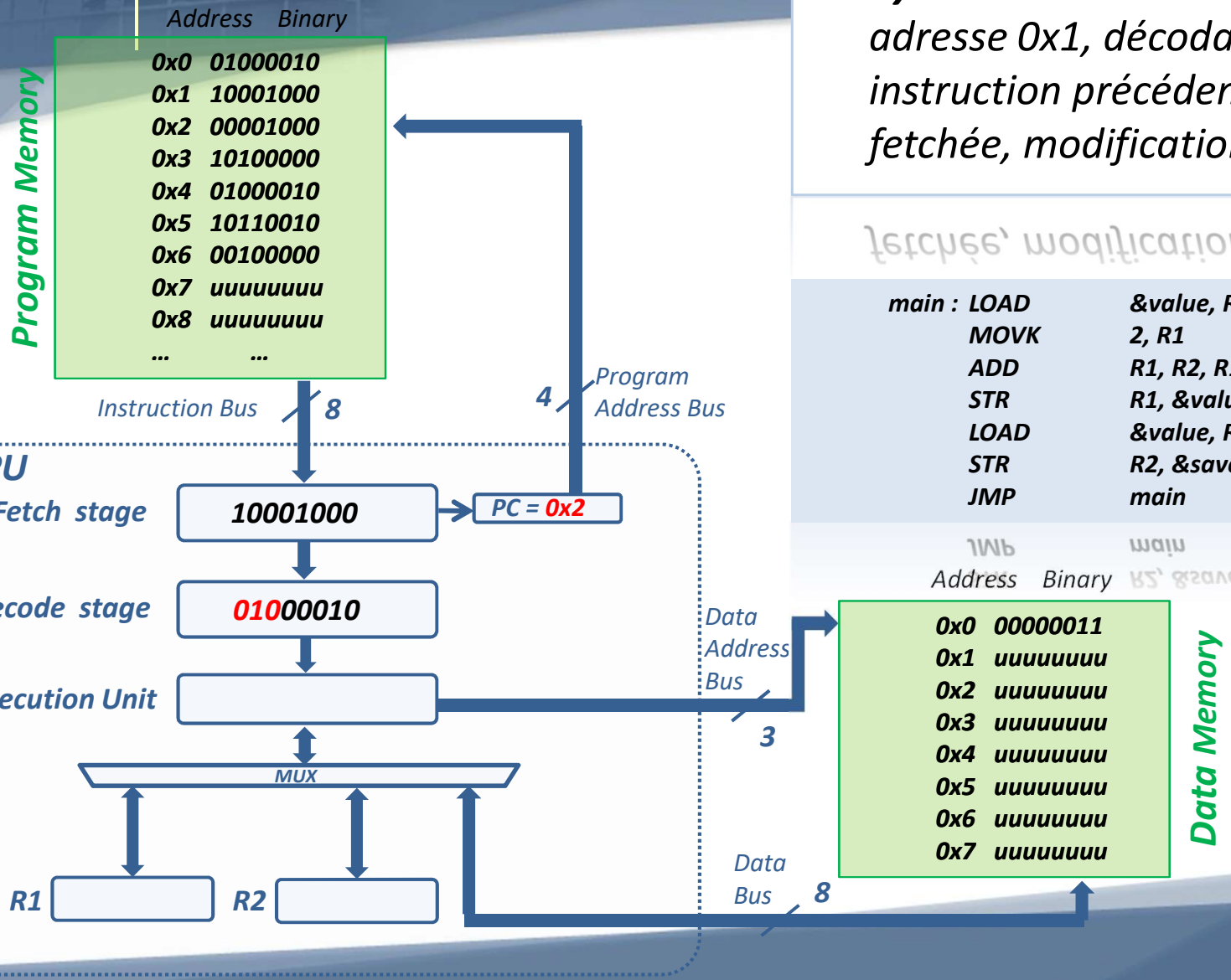

```
main :  LOAD    &value, R2
        MOVK    2, R1
        ADD     R1, R2, R1
        STR     R1, &value
        LOAD    &value, R2
        STR     R2, &saveValue
        JMP     main
```

- *Démarrage application*
- ***Cycle n°1 :*** *Fetch instruction adresse 0x0, modification PC (Program Counter)*

```
main :  LOAD    &value, R2
        MOVK    2, R1
        ADD     R1, R2, R1
        STR     R1, &value
        LOAD    &value, R2
        STR     R2, &saveValue
        JMP     main
```



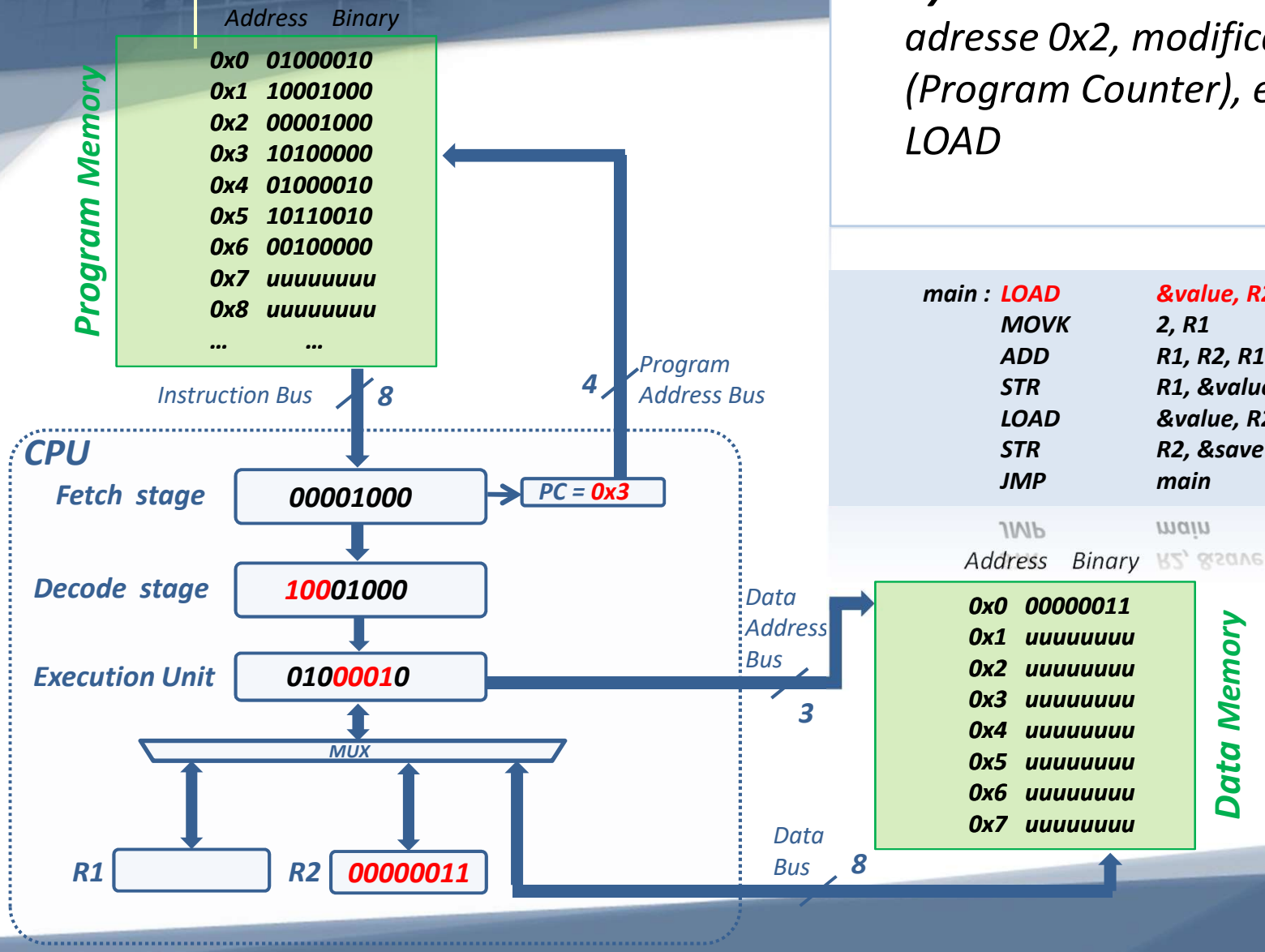
- ***Cycle n°2 :*** *Fetch instruction adresse 0x1, décodage instruction précédemment fetchée, modification PC*




```
main : LOAD   &value, R2
       MOVK   2, R1
       ADD    R1, R2, R1
       STR    R1, &value
       LOAD   &value, R2
       STR    R2, &saveValue
       JMP    main
```

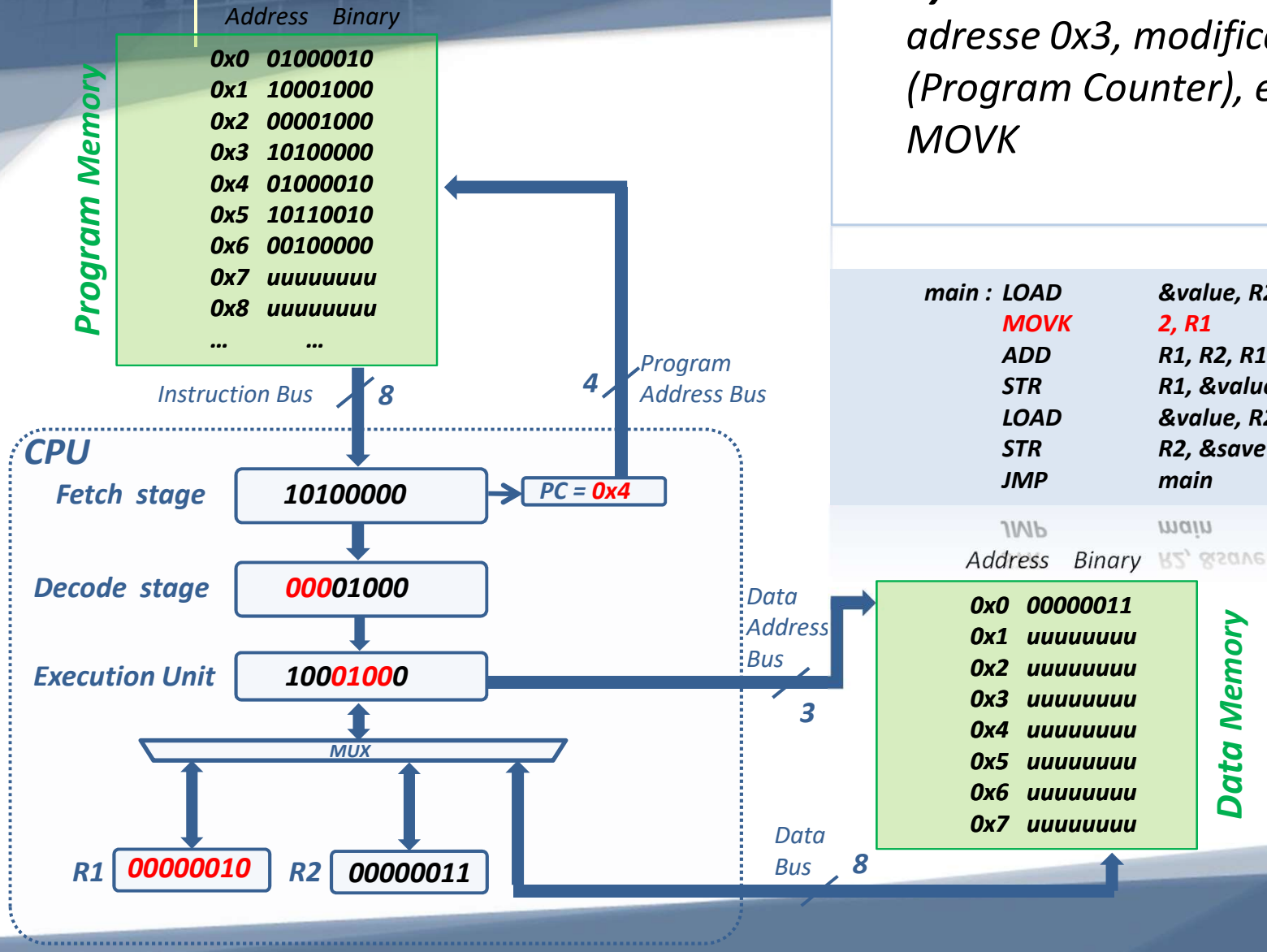
- ***Cycle n°3 :*** *Fetch instruction adresse 0x2, modification PC (Program Counter), exécution LOAD*

```
main :  LOAD    &value, R2
        MOVK    2, R1
        ADD     R1, R2, R1
        STR     R1, &value
        LOAD    &value, R2
        STR     R2, &saveValue
        JMP     main
```



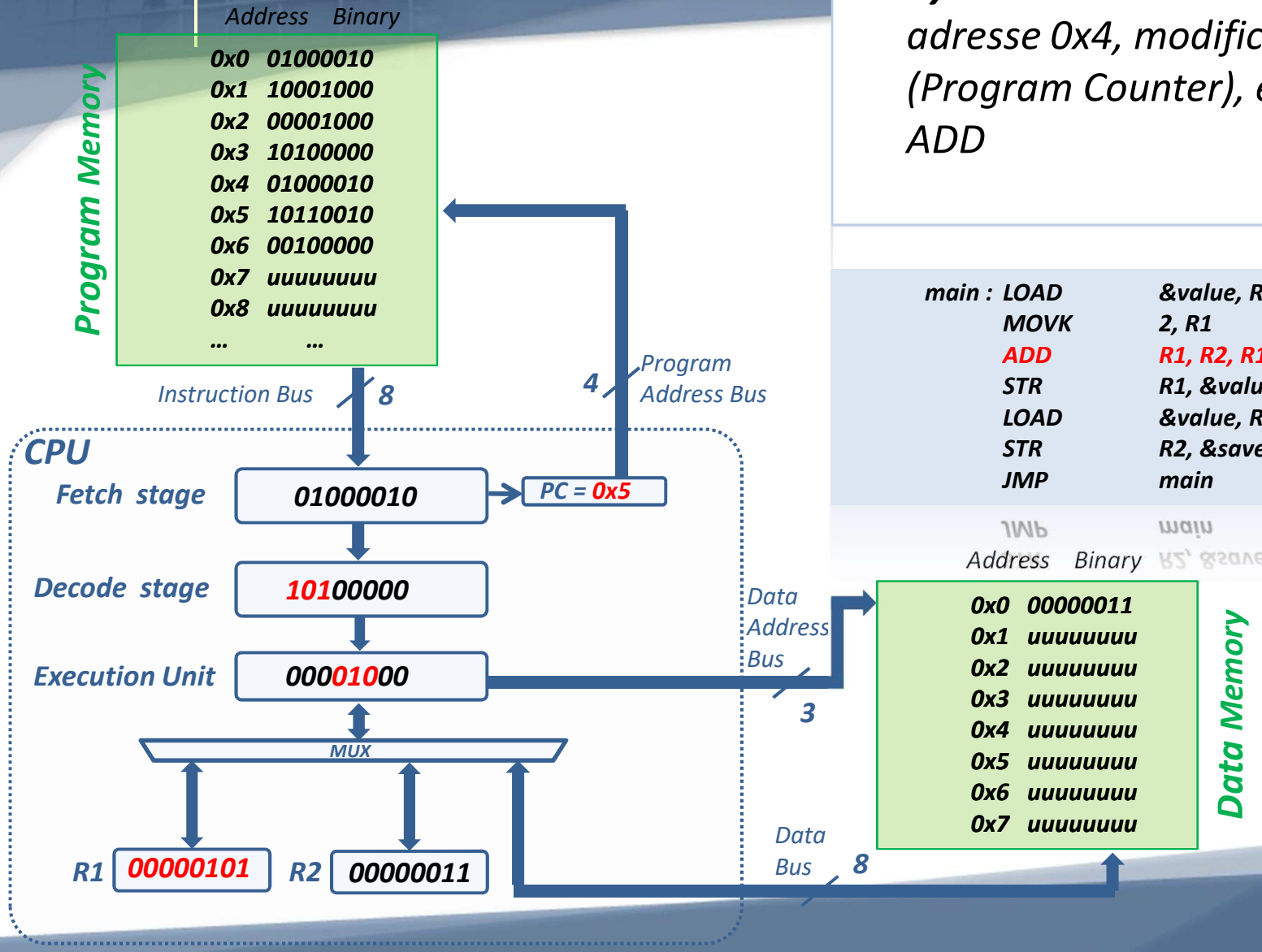
- ***Cycle n°4 :*** *Fetch instruction adresse 0x3, modification PC (Program Counter), exécution MOVK*

```
main :  LOAD    &value, R2
        MOVK    2, R1
        ADD     R1, R2, R1
        STR     R1, &value
        LOAD    &value, R2
        STR     R2, &saveValue
        JMP     main
```



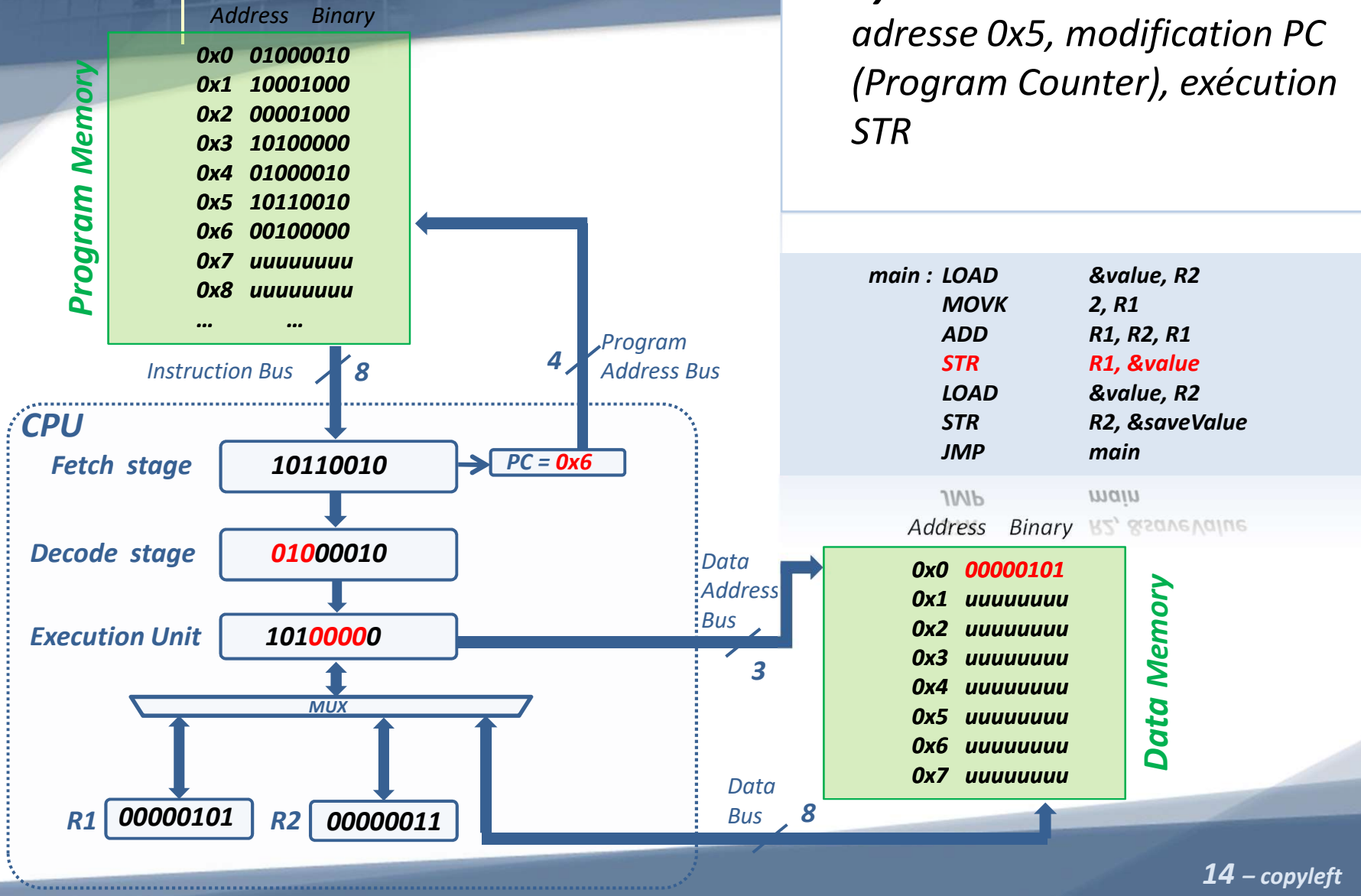
- ***Cycle n°5 :*** *Fetch instruction adresse 0x4, modification PC (Program Counter), exécution ADD*

```
main :  LOAD    &value, R2
        MOVK    2, R1
        ADD     R1, R2, R1
        STR     R1, &value
        LOAD    &value, R2
        STR     R2, &saveValue
        JMP     main
```



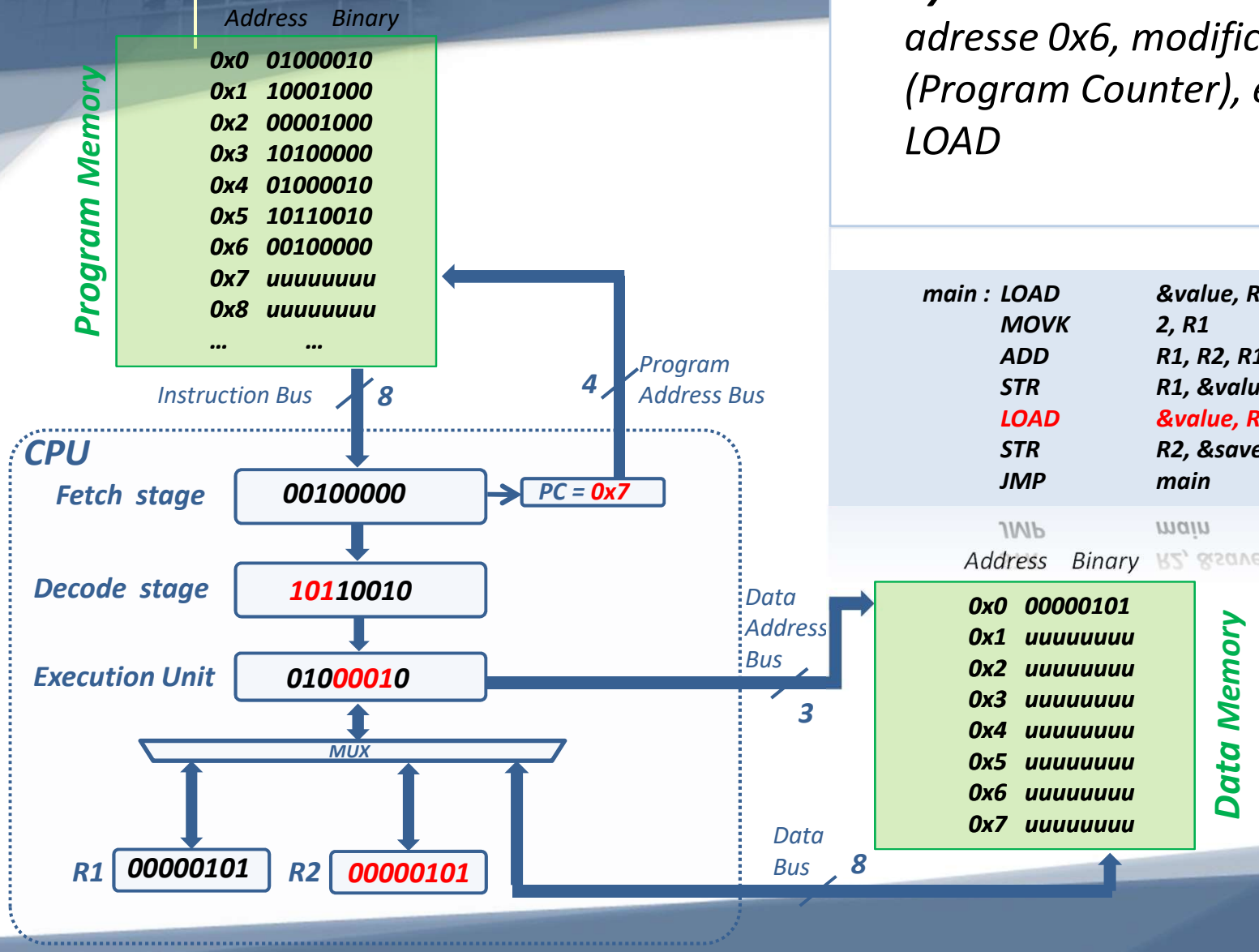
- ***Cycle n°6 :*** *Fetch instruction adresse 0x5, modification PC (Program Counter), exécution STR*

```
main :  LOAD    &value, R2
        MOVK    2, R1
        ADD     R1, R2, R1
        STR     R1, &value
        LOAD    &value, R2
        STR     R2, &saveValue
        JMP     main
```



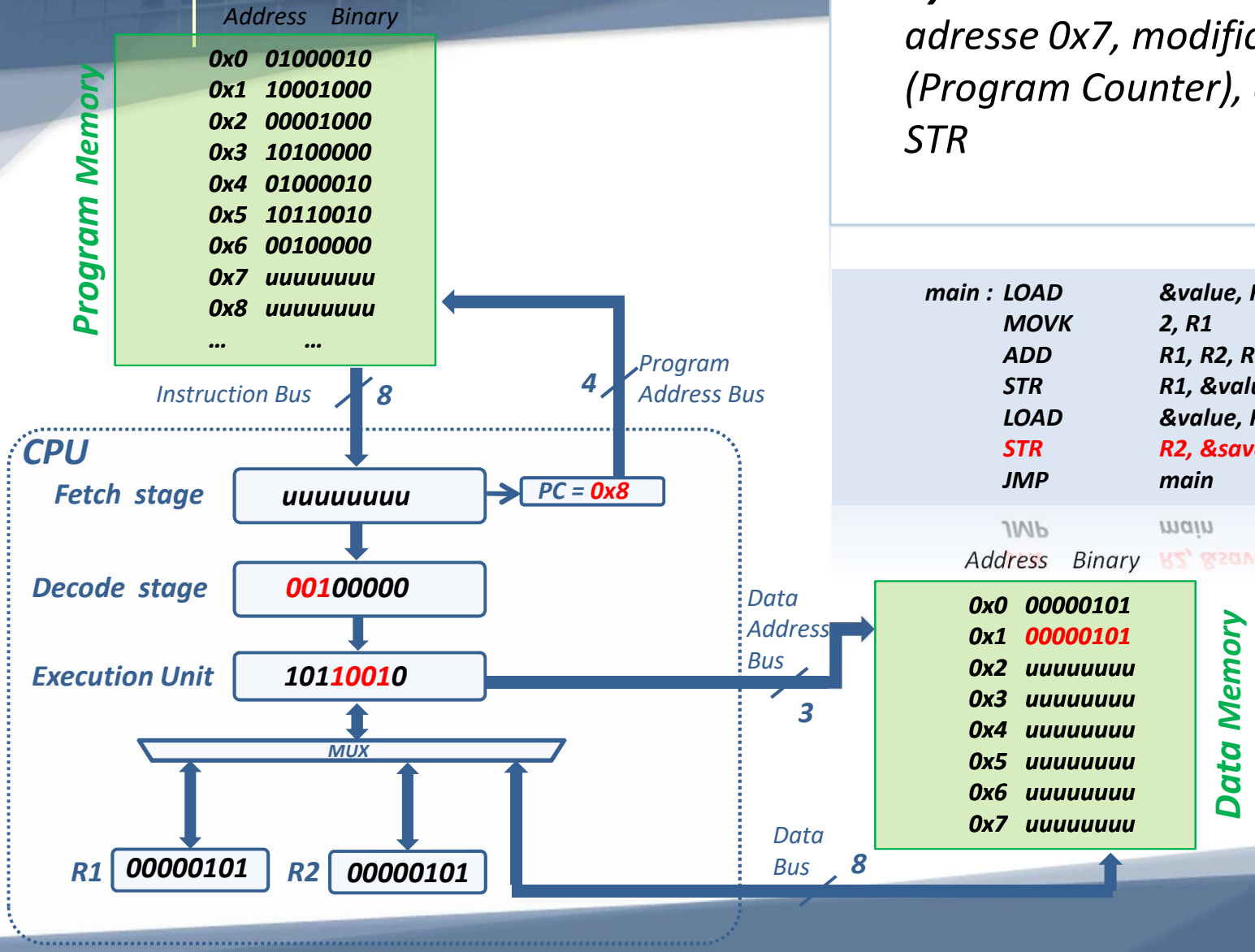
- ***Cycle n°7 :*** *Fetch instruction adresse 0x6, modification PC (Program Counter), exécution LOAD*

```
main :  LOAD    &value, R2
        MOVK    2, R1
        ADD     R1, R2, R1
        STR     R1, &value
        LOAD    &value, R2
        STR     R2, &saveValue
        JMP     main
```



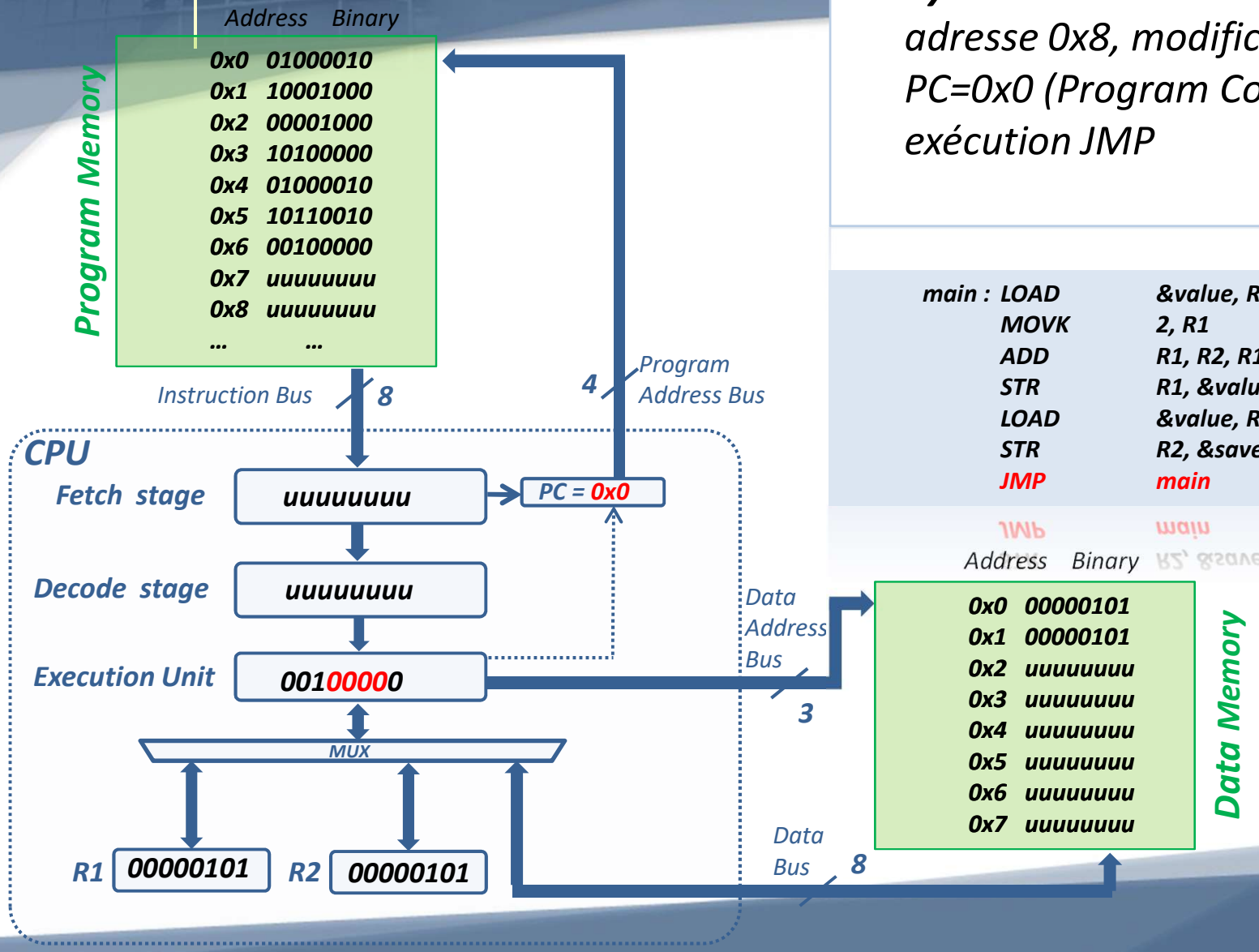
- ***Cycle n°8 :*** *Fetch instruction adresse 0x7, modification PC (Program Counter), exécution STR*

```
main :  LOAD    &value, R2
        MOVK    2, R1
        ADD     R1, R2, R1
        STR     R1, &value
        LOAD    &value, R2
        STR     R2, &saveValue
        JMP     main
```



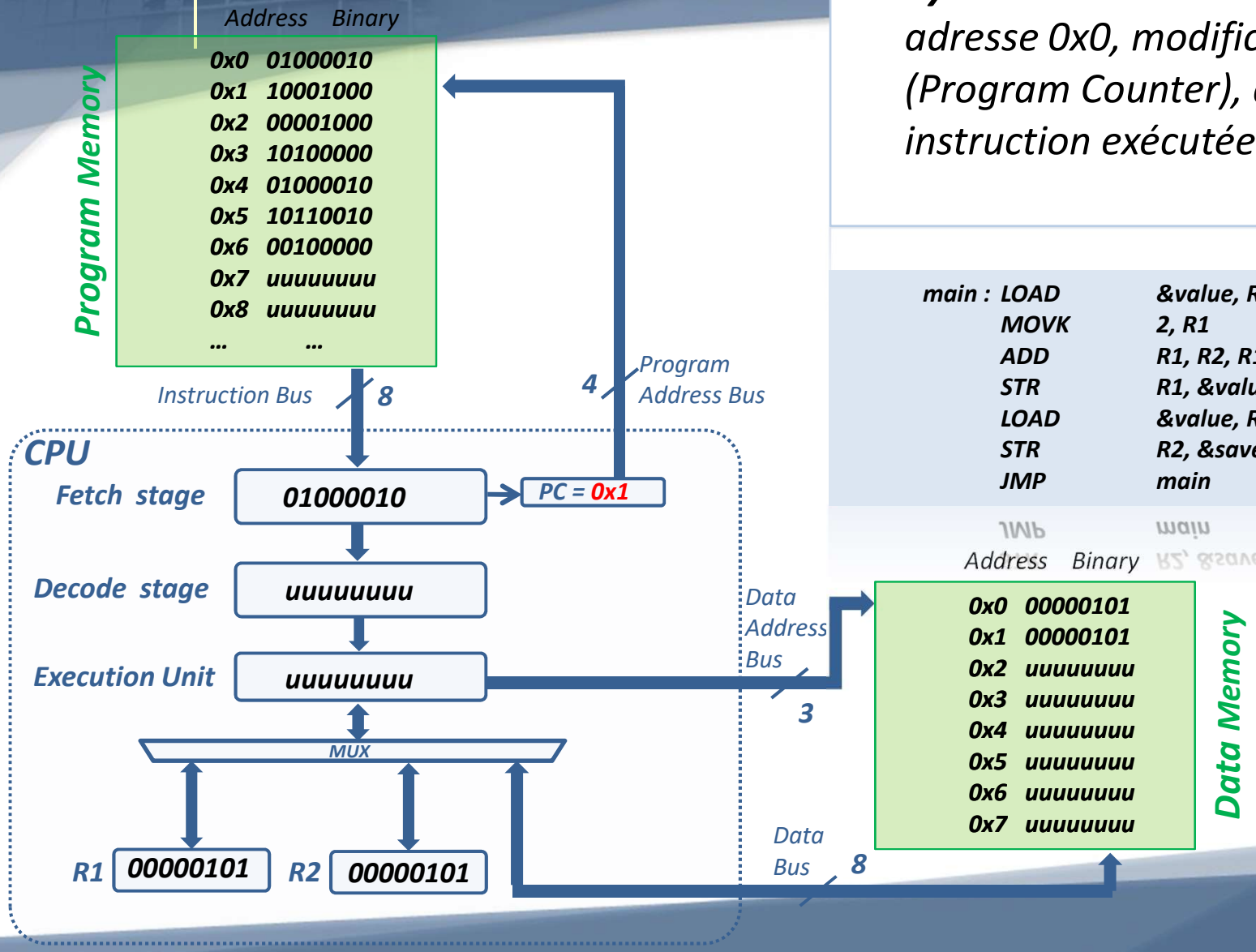
- ***Cycle n°9 :*** *Fetch instruction adresse 0x8, modification PC=0x0 (Program Counter), exécution JMP*

```
main : LOAD   &value, R2
       MOVK   2, R1
       ADD    R1, R2, R1
       STR    R1, &value
       LOAD   &value, R2
       STR    R2, &saveValue
       JMP    main
```



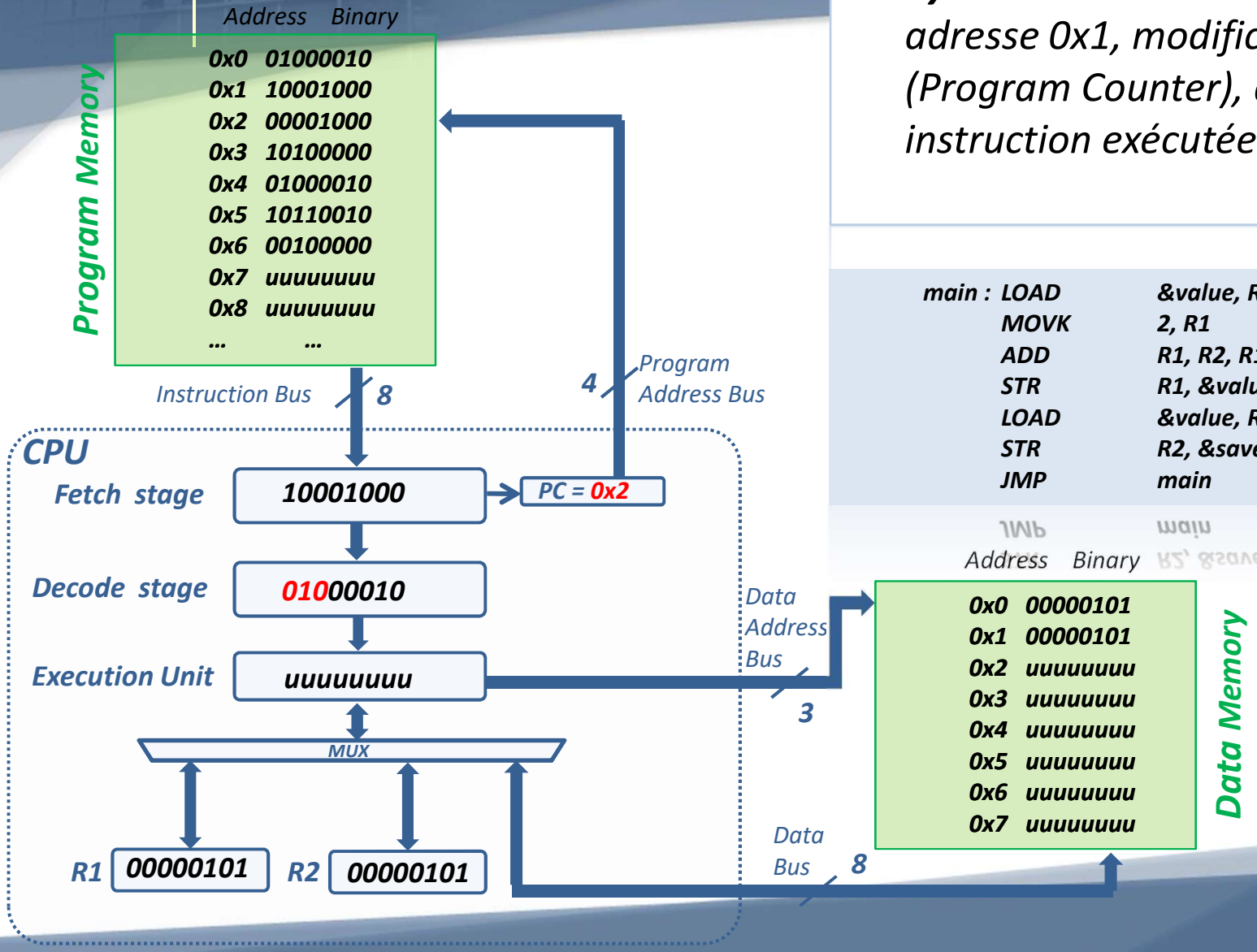
- **Cycle n°10 :** *Fetch instruction adresse 0x0, modification PC (Program Counter), aucune instruction exécutée*

```
main : LOAD   &value, R2
       MOVK   2, R1
       ADD    R1, R2, R1
       STR    R1, &value
       LOAD   &value, R2
       STR    R2, &saveValue
       JMP    main
```



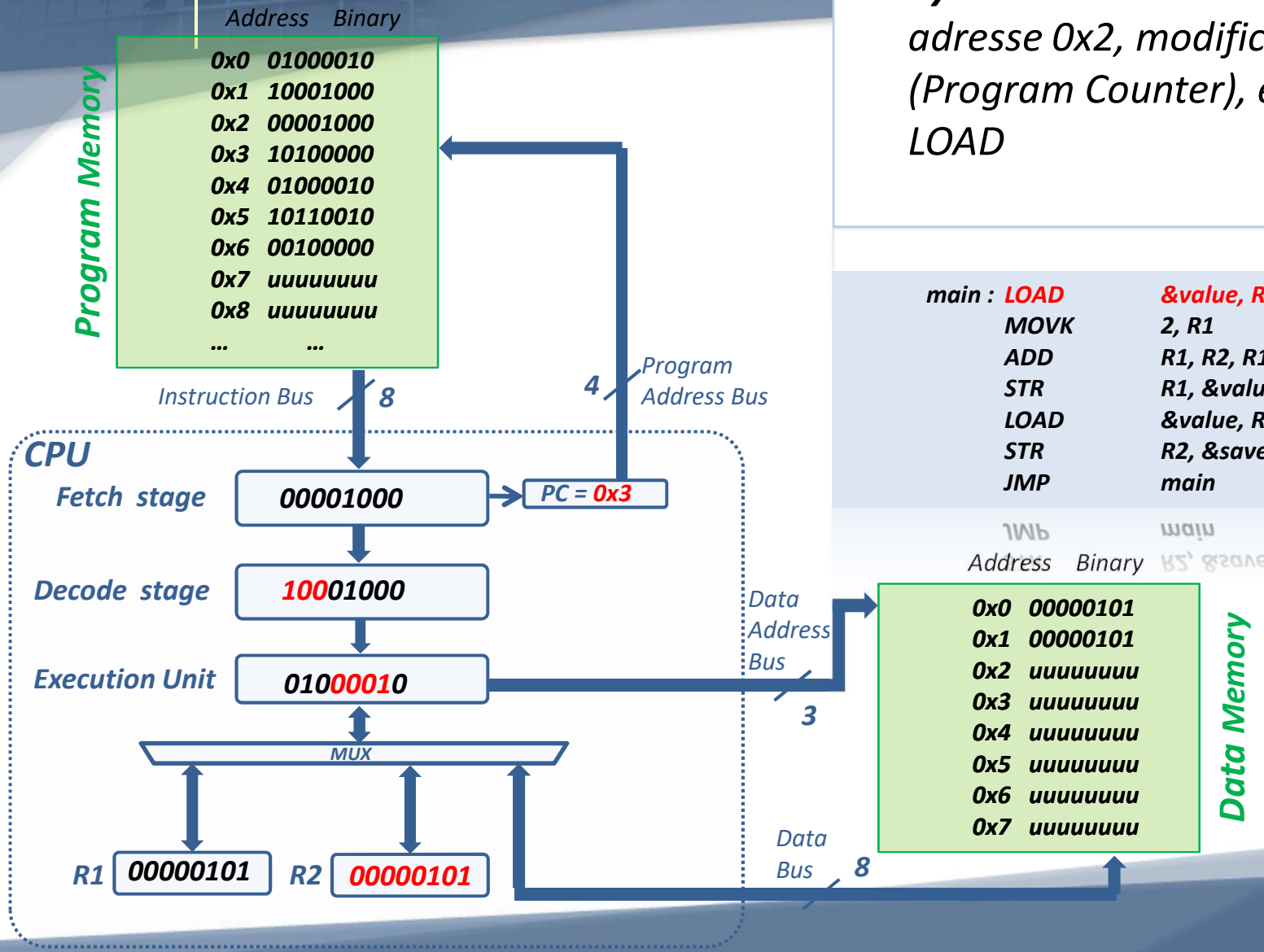
- ***Cycle n°11 :*** *Fetch instruction adresse 0x1, modification PC (Program Counter), aucune instruction exécutée*

```
main : LOAD   &value, R2
       MOVK   2, R1
       ADD    R1, R2, R1
       STR    R1, &value
       LOAD   &value, R2
       STR    R2, &saveValue
       JMP    main
```



- **Cycle n°12 :** *Fetch instruction adresse 0x2, modification PC (Program Counter), exécution LOAD*

```
main : LOAD   &value, R2
       MOVK   2, R1
       ADD    R1, R2, R1
       STR    R1, &value
       LOAD   &value, R2
       STR    R2, &saveValue
       JMP    main
```

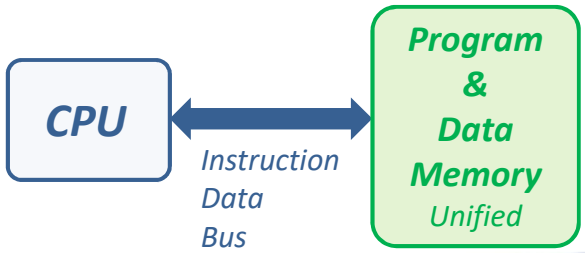


Etc …

- Von Neumann
- Harvard
- Harvard Modifié

Un CPU peut posséder différents modèles d'interconnexion avec les mémoires (program et data). Chaque modèle amène son lot d'avantages et d'inconvénients.

Historiquement, l'une des premières architectures rencontrées était celle dîtes de Von Neumann. Mapping mémoire voire mémoire unifiée (code et données). Le CPU 8086 de Intel possède une architecture de Von Neumann. Néanmoins via une astuce il possède un pipeline à 2 niveaux.



En 2012, certains CPU's actuels utilisent encore ce type de fonctionnement dans certains cas. Il s'agit d'architectures hybrides Harvard/Von Neumann, par exemple les PIC18 de Microchip. Possibilité de placer des données en mémoire programme.

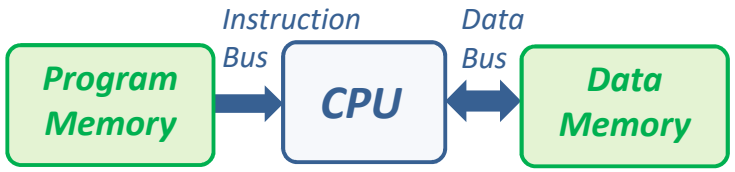
Observons quelques avantages et inconvénients de cette architecture :

- ***Mapping mémoire unique*** *(data et program)*
- ***Polyvalent si mémoire unifiée****. Applications code large et peu de données et vice versa.*
- *Mais,* ***pipeline matériel difficile*** *(fetch, decode, execute, writeback en parallèle).*

- Von Neumann
- Harvard
- Harvard Modifié

En 2012, l'architecture de Harvard est toujours rencontrée sur certains processeurs. Prenons les exemples des PIC18 de Microchip, AVR de Atmel …

Une architecture de Harvard offre une mémoire programme séparée de la mémoire donnée. Technologie, taille des adresses donc taille des mémoires et bus distincts.



Observons quelques avantages de ce type d'architecture :

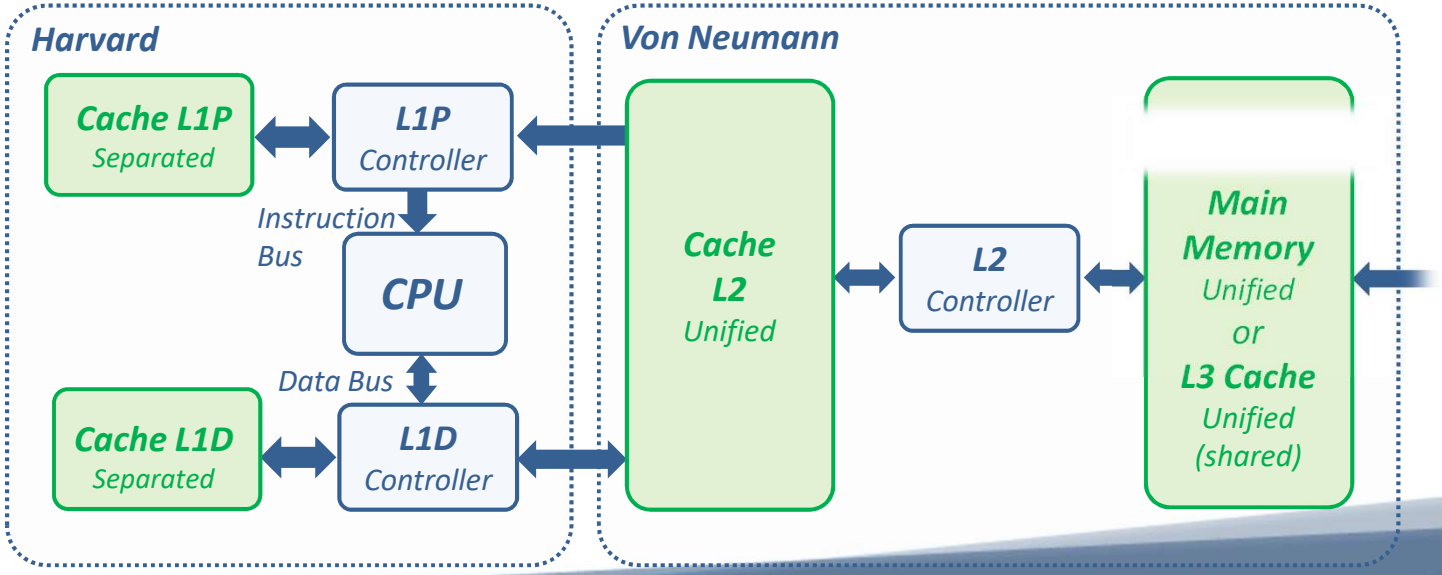
- ***pipeline matériel possible**. Fetch (program memory) en parallèle des phases decode (CPU), execute (CPU ou data memory) suivi du writeback (CPU ou data memory).*

Observons quelques inconvénients de ce type d'architecture :

- ***Mapping mémoires distincts** (adresse mémoire donnée différente adresse mémoire programme). Moins flexible pour le développeur.*
- ***Peu polyvalent**. Certaines applications exigent une large empreinte en mémoire donnée (traitement image et son, bases de données…) pour d'autres ce sera le code …*

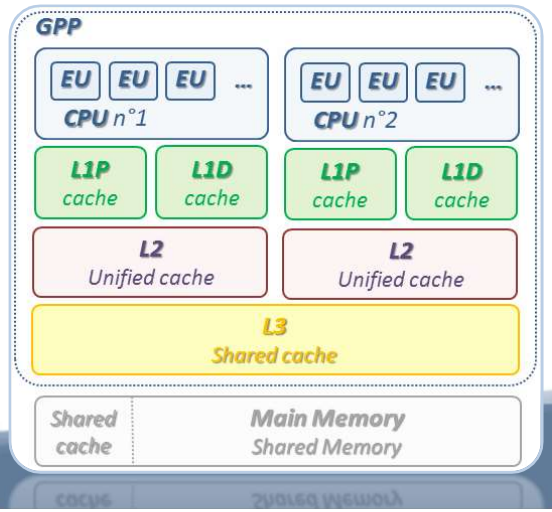
- Von Neumann
- Harvard
- Harvard Modifié

L'architecture de Harvard modifié tend à allier les avantages des deux architectures précédemment présentées. Elle amène cependant son lot d'inconvénients. La très grande majorité des CPU's modernes utilise ce type d'architectures. Prenons une liste non exhaustive de CPU : Core/Coreix de Intel, Cortex-A de ARM, C6xxx de Texas Instrument …



- Von Neumann
- Harvard
- Harvard Modifié

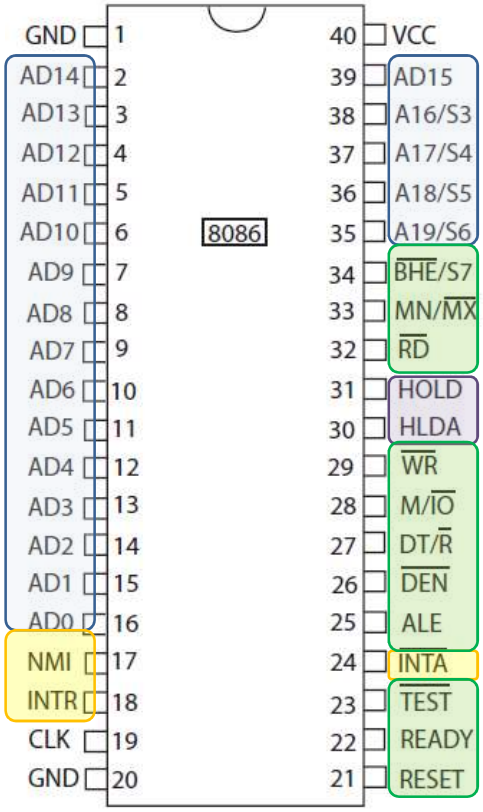
En informatique, une mémoire cache est chargée d'enregistrer et de partager temporairement des copies d'informations (données ou code) venant d'une autre source*, contrairement à une mémoire tampon qui ne réalise pas de copie. L'utilisation de mémoire cache est un mécanisme d'optimisation pouvant être matériel (Cache Processeur L1D, L1P, L2, L3 shared…) comme logiciel (cache DNS, cache ARP…). Sur processeur numérique, le cache est alors hiérarchisé en niveaux dépendants des technologies déployées :*



Ce type d'architecture allie les avantages associés aux architectures de Harvard et de Von Neumann via l'utilisation de mémoire cache. Un CPU est alors associé à son cache processeur (transparence de cache) et entraîne une empreinte silicium de l'ensemble plus importante. Pour un développeur bas niveau adepte de l'optimisation, une manipulation optimale de la mémoire cache exige une grande rigueur de développement (data coherency).

L'un des principaux dangers de ce type de mémoire, est la ***cohérence des informations*** *présentes dans la hiérarchie mémoire du processeurs. Par exemple pour un coreix de la famille sandy bridge, une même donnée peut exister avec différentes valeurs en mémoire principale (DDR), mémoire cache L3 (shared multi-core), L2 (unified mono-core), L1D (separeted mono-core) et dans les registres internes du CPU.*

Découvrons plus en détail le 8086 anciennement proposé par Intel. Rappelons que ce CPU est à la base des architectures x86 :

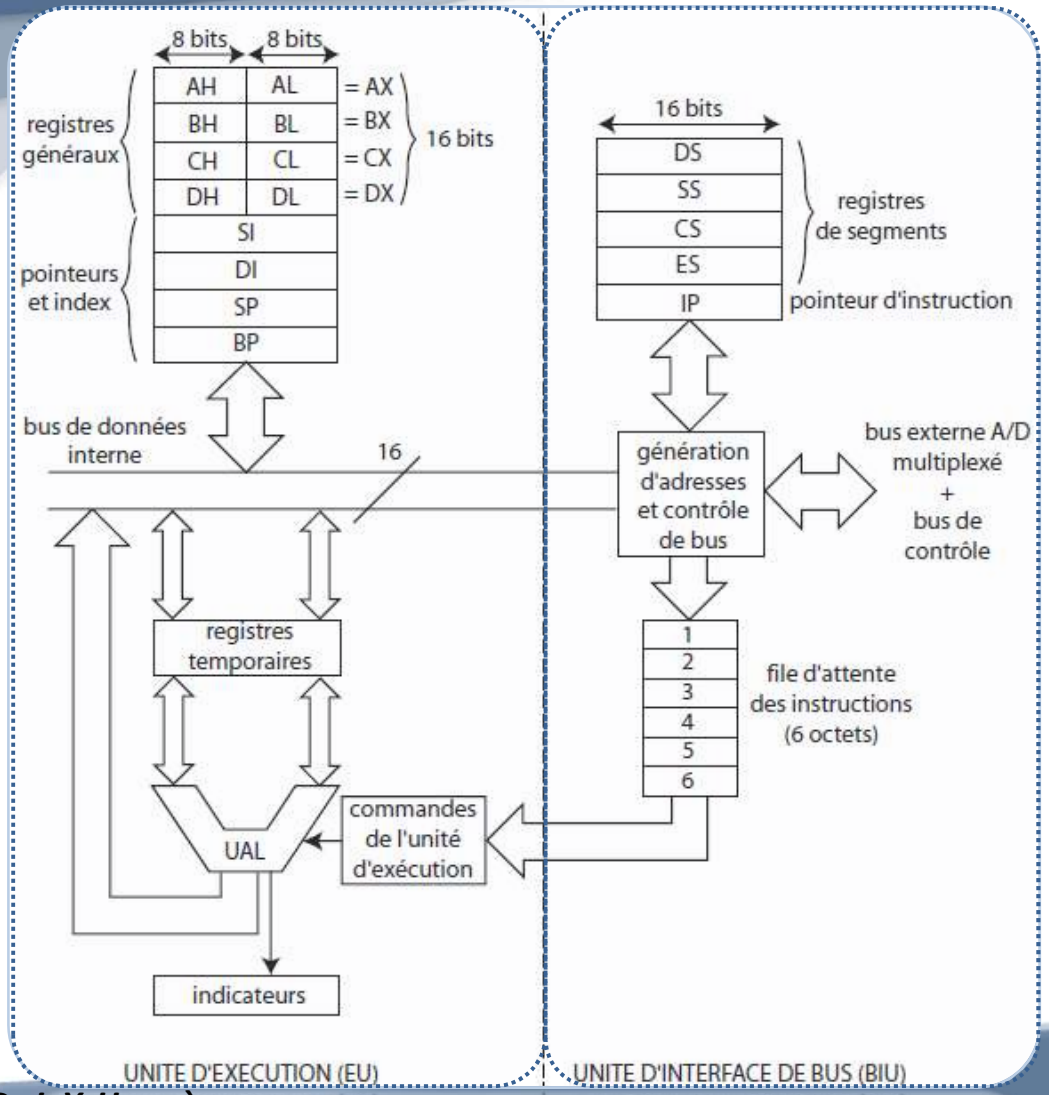


: bus d'adresse de donnée

: bus de contrôle

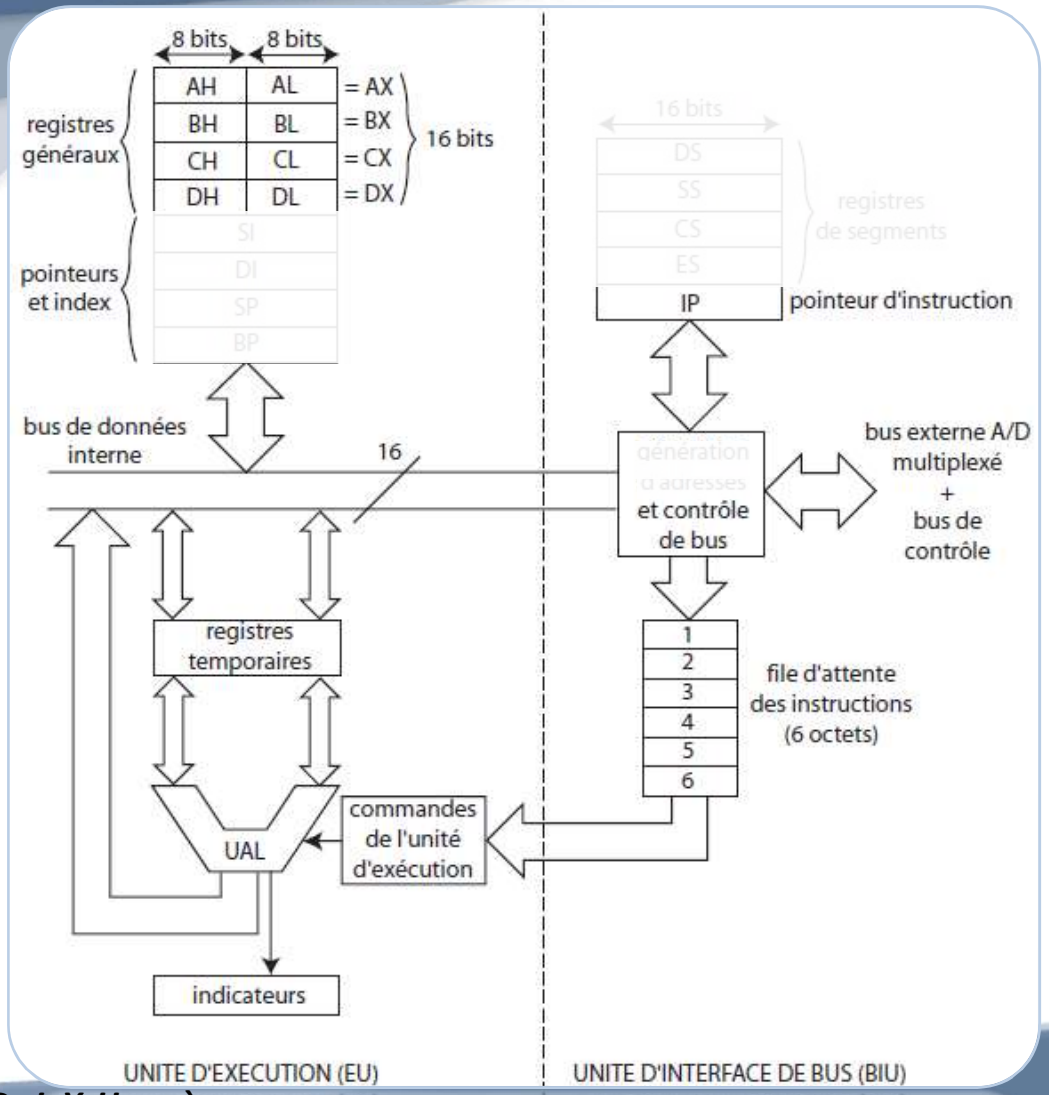
: Interruptions

: Direct Memory Access

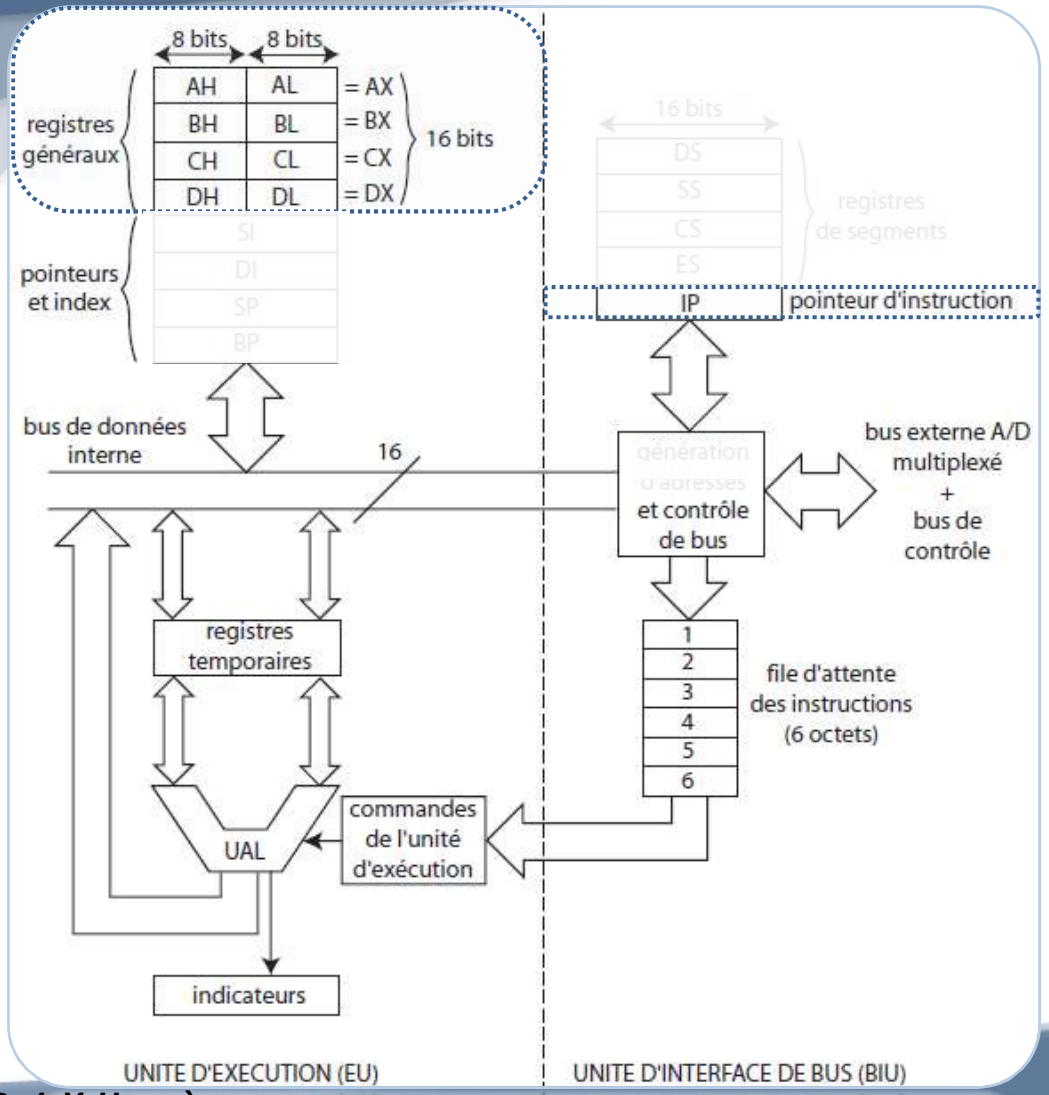


- ***Execution Unit :*** *décode puis exécute les instructions présentes dans la file d'attente*

- ***Bus Interface Unit :*** *contrôle des bus pour les accès mémoire. Calcul adresses physiques (segmentation). Gestion phases de fetch via IP ou Instruction Pointer (équivalent à PC ou Program Counter).*

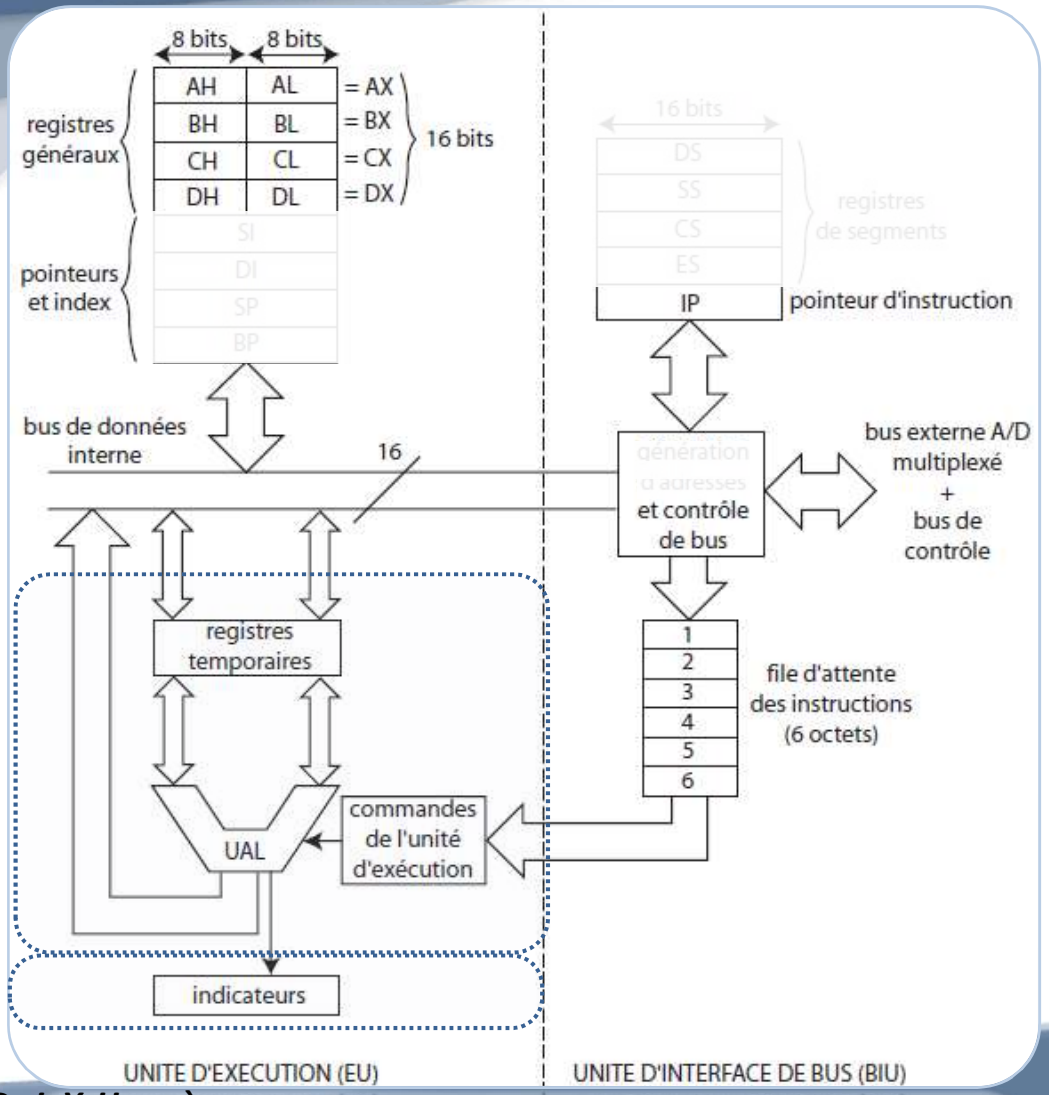


- ***Segmentation:*** *la segmentation mémoire sera vue plus tard dans le cours lorsque nous aborderons l'étude de la MMU (Memory Managment Unit).*
- ***Pile :*** *vu dans la suite du cours.*
- ***Indexage :*** *vu dans la suite du cours.*



- ***General Purpose Registers :*** *AX (AH+AL), BX (BH+BL), CX (CH+CL) et DX (DH+DL) sont des registres généralistes 16bits. Certains d'entre eux peuvent être spécialisés AX=accumulateur, CX=compteur…*

- ***Instruction Pointer :*** *contient l'adresse de la prochaine instruction à aller chercher.*



- ***Arithmetic Logical Unit :*** *l'UAL ou ALU est l'unité de calcul du CPU. Cette unité effectue des opérations arithmétiques et logiques élémentaires.*

- ***Flags :*** *des flags (indicateurs) sont associés à une unité de calcul : Carry (débordement), Z (zero), S (signe), O (overflow) …*

				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

Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions

- Architecture matérielle
- jeu d'instruction

Original 8086 Instruction set

AAA	ASCII adjust AL after addition
AAD	ASCII adjust AX before division
AAM	ASCII adjust AX after multiplication
AAS	ASCII adjust AL after subtraction
ADC	Add with carry
ADD	Add
AND	Logical AND
CALL	Call procedure
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP	Compare operands
CMPSB	Compare bytes in memory
CMPSW	Compare words
CWD	Convert word to doubleword
DAA	Decimal adjust AL after addition
DAS	Decimal adjust AL after subtraction
DEC	Decrement by 1
DIV	Unsigned divide
ESC	Used with floating-point unit

HLT	Enter halt state
IDIV	Signed divide
IMUL	Signed multiply
IN	Input from port
INC	Increment by 1
INT	Call to interrupt
INTO	Call to interrupt if overflow
IRET	Return from interrupt
Jcc	Jump if condition
JMP	Jump
LAHF	Load flags into AH register
LDS	Load pointer using DS
LEA	Load Effective Address
LES	Load ES with pointer
LOCK	Assert BUS LOCK# signal
LODSB	Load string byte
LODSW	Load string word
LOOP/LOOPx	Loop control
MOV	Move
MOVSB	Move byte from string to string
MOVSW	Move word from string to string
MUL	Unsigned multiply

Original 8086 Instruction set

NEG	Two's complement negation
NOP	No operation
NOT	Negate the operand, logical NOT
OR	Logical OR
OUT	Output to port
POP	Pop data from stack
POPF	Pop data from flags register
PUSH	Push data onto stack
PUSHF	Push flags onto stack
RCL	Rotate left (with carry)
RCR	Rotate right (with carry)
REPxx	Repeat MOVS/STOS/CMPS/LODS/SCAS
RET	Return from procedure
RETN	Return from near procedure
RETF	Return from far procedure
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift Arithmetically left (signed shift left)
SAR	Shift Arithmetically right (signed shift right)
SBB	Subtraction with borrow

SCASB	Compare byte string
SCASW	Compare word string
SHL	Shift left (unsigned shift left)
SHR	Shift right (unsigned shift right)
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOSB	Store byte in string
STOSW	Store word in string
SUB	Subtraction
TEST	Logical compare (AND)
WAIT	Wait until not busy
XCHG	Exchange data
XLAT	Table look-up translation
XOR	Exclusive OR

Nous allons maintenant découvrir quelques-unes des principales instructions supportées par le 8086 (documentation en ligne, http://zsmith.co/intel.html). Il ne s'agira pas d'une étude approfondie de chaque instruction et certaines subtilités ne seront pas abordées dans ce cours ou seront vues par la suite (adressage indexé, segmentation…). La présentation suivante sera découpée comme suit :

- ***Instructions de management de données***
- ***Instructions arithmétiques et logiques***
- ***Instructions de saut***

Comme tout CPU, le 8086 est capable de déplacer des données dans l'architecture du processeur :

- ***registre (CPU) vers mémoire***
- ***registre (CPU) vers registre (CPU)***
- ***mémoire vers registre (CPU)***

Un déplacement mémoire vers mémoire en passant par le CPU n'est pas implémenté et aurait que peu d'intérêt (mémoire vers CPU suivi de CPU vers mémoire). Si nous souhaitons réaliser des transferts mémoire/mémoire sans passer par le cœur, les périphériques spécialisés de type DMA (Direct Memory Access) peuvent s'en charger (si votre processeur en possède).

Commençons par l'instruction MOV. Vous constaterez que cette instruction supporte un grand nombre de modes d'adressages. Ceci est typique d'un CPU CISC. En général, les CPU RISC implémentent moins de modes d'adressage avec des instructions dédiées à chaque mode.

- ***Adressage registre :*** *déplacement de données dans le CPU. Registre vers registre.*

```
mov     %ax, %bx
```

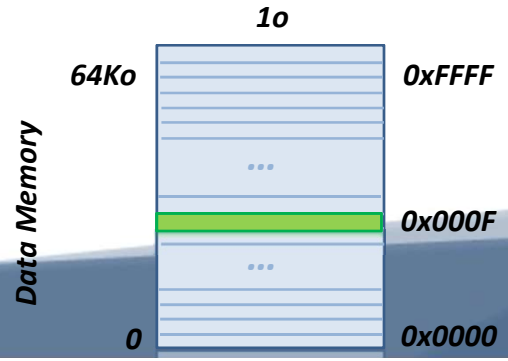
- ***Adressage immédiat :*** *affectation d'une constante dans un registre. Le déplacement d'une constante vers la mémoire est également possible.*

```
mov     $0x1A2F, %bx
```

Les modes d'adressage suivants manipulent tous la mémoire. ***Nous partirons pour le moment d'une hypothèse fausse. Supposons que nous ne pouvons manipuler que 64Ko de mémoire (données et programme unifiées) et donc des adresses sur 16bits uniquement****. Nous découvrirons la capacité mémoire réelle de 1Mo du 8086 lorsque nous présenterons la notion de segmentation.*

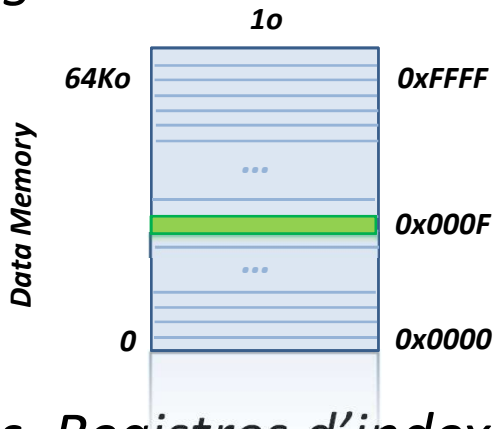
- ***Adressage direct :*** *déplacement de données du CPU vers la mémoire ou vice versa. L'adresse de la case mémoire à manipuler est directement passée avec l'opcode de l'instruction.*

```
mov        (0x000F), %bl
```



- ***Adressage indirect :*** *déplacement de données du CPU vers la mémoire ou vice versa. L'adresse de la case mémoire à manipuler est passée indirectement par un registre.*

```
mov     $0x000F, %bx
mov     (%bx), %al
```



- ***Adressage indexé :*** *non vu en cours. Registres d'index SI et DI.*

- ***Suffixes d'instruction :*** *permet de fixer le nombre d'octets à récupérer ou sauver en mémoire (uniquement en syntax AT&T).*

```
movb    %bl, %al     ;déplacement 1o
movw    %bx, %ax     ;déplacement 2o
movl    %ebx, %eax   ;déplacement 4o
                     ;(non supporté sur 8086)
```

Vous pouvez observer ci-dessous la totalité des modes d'adressage supportés sur architecture Intel x64 actuelle concernant l'instruction MOV "seule". La gestion de nombreux modes d'adressage implique une complexité accrue des unités de décodage et d'exécution :

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8***,r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8*.
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16*.
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32*.
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64*.
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r*
REX.W + 8C /r	MOV *r/m64,Sreg***	MR	Valid	Valid	Move zero extended 16-bit to *r/m64*.
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment re
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m* register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) t
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) t
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:o*
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*)
A2	MOV *moffs8*,AL	TD	Valid	Valid	Move AL to (*seg:offset*).
REX.W + A2	MOV *moffs8****,AL	TD	Valid	N.E.	Move AL to (*offset*).
A3	MOV *moffs16**,AX	TD	Valid	Valid	Move AX to (*seg:offset*).
A3	MOV *moffs32**,EAX	TD	Valid	Valid	Move EAX to (*seg:offset*).
REX.W + A3	MOV *moffs64**,RAX	TD	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb*	MOV *r8, imm8*	OI	Valid	Valid	Move *imm8* to *r8*.
REX + B0+ *rb*	MOV *r8***, imm8*	OI	Valid	N.E.	Move *imm8* to *r8*.
B8+ *rw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16*.
B8+ *rd*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32*.
REX.W + B8+ *rd*	MOV *r64, imm64*	OI	Valid	N.E.	Move *imm64* to *r64*.
C6 /0	MOV *r/m8, imm8*	MI	Valid	Valid	Move *imm8* to *r/m8*.
REX + C6 /0	MOV *r/m8***, imm8*	MI	Valid	N.E.	Move *imm8* to *r/m8*.
C7 /0	MOV *r/m16, imm16*	MI	Valid	Valid	Move *imm16* to *r/m16*.
C7 /0	MOV *r/m32, imm32*	MI	Valid	Valid	Move *imm32* to *r/m32*.
REX.W + C7 /0	MOV *r/m64, imm32*	MI	Valid	N.E.	Move *imm32 sign extended to 64-bits* to *r/m64*.

Quelque soit le langage d'assemblage rencontré, les opérandes manipulées par une instruction seront toujours l'une de celles qui suit :

- ***Immédiat** (constante) : imm>reg ou imm>mem*
- ***Registre** (contenant une donnée ou une adresse) : reg>reg*
- ***Addresse :** reg>mem ou mem>reg ou branchement mem*

Les combinaisons présentées ci-dessus permettent d'accéder et de manipuler la totalité de l'architecture du processeur.

- ***Instructions arithmétique :*** *attention, les modes d'adressage supportés diffèrent d'une instruction à une autre. Etudions quelques instructions arithmétique en mode d'adressage registre :*

```
movb    $14,%al     ; al=0x0E (4cy)
movb    $2,%bl      ; bl=0x02 (4cy)
add     %bl,%al     ; al=0x10 (3cy)
mul     %bl         ; ax=0x0020 (70-77cy)
div     %bl         ; al=0x10 (quotient)
                    ; bl=0x00 (reste)
                    ; (80-90cy)
sub     %bl,%al     ; al=0x10 (3cy)
```

- ***Instructions logique :*** *manipulation bit à bit de données :*

```
movb    $15,%al     ; al=0x0F (4cy)
movb    $0x01,%bl   ; bl=0x01 (4cy)
and     $0xFE,%al   ; al=0x0E (4cy)
or      %bl,%al     ; al=0x0F (3cy)
not     %al         ; al=0xF0 (3cy)
                    ; Complément à 1
shl     $1,%al      ; al=0xE0 (2cy)
                    ; flag carry C=1
```

Les instructions de saut ou de branchement en mémoire programme peuvent être conditionnels ou inconditionnels. En langage C, elles permettent par exemple d'implémenter : if, else if, else, switch, for, while, do while, appels de procédure.

- ***Structures de contrôle :*** *Observons une partie des instructions de saut conditionnelles. Elles utilisent toutes les flags retournés par l'ALU et doivent être pour la plupart utilisées après une instruction arithmétique, logique ou de comparaison.*

instruction	nom	condition
`JZ label`	Jump if Zero	saut si ZF = 1
`JNZ label`	Jump if Not Zero	saut si ZF = 0
`JE label`	Jump if Equal	saut si ZF = 1
`JNE label`	Jump if Not Equal	saut si ZF = 0
`JC label`	Jump if Carry	saut si CF = 1
`JNC label`	Jump if Not Carry	saut si CF = 0
`JS label`	Jump if Sign	saut si SF = 1
`JNS label`	Jump if Not Sign	saut si SF = 0
`JO label`	Jump if Overflow	saut si OF = 1
`JNO label`	Jump if Not Overflow	saut si OF = 0
`JP label`	Jump if Parity	saut si PF = 1
`JNP label`	Jump if Not Parity	saut si PF = 0

condition	nombres signés	nombres non signés
=	`JEQ label`	`JEQ label`
>	`JG label`	`JA label`
<	`JL label`	`JB label`
≠	`JNE label`	`JNE label`

Prenons un exemple de programme C et étudions une implémentation assembleur possible. La solution n'est bien sûr pas unique :

```
unsigned char varTest = 0;

void main (void) {

    while (1) {
        if ( varTest == 0 ) {
            // user code if
        }
        else {
            // user code else
        }
    }
}
```

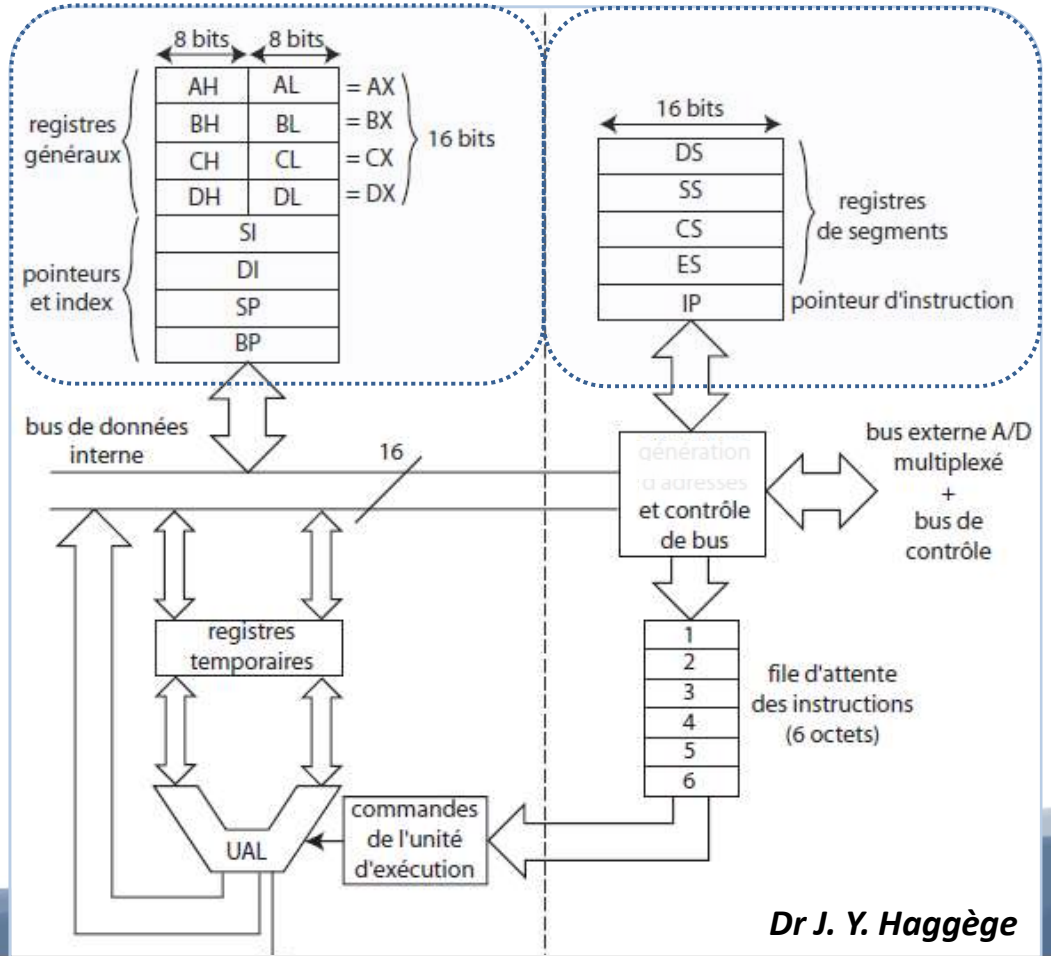
```
main:   mov     (addressVarTest),%al    ; al=0x00 (+8cy)
        mov     $0,%bl                  ; bl=0x00 (4cy)
        cmp     %bl,%al                 ; (3cy), flag Z=1
        jz      if1             ; IP = addresse if1
                                ; (16cy jump, 4cy no jump)
else1:  ; user code else
        ;…
        ;…
        ;…
        jmp     endif1          ; IP = addresse endif1 (15cy)
if1:    ; user code if
        ;…
        ;…
        ;…
endif1: jmp     main            ; IP = addresse main (15cy)
```

- ***Appel de procédure:*** *dans une premier temps, nous ne parlerons que des appels de procédure sans passage de paramètres. Cette partie sera vue dans la suite du cours lorsque nous aborderons la notion de pile ou stack. Juste après avoir vu la segmentation mémoire, notamment les segments SS=Stack Segment et CS=Code Segment).*

```
void fctTest (void);

void main (void) {

    while (1) {
        fctTest();
    }
}

void fctTest (void) {
    // user code
}
```

```
main:   call    fctTest     ; IP = adresse fctTest (19cy relatif)
        jmp     main        ; IP = adresse main (15cy)
        ;other code
        ;...
        ;...
fctTest: ; user code
        ;...
        ;...
        ;...
        ret                 ; IP = adresse jmp dans le main
                            ; (16-20cy)
```

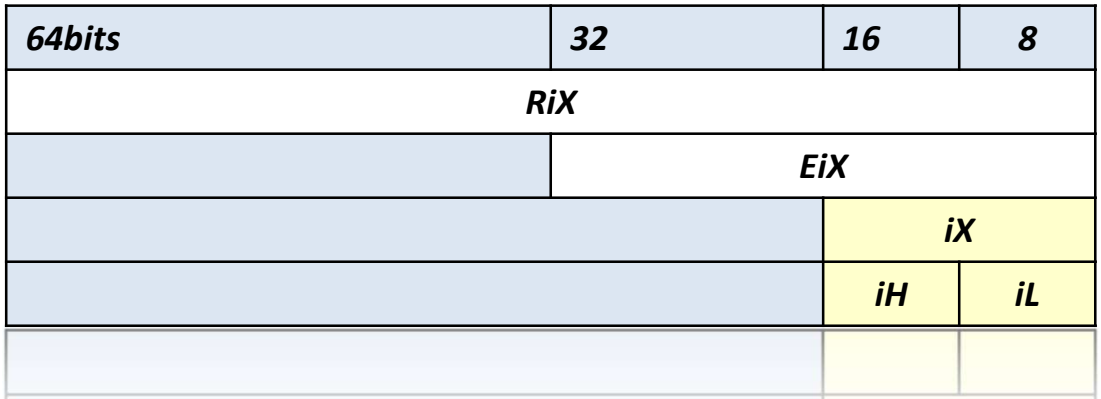
Rappelons les registres rencontrés sur architecture 8086 de Intel :



Dr J. Y. Haggège

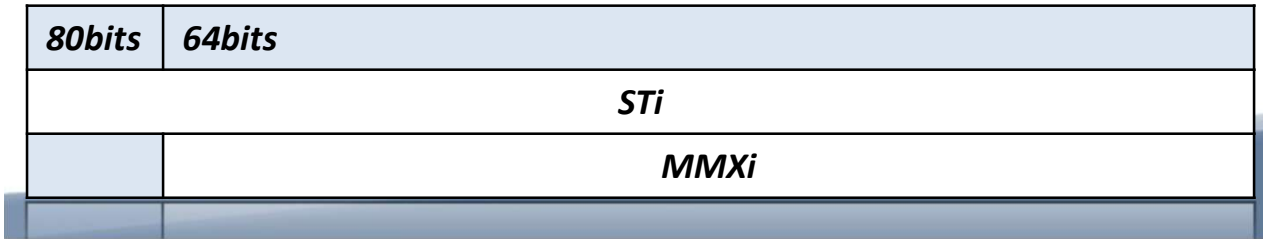
En même temps que les architectures des CPU's évoluent, le nombre et les tailles des registres de travail évoluent également. Observons les principaux registres de travail généralistes (iL/iH imbriqué dans iX imbriqué dans EiX lui-même imbriqué RiX). Nous parlerons d'environnement d'exécution :

General Purpose Registers (i = A, B, C, D)



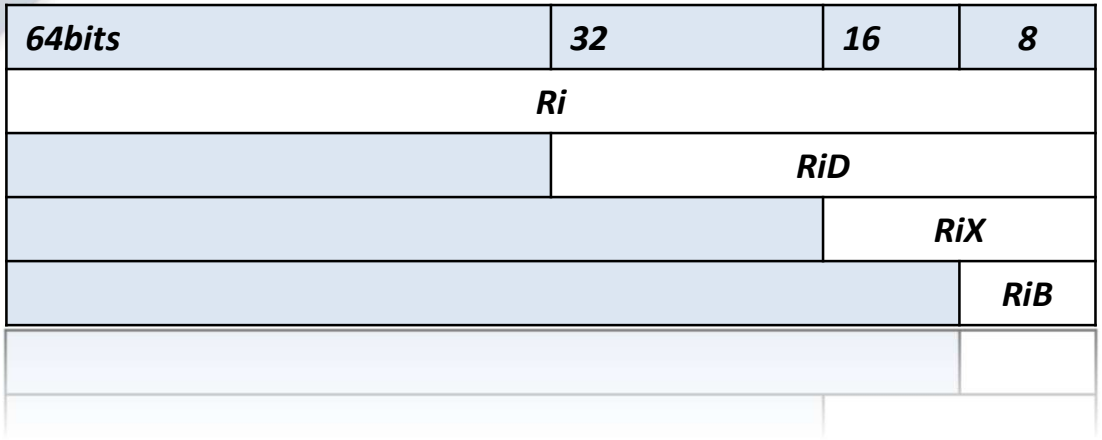
Depuis Intel 64 architecture (64bits mode-only)
Depuis 80386 architecture (E = Extended)
Depuis 8086 architecture

General Purpose Registers for Floating Point Unit : x87 and MMX extensions (i = 0 to 7)



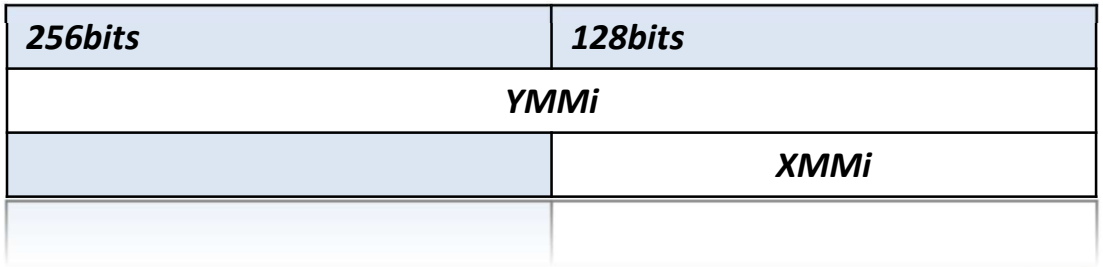
Depuis 80486 architecture (x87 extension)
Depuis Pentium MMX architecture

64bits mode-only General Purpose Registers (i = 8 to 15)



Depuis Intel 64 architecture (64bits mode-only)

General Purpose Registers for SIMD Execution Units (SSE extensions)
(i = 0 à 7 with Pentium III SSE)
(i = 0 à 15 with Intel 64)



Depuis Sandy Bridge architecture (AVX extension)

Depuis Pentium III architecture (SSE extension)

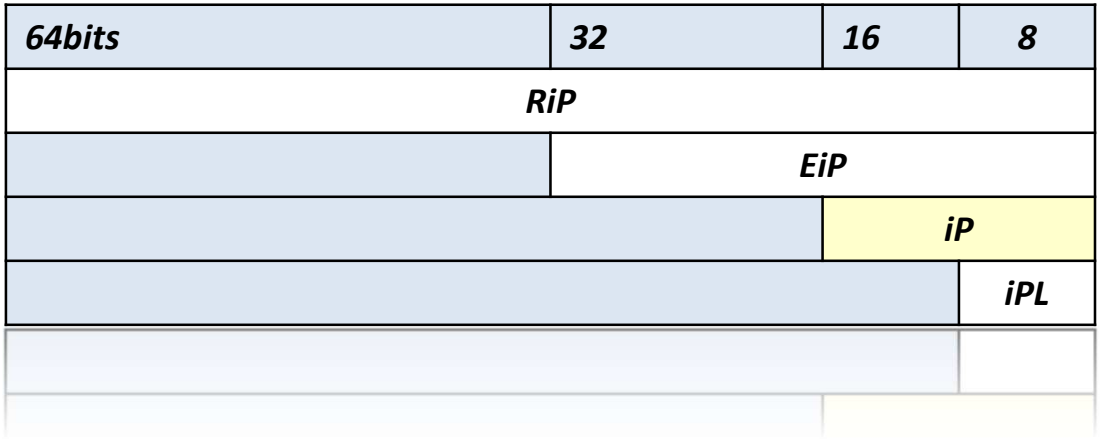
*Pour rappel, l'instruction **dpps** précédemment étudiée durant le chapitre précédent fut introduite avec l'extension SSE4.1 et utilise donc les registres 128bits XMMi :*

DPPS — Dot Product of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Selectively multiply packed SP floating-point values from *xmm1* with packed SP floating-point values from *xmm2*, add and selectively store the packed SP floating-point values or zero values to *xmm1*.

Registres pour la manipulation de pointeurs (SP, BP, SI, DI et xS) :

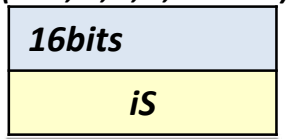
Pointer Registers (i = S and B)



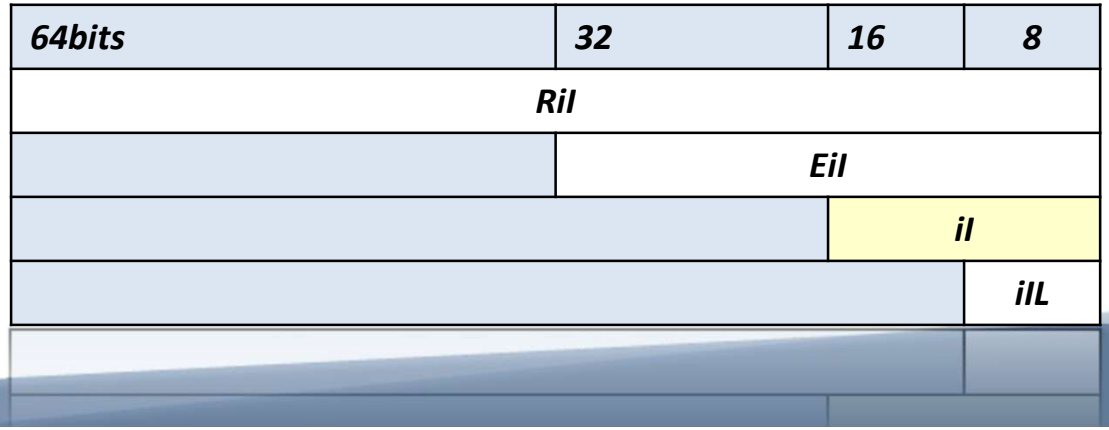
Depuis 8086 architecture

(64bits mode-only)

Segment Registers
(i = C, D, S, E, F and G)



Index Registers (i = S and D)



Depuis 8086 architecture

(64bits mode-only)

Instruction Pointer Register

64bits	32	16	8
RIP			
	EIP		
		IP	

Depuis 8086 architecture

D'autres registres divers ou spécialisés sont également arrivés au cours des évolutions des architectures : Descriptor Table Registers (GDTR, LDTR, IDTR), task register (TR), control registers CR0-CR8 64bits mode-only …

Environnements d'exécution en modes 32bits et 64bits :

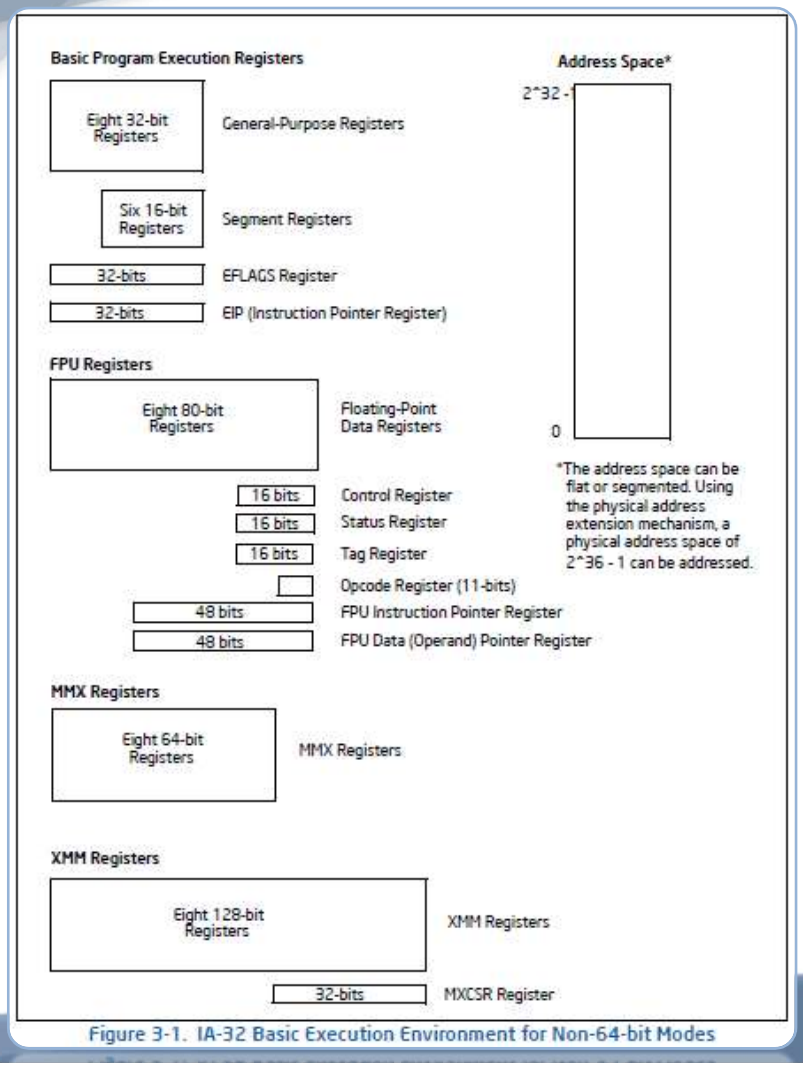


Figure 3-1. IA-32 Basic Execution Environment for Non-64-bit Modes

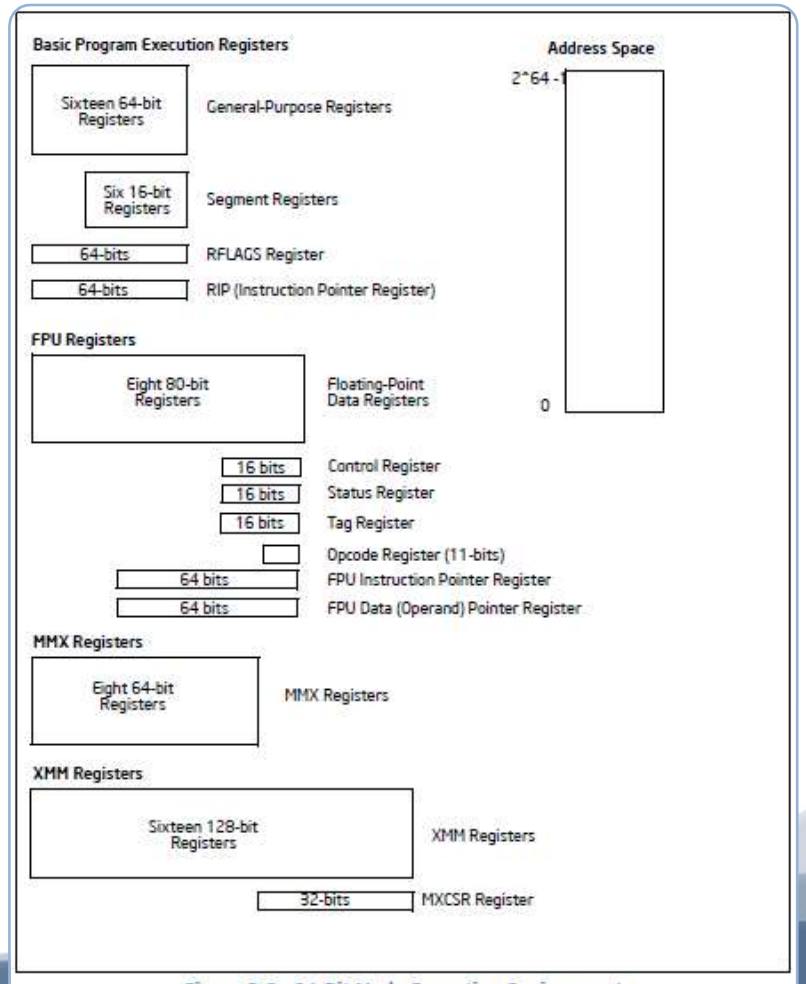


Figure 3-2. 64-Bit Mode Execution Environment

ENSICAEN
Computer Science

Merci de votre attention !