Programmation

Pr. Olivier Gruber

olivier.gruber@univ-grenoble-alpes.fr

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Info3 – Programmation – Overview

- Une fondation solide – des compétences complémentaires
  - Architecture matérielle
  - Programmation Assembleur
  - Programmation C
  - Programmation orientée-objet en Java
- Compétences développées tout au long de l'année
  - Sur les deux semestres et plusieurs cours
    - Enseignement transverse de la programmation
  - Projets au second semestre
    - En C, sur carte pour systèmes embarqués
    - En Java, un jeu graphique

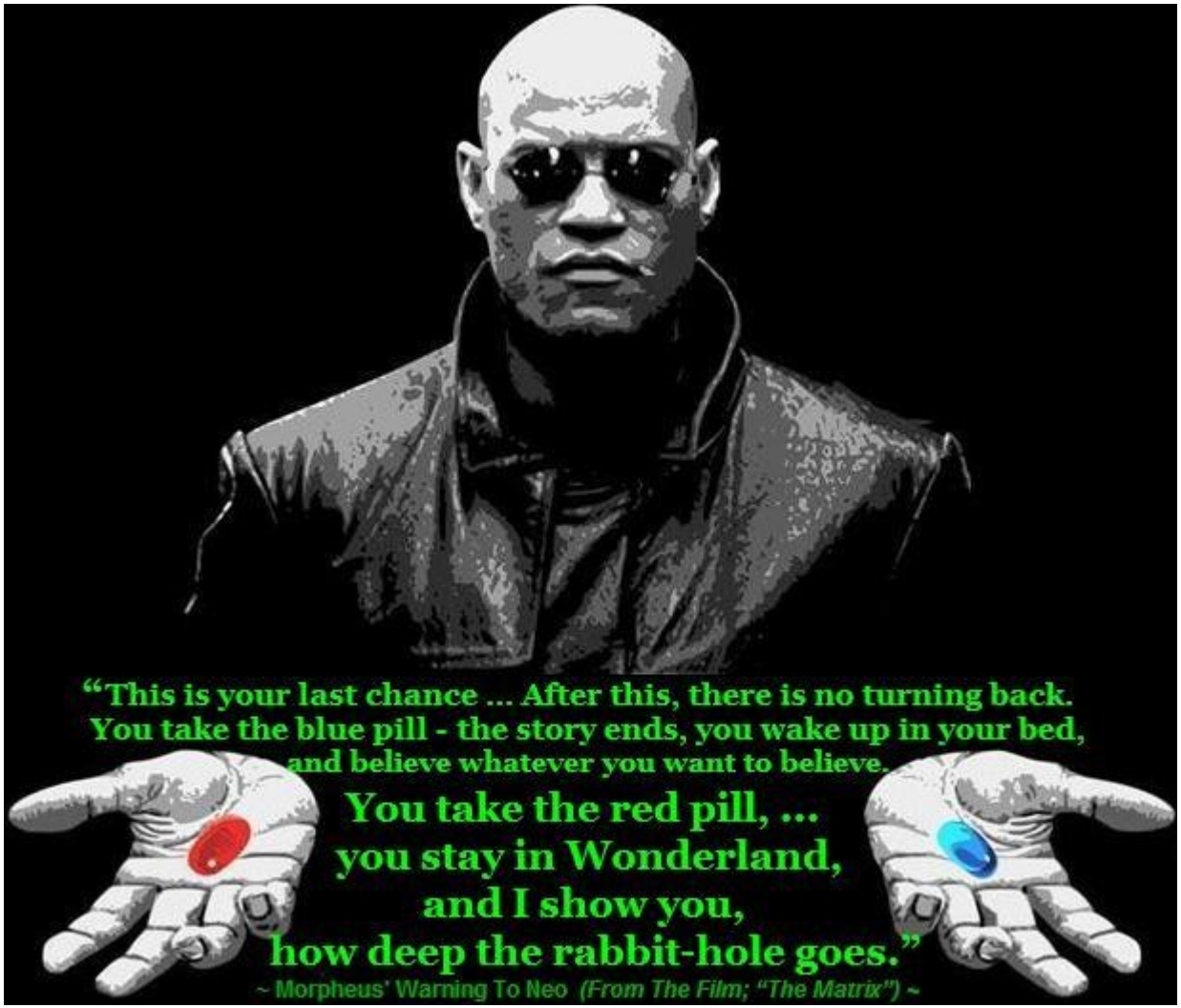
Cours PROG – Septembre

- Prélude – vous préparer au semestre
  - Complémentaire du cours sur Linux
  - **Techniquement :** revoir les bases de la programmation
  - **Humainement :** établir l'entraide, l'égalité des chances
- Approche Pédagogique
  - En autonomie, sur jusqu'à la fin de septembre (3 semaines)
  - Faire soi-même, pour apprendre
  - **Travailler tous ensemble, pour tous réussir**
  - Orienté vers la pratique, mais il faut comprendre les concepts
- Technologies
  - Programmation en C
  - GNU toolchain : gcc et gdb

Cours PROG – 3 semaines

- Un apprentissage guidé
  - Une succession de petites tâches d'apprentissage
  - Elles vous emmèneront vers la conception et le codage d'un jeu du pendu
  - Le but n'est pas le jeu, **le but est d'apprendre à savoir faire**
- Évaluation..
  - Rendre le jeu du pendu qui marche avant le 01 octobre (par email)
  - Vous devez l'avoir fait vous même
  - Vous devez le comprendre **parfaitement**
- Si nous avons un doute...
  - Oral surprise sur votre code
  - Vous devrez alors nous convaincre que vous l'avez écrit et compris

Rouge ou Bleu ?



A Partir d'Aujourd'hui...

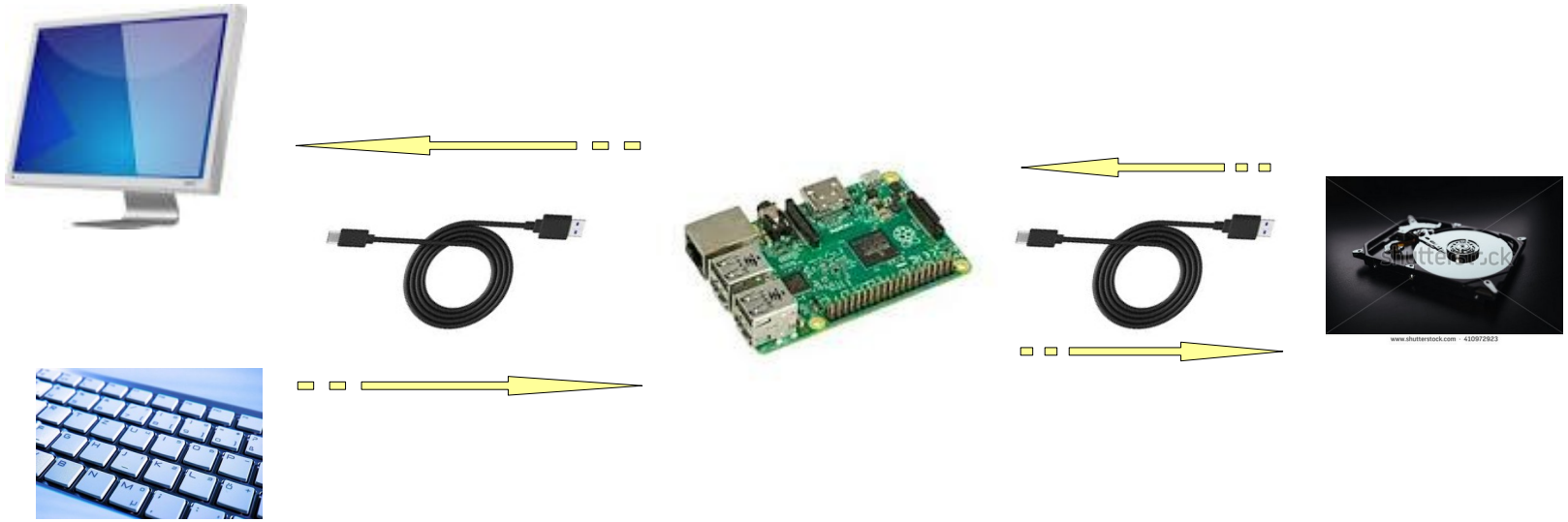
- Vous avez choisi un future métier
  - Ingénieur logiciel
- Les fondamentaux du métier
  - Le développement logiciel
- Quelle méthodologie de travail ?
  - Polytech est une école…
  - Mais vous n'êtes plus à l'école…
  - Il s'agit d'apprendre à savoir-faire...

Aujourd'hui… Les fondamentaux des fondamentaux...

- Architecture matérielle
  - La mémoire
  - Le processeur
  - Les périphériques
- Architecture logicielle
  - La notion d'instructions
  - La notion de flot d'exécution
  - Lecture et écriture en mémoire

Architecture Matérielle

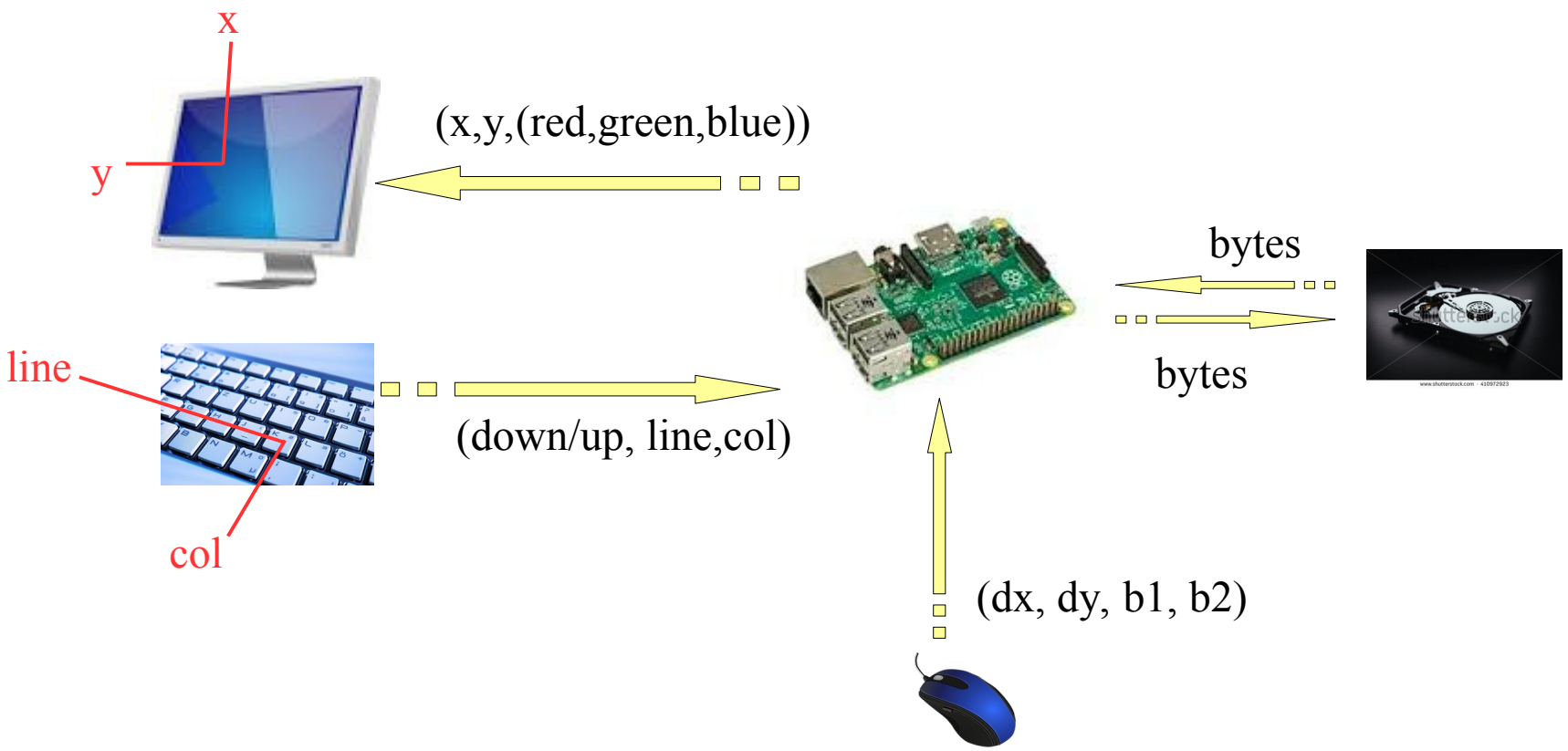
Une carte « mère » et des périphériques...



Une des plus belles machines inventées par l'homme…

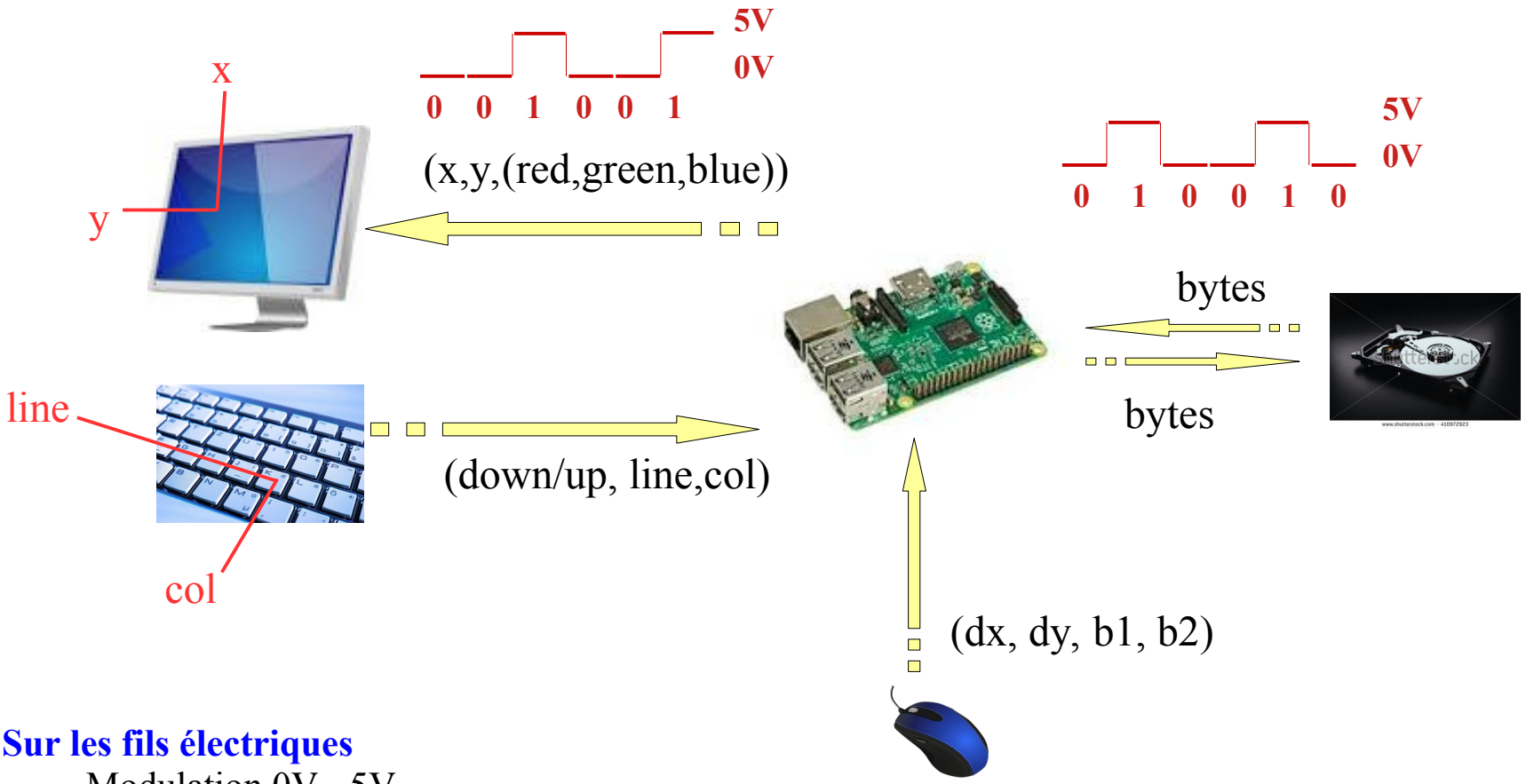
La roue…
L'écriture…
L'informatique...

Architecture Matérielle



Différentes interactions avec les périphériques,
tout se passe par échange de nombres entiers...

Architecture Matérielle



Sur les fils électriques

Modulation 0V - 5V

Interprétée comme une séquence de 0s et 1s

Les 0s et les 1s encodent des valeurs entières ou réelles

un pixel au coordonnée (x,y) avec la couleur (RGB)

un déplacement de souris (dx,dy)

Des lettres dans un fichier : ‘A’ ↔ 41 ou ‘B’ ↔ 42

Architecture Matérielle

- Le byte
  - Une boite pour 8 bits
  - Un bit vaut 0 ou 1
- Encodage en base 2 (*)
  - Bits de poids faible à droite

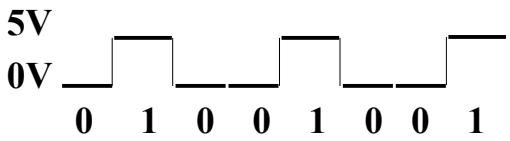
01001001

7 6 5 4 3 2 1 0 ← poids faible

$2^6+2^3+2^0$

01001001

Byte : 8 bits

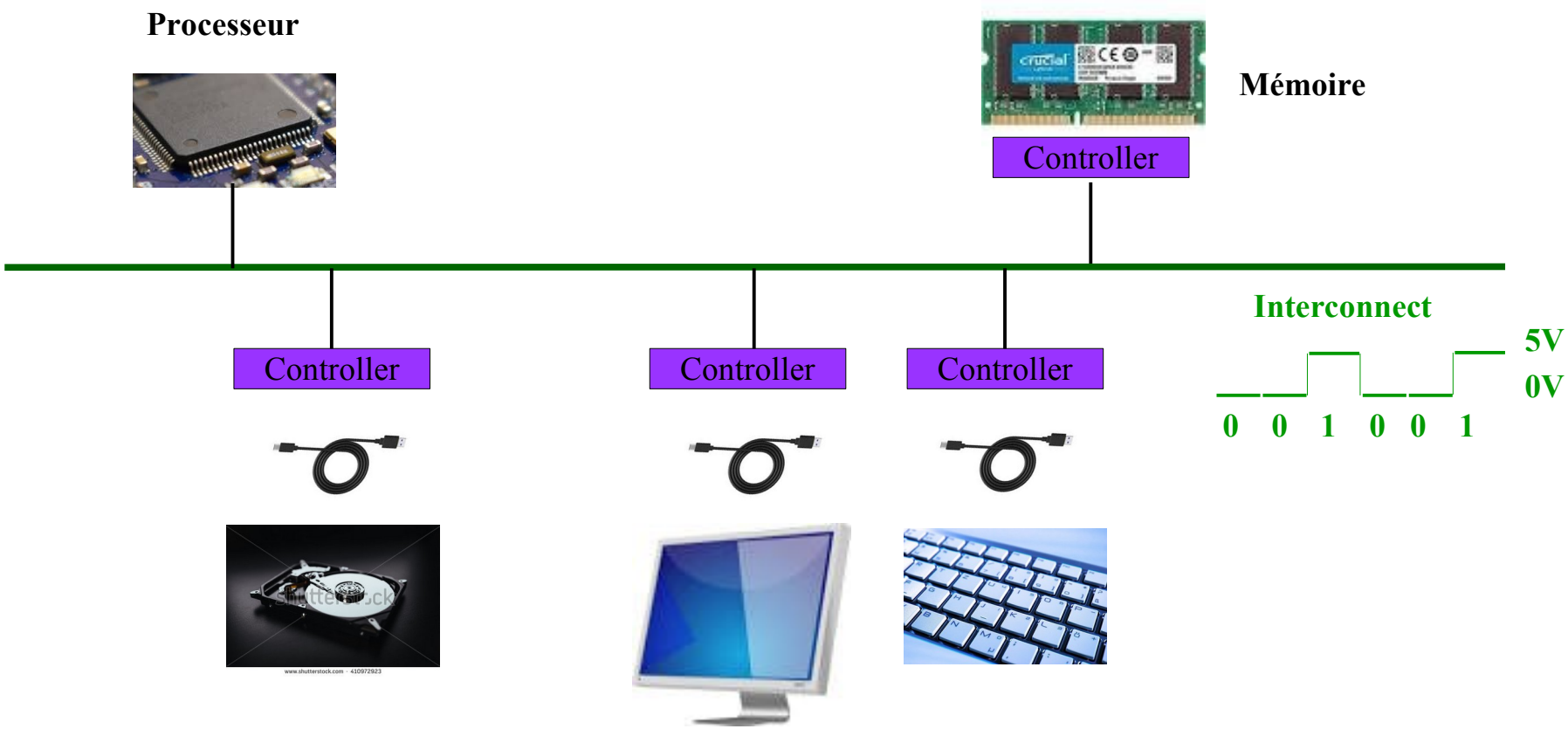


Binary	Decimal	Hexadecimal
01001001	73	49
$2^6+2^3+2^0$	$7*10^1+3*10^0$	$4*16^1+3*16^0$
b01001001	73	0x49

Binary : 0-1 Decimal : 0-9 HexaDecimal : 0-9-A-F

(*) Recherchez « système binaire » dans Google

Architecture Matérielle



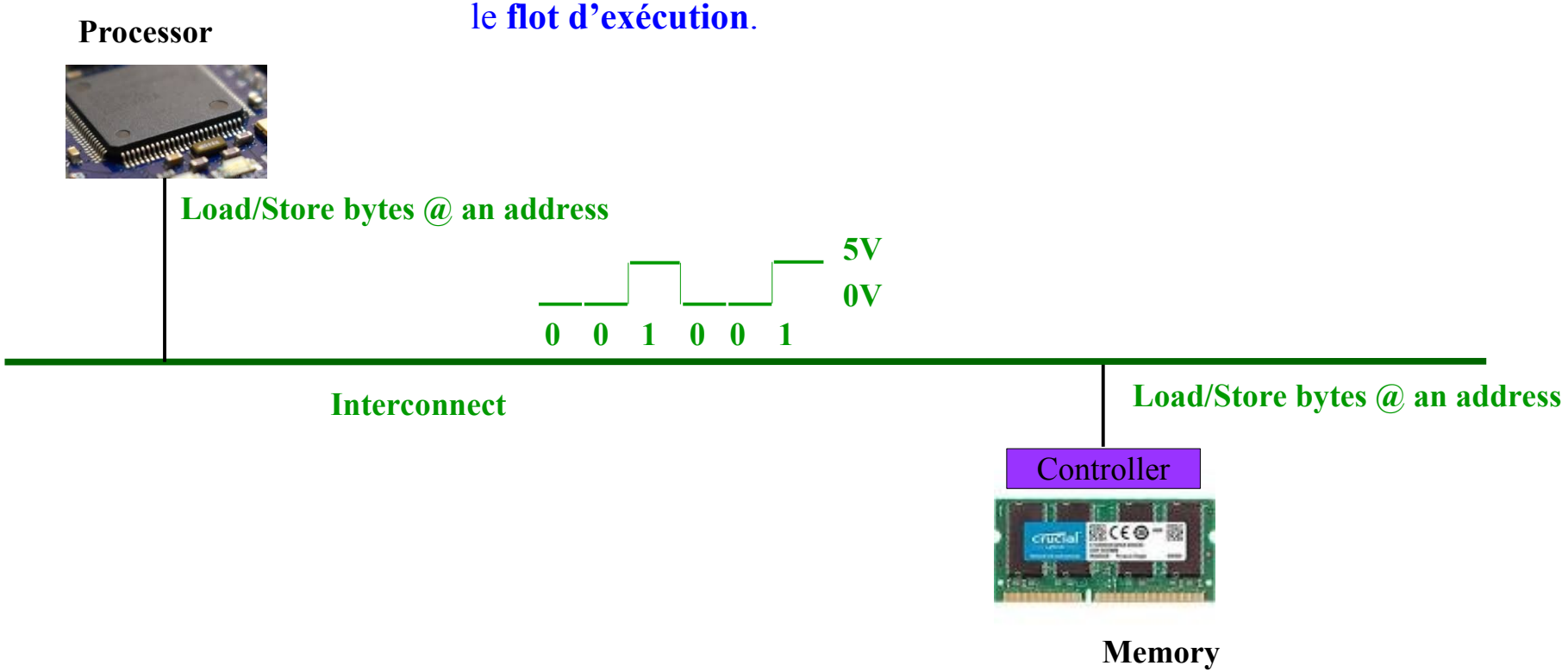
Architecture Matérielle

- Vue d'ensemble :

Le processeur sait **lire ou écrire des octets dans la mémoire à certaines adresses**… il manipule donc des données…

Mais aussi des instructions… **les instructions composent les programmes**.

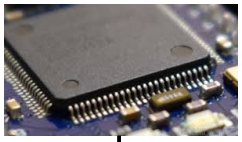
La suite d'instruction qu'exécute le processeur constitue le **flot d'exécution**.



Architecture Matérielle

- Les « *principaux acteurs* »
  - Le processeur, l'interconnect, et la mémoire
  - Load & store : data or instructions

Processor



Lit, comprends, et exécute les instructions d'une recette sur des « *valeurs* »

Les instructions sont elle-mêmes des valeurs

Interconnect

Transporte les «*valeurs*»
- vers une adresse
- depuis une adresse

Controller

Archive
et retrouve
les «*valeurs*»

Memory

Architecture Matérielle

- La mémoire
  - Les valeurs
    - 1 byte contient une valeur entre 0 et 255
  - Les adresses
    - 32bit : 4 * 1024 * 1024 = 4294967295 = 4GB
    - 64bit : $1{,}845 \times 10^{19}$

Controller

Les valeurs dans leur boite,
Bien rangées à leur adresse

Architecture Matérielle

- Interconnect
  - Relie le processeur à la mémoire
  - Transporte des valeurs et adresses

load(address) → value

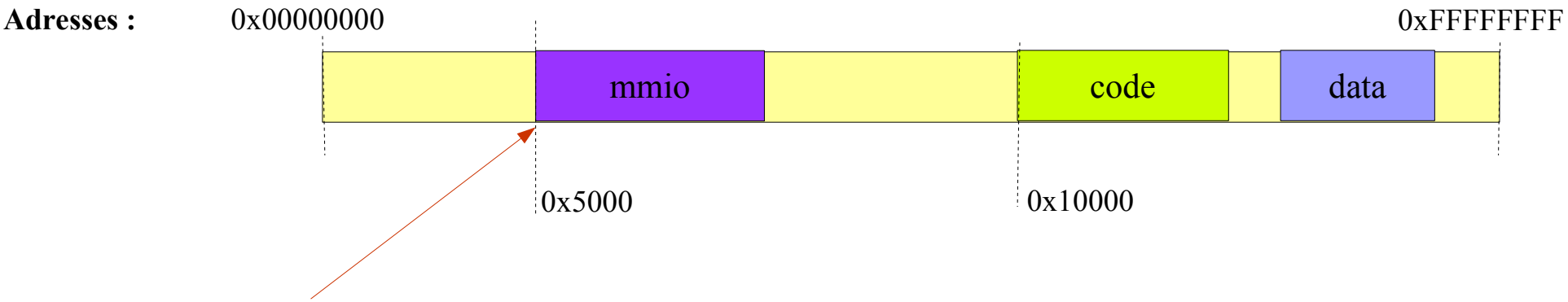
store(address, value)

Interconnect

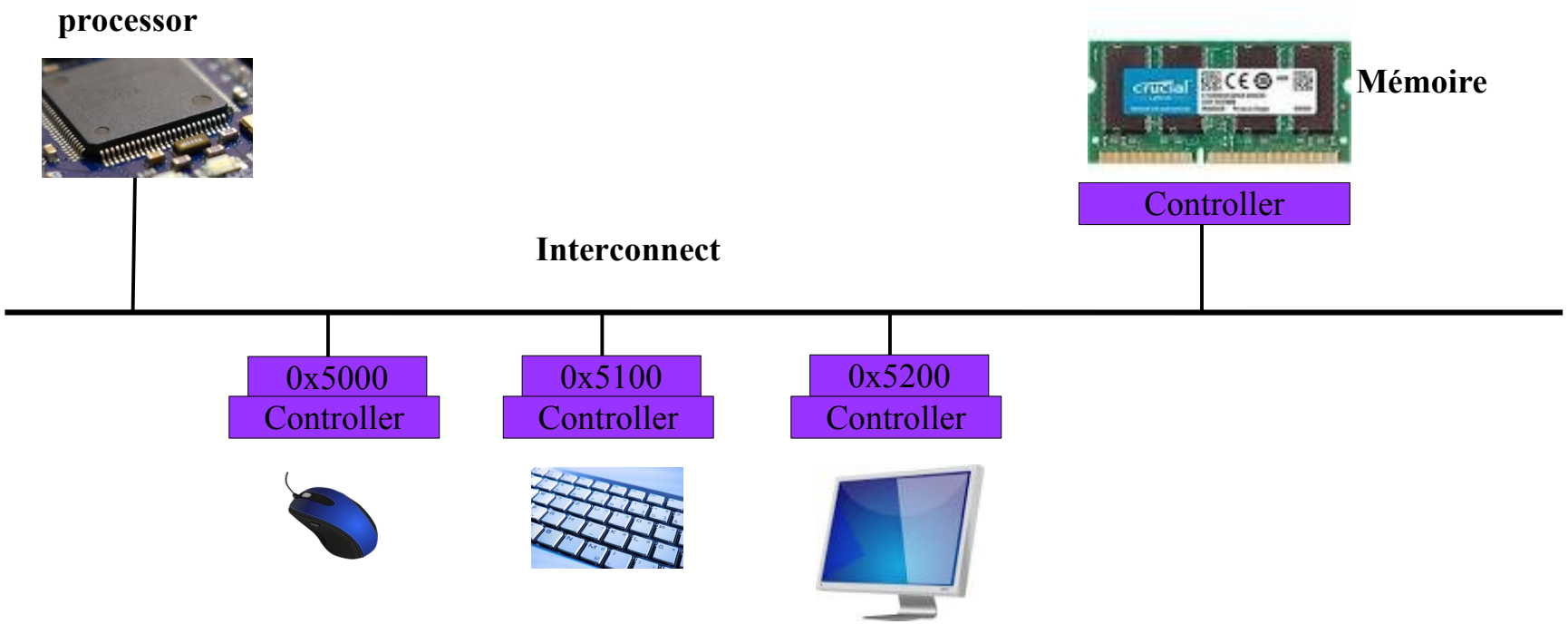
Transporte les «*valeurs*»
- vers une adresse
- depuis une adresse

Memory

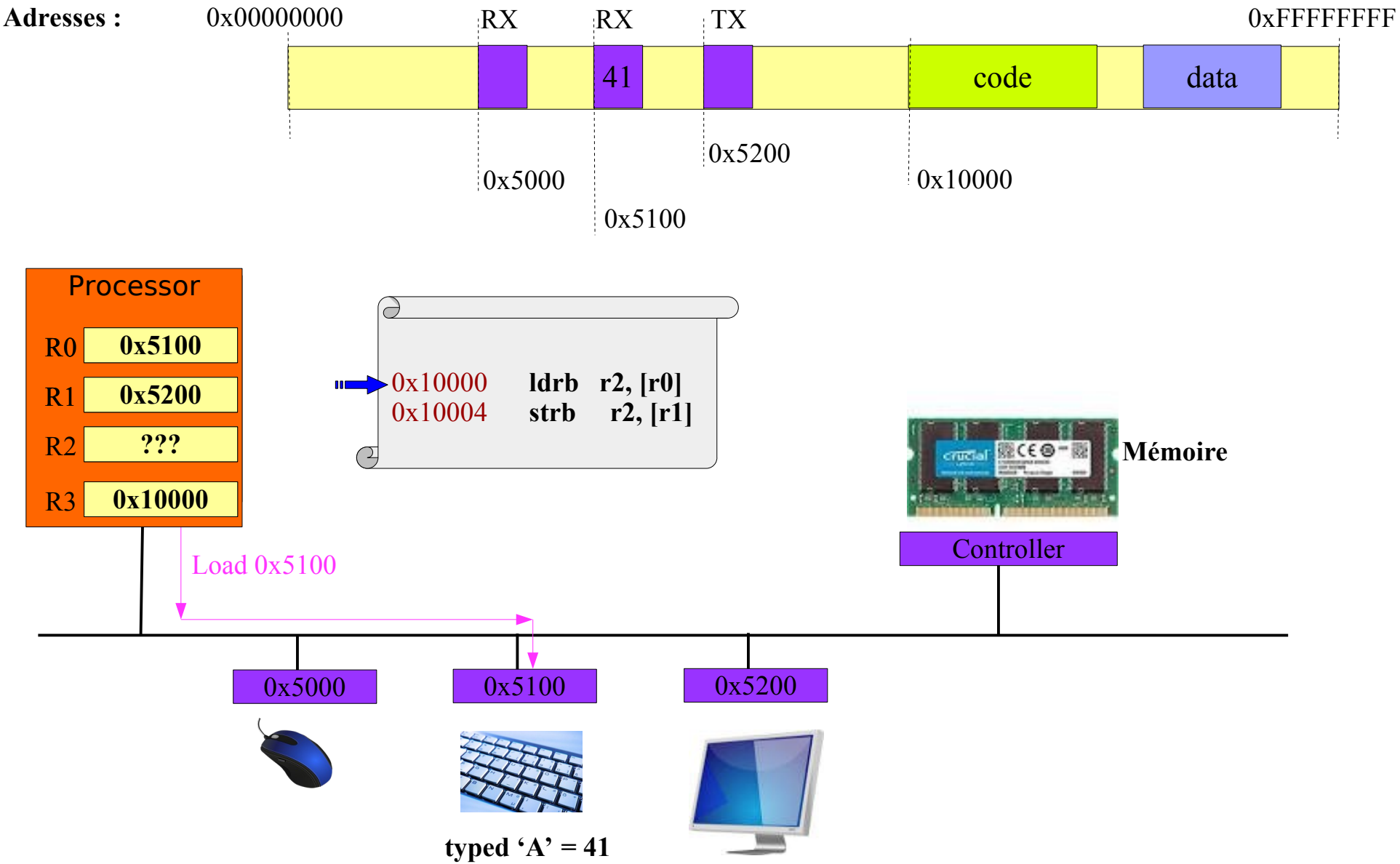
Périphériques – via MMIO registers



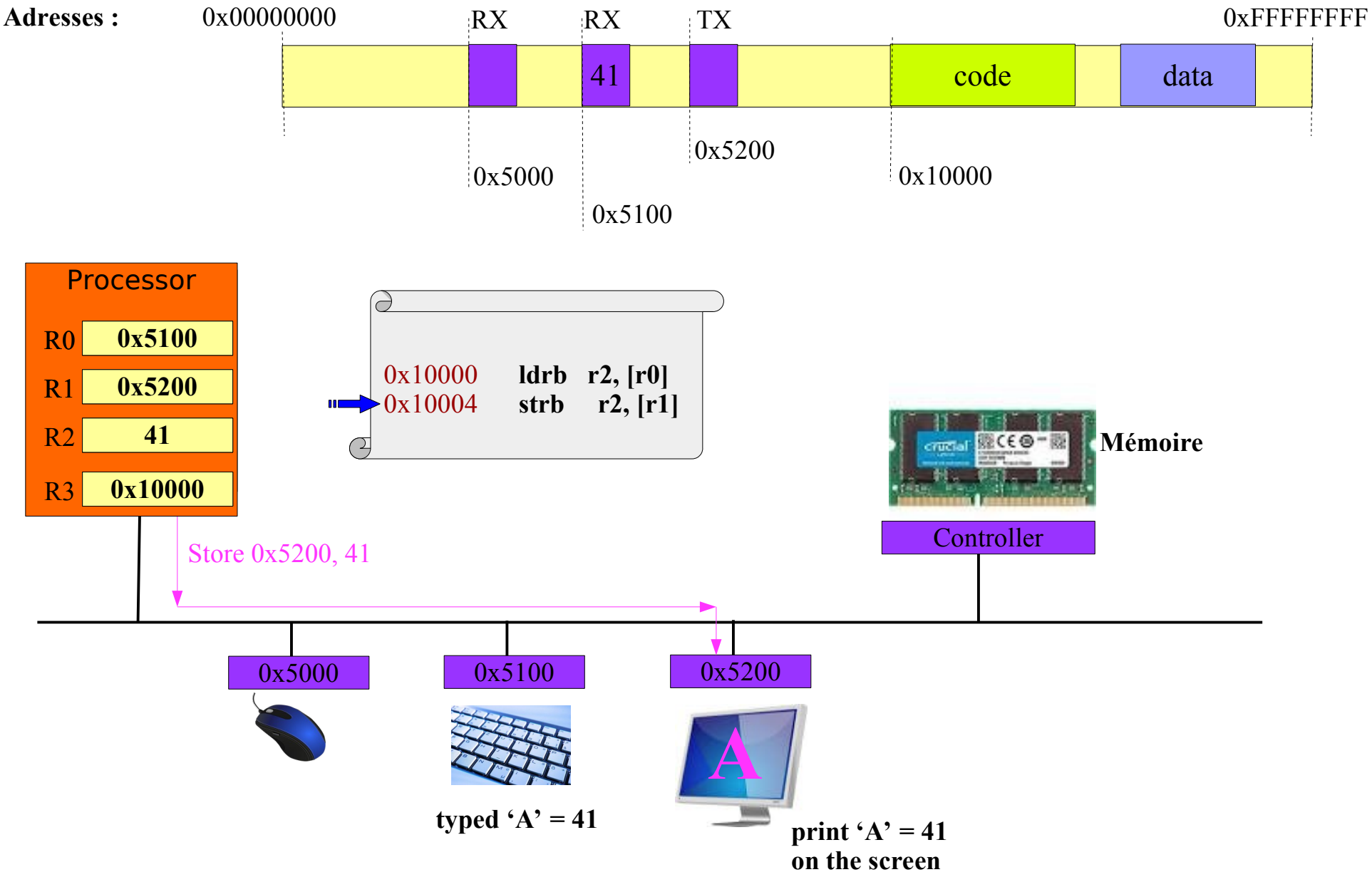
Memory-Mapped Input/Output Registers



Périphériques – Recevoir des valeurs

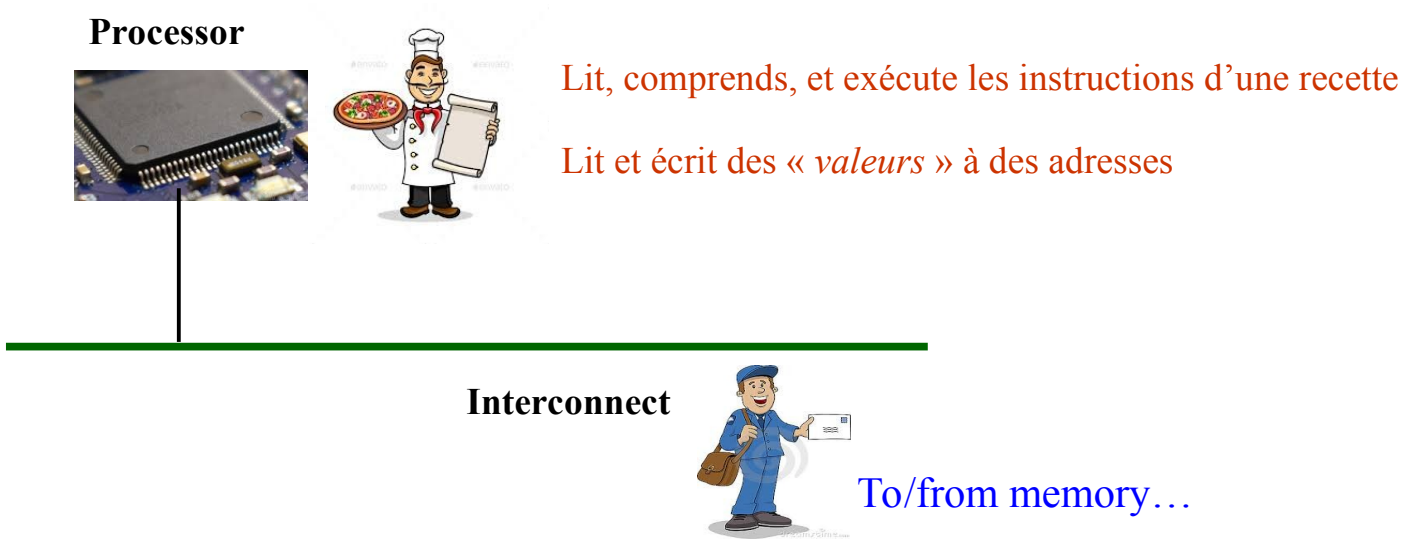


Périphériques – Envoyer des valeurs

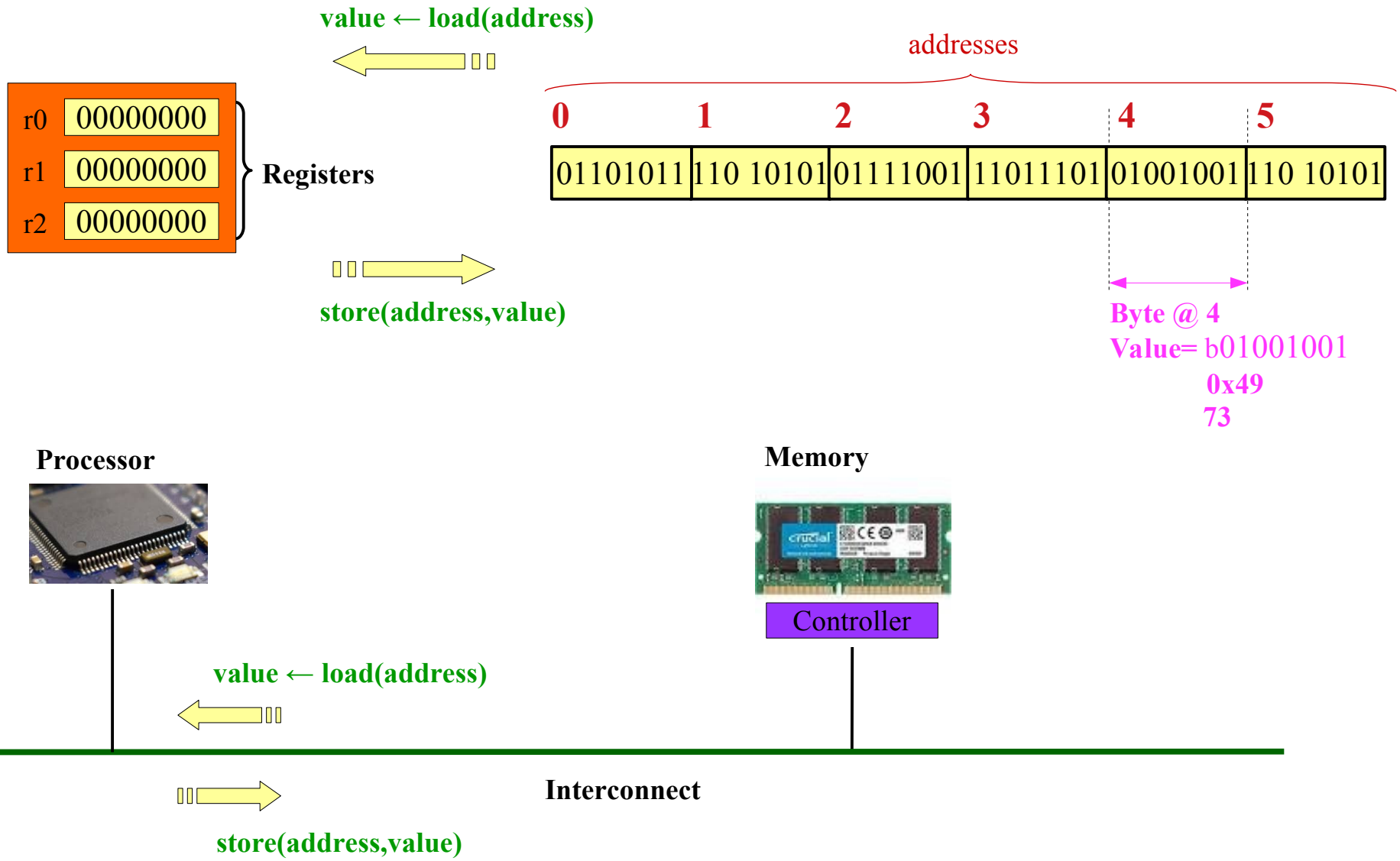


Le Processor

- Le processeur
  - Load/store de valeurs via l'interconnect
  - Pour les *instructions* du programme
  - Pour les *données* lues ou écrites par le programme dans la mémoire
  - Pour les *données* échangées par le programme avec les périphériques

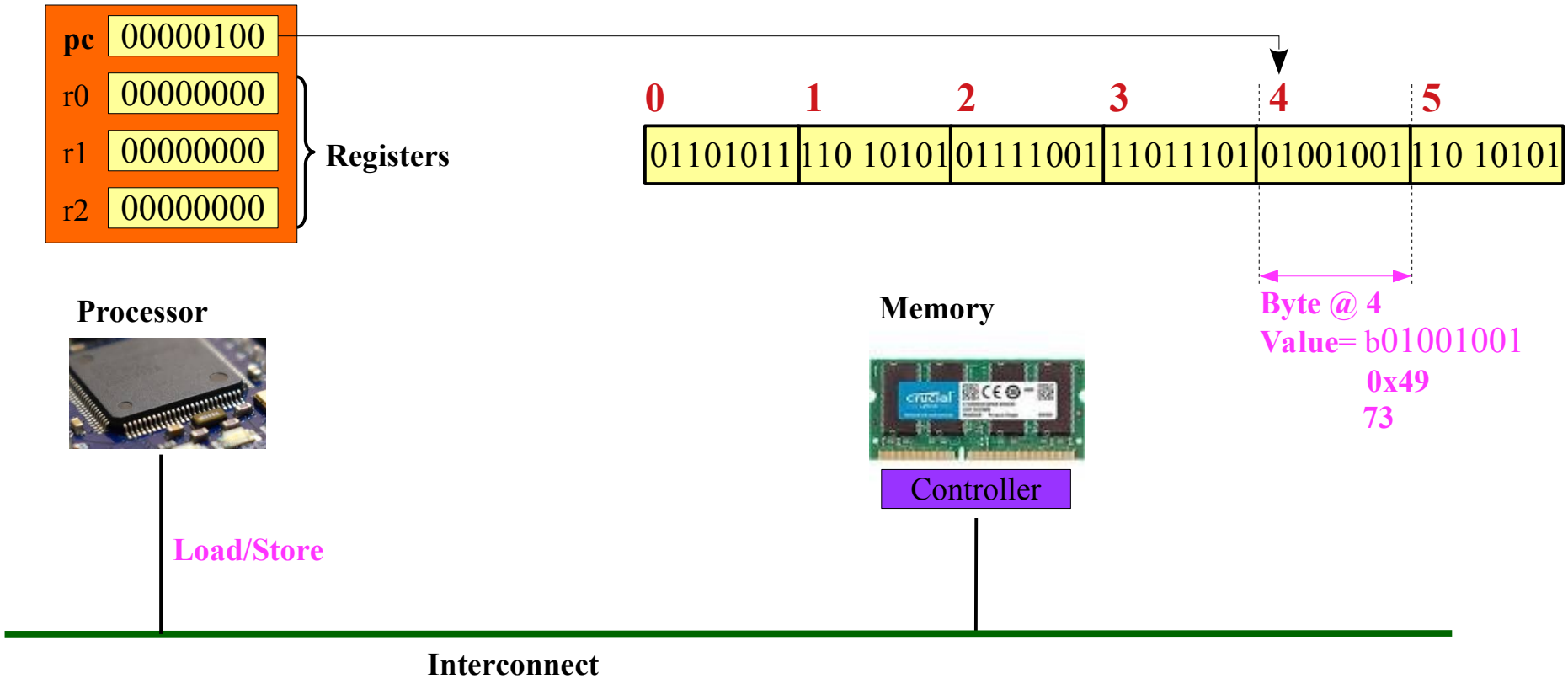


Le Processor et ses registres



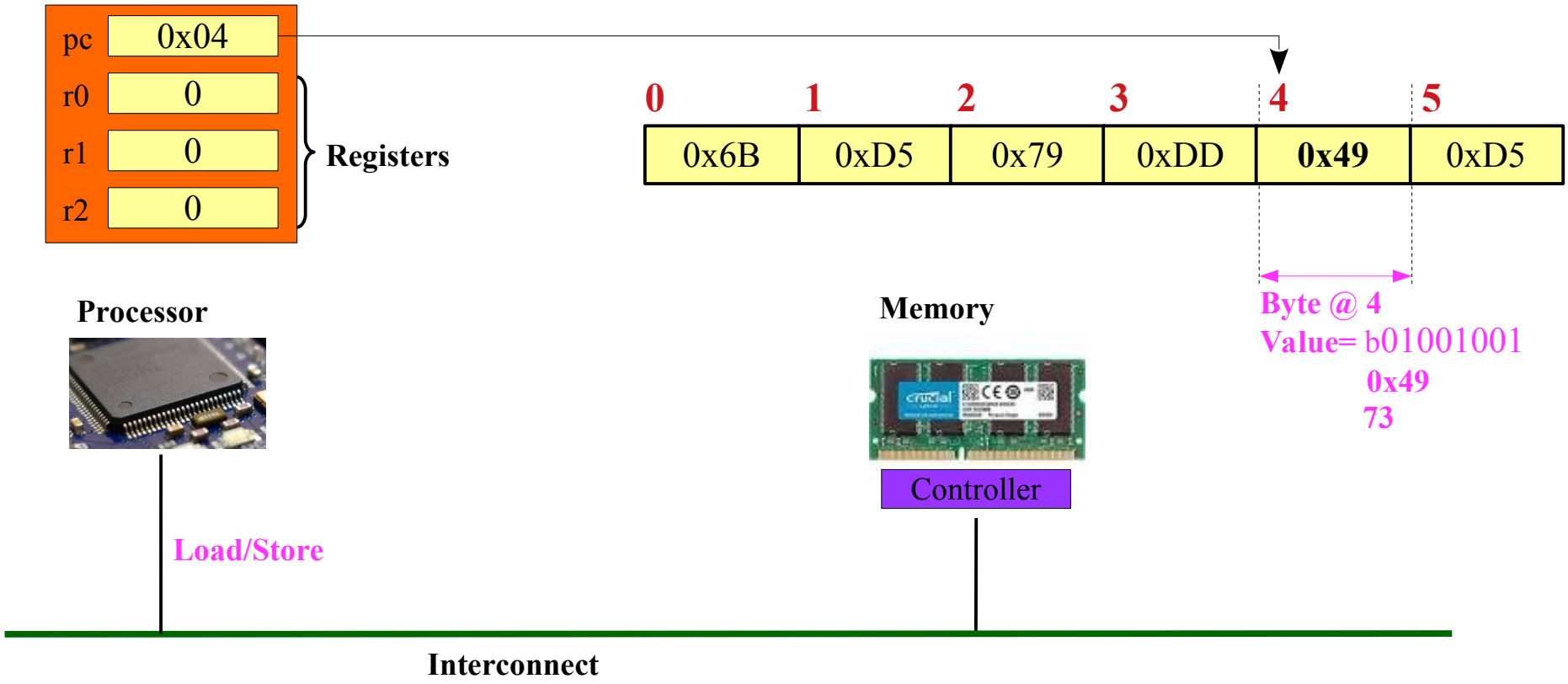
Le Processor et l'exécution

- Un registre spéciale : **program counter** (pc)
  - Le processeur exécute une boucle sans fin : *load – decode – execute*
  - *Traditionnellement appelé « fetch-decode-issue »*



Le Processor et l'exécution

- Un registre spéciale : program counter
  - Le processeur en boucle sans fin : fetch – *decode* – *issue*



Comment développer...

- Programmer avec des 0s et des 1s…
  - C'est le plus « basique » des langages de programmation
  - Mais pas top pour le développeur...

01001001	01101011	11011101

???

- Assembly Language

0x49	0x6B	0xDD

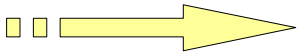
???

  - Le développeur écrit du texte
  - Ce texte est traduit par un outil, appelé un compilateur (assembleur)
  - Pour produire les 0s et les 1s que le processeur pourra comprendre

```
mov r0, #0x24
ldr   r1, [r0]
mov r2, #0x28
str   r1, [r0]
```

Le texte source du programme

La compilation

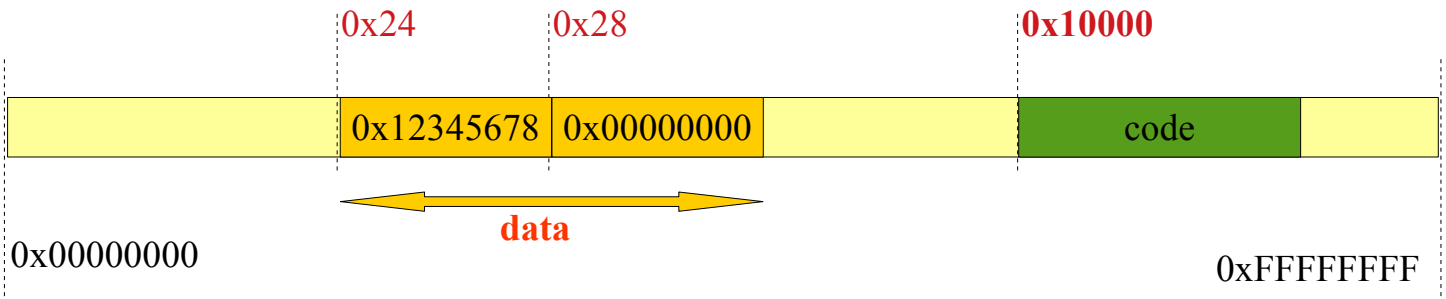


01001001	01101011	11011101	11010110

Le code binaire pour le processeur

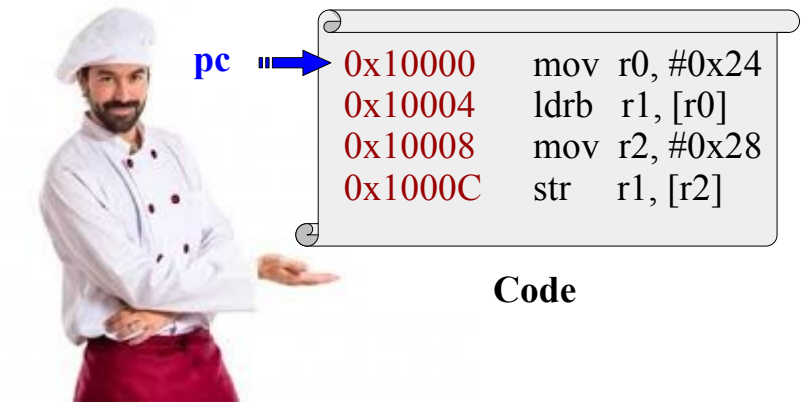
L'exécution

Memory: data + code

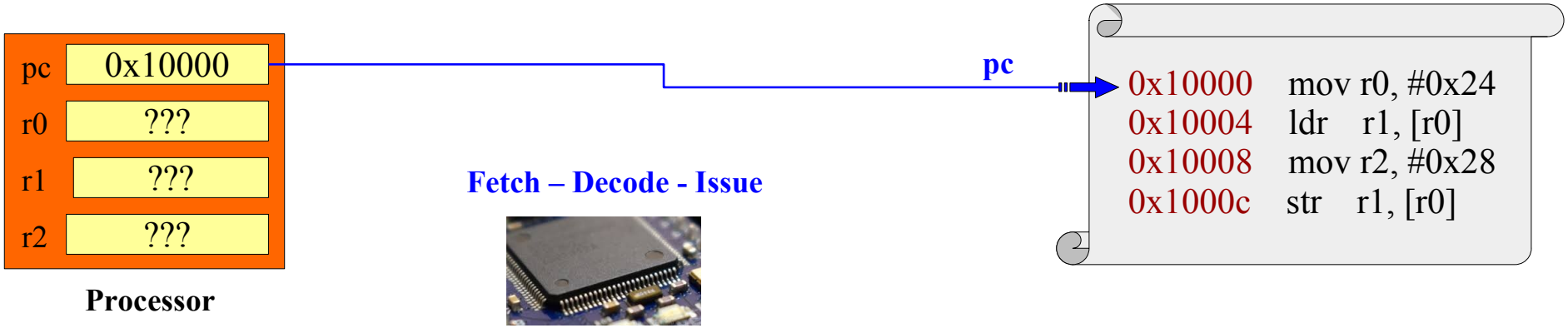


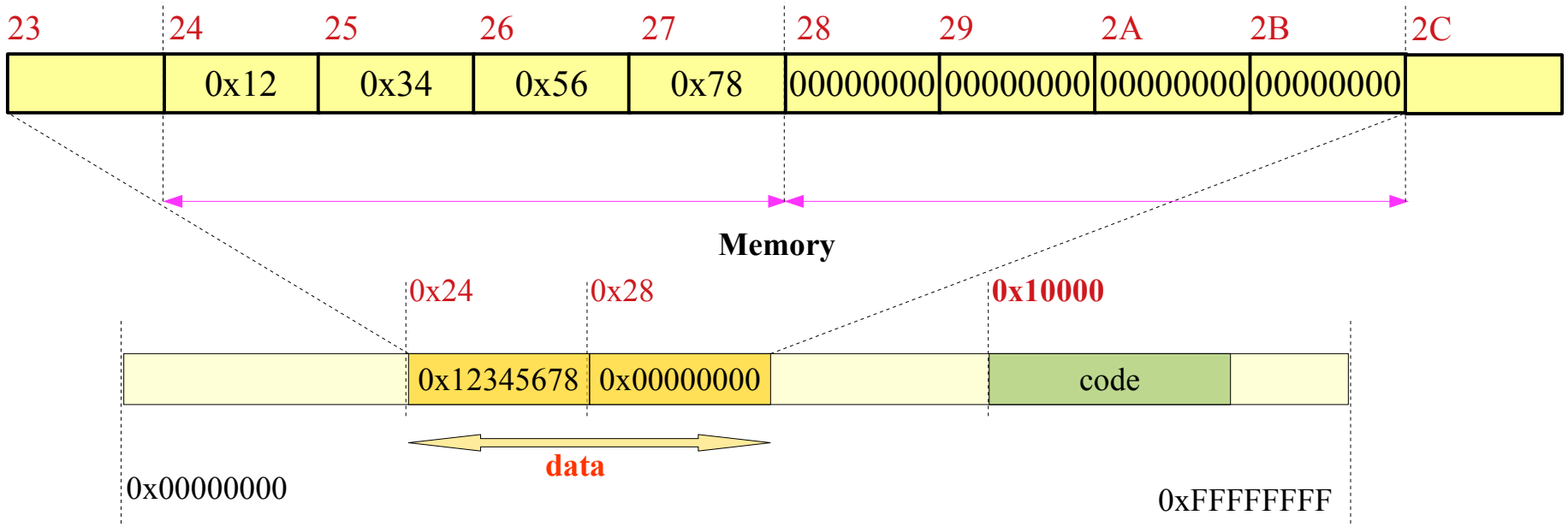
Processor :

- **Fetch** instruction **@pc**
- **Decode** instruction
- **Advance pc**
  pc = pc + sizeof(instruction)
- **Execute** instruction
- **Loop over**

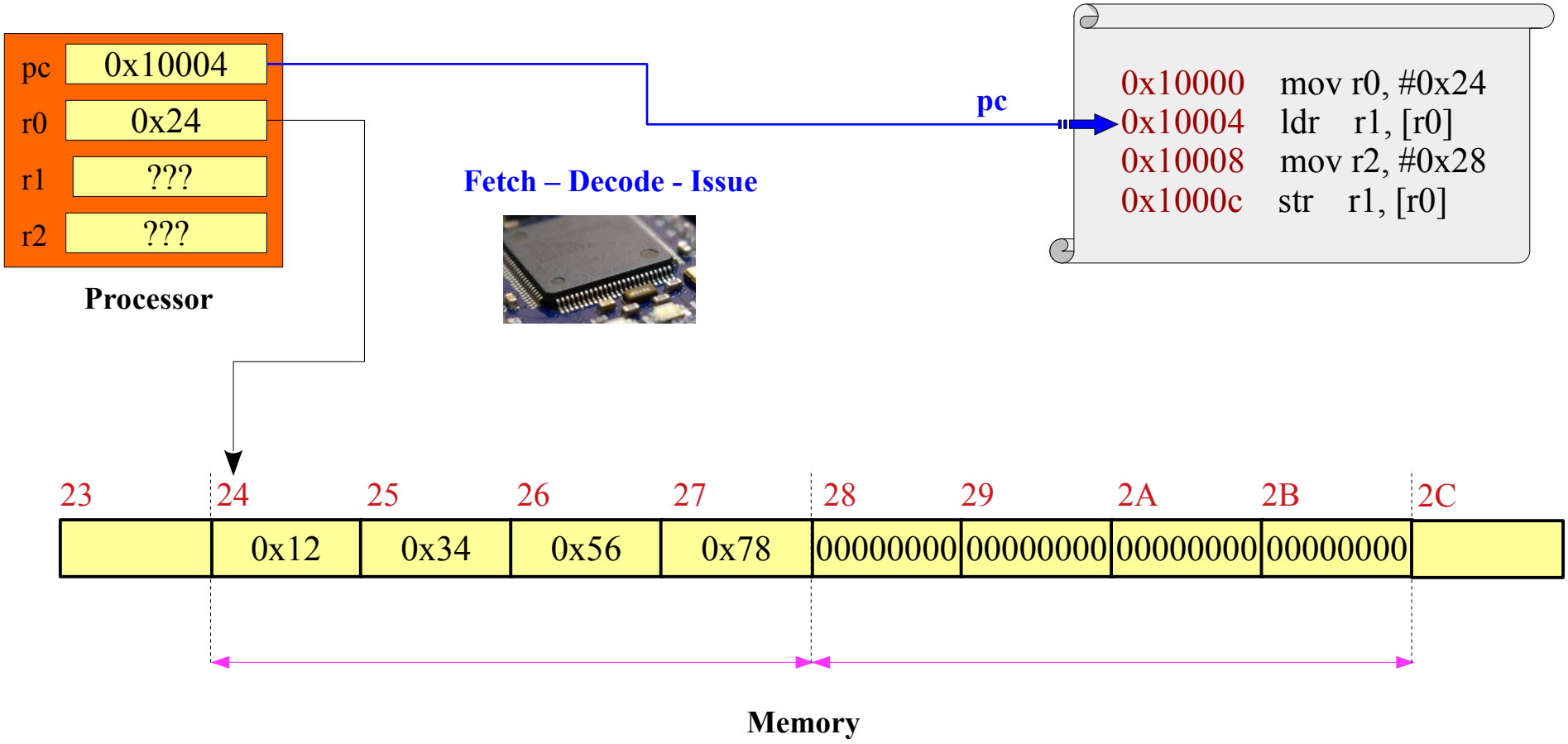


L'exécution...

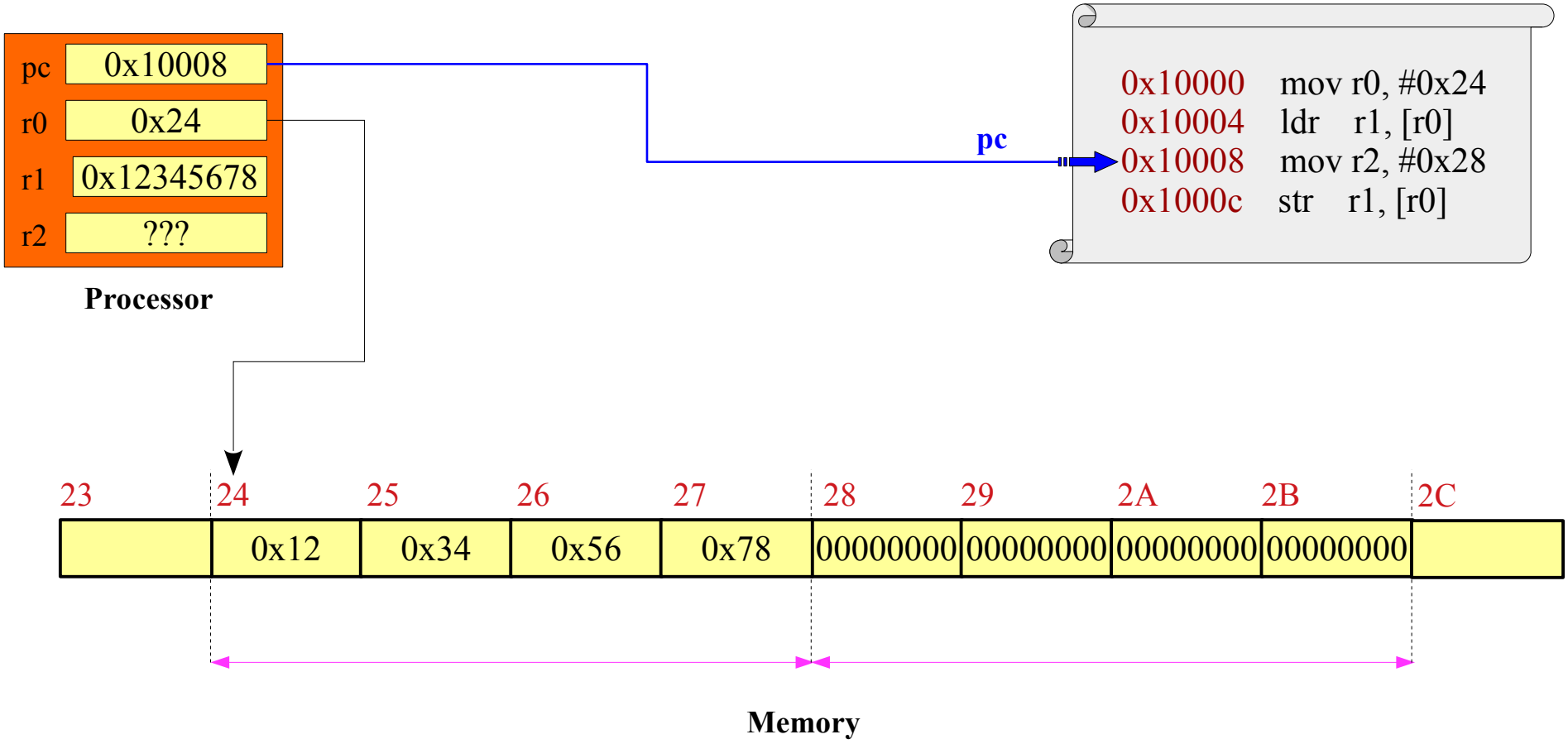




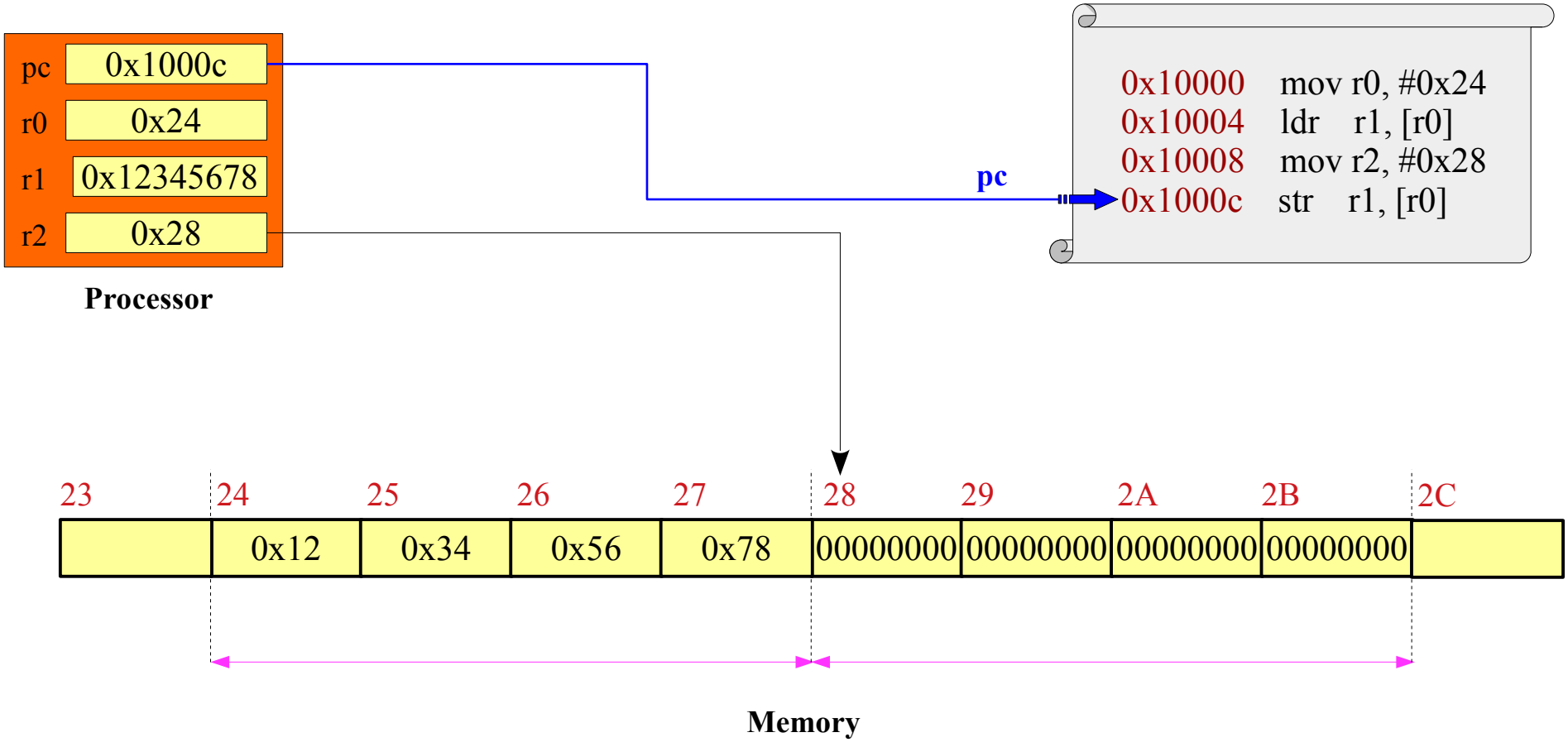
L'exécution...



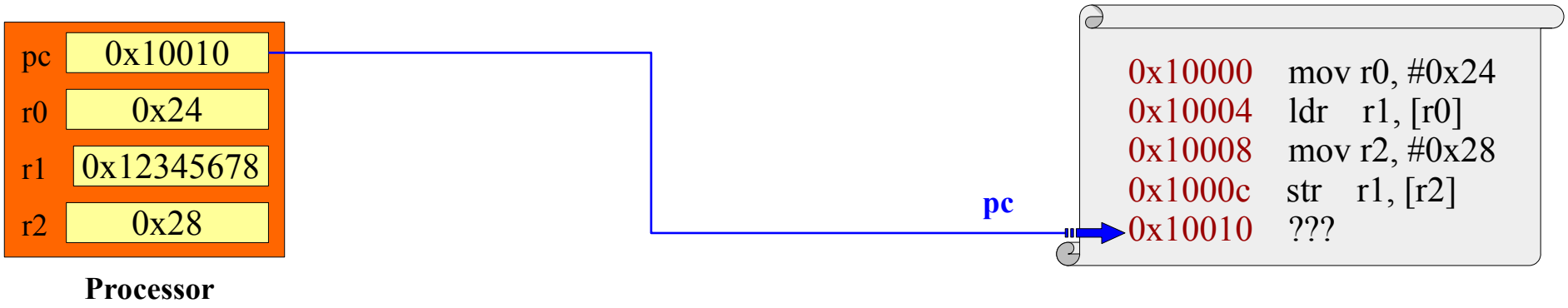
L'exécution...

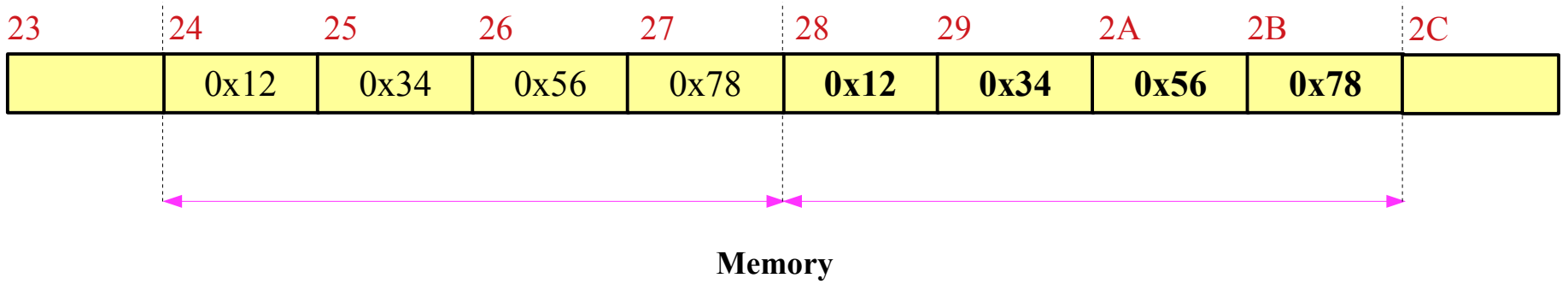


L'exécution...



L'exécution...





L'exécution...

Processor :

- **Fetch** instruction **@pc**
- **Decode** instruction
- **Advance pc**
  pc = pc + sizeof(instruction)
- **Execute** instruction
- **Loop over**

```
0x10000    mov  r3, pc
0x10004    mov  r0, #0x24
0x10008    ldrb  r1, [r0]
0x1000C    mov  r2, #0x28
0x10010    str   r1, [r2]
0x10014    mov  pc, r3
```

Que va-t-il se passer ?

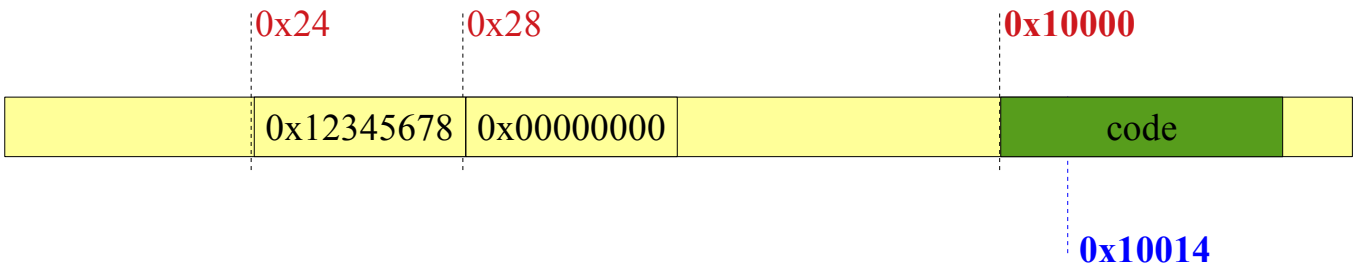
Le programme fait quelque chose d'utile ?

Et si 0x24 et 0x28 sont des registres mmio ?

Processor

pc	0x10014
r0	0x24
r1	0x12345678
r2	0x28
r3	???

Memory:



Assembly Language

Instructions:

- Load/Store from/to memory
- Move between registers
- **Compare instructions**
- **Conditional branches**
- **Arithmetic** (integer, floating)
- *Bitwise logical operations*

```
0x10000     mov  r0, #6
0x10004     bl     #0x10       ; branch-and-link
…

0x10014     mov  r1, #0
0x10018     mov  r2, #1
0x1001C     add   r1, r1, #1    ; r1 = r1 + 1
0x10020     mul  r2, r2, r1     ; r2 = r2 * r1
0x10024     cmp r1,r0           ; compare r1 & r0
0x10028     ble #0x1001C        ; branch if r1 <= r0
0x1002C     mov pc, lr          ; link register to pc
```

Code assembleur pour le calcul de factoriel

$$n ! = \prod_{1 \leqslant i \leqslant n} i = 1 \times 2 \times 3 \times \ldots \times (n - 1) \times n$$

Programming Languages

Assembly language:

- Le langage de programmation le plus bas, juste au dessus des 0s et des 1s

Le langage C:

- Un langage de programmation plus naturel pour les développeurs, qui doivent gérer encore beaucoup d'aspects de la machine (difficile)

Le langage Java:

- Une langage de programmation orienté-objet qui abstrait la machine complètement, plus facile et plus puissant.

```
int n = fact(6);
```

$$n ! = \prod_{1 \leqslant i \leqslant n} i = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

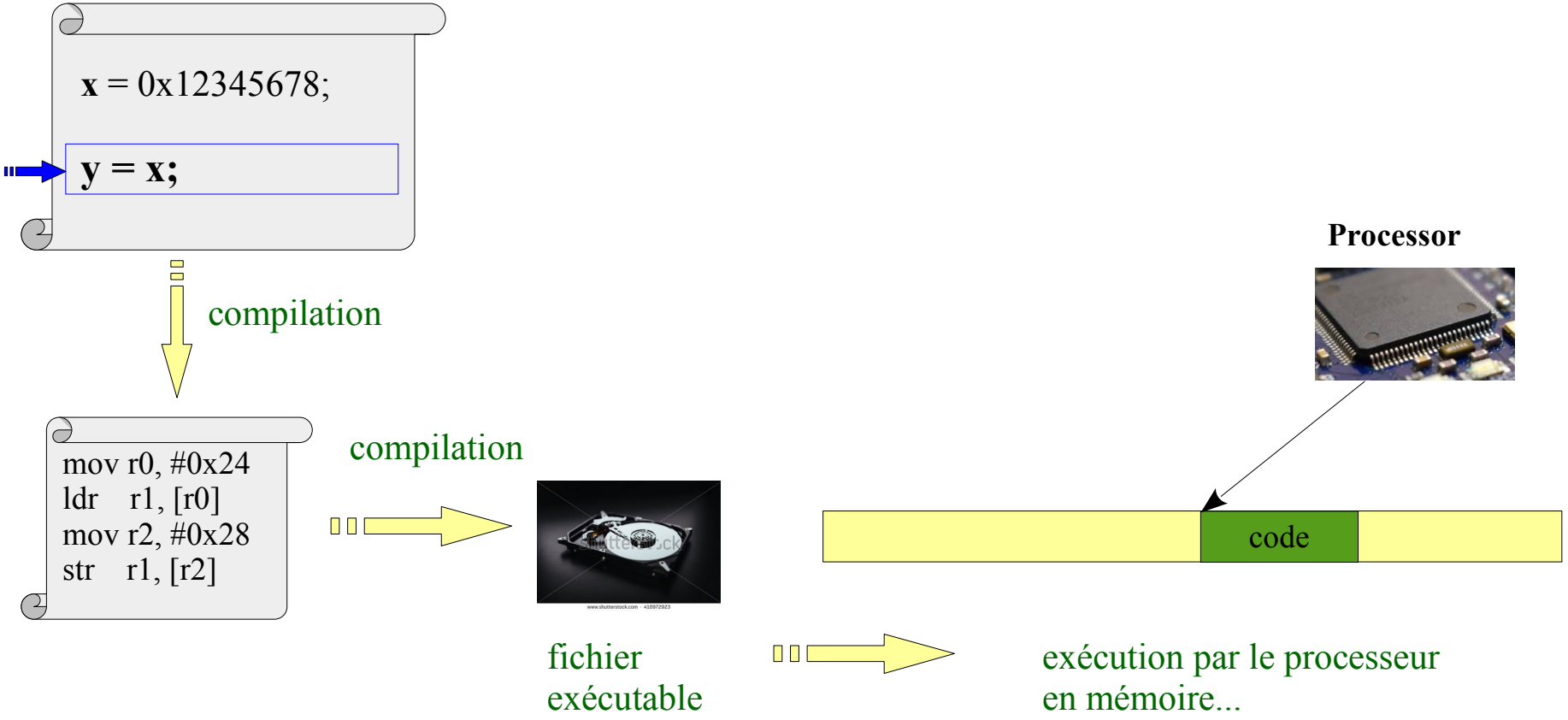
```
0x10000    mov  r0, #6
0x10004    bl   #0x10      ; branch-and-link
...

0x10014    mov  r1, #0
0x10018    mov  r2, #1
0x1001C    add  r1, r1, #1   ; r1 = r1 + 1
0x10020    mul  r2, r2, r1   ; r2 = r2 * r1
0x10024    cmp r1,r0         ; compare r1 & r0
0x10028    ble #0x1001C      ; branch if r1 <= r0
0x1002C    mov pc, lr        ; link register to pc
```

Le même code, écrit dans deux langages

```
int fact(int var0) {
  int var1 = 0;
  int var2 = 1;
  while (var1 <= var0) {
    var1 = var1 + 1;
    var2 = var2 * var1;
  }
  return var2;
}
```

Développer...

- Le cycle de développement
  - Le développeur écrit du texte (Java, Python, C, autres...)
  - Ce texte est compilé en plusieurs étapes avant exécution
  - La mise au point sous *"debugger"* et la validation par des tests



- Le concept de variable
  - Une variable **nomme** une zone de mémoire, souvent **typée**
  - Cette zone mémoire **contient** une valeur primitive (for e.g. int, float, char, memory address)

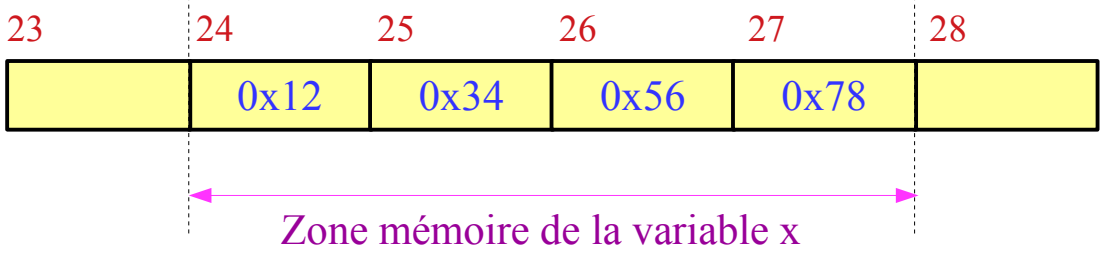
```
int32_t  x = 0x12345678;

int32_t  y = x;
```

compilation

```
mov r0, #0x24
ldr   r1, [r0]
mov r2, #0x28
str   r1, [r2]
```

La variable : x
- ***type : int32_t*** *(32 bits or 4 bytes)*
- ***adresse : 0x24*** *(décidé par le compilateur)*

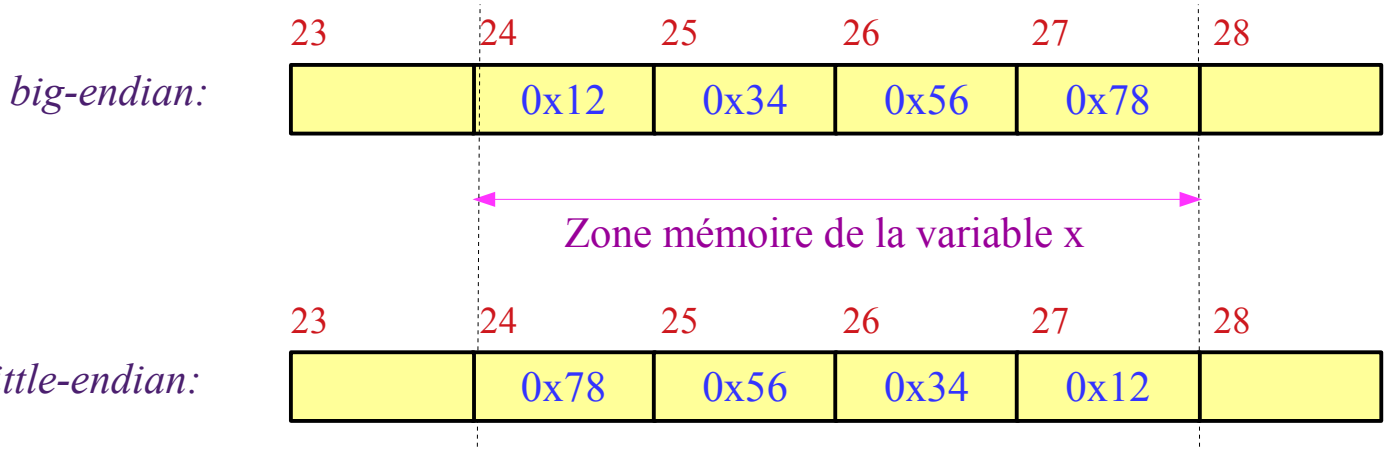
La variable : y
- ***type : int32_t*** *(32 bits or 4 bytes)*
- ***adresse : 0x28*** *(décidé par le compilateur)*



- Petit ou grand boutisme ?
  - Ou comment ordonner les octets d'une valeur en mémoire…
  - C'est le processeur qui décide : *little endian* or *big endian*

```
int32_t  x = 0x12345678;
```

La variable** : **x
- type** : **int32_t *(32 bits or 4 bytes)*
- adresse** : **0x24



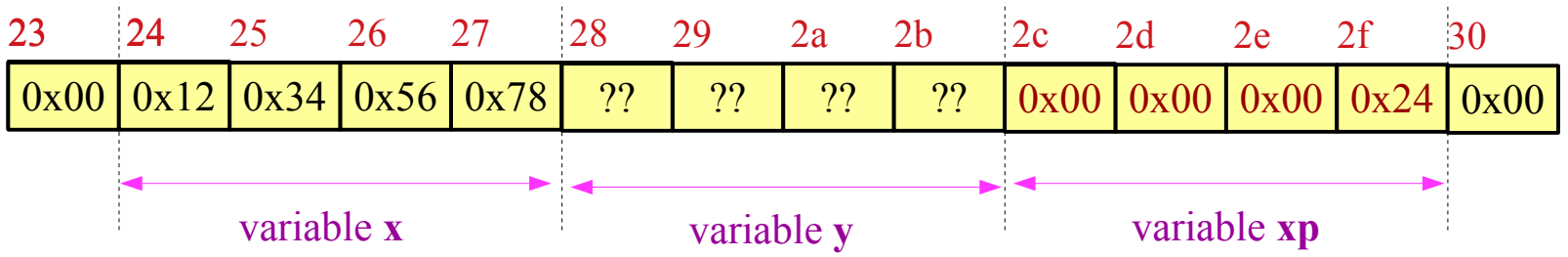
- Le concept de pointeur
  - Un pointeur est une variable
  - Dont la **valeur** est l'**adresse** d'une zone mémoire

```
int32_t x,y;

int32_t*  xp = &x;

x = 0x12345678;
y = *xp;
```

Le pointeur : *xp*
- Son type est « pointeur sur un int32_t »
- Son adresse est 0x2c (décidé par le compilateur)
- Contient l'adresse : ***0x00000024***
- Qui est l'adresse de la variable x (décidé par le compilateur)

Via le pointeur xp, on « voit » la zone mémoire en 0x00000024 comme stockant un entier sur 32 bits

Quelle est la valeur de la variable y après exécution ?

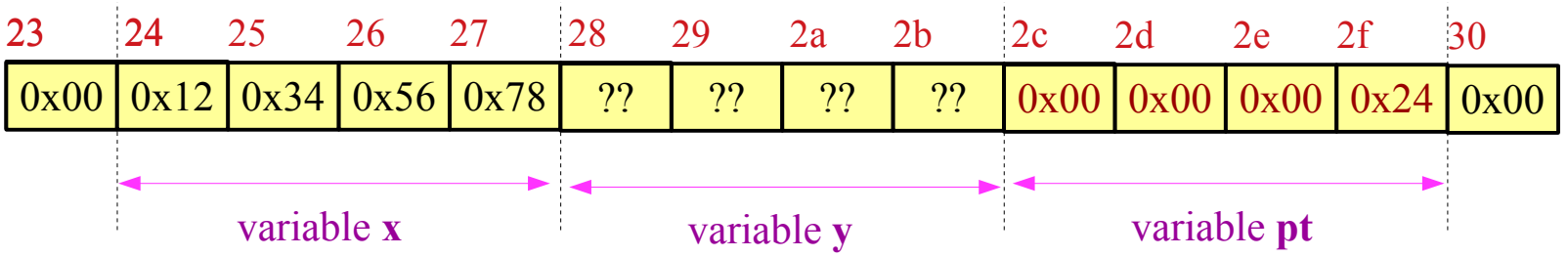


- Arithmétique sur les pointeurs

Quelles sont les valeurs successives de la variable « y » ?

Nota Bene : *ne pas oublier que cela sera dépendant du boutisme de votre processeur…*

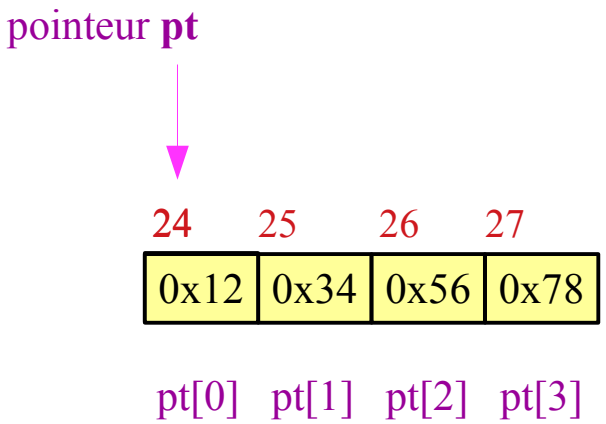
```
int32_t x,y;

int8_t*  pt = &x;

x = 0x12345678;

y = *pt;
y = *(pt+1);
y = *(pt+2);
y = *(pt+3);
```



- Arithmétique sur les pointeurs

La notion de tableau, ou séquence d'éléments...

Avec une notation simplifiée (sucre syntaxique)




```
int32_t x,y;

int8_t*  pt = &x;

x = 0x12345678;

y = pt[0];      // *(pt+0)
y = pt[1];      // *(pt+1)
y = pt[2];      // *(pt+2)
y = pt[3];      // *(pt+3)
```

Notez : après les '//', en vert, ce sont des commentaires, que le compilateur ignore

- Arithmétique sur les pointeurs

Que fait ce programme ?

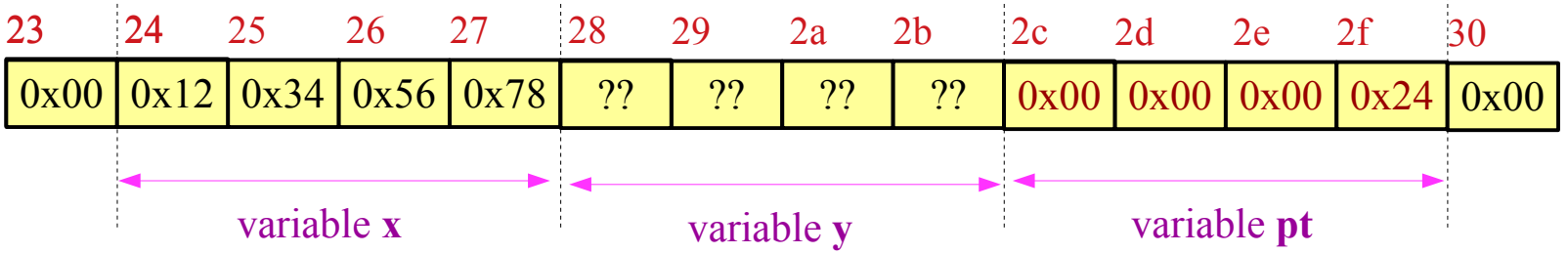
Attention au type du pointeur « pt »...

Hypothèses :

*la zone mémoire de **x** est à **0x24***

*la zone mémoire de **y** est à **0x28***

```
int32_t x,y;

x = 0x12345678;

int32_t*  pt = &x;

*(pt+1) = *pt;
```

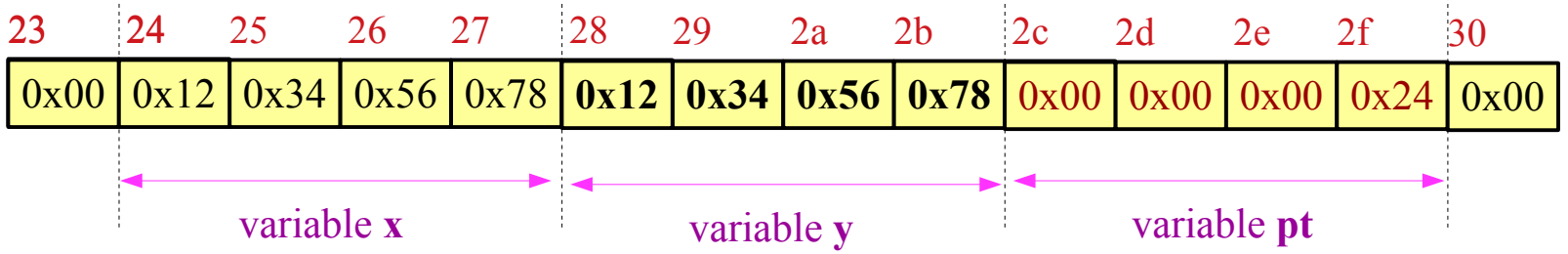


- Arithmétique sur les pointeurs

L'exécution de ce programme est donc équivalent à

y = x ;

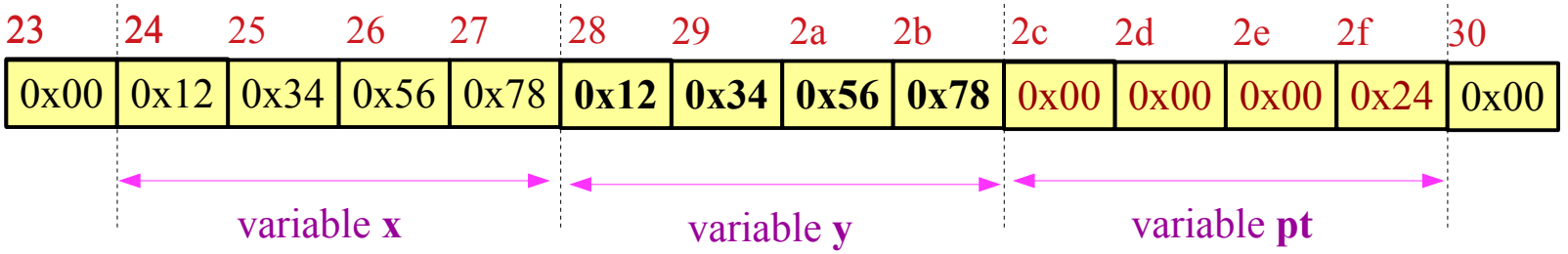
```
int32_t x,y;
x = 0x12345678;

int32_t*  pt = &x;
    → pt = 0x24

*(pt+1) = *pt;
    → *(0x24 + 1 * sizeof(int32_t))
    → *(0x28)

*(0x28) = *(0x24)
```



- Arithmétique sur les pointeurs

Mais, est-ce que ce programme est correct ?

```
int32_t x,y;
x = 0x12345678;

int32_t*  pt = &x;
    → pt = 0x24

*(pt+1) = *pt;
    → *(0x24 + 1 * sizeof(int32_t))
    → *(0x28)

*(0x28) = *(0x24)
```



- Arithmétique sur les pointeurs

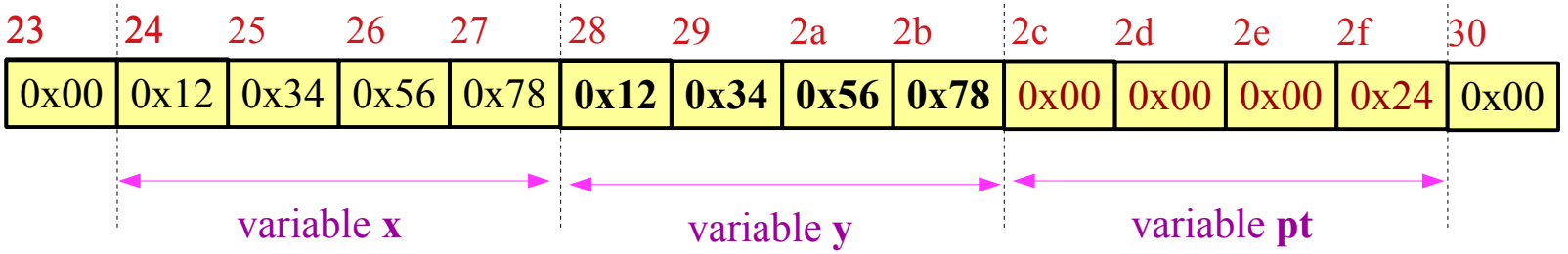
Ce programme n'est pas correct !

Rien ne vous garantie que la zone mémoire de la variable "y" soit juste après la zone mémoire de la variable "x".

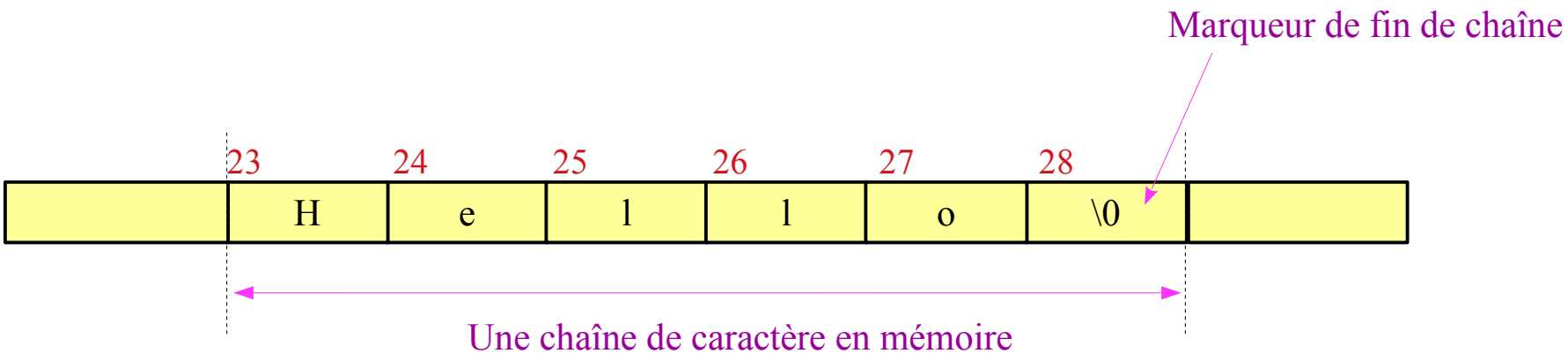
Ce code écrit donc quelque part en mémoire, mais vous ne savez pas ce qui s'y trouve...

Rien ? La variable y ? Du code ? Est-ce même une adresse invalide ? Ou encore des registres mmio ?

```
int32_t x,y;

x = 0x12345678;

int32_t*  pt = &x;

*(pt+1) = *pt;
```



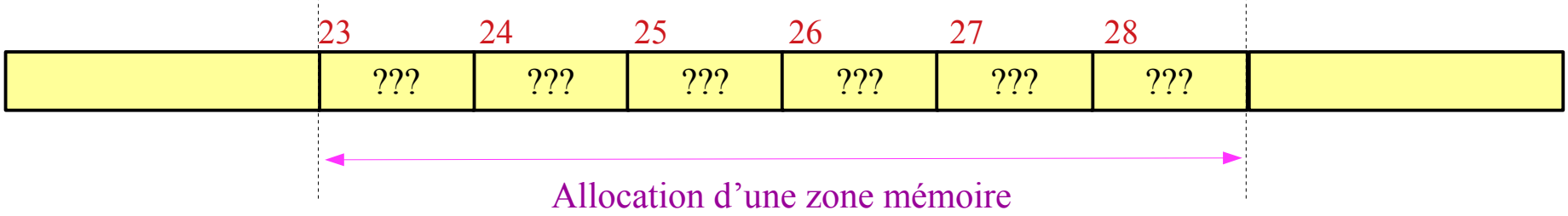
- Les chaînes de caractères
  - Les ordinateurs sont excellent pour manipuler des nombres
  - Mais aussi du texte, on aimerait donc avoir des chaînes de caractères en mémoire



- Les chaînes de caractères

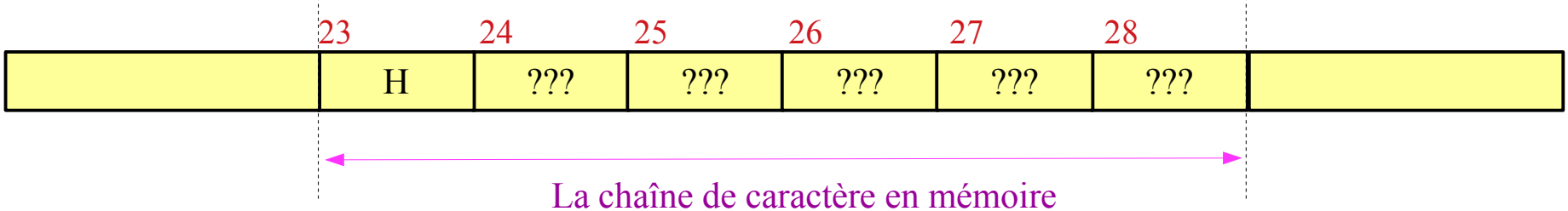
```
char* s = malloc(6);

*s = 'H';
*(s+1) = 'e';
*(s+2) = 'l';
*(s+3) = 'l';
*(s+4) = 'o';
*(s+5) = '\0';

print(s);

free(s);
```

***La variable** : s*
- ***type** : char**
- ***adresse** : **0x23*** ← ***mémoire allouée** par la fonction malloc*



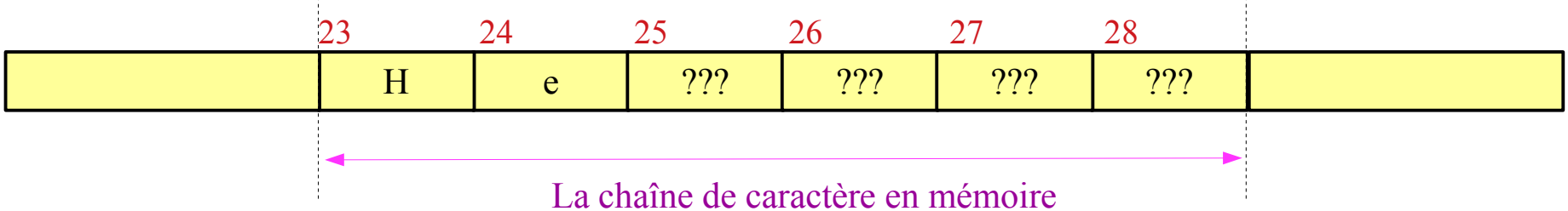
- Les chaînes de caractères

```
char* s = malloc(6);

*s = 'H';
*(s+1) = 'e';
*(s+2) = 'l';
*(s+3) = 'l';
*(s+4) = 'o';
*(s+5) = '\0';

print(s);

free(s);
```



- Les chaînes de caractères

```
char* s = malloc(6);

*s = 'H';
*(s+1) = 'e';
*(s+2) = 'l';
*(s+3) = 'l';
*(s+4) = 'o';
*(s+5) = '\0';

print(s);

free(s);
```

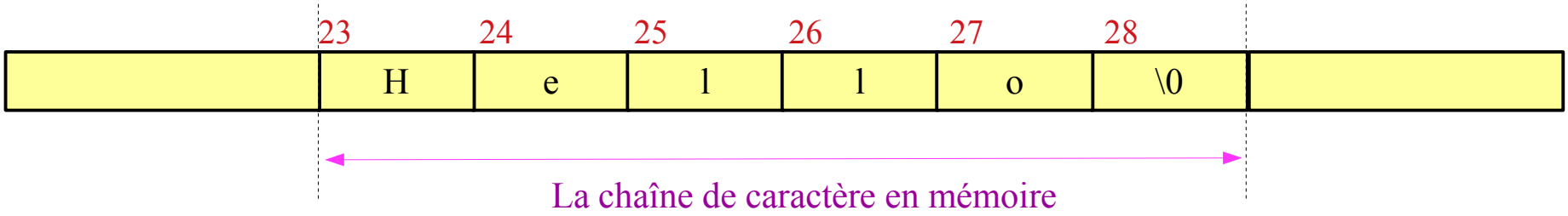


- Les chaînes de caractères

```
char* s = malloc(6);

*s = 'H';
*(s+1) = 'e';
*(s+2) = 'l';
*(s+3) = 'l';
*(s+4) = 'o';
*(s+5) = '\0';

print(s);

free(s);
```

Chaîne de caractère *:*
- séquence de caractère
- terminée par un zéro

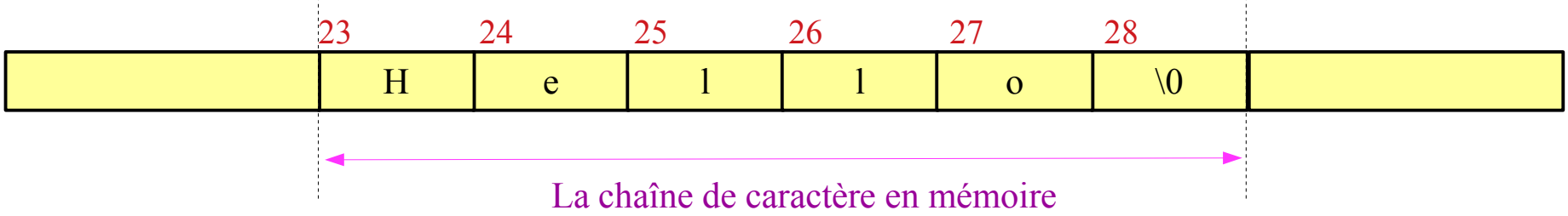
Pourquoi terminée par un '\0' ?



- Les chaînes de caractères

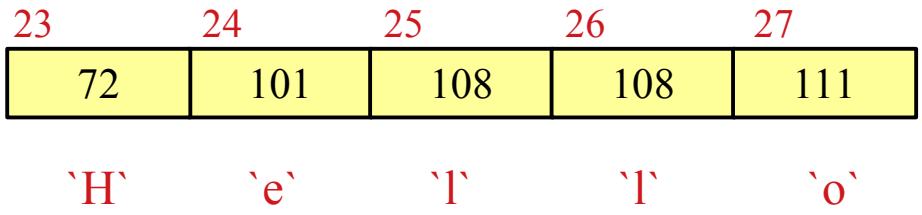
```
char* s = malloc(6);

*s = 'H';
*(s+1) = 'e';
*(s+2) = 'l';
*(s+3) = 'l';
*(s+4) = 'o';
*(s+5) = '\0';

print(s);

free(s);
```

Mais la mémoire contient des nombres !
Pas des caractères…
Et du coup, on affiche des nombres à l'écran?



Langage de programmation

- ASCII encoding



- Le périphérique écran
  - Reçoit des nombres via un mmio register
  - Affiche les caractères correspondants

Code	Char	Code	Char	Code	Char	Code	Char
0	NUL	32	SPACE	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

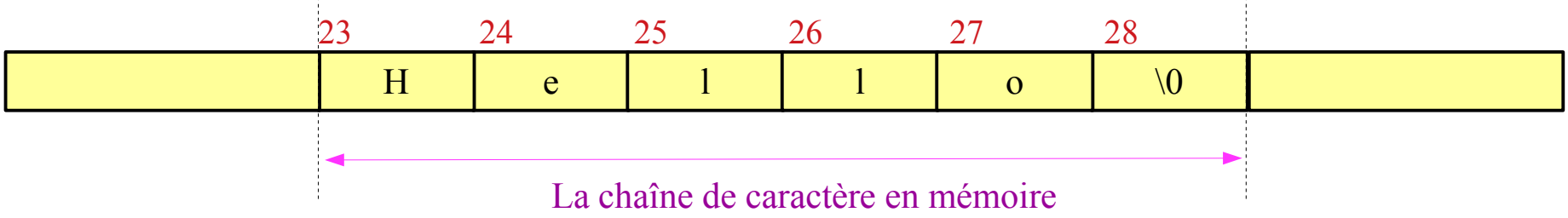
- Les chaînes de caractères

```
char* s = malloc(6);

*s = 'H';
*(s+1) = 'e';
*(s+2) = 'l';
*(s+3) = 'l';
*(s+4) = 'o';
*(s+5) = 0;

print(s);

free(s);
```

***La variable** : s*
*- **type** : char**
*- **adresse** : **0x23** ← **mémoire allouée** par la fonction malloc*

... on utilise la mémoire allouée...

*On n'oublie pas de **libérer la mémoire allouée**, lorsqu'elle n'est plus utilisée...*



Votre premier programme

- L'inévitable programme « Hello World » !
  - Un fichier source
  - Une compilation vers un exécutable
  - Lancer l'exécution dans un shell
- Le debugger
  - Permets de suivre pas-à-pas l'exécution
  - Un outil formidable pour apprendre

```
#include <stdint.h>
#include <stdio.h>

void main(void) {
  char* s = "Hello World!";
  printf("%s\n", s);
}
```

Et l'aventure commence...

Moodle : UGA / Polytech / INFO / INFO3 / Programmation C
Password : Prog-C

https://im2ag-moodle.e.ujf-grenoble.fr/course/view.php?id=373