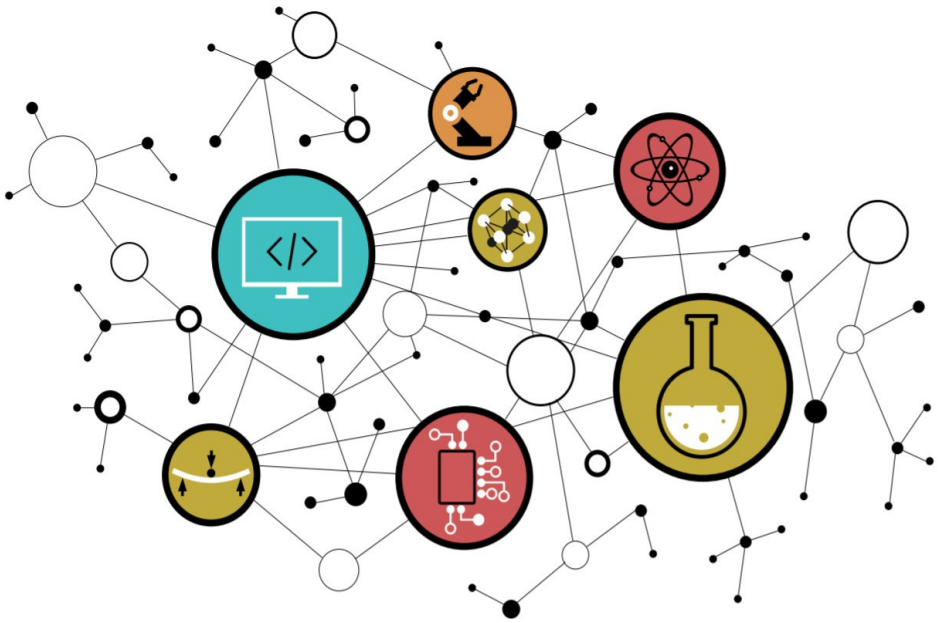
ARCHITECTURE DES ORDINATEURS

COURS



CONTACTS

Équipe enseignante

hugo descoubes - COURS
hugo.descoubes@ensicaen.fr
+33 (0)2 31 45 27 61

Patrick Lacharme
patrick.lacharme@ensicaen.fr

André Lépine
andre.lepine@ensicaen.fr

Sebastien Fourey
sebastien.fourey@ensicaen.fr

ENSICAEN
6 boulevard Maréchal Juin
CS 45053
14050 CAEN cedex 04

RESSOURCES

Les différentes ressources numériques sont accessibles sur la plateforme pédagogique de l'ENSICAEN. Télécharger l'archive complète de travail arch.zip

https://foad.ensicaen.fr/course/view.php?id=99

ÉVALUATION

- QCM Moodle (30m) - mi-parcours

Questions sous forme de QCM (~30 questions) sur la compilation, l'édition des liens sous GCC et l'analyse de programme assembleur x86_64

- Examen sur table (1h30) - terminal

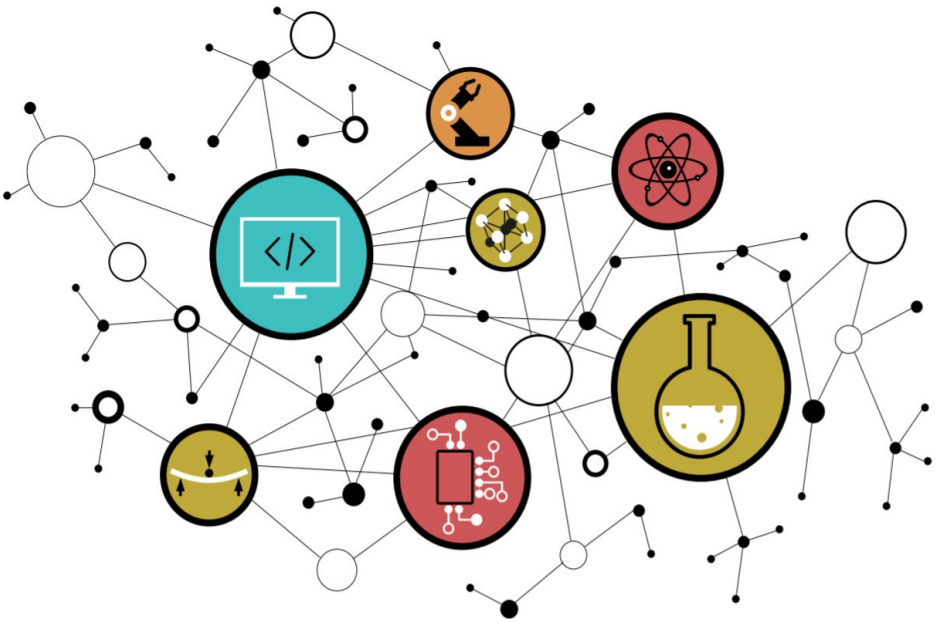
L'évaluation sur table portera sur les séances de Cours Magistral (potentiellement sur tout point présent dans les supports ou présenté à l'oral) ainsi que sur la trame de Travaux Pratiques. S'aider des conseils et exercices présents dans l'archive de travail en ligne (arch/cm/eval) :

  - SAVOIR – 7pts : Questions de culture générale pouvant traiter sur tout point abordé en séance de cours présentiel ou présent dans le support de travail. *Connaissances fondamentales et culture scientifique de l'ingénieur électronicien.*

  - ANALYSER – 13pts : Exercice de traduction d'un programme assembleur vers un programme C et analyse des mécanismes de gestion de la pile conjointement réalisés par le système d'exploitation, la chaîne de compilation et le processeur. *Comprendre et maîtriser le travail d'un processeur numérique, d'une chaîne de compilation et des mécanismes de gestion de la mémoire. Adaptabilité de l'élève ingénieur.*

Il est possible de s'entraîner très simplement concernant l'exercice d'analyse de code. Éditer un programme C simple (1 à 2 appels de fonctions imbriqués, pas plus de 2 paramètres par fonction). Les fonctions réaliseront des traitements élémentaires. Générer le fichier assembleur 64bits x64 correspondant à votre programme et le fournir à un camarade de promotion sans lui donner le programme C équivalent. Que votre camarade réalise le même travail. En partant du fichier assembleur, essayer de proposer un programme C pouvant générer le même assembleur (plusieurs solutions possibles). De même, proposer le contenu exhaustif de la pile lorsque le pointeur de sommet de pile (SP) se trouve au plus haut sur la pile. Confronter par la suite les solutions proposées.

PRÉLUDE
PRE-REQUIS



PRELUDE

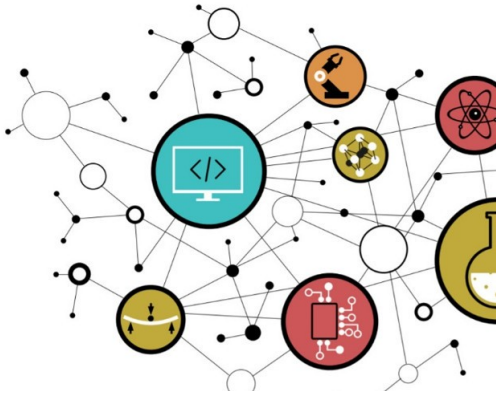
hugo descoubes – enseignant Systèmes Embarqués – ENSICAEN - France
Outils - GNU\Linux Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2022

hugo descoubes - neku

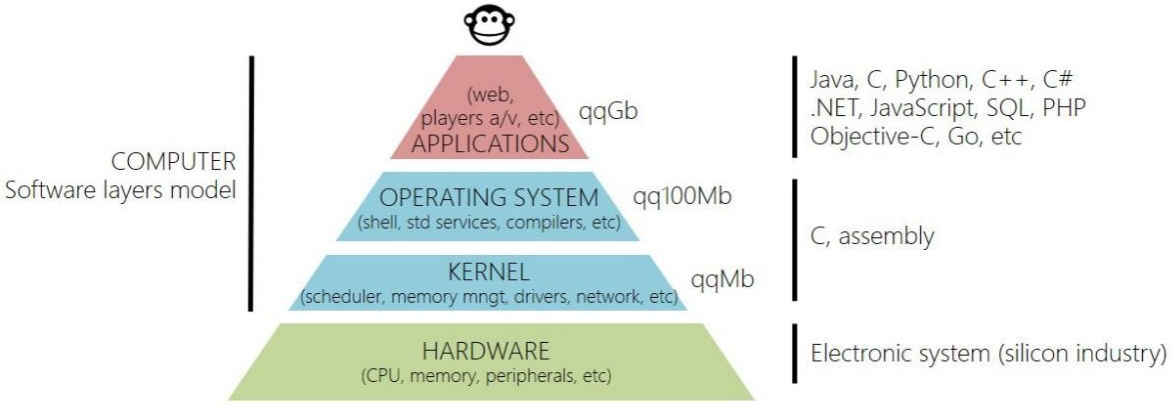
Électronicien spécialisé en Systèmes Embarqués, je suis à votre écoute et ferais de mon mieux pour vous accompagner et vous aiguiller dans votre projet professionnel !

- Bureau en A202 (accès digicode par la salle A203 - *carré + A203 + triangle*) – 02 31 45 27 61
- hugo.descoubes@ensicaen.fr

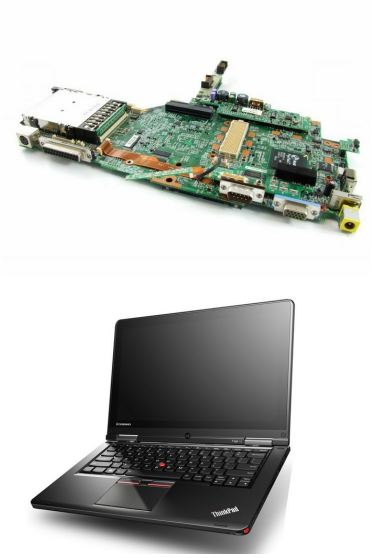
- Objectifs
- Ressources pédagogiques
- Évaluations des compétences
- Pré-requis

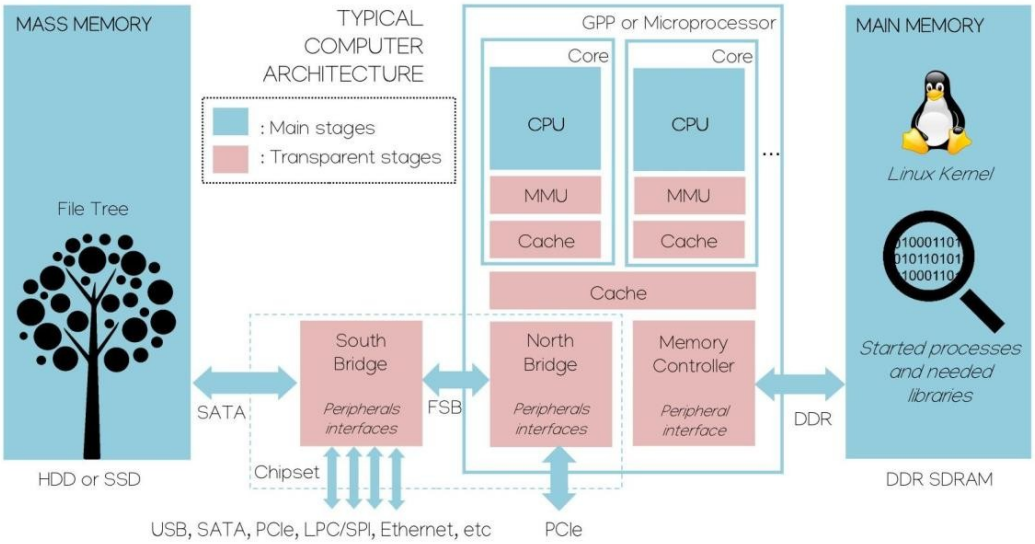


Une bonne maîtrise des langages de programmation système (C et ASM), du processus de compilation et d'édition des liens, du travail de cloisonnement du noyau système au chargement d'un programme en mémoire principale (segmentation et pagination), ainsi que la compréhension des rôles des principaux composants matériels et du fonctionnement global de l'ordinateur sont des bases fondamentales pour un développeur logiciel, même haut niveau (JAVA, C++, etc).

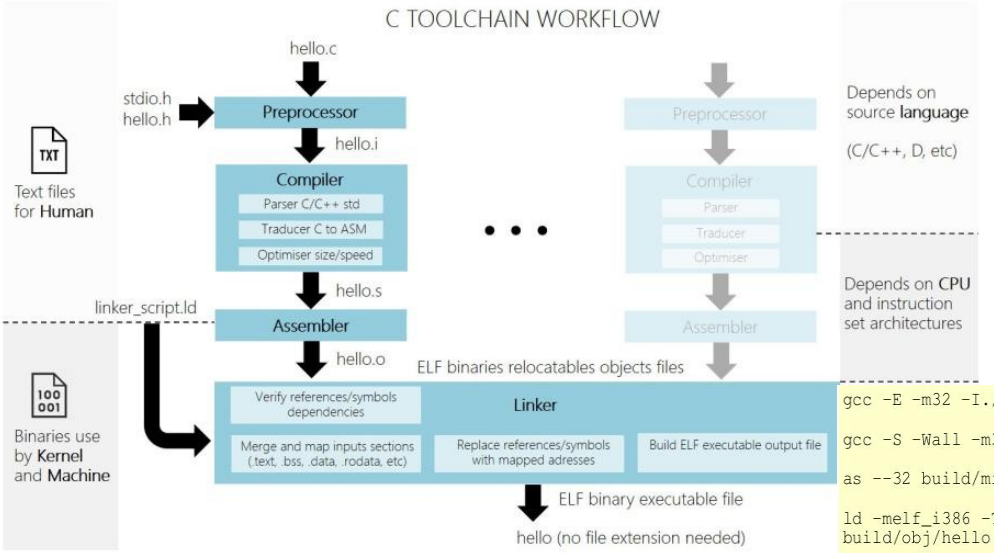


- Connaître les principaux composants matériels d'un ordinateur. Comprendre l'architecture et le fonctionnement global d'un système numérique d'information géré par un système d'exploitation évolué exploitant une MMU et un espace mémoire virtualisé (Linux, NT, etc)





- Comprendre le processus de compilation et d'édition des liens. Connaître les principaux outils d'analyse et de compilation du projet GNU (gcc, as, ld, objdump, readelf, strip, ar, etc)

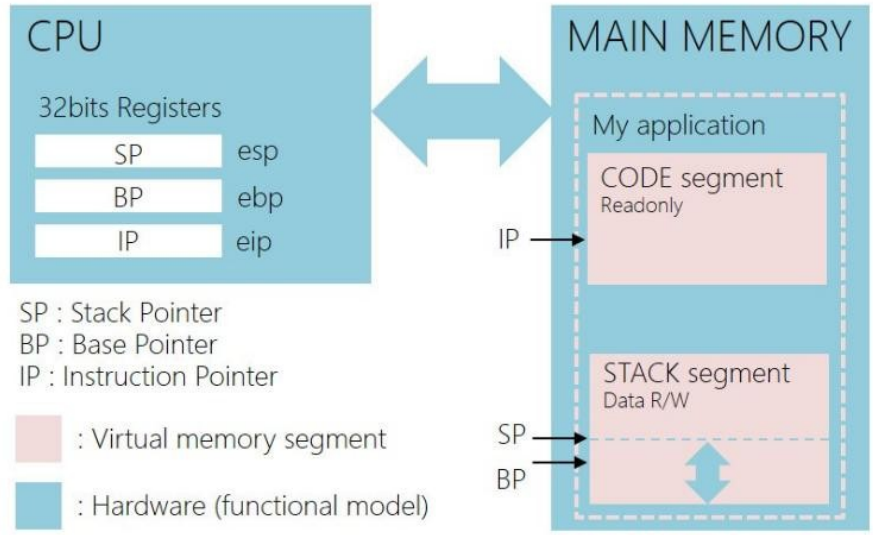



```
gcc -E -m32 -I./inc src/hello.c > build/misc/hello.i

gcc -S -Wall -m32 build/misc/hello.i -o build/misc/hello.s

as --32 build/misc/hello.s -o build/obj/hello.o

ld -melf_i386 -T build/script/linker_script.ld build/obj/crt0.o
build/obj/hello.o -o build/bin/hello
```

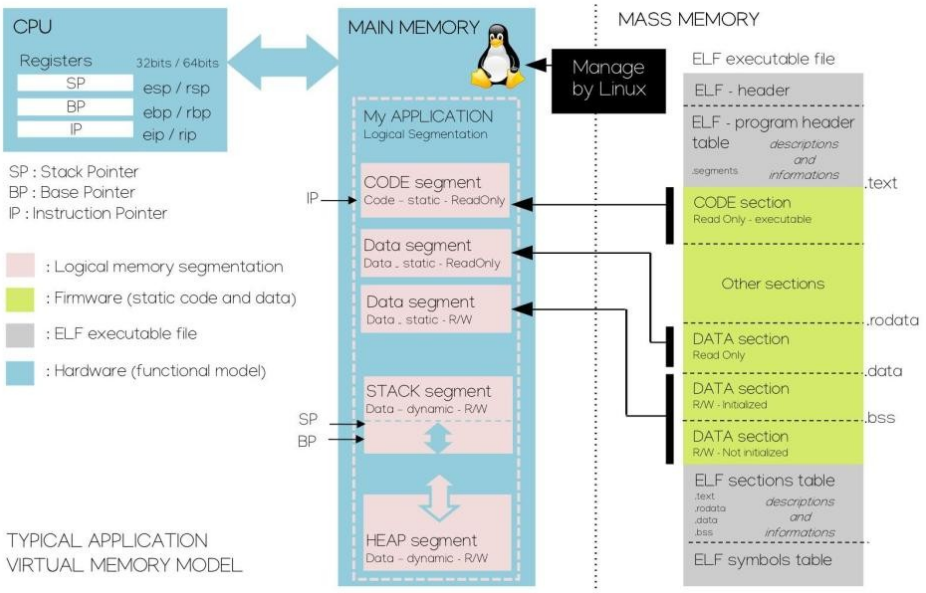
- Analyser un programme ASM x86_64. Comprendre le mécanisme de gestion des variables locales sur la pile réalisé par les outils de compilation, par le système ainsi que la machine

```
int main (void)
{

return 0 ;
}
```

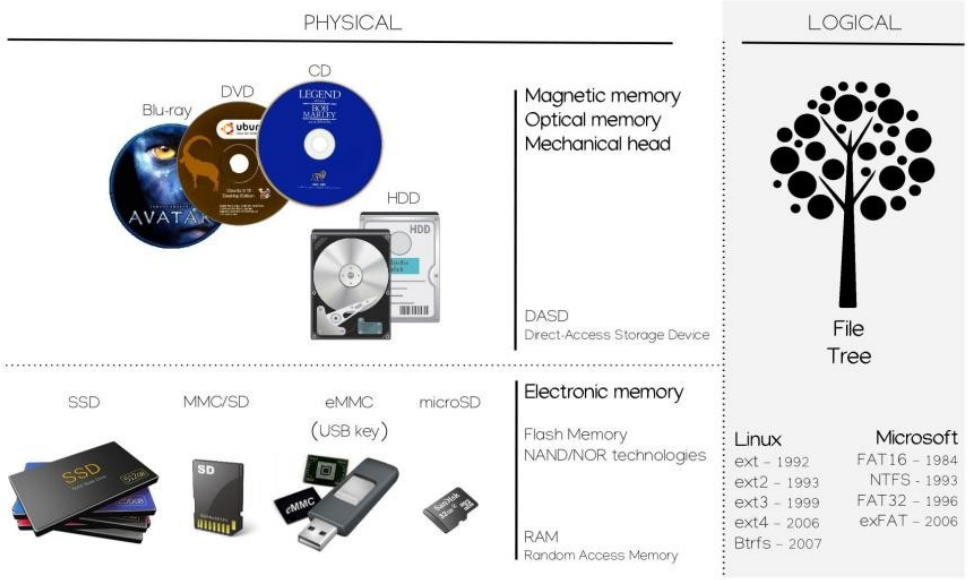
```
main:
     pushl   %ebp
     movl    %esp,%ebp
     movl    $0,%eax
     popl    %ebp
     ret
```

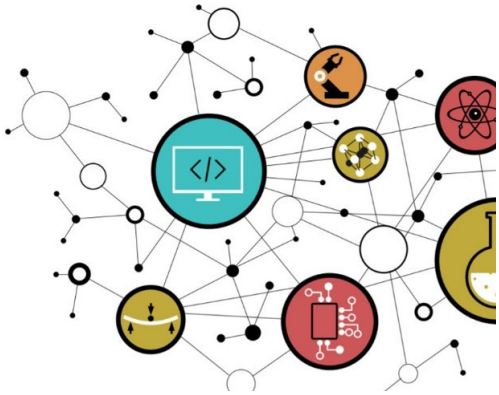


- Comprendre le processus et l'environnement d'exécution d'une application logicielle. Connaître les limitations mémoire imposées par le système et l'ordinateur (segmentation et pagination)



- Comprendre l'architecture logique des supports de stockage de masse (table des partitions, systèmes de fichiers, etc). Être apte à préparer un média de masse pour un besoin spécifique.





- Archive complète de travail Cours/TP sur la plateforme moodle ENSICAEN : **arch.zip**

https://foad.ensicaen.fr/course/view.php?id=99

- Polycopiés séparés de cours et de TP

- Les TP peuvent être réalisés dans les salles informatique du bâtiment E ou en salles A203 et A201 (digicode carré + A203 ou A201 + triangle)

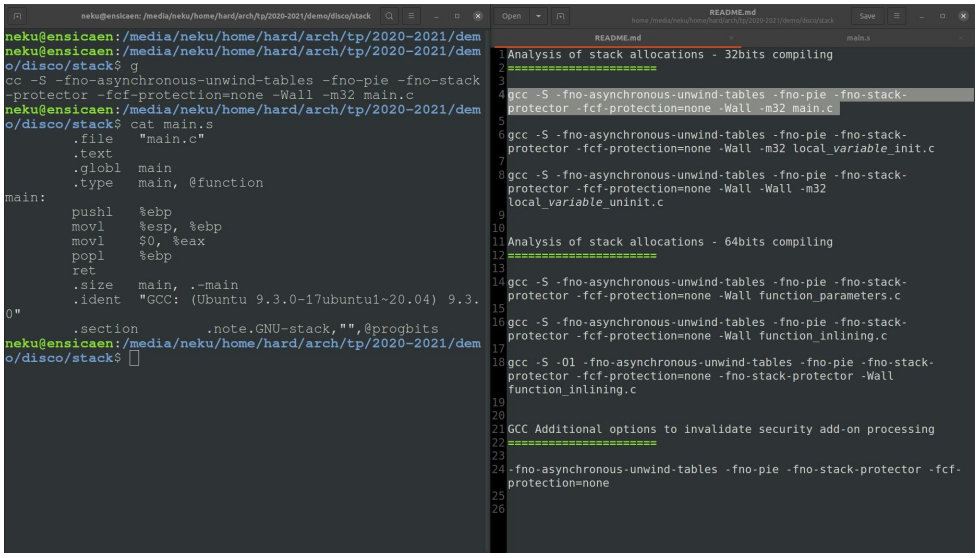
- Nous vous conseillons néanmoins d'utiliser vos machines personnelles avec un système GNU\Linux 64bits. Afin d'utiliser les même configurations système que l'école, nous vous conseillons d'installer un système **Ubuntu 20.04 LTS**. Vous trouverez ci-dessous un tutoriel ENSICAEN pour vous aiguiller dans vos installations

https://ubuntu.com/download/desktop?version=20.04&architecture=amd64

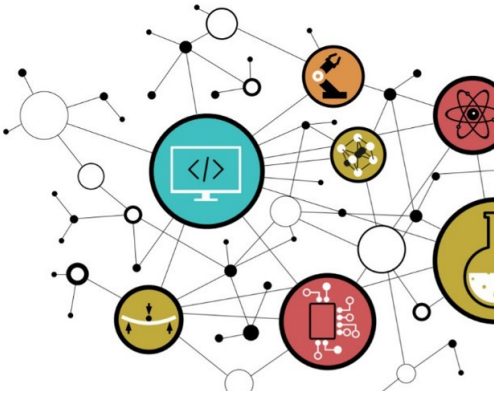
https://foad.ensicaen.fr/mod/page/view.php?id=22125

https://foad.ensicaen.fr/mod/page/view.php?id=25095

- L'environnement de travail se voudra volontairement minimaliste. Un système GNU\Linux 64bits, quelques programmes sources C et ASM x86_64 à analyser, le shell, une toolchain GNU GCC, etc et nous pourrons explorer les entrailles de notre ordinateur …



- Objectifs
- Ressources pédagogiques
- Évaluations des compétences
- Pré-requis



- Partiel 30mn QCM sur moodle :
  - Compilation et édition des liens
  - Assembleur 32bits x86 et 64bits x86_64

- Examen 1h30 à l'écrit sur table :
  - 7pts : 3-4 questions ouvertes d'ordre général. Illustrer à l'aide d'un schéma exhaustif avec définitions et fonctionnement du système à présenter. Peut-être vu comme un échange d'ingénieur à ingénieur.
  - 13pts : Exercice de traduction et de retro-ingénierie ASM x86_64 vers langage C. Analyse d'un programme ASM x86_64 et du fichier objet associé au format ELF. Analyse du comportement et trace du contenu de la pile

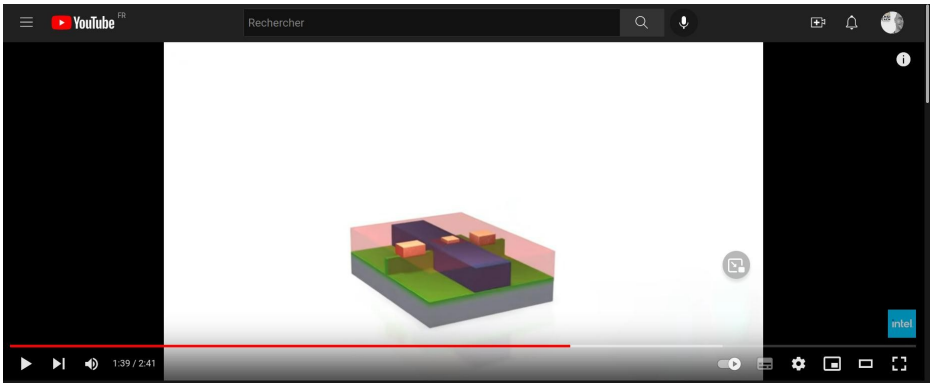
Merci !

Architecture des ordinateurs

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

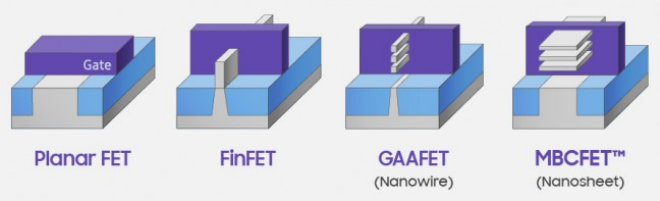
2021-2022

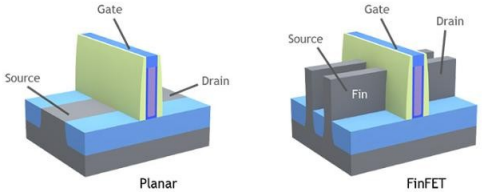
Du transistor MOS au Processeur !

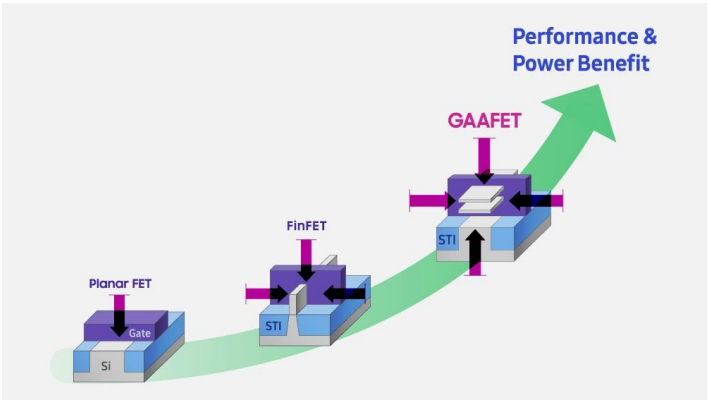


https://www.youtube.com/watch?v=d9SWNLZvA8g

Les technologies de transistors MOS







Les portes logiques (technologie FinFET – exemple Intel Trigate à notre époque)

YES

INPUT A	OUTPUT
0	0
1	1

NOT

INPUT A	OUTPUT
0	1
1	0

AND

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	0
0	1	0
1	1	1

OR

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	1

XOR

INPUT A	INPUT B	OUTPUT
0	0	0
1	0	1
0	1	1
1	1	0

NAND

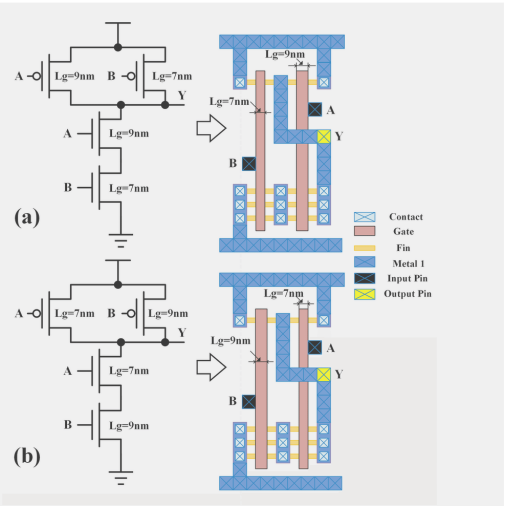
INPUT A	INPUT B	OUTPUT
0	0	1
1	0	1
0	1	1
1	1	0

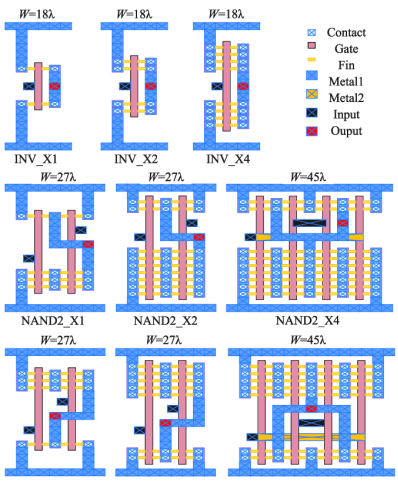
NOR

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	0

XNOR

INPUT A	INPUT B	OUTPUT
0	0	1
1	0	0
0	1	0
1	1	1





La bascule D : Exemple mémoire SRAM à 1bit à 6 FinFET – Static RAM

D Flip-flop

Symbol

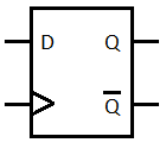
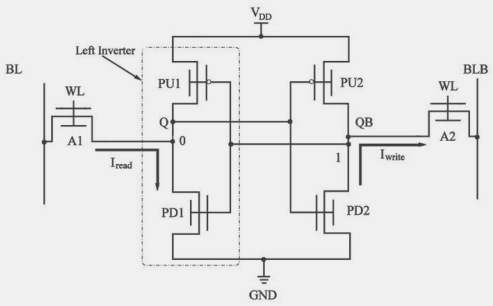
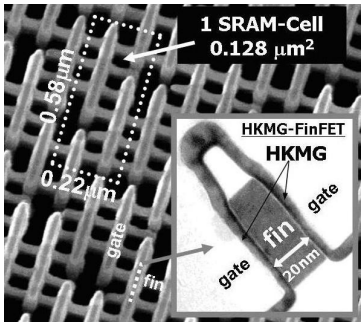


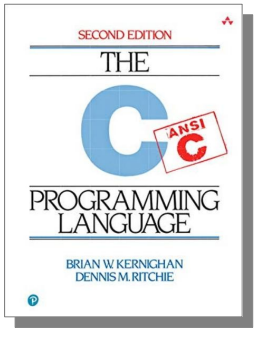
Table of truth:

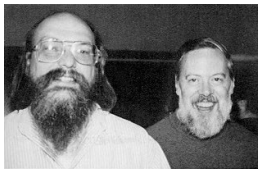
clk	D	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	Q	$\overline{Q}$
1	0	0	1
1	1	1	0





Langage C : Analyse de programmes simples (fonction, variable, pointeur, etc) !

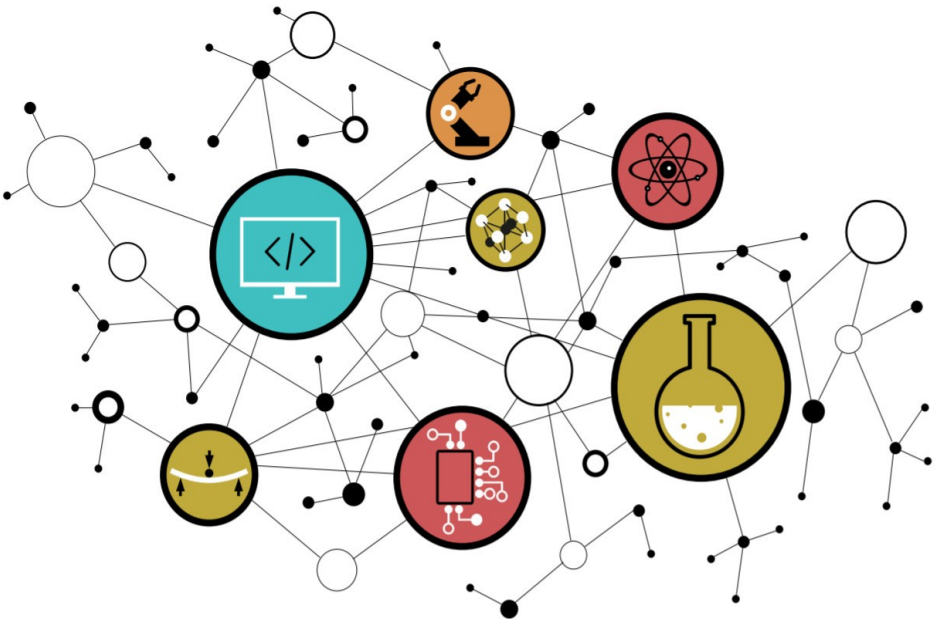




```c
/* ANSI C standard syntax - 1989 */
void function_1 (void) ;
int function_2 (int a, int b, int c);

int main(void)
{

    function_1();

return 0;
}

void function_1 (void)
{
    int ret_1;

    ret_1 = function_2 (1, 2, 3);
}

/* K&R C original syntax - 1978 */
int function_2 (a_2, b_2, c_2)
int a_2;
int b_2;
int c_2;
{
    return a_2 + b_2 + c_2;
}
```

```c
#include <stdio.h>
#include <sys/resource.h>

void goto_segmentation_fault (char* pt_stack_bottom);

int main(void)
{
    char* pt_bottom_of_stack = (char*) &pt_bottom_of_stack;

    /*
    struct rlimit myprogram_ressources;
    myprogram_ressources.rlim_cur = 16000000;
    setrlimit (RLIMIT_STACK, &myprogram_ressources);
    */

    goto_segmentation_fault (pt_bottom_of_stack);

return 0;
}

void goto_segmentation_fault (char* pt_bottom_of_stack)
{
    char current_top_of_stack;

    printf("\rSize of stack = %lu ", pt_bottom_of_stack - &current_top_of_stack);

    fflush(stdout);

    goto_segmentation_fault (pt_bottom_of_stack);
}
```

PROCESSEUR "HOMEMADE"



Chapter 1

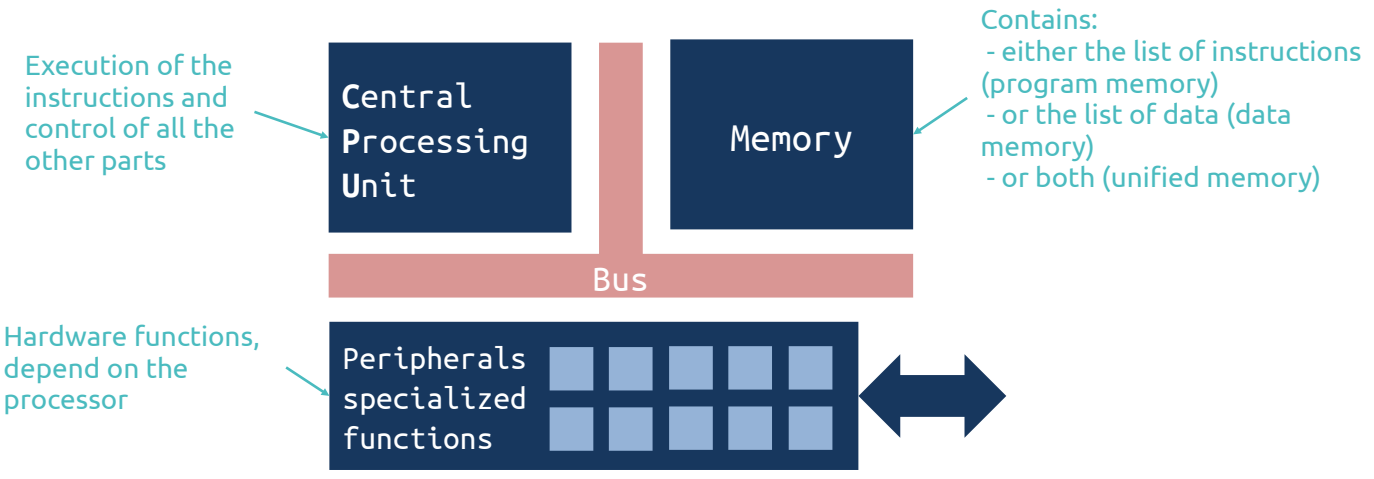
Inside Processors

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

PROCESSOR

Whatever its type (MCU, DSP, GPP… all Turing machines model design by John Von Neumann), a CPU-based processor can be described by the following diagram.



All modern processors use a CPU or a set of CPUs, which functionalities depend on the CPU family.

The memory can be internal (within the processor) or external (as a separate IC). There are also different uses for the memory: work with the CPU (main memory) or store information on a long-time scale (mass storage).

The peripherals also depend on the processor architecture. For now, let's just say peripherals allow interactions between the processor and its environment.

Different CPU/memory/peripherals configurations lead to different architectures. The most common architectures will be described in the "Processor Architectures" chapter.

CENTRAL PROCESSING UNIT

Control Unit

Processing Unit (ALU)

Register file

The **Central Processing Unit (CPU, fr: *Unité Centrale de Traitement*)** is the brain of modern processors, from low-power MCUs to high-performances GPUs.

The CPU's role is to control the information flow within the processor.

As a consequence it controls internal buses, which gives it also has an indirect control of all others hardware functionalities.

The way the CPU reads the program instructions is sequential: this is exactly the way you write your programs (using C, C++, assembly, Python, …).

The CPU will fetch an instruction from the memory, understand it and then execute it. And it will start over and over again, one instruction after the other.

The CPU's **Control Unit** is in charge of running the instruction flow, following this cycle:

Fetch	Read the next instruction from program memory
Decode	Understand the operation to perform by reading the opcode (operation code)
Execute	Perform the operation, or usually ask the Processing Unit to do it
Store	Save the result into internal registers or back to the data memory

The **Processing Unit** of the CPU is responsible for processing most of the instructions.

Depending on the CPU family, it may include:

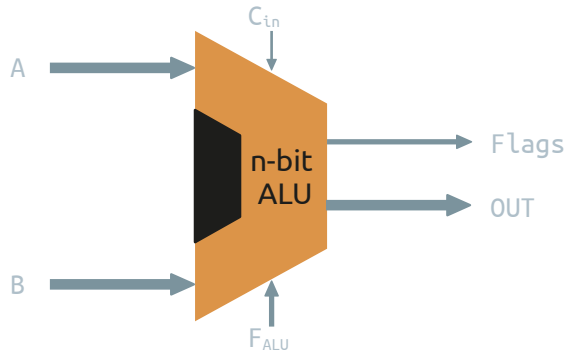
- An Arithmetic and Logic Unit (ALU)
  - Logic operations and simple maths
- A Floating-Point Unit (FPU)
  - For advanced processors
- A multiplier
- A shift register
- ...

The **Arithmetic and Logic Unit (ALU)** is the heart of the Processing Unit.

On this example diagram, data to be processed are on inputs A and B.

The choice of the operation is given by the Control Unit using F_{ALU} bits (thanks to the DECODE stage).

The result is produced on Out output while signal flags (S, Z, C, O, …) are updates according to the result. The Control Unit will read them so it can adapt the instructions to be executed (e.g. if, while, for instructions).




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

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

The **Register file (fr: *banque de registres*)** contains, as you may guess, the CPU registers.

Registers are small memory cells placed in the heart of the CPU: they are very fast, but can only contain few data.

Some are general purpose registers (or working registers), which can store any value (input or output of ALU, temporary variable, …).

Others are specific registers, which can only be used for a given objective.

For instance the Status Register contains some flags, the Program Counter register contains the address of the next instruction to be executed, and you'll discover more in the future.

MEMORY

Volatile memory

Remanent mass storage

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

MEMORY

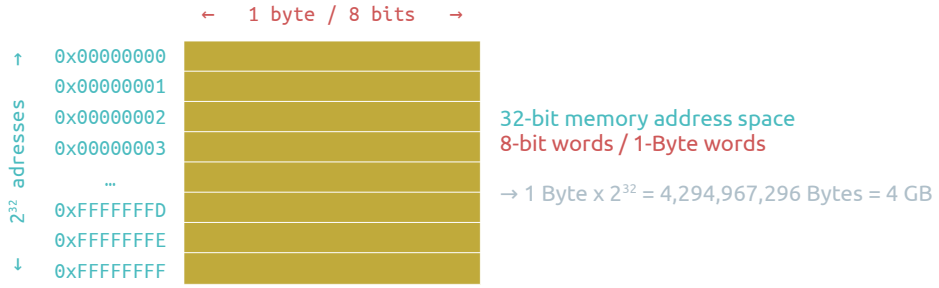
Byte-addressable memory

Memory is an electronic device that allows to store information (data and instructions).

Most common usages are volatile memory (that works with the processor) and remanent mass storage (that stores information when not used).

Memories used during the program execution are addressable by byte (unit of storage).

However this is not true for cache memories (built within the CPU) and mass storage that uses file systems (ext4, FAT32, NTFS, …)



Let's make it clear:

When switched off, a volatile memory will lose its data but a remanent memory will preserve it.

ROM (Read-Only Memory) is an obsolete technology, with which the memory could be written only once. It has been replaced by PROM (Programmable ROM), especially UVPROM (Ultra-Violet PROM, now obsolete) and EEPROM (Electrically Erasable PROM). Please be aware that some still use the word "ROM" to refer to EEPROM.

RAM (Random Access Memory) is a volatile memory technology. "Random Access" means you can access and random address with a constant latency.

The mass storage memory is a remanent memory that keeps your data even when the power is off.

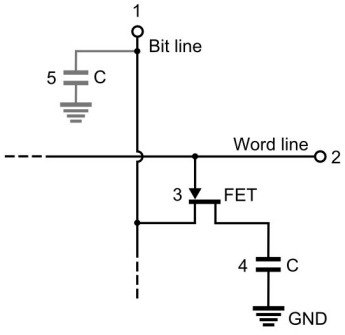
The main memory is a volatile but very fast memory. The processor uses it to store data that is actively used.

En français, "mémoire morte" est aussi obsolète que "ROM", "mémoire vive" est encore utilisé pour parler de mémoire volatile, souvent sous-entendant la RAM.

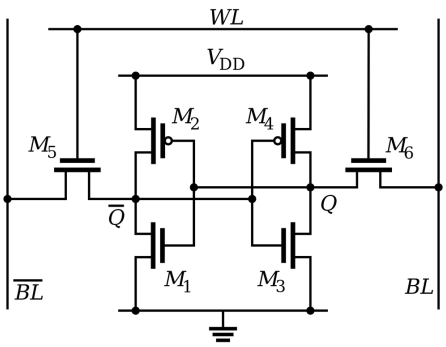
Volatile memory comes in two types: DRAM (Dynamic RAM) and SRAM (Static RAM).

DRAM needs to be periodically updated because of the pico-capacitors. Used for computer memory. Small silicon footprint but slower than SRAM. Current technologies are DDR4 SDRAM (4th generation of Double Data Rate Synchronous DRAM)

SRAM is based on latching circuitry. Used for registers and L1/L2/L3 cache memories. Way faster but bigger silicon footprint.



DRAM
1 bit requires 1 transistor and 1 pico-capacitor



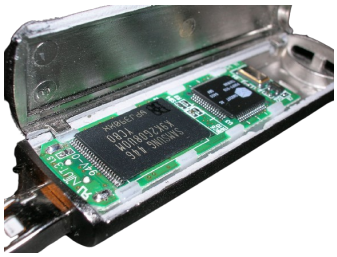
SRAM
1 bit requires 6 CMOS transistors

Remanent mass storage comes different technologies:

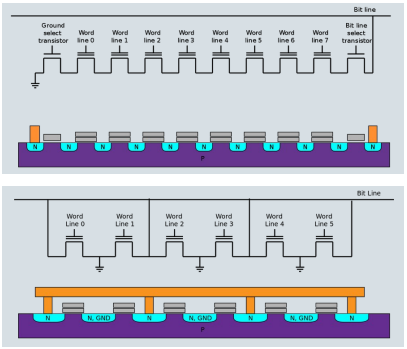
Magnetic storage is used by floppy disks (fr: disquettes) and HDDs (Hard Drive Disks, fr: disque dur).

Electrical charge storage with logic circuitry is used by EEPROMs (Electrically Erasable Programmable ROMs). The most common EEPROM technology is Flash memory (NAND and NOR), which has a constant access time to the information. SSDs (Solid-State Drives) also use Flash technology.

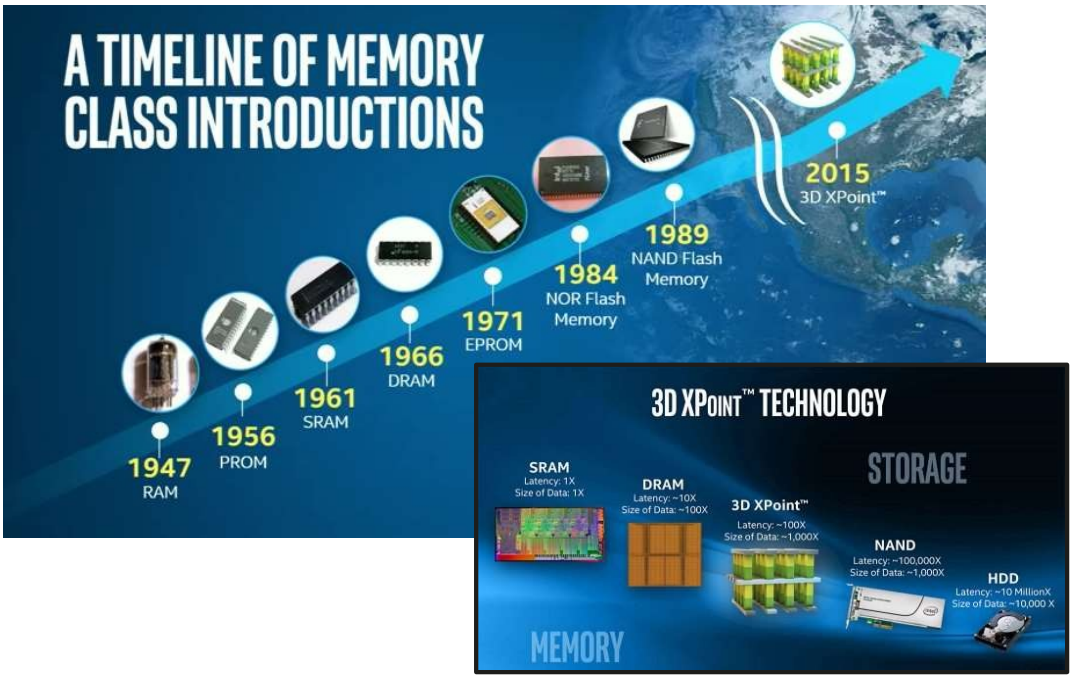
Hard Drive Disk (HDD)



Flash drive
(Flash memory on the left)



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



EXECUTING A PROGRAM

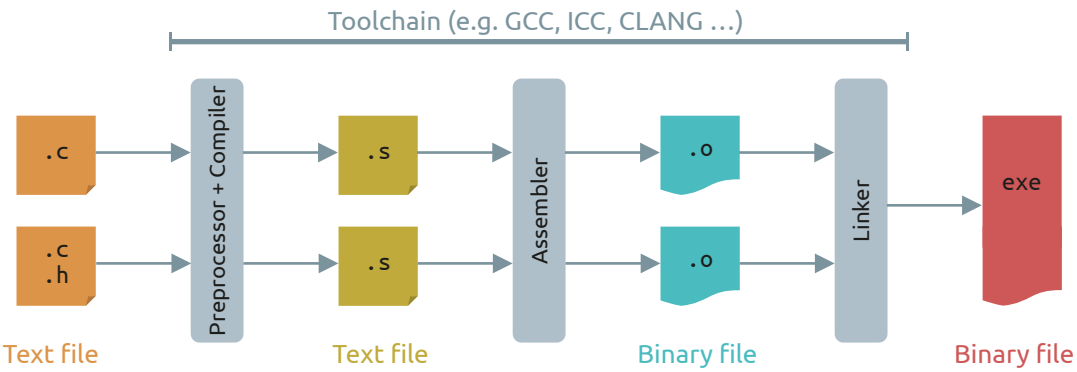
From the C file to an executable binary program

Execution on a home-made processor

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

We'll keep it simple here, as you will see this in details next year.

The **toolchain (fr: *chaîne de compilation*)** is the software tool that "converts" your C source files into an executable binary file.

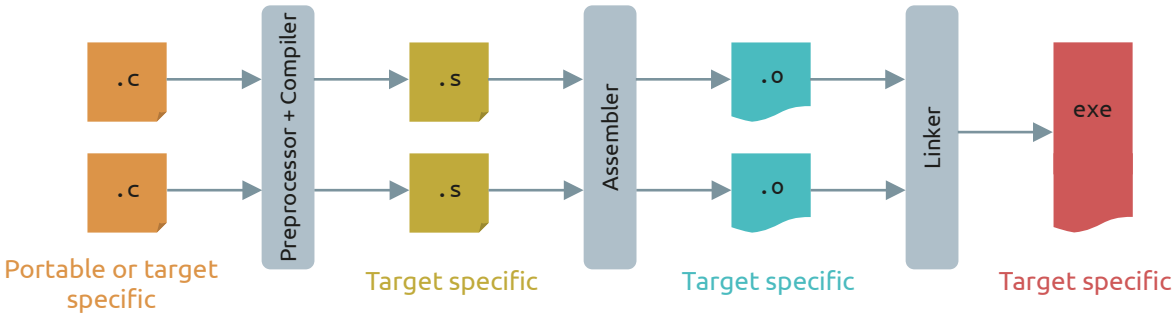


Why ever use a toolchain?

The C language is portable, which means it can be used on different computer systems.

But the processor you choose only understands its own set of instructions. That is the opposite of portability: the code that the processor understands can only run by itself.

The toolchain is a way of writing a universal program (using a portable language) only once, and then create an executable binary for the target processor.



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

C language program

```
char inc(char bar);

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

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

Assembly language program

```
Instructions    Operands

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

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

Binary program

```
Program memory address    Binary instructions

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

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

Behold our home-made processor!

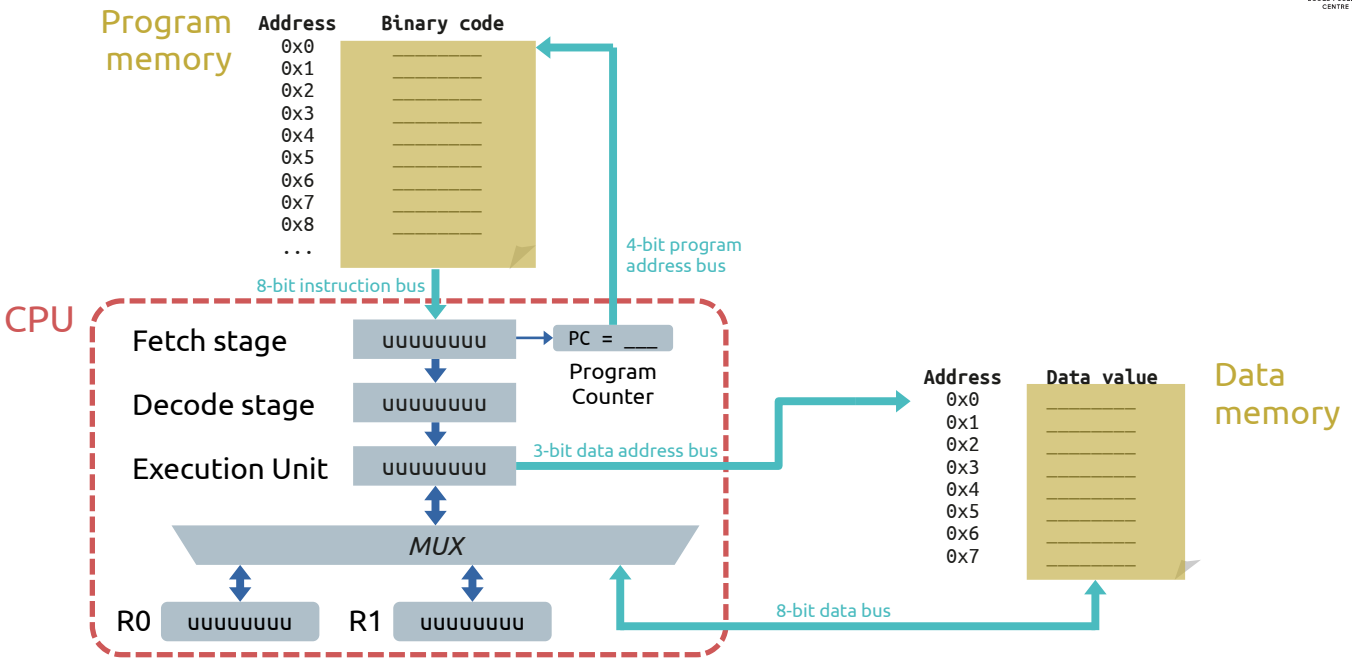
This is a RISC-like (Reduced Instruction Set Computer) elementary CPU.

Its simple ISA (Instruction Set Architecture) is not related to any commercial CPU.

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



Now translate this C program into assembly language for our custom CPU!

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

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

} Data memory map

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

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

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

Solution

C language program

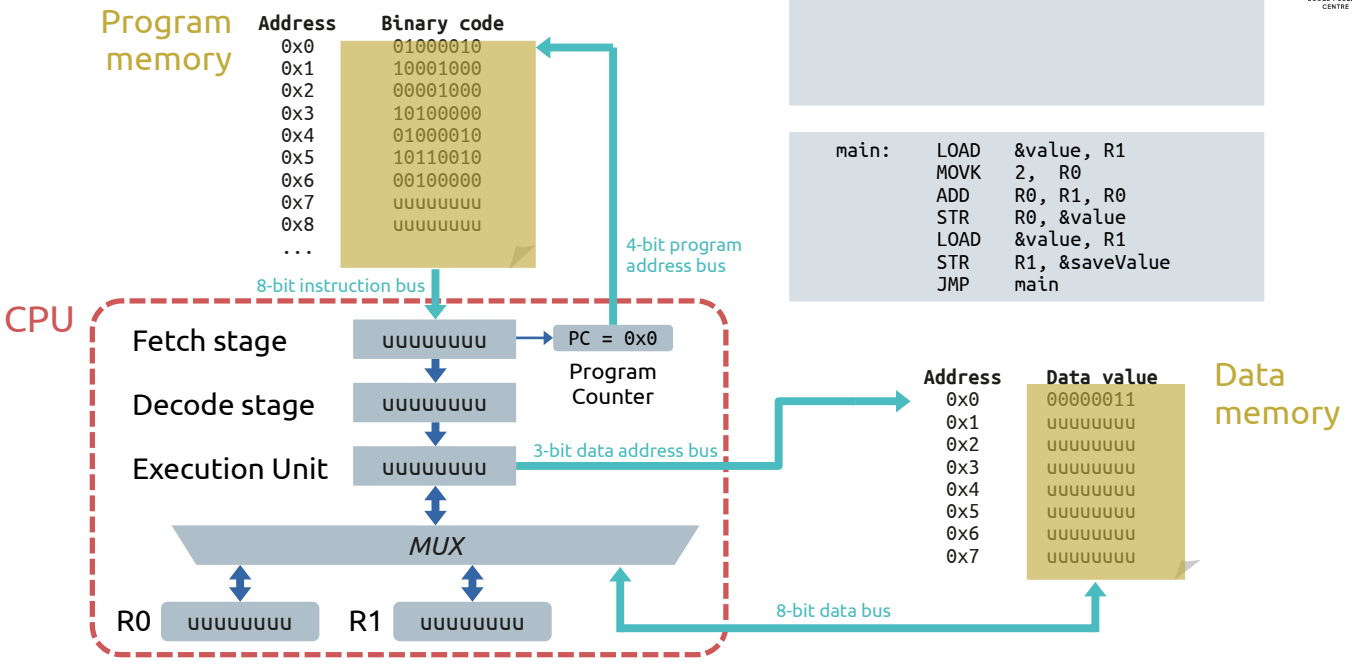
```c
char value = 3; // Stored at 0x0
char saveValue; // Stored at 0x1

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

Assembly language program

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

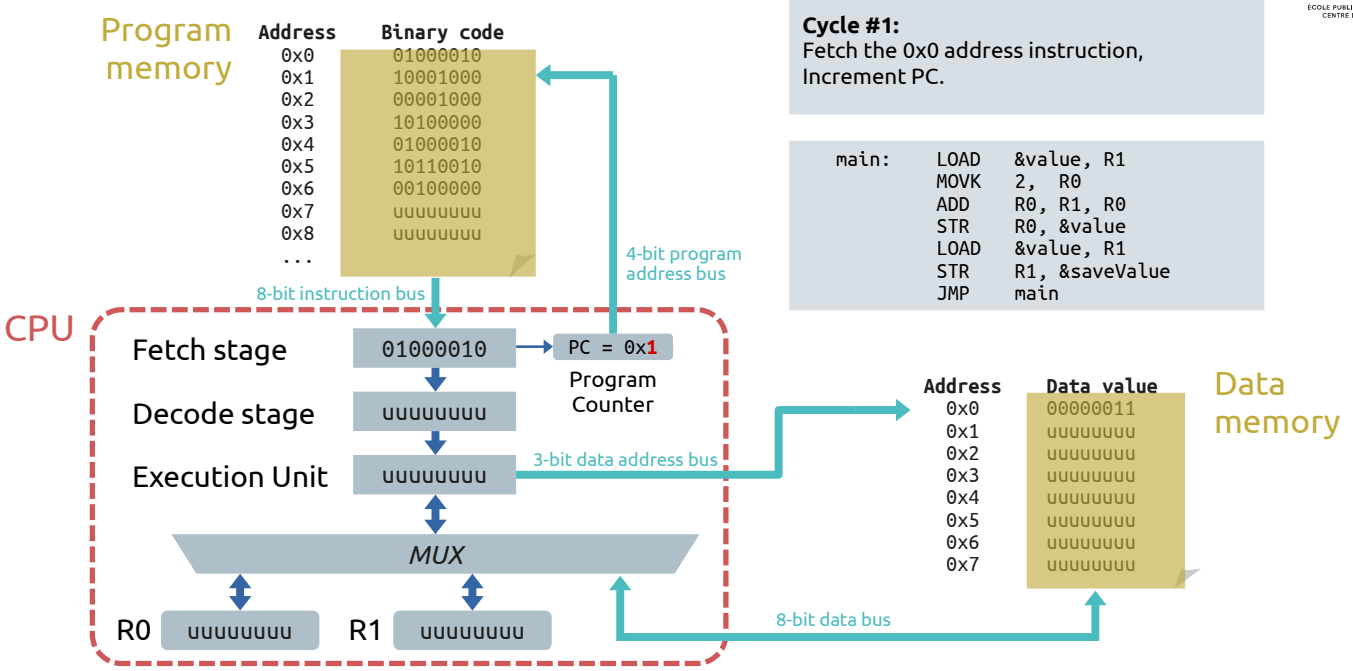
Follow the CPU work step by step

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



EXECUTING A PROGAM

Home-made processor



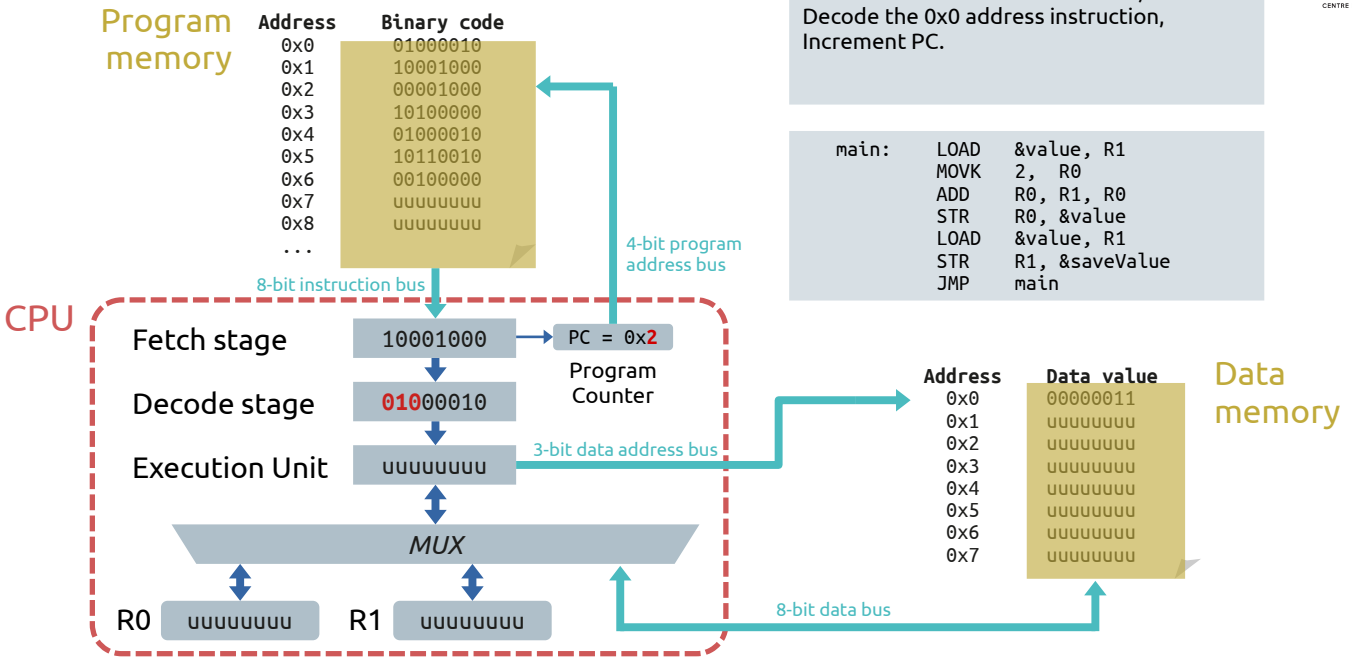
Application starts

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
```


EXECUTING A PROGAM

Home-made processor



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

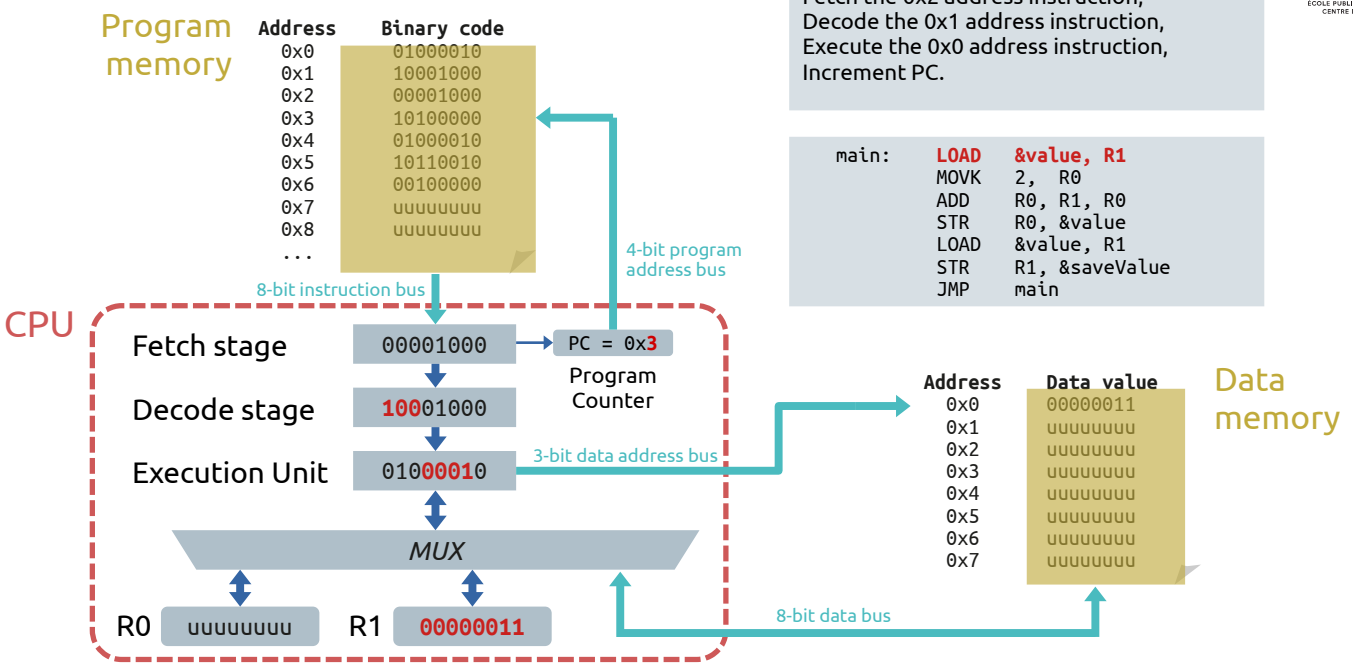
```
main:   LOAD   &value, R1
        MOVK   2,  R0
        ADD    R0, R1, R0
        STR    R0, &value
        LOAD   &value, R1
        STR    R1, &saveValue
        JMP    main
```

EXECUTING A PROGAM

Home-made processor

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
```

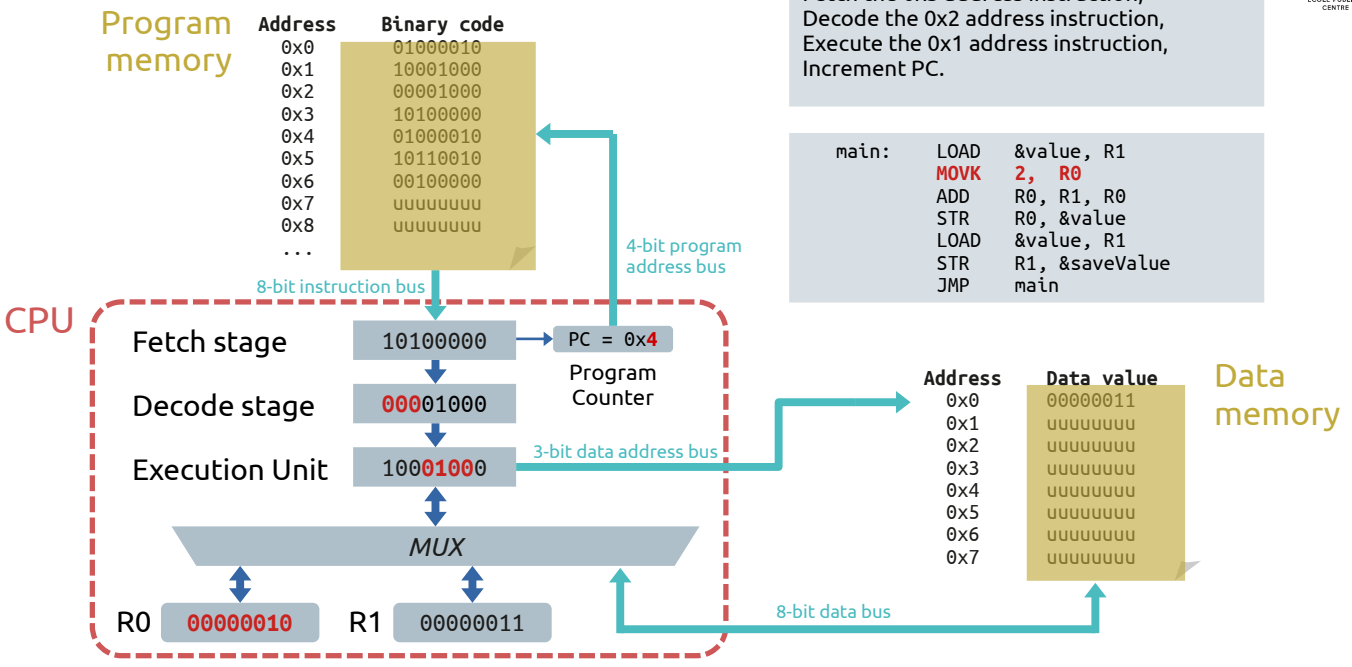


EXECUTING A PROGAM

Home-made processor

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
```

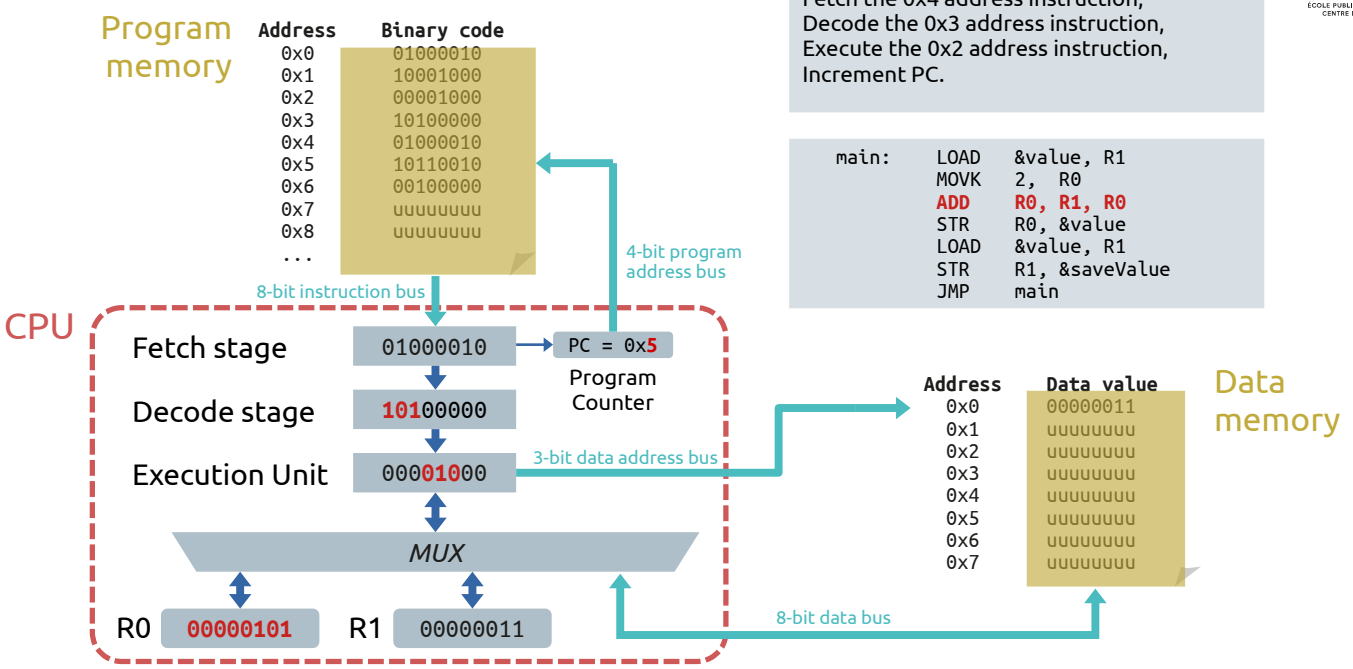


EXECUTING A PROGAM

Home-made processor

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
```

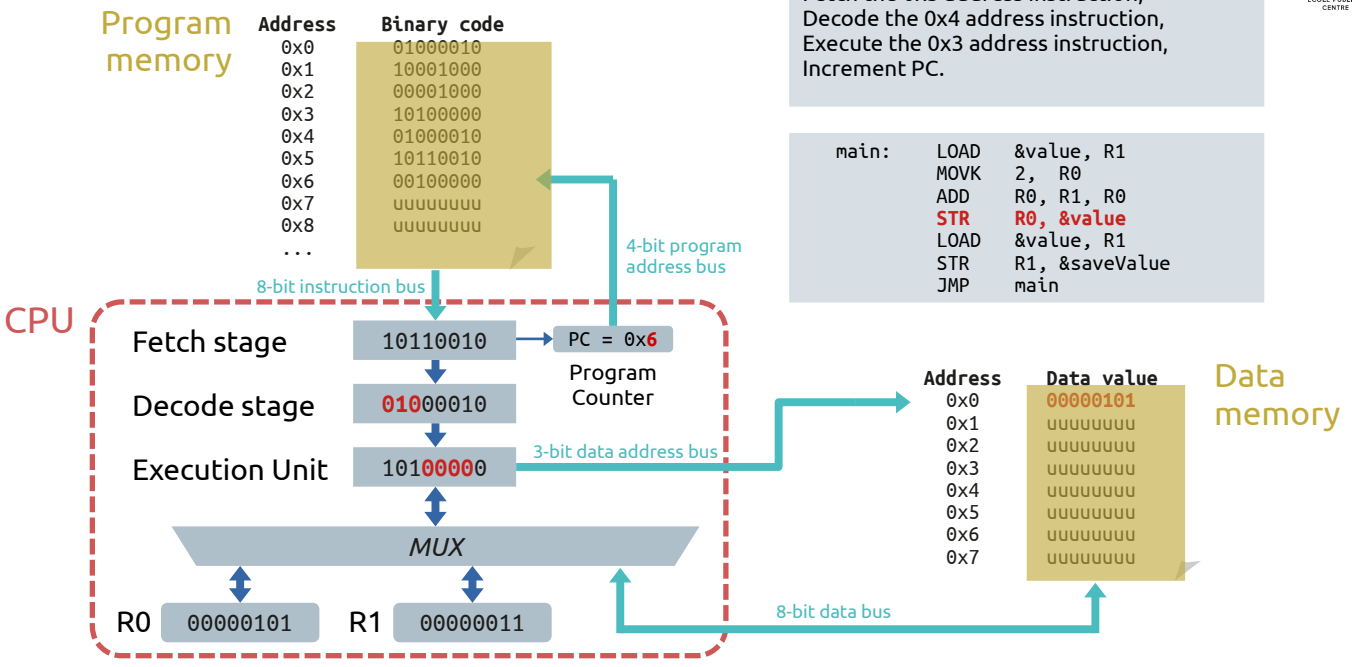


EXECUTING A PROGAM

Home-made processor

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
```

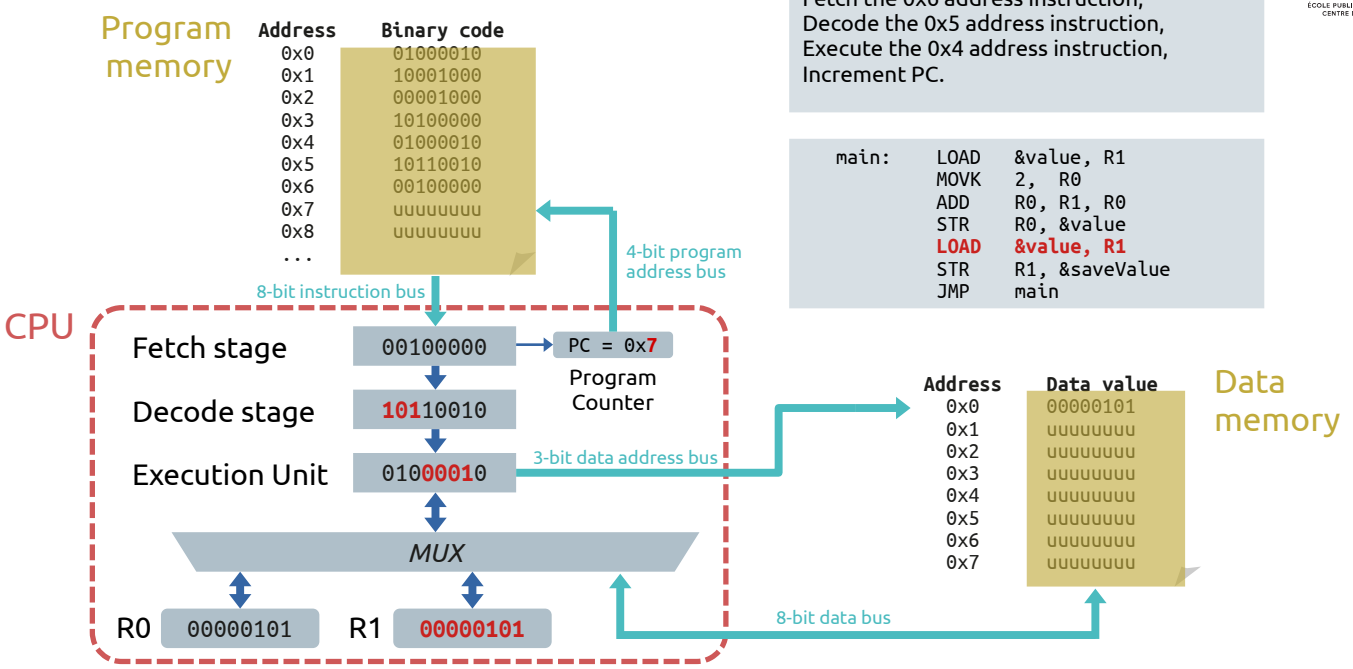


EXECUTING A PROGAM

Home-made processor

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
```

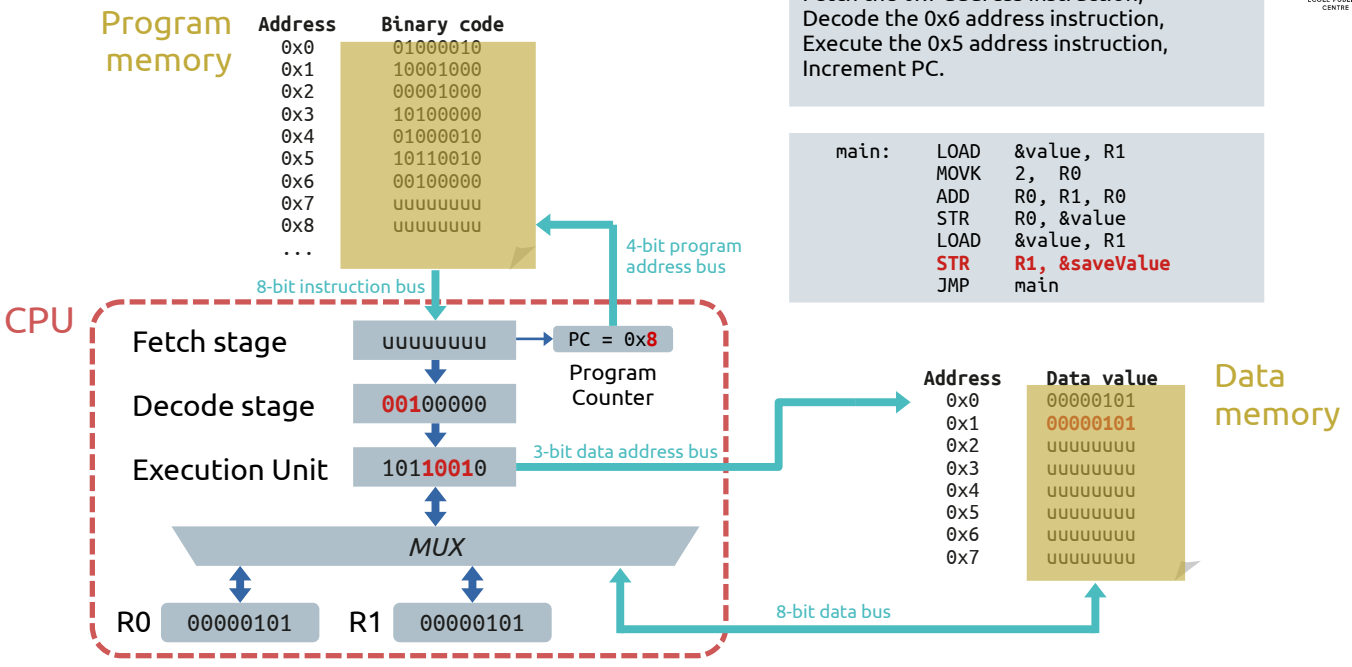


EXECUTING A PROGAM

Home-made processor

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
```

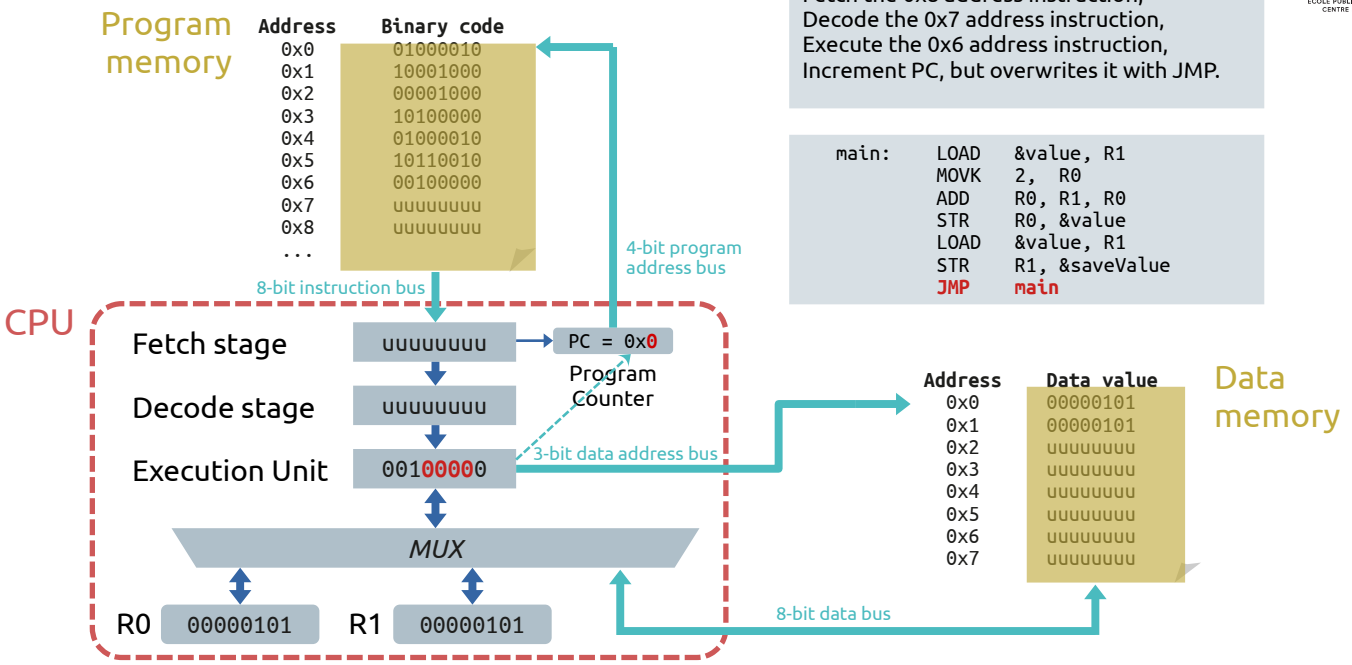


EXECUTING A PROGAM

Home-made processor

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
```

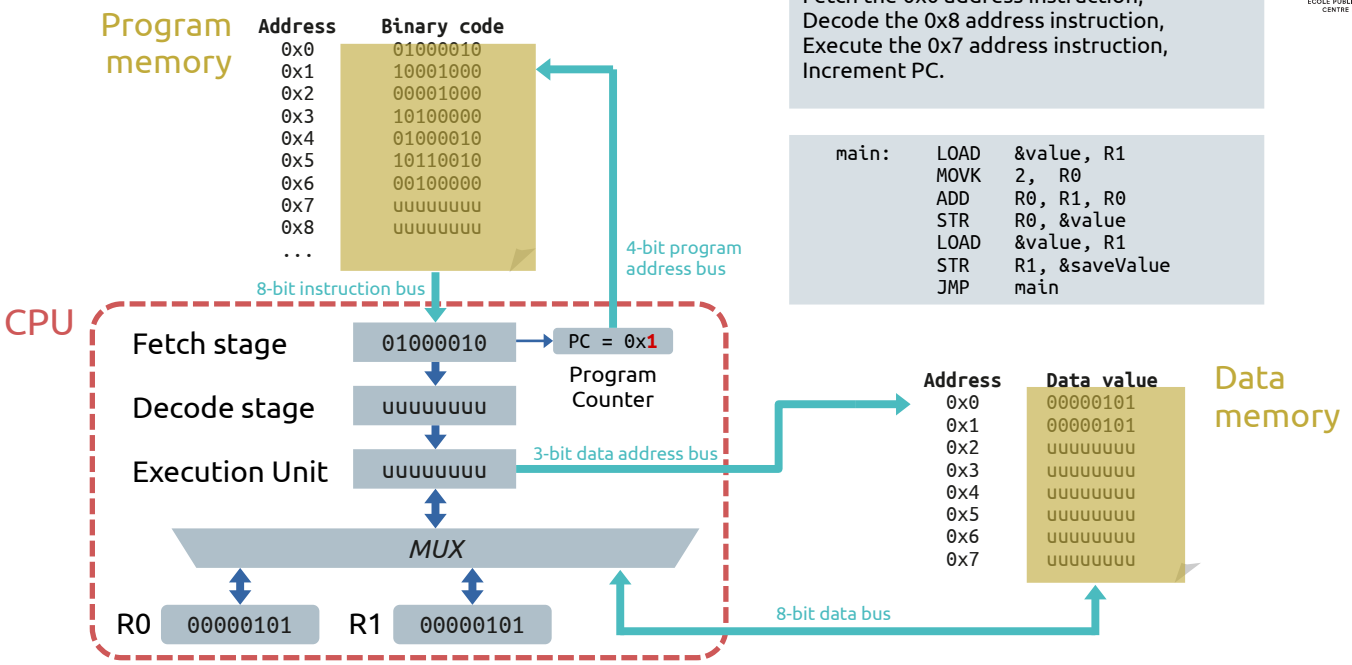


EXECUTING A PROGAM

Home-made processor

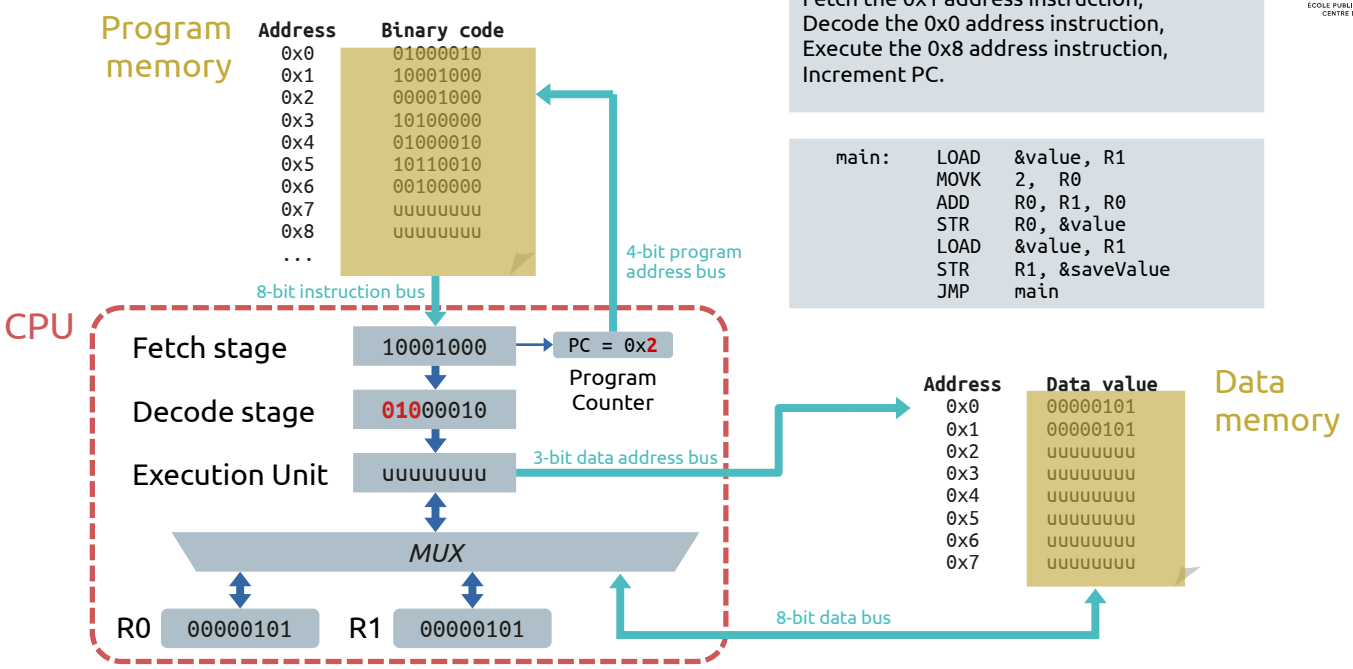
Cycle #10:
Fetch the 0x0 address instruction,
Decode the 0x8 address instruction,
Execute the 0x7 address instruction,
Increment PC.

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



EXECUTING A PROGAM

Home-made processor

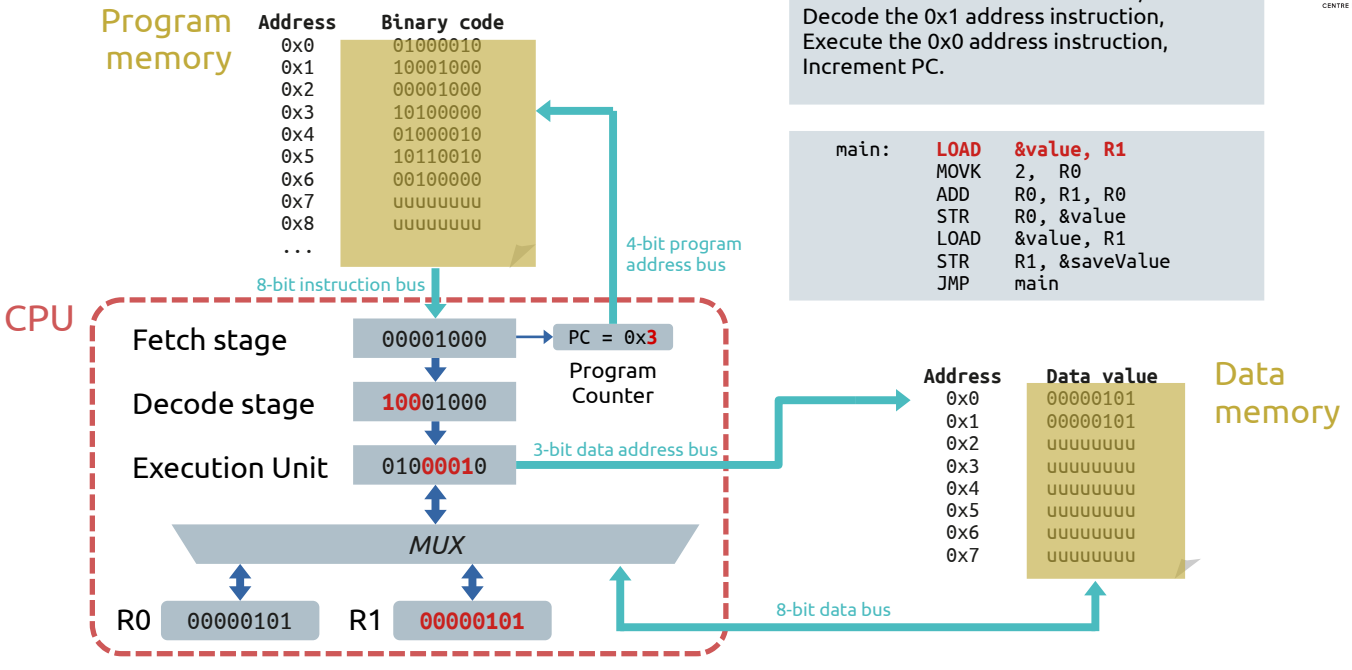


Cycle #11:
Fetch the 0x1 address instruction,
Decode the 0x0 address instruction,
Execute the 0x8 address instruction,
Increment PC.

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

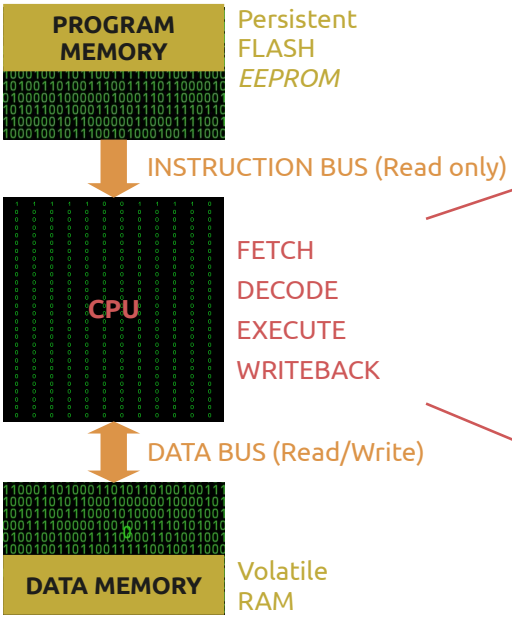

EXECUTING A PROGAM

Home-made processor



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
```



CPU CYCLE

Fetch	Read the next instruction from program memory
Decode	Understand the operation to perform by read the opcode
Execute	Perform the operation (usually performed by ALU)
Store	Save the result into internal registers or the data memory

PERIPHERALS

Examples

PERIPHERALS

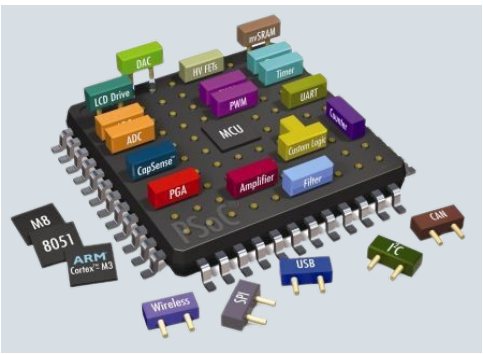
Definition

Peripherals are hardware functions built for specific processing.

The CPU can delegate some operations to dedicated peripherals (counting, FFT, …) in order to keep the CPU executing the application program.

But most of the peripherals are input/output interfaces (General Purpose I/O, analogue I/O, communication…).

Peripherals form a set of hardware services (GPIOs, ADC, timers, SPI/I²C/UART/USB/Eth, …) that differ from a processor to another.

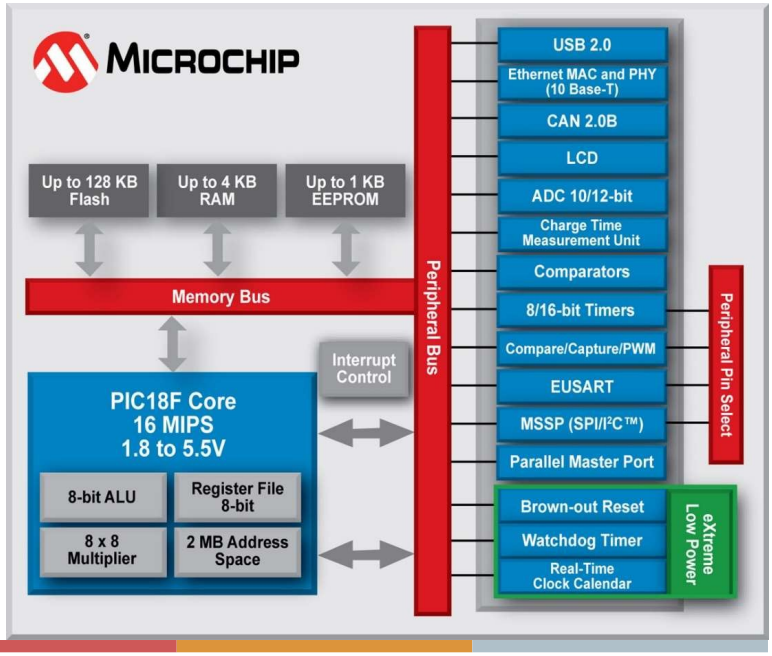


Example

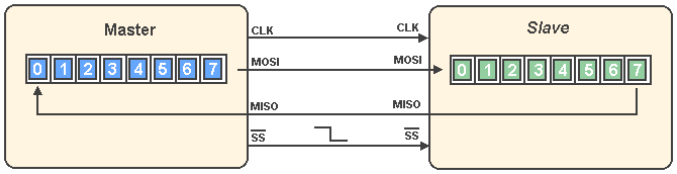
Microchip's PIC18 (8-bit MCU)

This MCU architecture will be used as an example during lessons and will be used in practical labs.

That is why peripherals will not be detailed in this chapter.



The SPI (Serial Peripheral Interface) is a communication protocol widely used on PCBs (Printed Circuit Boards). Designed by Motorola, it operates in full-duplex and use a Master-Slave scheme. The master initiates all communications and command the slaves.

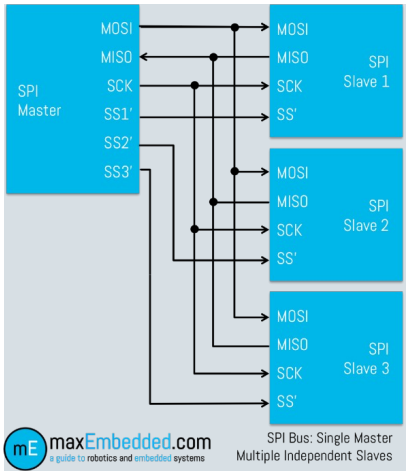


MOSI: Master Output, Slave Input

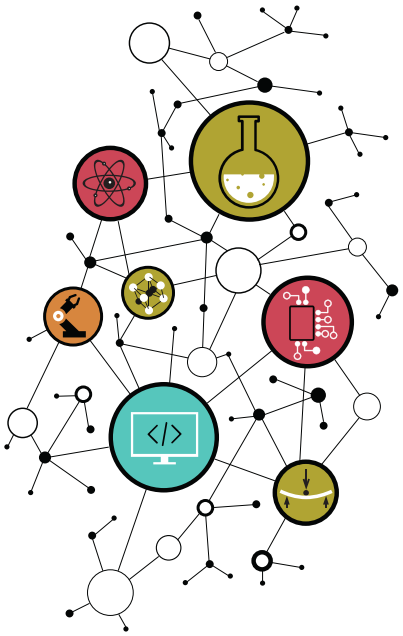
MISO: Master Input, Slave Output

SCK: Serial Clock

SSx: Slave Select (for slave #x)



CONTACT

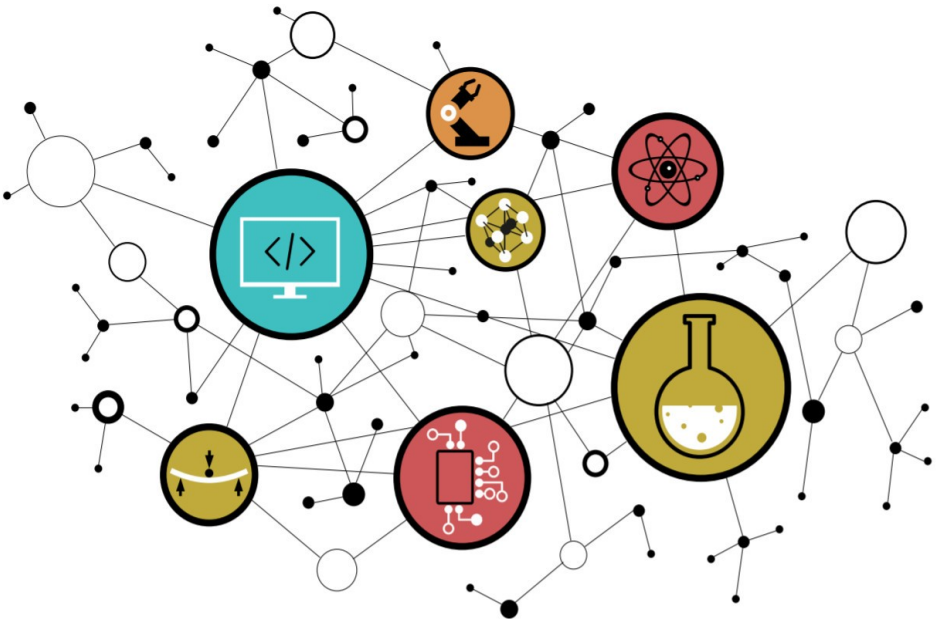


Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

With the precious help of:

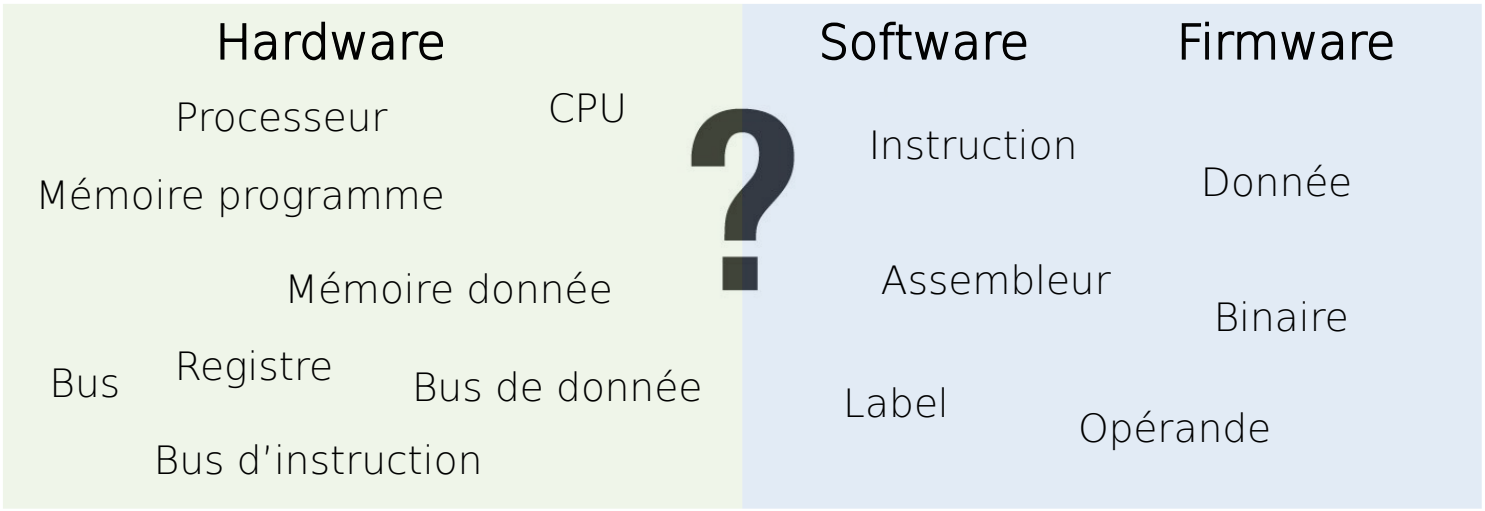
- Hugo Descoubes (PRAG ENSICAEN)
- Bogdan Cretu (MCF ENSICAEN)

MOTS CLÉS

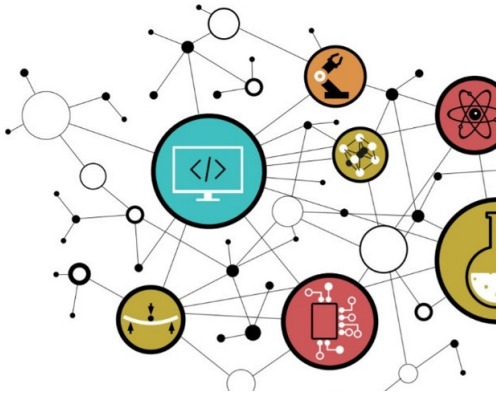


MOTS CLES

Avant d'aborder dans l'enseignement des parties plus technologiques voire standards (toolchain GNU GCC, Kernel Linux, shell Unix, ASM x86_64, carte mère avec chipset Intel, etc), nous allons nous attacher à représenter un jeu de mots clés de base lié au domaine des couches basses des systèmes numériques d'information.



- Langage d'assemblage
- Mémoire et CPU
- Software, Firmware et Hardware
- Périphérique et processeur



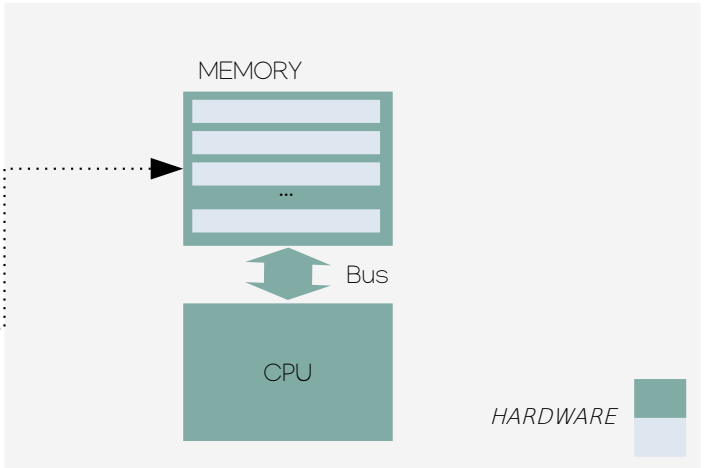
Dans ce document, nous ferons la distinction entre ce qui est matériel (**Hardware – bleu**) et donc qui peut être touché et tenu par l'homme. De ce qui est la représentation logique de signaux électriques (**Software et Firmware – beige**). Une machine électronique numérique de traitement de l'information est chargée de stocker, partager et traiter l'information. Les programmes et données sont stockés en mémoire.

For Human

```
MOVE    data1 to data2
ADD     data1 with data2
SUB     data1 with data2
...
```

For Machine

```
01010101
01001000
10001001
   ...
```



SOFTWARE / FIRMWARE

Quelque soit la famille d'un processeur (GPP, MCU, PA, DSP, etc) et sa technologie (RISC-V, Intel, ARM, AMD, PIC18, etc), nous ne trouverons au niveau physique dans la mémoire de la machine que des instructions et des données :

```
MOVE        data1 to data2
ADD         data1 with data2
SUB         data1 with data2
```

- Une instruction est un ordre impératif pour la machine. Un programme est une suite séquentielle d'instructions répondant à un besoin spécifique ou application. Une fois la version d'un programme figée, celui-ci est statique. La séquence d'instructions est inchangée.

- Une donnée représente une information (valeur, caractère, etc). Les données sont en perpétuel changement et mouvement dans la machine. Elles sont manipulées indirectement par les instructions.

L'assembleur (assembly) ou langage d'assemblage est le langage de programmation de plus bas niveau sur la machine. Il est la conversion directe lisible voire éditable par l'homme (texte) du programme exécutable par le CPU de la machine (binaire). Il est de ce fait, le langage le moins universel au monde, car dépendant du jeu d'instructions supporté par le CPU cible (ISA – Instruction Set Architecture). Entre les marchés des ordinateurs et des systèmes embarqués, un grand nombre de fabricants implémentent des modèles d'exécution (RISC ou CISC, Von Neumann ou Harvard, 8-16-32-64bits, entier voire flottant, VLIW ou superscalaire, vectoriel ou scalaire, etc) sur des technologies différentes (x86, x64, ARM, MIPS, C6000, PIC18, etc).

Labels	Instructions	Opérandes
label:	MOVE	register1, register2
	ADD	address_data1, register2
	SUB	constant, register1

Instructions présentées à titre illustratif, aucune technologie rattachée

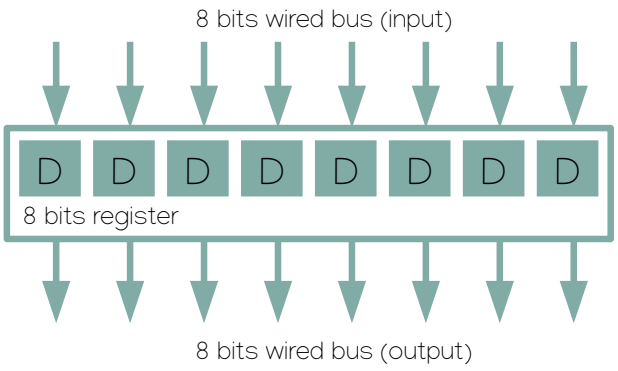
Labels	Instructions	Opérandes
label:	**MOVE**	**register1, register2**
	ADD	**address_data1, register2**
	SUB	**constant, register1**

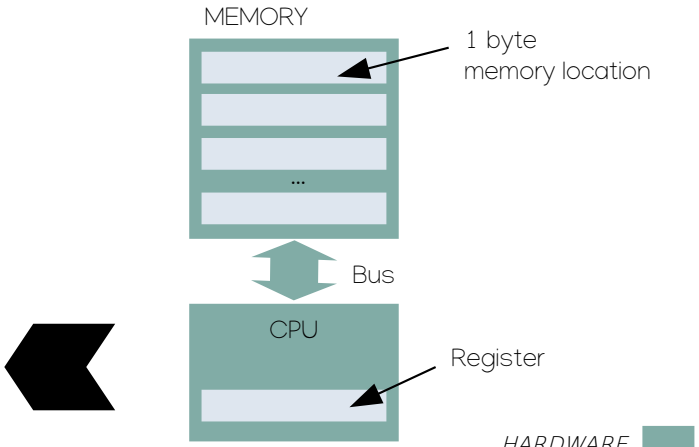
- un label ou étiquette est une référence symbolique (simple chaîne de caractères) représentant l'adresse mémoire logique (emplacement) de la première instruction suivant celui-ci voire d'une variable statique.

- une instruction assembleur est un traitement élémentaire à réaliser par le CPU. Par exemple, charger/load ou sauver/store une donnée depuis ou vers la mémoire, réaliser une opération arithmétique ou logique, déplacer une donnée de registre à registre, etc.

- une opérande, lorsque l'instruction en utilise, est une donnée ou l'emplacement d'une donnée (registre ou adresse mémoire). Nous distinguons l'(es) opérande(s) source(s), de l'opérande de destination pour sauver le résultat (à gauche ou à droite selon convention).

Dans la machine, une donnée ne peut être stockée que dans une case mémoire (technologies SRAM, DRAM, FLASH, etc) ou dans un registre (technologie SRAM – Bascules D). Un registre est un ensemble de bascules. Un registre N bits est constitué généralement N bascules D. De même, les données circulent dans la machine sur des bus de communication filaires. Un bus est un ensemble de conducteurs électriques physiques.





http://ressources.univ-lemans.fr/AccesLibre/UM/Pedago/physique/02/electro/registres.html

Observons une implémentation technologique : Programme source en langage C, ASM x86_64 en syntaxe AT&T généré par GCC sous Linux

Programme source C

```
char inc(char bar);

int main(void)
{
char foo;

    foo = inc(1);

return 0;
}

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

Programme ASM traduit depuis le C

Labels	Instructions	Opérandes
main:	push	%rbp
	mov	%rsp,%rbp
	sub	$0x10,%rsp
	mov	$0x1,%edi
	call	4004f2 <inc>
	mov	%al,-0x1(%rbp)
	mov	$0x0,%eax
	leave	
	ret	
inc:	push	%rbp
	mov	%rsp,%rbp
	mov	%edi,%eax
	mov	%al,-0x4(%rbp)
	movzbl	-0x4(%rbp),%eax
	add	$0x1,%eax
	pop	%rbp
	ret	

Programme binaire converti depuis l'ASM

Instructions binaires	Adresses
55	4004d6 <main>:
48 89 e5	4004d7:
48 83 ec 10	4004da:
bf 01 00 00 00	4004de:
e8 0a 00 00 00	4004e3:
88 45 ff	4004e8:
b8 00 00 00 00	4004eb:
c9	4004f0:
c3	4004f1:
55	4004f2 <inc> :
48 89 e5	4004f3:
89 f8	4004f6:
88 45 fc	4004f8:
0f b6 45 fc	4004fb:
83 c0 01	4004ff:
5d	400502:
c3	400503:

SOFTWARE ou Logiciel

FIRMWARE ou Micrologiciel

MEMORY MAP ou Mapping mémoire


```
char inc(char bar);

int main(void)
{
char foo;

    foo = inc(1);

return 0;
}

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

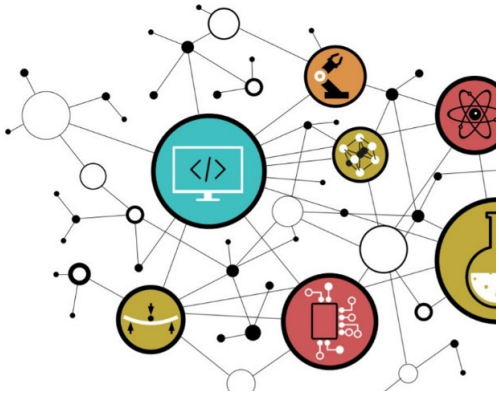
main:	push	%rbp
	mov	%rsp,%rbp
	sub	$0x10,%rsp
	mov	$0x1,%edi
	call	4004f2 <inc>
	mov	%al,-0x1(%rbp)
	mov	$0x0,%eax
	leave	
	ret	
inc:	push	%rbp
	mov	%rsp,%rbp
	mov	%edi,%eax
	mov	%al,-0x4(%rbp)
	movzbl	-0x4(%rbp),%eax
	add	$0x1,%eax
	pop	%rbp
	ret	

55	4004d6 <main>:
48 89 e5	4004d7:
48 83 ec 10	4004da:
bf 01 00 00 00	4004de:
e8 0a 00 00 00	4004e3:
88 45 ff	4004e8:
b8 00 00 00 00	4004eb:
c9	4004f0:
c3	4004f1:
55	4004f2 <inc>:
48 89 e5	4004f3:
89 f8	4004f6:
88 45 fc	4004f8:
0f b6 45 fc	4004fb:
83 c0 01	4004ff:
5d	400502:
c3	400503:

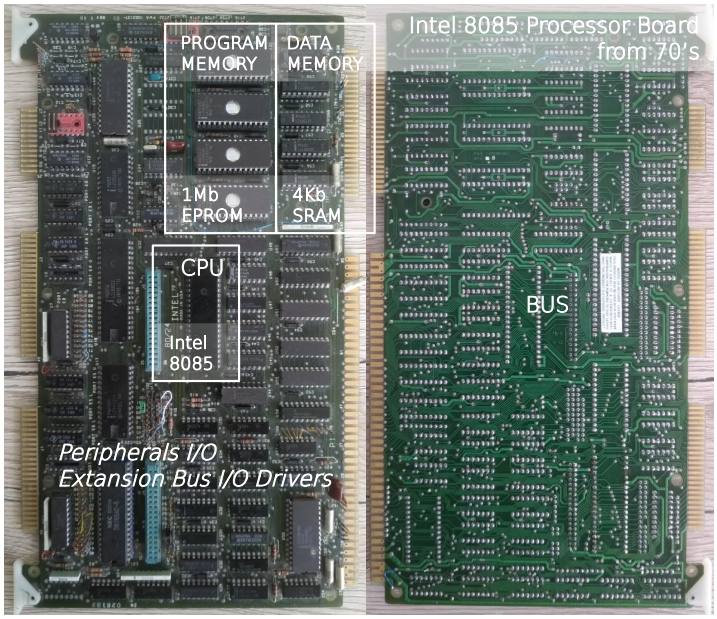
Programme C et ASM dans fichier texte (.c, .h, .s, .asm, etc) pour l'humain (développeur)

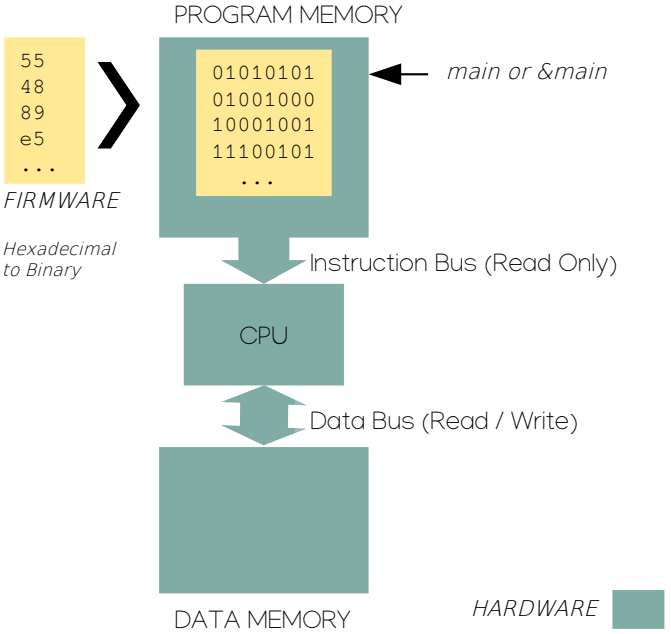
Programme binaire dans fichier binaire (ELF, COFF,etc) pour la machine

- Langage d'assemblage
- **Mémoire et CPU**
- Software, Firmware et Hardware
- Périphérique et processeur

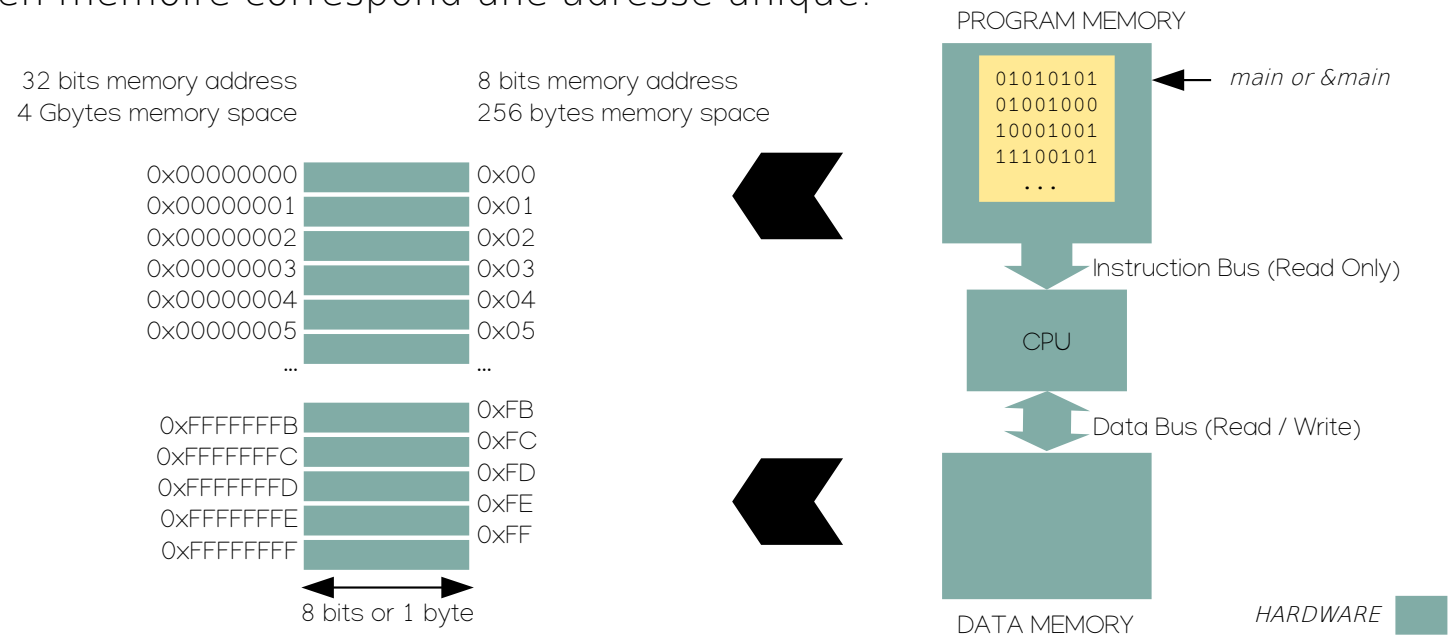


Sur la majorité des processeurs actuels, le stockage des programmes binaires et des données se font dans des mémoires et technologies physiquement séparées autour du CPU.

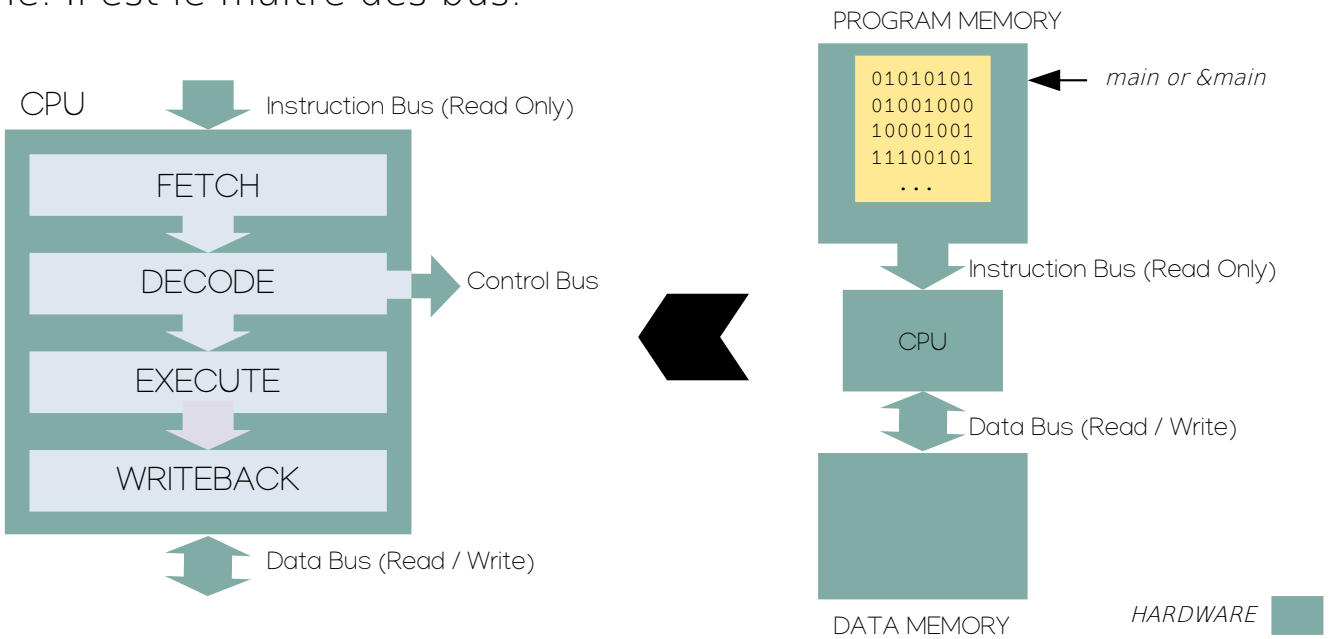




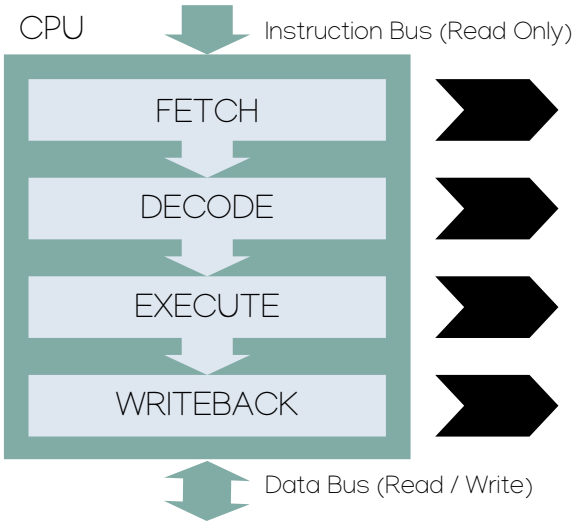
Les modèles mémoires vus du CPU (hors mémoire cache et mémoire de masse) sont dits **adressables par octet**. A chaque octet de stockage en mémoire correspond une adresse unique.



Le CPU (Central Processing Unit) est l'unité de contrôle du processeur dans son ensemble. Le CPU ordonne à tous les composants de la machine. Il est le maître des bus.



Un CPU implémente une machine d'états matérielle. Sauf en veille, le CPU exécutera sans arrêt la même suite séquentielle de traitements (Fetch, Decode, Execute et Writeback). Chaque étape se fait en parallèle des autres sur un même rythme. Nous parlons de **Pipeline hardware**.



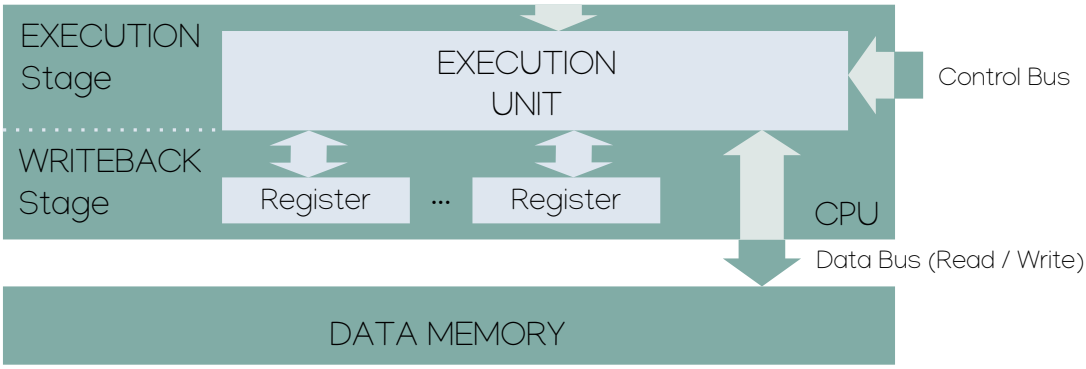
COPIER une instruction binaire à exécuter de la mémoire programme vers le registre de fetch du CPU

IDENTIFIER l'instruction binaire et ORDONNER par le bus de contrôle aux composants de la machine

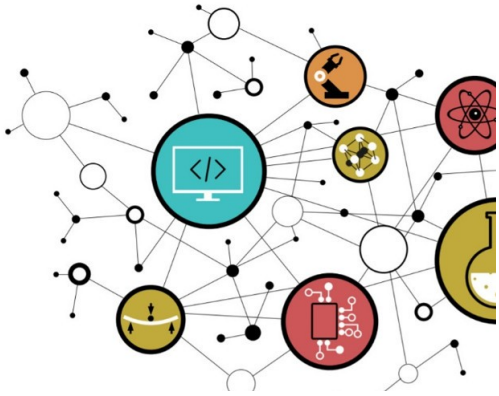
FAIRE le traitement (opération Arithmétique et Logique, déplacement de donnée, calcul d'adresse, etc)

SAUVER " éventuellement " le résultat d'un traitement dans un registre CPU ou en mémoire donnée

L'étage d'exécution est constitué d'une voire plusieurs unités matérielles d'exécution (EU ou Execution Unit). L'unité d'exécution est capable de réaliser des opérations de calcul arithmétique et logique (+, -, *, etc) sur différents formats de donnée selon la technologie CPU (entier 8-16-32-64bits voire flottants). L'unité de calcul est parfois nommée **ALU** (Arithmetic Logic Unit). L'unité d'exécution est également capable de manipuler (chercher et stocker) une donnée soit dans un registre CPU pour une utilisation en cours soit dans la mémoire donnée.



- Langage d'assemblage
- Mémoire et CPU
- **Software, Firmware et Hardware**
- Périphérique et processeur



SOFTWARE

```
int data1 = 1, data2 ;

data2 = data1 ;
data1 = data1 + data2 ;
data1 = data1 - data2 ;
```

```
MOVE    data1 to data2
ADD     data1 with data2
SUB     data1 with data2
...
```

FIRMWARE

```
00110011
11000010
11010010
...
```

- Un software ou logiciel (hérité du mot logique) représente une séquence d'instructions abstraites au regard du fonctionnement physique et électronique de la machine (signaux électriques à logique binaire). Les programmes logiciel sont rangés dans des fichiers texte (.c, .h, .s, .cpp, etc pour l'humain) dépendant de la technologie du langage de programmation utilisé (C, ASM, C++, D, JAVA, etc). Ces fichiers et programmes sont à visée de l'homme (développeur ou administrateur). Un logiciel peut être qualifié de système, applicatif, standard, spécifique, libre, propriétaire, etc

SOFTWARE

```
int data1 = 1, data2 ;

data2 = data1 ;
data1 = data1 + data2 ;
data1 = data1 – data2 ;
```

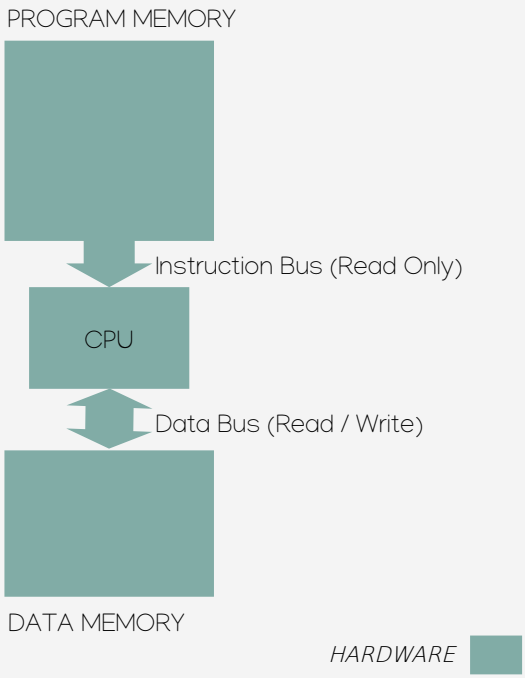
```
MOVE    data1 to data2
ADD     data1 with data2
SUB     data1 with data2
...
```

FIRMWARE

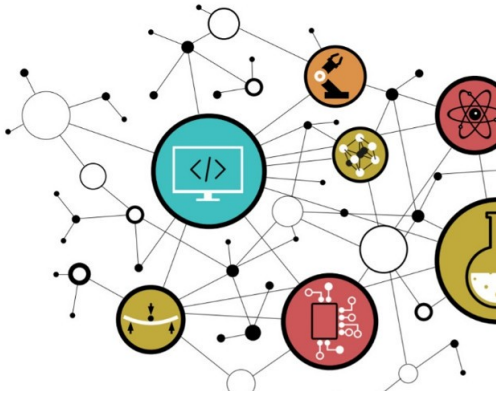
```
00110011
11000010
11010010
...
```

- Un firmware ou micrologiciel représentera dans cet enseignement la conversion directe sans interprétation de la séquence d'instructions assembleur ainsi que l'ensemble des données statiques générées ou allouées à la compilation et l'édition des liens. Un langage d'assemblage n'étant ni standard ni normé mais étant spécifique à l'architecture CPU cible, le code binaire correspondant est également spécifique. Un CPU Intel x86_64 sur ordinateur ne pourra pas exécuter un code binaire pour CPU ARM pour smartphone. En fonction du contexte, nous pourrons appeler quelquefois le code binaire seul le firmware (sans les données et sections de données statiques)

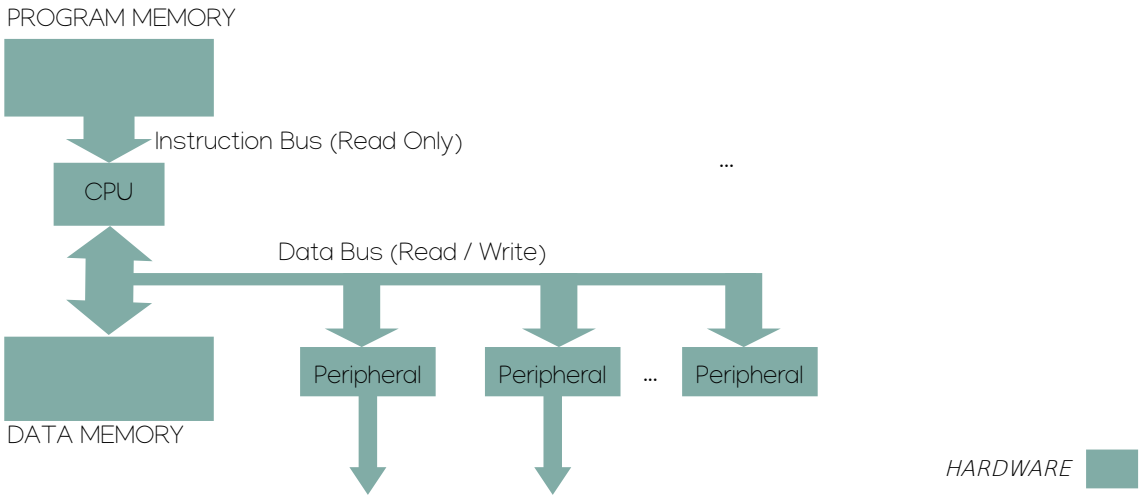
- L'architecture hardware ou matérielle correspond au système électronique numérique physique de traitement de l'information. Il s'agit d'un système à logique binaire (état logique 1 ou 0, haut ou bas). Ces systèmes proposent 3 services :
    - STOCKER l'information : Mémoires programme et donnée
    - TRAITER l'information : CPU
    - PARTAGER l'information : Bus et périphériques



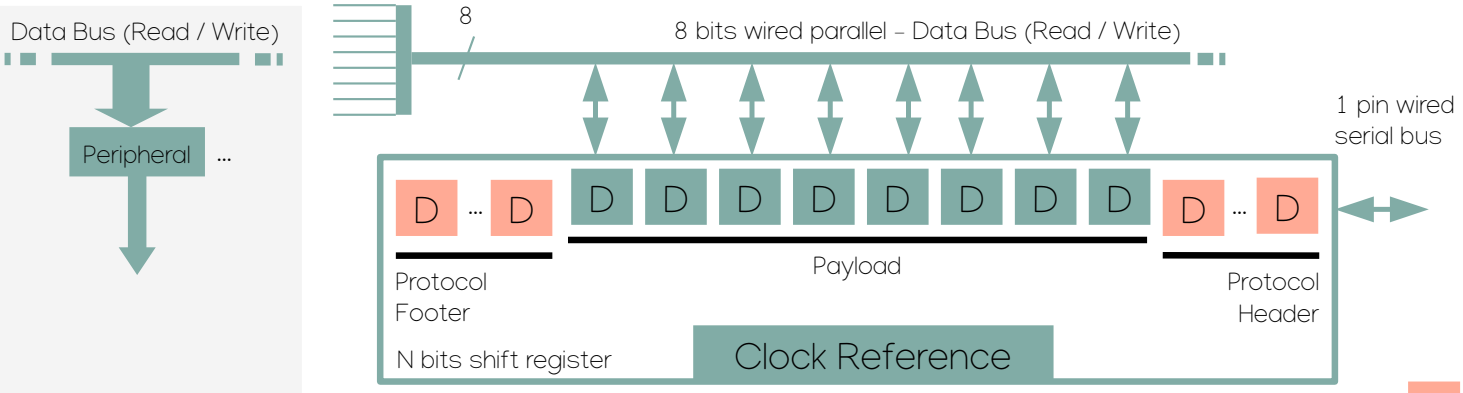
- Langage d'assemblage
- Mémoire et CPU
- Software, Firmware et Hardware
- **Périphérique et processeur**



L'échange d'information avec l'extérieur de la machine se fait en passant par les **fonctions matérielles périphériques**, plus communément appelés périphériques. Nous parlons de périphériques par rapport au couple CPU/Mémoire permettant de stocker et traiter l'information. Chaque périphérique implémente une fonction matérielle spécifique.



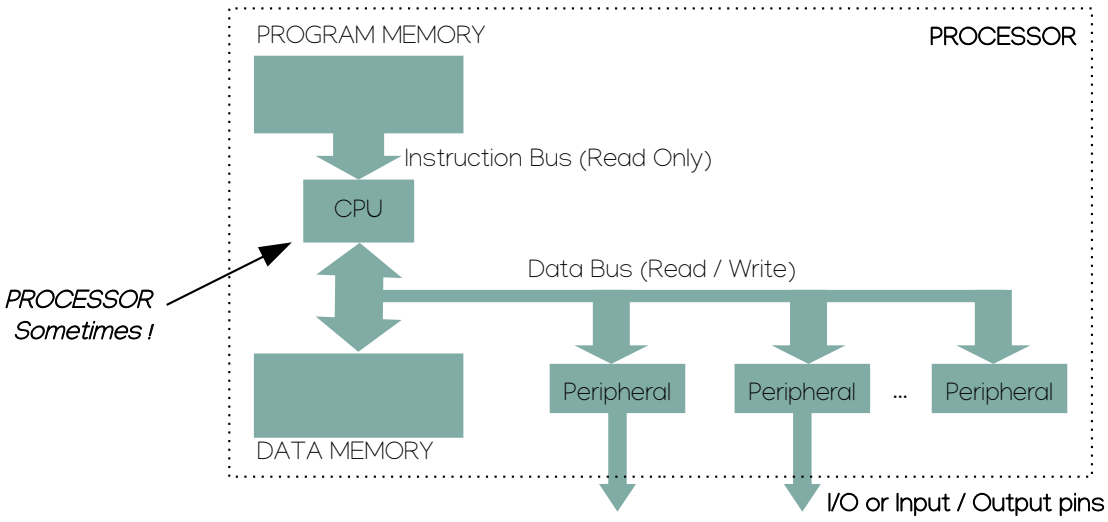
Beaucoup de fonctions périphériques sont chargées de partager l'information avec l'extérieur de la machine. Chaque périphérique implémente alors un protocole de communication particulier (USB, Ethernet, UART, SPI, I2C, etc). Les données utiles (payload) circulent par le bus parallèle de donnée du processeur pour être chargées dans les registres à décalage des périphériques de communication. Les données sont transmises ensuite bit après bit en série à un rythme donné.



http://ressources.univ-lemans.fr/AccesLibre/UM/Pedago/physique/02/electro/registres.html

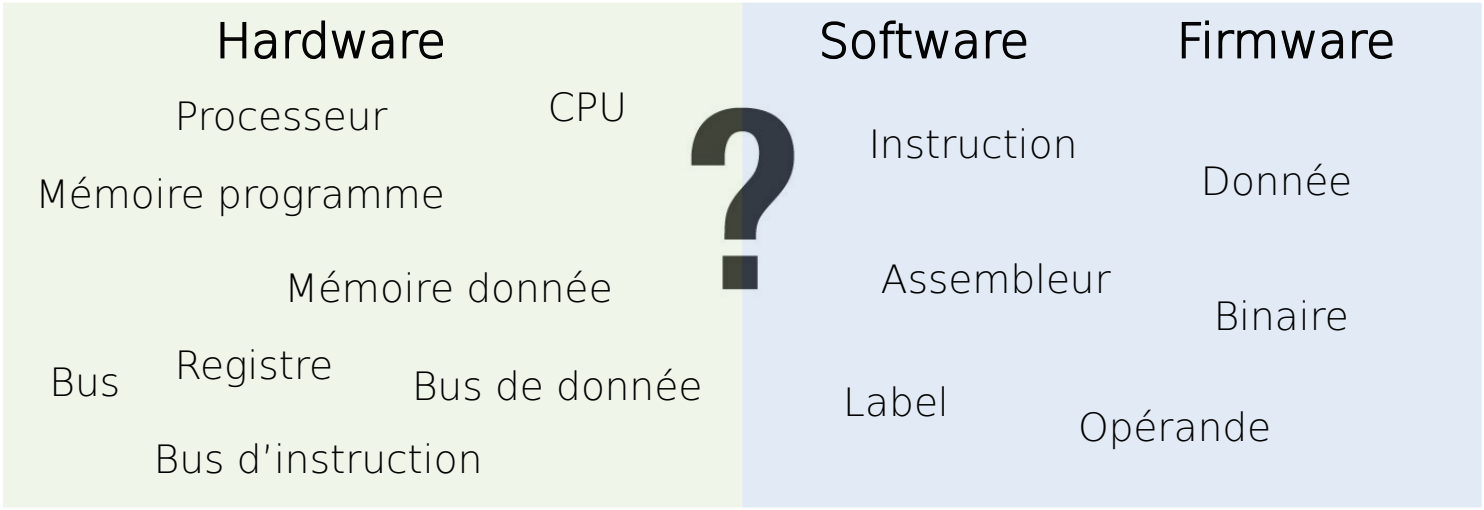
HARDWARE

Dans cet enseignement, en fonction du contexte, nous nommerons **processeur** soient les familles de machines numériques matérielles implémentant un ensemble CPU/Mémoires/Bus/Périphériques (MCU, AP, DSP, MMPA, GPU, etc) soit dans certains cas le CPU seul (GPP, etc).



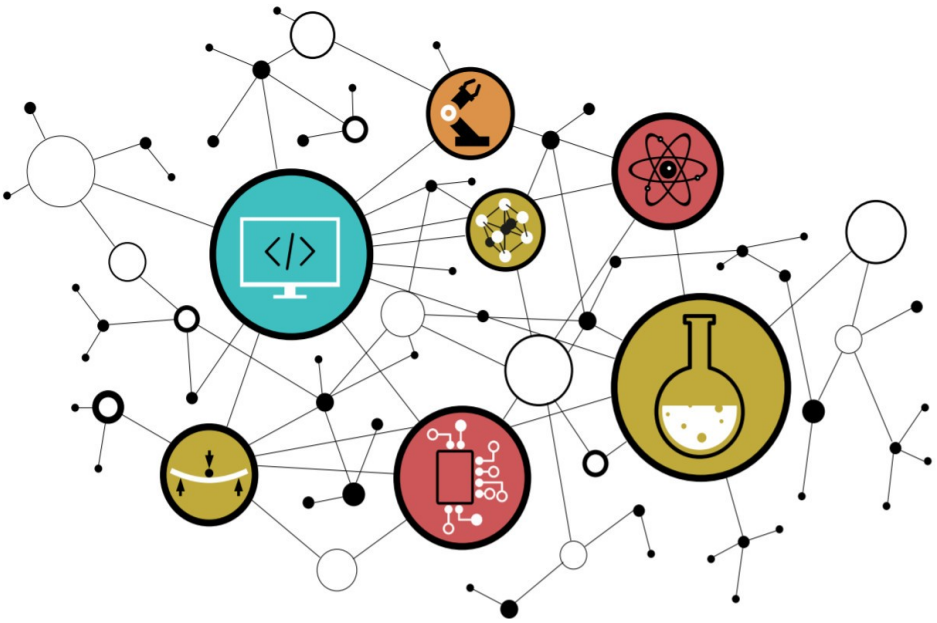
HARDWARE

Ce jeu de mots clés est central afin de bien se comprendre dans la suite des enseignements en Systèmes Embarqués en formation. Progressivement, la suite de l'enseignement nous permettra d'apprécier certaines subtilités d'implémentation technologiques logicielles et matérielles de solutions leaders du marché (GNU\Linux, GCC, Intel, etc)



Merci !

DIVERSITÉ DES PROCESSEURS



Chapitre 1

Diversité des Architectures Processeur

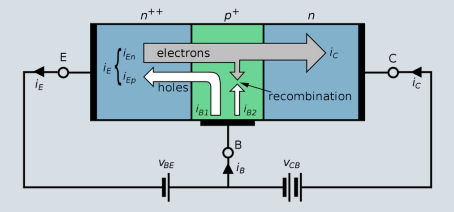
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

Bref rappel (cf cours de Systèmes embarqués)

1947: Invention du Transistor à Jonction Bipolaire →

par Bardeen, Schokley et Brattain (Bell labs), Lauréats du Prix Nobel

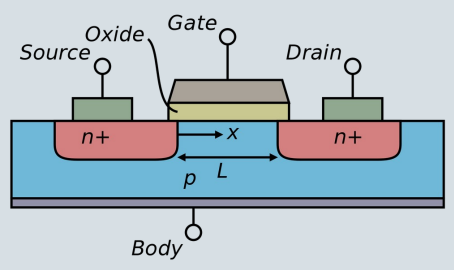


1958/1959: Création des Circuits Intégrés

par Texas Instruments (IC hybride), puis Fairchild (vrai IC monolithique)

1960: Invention du Transistor à Effet de Champ MOS →

par Mohammed Atalla et Dawon Kahng



Le premier processeur commercial est le 4004, annoncé par Intel le 15 novembre 1971.

En réalité, l'armée américaine avait déjà développé un processeur en juin 1970, gardé secret pour le F-14.

À titre de comparaison, la mission Apollo 11 s'est déroulée deux ans plus tôt !

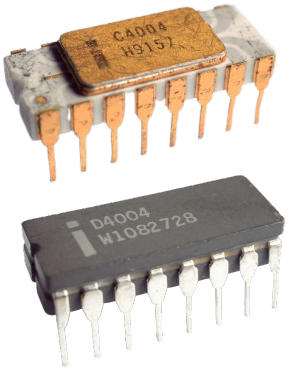
Le 4004 possède 2 300 transistors gravés en 10 µm.

C'est un processeur 4 bits, à 16 broches.

Son ISA compte 45 instructions,
dont du saut conditionnel et de l'appel de fonction.

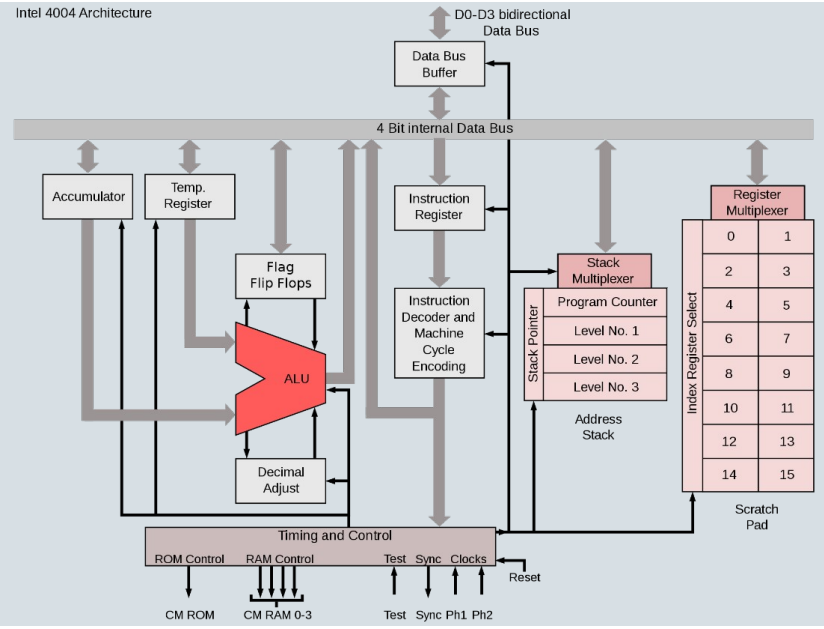
Cadencé à 740 kHz, il peut alors réaliser 90 kIPS.

Le tout pour la modique somme de 60 $!



L'architecture du 4004 reste la base de tous les processeurs modernes.

À comparer avec le PIC18 une fois son architecture étudiée !



Pour les fans de transistors, le schéma est visible ici :

https://www.framboise314.fr/le-microprocesseur-a-50-ans-intel-4004/

Intel 4004

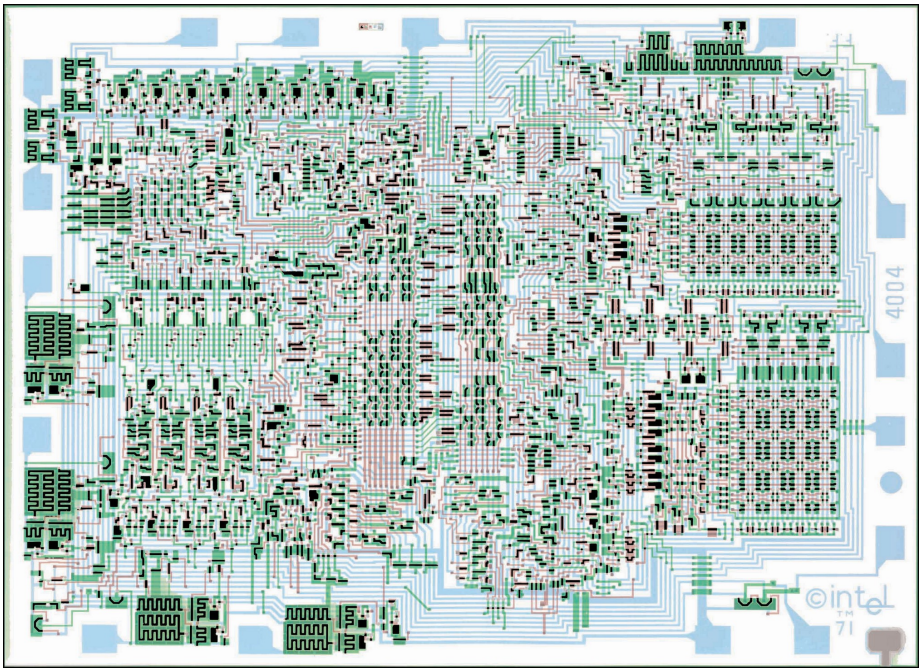


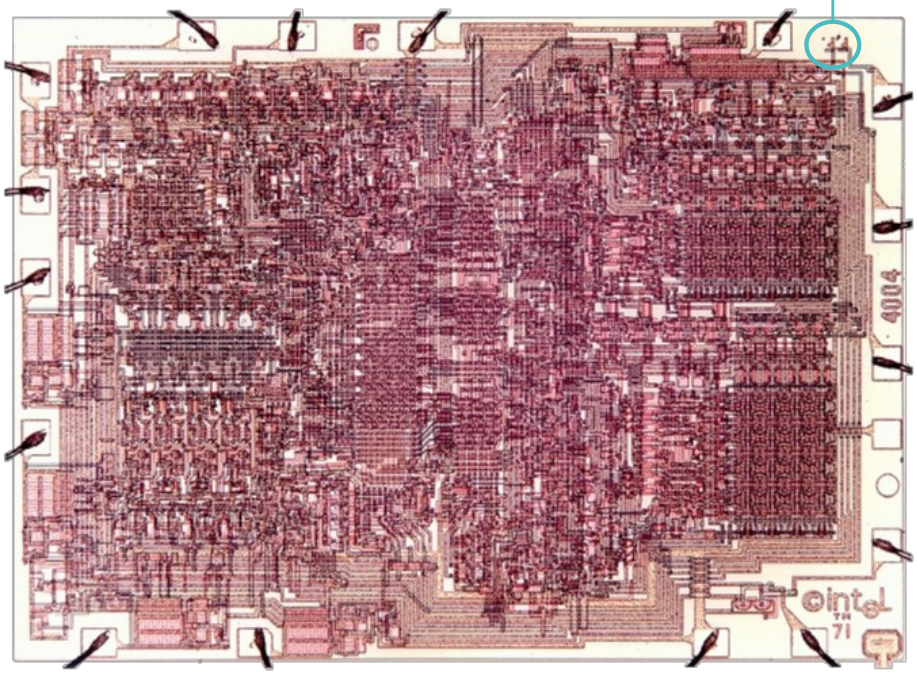
Schéma d'implantation.

Objectif :

Polariser, assembler les 2300 transistors pour réaliser les différentes fonctions logiques.

À l'époque, pas d'outil informatique : tout ce travail est fait à la main !

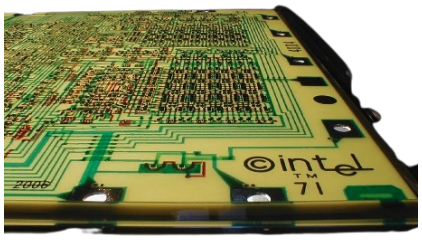
Intel 4004



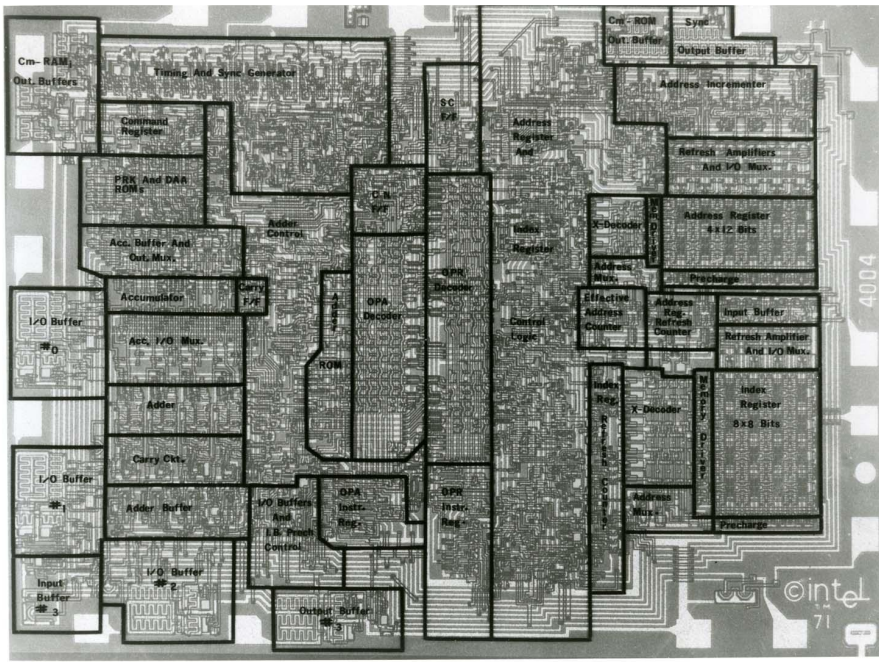
Initiales de Federico Faggin, concepteur du 4004, 8008, 4040 et 8080 !

Photographie par MEB.

On voit encore les fils d'or reliant les pads du die aux broches du boîtier.



Intel 4004



Découpage du schéma d'implantation en blocs fonctionnels.

Simulation à 6 cycles par seconde (90 kIPS IRL) :

https://www.youtube.com/watch?v=0Fixr39X8S4

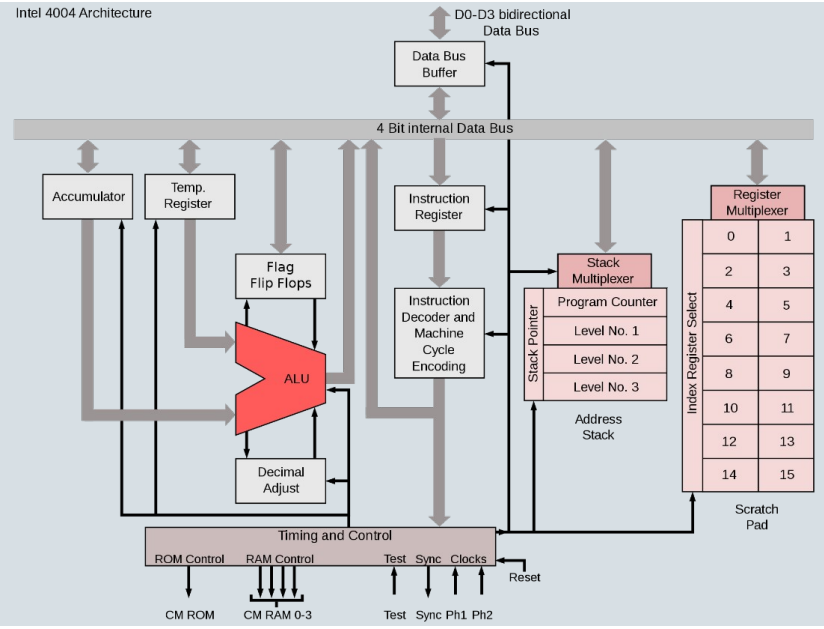
Intel 4004

L'architecture du 4004 reste la base de tous les processeurs modernes.

À comparer avec le PIC18 une fois son architecture étudiée !

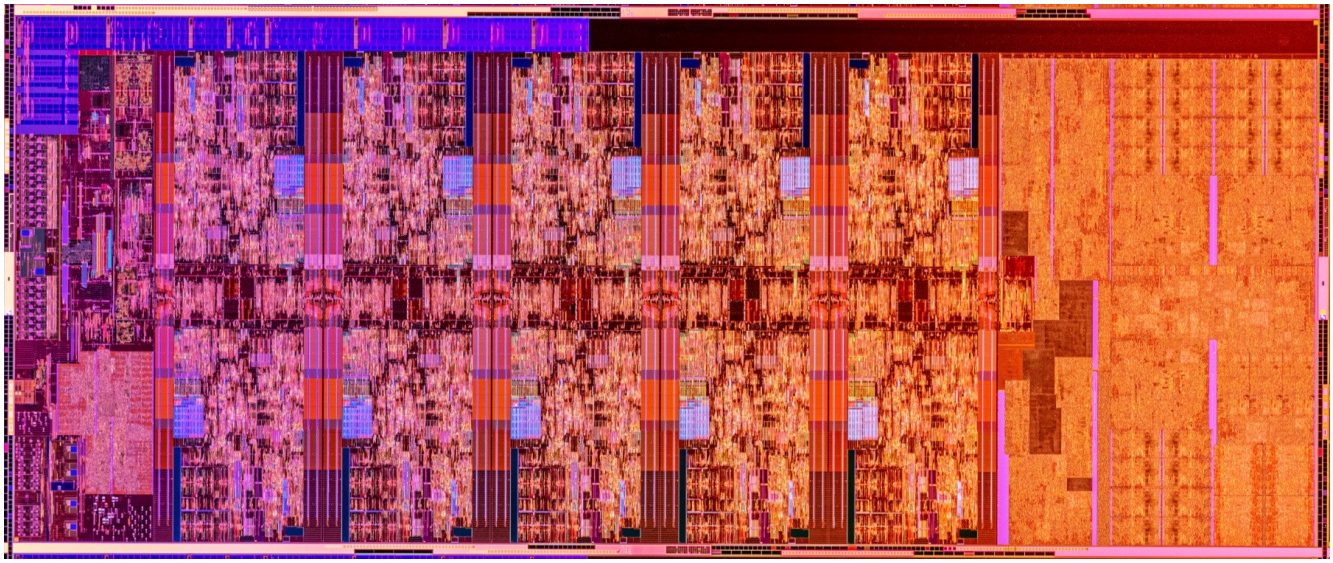
Pour les fans de transistors, le schéma est visible ici :

https://www.framboise314.fr/le-microprocesseur-a-50-ans-intel-4004/

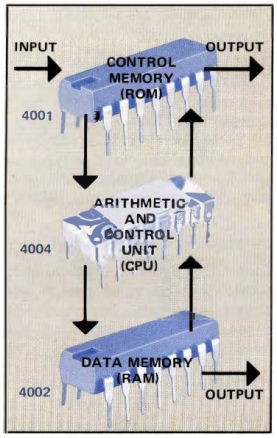


Intel® Core™ i9-10900K Processor

2e trimestre 2020
14 nm (estimé à 7 milliards de transistors)
10 coeurs, 5.30 GHz, 460.8 GFlops



Le 4004 a été conçu pour une machine à calculer de *Busicom Corporation (la 141-PF)*.
Il est alors associé à d'autres composants pour former le **chipset Intel MCS-4**.

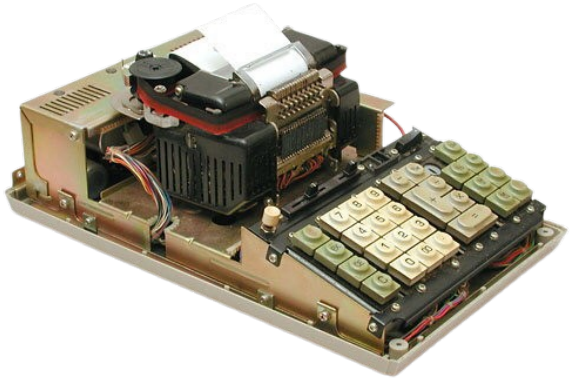


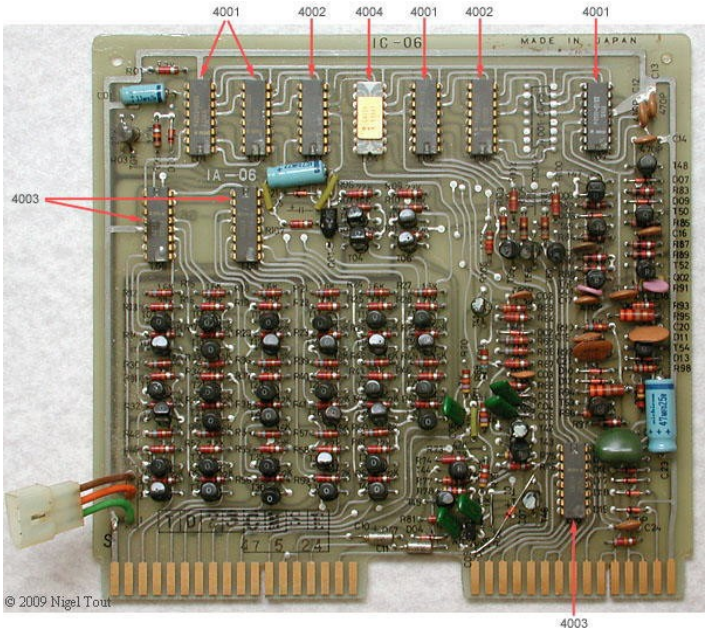
intel MCS-4 MICRO COMPUTER SET
NOVEMBER 1971

4001 : 256 x 8 bit ROM
4002 : 320 bit RAM
4003 : 10 bit shift register
4004 : 4 bit CPU

Source : http://ieeemilestones.ethw.org/images/2/2d/Ref2-Intel_MCS-4_DataSheet.pdf

Intel 4004





Busicom 141-PF (ou NCR-18-36)

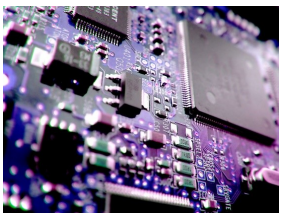
Bloc d'alimentation au fond,
PCB unique en dessous.

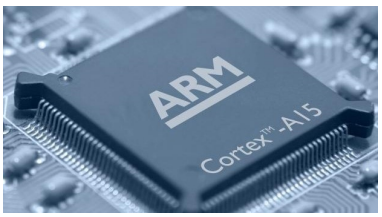
Source : http://www.vintagecalculators.com/html/busicom_141-pf.html

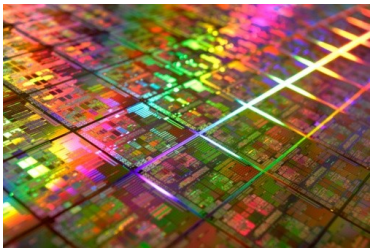
Évolution des processeurs

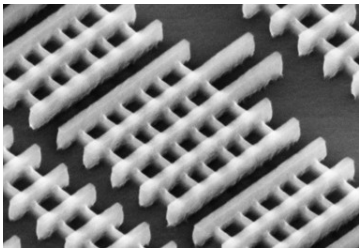
Depuis les processeurs évoluent en suivant la loi de sélection naturelle.

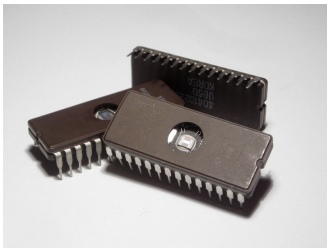
Ceux répondant à des besoins spécifiques ont évolués (et se sont améliorés) tandis que d'autres ont disparu des marchés et laboratoires de recherche.

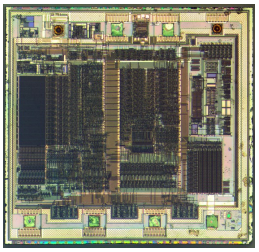












Comme pour le vivant, le processus d'évolution des processeurs ne s'arrêtera pas.
De nouvelles architectures sont susceptibles d'apparaître dans les prochaines années !

Jetons un œil aux architectures actuelles.

Architectures généralistes *Processeurs de contrôle*			**Architectures hybrides**	**Architectures spécialisées** *Coprocesseurs ou processeurs de calcul*		
MCU	**AP**	**GPP**	**SoC / SoB**	**FPGA**	**DSP**	**(GP) GPU**
Micro Controller Unit	Application Processor	General Purpose Processor	System on Chip / Board - FPGA-AP - FPGA-MCU - GPP-GPU - AP - MCU-analog	Field Programmable Gate Array	Digital Signal Processor	Graphics Processing Unit General Purpose GPU
		Computer →				
CONTROL					**CALCULUS**	

Architectures généralistes
Processeurs de contrôle

Architectures hybrides

Architectures spécialisées
Coprocesseurs ou processeurs de calcul

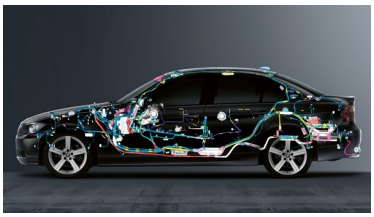
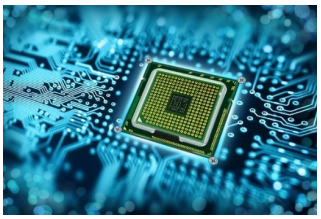
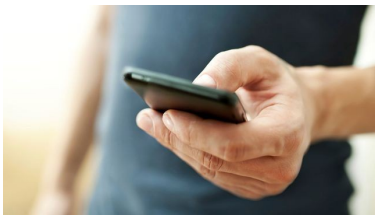
MCU	AP	GPP	SoC / SoB	FPGA	DSP	(GP) GPU
Micro Controller Unit	Application Processor	General Purpose Processor	System on Chip / Board - FPGA-AP - FPGA-MCU - GPP-GPU - AP - MCU-analog	Field Programmable Gate Array	Digital Signal Processor	Graphics Processing Unit General Purpose GPU
CPU				LOGIC	CPU	

MCU
MICROCONTROLLER UNIT

Applications
Architectures
Fabricants et produits
Parts de marché

Les micro-contrôleurs (MCU, *Microcontroller Units*) sont les processeurs les plus répandus dans notre environnement.

De près ou de loin, nous utilisons environ 200 processeurs par jour !



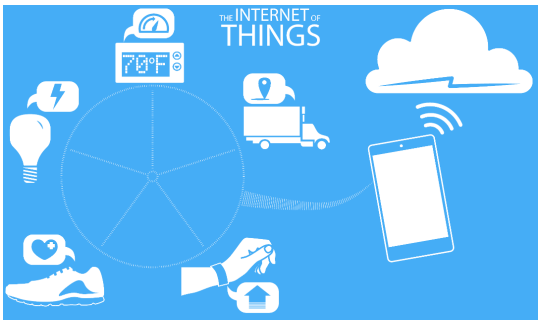


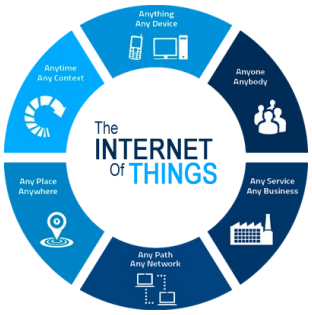
Les micro-contrôleurs sont des processurs dédiés à la supervision des systèmes électroniques. Ils contrôlent leur environnement via leurs interfaces et leur firmware embarqué développé pour une application spécifique.

Ils ciblent des marchés où les applications sont faible coût, faible consommation, faible encombrement et gros volumes de production.

L'un des marchés phares actuels des MCU est celui des objets connectés (IoT ou *Internet of Things*). L'IoT représente l'extension d'Internet à des objets et lieux du monde physique. Il est considéré comme la troisième évolution d'Internet et, à ce titre, a été baptisé « Web 3.0 ».

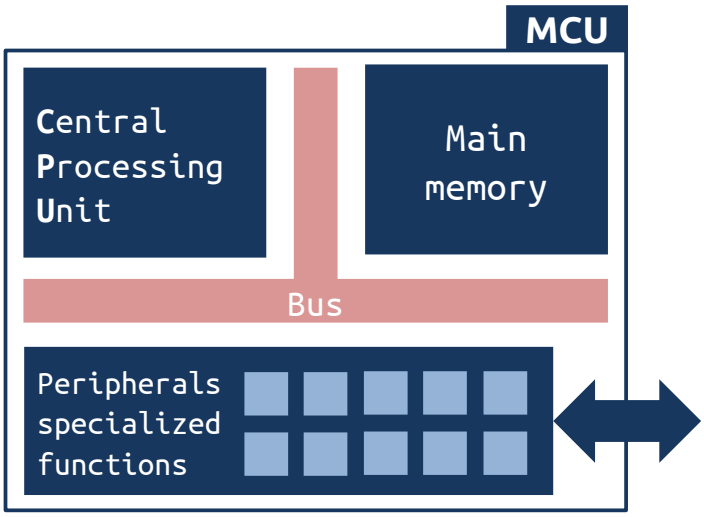
Avec 3,6 milliards de connexions actives en 2015, 11,7 milliards en 2020 et 30 milliards prévues en 2025, l'IoT représentait 18 % des MCU en 2019 et 29 % en 2025.





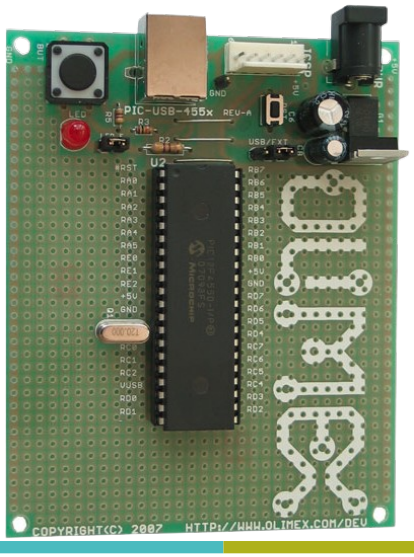
Ces processeurs sont des systèmes numériques intégrés sur puce.

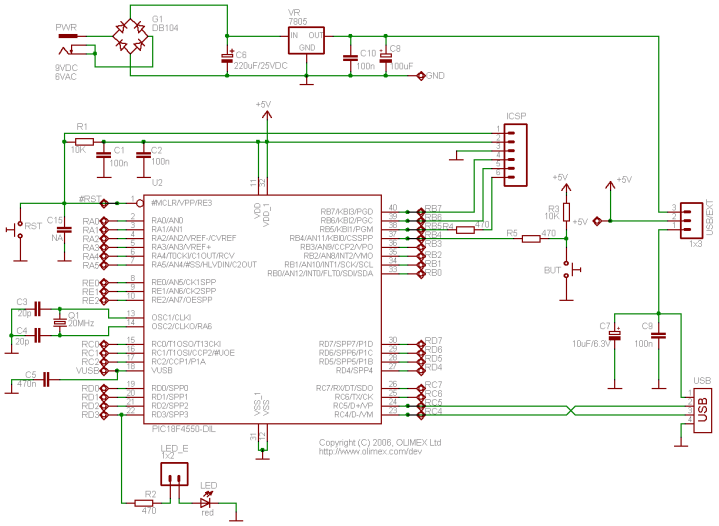
Ils sont pensés pour être autonomes (pas besoin de RAM, de HDD, …).



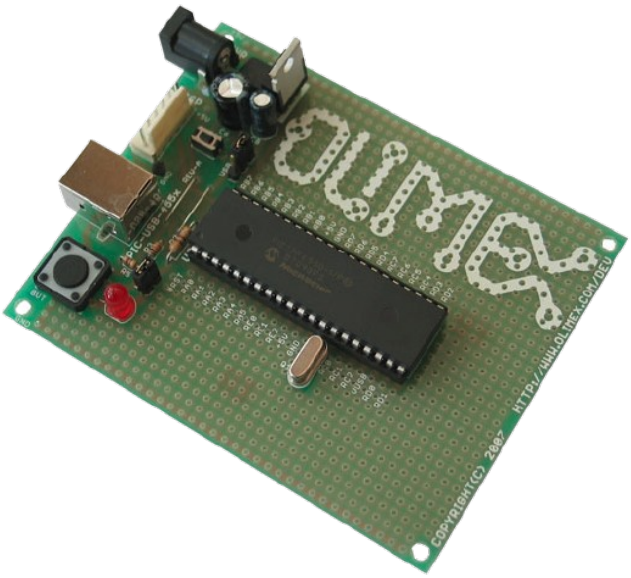
Exemple de schéma utilisant un PIC18 de Microchip.

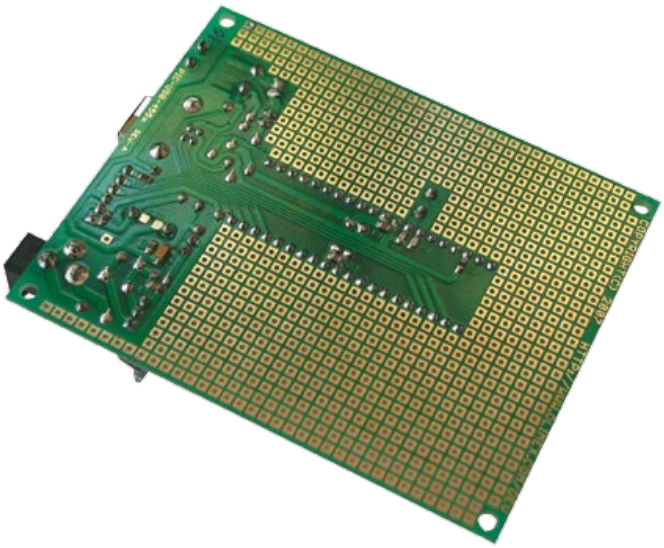
Olimex PIC-USB-4550 board.





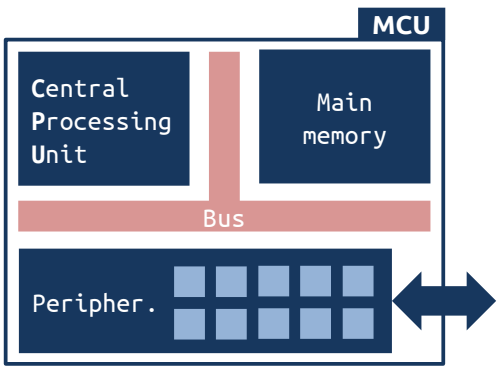
Exercice : repérez les composants du schéma précédent sur les photos ci-dessous.



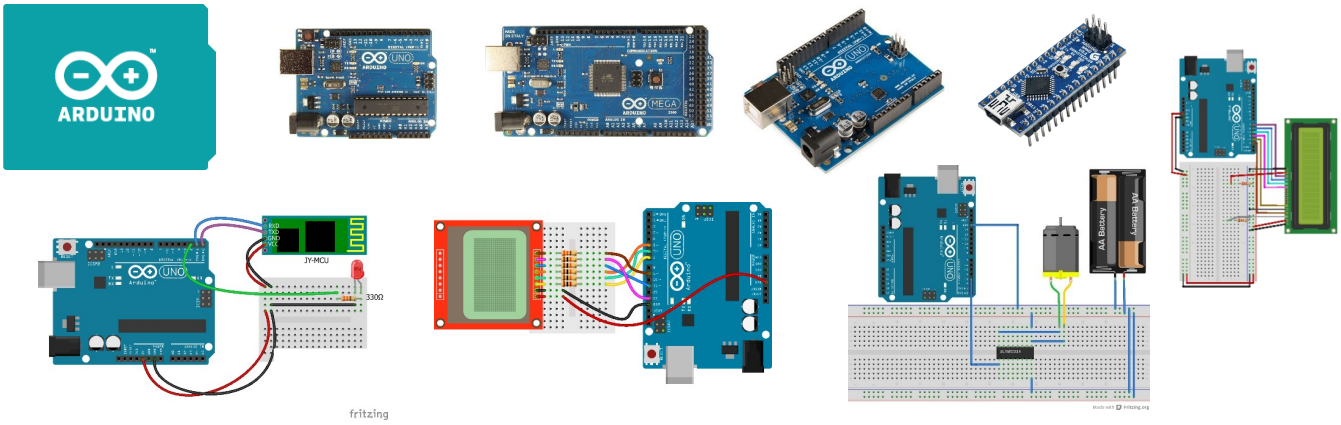


Il existe un très grand nombre de solutions MCU chez différents fondeurs, permettant de résoudre un cahier des charges.

Les MCU d'une même famille sont caractérisés par le même CPU et bus associés. Le jeu d'instructions (ISA, *Instruction Set Architecture*) et donc les outils de compilation sont similaires. Ce qui différencie les MCU d'une même famille sera le jeu de périphériques associés et les ressources mémoire disponibles.

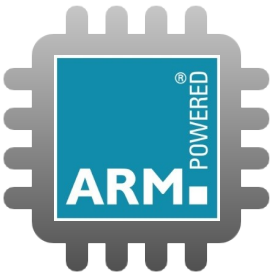


Sûrement le plus populaire des projets électroniques basés sur un MCU, il reste déprécié en enseignements ingénieurs pour son côté trop *friendly/maker* et sa non-application aux marchés en sortie d'école.



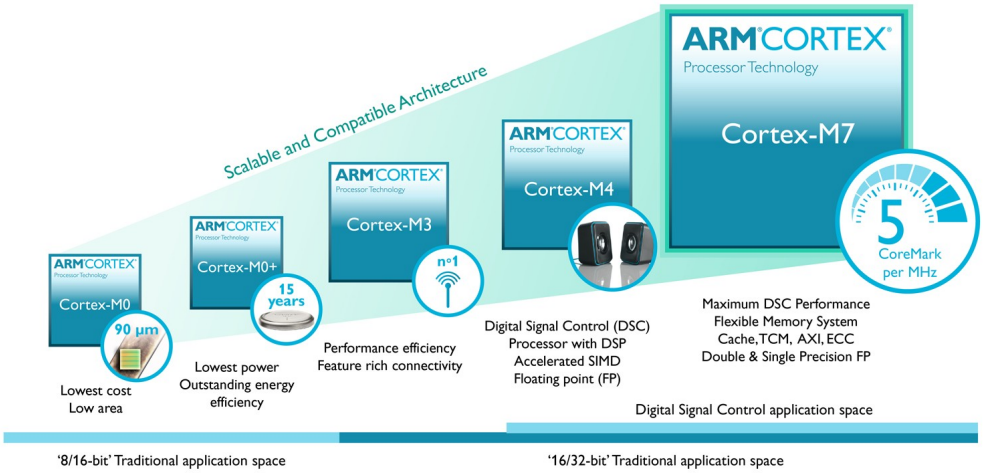
Même si le marché des MCU reste concurrentiel, la grande majorité des fondeurs de MCU (STMicro, Renesas, Texas Instruments, NXP, …) utilisent des architectures CPU similaires, toutes proposées par la société ARM : la famille des **Cortex-M**.

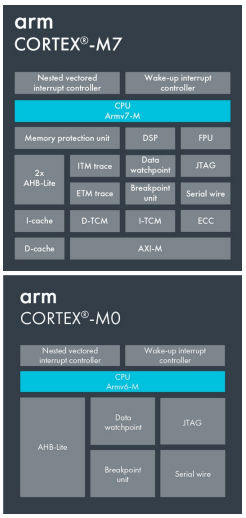
Cela garanti un accès à des outils de développement, bibliothèques et services logiciels fiables, pouvant être libres et open-source (IP / *Graphical* / USB / Bluetooth, *stack*, RTOS, …).



ARM propose la série des processeurs Cortex-M, où M signifie MCU.

Cette série comporte toute une famille de cœurs pour MCU adaptée à un large choix d'application.

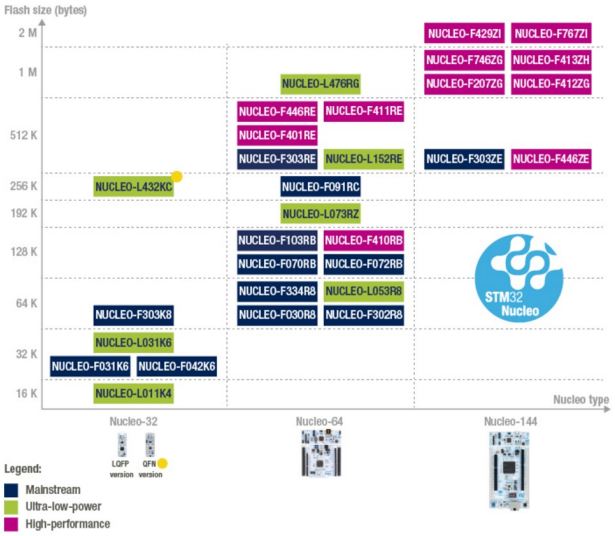


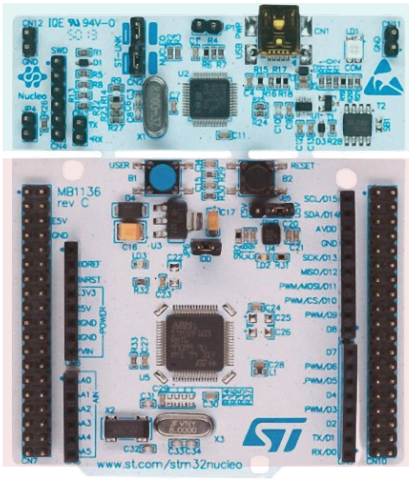


Observons à titre d'illustration les gammes des STM32, qui sont des MCU 32-bits basés sur un coeur ARM Cortex-M.

Ils sont proposés par la société STMicroelectronics, société franco-italienne et principal fondeur européen.

Le projet Nucleo propose des maquettes d'évaluation à bas coût utilisant des solutions MCU et outils de développement de l'industrie (≈ 10 €).





-Power supply
-Programmer (JTAG emulator)

-Target MCU
-Switch and LED
-External ports
-Shields connectors
-Arduino shield connectors

Nucleo-64

Observons les résultats d'une étude de marché réalisée chaque année.

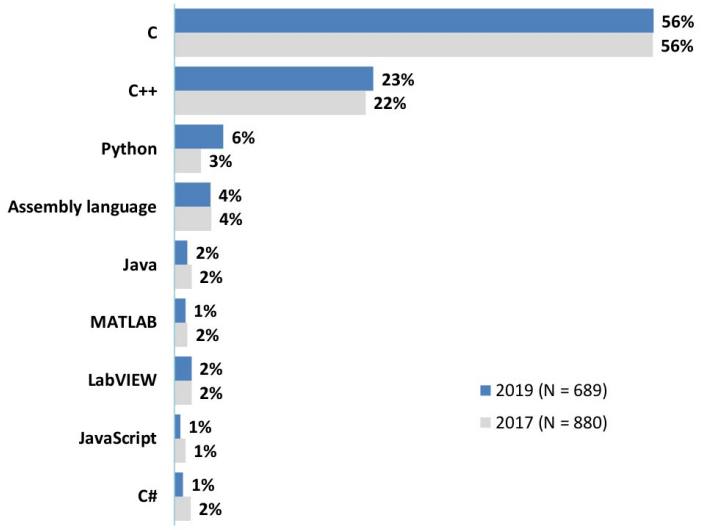
ASPENCORE

2019 Embedded Markets Study

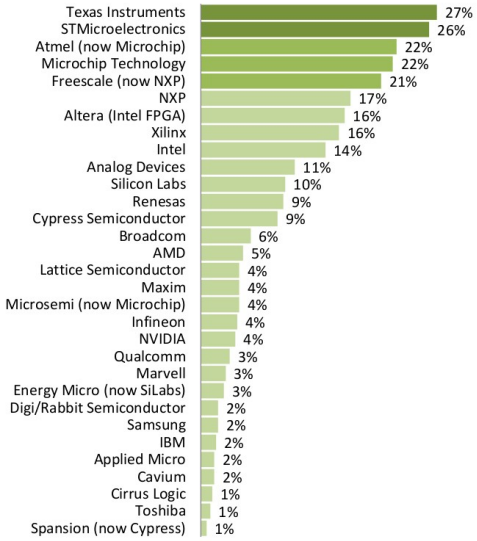
Integrating IoT and Advanced Technology Designs,
Application Development & Processing Environments
March 2019

Presented By: EETimes embedded

My *current* embedded project is programmed mostly in:



Please select the processor vendors you are currently using.

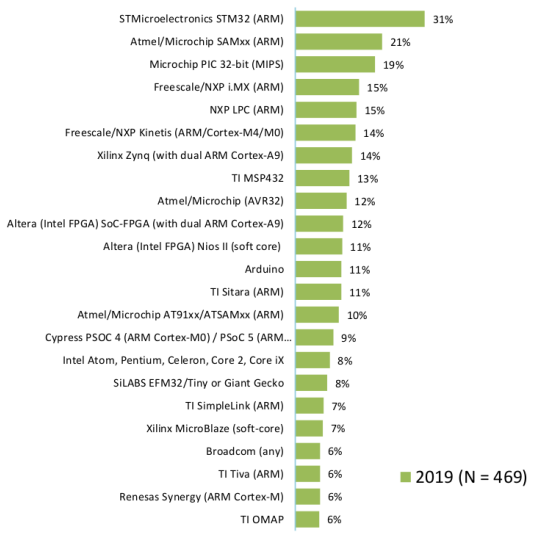


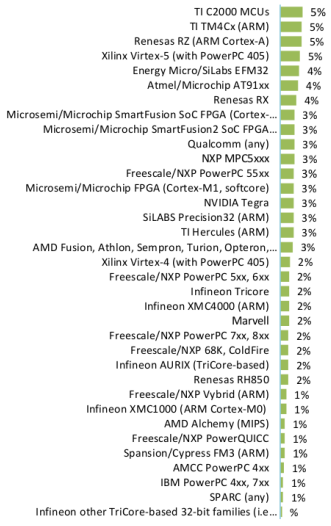
Merged Brands Combined	%
Microchip/Atmel/Microsemi (Net)	40
NXP/Freescale (Net)	28
Intel/Altera (Net)	26
Silicon Labs/Energy (Net)	10
Cypress/Spansion (Net)	9

Top Four Brands by Region:
Americas: TI, Microchip, STMicro, Atmel
EMEA: STMicro, NXP, TI, Atmel
APAC: TI, Atmel, Freescale, STMicro

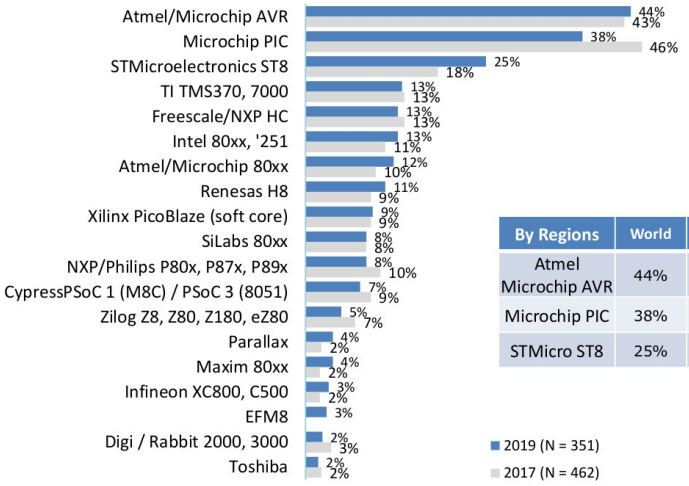
2019 (N = 458)

Which of the following 32-bit chip families would you consider for your next embedded project?





Which of the following 8-bit chip families would you consider for your next embedded project?



By Regions	World	Americas	EMEA	APAC
Atmel Microchip AVR	44%	44%	**52%**	39%
Microchip PIC	38%	41%	**43%**	**23%**
STMicro ST8	25%	22%	**31%**	28%

GPP
GENERAL PURPOSE PROCESSORS

Applications
Architecture
Carte mère
Processeur superscalaire

Les **General Purpose Processors (GPP)** possèdent une architecture CPU complexe leur offrant une grande polyvalence, notamment à l'exécution de code faiblement optimisé.

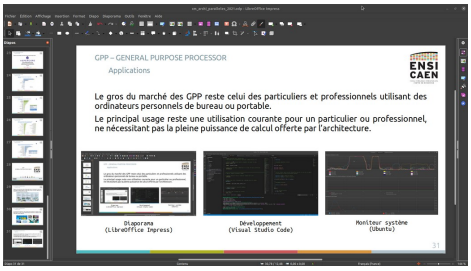
Il s'agit par exemple de programmes de contrôle offrant un code séquentiel avec un grand nombre de tests et d'appels de fonctions. Codes difficiles à accélérer.

```
444         prev = NULL;
445         for (mpnt = oldmm->mmap; mpnt; mpnt = mpnt->vm_next) {
446                 struct file *file;
447
448                 if (mpnt->vm_flags & VM_DONTCOPY) {
449                         vm_stat_account(mm, mpnt->vm_flags, -vma_pages(mpnt));
450                         continue;
451                 }
452                 charge = 0;
453                 if (mpnt->vm_flags & VM_ACCOUNT) {
454                         unsigned long len = vma_pages(mpnt);
455
456                         if (security_vm_enough_memory_mm(oldmm, len)) /* sic */
457                                 goto fail_nomem;
458                         charge = len;
459                 }
460                 tmp = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL);
461                 if (!tmp)
462                         goto fail_nomem;
463                 *tmp = *mpnt;
464                 INIT_LIST_HEAD(&tmp->anon_vma_chain);
465                 retval = vma_dup_policy(mpnt, tmp);
466                 if (retval)
467                         goto fail_nomem_policy;
```

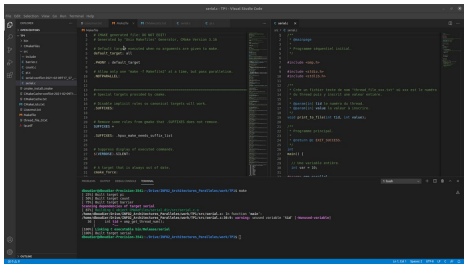
root/kernel/fork.c - www.kernel.org

Le gros du marché des GPP reste celui des particuliers et professionnels utilisant des ordinateurs personnels de bureau ou portable.

Le principal usage reste une utilisation courante pour un particulier ou professionnel, ne nécessitant pas la pleine puissance de calcul offerte par l'architecture.



Diaporama
(LibreOffice Impress)

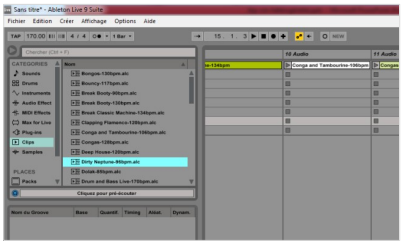


Développement
(Visual Studio Code)

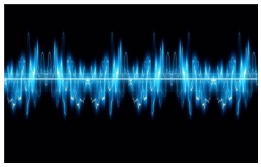
Moniteur système
(Ubuntu)

On peut également citer les applications de traitement du son, de traitement d'image, de traitement du signal, de développement logiciel ou de montages de médias.

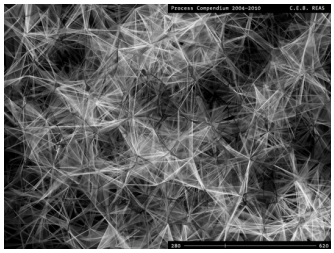
Celles-ci sont plus contraignantes au regard des ressources et exploitent souvent le plein potentiel du matériel.



Montage audio (Ableton)



Traitement du son



Traitement d'image

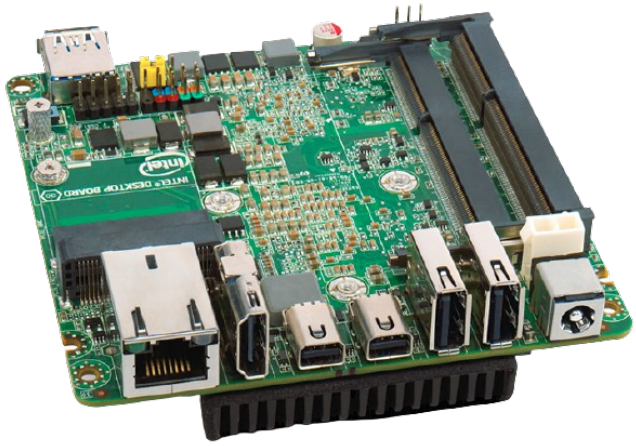
Les applications industrielles sont également un terrain historique des GPP.

Ils sont typiquement rencontrés sur des tâches de contrôle ou des fonctions de calculs spécialisés. Ce marché tend à utiliser des solutions intégrées (AP, SoC, DSP, FPGA).

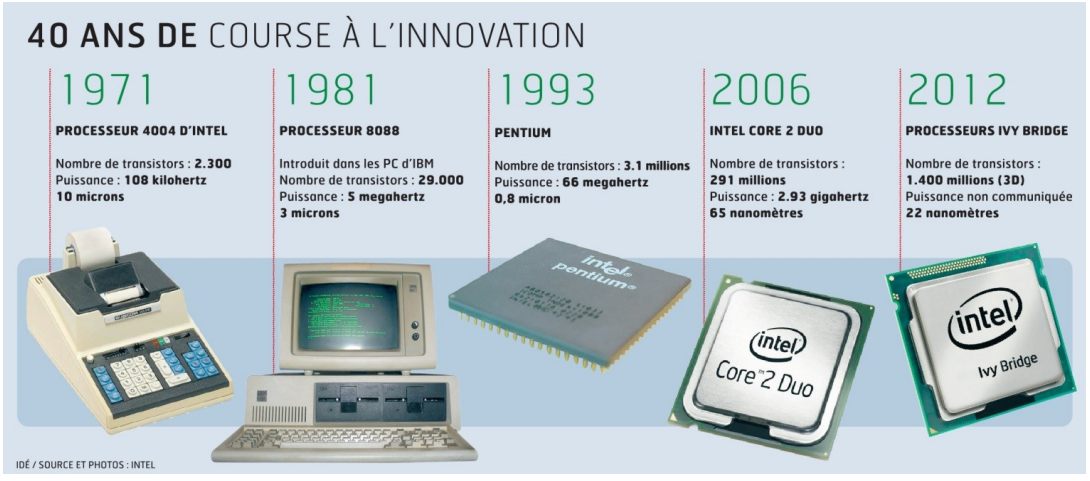
Rafale
(Dassault)

Notons que les GPP peuvent également être exploités par des applications rattachées au domaine des systèmes embarqués.

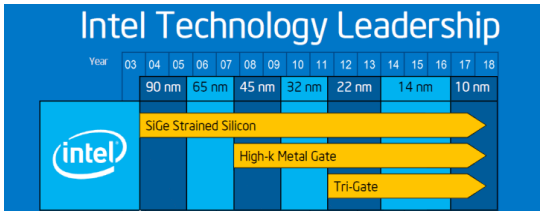
Voici par exemple la carte mère NUC Core i5 de Intel.

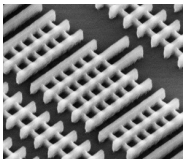


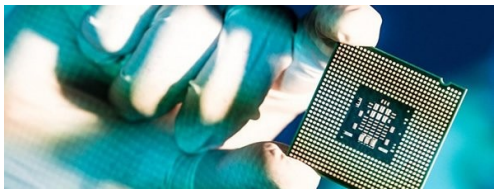
Observons les architectures phares d'Intel, leader actuel et historique du marché des GPP (*General Purpose Processor*) ou MPU (*MicroProcessor Unit*) mais également du marché des semi-conducteurs au sens large.

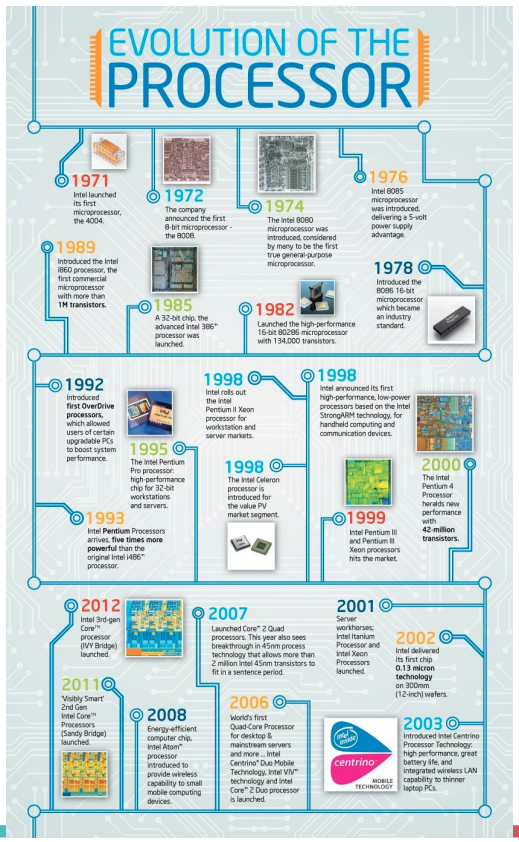


Les architectures GPP phares à notre époque sont les familles Core i3/i5/i7 de Intel. Mais prudence, il existe un grand nombre d'autres architectures et fondeurs de GPP ciblant divers marchés différents.

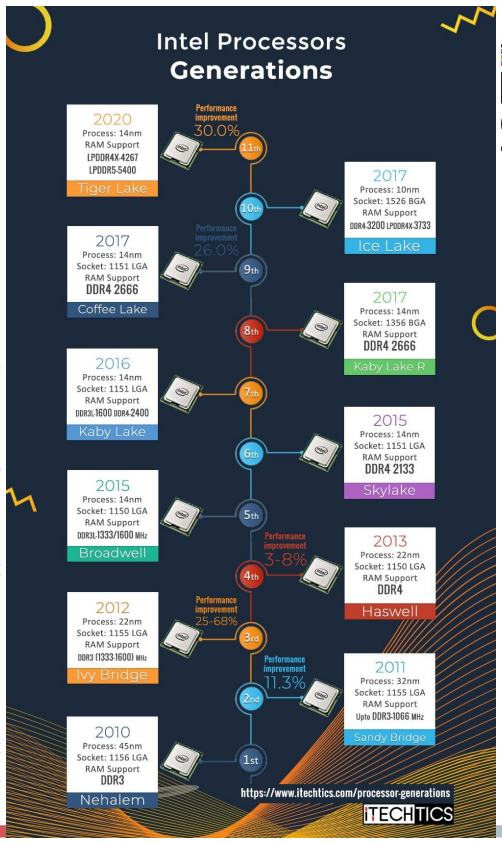




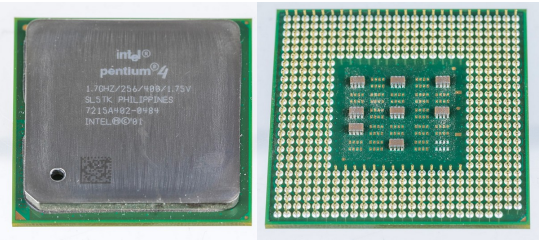




https://javadoc4dummies.blogspot.com/2013/03/intel-processor-evolution.html



https://www.itechtics.com/processor-generations/



4004	(1971)	Processeur 4 bit	
8008	(1972)	Processeur 8 bit	
8086	(1978)	Processeur 16 bit	→ Premier CPU x86 (ISA x86-16)
80386	(1985)	Processeur 32 bit	→ ISA x86-32, rétro-compatible x86-16
Pentium	(1993)	Processeur 32 bit	→ Premier superscalaire commercialisé
Pentium 4	(2000)	Processeur 32 bit	→ 2 cœurs logiques (2 threads)
Core 2 Duo	(2006)	Proc. 32/64 bit	

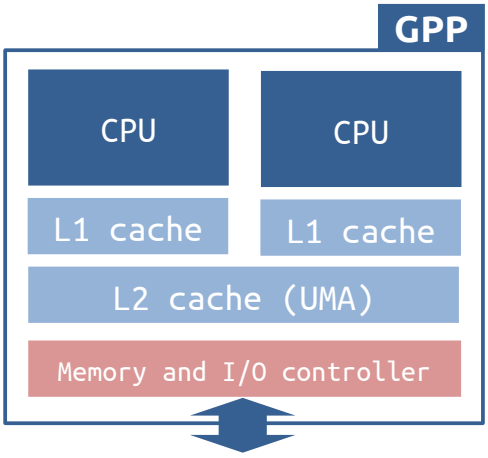
→ Apparition du multi-core chez Intel

→ Naissance de l'ISA x86-64 (calé sur celui d'AMD), rétro-compatible x86-32 et x86-16 !

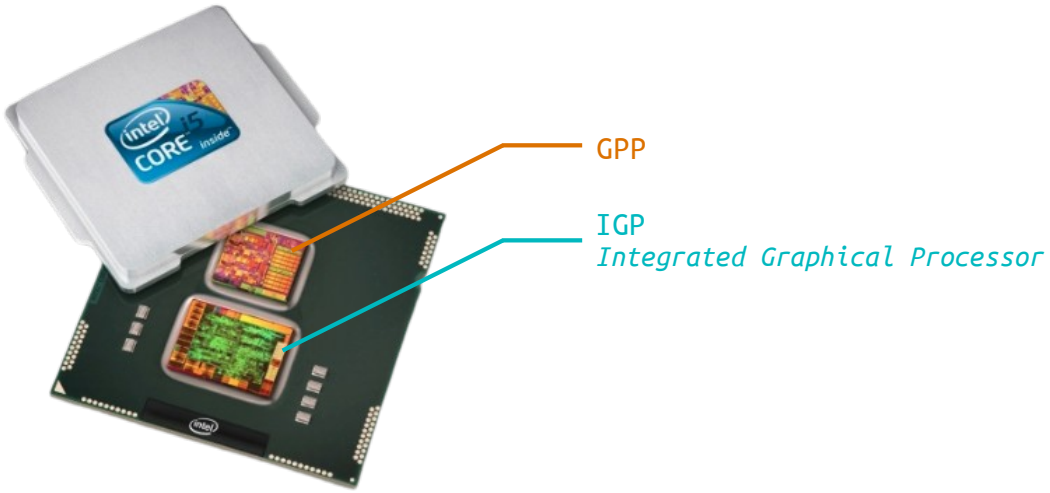
Core (2008) 12 générations se succèdent jusqu'à aujourd'hui (2022)

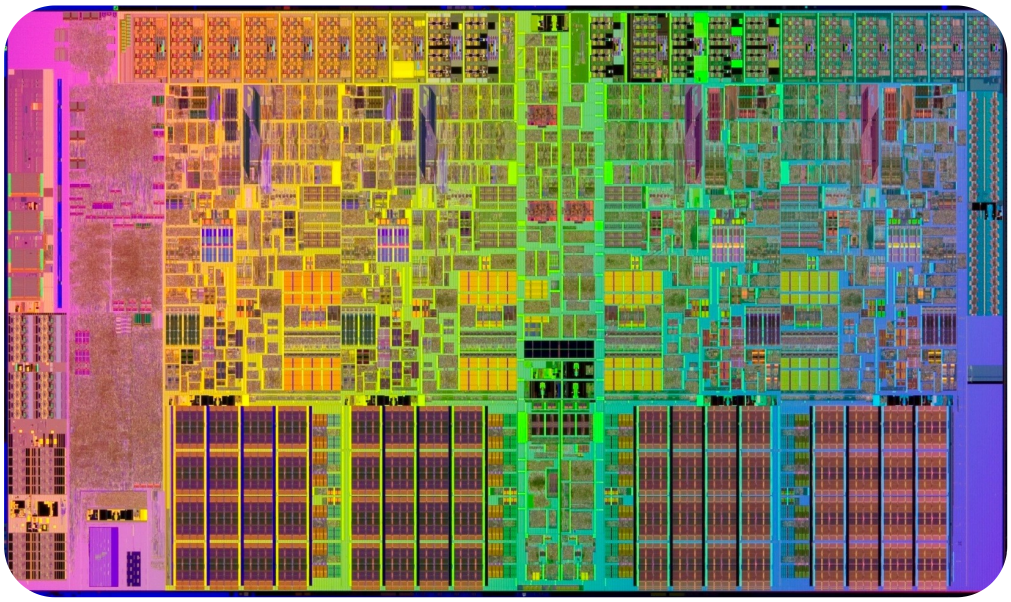
Processeur de traitement nu, dépourvu de mémoire principale.

Il embarque un ou plusieurs CPU (architecture homogène) mariés avec leurs caches, possède un modèle mémoire uniforme (UMA) et embarque un contrôleur d'interfaces.



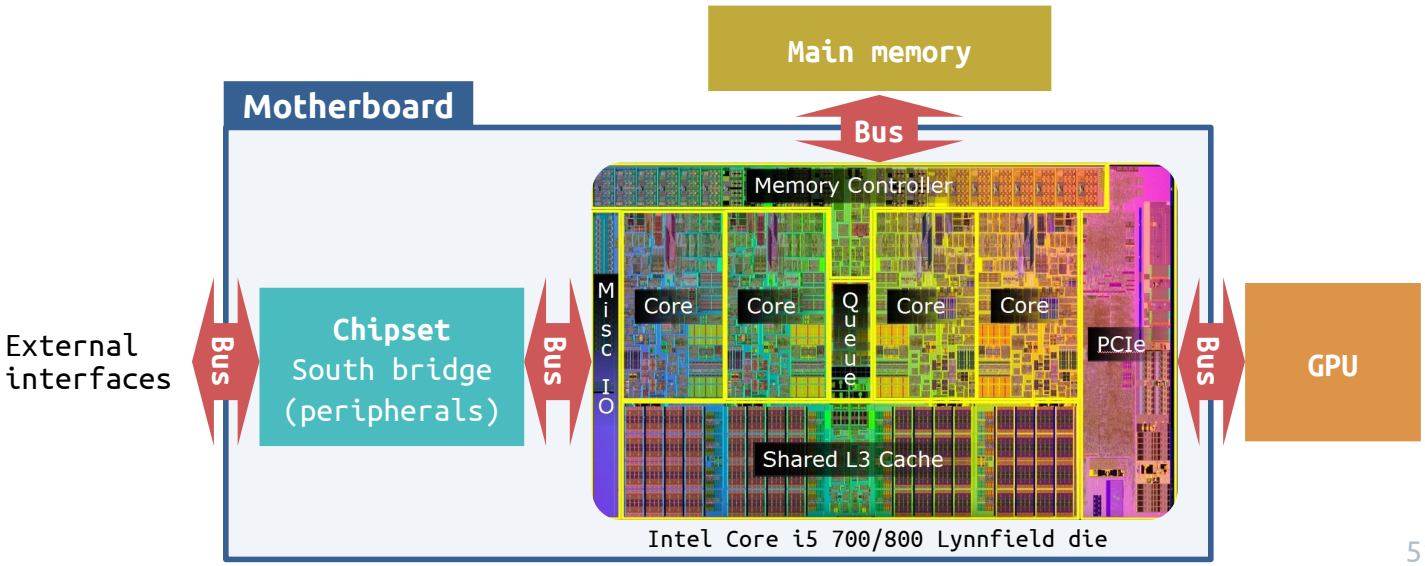
Exemple de la famille Core i5 de Intel.





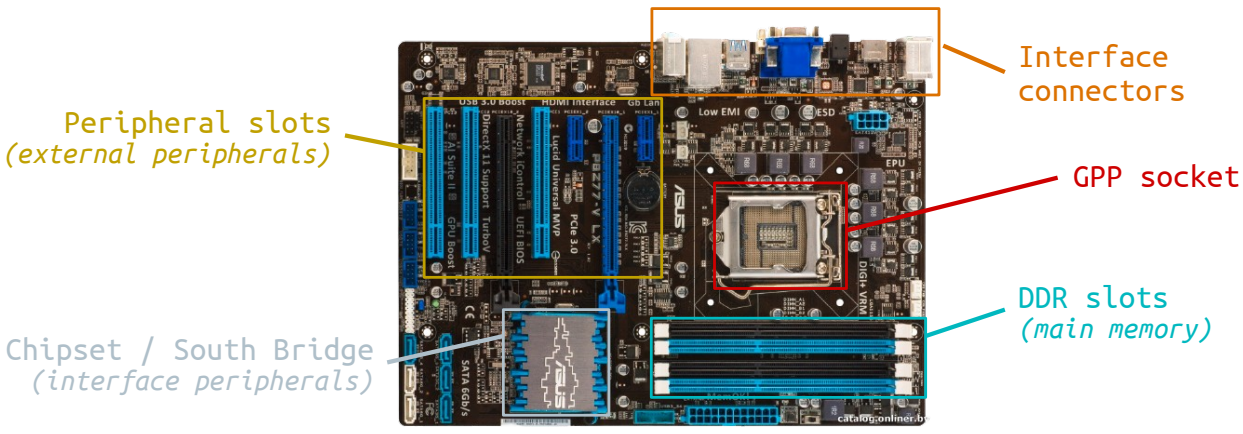
Intel Core i5 700/800 Lynnfield die

Intégration dans le système (carte mère)



Un GPP doit forcément être porté sur une carte mère avec mémoire principale et périphériques d'interfaces externes déportés.

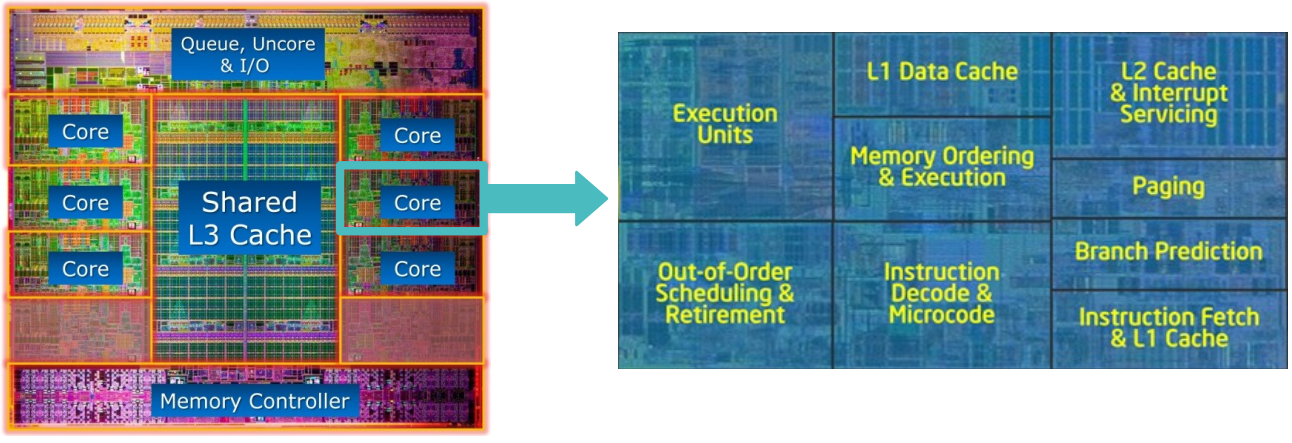
Exemple de carte mère ASUS, n°2 du marché mondial en 2016.



Les GPP possèdent un CPU dit superscalaire. Les processeurs possédant ce type de pipeline CPU se caractérisent le plus souvent par le déploiement des mécanismes d'accélération matériels suivants :

- Étage d'exécution *Out Of Order* : Exécution des instructions dans le désordre. Ordonnanceur matériel gérant les dépendances fonctionnelles et sur les données, étages de renommage des registres (résultats intermédiaires) et de ré-ordonnancement
- Étage de prédiction au branchement
- Étage d'exécution *RISC-like*, même si l'ISA est CISC

Die d'un CPU de la génération Sandy Bridge de Intel, illustré pour un Core i7.



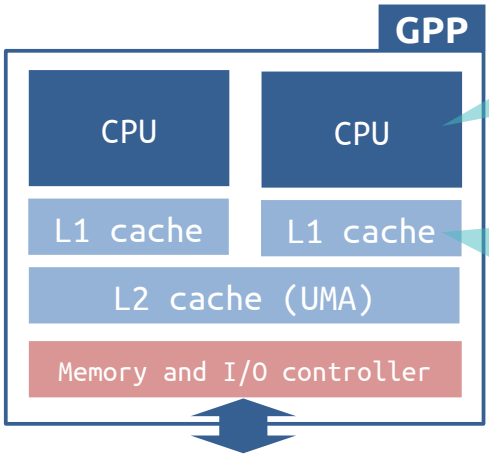
Intel Core i7

Sandy Bridge CPU/Core

Attention, cette grande polyvalence et complexité matérielle se paye par un manque de déterminisme voire de performance à l'exécution sur des traitements algorithmiques spécifiques.

Les GPP offrent un ratio performance de calcul ramené au coût et au Watt peu intéressant.

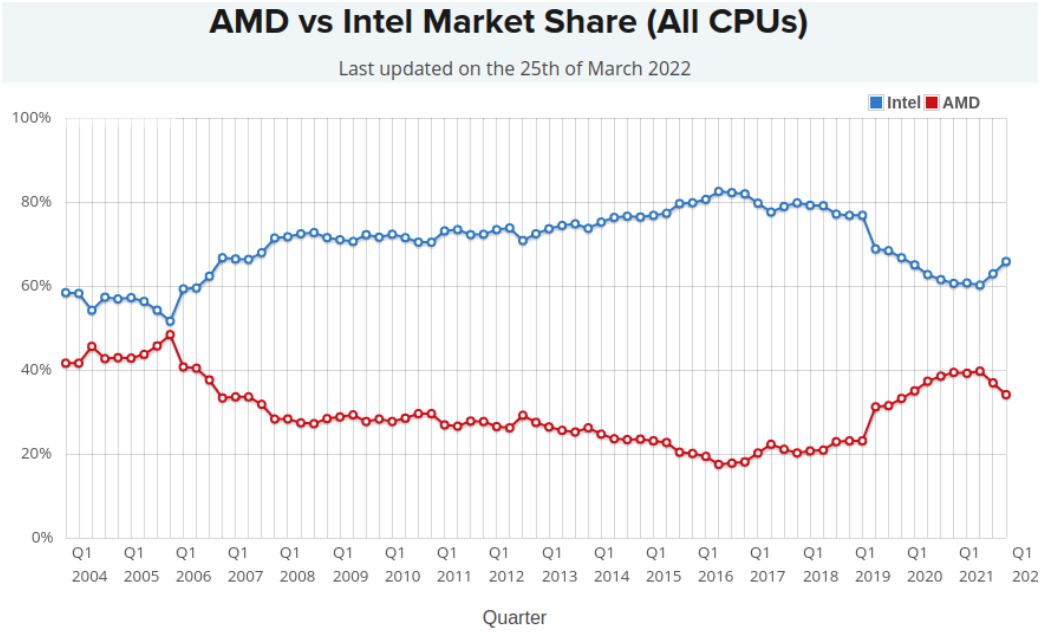
Ils sont pensés pour porter un OS (*Operating System*) évolué et exécuter du code applicatif. Prenons les exemples des applications de traitement du son, traitement d'image, traitement vidéo, traitement d'antenne … pour lesquels ils ne sont pas spécialisés.



CPU superscalaire
- exécution Out Of Order
- prédiction de branchement
- non déterministe
- mauvais ratio (puissance calcul) / (Watt x Coût)

Mémoire
- Modèle mémoire uniforme (UMA)
- Cache processeur
  - → Technologies de transfert rapides
  - → Copies d'informations depuis la mémoire principale (DATA ou INST.)
  - → Intelligence déportée dans les contrôleurs de caches (LRU)
  - → Non déterministe

Parts de marché : Intel vs. AMD



https://www.cpubenchmark.net/market_share.html

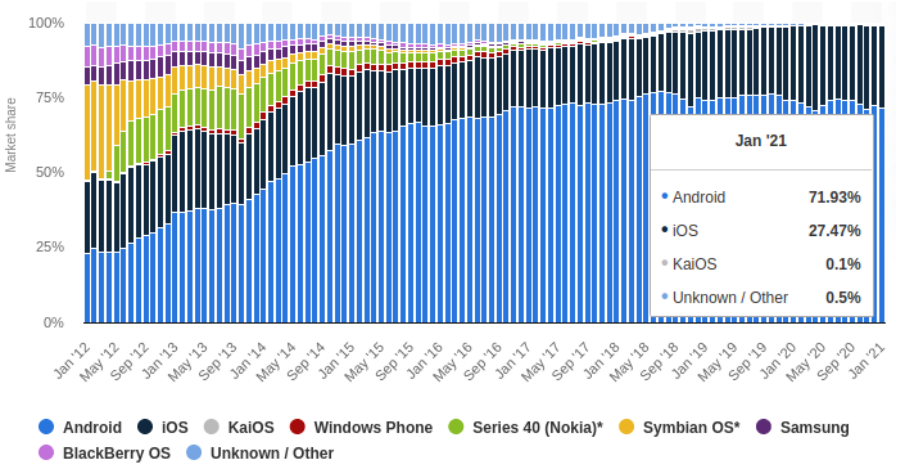
AP
APPLICATION PROCESSOR

Applications
Architecture
Solution Qualcomm
Solution ARM

Le marché des AP (*Application Processor*), processeurs riches en fonctionnalités et services matériels de type SoC (*System on Chip*), reste un marché récent qui a vu son envole avec celui des terminaux mobiles (smartphone, phablette et tablette).

Le principal marché des AP en terme de parts reste donc celui des terminaux mobiles.

Ce marché voit une utilisation écrasante du système d'exploitation Android en 2016, système basé sur un noyau Linux.



Source:
Statista 2021

Néanmoins les processeurs applications sont très rencontrés dans les systèmes embarqués au sens large, tous domaines confondus : *consumer*, défense, transport …

Ces systèmes embarquent généralement un OS et une interface graphique.

Freebox Revolution

Télévision 4K X94C Sony

Tablette Cook
(fait à Caen par EOLANE)

Dans la majorité des cas, ces processeurs sont exploités par des systèmes évolués.

Sur ce marché les systèmes GNU/Linux (très souvent customisés) règnent en maîtres.

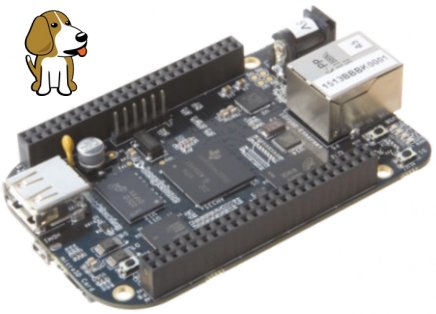
Exemple de plateforme industrielle durcie EOLANE (Français n°2 Européen) travaillant autour de SoC/AP iMX6 proposé par Freescale sur système GNU/Linux.

Voici les deux plateformes non-durcies à bas coût qui dominent le marché :

les projets Raspberry Pi et Beaglebone (SoC AM335x TI).

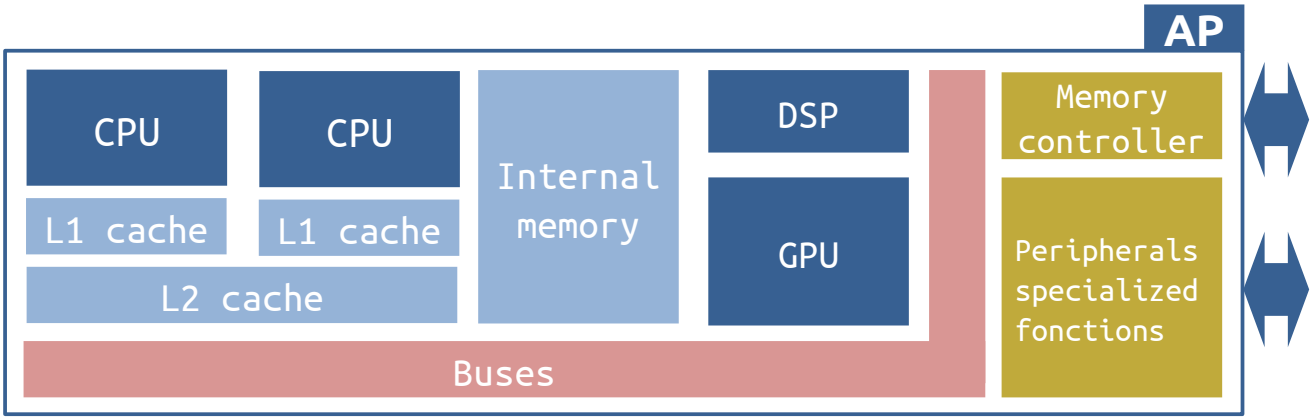
Ces solutions sont également basées sur des systèmes GNU/Linux.

Elles sont très rencontrées durant les phases de prototypage ou en milieu universitaire, mais ne peuvent être industrialisées. Néanmoins des versions durcies existent.



Les AP sont des systèmes numériques complets intégrés dans une puce (architecture hétérogène).

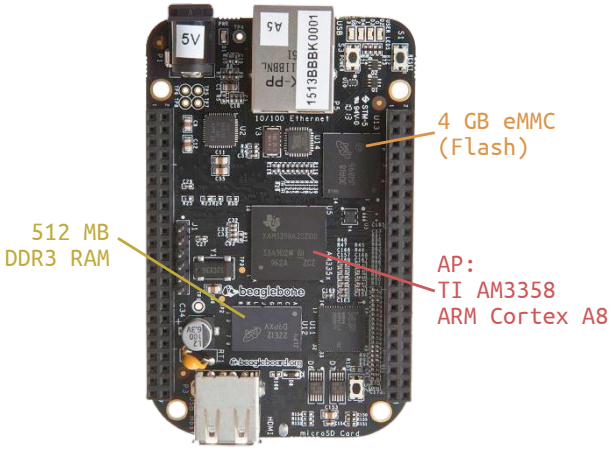
Néanmoins, la mémoire principale doit être ajoutée en externe.



Un processeur application embarque toujours un voire plusieurs CPU généralistes superscalaires. Ils sont dédiés à l'exécution du ou des systèmes d'exploitation évolués (virtualisés ou réels) ainsi que des applicatifs.

Un AP contient également une voire plusieurs fonctions spécialisées de calcul (GPU, DSP, crypto …), un jeu de périphériques évolués complet et une mémoire interne ne permettant pas d'accueillir le système (*bootloader*).

Par conséquent, une mémoire principale (DDR volatile) et une mémoire non-volatile de stockage de masse (MMC, eMMC, SDCard …) externes doivent lui être ajoutées.



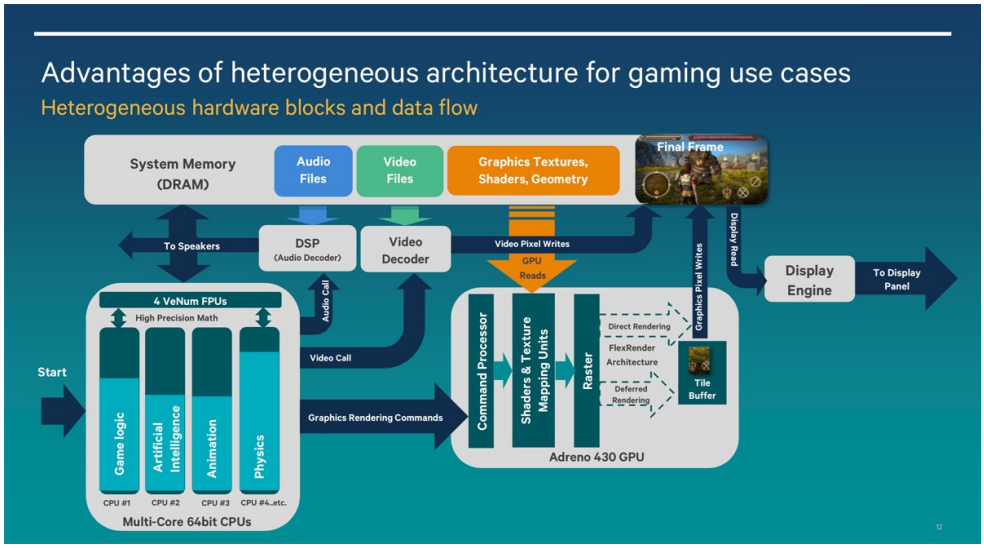
https://beagleboard.org/black

Contrairement aux MCU embarquant tous les services matériels sur la puce afin de contrôler un système (*on chip*), les AP exigent un coût unitaire non négligeable et restent dépréciés pour les applications à faible coût et fort volume.

Ils sont alors utilisés si il y a nécessité d'une interface et/ou de connectivités évoluées dans l'application.

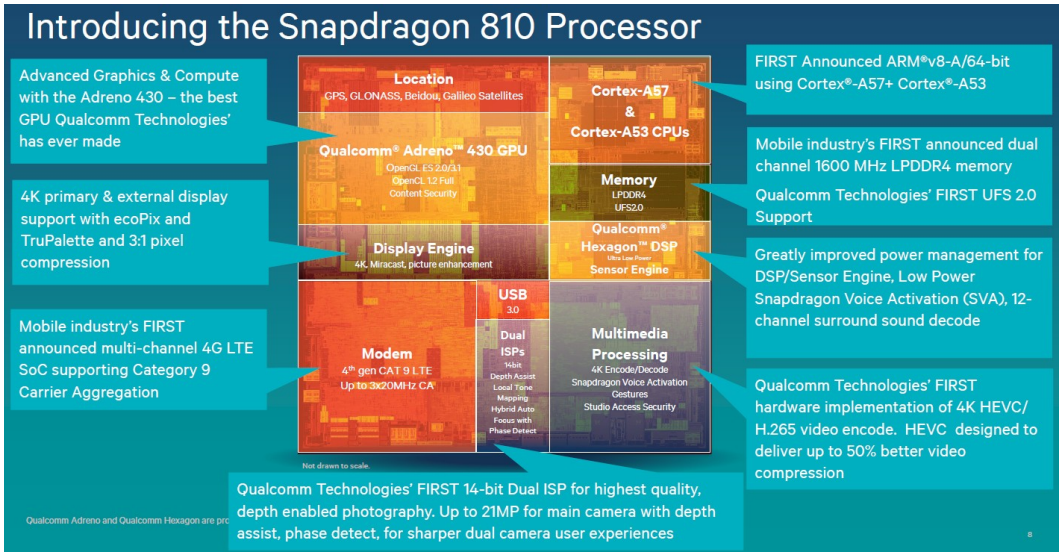
GPP	AP	MCU
+ RAM + hard drive + mother board	+ RAM + MMC + PCB	+ PCB
$$$	$$	$

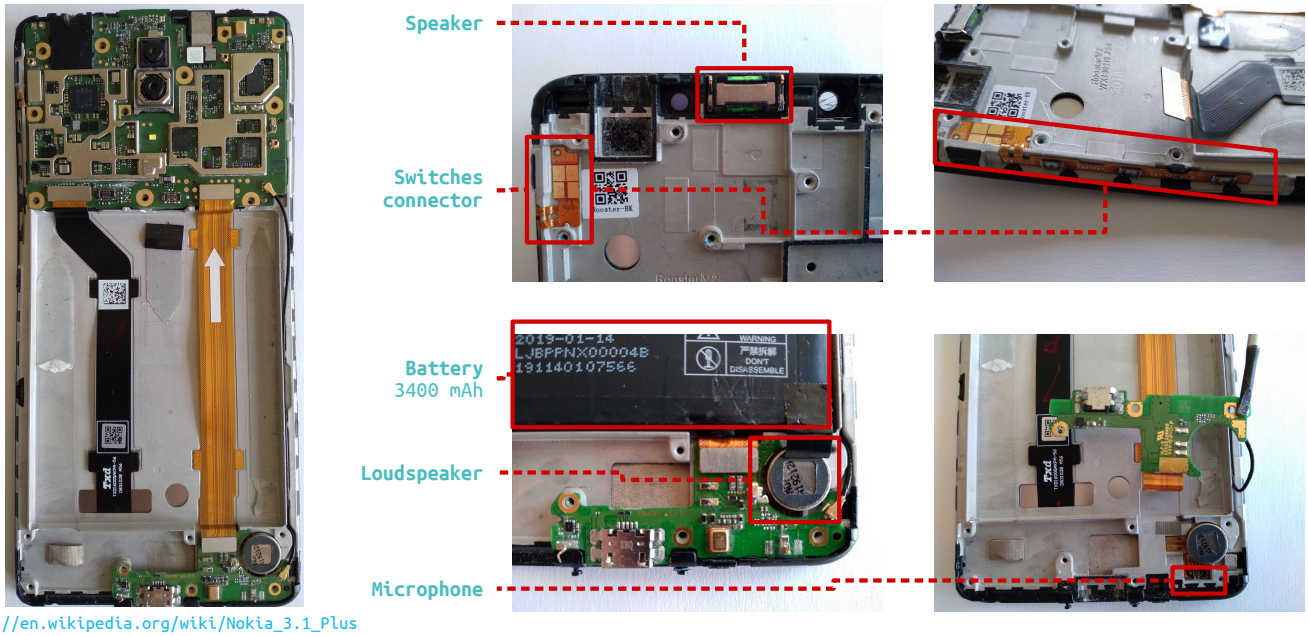
Observons l'intérêt d'une architecture hétérogène pour une application aux jeux vidéos



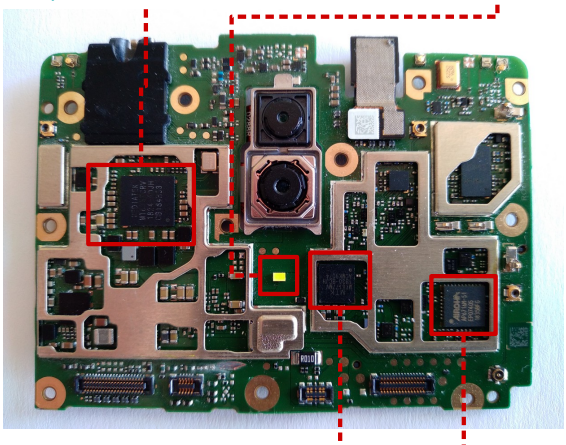
Le leader du marché en terme de part de marché est Qualcomm, grâce à sa famille Snapdragon dédiée au marché des terminaux mobiles.

Fonctions matérielles de l'architecture interne de la famille Qualcomm Snapdragon 810

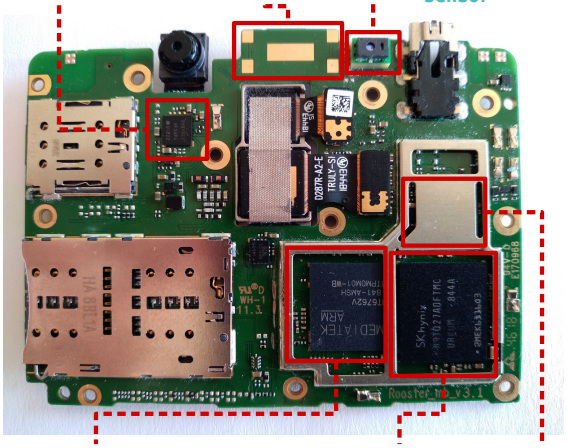




https://en.wikipedia.org/wiki/Nokia_3.1_Plus



Antennas in the back case



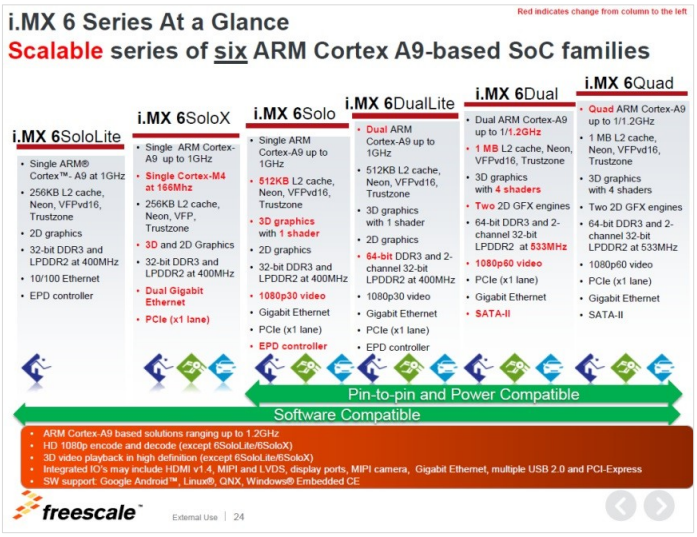
MEDIATEK MT6762V (Helio P22)
Qualcomm SDM439 Snapdragon (2018)
ARM v8-A (64-bit), Cortex-A53
8 cores, 2 GHz, 12 nm
2-core GPU, DSP

SK hynix
H9TQ27ADFTMC
32 GB Flash
Nand eMMC

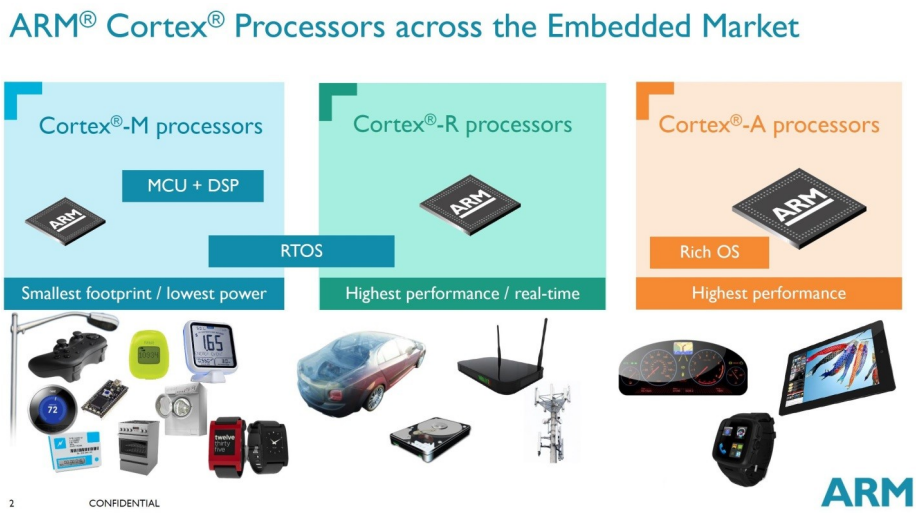
RAM ?
2 GB
LPDDR3
(Low-Power)

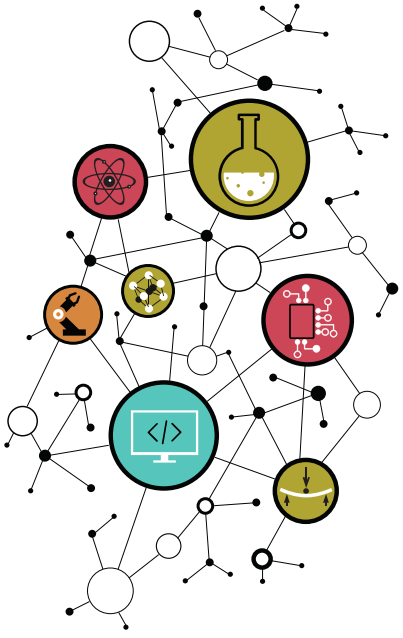
Les deux leaders du marché hors terminaux mobiles sont Texas Instruments et Freescale, deux fondeurs offrant de larges communautés d'utilisateurs.

Observons la famille i.MX6 de Freescale :



Hors marché des terminaux mobiles, sur le marché de l'embarqué les architectures Cortex-A de ARM sont également reines. Le « A » signifie *Application*.



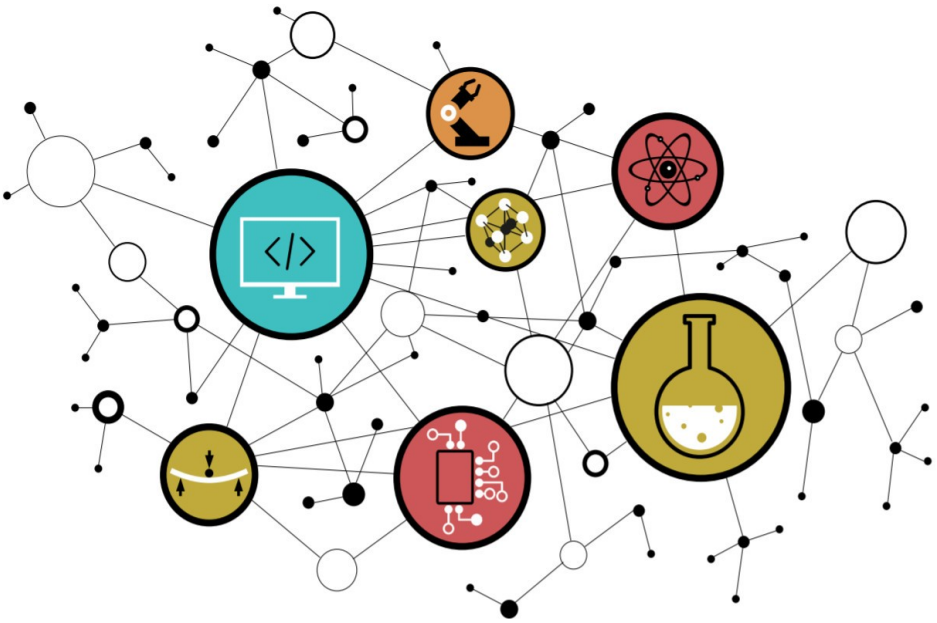


Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)

COMPILATION ET ÉDITION DES LIENS



Chapitre 3

Chaîne de Compilation

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

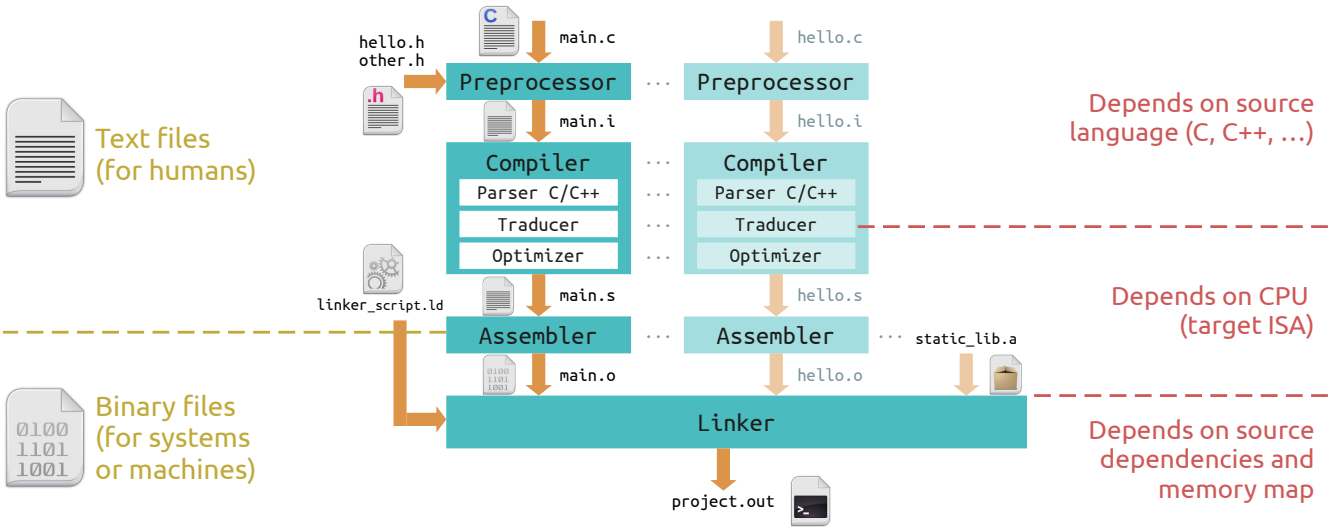
2021-2022

OBJECTIFS

Points-clés

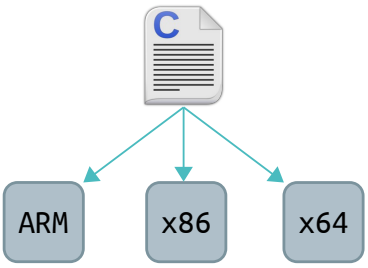
- Connaître les différents étages d'une chaîne de compilation C
- Comprendre le processus de compilation (rôle des étapes)
- Comprendre le processus d'édition des liens
- Analyser les fichiers issus des différents étages de la toolchain
  gcc, as, ld, objdump, readelf, strip, ar, …
- Savoir créer ou debugger un projet C/C++/asm en cours de construction

Schéma à connaître et comprendre



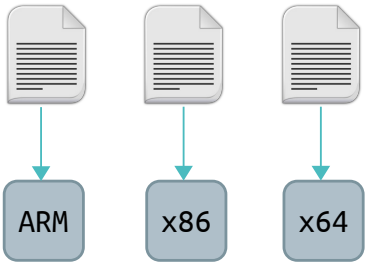
Le langage C est qualifié de **portable**.

Cela signifie qu'un programme initialement rédigé en C peut être exécuté sur n'importe quelle architecture de processeur.



Cela s'oppose aux langages d'assemblage, qui eux sont **spécifiques à une architecture**.

Ainsi, une nouvelle version du programme doit être écrite pour chaque architecture sur laquelle on veut exécuter le programme.



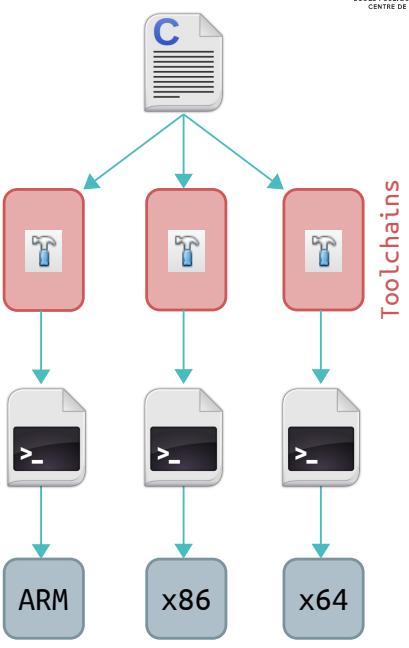
Toutefois, la diapo précédente est imprécise.

"un programme ***initialement*** rédigé en C peut être exécuté sur n'importe quelle architecture de processeur."

En effet les fichiers C ne sont pas exécutables. Ils doivent être compilés afin de créer un exécutable.

Ce processus est réalisé par la **chaîne de compilation, ou *toolchain***.

En partant d'un seul fichier C, il faut autant de *toolchains* que de processeurs sur lesquels exécuter le programme.



Ce chapitre est consacré à la chaîne de compilation.

Plusieurs *toolchains* existent. Vous avez peut-être entendu parler de GCC ou MinGW, qui sont pour les architectures x86 et x64. Vous avez aussi utilisé XC8, la *toolchain* Microchip pour MCU PIC18.

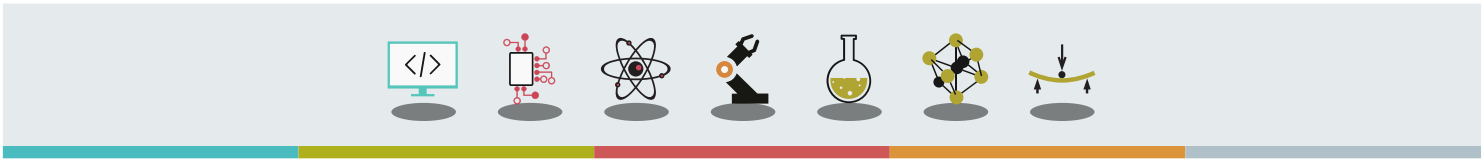
La *toolchain* GCC (GNU Compiler Collection, https://gcc.gnu.org) sera utilisée à titre d'exemple dans ce chapitre.

Le processus reste le même dans n'importe quelle toolchain C. En revanche les formats et extensions des fichiers intermédiaires ne sont pas standardisés, ils peuvent donc varier d'une *toolchain* à une autre, ou d'une plateforme matérielle à une autre.

LA TOOLCHAIN GCC

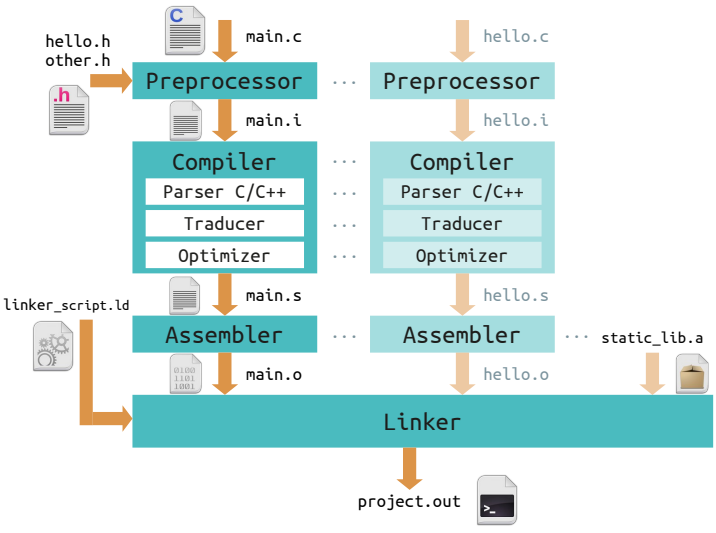
Du fichier source C vers le fichier exécutable

Les pages de cette partie sont à connaître par cœur.



LA TOOLCHAIN GCC

Vue d'ensemble

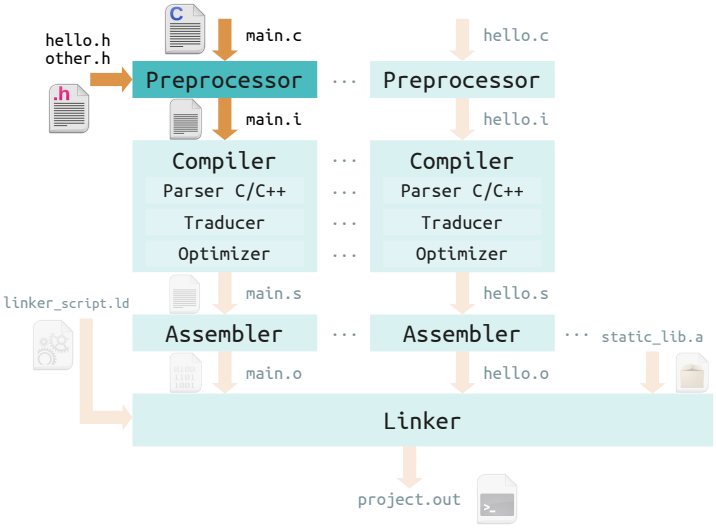


La toolchain est composée de 4 étages, pouvant produire un fichier de sortie.

Chaque fichier C passe indépendamment à travers les trois premiers étages, pour produire autant de fichiers objets ou *object files* (*.o).

Le dernier étage récupère tous les fichiers objets pour construire le fichier exécutable final.

C'est tout ce processus qui se déroule quand vous appuyez sur le bouton 'build' ou quand vous appelez la commande 'gcc'.



L'*étage de preprocessing* s'occupe de:

Inclusion de code (`#include <hello.h>`)

Compilation conditionnelle

```
#if CONDITION
#ifndef HELLO_H
```

Toute autre directive pré-processeur (débute par '#')

Opérateurs `#pragma` (depuis la norme C99)

Le fichier de sortie se nomme *preprocessed file*. C'est un fichier texte, générique, d'extension `.i` .

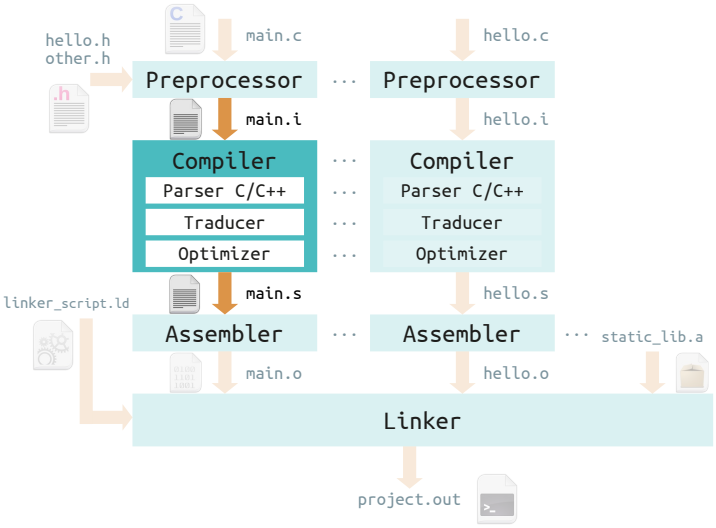
Exemple de pré-traitement avec `gcc -E test.c > test.i`

test.c

```
#define QUARANTE        40
#define QUARANTE_UN     41
int a = QUARANTE + QUARANTE_UN;

#define CHAINE_DE_CARACTERES "du texte"
char * c = CHAINE_DE_CARACTERES;

#define DEBUG
#ifdef DEBUG
printf("Version debug \n");
#endif
#ifndef DEBUG
printf("Version non-debug \n");
#endif
```

test.i

```
int a = 40 + 41;

char * c = "du texte";

printf("Version debug \n");
```



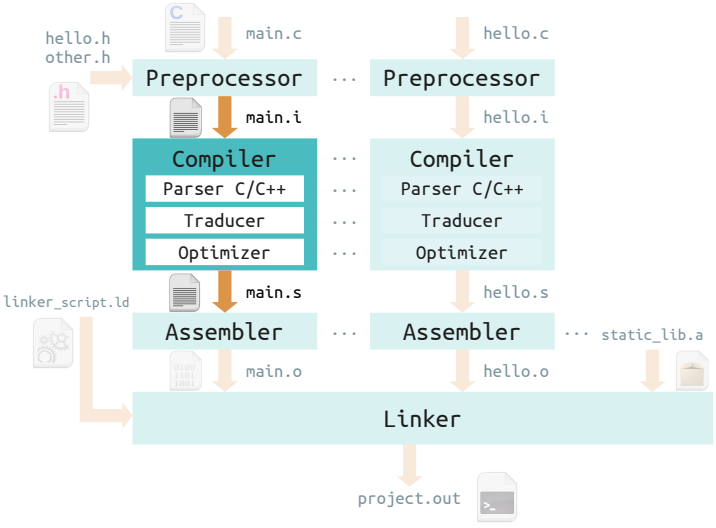
Le fichier généré passe ensuite à travers l'**étage de compilation**, qui comporte :

Parser : vérifie le respect des règles du langage.

Traducteur : fabrique un nouveau programme dans la langue d'assemblage du processeur cible.

Optimiseur : améliore le code grâce à sa connaissance poussée de l'architecture cible.

Le ficher de sortie est au format langage d'assemblage (.a, .asm). Il s'agit toujours d'un fichier texte, mais il est maintenant spécifique à l'architecture cible.



Tâche du *parser*

Analyse lexicale

Les mots-clés sont-ils corrects ?

ex : "`float`" vs "~~`flaot`~~"

Analyse syntaxique (ou *parsing*)

Les mots forment-ils des phrases correctes ?

ex : "`float tab[4];`" vs "~~`float[4] tab;`~~"

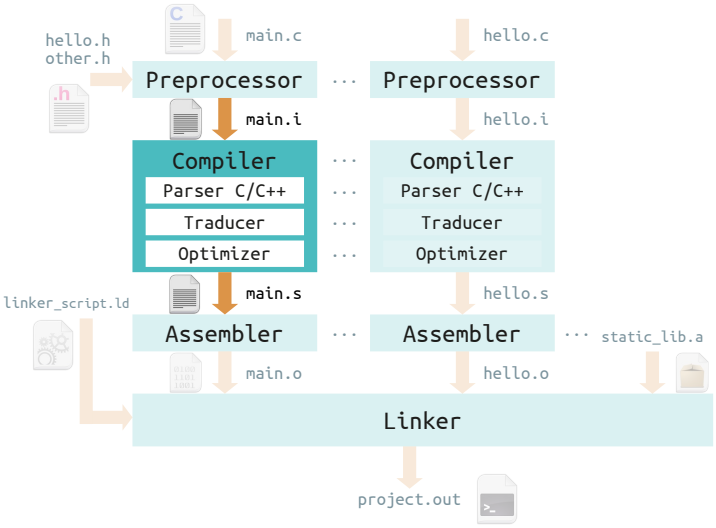
Analyse sémantique

Les phrases font-elles sens ?

ex : "`a = b / 0;`".

Cohérence entre déclaration et définition ?

L'analyseur construit la table des symboles.



Traducer (ou *code generator*)

Spécifique à l'architecture cible.

Si l'architecture cible est différente de l'architecture sur laquelle la toolchain tourne, on appelle cela de la cross-compilation.

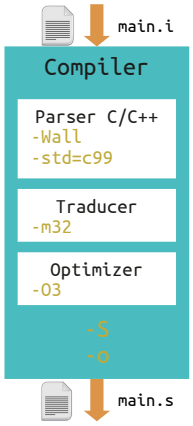
Optimizer

Phase optionnelle. Spécifique à la cible.

Modifie le code dans une forme plus rapide (ou plus compacte).

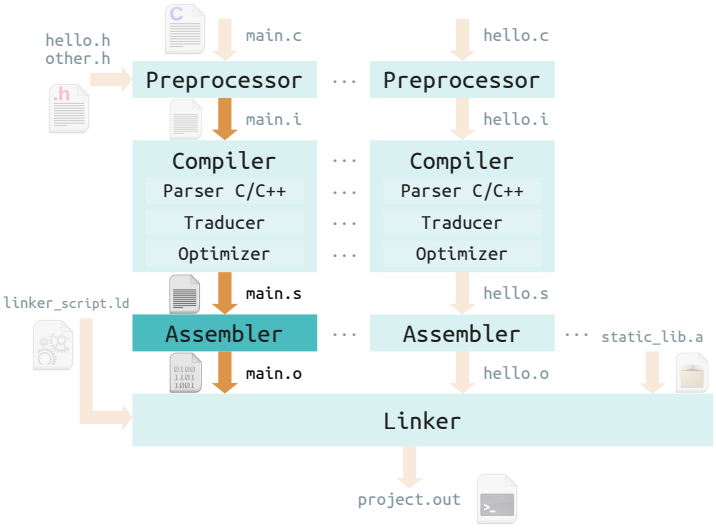
inline expansion, dead code elimination, constant folding, loop transformation, parallelization

Ex. : `gcc -S -Wall -std=c99 -m32 -O3 ./build/main.i -o ./build/main.o`



`./build/main.i`	Spécifie le fichier d'entrée à compiler
`-Wall`	Afficher tous les warnings (*Warning All*)
`-std=c99`	Norme du langage à respecter
`-m32`	Traduction en assembleur pour cible 32-bit
`-O3`	Niveau 3 d'optimisation
`-S`	S'arrêter à l'étage d'assemblage
`-o ./build/main.o`	Nom du fichier de sortie

Étage assembleur



L'assembleur traduit le fichier en langage d'assemblage vers un fichier objet binaire.

Le fichier objet est relogeable et non exécutable.

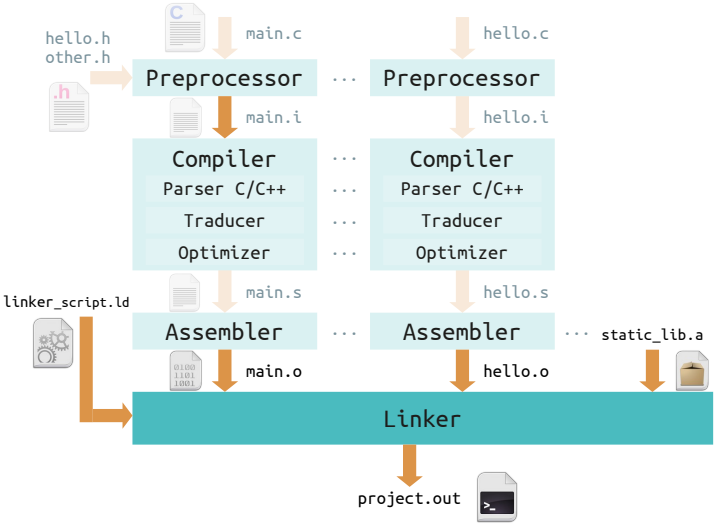
Il utilise des références symboliques : les variables et fonctions externes ont un nom, mais leur adresse est encore inconnue pour le moment.

Le fichier objet est spécifique à l'architecture cible, mais ne dépend pas de son modèle mémoire.

Le fichier de sortie est désormais binaire : il est illisible pour l'humain, compris seulement par la cible.

Note : « langage d'assemblage » != « étage assembleur »

Étage d'édition des liens



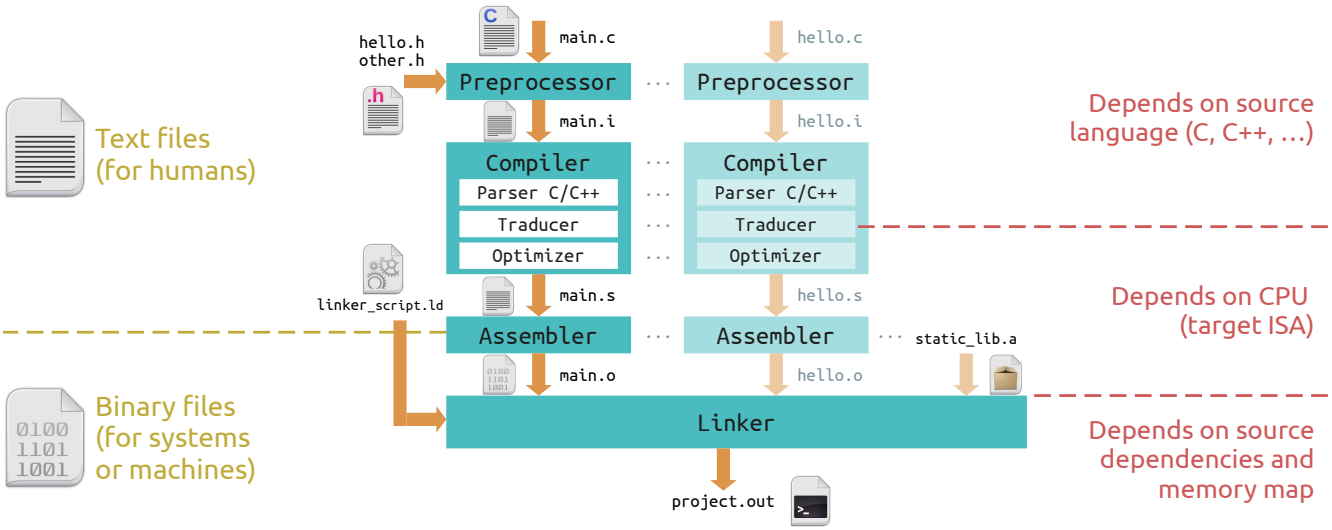
L'éditeur des liens (*linker*) fusionne tous les fichiers objet en un seul exécutable.

Plus précisément, le *linker* lie les fichiers ensemble : *object files*, *static libraries* (ensemble de fonctions pré-compilées), et *dynamic libraries* (chargées à l'exécution du programme).

C'est la résolution de la table des symboles et la résolution des liens avec le modèle mémoire de la cible.

Le fichier généré est habituellement un exécutable, mais ça peut être un fichier relogeable (*static library*).

Le fichier généré dépend de l'architecture et du modèle mémoire de l'architecture cible.



Plusieurs choix disponibles pour les utilisateurs pro-console !

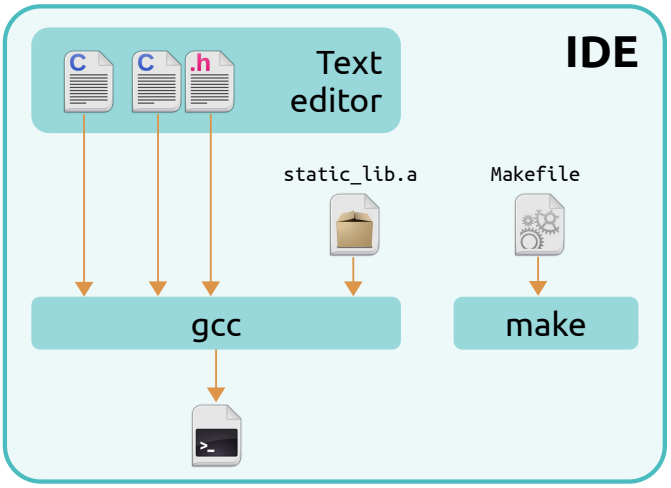
Si seulement quelques sources, compiler avec la commande "`gcc`".
Taper "`man gcc`" pour le manuel, ou https://linux.die.net/man/1/gcc.
Si le projet devient plus gros, n'utiliser que gcc devient vite pénible.

On automatise le processus de compilation avec la commande "`make`".
Toutes les règles de compilation sont écrites dans un fichier `Makefile`.

On peut même automatiser l'automatisation du processus de compulation avec CMake !

Mais la plupart des développeurs utilisent un IDE :

Integrated Development Environment ou Environnement de Développement Intégré.



Un IDE est pratique pour les projets de grande envergure.

Éditeur de texte avec coloration syntaxique, auto-complétion, …

Processus de compilation automatisé.

Plusieurs outils d'aide, le plus important étant le *debugger*.

FICHIERS BINAIRES

Analyse du format ELF
(*Executable and Linkable Format*)

Les fichiers sources (`*.c`, `*.cpp`, `*.h`, …), *preprocessed* (`*.i`), et compilés (`*.a`) sont des fichiers textes : ils sont lisibles par l'humain mais pas par la machine.

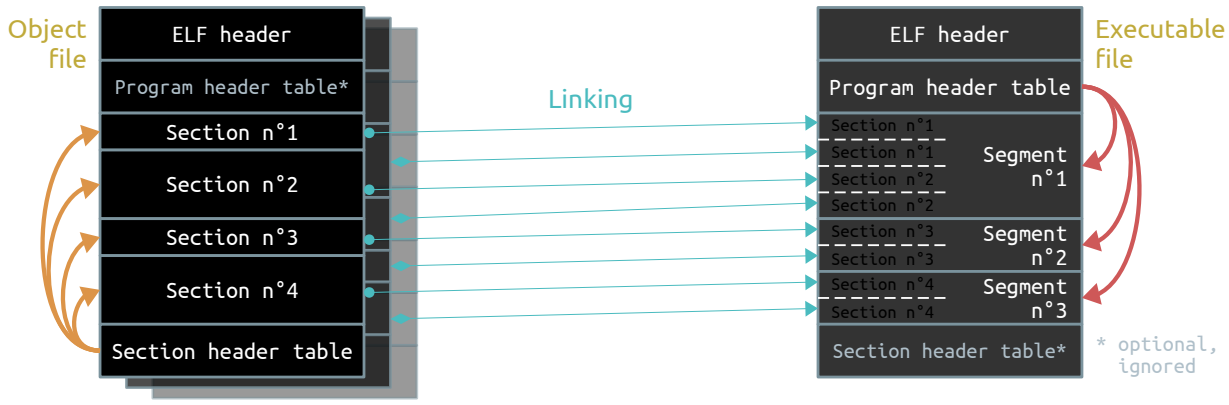
Les fichiers objets (`*.o`), bibliothèques statiques (`*.a`), exécutable (`*`, `*.out`) sont quant à eux binaires : impossible de les lires avec un éditeur de texte, seule la machine peut les comprendre.

Tous ces fichiers binaires sont au format ELF (*Executable and Linkable Format*).

Format utilisé pour l'enregistrement de code compilé : object (`*.o`), archive (`*.a`), shared object (`*.so`), kernel object (`*.ko`), core dumps, executable (`*`, `*.out`).

Extrêmement répandu sur systèmes UNIX-like (GNU/Linux, FreeBSD, OpenBSD, Solaris, Android, …) et sur d'autres plateformes (PlayStation 1 à 5, Dreamcast, Nintendo 64 à Wii U, PowerPC, …), MCU Atmel et Texas Instruments.

Un fichier ELF contient toujours un en-tête (*ELF header*). Le reste dépend du type de fichier.

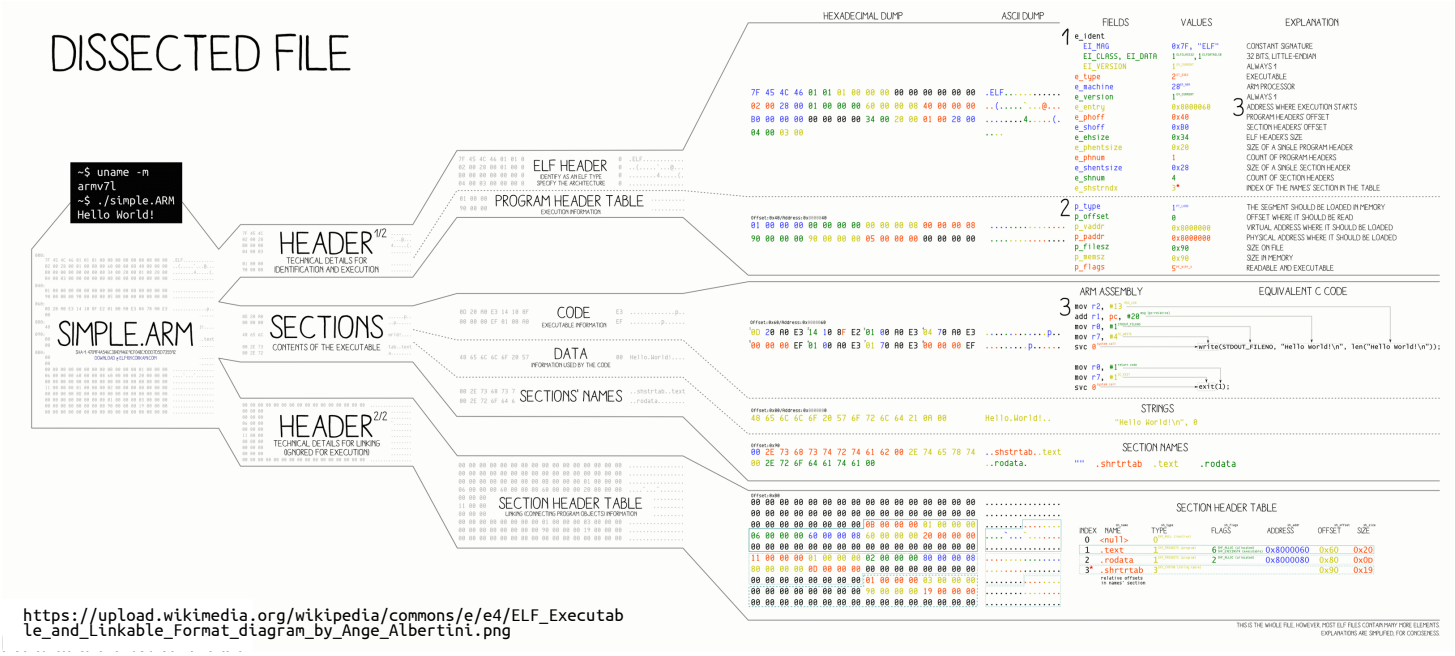


Un fichier objet est décomposé en **sections**, qui sont ensuite listées dans la *Section header table*.

Les sections contiennent des informations pour l'édition des liens et la relocalisation.

Un fichier exécutable contient une *Program header table*, qui décrit les **segments** qui suivent.

Un segment contient les informations utiles à l'exécution du fichier.



https://upload.wikimedia.org/wikipedia/commons/e/e4/ELF_Executable_and_Linkable_Format_diagram_by_Ange_Albertini.png

Un fichier binaire ne peut être lu qu'à l'aide d'utilitaires dédiés. Les commandes `readelf` (issue de `binutils`) et `objdump` permettent d'interpréter les informations des fichiers ELF.

Lisons l'en-tête (*ELF header*) de différents fichiers ELF avec la commande "`readelf -h`".

```
dboudier:toolchain$ readelf -h build/obj/hello.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into f
  Start of section headers:          320 (bytes into
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         9
  Section header string table index: 8
```

```
dboudier:~$ readelf -h /lib/i386-linux-gnu/libc.so.6
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x1f0a0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          2020040 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         13
  Size of section headers:           40 (bytes)
  Number of section headers:         68
  Section header string table index: 67
```

```
dboudier:toolchain$ readelf -h build/bin/hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
ddress:                              0x8049000
am headers:                          52 (bytes into file)
on headers:                          4388 (bytes into file)
                                     0x0
eader:                               52 (bytes)
m headers:                           32 (bytes)
ram headers:                         3
n headers:                           40 (bytes)
ion headers:                         6
 string table index:                 5
```

Lisons maintenant la table des sections d'un fichier objet avec "`readelf -S`".

```c
/**
* @file objdump-minimal.c
*/

#include <stdio.h>

char my_data[] = "Bonjour le monde\n";

/**
* program entry point
*/
int main(void)
{
    char* my_rodata = "Hello World\n";

    printf("%s%s", my_data, my_rodata);

    return 0;
}
```

```
dboudier:toolchain$ gcc objdump-minimal.c -c
dboudier:toolchain$ readelf -S -W objdump-minimal.o
There are 14 section headers, starting at offset 0x3a8:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00003d 00  AX  0   0  1
  [ 2] .rela.text        RELA            0000000000000000 0002b8 000060 18   I 11   1  8
  [ 3] .data             PROGBITS        0000000000000000 000080 000012 00  WA  0   0 16
  [ 4] .bss              NOBITS          0000000000000000 000092 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        0000000000000000 000092 000012 00   A  0   0  1
  [ 6] .comment          PROGBITS        0000000000000000 0000a4 00002b 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        0000000000000000 0000cf 000000 00      0   0  1
  [ 8] .note.gnu.property NOTE            0000000000000000 0000d0 000020 00   A  0   0  8
  [ 9] .eh_frame         PROGBITS        0000000000000000 0000f0 000038 00   A  0   0  8
  [10] .rela.eh_frame    RELA            0000000000000000 000318 000018 18   I 11   9  8
  [11] .symtab           SYMTAB          0000000000000000 000128 000150 18     12  10  8
  [12] .strtab           STRTAB          0000000000000000 000278 00003a 00      0   0  1
  [13] .shstrtab         STRTAB          0000000000000000 000330 000074 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)

dboudier:toolchain$ readelf -l objdump-minimal.o

There are no program headers in this file.
```

Note : on remarque avec "`readelf -l`" qu'il n'y a pas de *program header* dans un fichier objet.

Parmi les nombreuses sections d'un fichier ELF, nous nous concentrons sur celles-ci.

La section `.text` contient le code binaire du programme.

Les sections liées à l'allocation statique des variables :

`.bss` : contient les variables globales et variables statiques non-initialisées

`.data` : contient les variables globales et variables statiques initialisées

`.rodata` : contient les constantes (variables en lecture seule)

L'allocation statique est l'allocation mémoire à la compilation (*compile-time*).

Les variables qui seront utilisées tout au long du programme sont donc stockées dans les fichiers compilés (fichiers objets, puis exécutable).

Petite parenthèse : on verra en CM et TP les deux autres types d'allocation rencontrés.

- L'allocation automatique, utilisée pour les variables locales.
- L'allocation dynamique, utilisée pour de grandes quantités de données (`malloc`, `free`)

Ces deux types d'allocation s'effectuent pendant l'exécution du programme (*run-time*), par opposition à l'allocation statique qui se fait pendant la compilation (*compile-time*).

Analysons le contenu des sections qui nous intéressent avec la commande "`objdump -s`".

On remarque que l'adresse des informations (instructions ou données) est relative au début de section.

```c
/**
* @file objdump-minimal.c
*/

#include <stdio.h>

char my_data[] = "Bonjour le monde\n";

/**
* program entry point
*/
int main(void)
{
    char* my_rodata = "Hello World\n";

    printf("%s%s", my_data, my_rodata);

    return 0;
}
```

```
dboudier:toolchain$ objdump -s objdump-minimal.o

objdump-minimal.o:     file format elf64-x86-64

Contents of section .text
 0000 f30f1efa 554889e5 4883ec10 488d0500  ....UH..H...H...
 0010 00000048 8945f848 8b45f848 89c2488d  ...H.E.H.E.H..H.
 0020 35000000 00488d3d 00000000 b8000000  5....H.=........
 0030 00e80000 0000b800 000000c9 c3        .............
Contents of section .data
 0000 426f6e6a 6f757220 6c65206d 6f6e6465  Bonjour le monde
 0010 0a00                                 ..
Contents of section .rodata
 0000 48656c6c 6f20576f 726c640a 00257325  Hello World..%s%
 0010 7300                                 s.
```

Code binaire (instructions)

Allocation statique (données)

Adresse relative (format hexadécimal)

Contenu des sections (représentation hexadécimale)

Contenu des sections (représentation ASCII)

Fichiers objet : section `.text` et désassemblage

Il est possible de retrouver le programme en langage d'assemblage avec l'option *disambly* :

```
dboudier:toolchain$ objdump -s objdump-mini

objdump-minimal.o:     file format elf64-x8

Contents of section .text:
 0000 f30f1efa 554889e5 4883ec10 488d0500
 0010 00000048 8945f848 8b45f848 89c2488d
 0020 35000000 00488d3d 00000000 b8000000
 0030 00e80000 0000b800 000000c9 c3
```

```
dboudier:toolchain$ objdump -d objdump-minimal.o

objdump-minimal.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 10             sub    $0x10,%rsp
   c:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 13 <main+0x13>
  13:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  17:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  1b:   48 89 c2                mov    %rax,%rdx
  1e:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 25 <main+0x25>
  25:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 2c <main+0x2c>
  2c:   b8 00 00 00 00          mov    $0x0,%eax
  31:   e8 00 00 00 00          callq  36 <main+0x36>
  36:   b8 00 00 00 00          mov    $0x0,%eax
  3b:   c9                      leaveq
  3c:   c3                      retq
```


Fichiers objet : table des symboles

En lisant la table des sections, on peut noter la présence de la section `.symtab` : il s'agit de la table des symboles.

Les fichiers étant traités individuellement pendant les premières phases de la compilation, les données (variables, fonctions, …) sont employés uniquement par référence symbolique (par leur nom).

C'est l'éditeur des liens (*linker*) qui, ayant tous les fichiers nécessaires à sa portée, remplacera ces symboles par leur adresse respective.

```c
/**
* @file objdump-minimal.c
*/

#include <stdio.h>

char my_data[] = "Bonjour le monde\n";

/**
* program entry point
*/
int main(void)
{
    char* my_rodata = "Hello World\n";

    printf("%s%s", my_data, my_rodata);

    return 0;
}
```

```
dboudier:toolchain$ readelf -s objdump-minimal.o

Symbol table '.symtab' contains 14 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS objdump-minimal.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    9
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    10: 0000000000000000    18 OBJECT  GLOBAL DEFAULT    3 my_data
    11: 0000000000000000    61 FUNC    GLOBAL DEFAULT    1 main
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

Adresse dans la section

Symboles
(noms de fonction, de variable, ...)

Lire de droite à gauche :

my_data, dans la section 3 (.data), est un *global object* (variable globale) de 18 o. Son adresse dans sa section est 0000.

main, dans la section 1 (.text) est une fonction de 61 octets. Son adresse dans sa section est 0000.

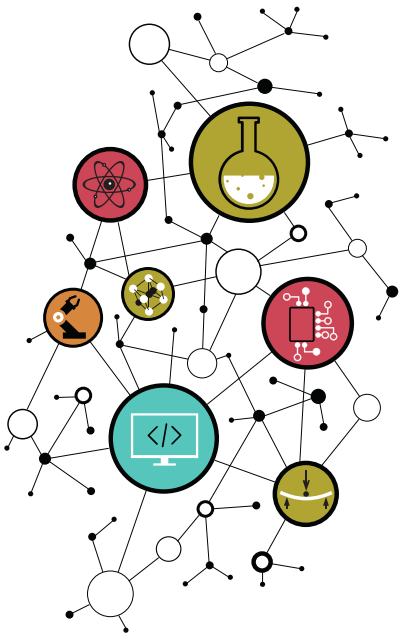
printf, existe mais sa localisation est inconnue.

Passons finalement au désassemblage de l'exécutable.

On remarque que la fonction `printf` est appelée via son symbole et non son adresse. En effet, cette fonction sera appelée à l'exécution puisqu'elle est compilée dans une bibliothèque dynamique (*shared object* *.so)

Pour cela, la section `.plt` (*procedure linkage table*) effectue une redirection à l'exécution (*run-time*) vers l'adresse absolue de la procédure cible (`printf` dans notre cas).

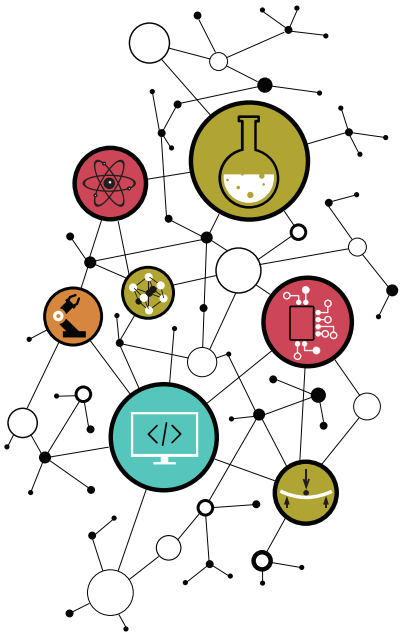
```
0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       48 83 ec 10             sub    $0x10,%rsp
    1155:       48 8d 05 a8 0e 00 00    lea    0xea8(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
    115c:       48 89 45 f8             mov    %rax,-0x8(%rbp)
    1160:       48 8b 45 f8             mov    -0x8(%rbp),%rax
    1164:       48 89 c2                mov    %rax,%rdx
    1167:       48 8d 35 a2 2e 00 00    lea    0x2ea2(%rip),%rsi        # 4010 <my_data>
    116e:       48 8d 3d 9c 0e 00 00    lea    0xe9c(%rip),%rdi        # 2011 <_IO_stdin_used+0x11>
    1175:       b8 00 00 00 00          mov    $0x0,%eax
    117a:       e8 d1 fe ff ff          callq  1050 <printf@plt>
    117f:       b8 00 00 00 00          mov    $0x0,%eax
    1184:       c9                      leaveq
    1185:       c3                      retq
    1186:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
    118d:       00 00 00
```

Description approfondie des fichiers ELF (PDF)

Dissection d'un fichier ELF (png)

CONTACT

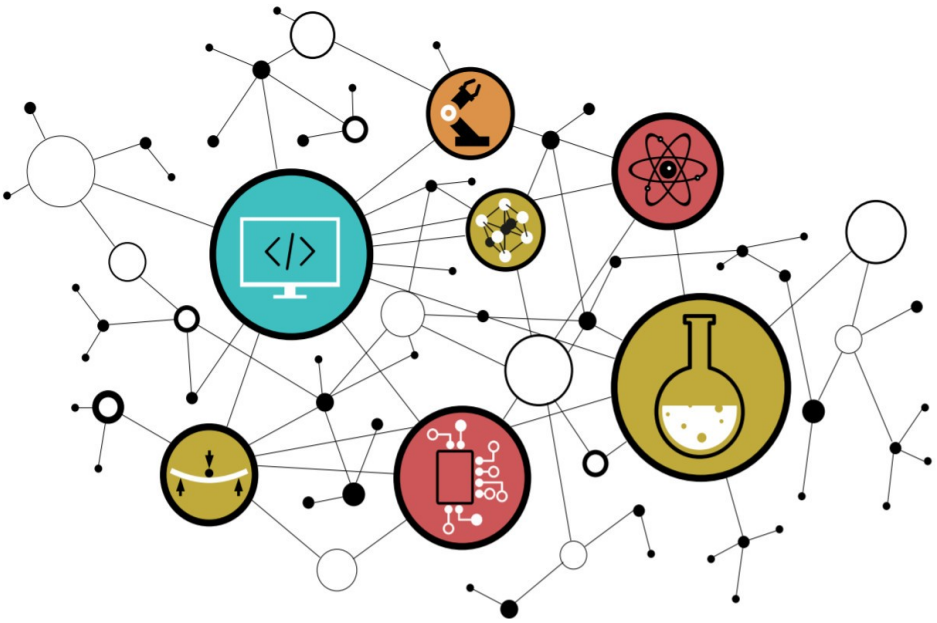


Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

Avec la participation de

- Hugo Descoubes (PRAG ENSICAEN)

LANGAGE D'ASSEMBLAGE



ENSICAEN
Computer Science

LANGAGE D'ASSEMBLAGE

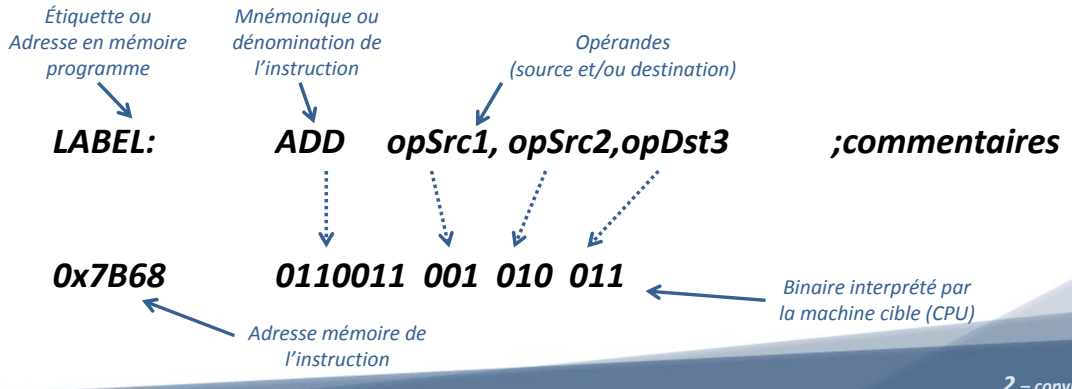
Architecture et Technologie des Ordinateurs

Hugo Descoubes - Juin 2013

ENSICAEN
Computer Science
ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Un langage d'assemblage ou assembleur ou ASM est un langage de programmation bas niveau représentant, sous forme lisible pour un être humain, le code binaire exécutable par un processeur (ou code machine). *Prenons l'exemple d'une instruction assembleur élémentaire raccrochée à aucune architecture connue :*



2 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Hormis label et commentaires, en général à tout champ d'une instruction assembleur correspond un champ dans le code binaire équivalent. Ce code binaire ne peut être compris et interprété que par le CPU cible.

LABEL: ***ADD*** ***opSrc1, opSrc2,opDst3*** ***;commentaires***

0110011 001 010 011

N'est utilisé que par les instructions de branchement

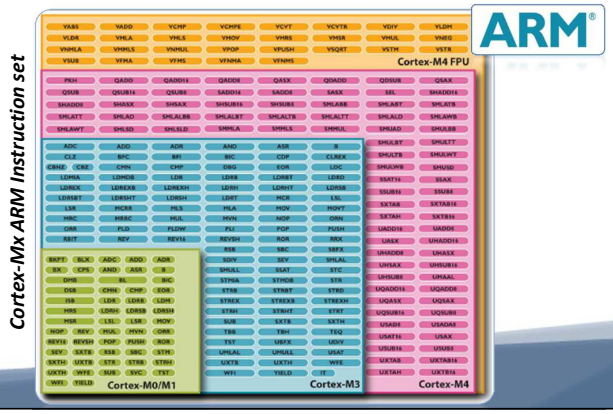
Opcode ou Code opératoire

3 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

L'assembleur est probablement le langage de programmation le moins universel au monde*. Il existe autant de langage d'assemblage que de familles de CPU. Prenons l'exemple des jeux d'instructions Cortex-M de ARM. La société Anglaise ARM propose à elle seule 3 familles de CPU, cortex-M, -R, -A possédant chacune des sous familles. Ne regardons que la famille cortex-M :*



4 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Observons les principaux acteurs dans le domaine des CPU's. Chaque fondeur présenté ci-dessous propose une voire plusieurs architectures de CPU qui lui sont propres et possédant donc les jeux d'instructions associés (CPU server et mainframe non présentés) :

- ***GPP CPU architectures :*** *Intel (IA-32 et Intel 64), AMD (x86 et AMD64), IBM (PowerPC), Renesas (RX CPU), Zilog (Z80), Motorola (6800 et 68000) …*
- ***Embedded CPU architectures (MCU, DSP, SoC) :*** *ARM (Cortex –M – R -A), MIPS (Rx000), Intel (Atom, 8051), Renesas, Texas Instrument (MSPxxx, C2xxx, C5xxx, C6xxx), Microchip (PICxx) , Atmel (AVR), Apple/IBM/Freescale (PowerPC) …*

5 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Tout CPU est capable de décoder puis d'exécuter un jeu d'instruction qui lui est propre (ou instruction set ou ISA ou Instruction Set Architecture*). Dans tous les cas, ces instructions peuvent être classées en grandes familles :*

- ***Calcul et comparaison :*** *opérations arithmétiques et logiques (en C : +, -, *, /, &, |, ! …) et opérations de comparaison, (en C : >=, <=, !=, == …). Les formats entiers courts seront toujours supportés nativement. En fonction de l'architecture du CPU, les formats entiers long (16bits et plus) voire flottants peuvent l'être également.*
- ***Management de données :*** *déplacement de données dans l'architecture matérielle (CPU vers CPU, CPU vers mémoire ou mémoire vers CPU)*

6 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- ***Contrôle programme :*** *saut en mémoire programme (saut dans le code). Par exemple en langage C : if, else if, else, switch, for, while, do while, appels de procédure. Nous pouvons rendre ces sauts conditionnels à l'aide d'opérations arithmétiques et logiques ou de comparaisons.*

Certaines architectures, comme les architectures compatibles x86-64 (Intel et AMD), possèdent des familles spécialisées :

- ***String manipulation :*** *manipulation au niveau assembleur de chaînes de caractères.*

- ***Divers :*** *arithmétique lourde (sinus, cosinus…), opérations vectorielles (produit vectoriel, produit scalaire…) …*

7 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- *Jeu d'instruction RISC 8051*
- *Jeu d'instruction CISC 8086*

Les jeux d'instructions et CPU associés peuvent être classés en 2 grandes familles, RISC et CISC*, respectivement Reduce et Complex Instruction Set Computer. Les architectures RISC n'implémentent en général que des instructions élémentaires (CPU's ARM, MIPS, 8051, PIC18 …). A l'inverse, les architectures CISC (CPU's x86-64, 68xxx …) implémentent nativement au niveau assembleur des traitements pouvant être très complexes (division, opérations vectorielles, opérations sur des chaînes de caractères …).*

En 2012, la frontière entre ces deux familles est de plus en plus fine. Par exemple, le jeu d'instructions des processeurs spécialisés DSP RISC-like TMS320C66xx de TI compte 323 instructions. Néanmoins, les architectures compatibles x86-64 sont des architectures CISC. Nous allons rapidement comprendre pourquoi.

8 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

Avantages architecture CISC :

- *Empreinte mémoire programme faible, donc plus d'instructions contenues en cache. Néanmoins sur CPU CISC, en moyenne près de 80% des instructions compilées sont de types RISC.*
- *Compatibles x86-64, rétrocompatibilité des applications développées sur anciennes architectures.*

Inconvénients architecture CISC :

- *Architecture CPU complexe (mécanismes d'accélération matériels, décodeurs, Execution Units …), donc moins de place pour le cache.*
- *Jeu d'instructions mal géré par les chaînes de compilation (mécanismes d'optimisation)*

9 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

Inconvénients architecture RISC :

- *Empreinte mémoire programme élevée, donc moins d'instructions contenues en cache et mémoire principale.*

Avantages architecture RISC :

- *Architecture du CPU moins complexe (mécanismes d'accélération matériels, décodeurs, unités d'exécution …).*
- *En général, tailles instructions fixes et souvent exécution en un ou deux cycles CPU.*
- *Jeu d'instructions plus simple à appréhender pour le développeur et donc le compilateur. Jeu d'instructions très bien géré par les chaînes de compilations (mécanismes d'optimisation). Beaucoup d'architectures RISC récentes, travaillent avec de nombreux registres de travail généralistes, facilite le travail du compilateur.*

10 – copyleft

ENSICAEN Computer Science
ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

Observons le jeu d'instructions complet d'un CPU RISC 8051 proposé par Intel en 1980. En 2012, cette famille de CPU, même si elle reste très ancienne, est toujours extrêmement répandue et intégrée dans de nombreux MCU's ou ASIC's (licence libre). Prenons quelques exemples de fondeurs les utilisant : NXP, silabs, Atmel …

8051 Intel CPU (only CPU)
(1980)

MCU Silabs with 8051 CPU
(2012)

11 – copyleft

ENSICAEN Computer Science
ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

8051 Instruction set

ACALL	*Absolute Call*	**MOV**	*Move Memory*
ADD, ADDC	*Add Accumulator (With Carry)*	**MOVC**	*Move Code Memory*
AJMP	*Absolute Jump*	**MOVX**	*Move Extended Memory*
ANL	*Bitwise AND*	**MUL**	*Multiply Accumulator by B*
CJNE	*Compare and Jump if Not Equal*	**NOP**	*No Operation*
CLR	*Clear Register*	**ORL**	*Bitwise OR*
CPL	*Complement Register*	**POP**	*Pop Value From Stack*
DA	*Decimal Adjust*	**PUSH**	*Push Value Onto Stack*
DEC	*Decrement Register*	**RET**	*Return From Subroutine*
DIV	*Divide Accumulator by B*	**RETI**	*Return From Interrupt*
DJNZ	*Decrement Register and Jump if Not Zero*	**RL**	*Rotate Accumulator Left*
INC	*Increment Register*	**RLC**	*Rotate Accumulator Left Through Carry*
JB	*Jump if Bit Set*	**RR**	*Rotate Accumulator Right*
JBC	*Jump if Bit Set and Clear Bit*	**RRC**	*Rotate Accumulator Right Through Carry*
JC	*Jump if Carry Set*	**SETB**	*Set Bit*
JMP	*Jump to Address*	**SJMP**	*Short Jump*
JNB	*Jump if Bit Not Set*	**SUBB**	*Subtract From Accumulator With Borrow*
JNC	*Jump if Carry Not Set*	**SWAP**	*Swap Accumulator Nibbles*
JNZ	*Jump if Accumulator Not Zero*	**XCH**	*Exchange Bytes*
JZ	*Jump if Accumulator Zero*	**XCHD**	*Exchange Digits*
LCALL	*Long Call*	**XRL**	*Bitwise Exclusive OR*
LJMP	*Long Jump*		

12 – copyleft

ENSICAEN Computer Science
ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

Observons le jeu d'instructions complet d'un CPU 16bits CISC 8086 proposé par Intel en 1978. Il s'agit du premier processeur de la famille x86. En 2012, un corei7 est toujours capable d'exécuter le jeu d'instruction d'un 8086. Bien sûr, la réciproque n'est pas vraie.

8086 Intel CPU
(1978)

13 – copyleft

ENSICAEN Computer Science
ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

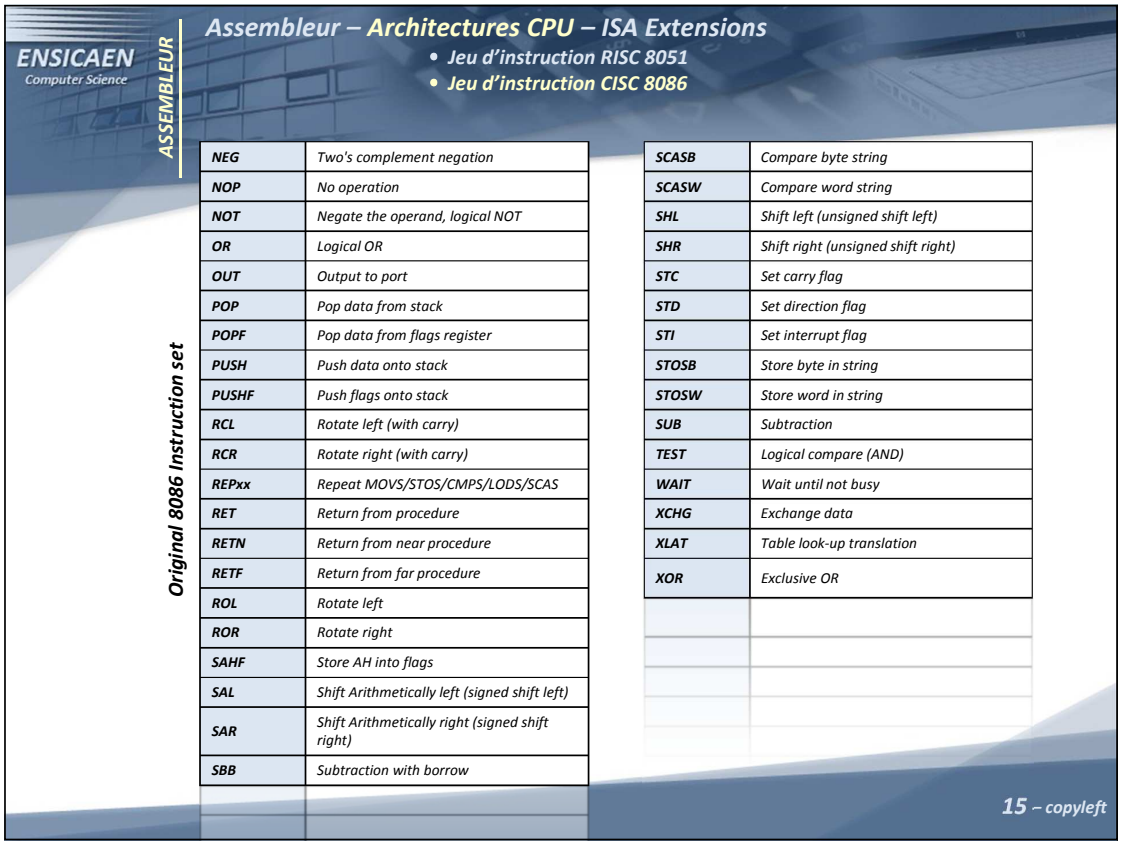
- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

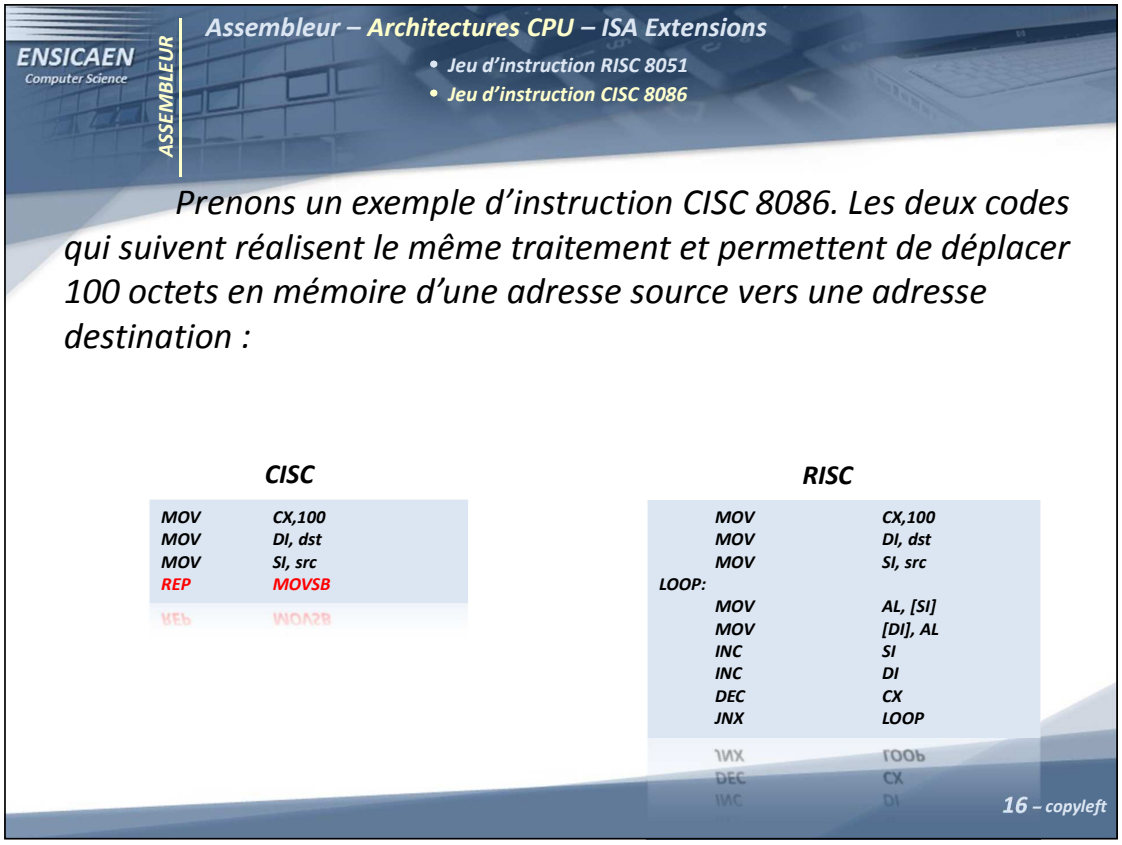
Original 8086 Instruction set

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*

14 – copyleft


ENSICAEN
Computer Science
ASSEMBLEUR
Assembleur – Architectures CPU – ISA Extensions
• Jeu d'instruction RISC 8051
• Jeu d'instruction CISC 8086
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
15 – copyleft


ENSICAEN
Computer Science
ASSEMBLEUR
Assembleur – Architectures CPU – ISA Extensions
• Jeu d'instruction RISC 8051
• Jeu d'instruction CISC 8086
Prenons un exemple d'instruction CISC 8086. Les deux codes qui suivent réalisent le même traitement et permettent de déplacer 100 octets en mémoire d'une adresse source vers une adresse destination :
CISC
MOV CX,100
MOV DI, dst
MOV SI, src
REP MOVSB
RISC
MOV CX,100
MOV DI, dst
MOV SI, src
LOOP:
MOV AL, [SI]
MOV [DI], AL
INC SI
INC DI
DEC CX
JNX LOOP
16 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

Attention, si vous lisez de l'assembleur x86-64, il existe deux syntaxes très répandues. La syntaxe Intel et la syntaxe AT&T utilisée par défaut par gcc (systèmes UNIX).

Intel Syntax

```
MOV     ebx,0FAh
```

AT&T Syntax

```
MOV     $0xFA, %ebx
```

Syntaxe AT&T :

- *Opérandes sources à gauche et destination à droite*
- *Constantes préfixées par $ (adressage immédiat)*
- *Constantes écrites avec syntaxe langage C (0x + valeur = hexadécimal)*
- *Registres préfixés par %*
- *Segmentation : [ds:20] devient %ds:20, [ss:bp] devient %ss:%bp …*

- *Adressage indirect [ebx] devient (%ebx), [ebx + 20h] devient 0x20(%ebx), [ebx+ecx*2h-1Fh] devient -0x1F(%ebx, %ecx, 0x2) …*
- *Suffixes, b=byte=1o, w=word=2o, s=short=4o, l=long=4o, q=quad=8o, t=ten=10o, o=octo=16o=128bits (x64)*
- *…*

17 – copyleft

Architecture et technologie des ordinateurs
ENSICAEN

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

- Jeu d'instruction RISC 8051
- Jeu d'instruction CISC 8086

Prenons un exemple de code écrit dans les 2 syntaxes :

Intel Syntax

```
        MOV     CX,100
        MOV     DI, dst
        MOV     SI, src
LOOP:
        MOV     AL, [SI]
        MOV     [DI], AL
        INC     SI
        INC     DI
        DEC     CX
        JNX     LOOP
```

AT&T Syntax

```
        movw    $100, %cx
        movw    dst, %di
        movw    src, %di
LOOP:
        movb    (%si), %al
        movb    %al, (%di)
        inc     %si
        inc     %di
        dec     %cx
        jnx     LOOP
```

18 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Par abus de langage, les CPU compatibles du jeu d'instruction 80x86 (8086, 80386, 80486..) sont nommés CPU x86. Depuis l'arrivée d'architectures 64bits ils sont par abus de langage nommés x64. Pour être rigoureux chez Intel, il faut nommer les jeux d'instructions et CPU 32bits associés IA-32 (depuis le 80386 en 1985) et les ISA 64bits Intel 64 ou EM64T (depuis le Pentium 4 Prescott en 2004).

L'une des grandes forces (et paradoxalement faiblesse) de ce jeu d'instruction est d'assurer une rétrocompatibilité avec les jeux d'instructions d'architectures antérieures. En contrepartie, il s'agit d'une architecture matérielle très complexe, difficile à accélérer imposant de fortes contraintes de consommation et d'échauffement.

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Extensions x86 et x64 n'opérant que sur des formats entiers :

CPU Architecture	Nom extension	Instructions
8086 Original x86	-	AAA, AAD, AAM, AAS, ADC, ADD, AND, CALL, CBW, CLC, CLD, CLI, CMC, CMP, CMPSzz, CWD, DAA, DAS, DEC, DIV, ESC, HLT, IDIV, IMUL, IN, INC, INT, INTO, IRET, Jcc, LAHF, LDS, LEA, LES, LOCK, LODSzz, LODSW, LOOPcc, MOV, MOVSzz, MUL, NEG, NOP, NOT, OR, OUT, POP, POPF, PUSH, PUSHF, RCL, RCR, REPcc, RET, RETF, ROL, ROR, SAHF, SAL, SALC, SAR, SBB, SCASzz, SHL, SAL, SHR, STC, STD, STI, STOSzz, SUB, TEST, WAIT, XCHG,XLAT, XOR
80186/80188	-	BOUND, ENTER, INSB, INSW, LEAVE, OUTSB, OUTSW, POPA, PUSHA, PUSHW
80286	-	ARPL, CLTS, LAR, LGDT, LIDT, LLDT, LMSW, LOADALL, LSL, LTR, SGDT, SIDT, SLDT, SMSW, STR, VERR, VERW
80386	-	BSF, BSR, BT, BTC, BTR, BTS, CDQ, CMPSD, CWDE, INSD, IRETD, IRETDF, IRETF, JECXZ, LFS, LGS, LSS, LODSD, LOOPD, LOOPED, LOOPNED, LOOPNZD, LOOPZD, MOVSD, MOVSX, MOVZX, OUTSD, POPAD, POPFD, PUSHAD, PUSHD, PUSHFD, SCASD, SETA, SETAE, SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETLE, SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG, SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, SETNZ, SETO, SETP, SETPE, SETPO, SETS, SETZ, SHLD, SHRD, STOSD
80486	-	BSWAP, CMPXCHG, INVD, INVLPG, WBINVD, XADD
Pentium	-	**CPUID**, CMPXCHG8B, RDMSR, RDPMC, WRMSR, RSM
Pentium pro	-	CMOVA, CMOVAE, CMOVB, CMOVB, CMOVE, CMOVG, CMOVGE, CMOVL, CMOVLE, CMOVNA, CMOVNAE, CMOVNB, CMOVNBE, CMOVNC, CMOVNE, CMOVNG, CMOVNGE, CMOVNL, CMOVNLE, CMOVNO, CMOVNP, CMOVNS, CMOVNZ, CMOVO, CMOVP, CMOVPE, CMOVPO, CMOVS, CMOVZ, RDPMC, SYSENTER, SYSEXIT, UD2
Pentium III	SSE	MASKMOVQ, MOVNTPS, MOVNTQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE
Pentium 4	SSE2	CLFLUSH, LFENCE, MASKMOVDQU, MFENCE, MOVNTDQ, MOVNTI, MOVNTPD, PAUSE
Pentium 4	SSE3 Hyper Threading	LDDQU, MONITOR, MWAIT
Pentium 4 6x2	VMX	VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON
X86-64	-	CDQE, CQO, CMPSQ, CMPXCHG16B, IRETQ, JRCXZ, LODSQ, MOVSXD, POPFQ, PUSHFQ, **RDTSC**, SCASQ, STOSQ, SWAPGS
Pentium 4	VT-x	VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Les extensions x87 ci-dessous n'opèrent que sur des formats flottants. Historiquement, le 8087 était un coprocesseur externe utilisé comme accélérateur matériel pour des opérations flottantes. Ce coprocesseur fut intégré dans le CPU principal sous forme d'unité d'exécution depuis l'architecture 80486. Cette unité est souvent nommée FPU (Floating Point Unit).

CPU Architecture	Nom extension	Instructions
8087 Original x87	-	F2XM1, FABS, FADD, FADDP, FBLD, FBSTP, FCHS, FCLEX, FCOM, FCOMP, FCOMPP, FDECSTP, FDISI, FDIV, FDIVP, FDIVR, FDIVRP, FENI, FFREE, FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FILD, FIMUL, FINCSTP, FINIT, FIST, FISTP, FISUB, FISUBR, FLD, FLD1, FLDCW, FLDENV, FLDENVW, FLDL2E, FLDL2T, FLDLG2, FLDLN2, FLDPI, FLDZ, FMUL, FMULP, FNCLEX, FNDISI, FNENI, FNINIT, FNOP, FNSAVE, FNSAVEW, FNSTCW, FNSTENV, FNSTENVW, FNSTSW, FPATAN, FPREM, FPTAN, FRNDINT, FRSTOR, FRSTORW, FSAVE, FSAVEW, FSCALE, FSQRT, FST, FSTCW, FSTENV, FSTENVW, FSTP, FSTSW, FSUB, FSUBP, FSUBR, FSUBRP, FTST, FWAIT, FXAM, FXCH, FXTRACT, FYL2X, FYL2XP1
80287	-	FSETPM
80387	-	FCOS, FLDENVD, FNSAVED, FNSTENVD, FPREM1, FRSTORD, FSAVED, FSIN, FSINCOS, FSTENVD, FUCOM, FUCOMP, FUCOMPP
Pentium pro	-	FCMOVB, FCMOVBE, FCMOVE, FCMOVNB, FCMOVNBE, FCMOVNE, FCMOVNU, FCMOVU, FCOMI, FCOMIP, FUCOMI, FUCOMIP, FXRSTOR, FXSAVE
Pentium 4	SSE3	FISTTP

21 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

*Les extensions présentées ci-dessous inplémentent toutes des instructions dites **SIMD (Single Instruction Multiple Data)** :*

- ***MMX :** MultiMedia eXtensions*
- ***SSE :** Streaming SIMD Extensions*
- ***AVX :** Advanced Vector Extensions*
- ***AES :** Advanced Encryption Standard*

CPU Architecture	Nom extension		Instructions
Pentium MMX	MMX		EMMS, MOVD, MOVQ, PACKSSDW, PACKSSWB, PACKUSWB, PADDB, PADDD, PADDSB, PADDSW, PADDUSB, PADDUSW, PADDW, PAND, PANDN, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PMADDWD, PMULHW, PMULLW, POR, PSLLD, PSLLQ, PSLLW, PSRAD, PSRAW, PSRLD, PSRLQ, PSRLW, PSUBB, PSUBD, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PSUBW, PUNPCKHBW, PUNPCKHDQ, PUNPCKHWD, PUNPCKLBW, PUNPCKLDQ, PUNPCKLWD, PXOR
Pentium III	SSE	Float Inst.	ADDPS, ADDSS, CMPPS, CMPSS, COMISS, CVTPI2PS, CVTPS2PI, CVTSI2SS, CVTSS2SI, CVTTPS2PI, CVTTSS2SI, DIVPS, DIVSS, LDMXCSR, MAXPS, MAXSS, MINPS, MINSS, MOVAPS, MOVHLPS, MOVHPS, MOVLHPS, MOVLPS, MOVMSKPS, MOVNTPS, MOVSS, MOVUPS, MULPS, MULSS, RCPPS, RCPSS, RSQRTPS, RSQRTSS, SHUFPS, SQRTPS, SQRTSS, STMXCSR, SUBPS, SUBSS, UCOMISS, UNPCKHPS, UNPCKLPS
		Integer Inst.	ANDNPS, ANDPS, ORPS, PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADBW, PSHUFW, XORPS
Pentium 4	SSE2	Float Inst.	ADDPD, ADDSD, ANDNPD, ANDPD, CMPPD, CMPSD, COMISD, CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSS2SD, CVTTPD2DQ, CVTTPD2PI, CVTTPS2DQ, CVTTSD2SI, DIVPD, DIVSD, MAXPD, MAXSD, MINPD, MINSD, MOVAPD, MOVHPD, MOVLPD, MOVMSKPD, MOVSD, MOVUPD, MULPD, MULSD, ORPD, SHUFPD, SQRTPD, SQRTSD, SUBPD, SUBSD, UCOMISD, UNPCKHPD, UNPCKLPD, XORPD
		Integer Inst.	MOVDQ2Q, MOVDQA, MOVDQU, MOVQ2DQ, PADDQ, PSUBQ, PMULUDQ, PSHUFHW, PSHUFLW, PSHUFD, PSLLDQ, PSRLDQ, PUNPCKHQDQ, PUNPCKLQDQ
	SSE3	Float Inst.	ADDSUBPD, ADDSUBPS, HADDPD, HADDPS, HSUBPD, HSUBPS, MOVDDUP, MOVSHDUP, MOVSLDUP

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Les instructions et opérandes usuellement manipulées par grand nombre de CPU sur le marché sont dites scalaires. Nous parlerons de ***processeur scalaire*** *(PIC de Microchip, 8051 de Intel, AVR de Atmel, C5xxx de TI…). Par exemple sur 8086 de Intel , prenons l'exemple d'une addition : scalaire + scalaire = scalaire :*

```
add     %bl,%al
```

A titre indicatif, les instructions MMX, SSE, AVX, AES … sont dîtes vectorielles. Les opérandes ne sont plus des grandeurs scalaires mais des grandeurs vectorielles. Nous parlerons de ***processeur vectoriel*** *(d'autres architectures vectorielles existent). Prenons un exemple d'instruction vectorielle SIMD SSE4.1, vecteur . vecteur = scalaire :*

```
dpps    0xF1, %xmm2,%xmm1
```

23 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Cette instruction vectorielle peut notamment être très intéressante pour des applications de traitement numérique du signal : ***dpps signifie dot product packet single****, soit produit scalaire sur un paquet de données au format flottant en simple précision (IEEE-754). Observons le descriptif de l'instruction ainsi qu'un exemple :*

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*.

http://www.intel.com

24 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Etudions un exemple d'exécution de l'instruction dpps :

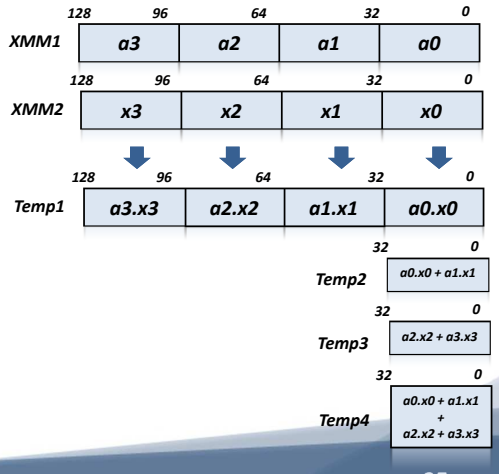
dpps 0xF1, %xmm2,%xmm1

Operation

```
DP_primitive (SRC1, SRC2)
IF (imm8[4] = 1)
    THEN Temp1[31:0] ← DEST[31:0] * SRC[31:0];
    ELSE Temp1[31:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[63:32] ← DEST[63:32] * SRC[63:32];
    ELSE Temp1[63:32] ← +0.0; FI;
IF (imm8[6] = 1)
    THEN Temp1[95:64] ← DEST[95:64] * SRC[95:64];
    ELSE Temp1[95:64] ← +0.0; FI;
IF (imm8[7] = 1)
    THEN Temp1[127:96] ← DEST[127:96] * SRC[127:96];
    ELSE Temp1[127:96] ← +0.0; FI;

Temp2[31:0] ← Temp1[31:0] + Temp1[63:32];
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96];
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0];
```

http://www.intel.com

XMMi (i = 0 à 15 with Intel 64)
128bits General Purpose Registers
for SIMD Execution Units



25 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

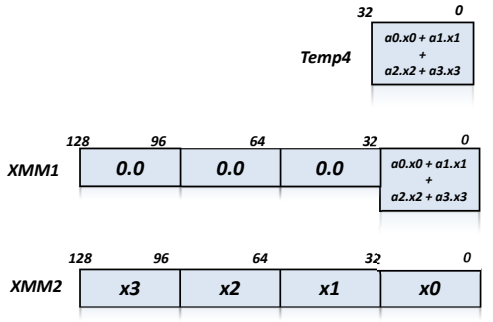
Etudions un exemple d'exécution de l'instruction dpps :

dpps 0xF1, %xmm2,%xmm1

```
IF (imm8[0] = 1)
    THEN DEST[31:0] ← Temp4[31:0];
    ELSE DEST[31:0] ← +0.0; FI;
IF (imm8[1] = 1)
    THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0; FI;
IF (imm8[2] = 1)
    THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0; FI;
IF (imm8[3] = 1)
    THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0; FI;

DPPS (128-bit Legacy SSE version)
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] (Unmodified)
```

http://www.intel.com

XMMi (i = 0 à 15 with Intel 64)
128bits General Purpose Registers
for SIMD Execution Units



26 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Les extensions x86-64 présentées jusqu'à maintenant ne présentent que les évolutions des jeux d'instructions apportées par Intel. Les extensions amenées par AMD ne seront pas présentées (MMX+, K6-2, 3DNow, 3DNow!+, SSE4a..).

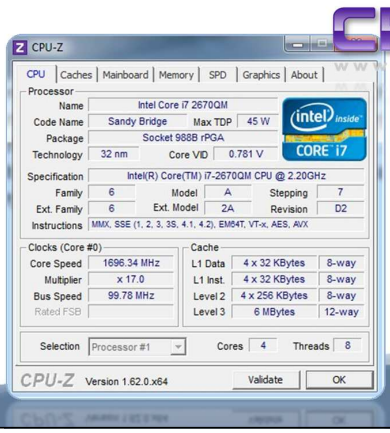
CPU Architecture	Nom extension	Instructions
Core2	SSSE3	PSIGNW, PSIGND, PSIGNB, PSHUFB, PMULHRSW, PMADDUBSW, PHSUBW, PHSUBSW, PHSUBD, PHADDW, PHADDSW, PHADDD, PALIGNR, PABSW, PABSD, PABSB
Core2 (45nm)	SSE4.1	MPSADBW, PHMINPOSUW, PMULLD, PMULDQ, **DPPS**, DPPD, BLENDPS, BLENDPD, BLENDVPS, BLENDVPD, PBLENDVB, PBLENDW, PMINSB, PMAXSB, PMINUW, PMAXUW, PMINUD, PMAXUD, PMINSD, PMAXSD, ROUNDPS, ROUNDSS, ROUNDPD, ROUNDSD, INSERTPS, PINSRB, PINSRD/PINSRQ, EXTRACTPS, PEXTRB, PEXTRW, PEXTRD/PEXTRQ, PMOVSXBW, PMOVZXBW, PMOVSXBD, PMOVZXBD, PMOVSXBQ, PMOVZXBQ, PMOVSXWD, PMOVZXWD, PMOVSXWQ, PMOVZXWQ, PMOVSXDQ, PMOVZXDQ, PTEST, PCMPEQQ, PACKUSDW, MOVNTDQA
Nehalem	SSE4.2	CRC32, PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM, PCMPGTQ
Sandy Bridge	AVX	VFMADDPD, VFMADDPS, VFMADDSD, VFMADDSS, VFMADDSUBPD, VFMADDSUBPS, VFMSUBADDPD, VFMSUBADDPS, VFMSUBPD, VFMSUBPS, VFMSUBSD, VFMSUBSS, VFNMADDPD, VFNMADDPS, VFNMADDSD, VFNMADDSS, VFNMSUBPD, VFNMSUBPS, VFNMSUBSD, VFNMSUBSS
Nehalem	AES	AESENC, AESENCLAST, AESDEC, AESDECLAST, AESKEYGENASSIST, AESIMC

27 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

L'instruction CPUID arrivée avec l'architecture Pentium permet de récupérer très facilement toutes les informations relatives à l'architecture matérielle du GPP (CPU's, Caches, adressage virtuel..). L'utilitaire libre CPU-Z utilise notamment ce registre pour retourner des informations sur l'architecture :



28 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

Assembleur – Architectures CPU – ISA Extensions

Sous Linux, vous pouvez également consulter le fichier ***/proc/cpuinfo*** *listant les informations retournées par l'instruction CPUID :*

```
vmlinux@vmlinux: ~
vmlinux@vmlinux:~$ cat /proc/cpuinfo | more
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 42
model name      : Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz
stepping        : 7
cpu MHz         : 2107.531
cache size      : 6144 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic se
pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm
e3 lahf_lm
bogomips        : 4215.06
clflush size    : 64
cache_alignment : 64
```

29 – copyleft

ENSICAEN Computer Science — ASSEMBLEUR

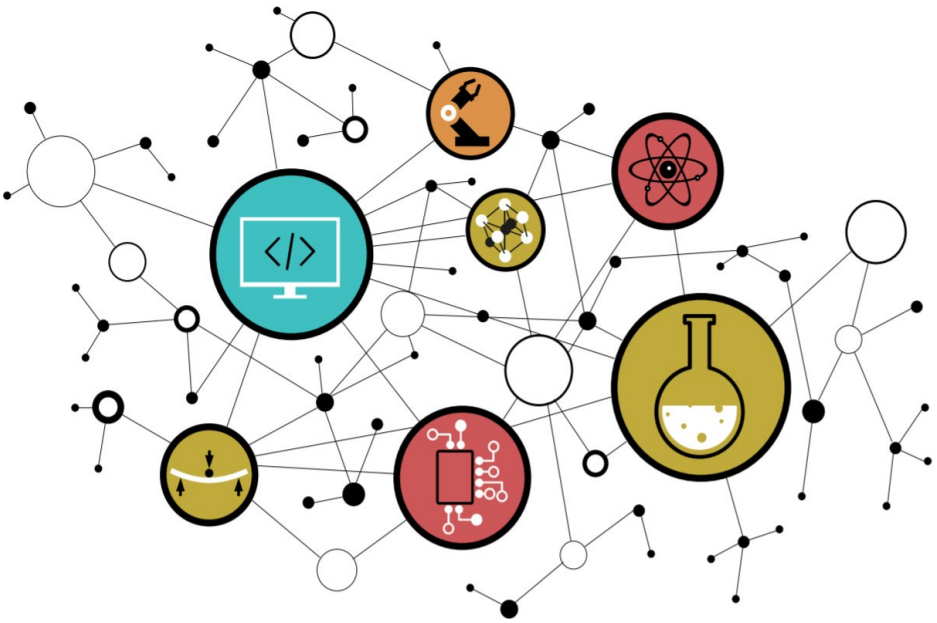
Assembleur – Architectures CPU – ISA Extensions

De même, lorsque l'on est amené à développer sur un processeur donné, il est essentiel de travailler avec les documents de référence proposés par le fondeur, Intel dans notre cas. Vous pouvez télécharger les différents documents de référence à cette URL : [*http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*](http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html)

30 – copyleft


ENSICAEN
Computer Science
Merci de votre attention !

CENTRAL PROCESSING UNIT



ENSICAEN
Computer Science

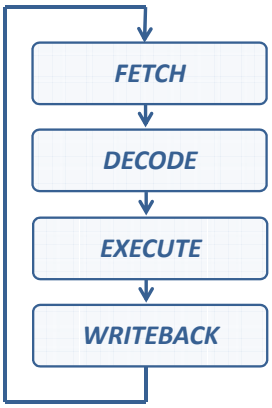
CENTRAL PROCESSING UNIT

Architecture et Technologie des Ordinateurs

Hugo Descoubes - Juin 2013

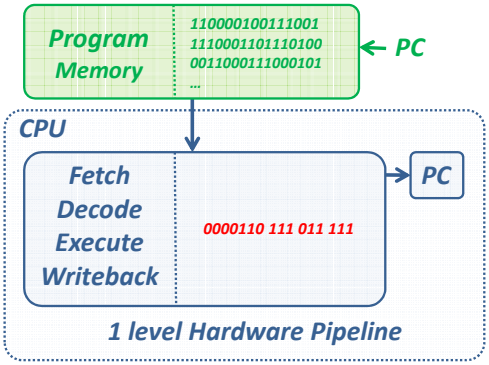
ENSICAEN
Computer Science
CPU

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.*

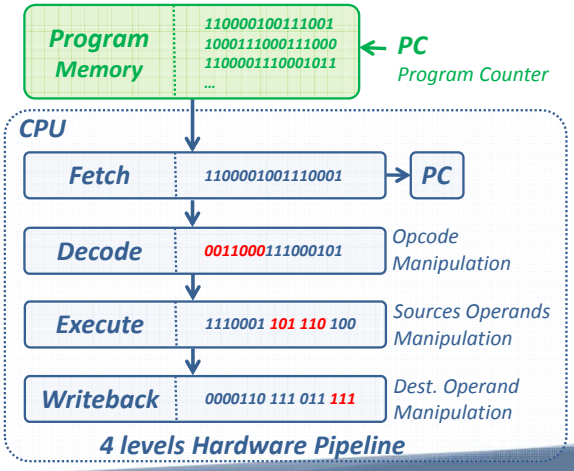
La très grande majorité des architectures modernes sont capables de réaliser une partie voire toutes ces étapes en parallèle. Nous parlerons de pipelining hardware.



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).

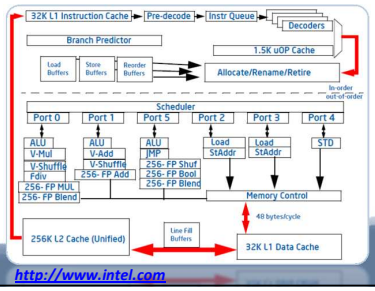
Il faudrait donc 4cy pour la première instruction et 1cy (théoriquement) pour les suivantes.

ENSICAEN Computer Science — CPU

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

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

5 – copyleft

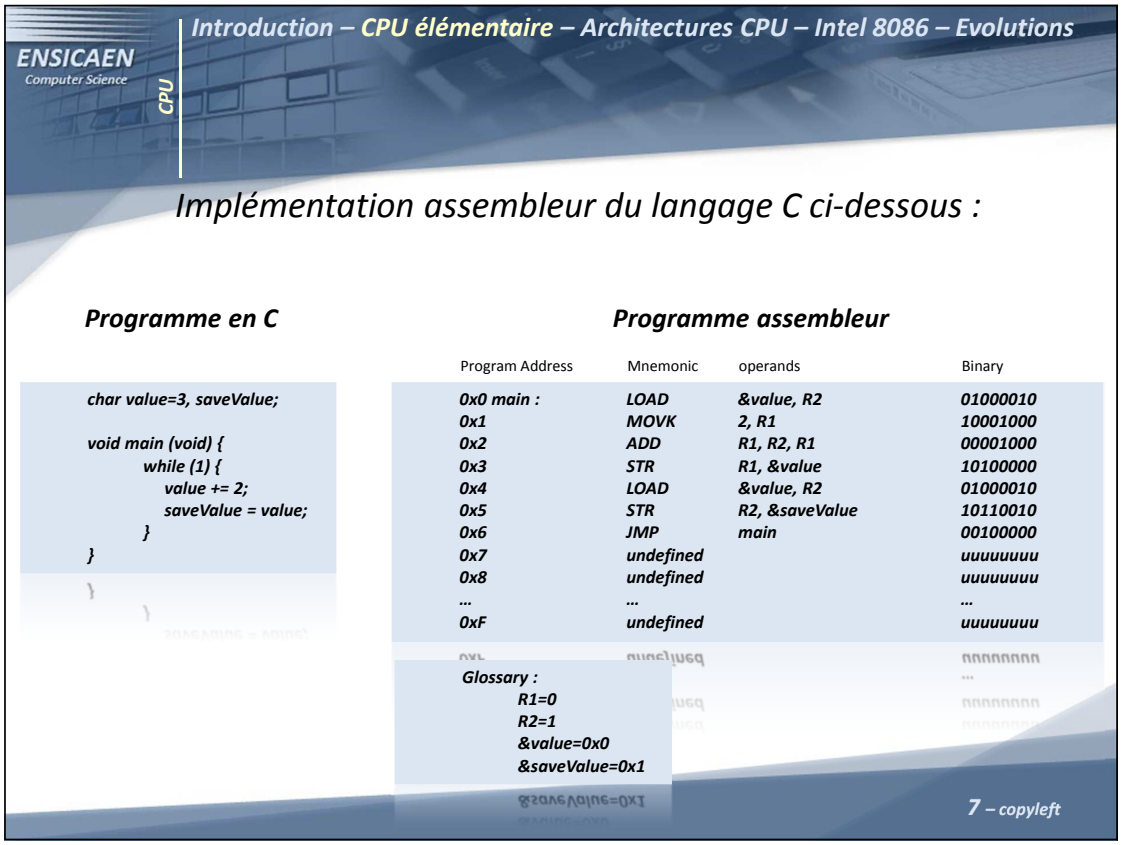
ENSICAEN Computer Science — CPU

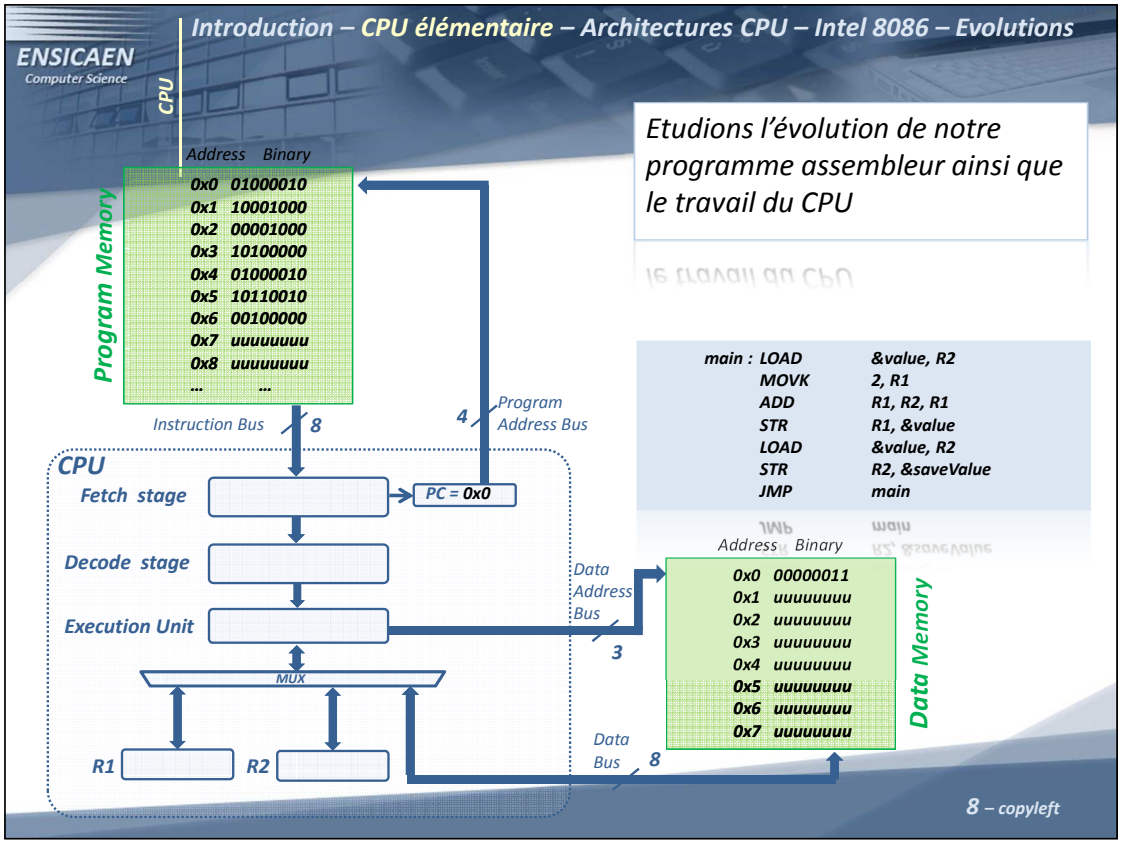
Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions

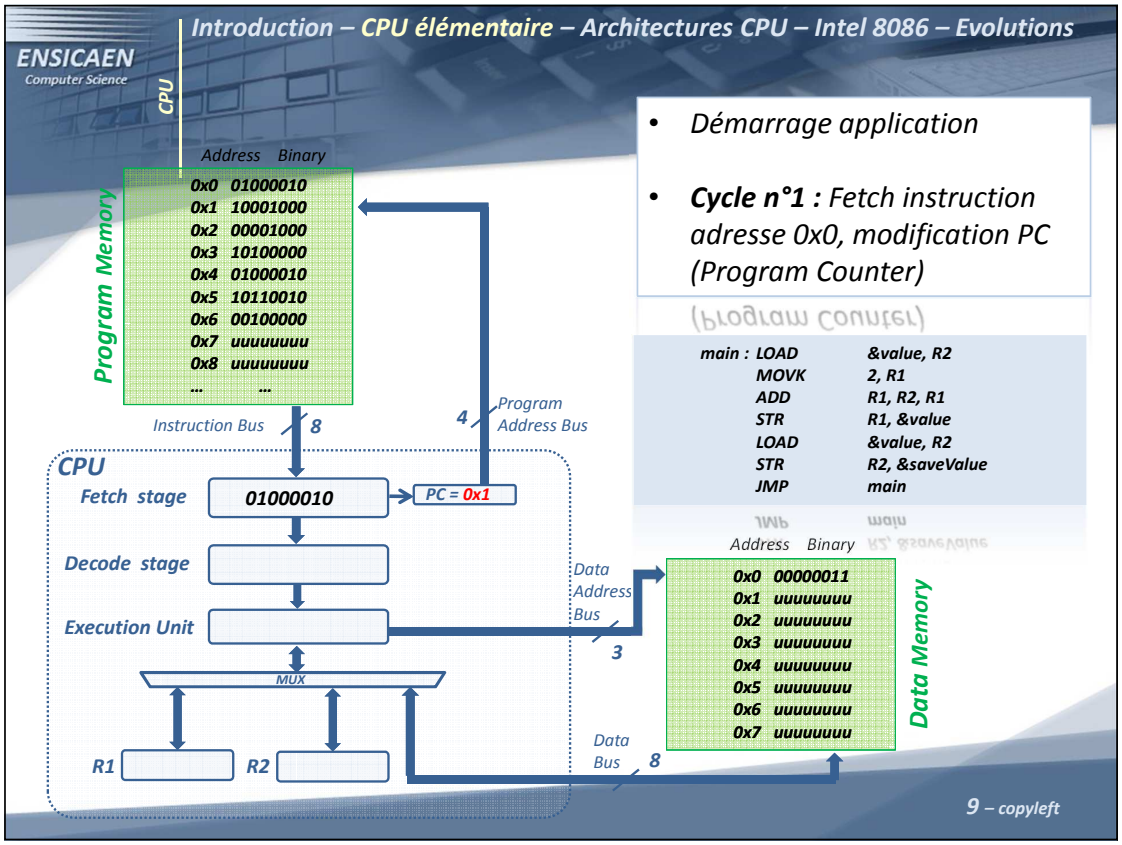
Etudions un CPU élémentaire RISC-like n'étant rattaché à aucune architecture connue. Observons le jeu d'instruction très très très réduit associé :

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				

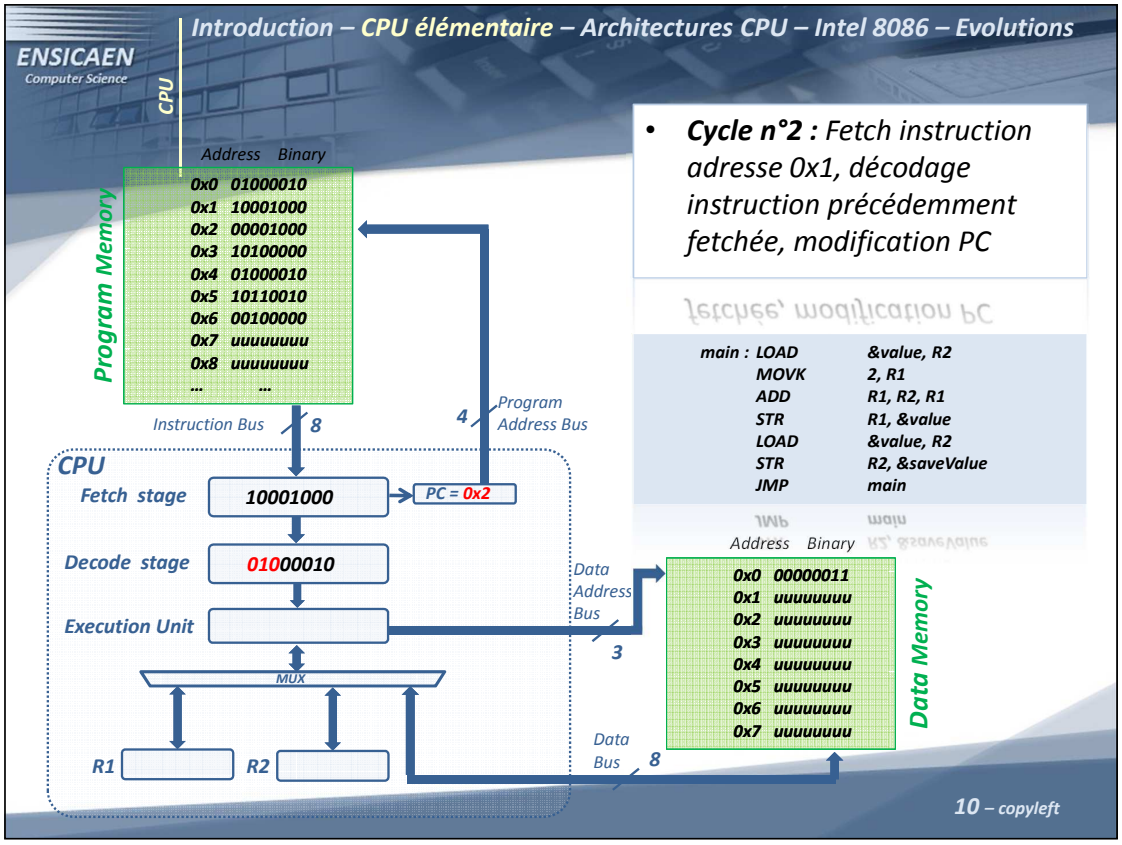
6 – copyleft

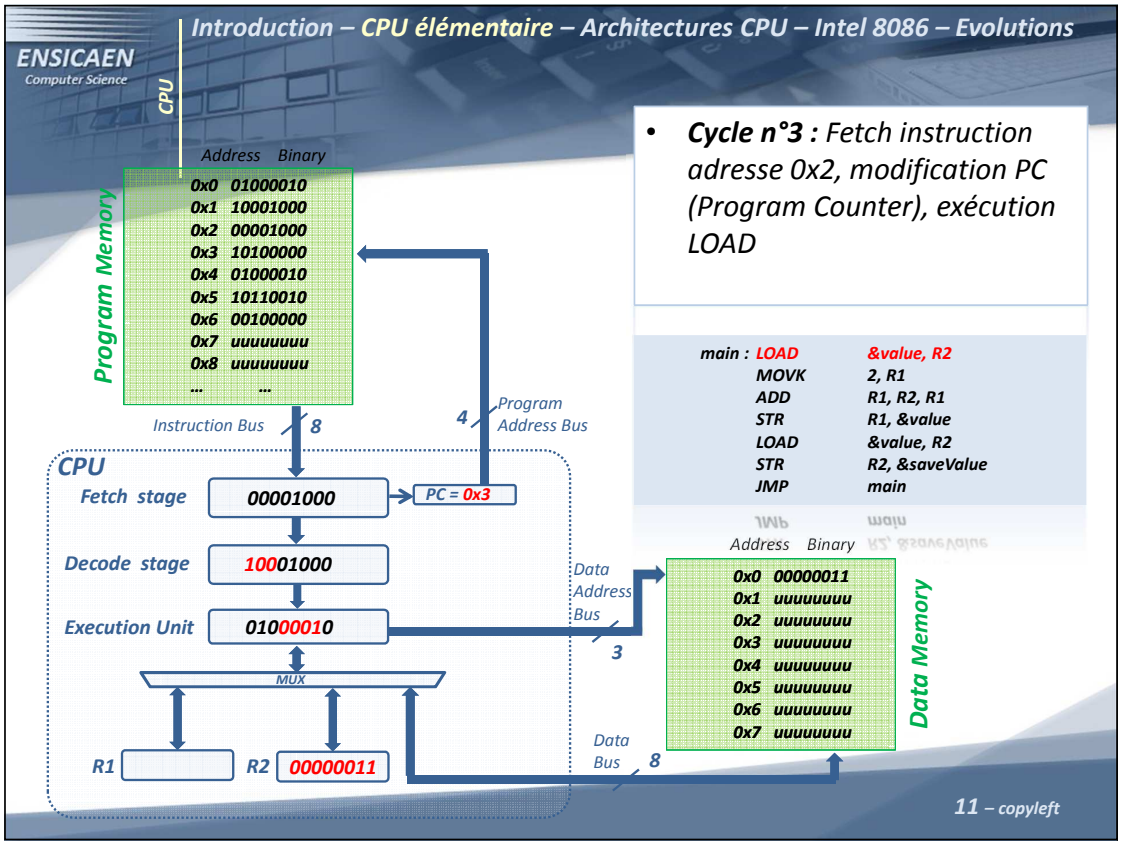

ENSICAEN
Computer Science
CPU
Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
Implémentation assembleur du langage C ci-dessous :
Programme en C
char value=3, saveValue;
void main (void) {
while (1) {
value += 2;
saveValue = value;
}
}
Programme assembleur
Program Address 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
7 – copyleft

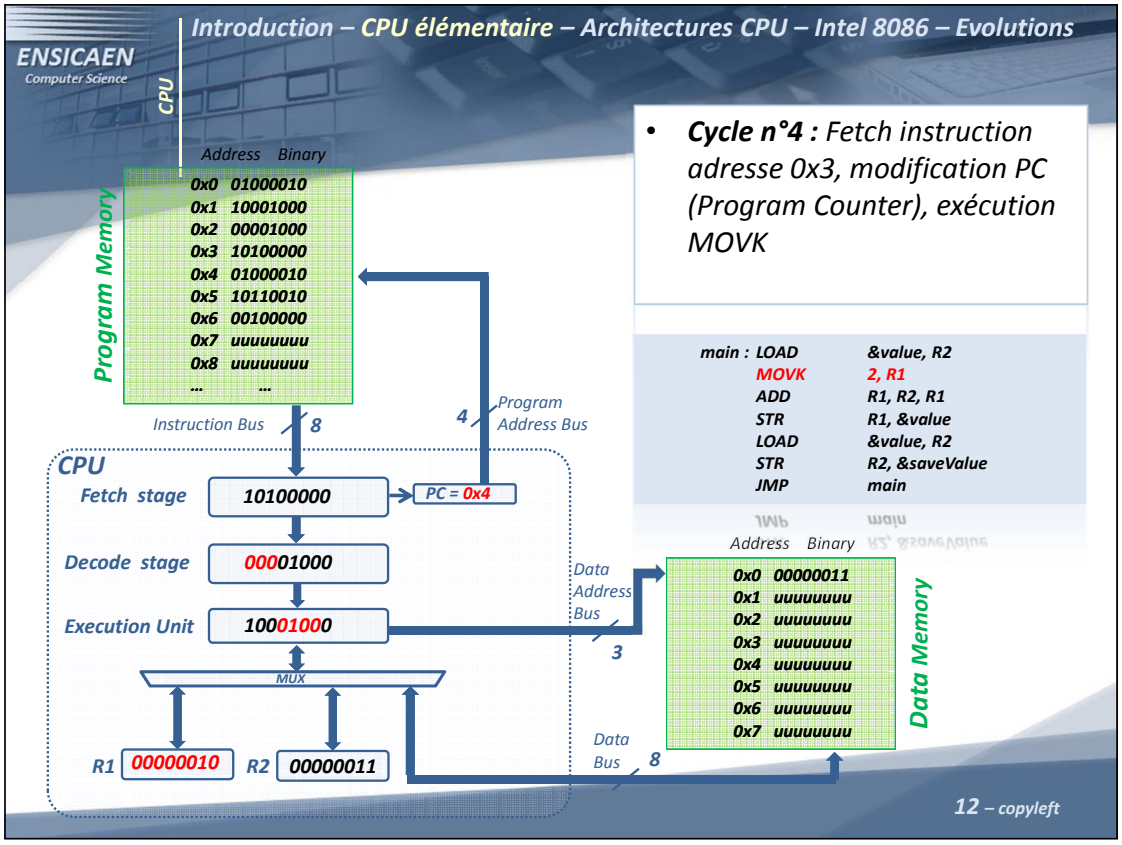

ENSICAEN
Computer Science
CPU
Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
Etudions l'évolution de notre programme assembleur ainsi que le travail du CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus 8
4 Program Address Bus
CPU
Fetch stage
PC = 0x0
Decode stage
Execution Unit
MUX
R1
R2
main : LOAD &value, R2
MOVK 2, R1
ADD R1, R2, R1
STR R1, &value
LOAD &value, R2
STR R2, &saveValue
JMP main
Address Binary
0x0 00000011
0x1 uuuuuuuu
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
Data Address Bus
3
Data Bus
8
Data Memory
8 – copyleft

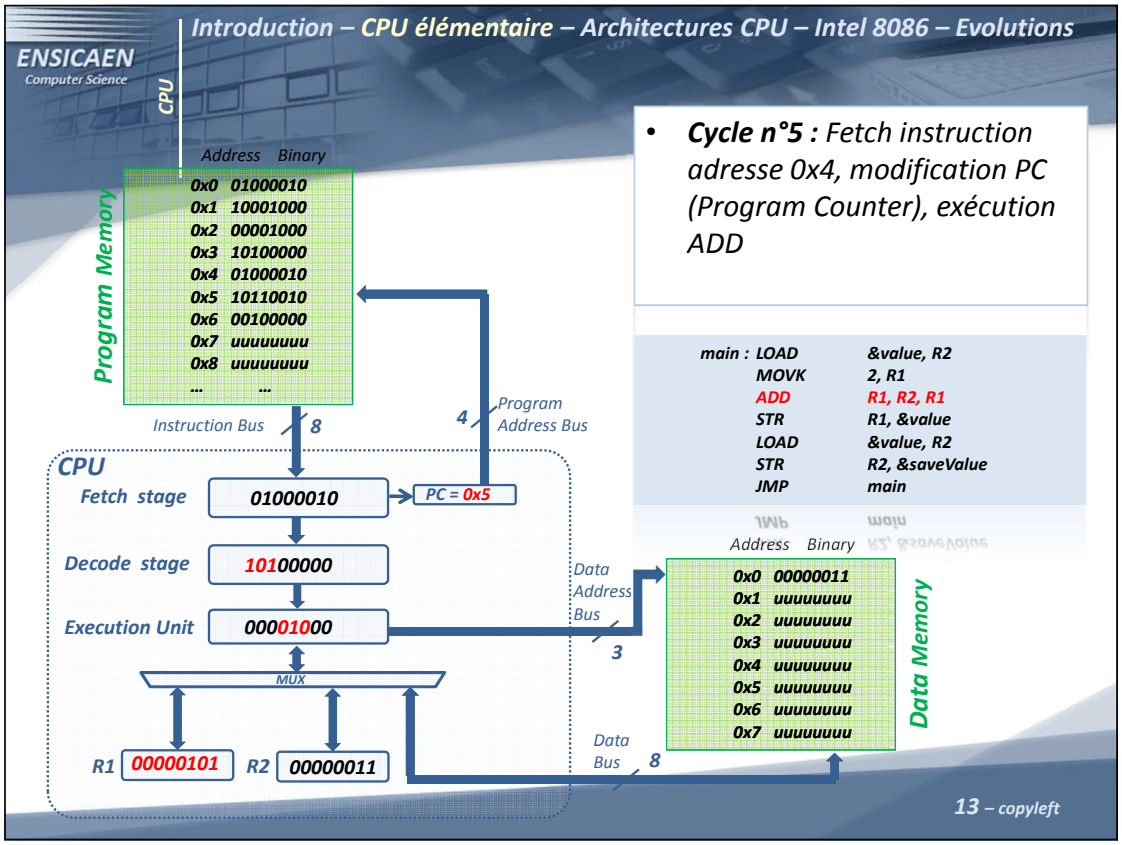


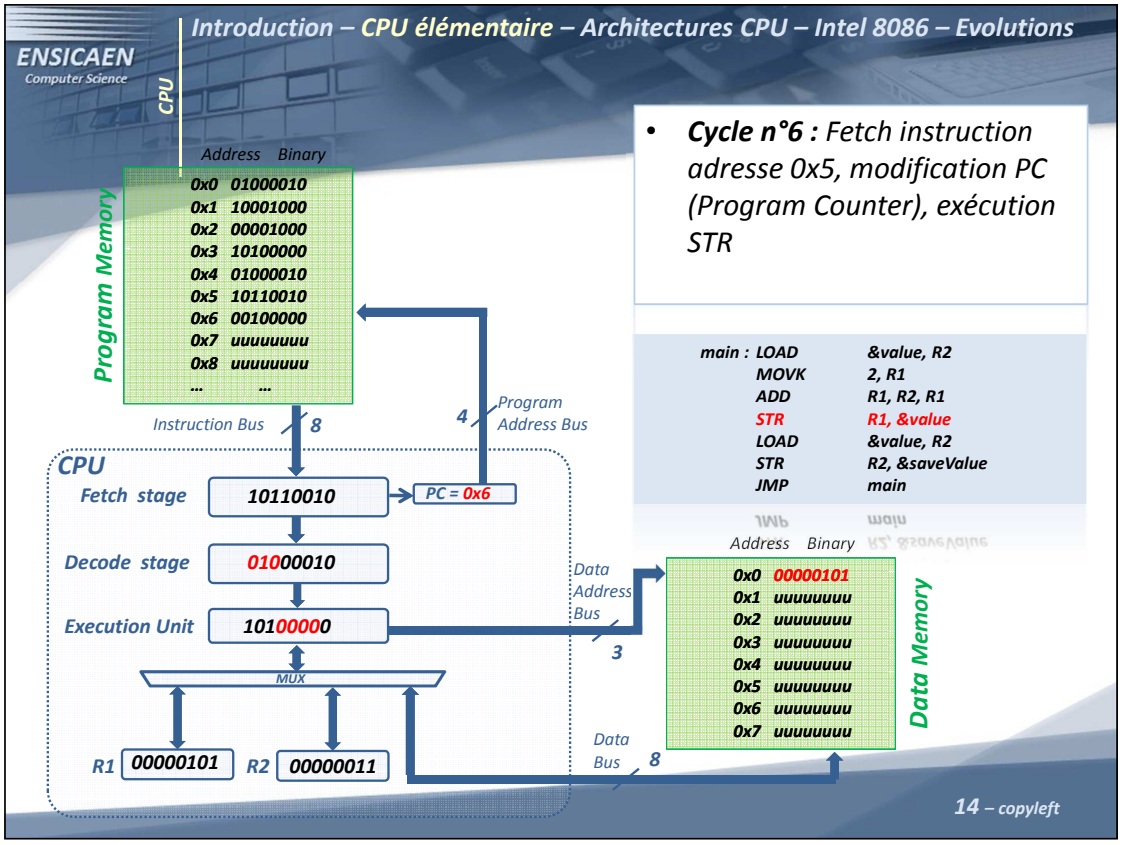
Architecture et technologie des ordinateurs
ENSICAEN

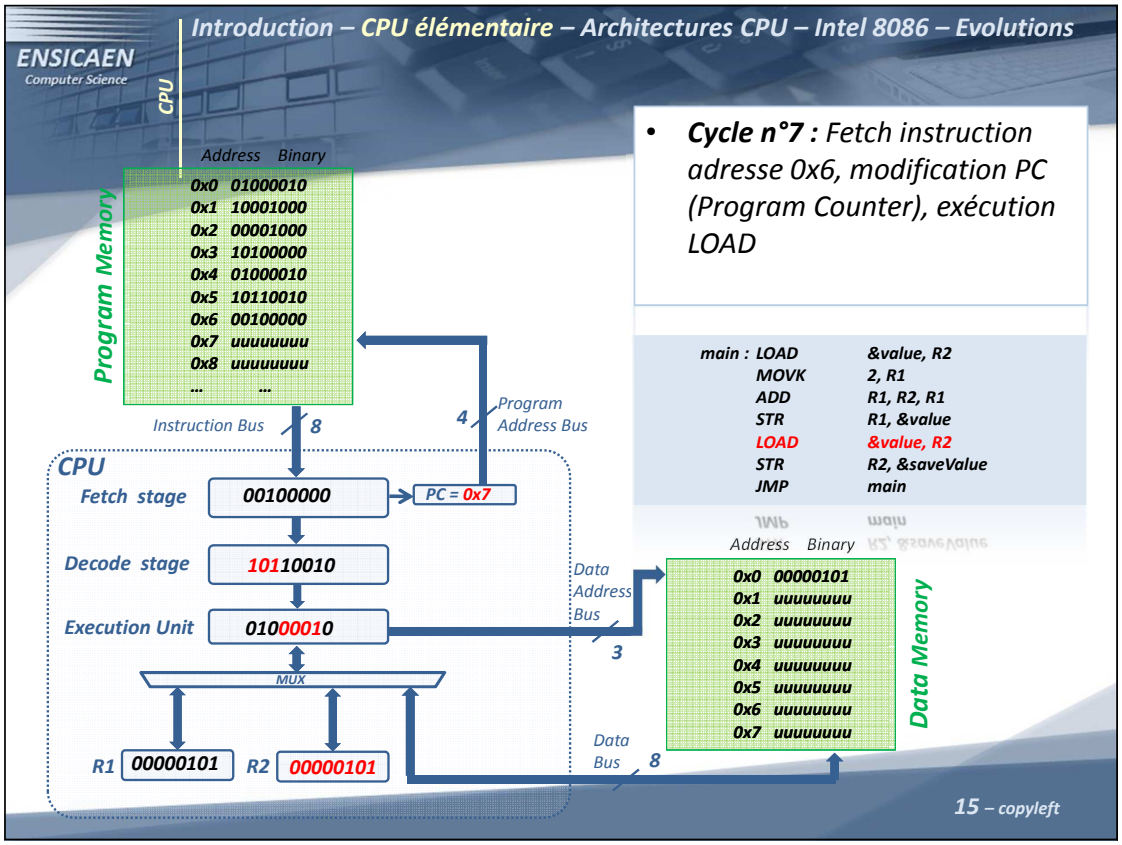


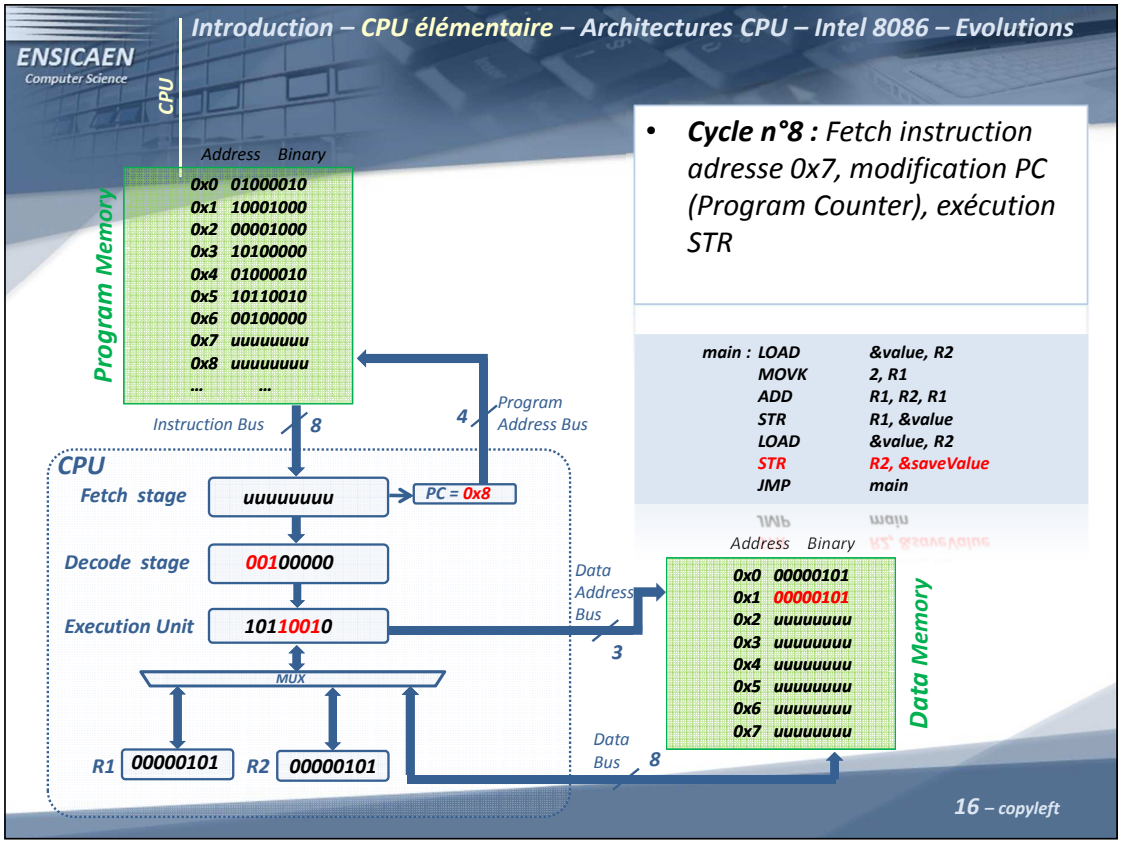

Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
ENSICAEN
Computer Science
CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus
8
4
Program Address Bus
CPU
Fetch stage
00001000
PC = 0x3
Decode stage
10001000
Execution Unit
01000010
MUX
R1
R2
00000011
Data Address Bus
3
Data Bus
8
Address Binary
0x0 00000011
0x1 uuuuuuuu
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
Data Memory
• 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
11 – copyleft

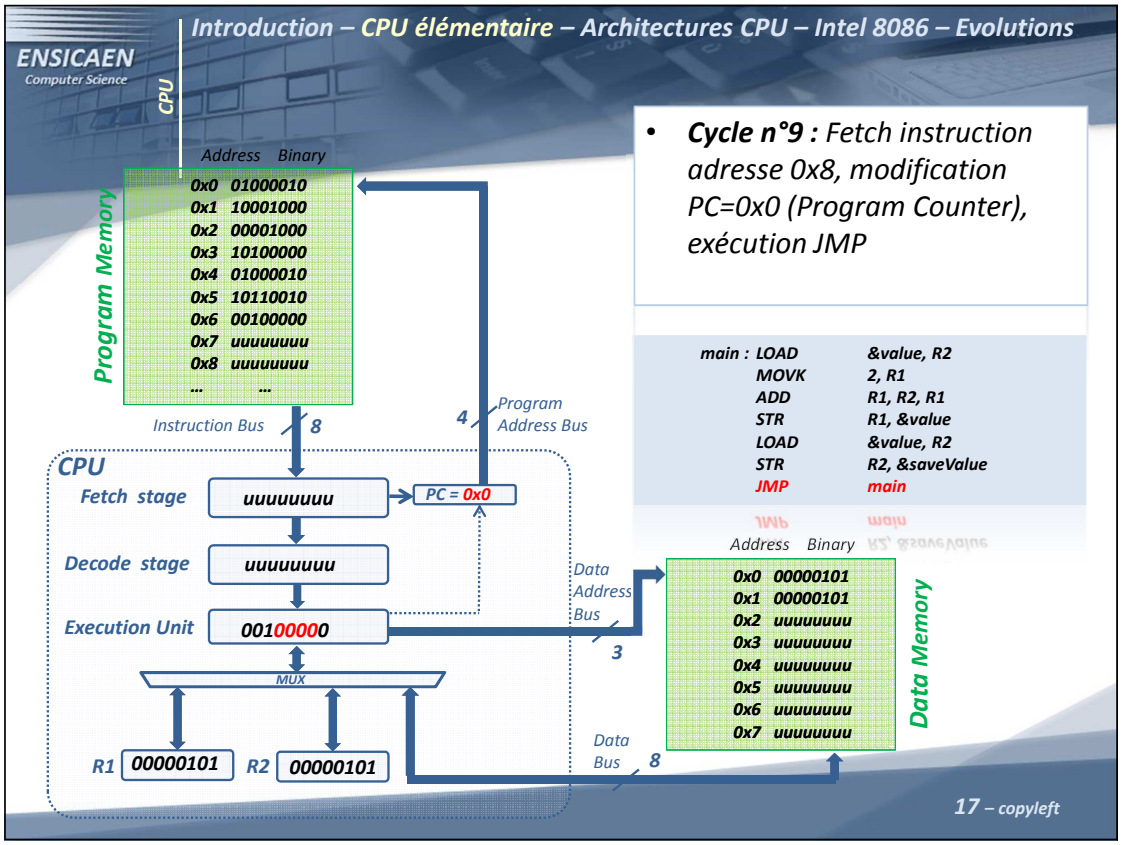

Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
ENSICAEN
Computer Science
CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus
8
4
Program Address Bus
CPU
Fetch stage
10100000
PC = 0x4
Decode stage
00001000
Execution Unit
10001000
MUX
R1
00000010
R2
00000011
Data Address Bus
3
Data Bus
8
Address Binary
0x0 00000011
0x1 uuuuuuuu
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
Data Memory
• 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
12 – copyleft

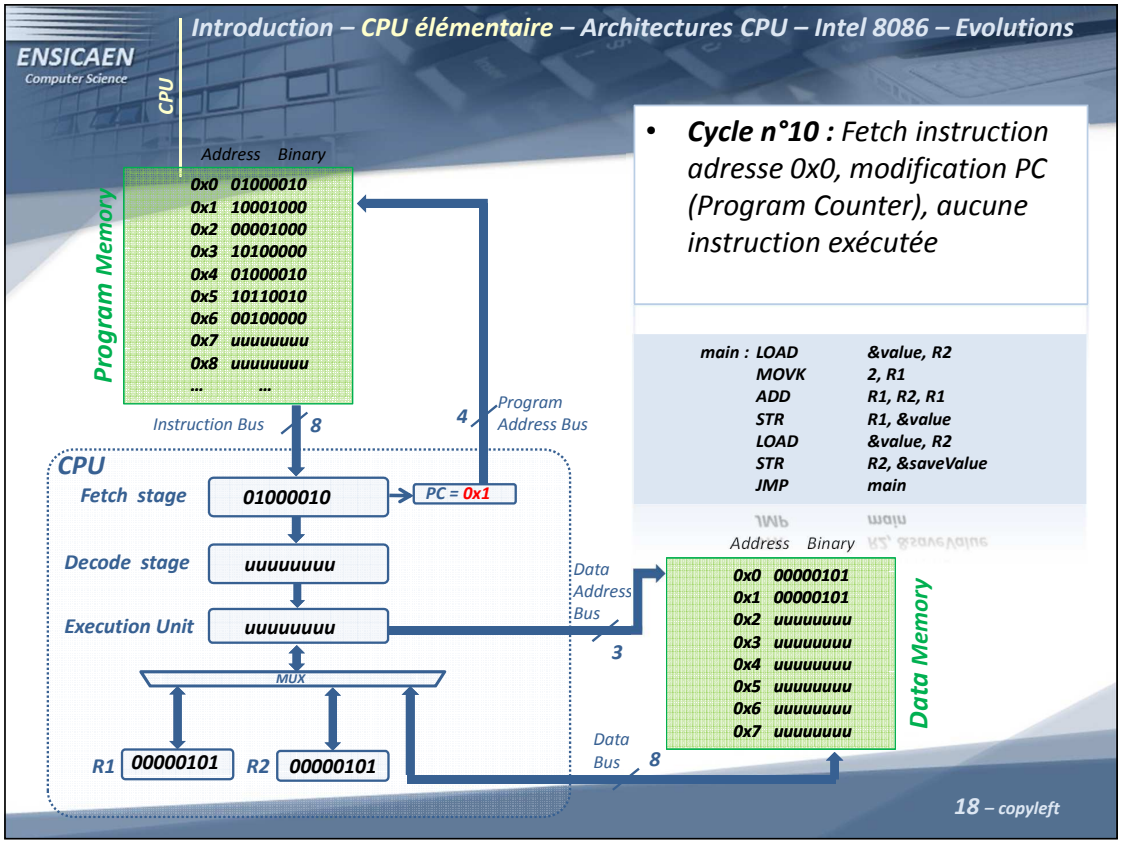

Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
ENSICAEN
Computer Science
CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus 8
4
Program Address Bus
CPU
Fetch stage 01000010
PC = 0x5
Decode stage 10100000
Execution Unit 000010000
MUX
R1 00000101
R2 00000011
Data Address Bus 3
Data Bus 8
Address Binary
0x0 00000011
0x1 uuuuuuuu
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
Data Memory
• 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
13 – copyleft

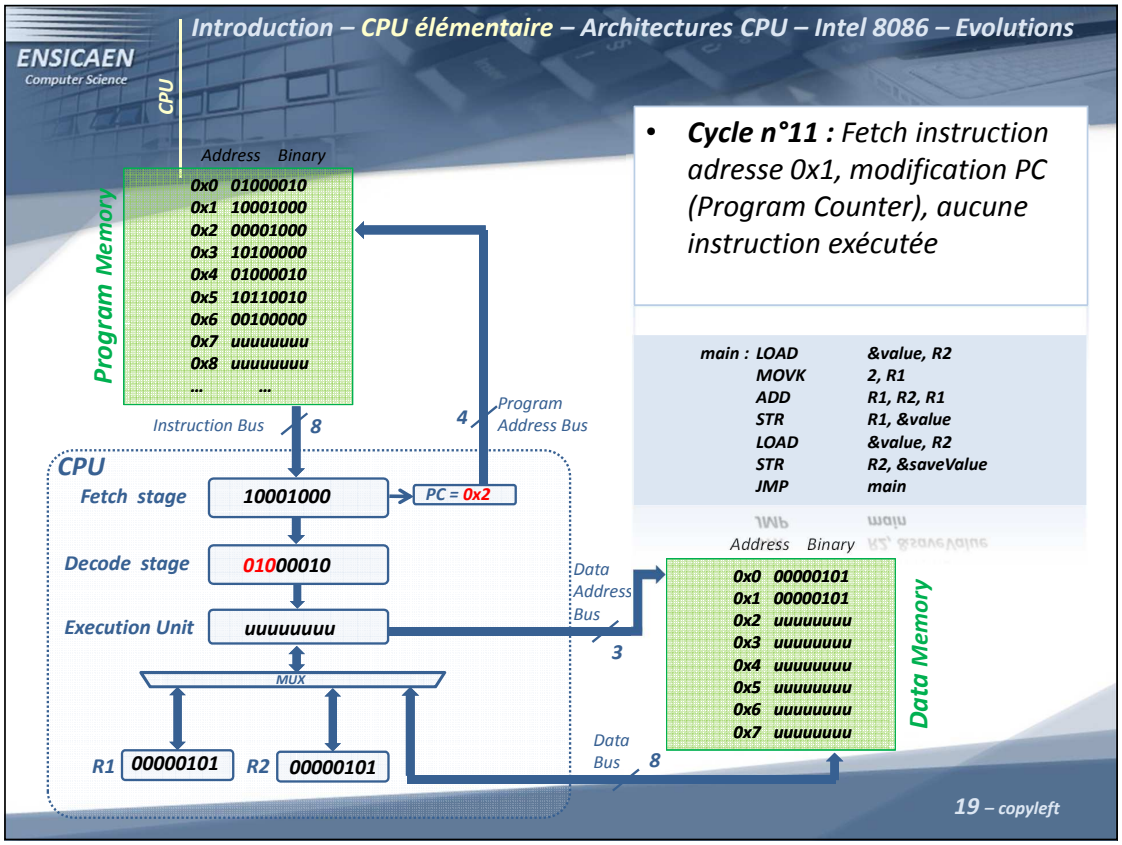

Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
ENSICAEN
Computer Science
CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus 8
4
Program Address Bus
CPU
Fetch stage 10110010
PC = 0x6
Decode stage 01000010
Execution Unit 101000000
MUX
R1 00000101
R2 00000011
Data Address Bus 3
Data Bus 8
Address Binary
0x0 00000101
0x1 uuuuuuuu
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
Data Memory
• 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
14 – copyleft


Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
ENSICAEN
Computer Science
CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus 8
4
Program Address Bus
CPU
Fetch stage 00100000
PC = 0x7
Decode stage 10110010
Execution Unit 01000010
MUX
R1 00000101
R2 00000101
Data Address Bus
3
Data Bus 8
Data Memory
Address Binary
0x0 00000101
0x1 uuuuuuuu
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
• 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
15 – copyleft

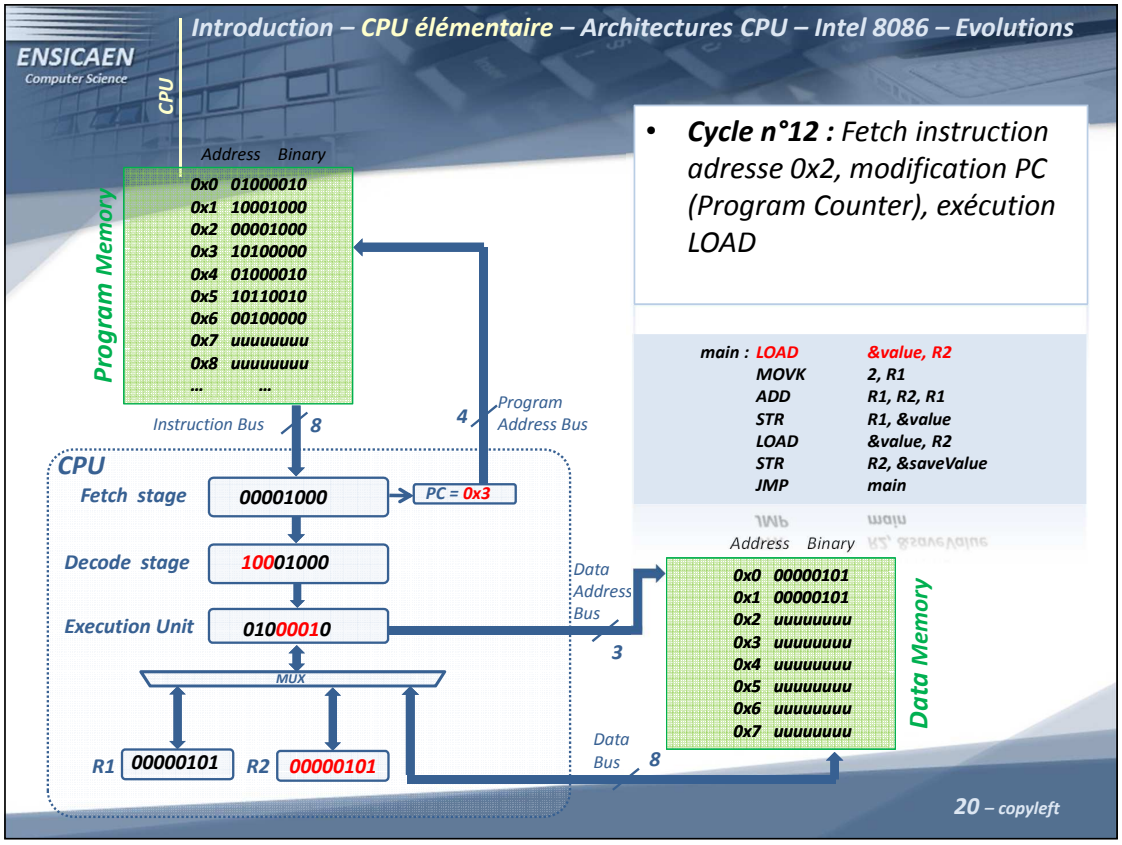

Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
ENSICAEN
Computer Science
CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus 8
4
Program Address Bus
CPU
Fetch stage uuuuuuuu
PC = 0x8
Decode stage 00100000
Execution Unit 10110010
MUX
R1 00000101
R2 00000101
Data Address Bus
3
Data Bus 8
Data Memory
Address Binary
0x0 00000101
0x1 00000101
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
• 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
16 – copyleft


Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
ENSICAEN
Computer Science
CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus
8
4
Program Address Bus
CPU
Fetch stage
uuuuuuuu
PC = 0x0
Decode stage
uuuuuuuu
Execution Unit
00100000
MUX
R1 00000101
R2 00000101
Data Address Bus
3
Data Bus
8
Address Binary
0x0 00000101
0x1 00000101
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
Data Memory
• 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
17 – copyleft


Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
ENSICAEN
Computer Science
CPU
Program Memory
Address Binary
0x0 01000010
0x1 10001000
0x2 00001000
0x3 10100000
0x4 01000010
0x5 10110010
0x6 00100000
0x7 uuuuuuuu
0x8 uuuuuuuu
... ...
Instruction Bus
8
4
Program Address Bus
CPU
Fetch stage
01000010
PC = 0x1
Decode stage
uuuuuuuu
Execution Unit
uuuuuuuu
MUX
R1 00000101
R2 00000101
Data Address Bus
3
Data Bus
8
Address Binary
0x0 00000101
0x1 00000101
0x2 uuuuuuuu
0x3 uuuuuuuu
0x4 uuuuuuuu
0x5 uuuuuuuu
0x6 uuuuuuuu
0x7 uuuuuuuu
Data Memory
• 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
18 – copyleft



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



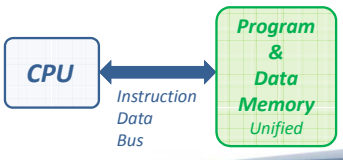
- **Cycle n°12 :** *Fetch instruction adresse 0x2, modification PC (Program Counter), exécution LOAD*

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.



ENSICAEN Computer Science — CPU

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

- Von Neumann
- Harvard
- Harvard Modifié

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).*

23 – copyleft

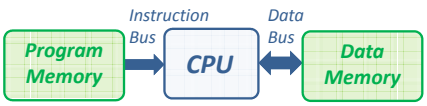
ENSICAEN Computer Science — CPU

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

- 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.



24 – copyleft

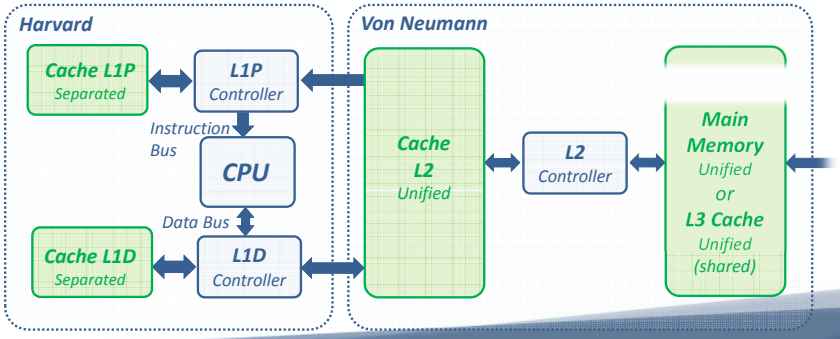
Observons quelques avantages de ce type d'architecture :

- ***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 :

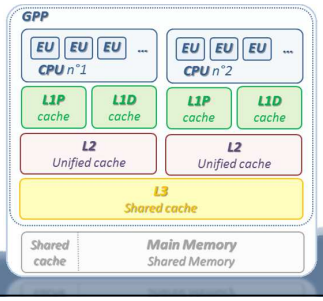
- ***Mapping mémoires distincts** (adresse mémoire donnée différente adresse mémoire programme). Moins flexible pour le développeur.*
- ***Peu polyvalent**. Certaines applications exigent une large empreinte en mémoire donnée (traitement image et son, bases de données…) pour d'autres ce sera le code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

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 :*



- Von Neumann
- Harvard
- Harvard Modifié

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).

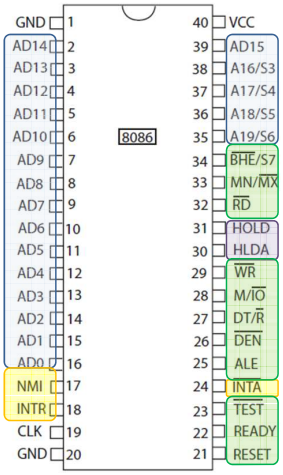
L'un des principaux dangers de ce type de mémoire, est la ***cohérence des informations*** *présentes dans la hiérarchie mémoire du processeurs. Par exemple pour un coreix de la famille sandy bridge, une même donnée peut exister avec différentes valeurs en mémoire principale (DDR), mémoire cache L3 (shared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

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction

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

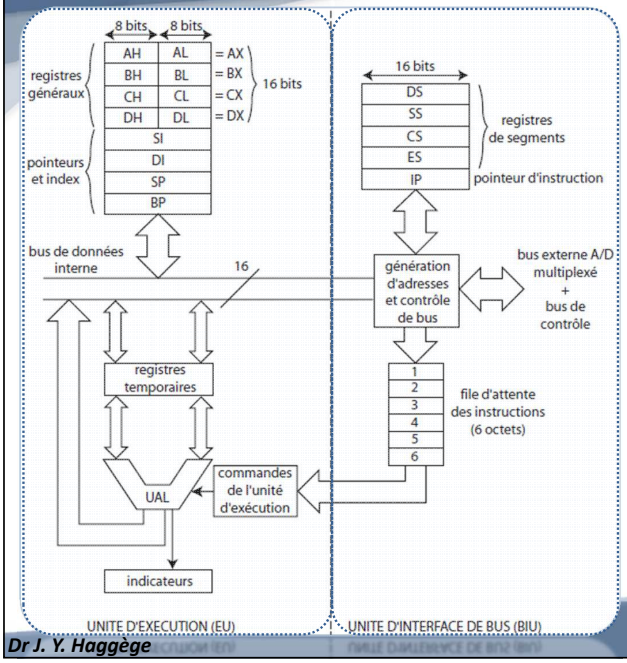
: Interruptions

: Direct Memory Access

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction



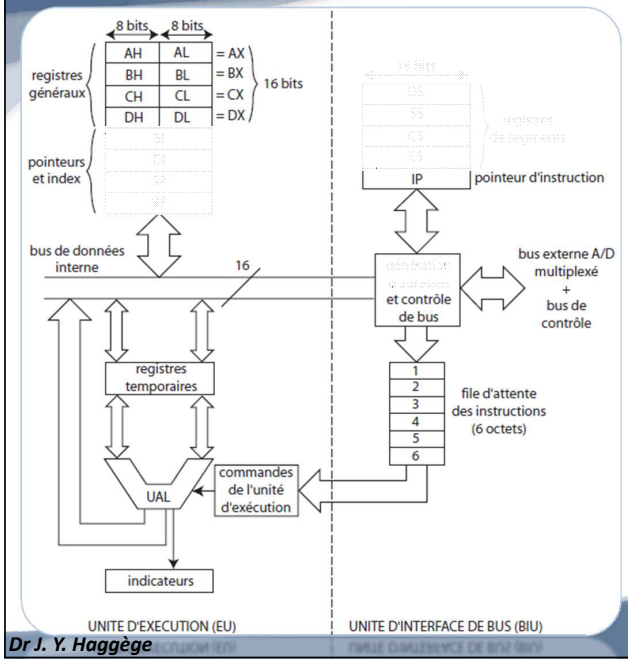
- ***Execution Unit :*** *décode puis exécute les instructions présentes dans la file d'attente*

- ***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).*

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction



- ***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.*

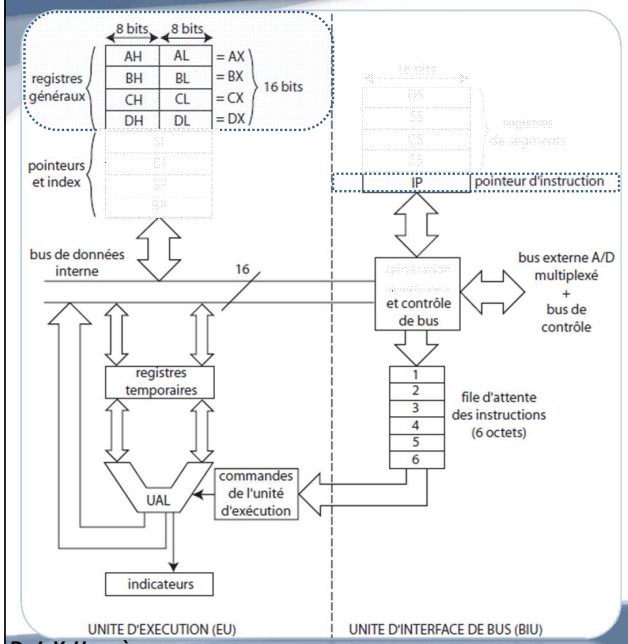
Dr J. Y. Haggège

31 – copyleft

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction



- ***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.*

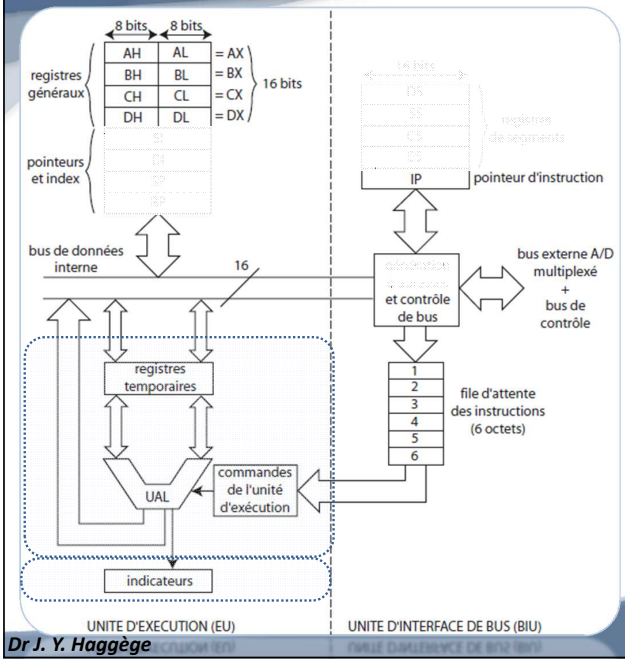
Dr J. Y. Haggège

32 – copyleft

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction



- ***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

Dr J. Y. Haggège

33 – copyleft

ENSICAEN Computer Science — CPU

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

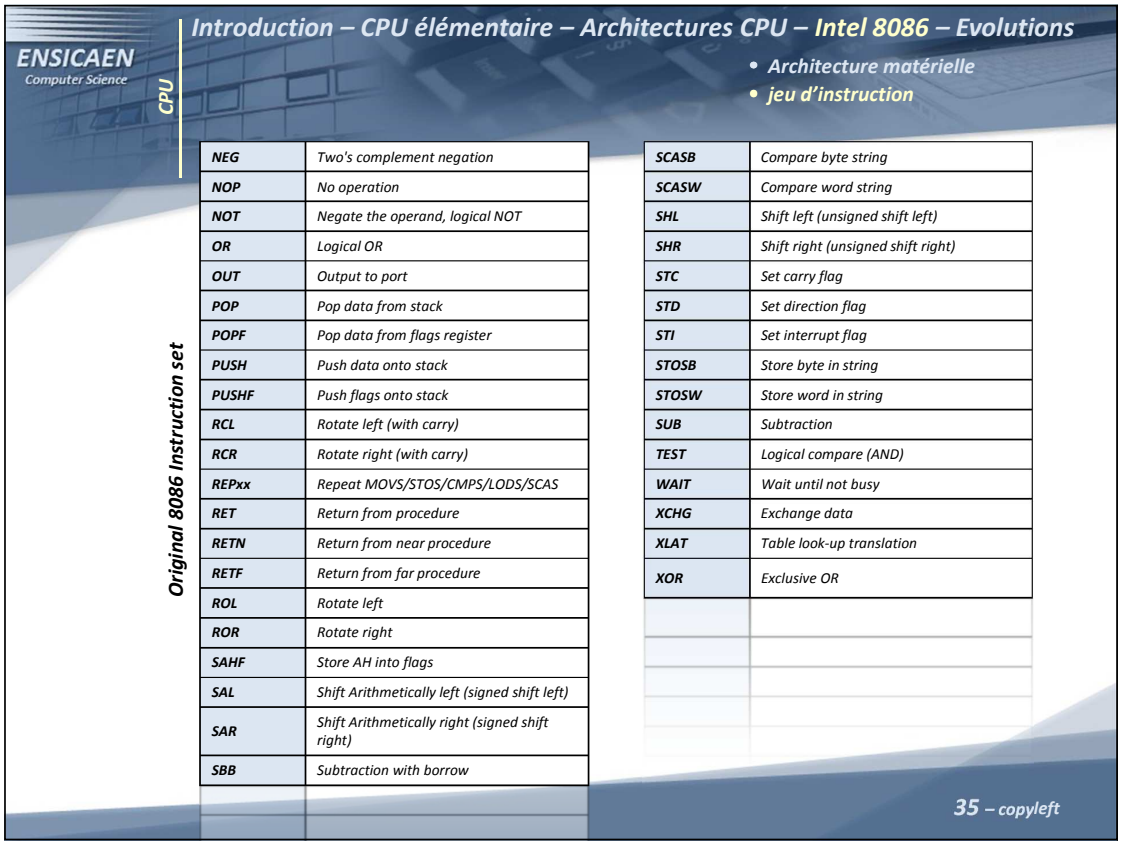
- Architecture matérielle
- jeu d'instruction

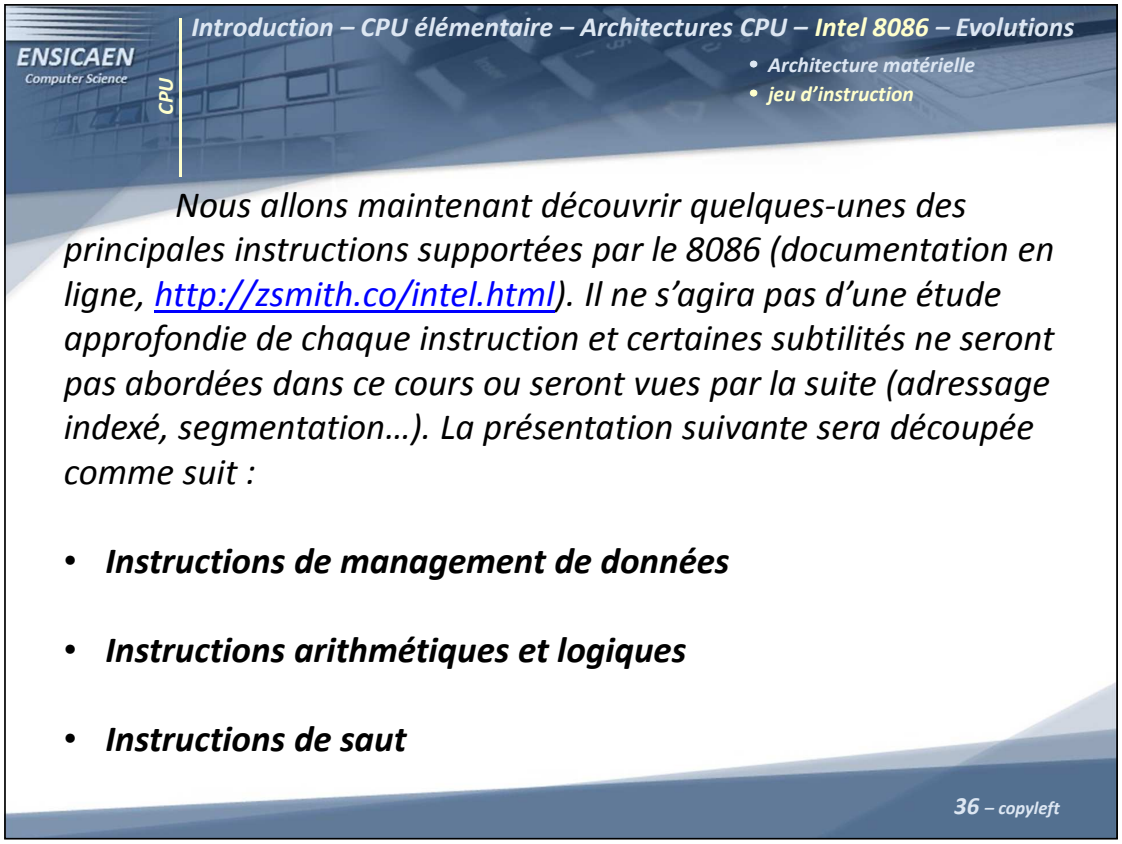
Original 8086 Instruction set

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

34 – copyleft


ENSICAEN
Computer Science
CPU
Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
• Architecture matérielle
• jeu d'instruction
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
35 – copyleft


ENSICAEN
Computer Science
CPU
Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
• Architecture matérielle
• jeu d'instruction
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
36 – copyleft

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
```

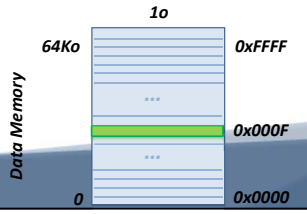
- ***Adressage immédiat :*** *affectation d'une constante dans un registre. Le déplacement d'une constante vers la mémoire est également possible.*

```
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.*

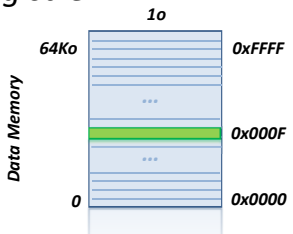
- ***Adressage direct :*** *déplacement de données du CPU vers la mémoire ou vice versa. L'adresse de la case mémoire à manipuler est directement passée avec l'opcode de l'instruction.*

```
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)
```

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction

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/m6 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.

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction

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.

42 – copyleft

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction

- ***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
```

43 – copyleft

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction

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

Dr J. Y. Haggège

44 – copyleft

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction

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)
```

45 – copyleft

ENSICAEN Computer Science — CPU

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

- Architecture matérielle
- jeu d'instruction

- ***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
}
```

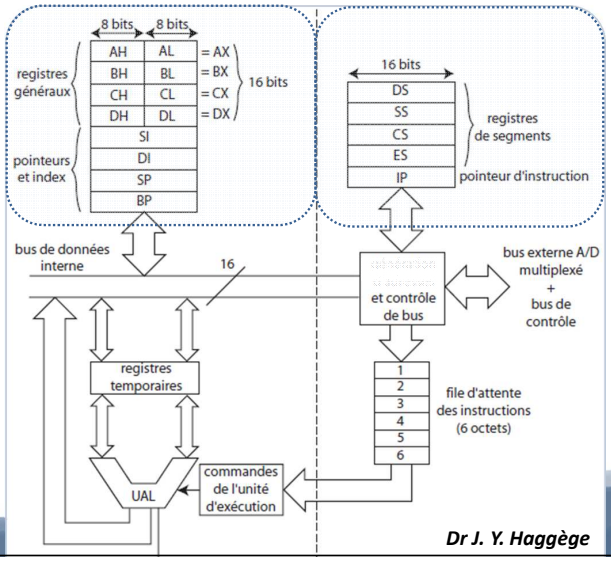
```
main:   call    fctTest     ; IP = adresse fctTest (19cy relatif)
        jmp     main        ; IP = adresse main (15cy)
        ;other code
        ;...
        ;...
fctTest: ; user code
        ;...
        ;...
        ;...
        ret                 ; IP = adresse jmp dans le main
         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

46 – copyleft

ENSICAEN Computer Science — CPU

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

Rappelons les registres rencontrés sur architecture 8086 de Intel :



Dr J. Y. Haggège

47 – copyleft

ENSICAEN Computer Science — CPU

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

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 :

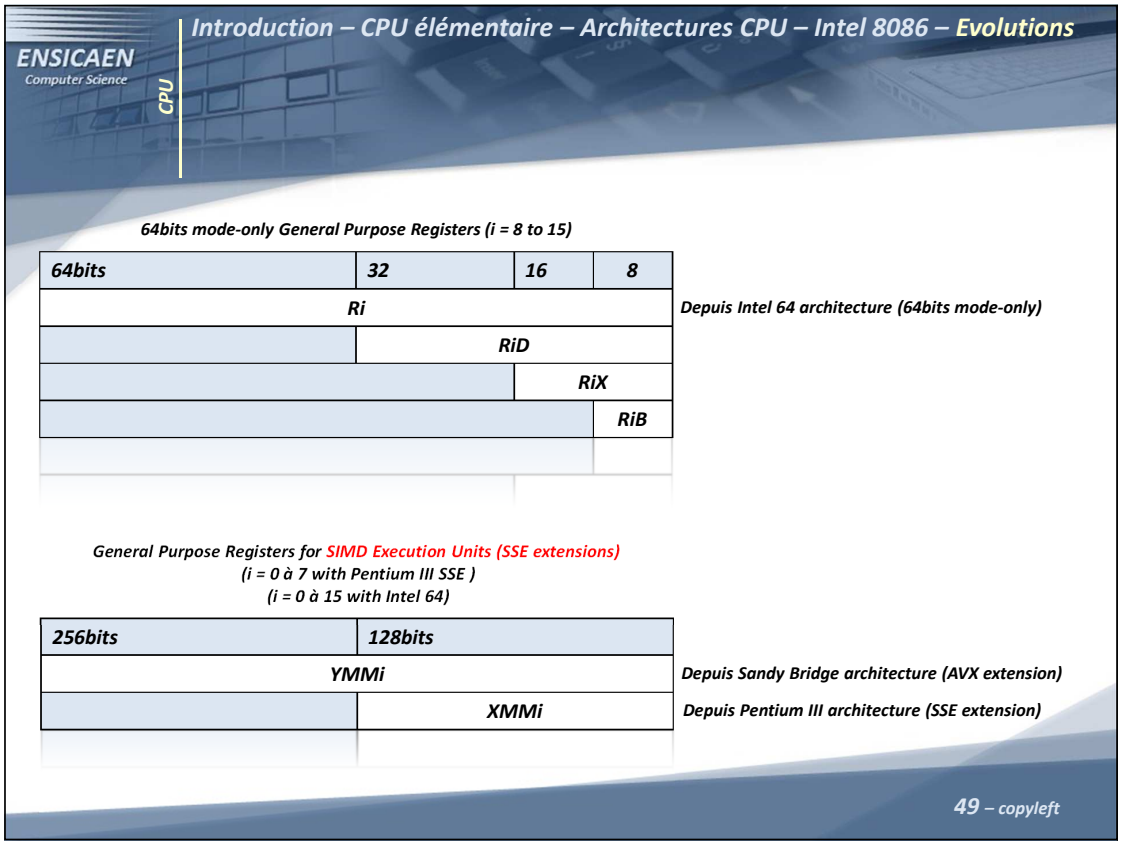
General Purpose Registers (i = A, B, C, D)

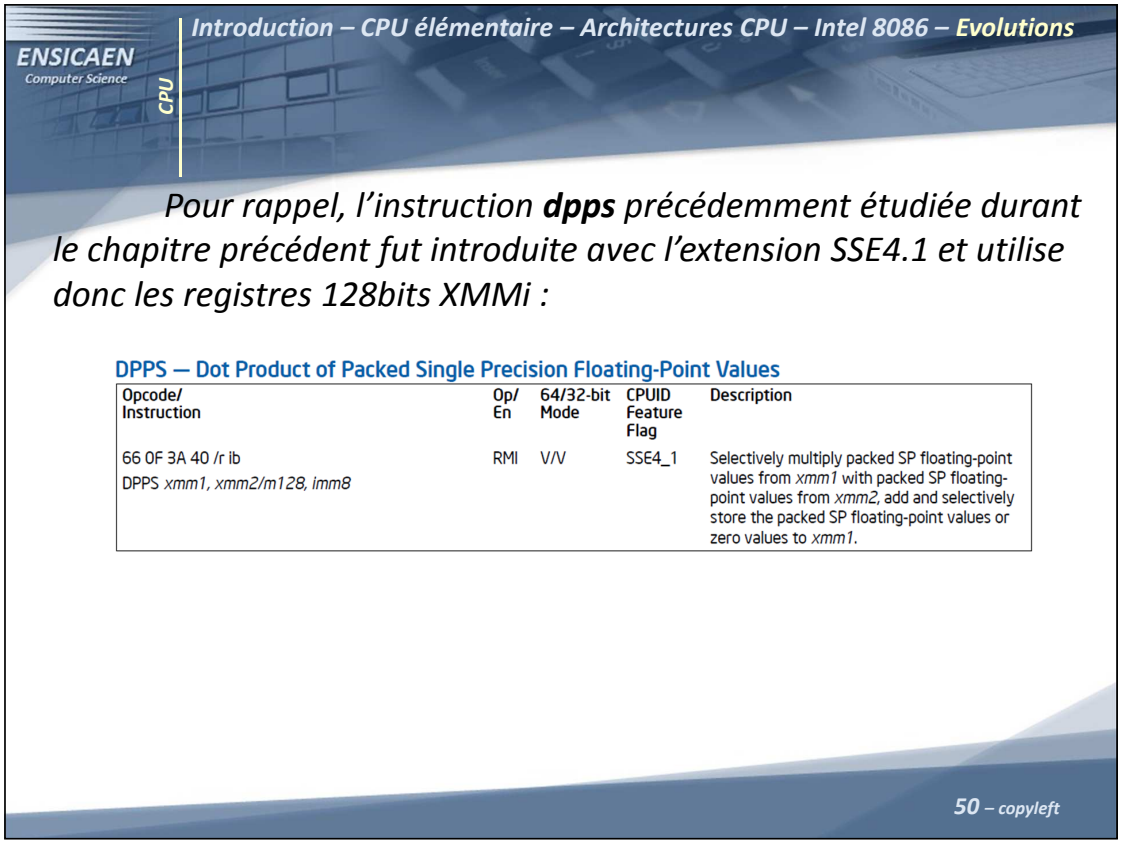
64bits	32	16	8	
RiX				Depuis Intel 64 architecture (64bits mode-only)
	EiX			Depuis 80386 architecture (E = Extended)
		iX		Depuis 8086 architecture
		iH	iL	

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

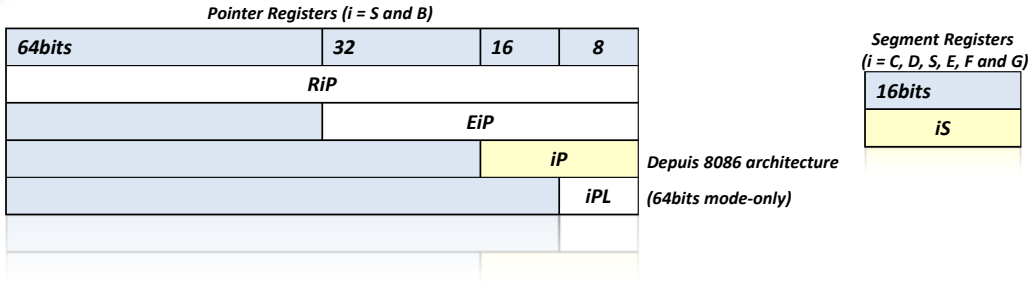
80bits	64bits	
STi		Depuis 80486 architecture (x87 extension)
	MMXi	Depuis Pentium MMX architecture

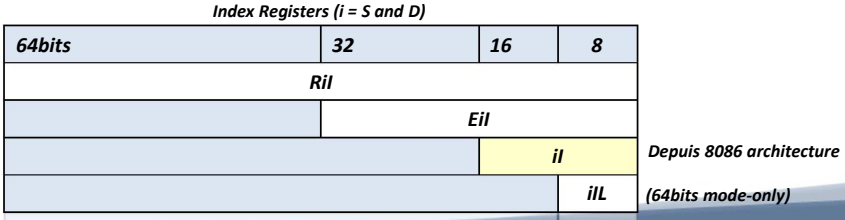
48 – copyleft

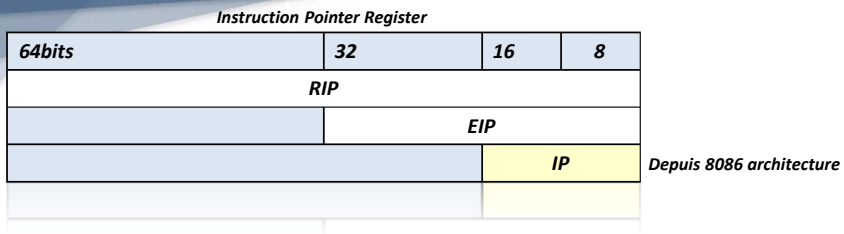

ENSICAEN
Computer Science
CPU
Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
64bits mode-only General Purpose Registers (i = 8 to 15)
64bits
32
16
8
Ri
RiD
RiX
RiB
Depuis Intel 64 architecture (64bits mode-only)
General Purpose Registers for SIMD Execution Units (SSE extensions)
(i = 0 à 7 with Pentium III SSE)
(i = 0 à 15 with Intel 64)
256bits
128bits
YMMi
XMMi
Depuis Sandy Bridge architecture (AVX extension)
Depuis Pentium III architecture (SSE extension)
49 – copyleft


ENSICAEN
Computer Science
CPU
Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions
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.
50 – copyleft

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







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 …

ENSICAEN
Computer Science
CPU
Introduction – CPU élémentaire – Architectures CPU – Intel 8086 – Evolutions

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

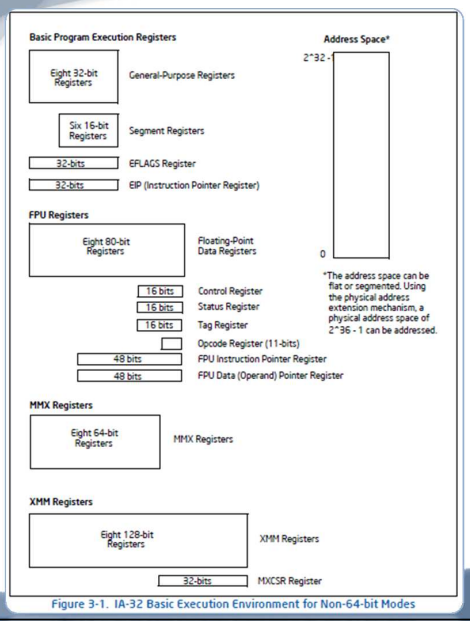


Figure 3-1. IA-32 Basic Execution Environment for Non-64-bit Modes

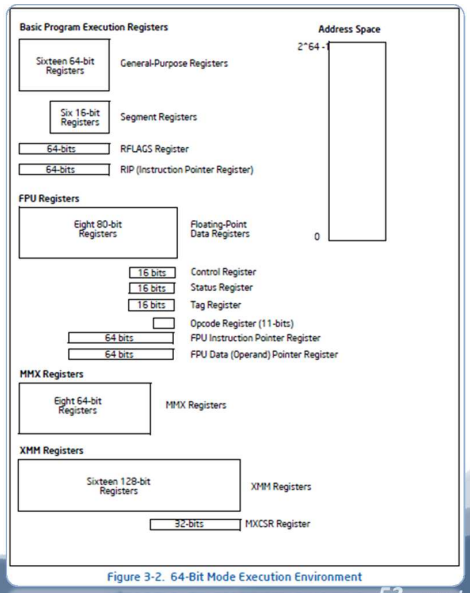


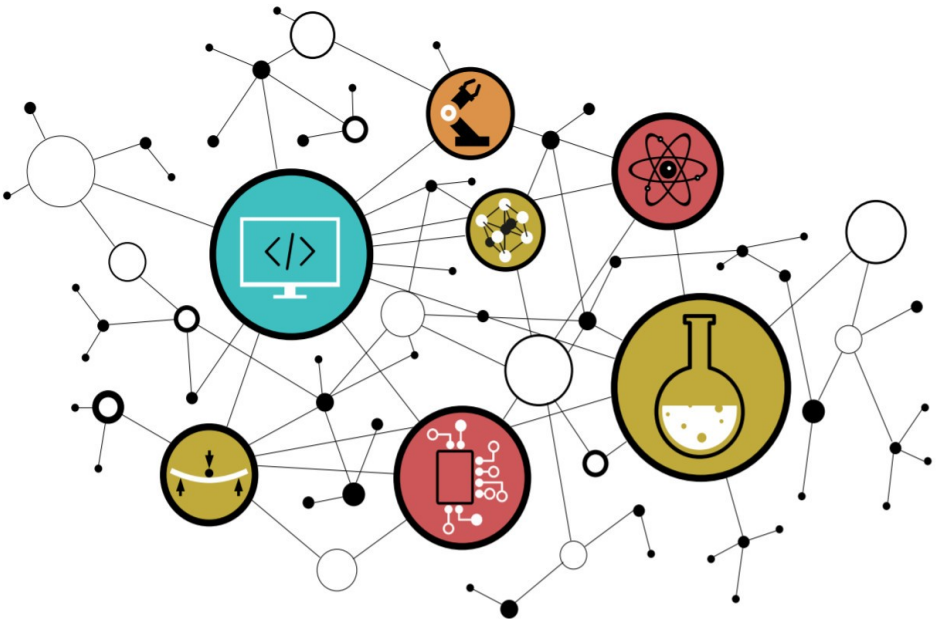
Figure 3-2. 64-Bit Mode Execution Environment

http://www.intel.com
53 – copyleft

ENSICAEN
Computer Science

Merci de votre attention !

MÉMOIRES



Chapitre 6

Mémoire

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

Une mémoire peut être classée selon différents critères comme la volatilité (volatile ou non), l'accessibilité (accès aléatoire RAM ou séquentiel), l'adressage (par octet ou associatif), la capacité, les performances … En voici quelques uns.

Mémoire volatile : mémoire ne conservant pas les informations stockées lorsqu'elle est mise hors tension.

Ex : SDRAM, DDRAM, mémoire cache, registres du processeur.

Mémoire non-volatile : mémoire conservant les données même hors tension.

Souvent des médias de stockage de masse (HDD, SSD, SD-card, DVD, Blu-Ray, …), mais aussi la mémoire du BIOS ou l'EEPROM des micro-contrôleurs.

Mémoire morte ou ROM (Read-Only Memory) : mémoire non-volatile accessible uniquement en lecture, elle est donc pré-programmée.

Ex. : mémoire du BIOS, anciens systèmes embarqués, …

Mémoire vive ou RAM (Random-Access Memory) : mémoire généralement volatile accessible en lecture et écriture.

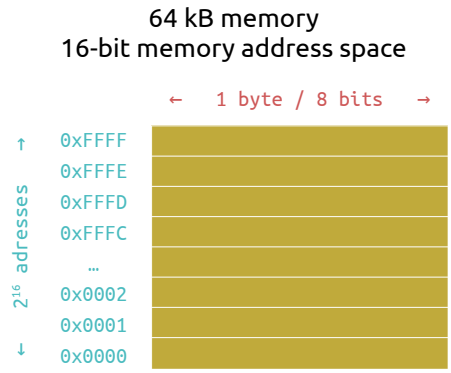
Ex : mémoire principale (SDRAM, DDRAM) d'un ordinateur ou micro-contrôleur.

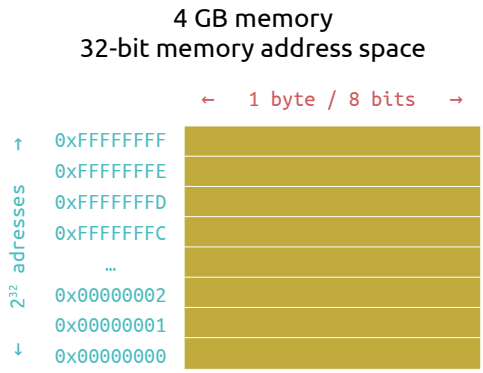
Attention : beaucoup d'ambiguïtés autour de ces termes.

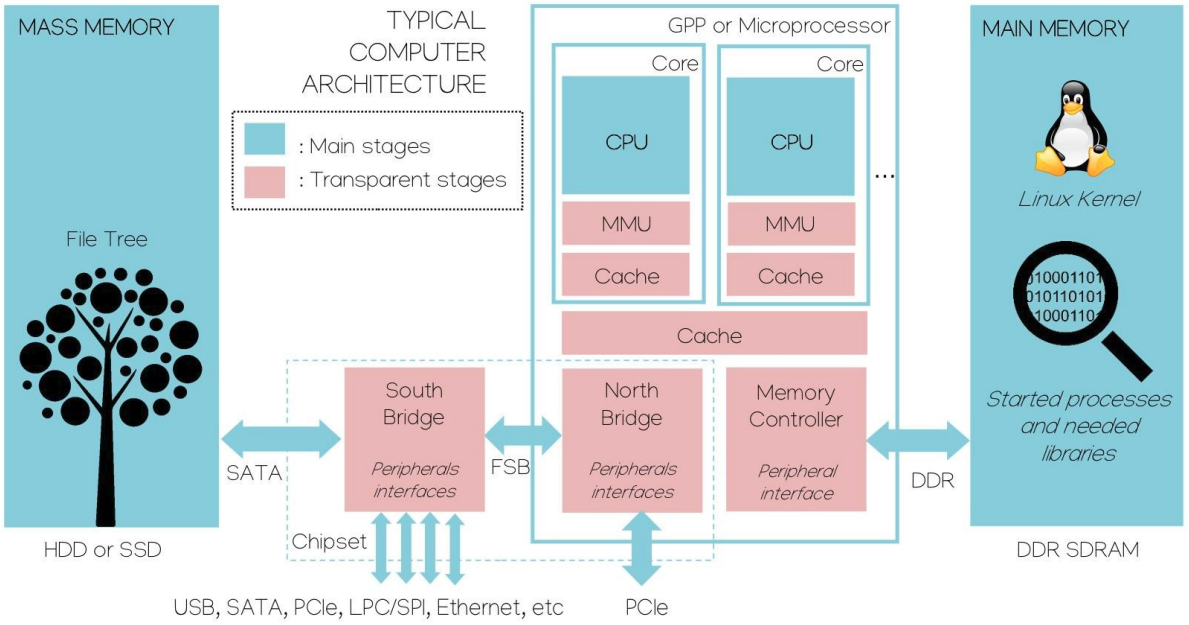
Dans ce cours nous parlerons de mémoire de stockage de masse (disque dur, …) et de mémoire principale (RAM).

Mémoire adressable par octet : beaucoup de familles de mémoire d'ordinateur ou de processeur embarqué (MCU, DSP, SoC, …) sont adressables par octets.

À chaque adresse mémoire correspond 1 octet.







Analyses de circuit électroniques : Deus Ex Silicium

Parfait pour l'analyse d'ordinateurs sur-mesure ;)

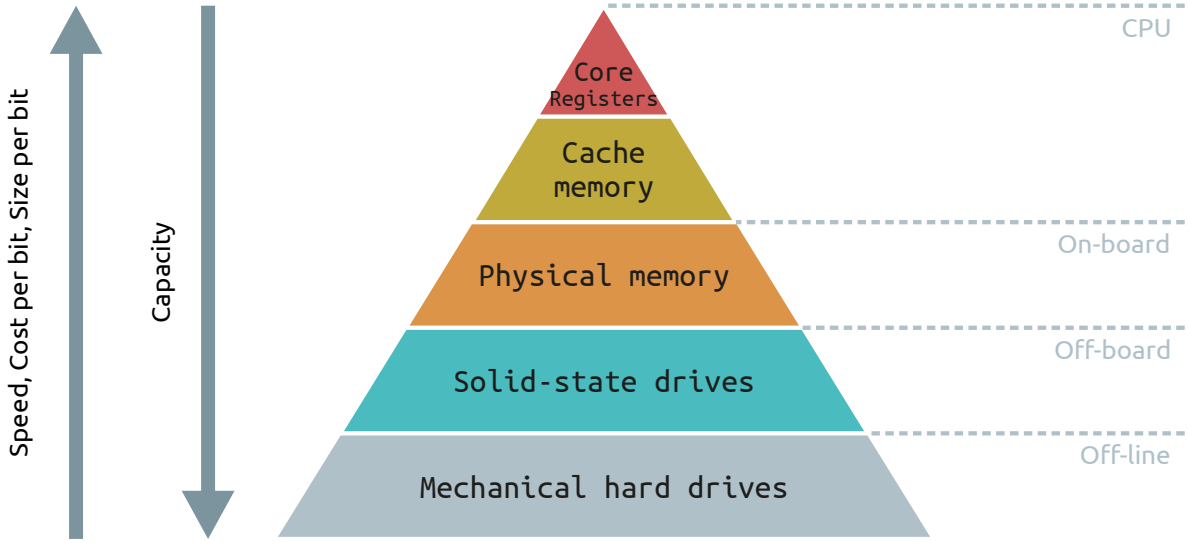
Processeur, SSD, RAM, carte mère, South bridge, alimentation ...

Démontage et remontage d'une XBox :

XBox Series X - Décorticage intégral, mesures et analyses

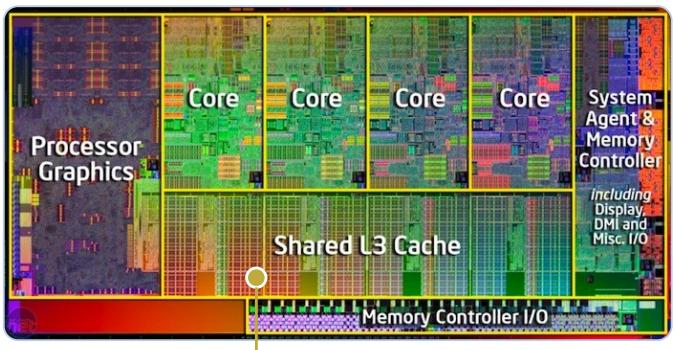
Démontage définitif d'une Nintendo Switch :

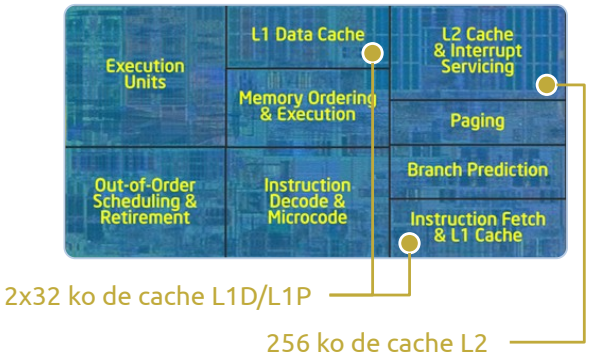
Au plus profond des entrailles de la Nintendo Switch OLED



Le modèle en couche peut être affiné en fonction des différentes technologies d'intégration. Par exemple, l'empreinte sur silicium de 32 ko de mémoire cache L1 n'est pas la même que pour 32 ko de cache L2 ou encore L3.

Exemple d'un GPP Intel Sandy Bridge (die complet à gauche, zoom sur le core à droite).





STOCKAGE DE MASSE

Supports physiques
Volumes logiques
Système de fichiers

Les mémoires de stockage de masse sont des **mémoires non-volatiles**, destinées à stocker de **grandes quantités de données** sur le long terme y compris **en l'absence d'alimentation** (« grandes » étant à mettre dans le contexte historique ou applicatif).

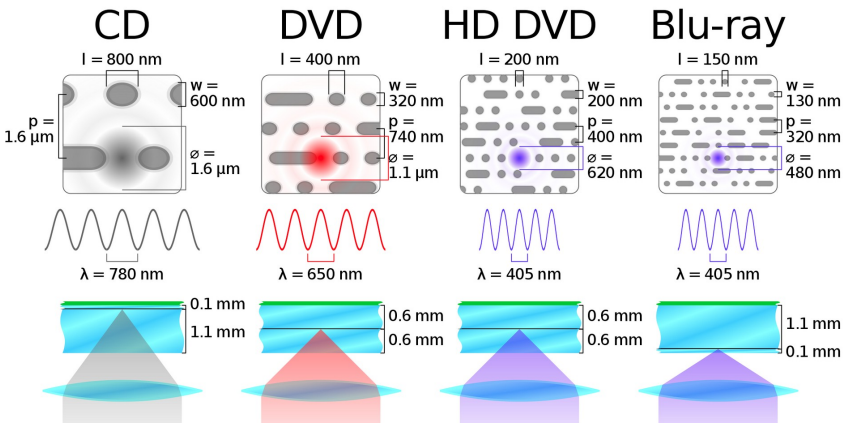
Les mémoires modernes utilisent une représentation logique des données sous forme de fichiers (voir « *file system* »), mais ça n'a pas toujours été le cas.

Les principales caractéristiques recherchées sont la capacités de stockage, la vitesse de lecture/écriture, la durée de vie des données et le coût.

La fonction de **stockage de masse** a été effectué par plusieurs technologies, dont les plus connues sont le stockage magnétique, optique ou électronique.

Les supports optiques sont généralement des **supports amovibles** et généralement **en lecture seule** (distribution de logiciels, musiques, films, ...).

La gravure sur disque se fait par laser, la densité des informations dépendant de la longueur d'onde, du diamètre et de l'angle d'ouverture du-dit laser.



Comparaison des caractéristiques physiques des quelques support optiques.

CD-ROM : 650-700 MB
Licence Sony et Philips

DVD : 4,7 – 8 GB
Licence DVD forum

Blu-ray : 25-50 GB
Blu-ray UHD : 100 GB
Licence Sony

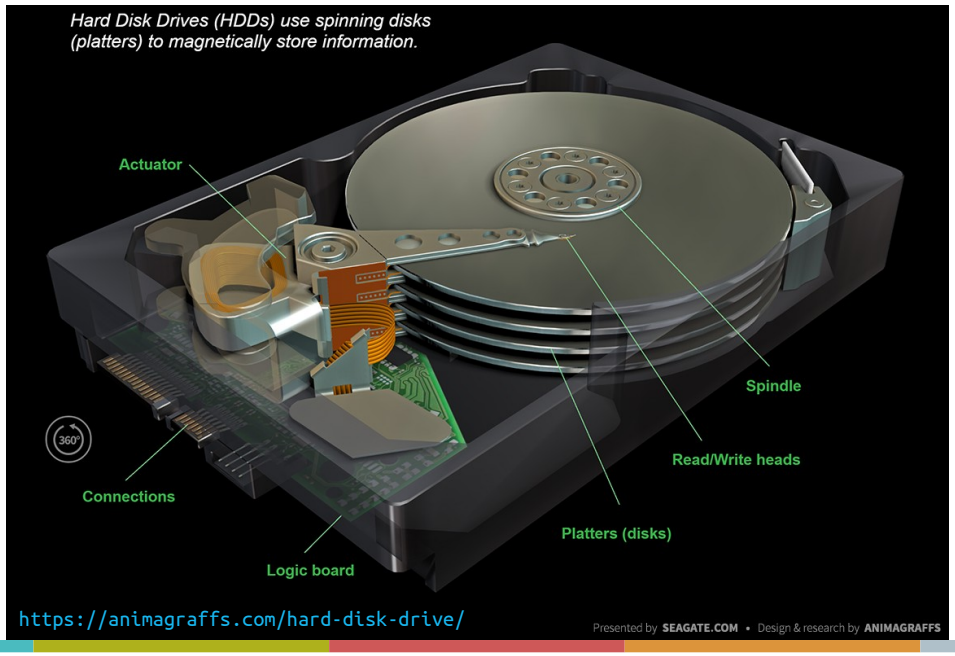
Les disquettes (*floppy disks*) et **disques durs (HDD, *Hard Drive Disk*)** utilisent le stockage magnétique.

Les HDD sont moins rapides que les SSD, mais coûtent également bien moins cher à capacité égale.

Floppy disks
8-inch ~243 kB ; $5^{1/4}$-inch = 360 kB ; $3^{1/2}$-inch = 1.44 MB

Hard Drive Disk (HDD)
3.5-inch / 2.5-inch

Disque dur mécanique (HDD)

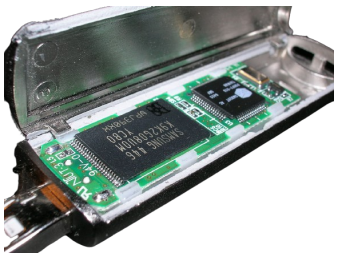


Supports électroniques

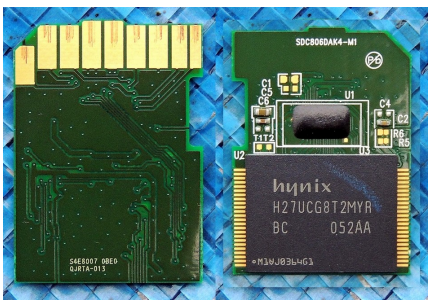
Les **EEPROM** utilisent de la circuiterie basique pour stocker des charges électriques. La technologie EEPROM la plus commune est la **mémoire Flash** (NAND et NOR), qui a un temps constant d'accès à l'information.

Les clés USB, cartes SD, **SSD (*Solid-State Drive*)** utilisent aussi une technologie Flash.

120 GB, 2.5-inch Samsung SSD



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

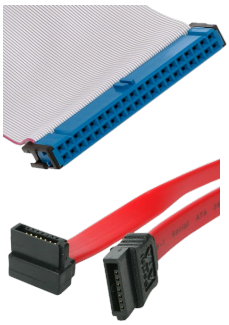


8-GB SD Card
Internal circuit

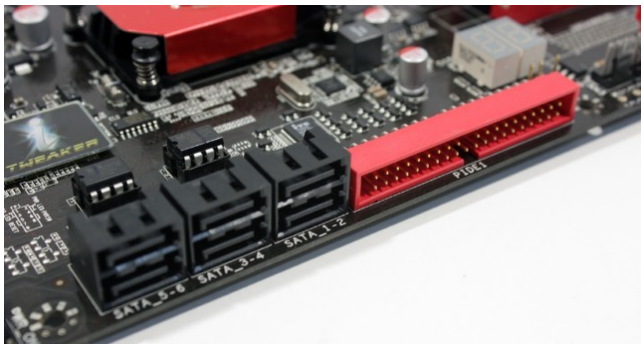
Fin 80-début 2000, les disques durs et autres lecteurs utilisaient la norme **Parallel-ATA** (ou **IDE (*Integrated Drive Electronics*)**) pour communiquer avec la carte mère.

Elle a été remplacée début des années 2000 par **S-ATA (*Serial ATA*)**, toujours utilisée.

Avantages : débit jusqu'à 6 Gb/s (norme III), moins de connexions que P-ATA et *hot-plug*.

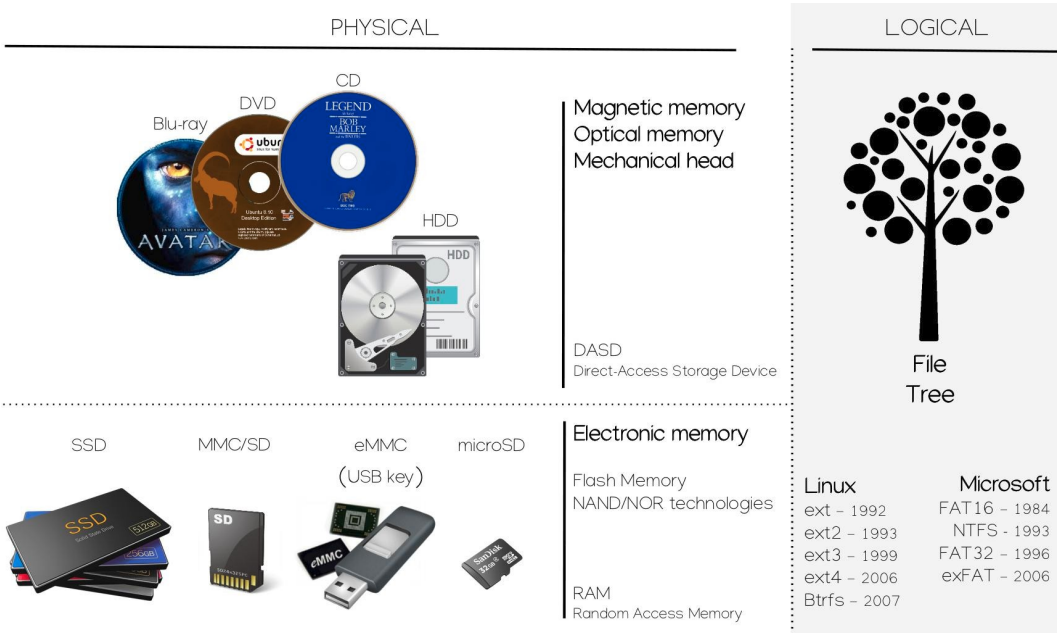


Câbles P-ATA (haut)
et S-ATA (bas)



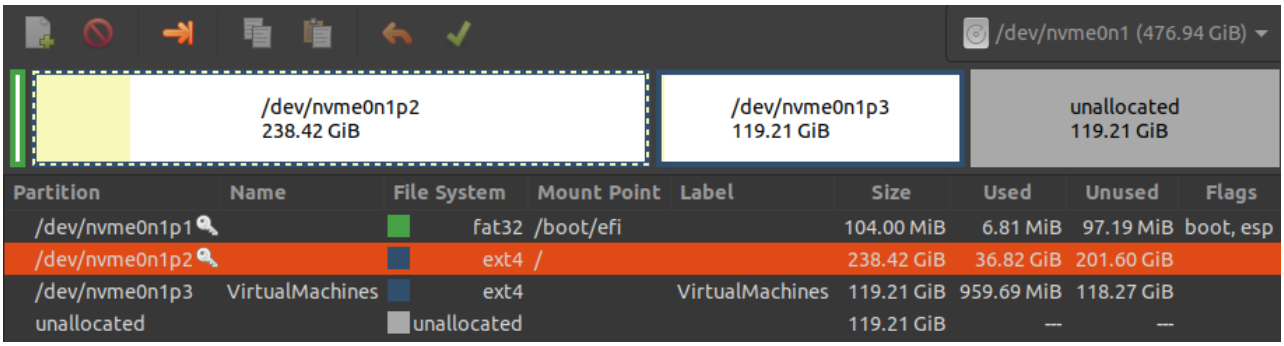
6 ports S-ATA (à gauche)
Et 1 port P-ATA (à droite)

SSD format 2.5-inch (haut)
et format M.2 (bas)



Un **volume physique** (HDD, clé USB, …) peut être découpé en plusieurs **volumes logiques (partitions)**.

C'est généralement le cas des ordinateurs en dual-boot, qui vont avoir une partition pour chaque OS voire aussi une partition partagée entre les deux OS, pour les données.



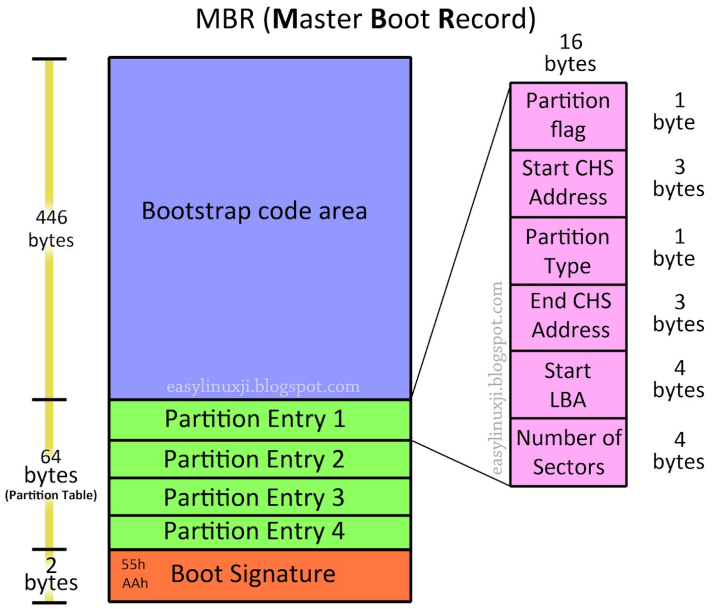
Affichage des partitions d'un disque avec gparted (Ubuntu)

Avec plusieurs partitions sur un même support physique, il faut qu'un système soit capable d'identifier toutes les partitions d'un disque.

Pour cela, on trouve un **table des partitions** au début du volume physique et en espace non-partitionné (donc hors de toute volume logique).

Il s'agit d'une zone purement binaire (pas de fichier) pouvant être au format MBR (Master Boot Record) ou GPT (GUID Partition Table).

C'est cette zone que le BIOS (ou UEFI) ira lire pour détecter la partition à démarrer, ou que l'OS va lire pour afficher l'image de la page précédente.



Le MBR (Master Boot Record) est un vieux standard (1983) développé par PC DOS 2.0 (IBM).

Désigne maximum 4 partitions.

Volumes physiques de taille < 2 TiB (2^{32} x 512-Byte).

Ces limitations ont fait diminuer son usage dans l'informatique, mais restent un avantage pour les systèmes embarqués.

Le GUID Partition Table (GPT) est apparu à la fin des années 90, développé par Intel pour s'allier avec son UEFI (remplaçant du BIOS).

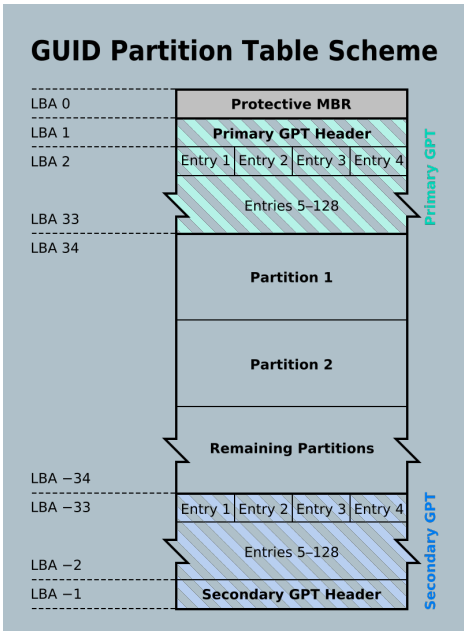
Vu les inconvénients du MBR, le GPT a rapidement remplacé son ancêtre dans l'informatique professionnelle et personnelle.

Maximum 128 partitions

Maximum 8 ZiB (2^{64} x 512-Byte sectors)

CRC32 checksum

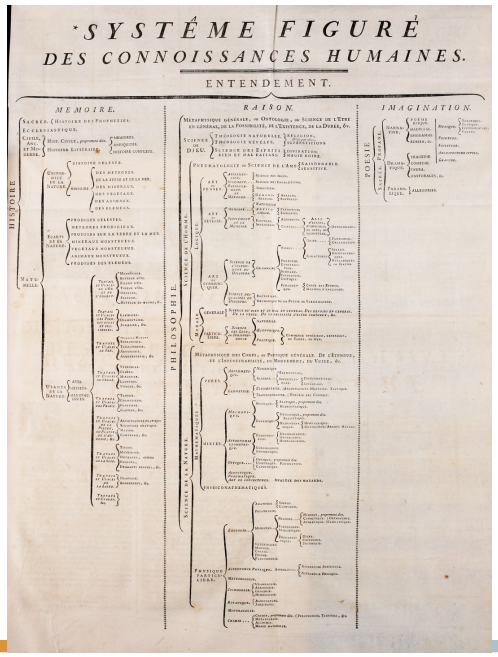
GUID = Globally Unique Identifier



Au sein d'une partition, les fichiers sont organisés selon un **système de fichiers (FS, *File System*)**.

C'est ce qui permet à l'utilisateur d'observer ses fichiers sous forme d'une arborescence.

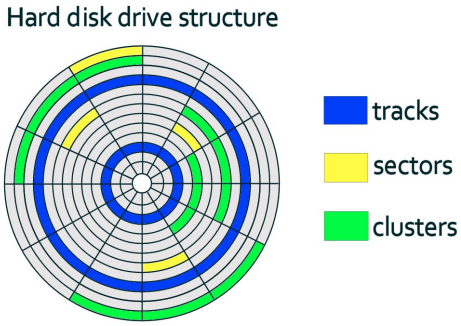
Archivage de cartons contenant jusqu'à 2000 cartes perforées (NARA, 1959).



« Système figuré des connaissances humaines » ou « Arbre de Diderot et d'Alembert » (1751-1772)

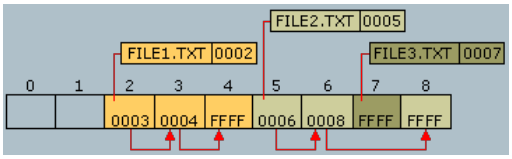
Contrairement aux mémoires adressables par octet, les médias de stockage de masse sont généralement adressables par blocs.

Par exemple, un disque dur (HDD) est découpé en secteurs de 512 octets. Un fichier se trouve vraisemblablement réparti sur plusieurs secteurs, et pas forcément contigus.



Le *file system* a donc aussi le rôle de présenter à l'utilisateur des fichiers entiers (alors qu'ils sont morcelés sur le support physique) et regroupés par dossiers (alors qu'ils sont disséminés sur tout le support physique).

FAT (1977, pour Windows) est un *file system* extrêmement simple, travaillant par clusters (groupes de secteurs). Il existe une table liant les noms de fichiers à l'index du premier cluster de chaque fichier. Chaque cluster occupé contient une partie du fichier et l'indice vers le cluster suivant.



Depuis sa naissance, de nombreuses versions sont apparues (FAT16, FAT32, exFAT, …).

Déprécié par Windows au profit de NTFS, il reste très répandu pour les médias amovibles.

https://www.pctechguide.com/hard-disks/file-systems-fat-fat8-fat16-fat32-and-ntfs-explained

Le principal *file system* utilisé par Windows est **NTFS (New Technology File System)** arrivé en 1993 avec Windows NT 3.1.

Pendant longtemps (avant Windows 10), les OS de Microsoft n'ont su gérer que les *file systems* FAT et NTFS.

Les *file systems* conçus pour Linux sur ordinateur sont ceux de la famille **ext** (**ext** en 1992, **ext2**, **ext3** et **ext4** depuis 2006), même si Linux a toujours su supporter de nombreux *file systems* (y compris FAT et NTFS).

Depuis l'intégration en 2016 de WSL (*Windows Subsystem for Linux*) dans Windows 10, l'OS de Redmond supporte le *file system* ext4 (même si ce n'est pas si simple).

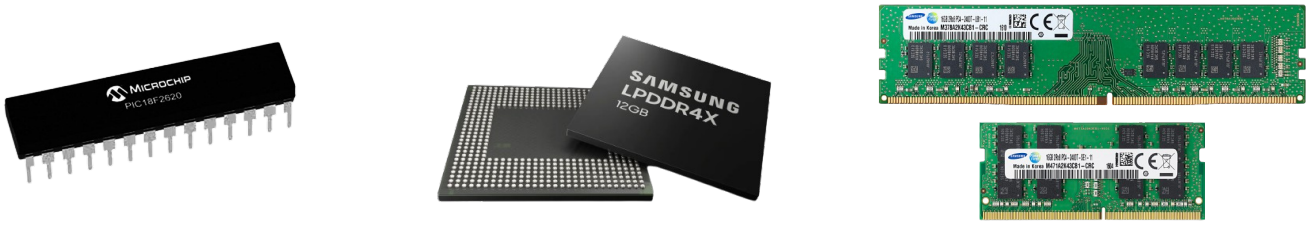
MÉMOIRE PRINCIPALE

Quelques technologies
MMU – Memory Management Unit
Unité de segmentation
Unité de pagination
Virtualisation

Voir annexes

La **mémoire principale (*main memory*)** ou parfois mémoire de travail est la mémoire dans laquelle se trouvent les informations en cours d'utilisation par le processeur.

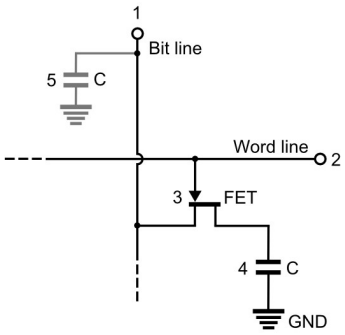
Ces informations peuvent être des données seulement (cas d'un MCU) ou des données ET des instructions (cas d'un smartphone ou ordinateur).



Mémoire principale dans un MCU (en interne), smartphone (chip) et ordinateur (barrette).

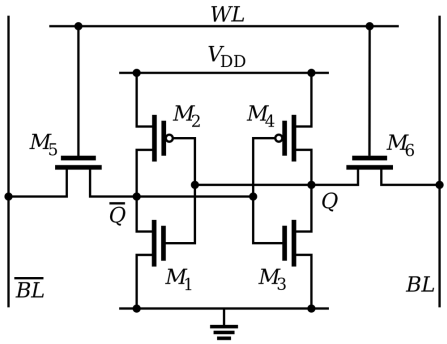
Technologiquement, la mémoire principale est une **RAM (*Random Access Memory*)** et plus précisément d'une **DRAM (Dynamic RAM)**.

La DRAM doit être régulièrement rafraîchie (qq ms) à cause du courant de fuite de ses pico-condensateurs. Utilisée pour la mémoire d'ordinateur. Faible empreinte silicium mais plus lente que la SRAM (voir mém. cache).



DRAM
1 bit requires 1 transistor
and 1 pico-capacitor

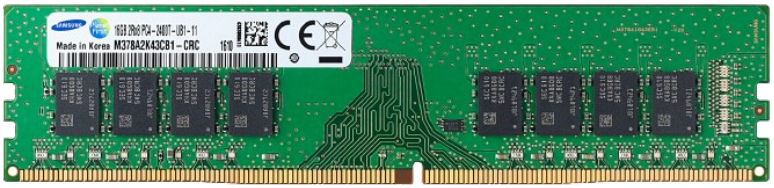
SRAM
1 bit requires 6
CMOS transistors



Les RAM utilisées aujourd'hui fonctionnent en DDR5 SDRAM (5*th generation of Double Data Rate Synchronous DRAM*).

Les premières mémoires en DDR (1998) donnaient une bande passante de 1600 Mo/s pour une fréquence d'horloge de 100 MHz (soit 200 MTransferts/s).

Le dernier standard DDR5 (2021, DDR5-7200 de Samsung) donne un débit théorique de 57000 Mo/s avec une horloge cadencée à 3,6 GHz (soit 7200 MTransferts/s).



Samsung 16 GB DDR4 SDRAM

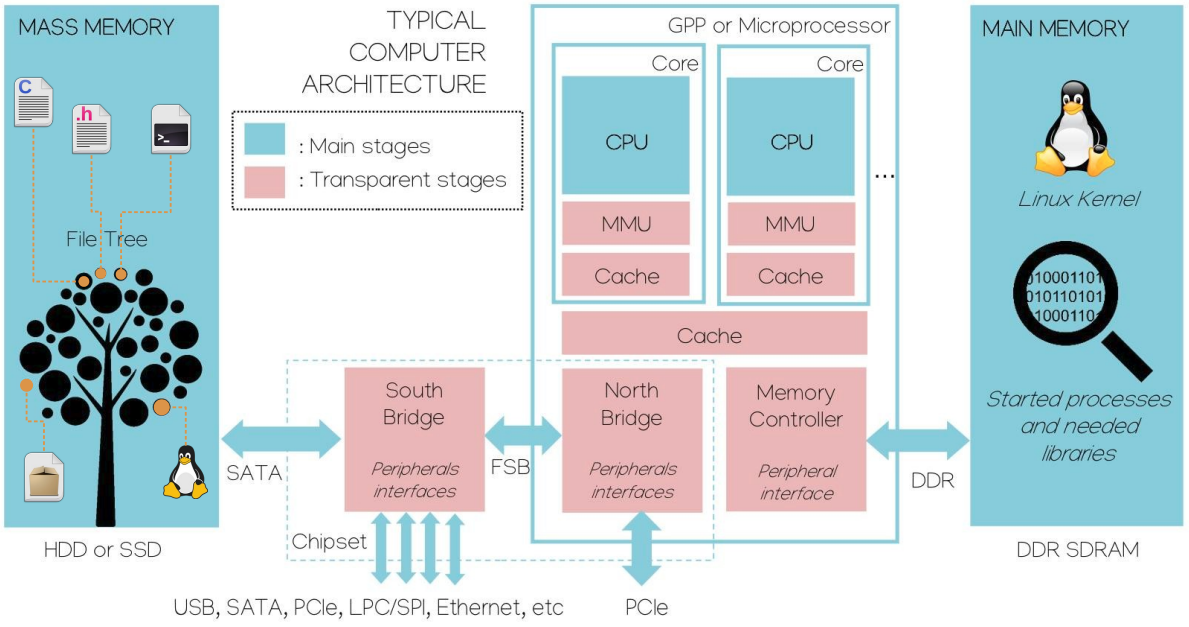
Crucial SO-DIMM 16 GB DDR4 SDRAM

EXÉCUTION D'UN PROGRAMME

Segments mémoire
Segment de code
Pile / Stack
Tas / Heap

EXÉCUTION D'UN PROGRAMME

Que se passe-t-il quand on lance un exécutable ?



Pour rappel, sous Linux, le fichier exécutable est un fichier **binaire** au **format ELF**.
C'est aussi le cas des librairies statiques, dynamiques ou autres objets partagés.

Ce fichier contient différentes sections, dont :

- `.text` : contient le code sous forme binaire ;
- `.data` : les variables globales et statiques initialisées ;
- `.bss` : idem, mais non-initialisées ;
- `.rodata` : les variables en lecture seule (les constantes).

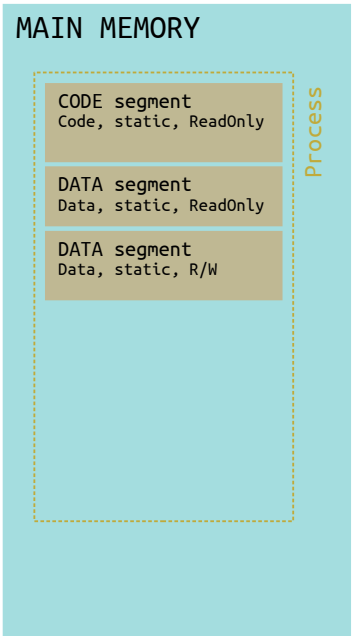
```
dboudier:toolchain$ objdump -s objdump-minimal.o

objdump-minimal.o:     file format elf64-x86-64

Contents of section .text:
 0000 f30f1efa 554889e5 4883ec10 488d0500  ....UH..H...H...
 0010 00000048 8945f848 8b45f848 89c2488d  ...H.E.H.E.H..H.
 0020 35000000 00488d3d 00000000 b8000000  5....H.=........
 0030 00e80000 0000b800 000000c9 c3        .............
Contents of section .data:
 0000 426f6e6a 6f757220 6c65206d 6f6e6465  Bonjour le monde
 0010 0a00                                 ..
Contents of section .rodata:
 0000 48656c6c 6f20576f 726c640a 00257325  Hello World..%s%
 0010 7300                                 s.
```

Au démarrage de l'application, ces sections passeront de la mémoire de stockage de masse (disque dur) vers la mémoire principale (RAM).

Lors du lancement d'un programme (fichier situé dans la mémoire de masse), le système d'exploitation ne charge en mémoire que les sections essentielles.

La section `.text` devient le **segment de code**.
Et `.data`/`.bss`/`.rodata`, des **segments de données**.

Cette phase de chargement, c'est l'**allocation statique** : la taille des segments placés en mémoire est immuable (fixée par la chaîne de compilation).



NB : ceci est une représentation logique de ce qu'il se passe en mémoire virtuelle.

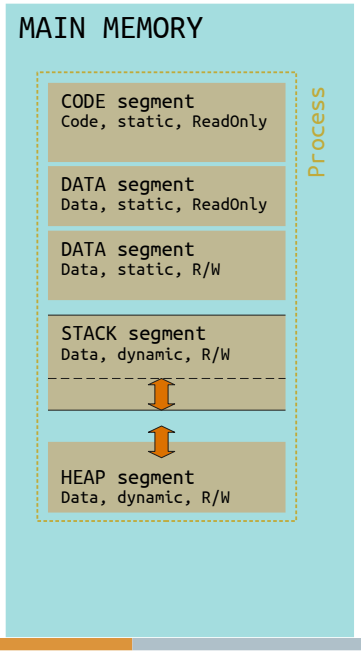
Pour chaque **processus** (programme en cours), on trouve deux autres segments, dédiés à l'**allocation dynamique**.

Le **segment de pile (*stack*)** contient les variables locales des fonctions et sauvegardes de contextes.

La *stack* est réservée aux variables locales de taille modeste.

Le **segment de tas (*heap*)** contient les variables allouées dynamiquement : fonctions `malloc()`, `free()`, …

Très utile pour les tableaux dynamiques par exemple, mais nécessite la présence d'une MMU.



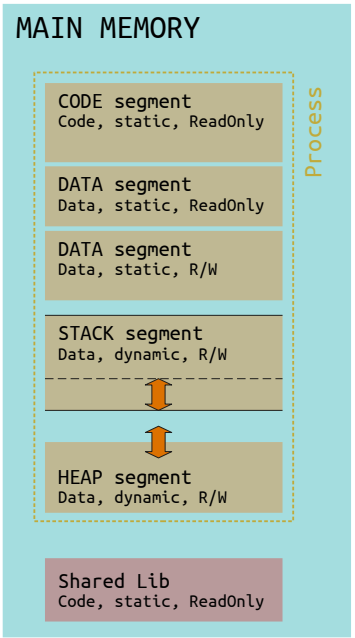
NB : ceci est une représentation logique de ce qu'il se passe en mémoire virtuelle.

Les processus peuvent utiliser d'autres zones de la mémoire principale.

On peut citer par exemple les librairies partagées (`glibc` sous Linux) qui sont accessibles par n'importe quel processus (pour un `printf()` par exemple).

Elle sont utilisées grâce à un *linker* dynamique.

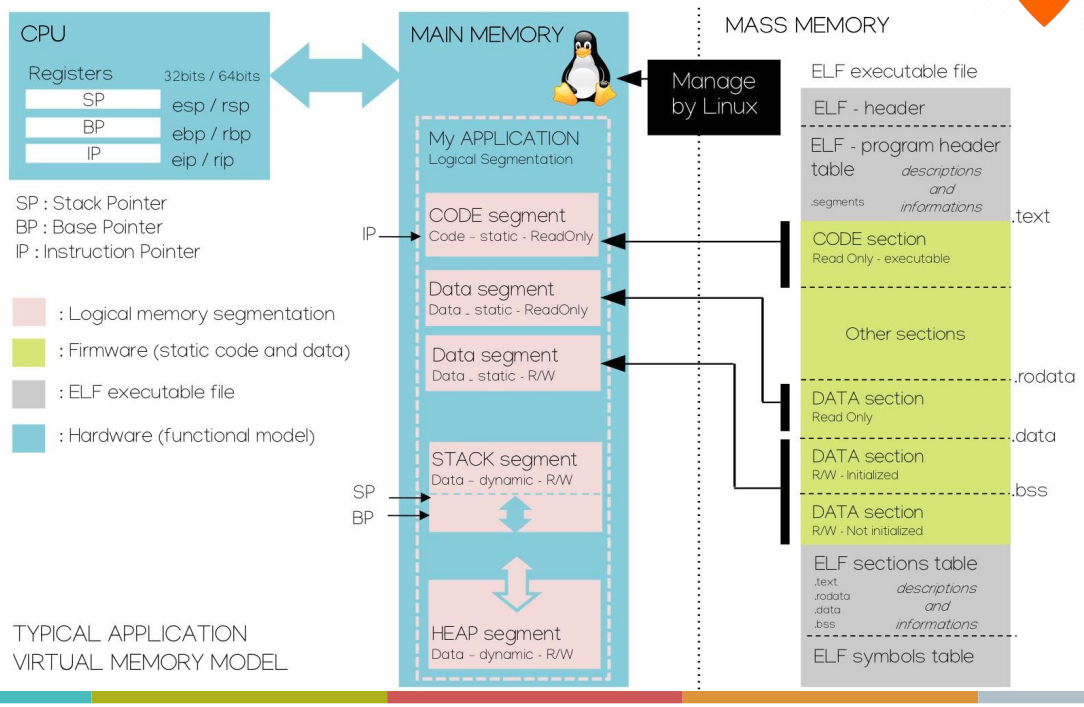
On compte aussi des zones de communication entre les processus (IPC, *Inter-Process Communication*).



NB : ceci est une représentation logique de ce qu'il se passe en mémoire virtuelle.

EXÉCUTION D'UN PROGRAMME

Utilisation de la mémoire principale par un processus



EXÉCUTION D'UN PROGRAMME

Utilisation de la mémoire principale par un processus

```
dboudier:toolchain$ ps -Af
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 09:41 ?        00:00:01 /sbin/init splash
root           2       0  0 09:41 ?        00:00:00 [kthreadd]
root           3       2  0 09:41 ?        00:00:00 [rcu_gp]
dboudier   24039    1292  0 18:47 ?        00:00:00 /usr/libexec/tracker-store
root       24186       2  0 18:47 ?        00:00:00 [kworker/14:0]
dboudier   24190   12732 99 18:47 pts/0    00:00:06 ./a.out
dboudier   24193   12724  0 18:47 pts/1    00:00:00 bash
dboudier   24211   24193  0 18:47 pts/1    00:00:00 ps -Af
dboudier:toolchain$ sudo cat /proc/24190/maps
[sudo] password for dboudier:
55faba9de000-55faba9df000 r--p 00000000 08:01 21760669                   /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba9df000-55faba9e0000 r-xp 00001000 08:01 21760669                   /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba9e0000-55faba9e1000 r--p 00002000 08:01 21760669                   /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba9e1000-55faba9e2000 r--p 00002000 08:01 21760669                   /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55faba9e2000-55faba9e3000 rw-p 00003000 08:01 21760669                   /home/dboudier/Documents/FISE/1A_Architectures_des_ordinateurs/tp_prof/work/toolchain/a.out
55fabaa07000-55fabaa28000 rw-p 00000000 00:00 0                          [heap]
7fa9c05f4000-7fa9c0616000 r--p 00000000 103:02 6817925                   /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c0616000-7fa9c078e000 r-xp 00022000 103:02 6817925                   /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c078e000-7fa9c07dc000 r--p 0019a000 103:02 6817925                   /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c07dc000-7fa9c07e0000 r--p 001e7000 103:02 6817925                   /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c07e0000-7fa9c07e2000 rw-p 001eb000 103:02 6817925                   /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fa9c07e2000-7fa9c07e8000 rw-p 00000000 00:00 0
7fa9c07fe000-7fa9c07ff000 r--p 00000000 103:02 6817918                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c07ff000-7fa9c0822000 r-xp 00001000 103:02 6817918                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c0822000-7fa9c082a000 r--p 00024000 103:02 6817918                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c082b000-7fa9c082c000 r--p 0002c000 103:02 6817918                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c082c000-7fa9c082d000 rw-p 0002d000 103:02 6817918                   /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fa9c082d000-7fa9c082e000 rw-p 00000000 00:00 0
7ffe5f5e8000-7ffe5f609000 rw-p 00000000 00:00 0                          [stack]
7ffe5f7bc000-7ffe5f7c0000 r--p 00000000 00:00 0                          [vvar]
7ffe5f7c0000-7ffe5f7c2000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

`ps -Af`
Affiche la liste des processus (tronquée ici)
`PID` = Process Identifier ; `CMD` = nom de l'exécutable

`cat /proc/<PID>/maps`
Affiche l'utilisation de la mémoire par le processus <PID>

Allocation statique
(issus des sections du fichier ELF)

Librairie partagée (libc)
Linker dynamique (ld)

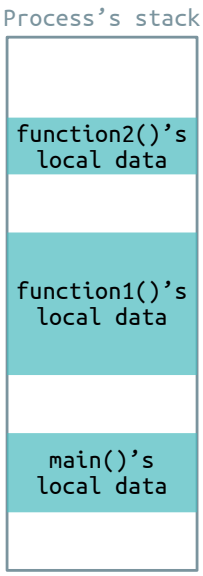
Allocation dynamique
(tas/heap et pile/stack)

La **stack (ou pile)** d'un processus (ou d'un thread) contient les variables locales des fonctions qui le composent.

Chaque fonction appelée viendra « poser » ses données en sommet de pile, au dessus des autres fonctions.

Cette pile se « videra » progressivement, à chaque retour de fonction.

Ce comportement est celui d'une pile **LIFO (Last In, First Out)**.



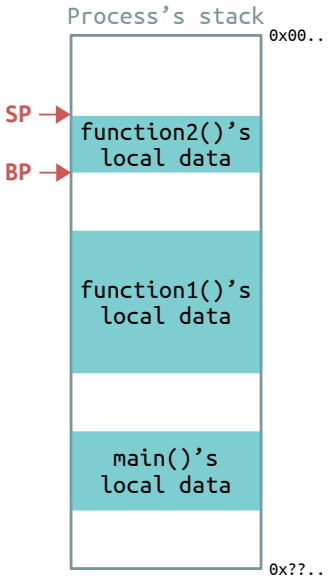
Afin de manipuler la pile, chaque CPU possède deux registres nommés **Base Pointer BP** et **Stack Pointer SP**.

Le **Stack Pointer SP** évolue très fréquemment, suivant le **sommet de pile** selon ce qui y est posé ou retiré.

Le **Base Pointer BP** évolue à chaque appel ou retour de fonction, son rôle étant d'indiquer la zone de données de la **fonction courante**.

Mnémotechnie : SP = Stack Pointer ≈ Sommet de Pile

BP = Base Pointer ≈ Bas de Pile

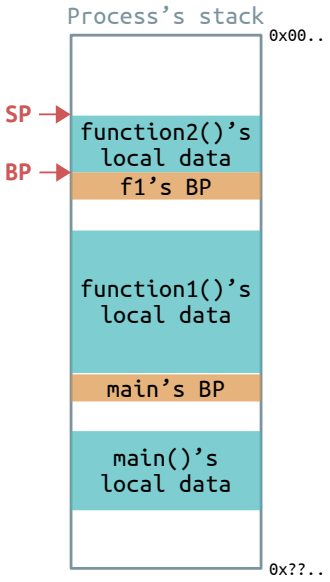


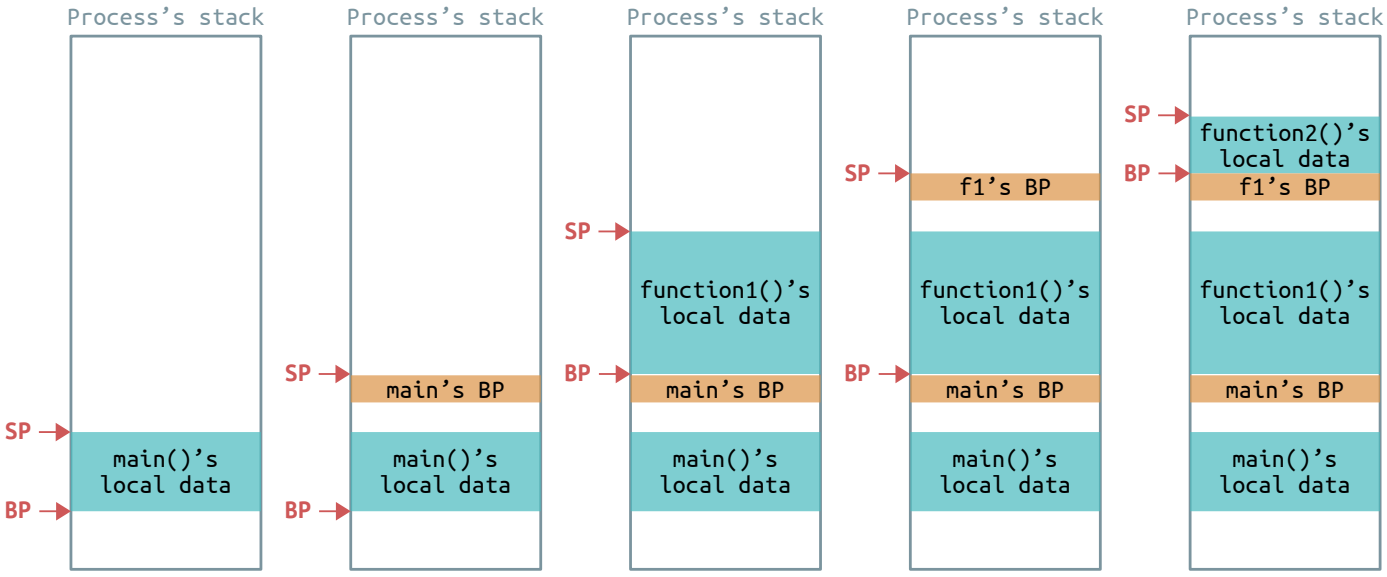
La valeur du **Stack Pointer SP** étant toujours associé au sommet de pile (indépendamment des fonctions), il n'est pas nécessaire de sauvegarder sa valeur.

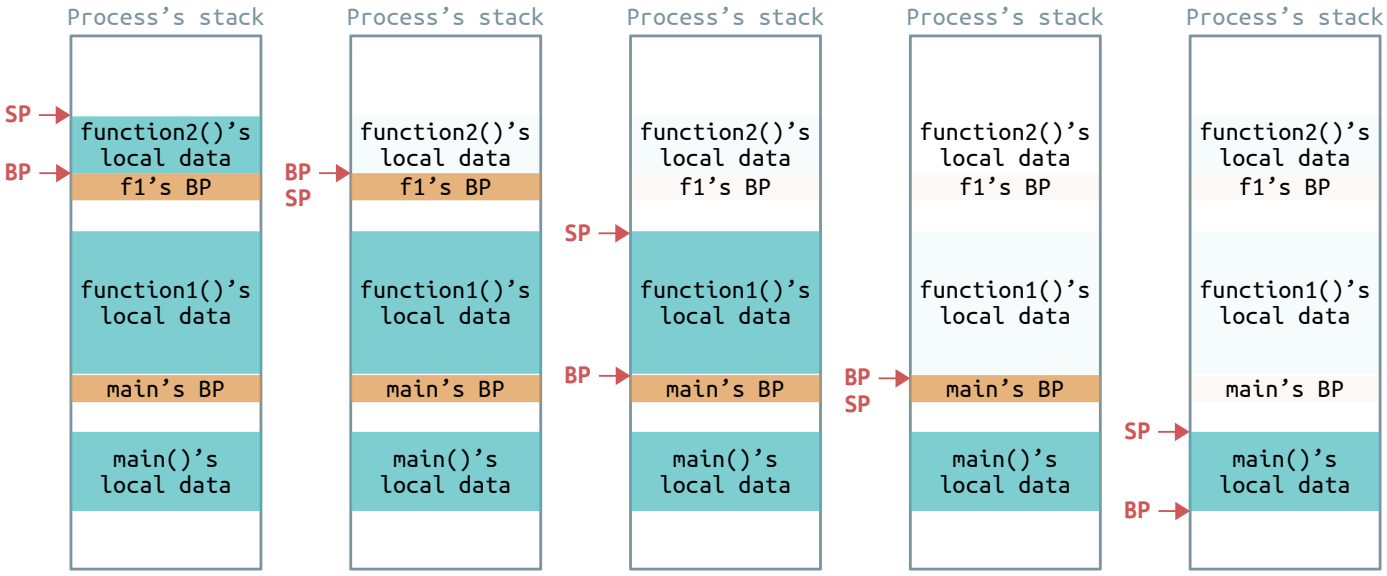
Le **Base Pointer BP** en revanche est associé à la fonction en cours d'exécution. Quand une nouvelle fonction est appelée, sa valeur sera amenée à changer.

Sa valeur sera donc sauvegardée avant d'être modifiée pour correspondre à la nouvelle fonction.

Ainsi, il sera possible de retrouver la valeur initiale de BP lorsque la fonction appelée effectuera son `return`.







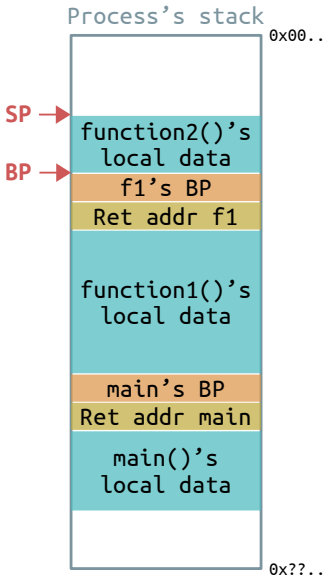
Le **Instruction Pointer IP*** contient l'adresse de la prochaine instruction à exécuter.

* IP est parfois appelé Program Counter PC, comme chez le PIC18 par exemple.

Dans le cas d'un code purement linéaire, il s'incrémente automatiquement pour pointer vers l'instruction suivante.

Lors d'un appel de fonction, sa valeur est écrasée. Il faut alors trouver un moyen de sauvegarder sa valeur : en la posant sur la pile.

```
0000000000001129 <main>:
    1129:    pushq  %rbp
    112a:    movq   %rsp, %rbp
    112d:    callq  1139 <function_1>
    1132:    movl   $0x0, %eax
    1137:    popq   %rbp
    1138:    retq

0000000000001139 <function_1>:
    1139:    pushq  %rbp
    113a:    movq   %rsp,  %rbp
    113d:    subq   $0x10, %rsp
    1141:    movl   $0x3,  %edx
    1146:    movl   $0x2,  %esi
    114b:    movl   $0x1,  %edi
    1150:    callq  115b <function_2>
    1155:    movl   %eax,  -0x4(%rbp)
    1158:    nop
    1159:    leaveq
    115a:    retq

000000000000115b <function_2>:
    115b:    pushq  %rbp
    115c:    movq   %rsp,  %rbp
    115f:    movl   %edi,  -0x4(%rbp)
    1162:    movl   %esi,  -0x8(%rbp)
    1165:    movl   %edx,  -0xc(%rbp)
    ...
    1175:    popq   %rbp
    1176:    retq
```



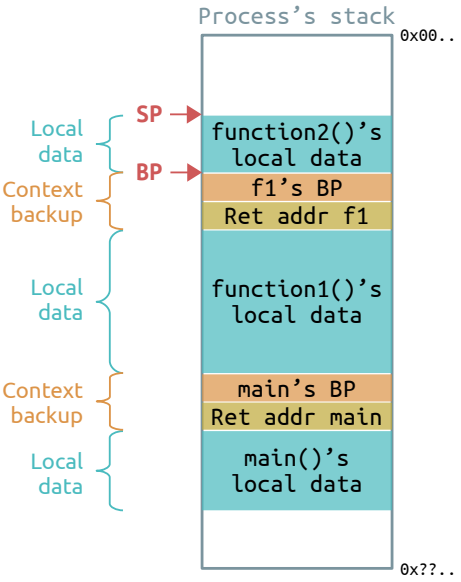
Dans la pile du process, on retrouve donc une alternance de zones contenant les données locales des fonctions et de zones de sauvegarde de contexte de ces fonctions.

Un **contexte d'exécution** est la valeur des registres du CPU à un instant *t*. Ces valeurs ne sont valides que dans le contexte (= cadre) de la fonction en cours.

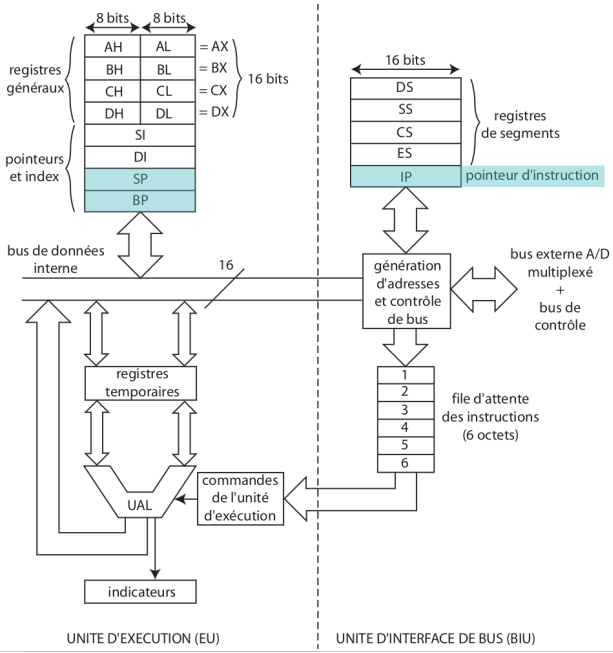
Exemple : les registres IP et BP, mais ça peut aussi être le cas pour les registres de travail.

Si la fonction courante appelle une autre fonction, alors les registres IP et BP qu'elle utilise seront écrasés par la fonction appelée.

On sauvegarde donc la valeur de ces registres afin de restituer le contexte de la fonction appelante au retour de la fonction appelée.



Exemple du Intel 8086



Dr. J.Y. Haggège, 2003
Microprocesseur, Support de cours
https://www.exoco-lmd.com/systemes-a-microprocesseurs/cours-microprocesseur/

```c
int main(void)
{
    function_1();
    return 0;
}

void function_1 (void)
{
    int ret_1;
    ret_1 = function_2 (1, 2, 3);
}

int function_2 (int a_2, int b_2, int c_2)
{
    return a_2 + b_2 + c_2;
}
```

① ③ ⑤ Sauvegarde du BP de la fonction appelante

② ④ ⑥ Restauration du BP de la fonction appelante

⑦ ⑨ Sauvegarde du IP de la fonction appelante

⑧ ⑩ Restauration du IP de la fonction appelante

Passage d'arguments par les registres CPU

```
0000000000001129 <main>:
①  1129:  pushq  %rbp
   112a:  movq   %rsp, %rbp
⑦  112d:  callq  1139 <function_1>
   1132:  movl   $0x0, %eax
②  1137:  popq   %rbp
   1138:  retq

0000000000001139 <function_1>:
③  1139:  pushq  %rbp
   113a:  movq   %rsp,  %rbp
   113d:  subq   $0x10, %rsp
   1141:  movl   $0x3,  %edx
   1146:  movl   $0x2,  %esi
   114b:  movl   $0x1,  %edi
⑨  1150:  callq  115b <function_2>
   1155:  movl   %eax,  -0x4(%rbp)
   1158:  nop
④  1159:  leaveq
⑧  115a:  retq

000000000000115b <function_2>:
⑤  115b:  pushq  %rbp
   115c:  movq   %rsp,  %rbp
   115f:  movl   %edi,  -0x4(%rbp)
   1162:  movl   %esi,  -0x8(%rbp)
   1165:  movl   %edx,  -0xc(%rbp)
   ...
⑥  1175:  popq   %rbp
⑩  1176:  retq
```

Allocation pour variable locale

ret_1 : adresse relative à BP

Variables locales : adr. relatives à BP

La taille de la pile est gérée par l'OS. Par défaut elle est de **8 MB** sous Linux, mais il est possible de la modifier (minimum de 128 kB) : soit depuis le programme en utilisant les fonctions `getrlimit()` et `setrlimit()` ; soit depuis l'OS avec la commande `ulimit`.

```
dboudier:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 62349
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 62349
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Cette taille relativement petite rend l'usage de la stack inintéressant pour les grandes quantités de données.

Dans ce cas, on lui préférera le tas/heap.

```
dboudier:~$ ulimit -s
8192
dboudier:~$ ulimit -s 16384
dboudier:~$ ulimit -s
16384
```

La stack/pile en bref :

- Allocation dynamique (faite à l'exécution ou run-time)
- Allocation automatique (instructions ajoutées par la *toolchain*, non par le dév.)

- Pile LIFO (Last In, First Out)
- Pour les données locales aux fonctions
- Pour les sauvegardes de contexte (BP, IP)
- Passage d'arguments de fonction par les registres CPU

- De taille modeste (par défaut 8 Mo)

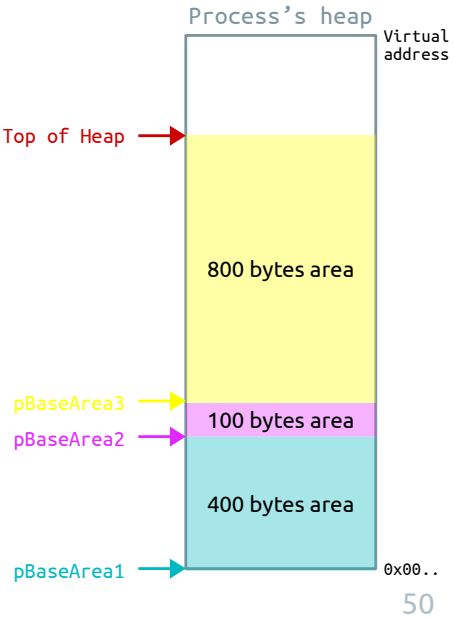
Passons maintenant à l'étude du segment de **tas ou heap**. Chaque processus possède son propre tas (partagé entre les threads d'un même processus).

Les ressources y sont allouées dynamiquement, par demande explicite du développeur avec les fonctions dédiées du langage :

- `calloc()`, `malloc()`, `free()` pour le langage C
- `new`, `delete` pour le langage C++
- ...

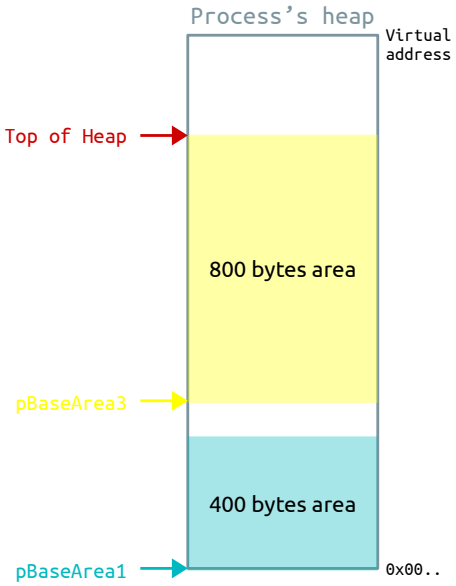
Chaque demande d'allocation renvoie un pointeur vers la base de la zone effectivement allouée.

```c
int main (void) {
    float*  pBaseArea1;
    char*   pBaseArea2;
    double* pBaseArea3;

    pBaseArea1 = (float*)  malloc( 100 * sizeof(float)  );
    pBaseArea2 = (char*)   malloc( 100 * sizeof(char)   );
    pBaseArea3 = (double*) malloc( 100 * sizeof(double) );

    // user application …

    free( pBaseArea2 );
    pBaseArea2 = (char*) malloc( 200 * sizeof(char) );

    // user application …

    free( pBaseArea1 );
    free( pBaseArea2 );
    free( pBaseArea3 );

    return 0;
}
```



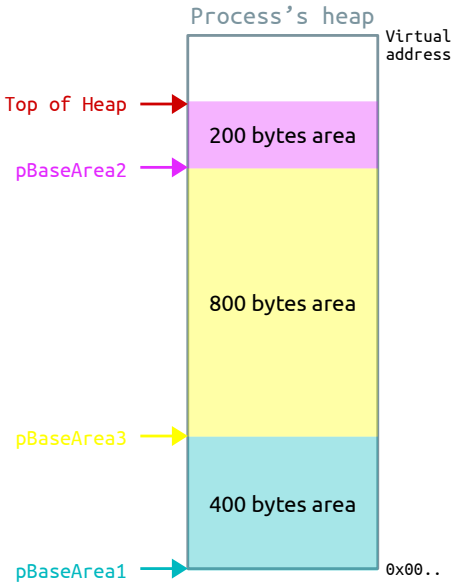
Une demande de désallocation déréférence le pointeur.

```c
int main (void) {
    float*  pBaseArea1;
    char*   pBaseArea2;
    double* pBaseArea3;

    pBaseArea1 = (float*)  malloc( 100 * sizeof(float)  );
    pBaseArea2 = (char*)   malloc( 100 * sizeof(char)   );
    pBaseArea3 = (double*) malloc( 100 * sizeof(double) );

    // user application …

    free( pBaseArea2 );
    pBaseArea2 = (char*) malloc( 200 * sizeof(char) );

    // user application …

    free( pBaseArea1 );
    free( pBaseArea2 );
    free( pBaseArea3 );

    return 0;
}
```



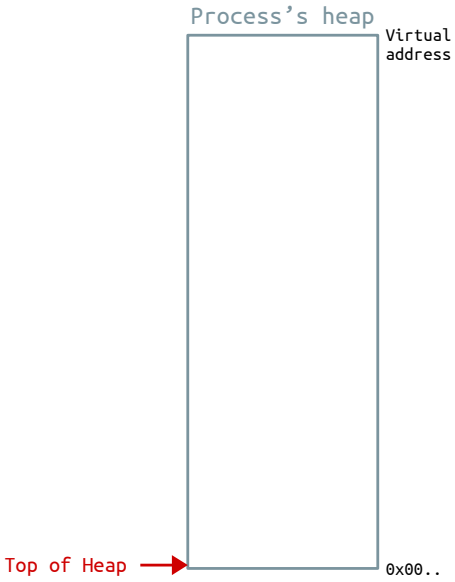
L'espace de mémoire virtuelle sera toujours contigu, même si l'unité de pagination de la MMU gère en réalité la fragmentation de la mémoire physique.

```c
int main (void) {
    float*  pBaseArea1;
    char*   pBaseArea2;
    double* pBaseArea3;

    pBaseArea1 = (float*)  malloc( 100 * sizeof(float)  );
    pBaseArea2 = (char*)   malloc( 100 * sizeof(char)   );
    pBaseArea3 = (double*) malloc( 100 * sizeof(double) );

    // user application …

    free( pBaseArea2 );
    pBaseArea2 = (char*) malloc( 200 * sizeof(char) );

    // user application …

    free( pBaseArea1 );
    free( pBaseArea2 );
    free( pBaseArea3 );

    return 0;
}
```



Il faut toujours libérer les ressources mémoires après utilisation, sinon les données encombrent la mémoire.

```c
int main (void) {
    float*  pBaseArea1;
    char*   pBaseArea2;
    double* pBaseArea3;

    pBaseArea1 = (float*)  malloc( 100 * sizeof(float)  );
    pBaseArea2 = (char*)   malloc( 100 * sizeof(char)   );
    pBaseArea3 = (double*) malloc( 100 * sizeof(double) );

    // user application …

    free( pBaseArea2 );
    pBaseArea2 = (char*) malloc( 200 * sizeof(char) );

    // user application …

    free( pBaseArea1 );
    free( pBaseArea2 );
    free( pBaseArea3 );

    return 0;
}
```



Comme pour la pile, des débordements de tas (*heap overflow*) sont possibles.

Mais contrairement à la pile, **le tas dispose de quasiment tout l'espace de la mémoire principale**, ce qui permet d'y allouer de graaaandes quantités de données.

```
dboudier:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 62349
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 62349
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Mémoire virtuelle : illimitée (en théorie).

En pratique, limitée par la taille réelle de la RAM (moins l'espace occupé par l'OS, ~ 1 Go).

Vu par le CPU, l'espace des données allouées sur le tas est toujours contigu.

Ceci est dû au travail de la **MMU (*Memory Management Unit*)** qui gère la fragmentation de la mémoire physique d'un côté (avec son unité de pagination) et la présente majestueusement au CPU.

Les processeurs sans MMU (typiquement, les micro-contrôleurs) et leurs *toolchains* associées gèrent difficilement les mécanismes de fragmentation de leur mémoire physique.

Dans ces cas, l'utilisation des fonctions `malloc()` et `free()` doit être prohibé, ou a minima surveillé avec de grandes précautions.

Pas de MMU → pas d'allocation dynamique

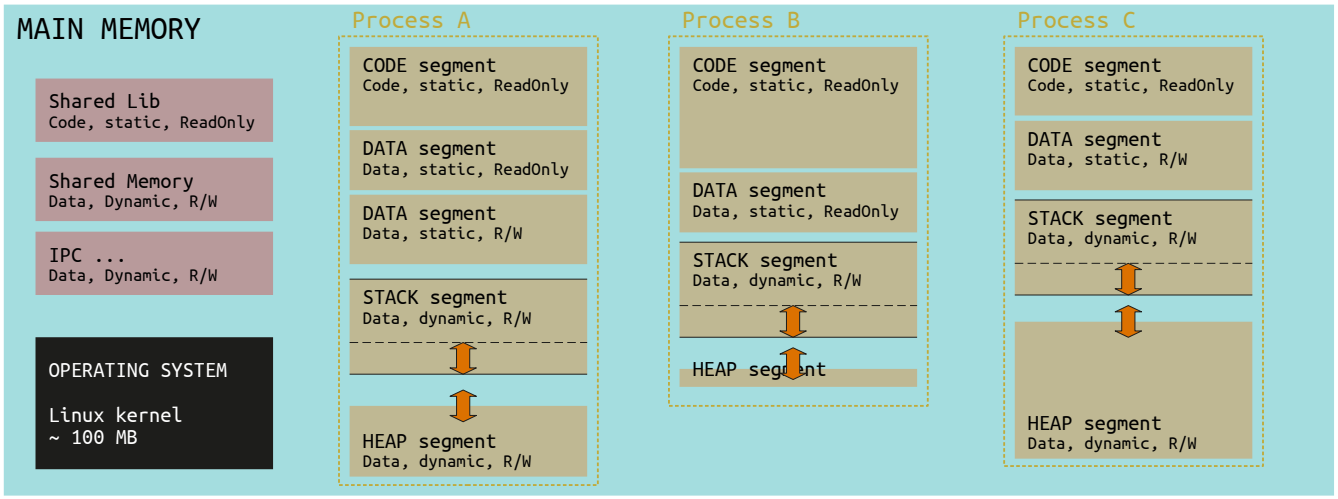
Le heap/tas en bref :

- Allocation dynamique (faite à l'exécution ou run-time)
- Allocation explicite (fonctions appelées par le développeur)
  - `malloc(), free()`

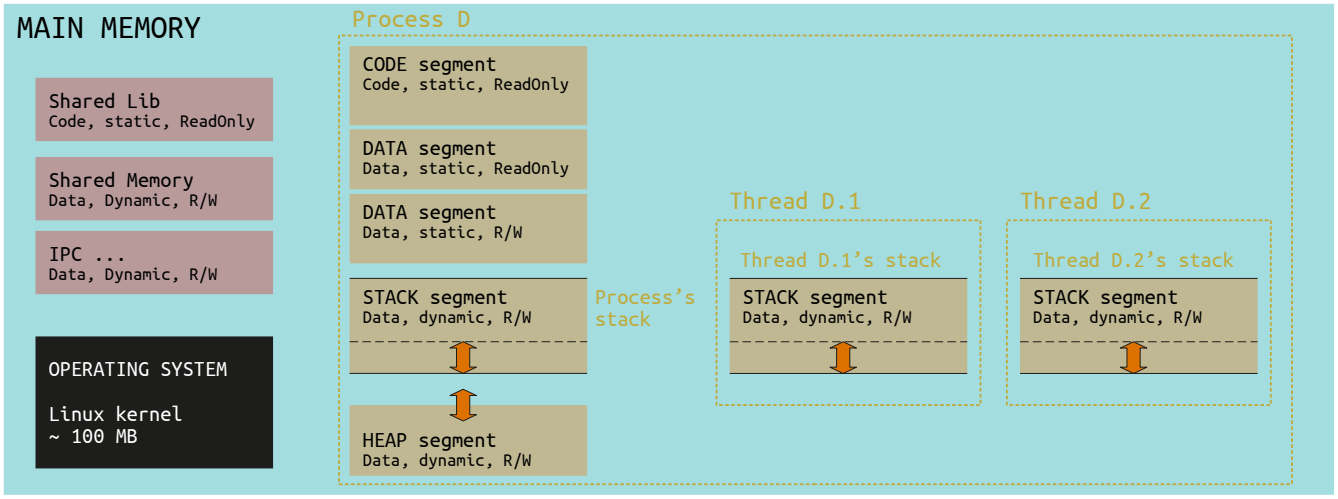
- Pour les grandes quantités de données (tableaux dynamiques, ...)
- Toujours libérer les ressources allouées après utilisation
- Nécessité d'avoir une MMU (Memory Management Unit)

- De taille infinie en théorie, limitée à la taille de la RAM en pratique

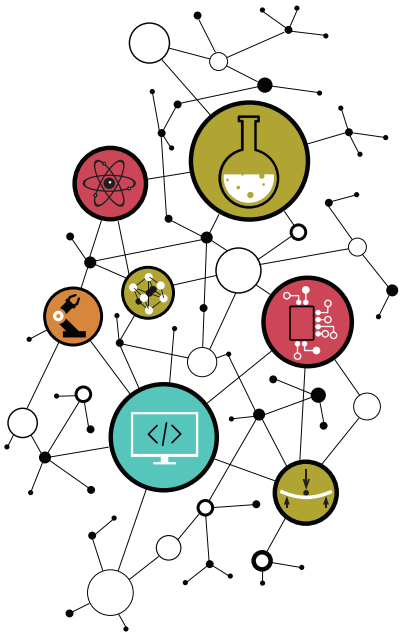
Sur un ordinateur équipé d'un système d'exploitation, plusieurs processus sont actifs en même temps. Ils possèdent chacun leur espace mémoire dédié, auquel s'ajoutent les zones partagées (librairies dynamiques, IPC, ...)



Un processus typique possède plusieurs **threads** (processus légers). Chaque thread dispose de sa propre *stack*, mais tous les autres segments du processus sont partagés.



CONTACT



Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES