

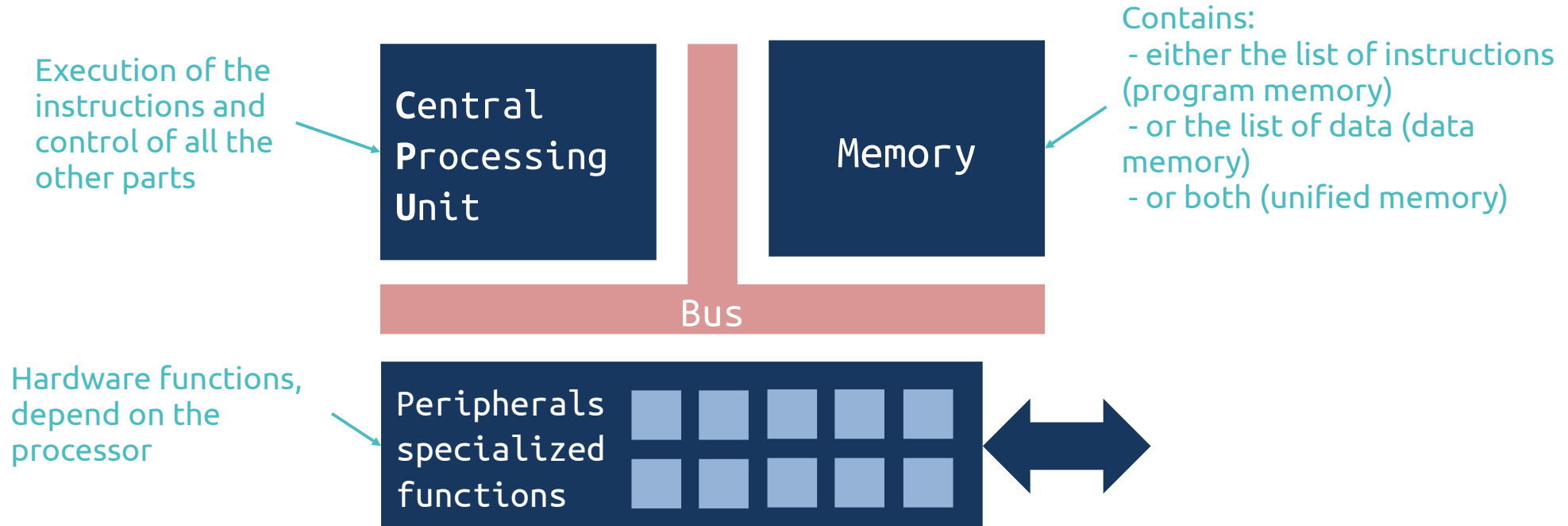
# Chapter 1

# Inside Processors



2021-2022

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.

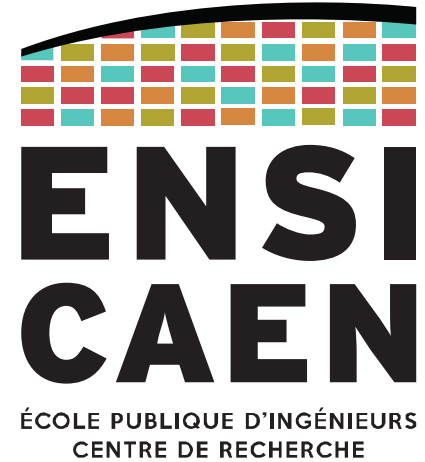
Different CPU/memory/peripherals configurations lead to different architectures. The most common architectures will be described in the “**Processor Architectures**” chapter.

# 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:

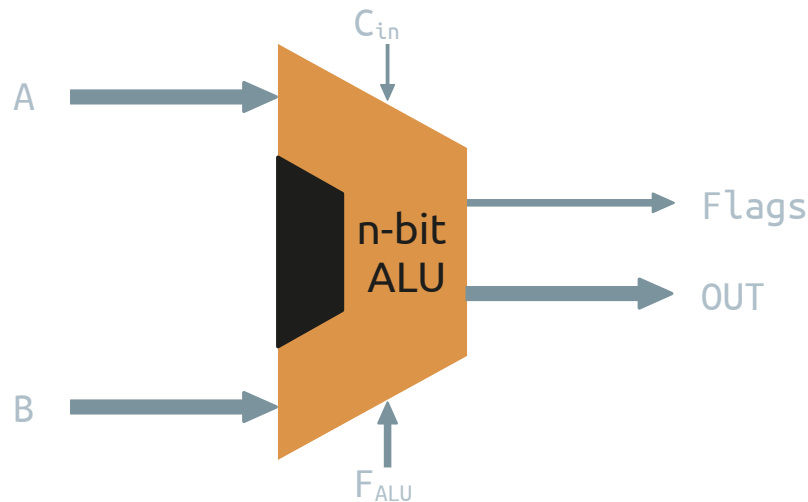
- An Arithmetic and Logic Unit (ALU)
  - Logic operations and simple maths
- A Floating-Point Unit (FPU)
  - For advanced processors
- A multiplier
- A shift register
- ...

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  
...



What is a register ?

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

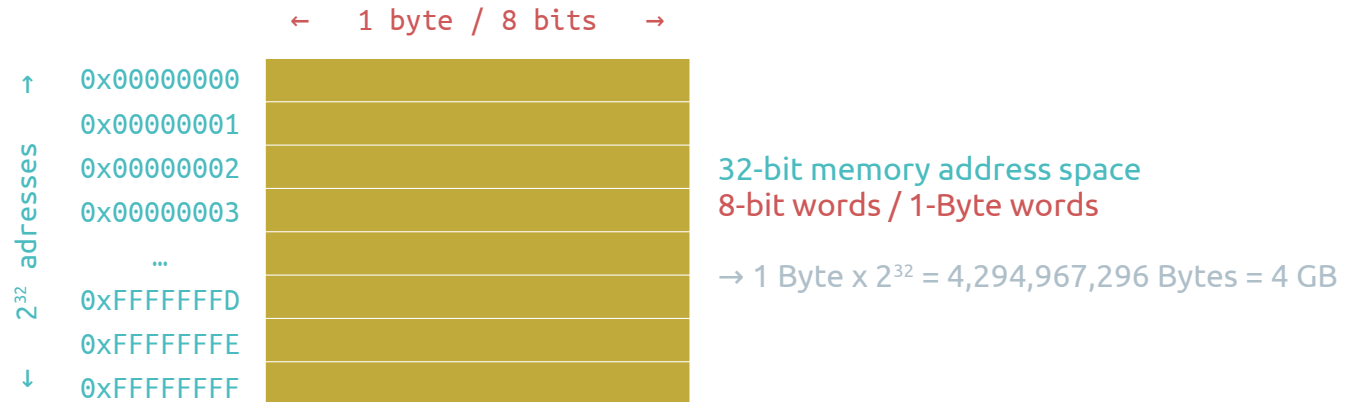
Remanent mass storage



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 UVROM (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.

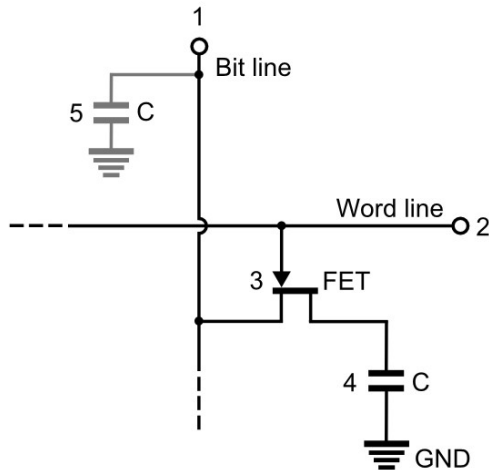
The **main memory** is a volatile but very fast memory. The processor uses it to store data that is actively used.

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

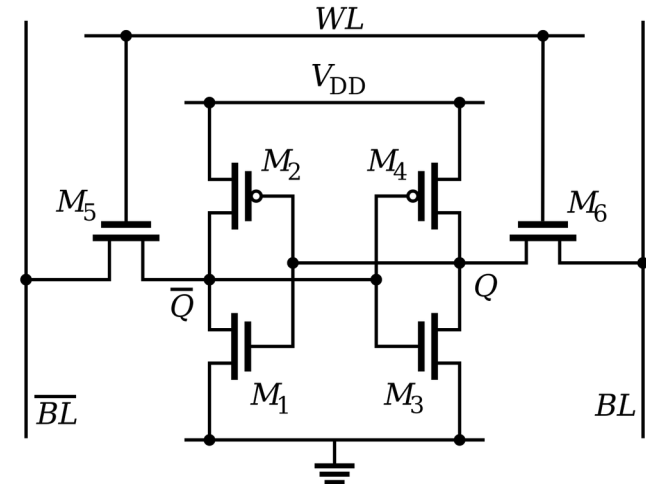
Volatile memory comes in two types: **DRAM (Dynamic RAM)** and **SRAM (Static RAM)**.

**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 (4<sup>th</sup> 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

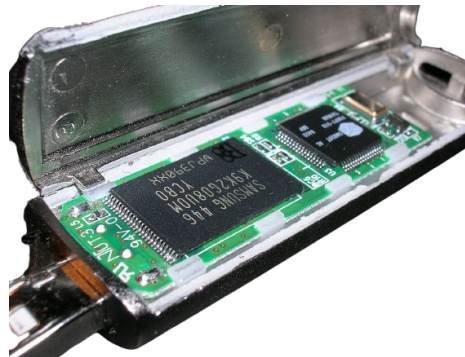
Remanent mass storage comes different technologies:

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

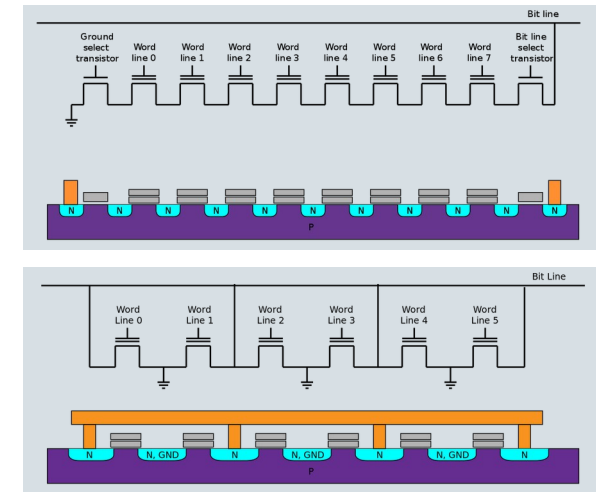
Electrical charge storage with logic circuitry is used by EEPROMs (Electrically Erasable Programmable ROMs). The most common EEPROM technology is Flash memory (NAND and NOR), which has a constant access time to the information. SSDs (Solid-State Drives) also use Flash technology.



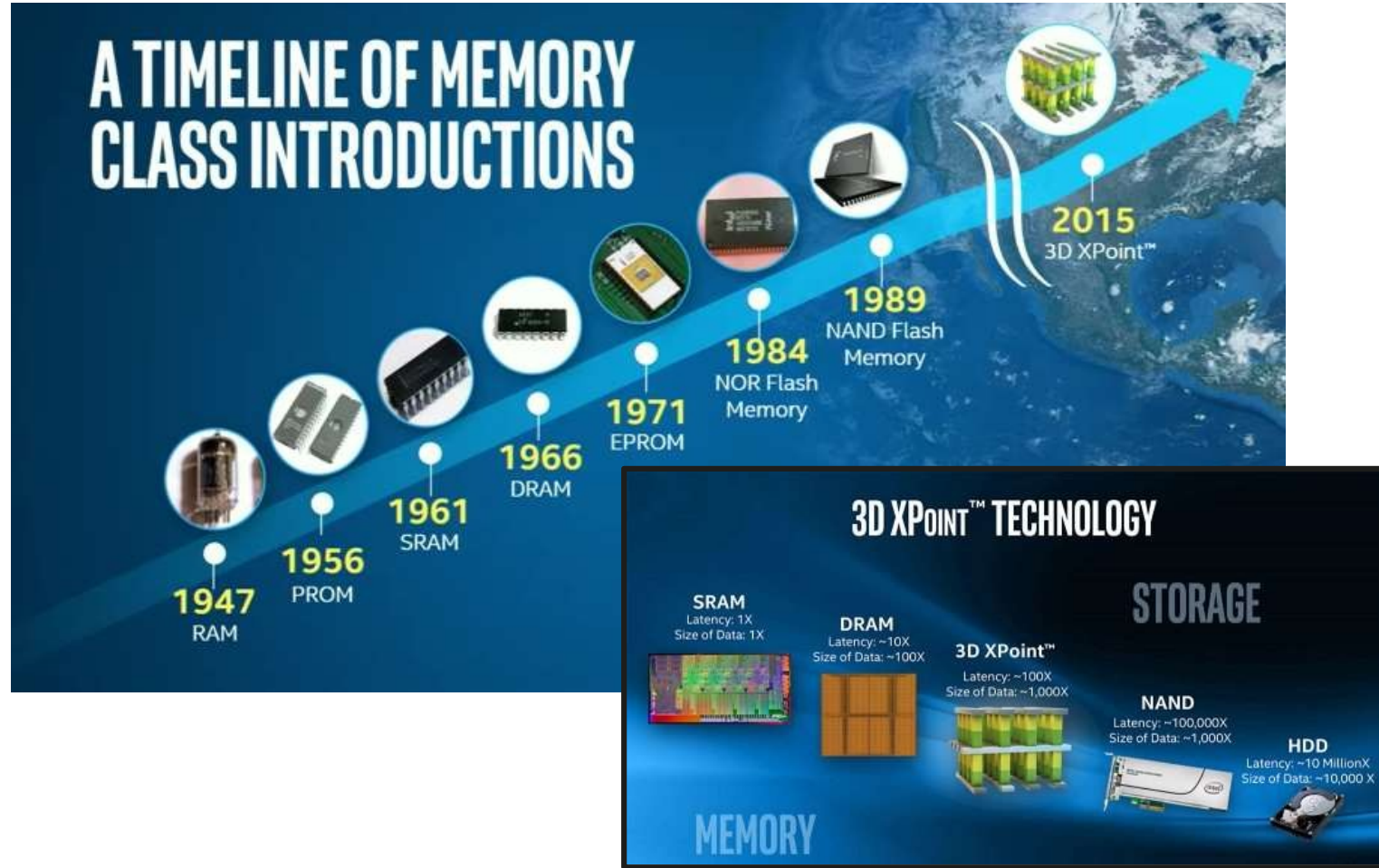
Hard Drive Disk (HDD)



Flash drive  
(Flash memory on the left)



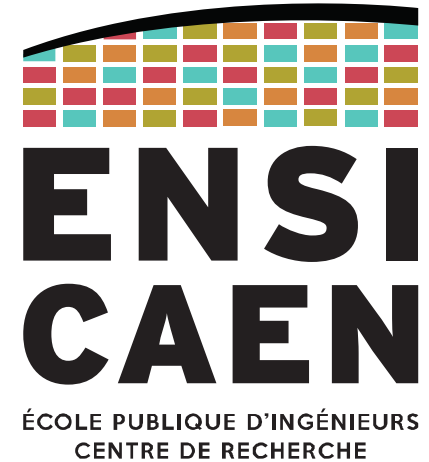
NAND (top) and NOR (bottom)  
Flash memory structures





# EXECUTING A PROGRAM

From the C file to an executable binary program  
Execution on a home-made processor

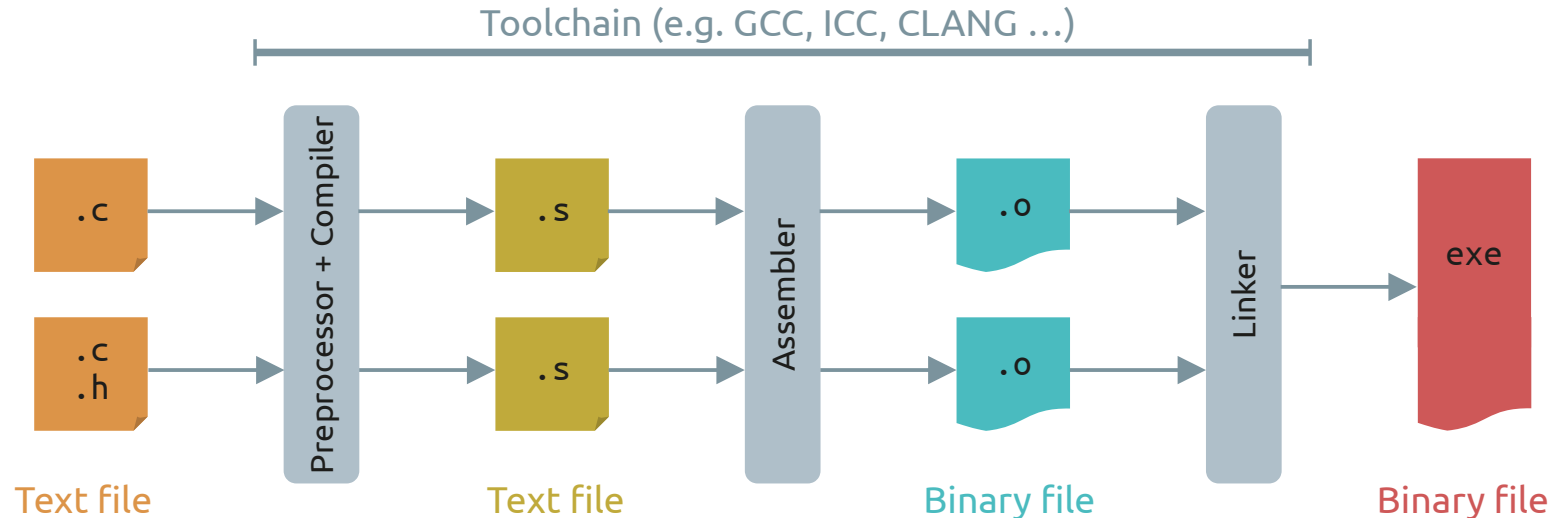


## EXECUTING A PROGRAM

From the C file to an executable program

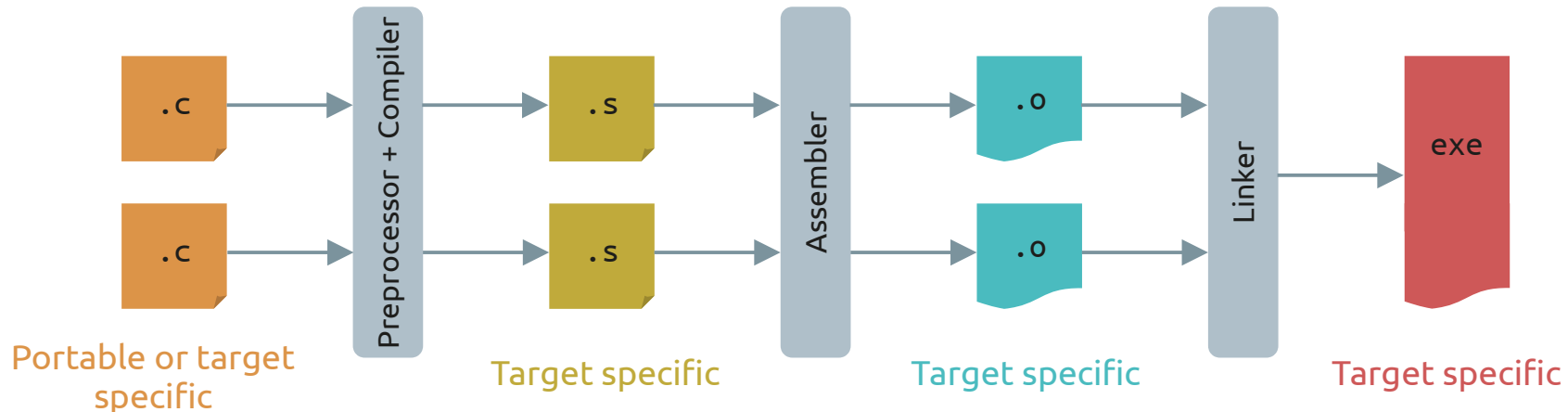
We'll keep it simple here, as you will see this in details next year.

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.

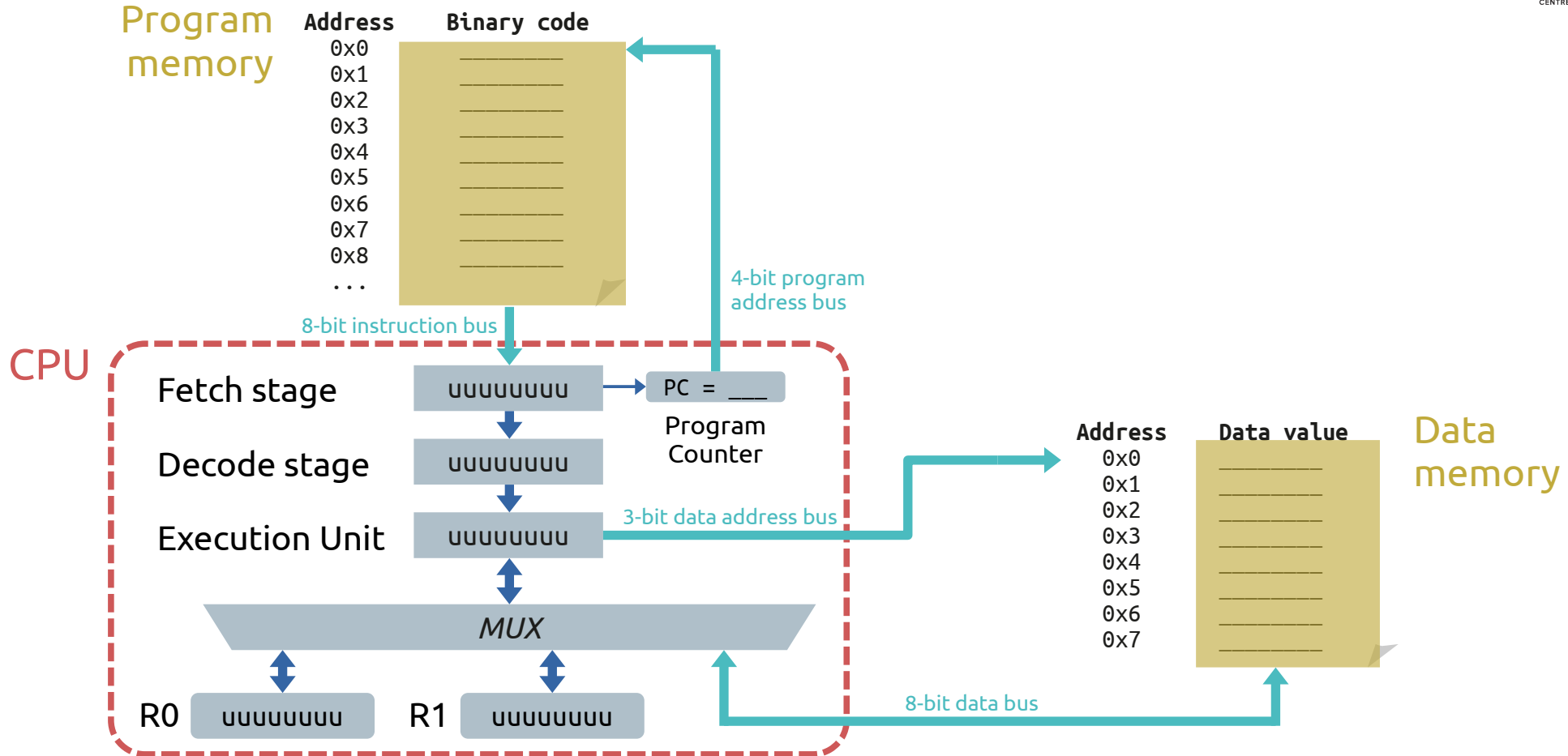
Operation	Syntax	Description	Example	Binary Opcode
ADD	ADD srcReg, srcReg, dstReg	Add two register values	ADD R0, R1, R0	000 r r r uu
JMP	JMP label	Program memory jump	JMP main	001 aaaa u
LOAD	LOAD address, dstReg	Load data memory value to register	LOAD var1, R1	010 aaa r u
MOV	MOV srcReg, dstReg	Copy 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

# EXECUTING A PROGRAM

## Home-made processor



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

```
char value = 3; // Stored at 0x0
char saveValue; // Stored at 0x1 } Data memory map

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

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

# EXECUTING A PROGRAM

## Home-made processor





## Solution

## C language program

```

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

# EXECUTING A PROGRAM

## Home-made processor

Follow the CPU work step by step

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	uuuuuuuu
0x8	uuuuuuuu
...	

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

### CPU

Fetch stage

Decode stage

Execution Unit

uuuuuuuu

PC = 0x0

Program Counter

uuuuuuuu

uuuuuuuu

MUX

R0

uuuuuuuu

R1

uuuuuuuu

4-bit program address bus

8-bit instruction bus

3-bit data address bus

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000011  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	uuuuuuuu
0x8	uuuuuuuu
...	

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
```

CPU

Fetch stage

01000010

PC = 0x1

Decode stage

uuuuuuuu

Program Counter

Execution Unit

uuuuuuuu

3-bit data address bus

MUX

R0

uuuuuuuu

R1

uuuuuuuu

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000011  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	uuuuuuuu
0x8	uuuuuuuu
...	

### 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
```

CPU

Fetch stage

10001000

PC = 0x2

Decode stage

01000010

Program Counter

Execution Unit

uuuuuuuu

3-bit data address bus

MUX

R0

uuuuuuuu

R1

uuuuuuuu

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000011  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	UUUUUUUU
0x8	UUUUUUUU
...	

### 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
```

CPU

Fetch stage

00001000

PC = 0x3

Program Counter

Decode stage

10001000

Execution Unit

01000010

3-bit data address bus

MUX

R0

UUUUUUUU

R1

00000011

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000011  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	UUUUUUUU
0x8	UUUUUUUU
...	

### 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
```

CPU

Fetch stage

10100000

PC = 0x4

Decode stage

00001000

Program Counter

Execution Unit

10001000

3-bit data address bus

MUX

R0

00000010

R1

00000011

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000011  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

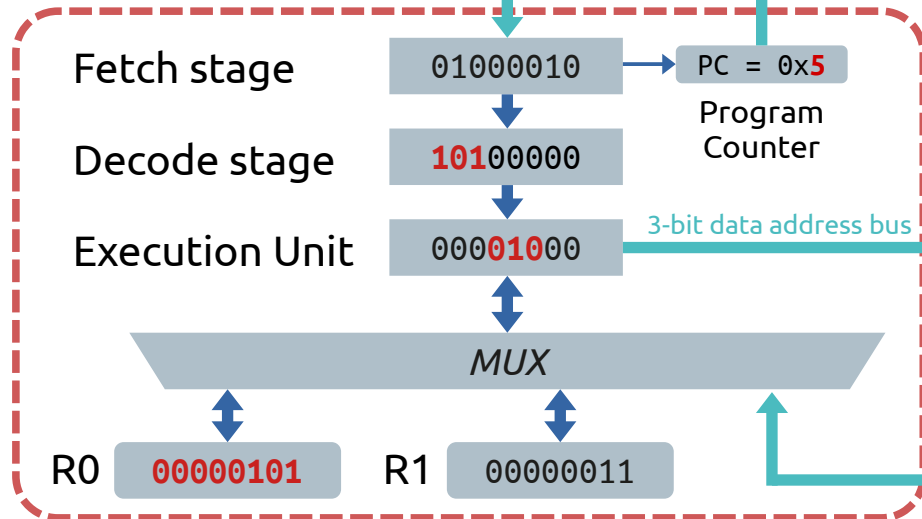
Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	UUUUUUUU
0x8	UUUUUUUU
...	

### 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
```

CPU



Address	Data value
0x0	00000011
0x1	UUUUUUUU
0x2	UUUUUUUU
0x3	UUUUUUUU
0x4	UUUUUUUU
0x5	UUUUUUUU
0x6	UUUUUUUU
0x7	UUUUUUUU

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

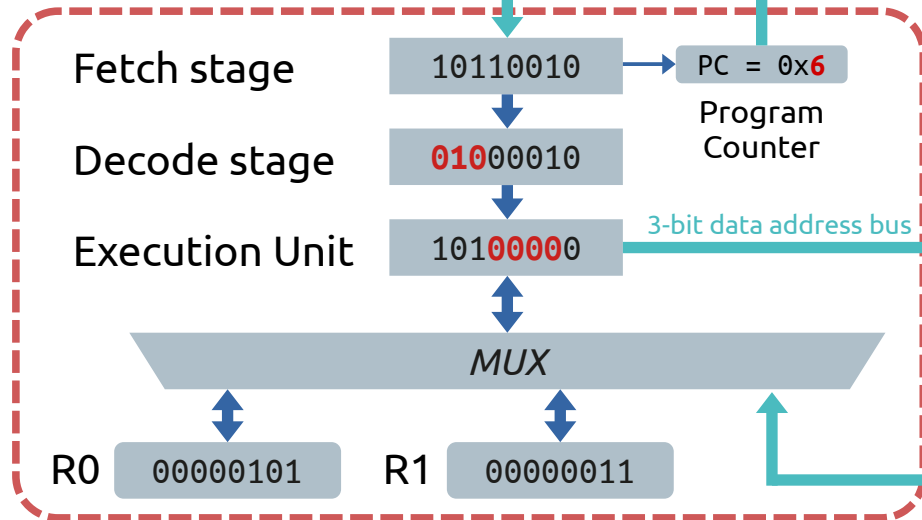
Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	UUUUUUUU
0x8	UUUUUUUU
...	

### 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
```

CPU



Address	Data value
0x0	00000101
0x1	UUUUUUUU
0x2	UUUUUUUU
0x3	UUUUUUUU
0x4	UUUUUUUU
0x5	UUUUUUUU
0x6	UUUUUUUU
0x7	UUUUUUUU

Data memory



# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	UUUUUUUU
0x8	UUUUUUUU
...	

### 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
```

CPU

Fetch stage

00100000

PC = 0x7

Decode stage

10110010

Program Counter

Execution Unit

01000010

3-bit data address bus

MUX

R0

00000101

R1

00000101

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000101  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	UUUUUUUU
0x8	UUUUUUUU
...	

### 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
```

CPU

Fetch stage

UUUUUUUU

PC = 0x8

Decode stage

00100000

Program Counter

Execution Unit

10110010

3-bit data address bus

MUX

R0

00000101

R1

00000101

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000101  
00000101  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	UUUUUUUU
0x8	UUUUUUUU
...	

### 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
```

### CPU

#### Fetch stage

UUUUUUUU

PC = 0x0

#### Decode stage

UUUUUUUU

Program  
Counter

#### Execution Unit

00100000

3-bit data address bus

MUX

R0

00000101

R1

00000101

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000101  
00000101  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU

### Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	uuuuuuuu
0x8	uuuuuuuu
...	

### 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
```

### CPU

Fetch stage

01000010

PC = 0x1

Decode stage

uuuuuuuu

Program  
Counter

Execution Unit

uuuuuuuu

3-bit data address bus

MUX

R0

00000101

R1

00000101

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000101  
00000101  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu

Data  
memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	uuuuuuuu
0x8	uuuuuuuu
...	

### 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
```

CPU

Fetch stage

10001000

PC = 0x2

Decode stage

01000010

Program Counter

Execution Unit

uuuuuuuu

3-bit data address bus

MUX

R0

00000101

R1

00000101

8-bit data bus

Address

Data value

0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000101  
00000101  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu  
uuuuuuuu

Data memory

# EXECUTING A PROGRAM

## Home-made processor

### Program memory

Address	Binary code
0x0	01000010
0x1	10001000
0x2	00001000
0x3	10100000
0x4	01000010
0x5	10110010
0x6	00100000
0x7	UUUUUUUU
0x8	UUUUUUUU
...	

### 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

Fetch stage

00001000

PC = 0x3

Decode stage

10001000

Program Counter

Execution Unit

01000010

3-bit data address bus

MUX

R0

00000101

R1

00000101

8-bit data bus

Address

Data value

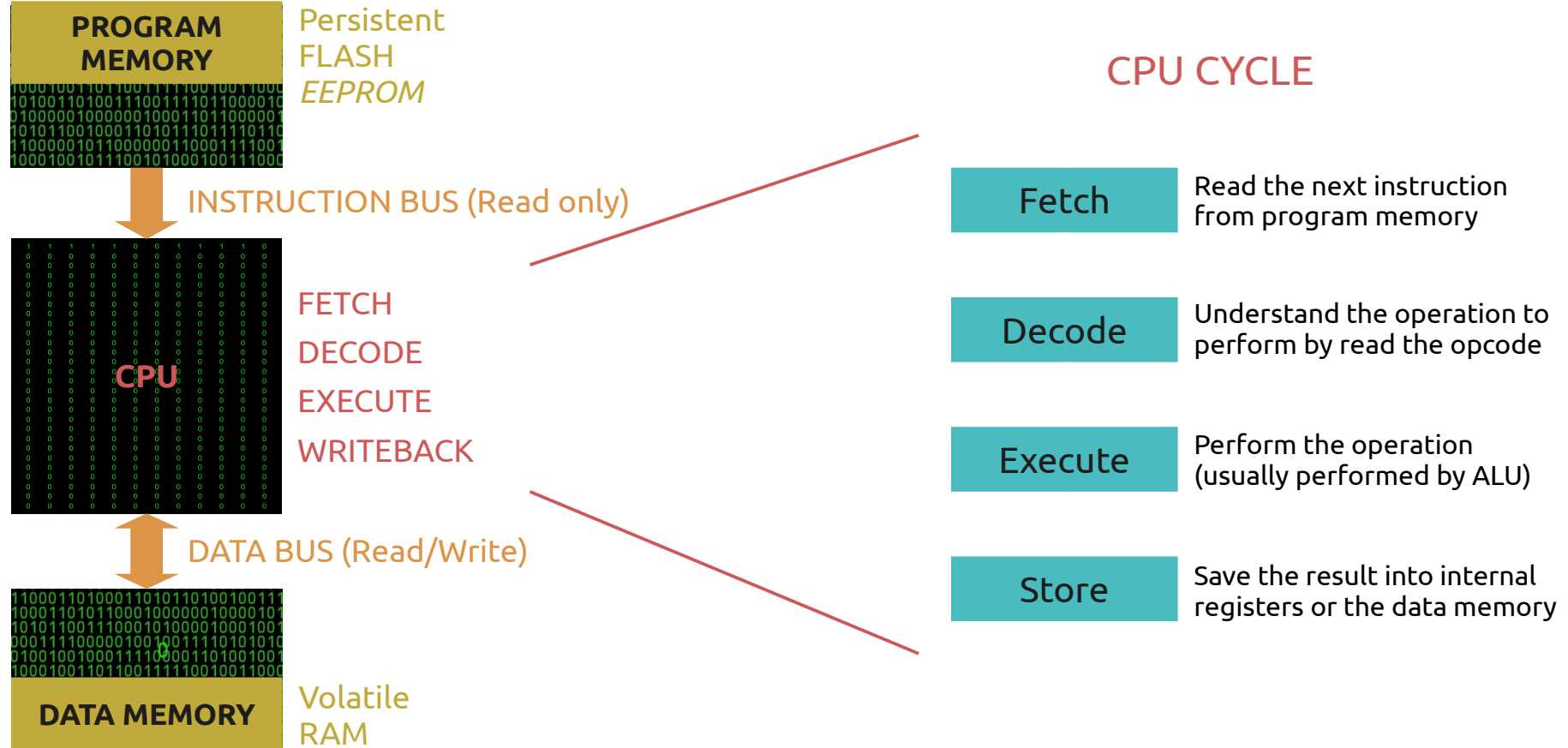
0x0  
0x1  
0x2  
0x3  
0x4  
0x5  
0x6  
0x7

00000101  
00000101  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU  
UUUUUUUU

Data memory

# EXECUTING A PROGRAM

## Program execution

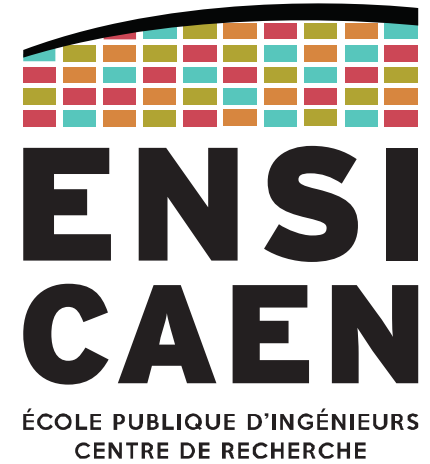






# PERIPHERALS

Examples

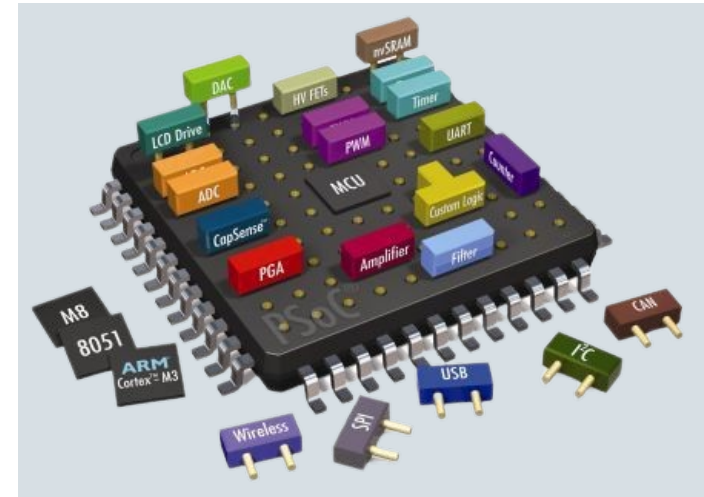


Peripherals are **hardware functions** built for specific processing.

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/I<sup>2</sup>C/UART/USB/Eth, ...) that differ from a processor to another.

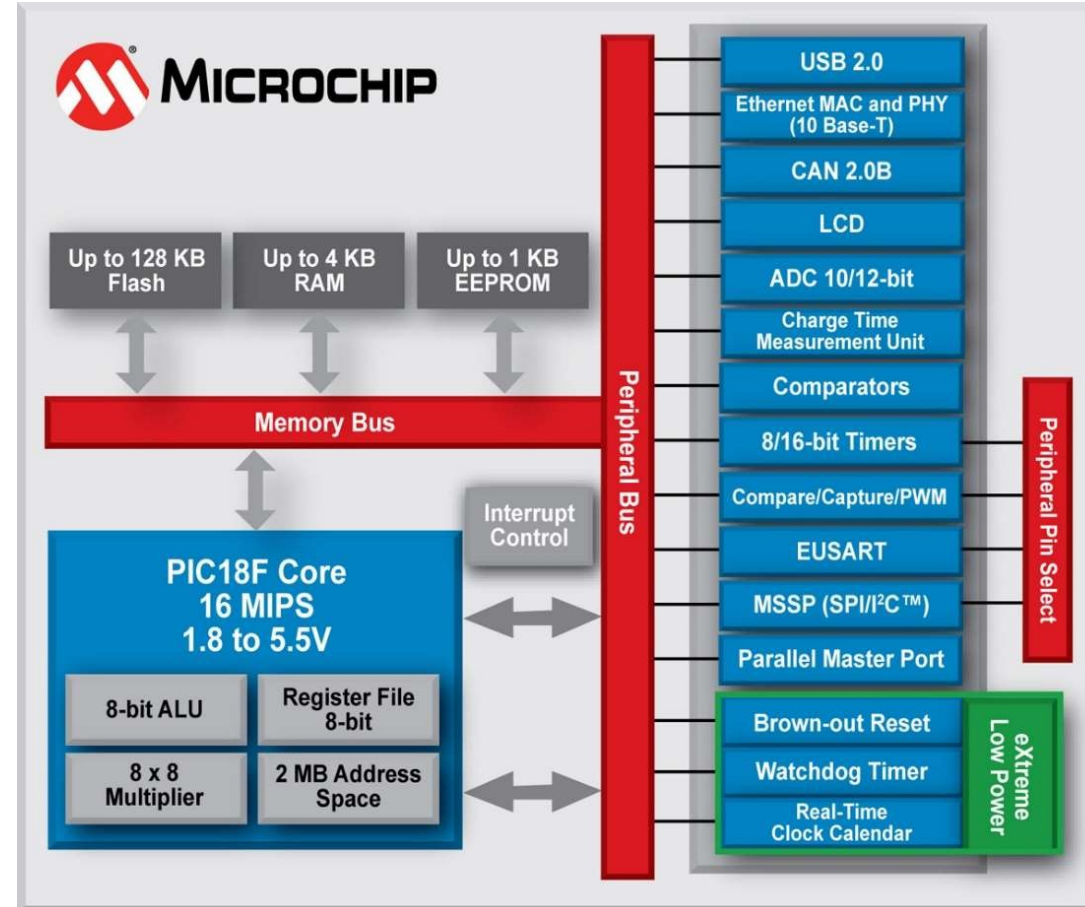


### Example

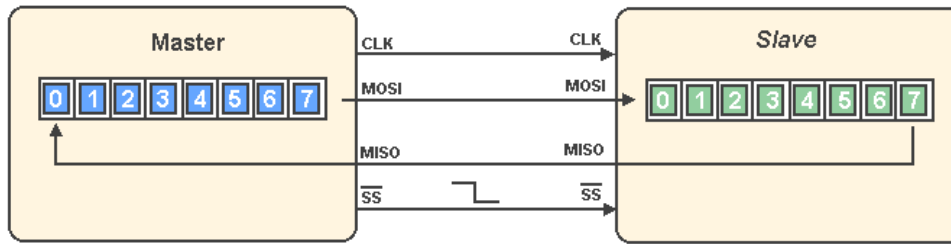
#### Microchip's PIC18 (8-bit MCU)

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.

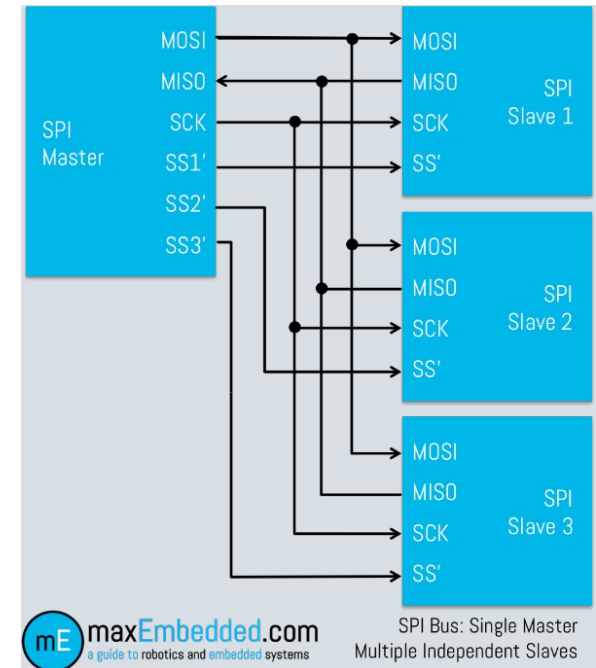


MOSI: Master Output, Slave Input

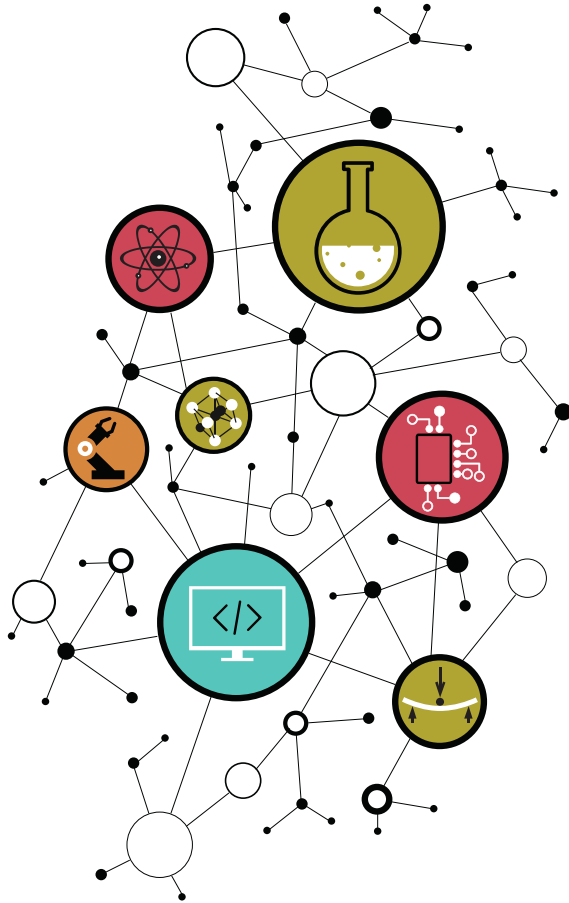
MISO: Master Input, Slave Output

SCK: Serial Clock

SSx: Slave Select (for slave #x)



## CONTACT



Dimitri Boudier – PRAG ENSICAEN  
[dimitri.boudier@ensicaen.fr](mailto:dimitri.boudier@ensicaen.fr)

With the precious help of:

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