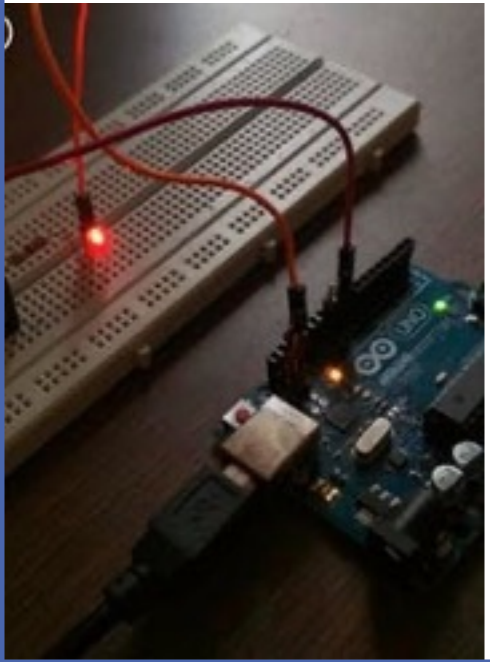
Paul Pinault

Blog/contact : www.disk91.com
Twitter : @disk_91
YouTube: https://www.youtube.com/c/PaulPinault

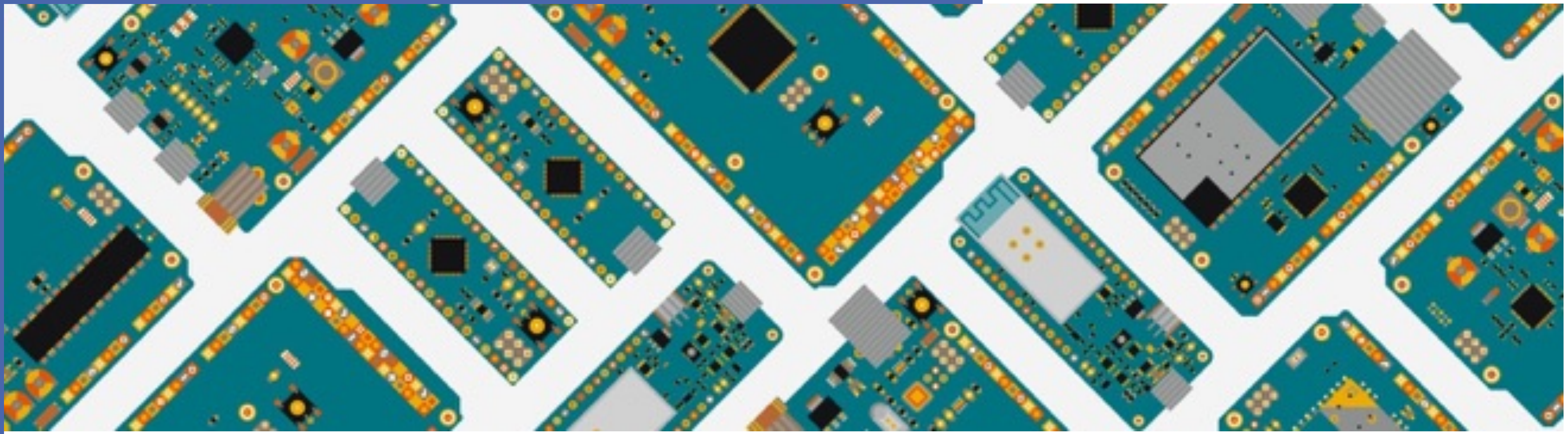
Low Level Architecture & Arduino

Introduction to processor and micro-controller architecture and Arduino environment.
Training for 1^{st} IS students.



Computer science history and processor architecture

```
This example code is in the public domain.

http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin LED_BUILTIN as an output
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED
  delay(1000);                       // wait for a s
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED
  delay(1000);                       // wait for a s
}
```



Author – Paul Pinault / Disk91.com

From mechanical approach to fully integrated, high density circuits, 400 years of engineering.

Evolution of the computing technics

1642 – 1890 – 1941 – 1955 – 1959 – 202X

Mechanical

Pascaline

Early ages of computing, developed by Blaise Pascal, capable of addition and subtraction.

Electro-Mechanical

Herman Hollerith Machine

Punch card-based systems able to manage multiple accumulators.

Lamps

ENIAC

Programmable system able to execute about 5.000 instructions per seconds for 160KWh with 17K tubes

Transistor

TRADIC

Programmable system running 1M instruction / s for 100Wh with 10K transistors

Integrated Circuit

IBM 360

Programmable system running 1M instruction / s

Up to 2T inscription / s in 2020

Author – Paul Pinault / Disk91.com

LAMP technology

Is acting as power switch controlled by an electrical signal.

- 2 wires are heating the lamp
- 1 wire control the on/off switch
- 2 wires are passing the currant when on

LAMP NEEDS TO BE WARM TO WORK

Bootup time is quite long due to warming up time.

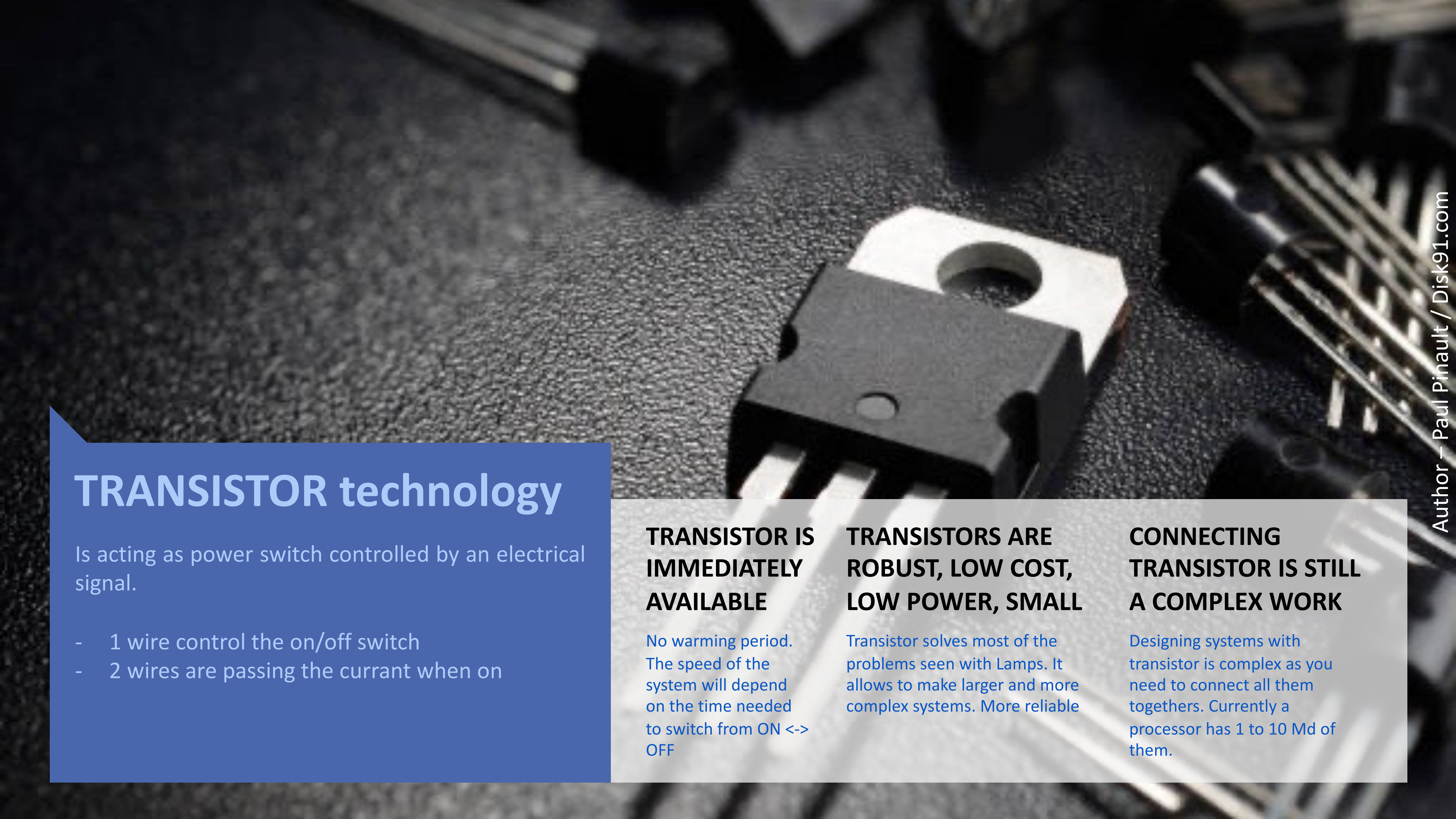
LAMPS ARE INSECURED / FRAGILE

As a classical lamp bulb, they are fragile. Lights is attracting BUGS and bugs crash the lamps ... And the associated programs.

LAMPS ARE BIG, EXPENSIVE, ENERGY CONSUMMING

Due to size, integration complexity and technology, the LAMP systems are not scalable, expensive and complex to maintain.

Author – Paul Pinault / Disk91.com



TRANSISTOR technology

Is acting as power switch controlled by an electrical signal.

- 1 wire control the on/off switch
- 2 wires are passing the currant when on

TRANSISTOR IS IMMEDIATELY AVAILABLE

No warming period. The speed of the system will depend on the time needed to switch from ON <-> OFF

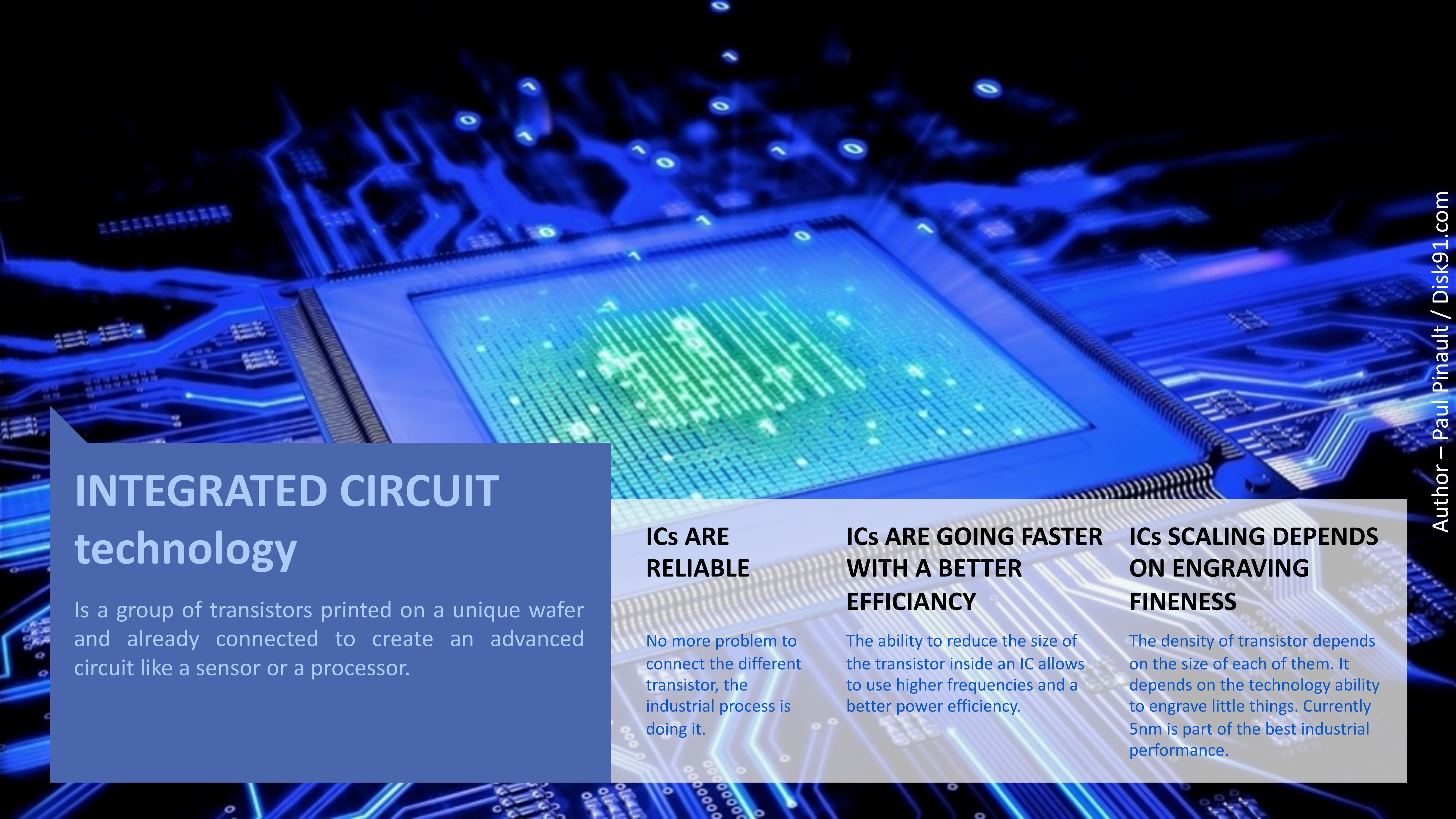
TRANSISTORS ARE ROBUST, LOW COST, LOW POWER, SMALL

Transistor solves most of the problems seen with Lamps. It allows to make larger and more complex systems. More reliable

CONNECTING TRANSISTOR IS STILL A COMPLEX WORK

Designing systems with transistor is complex as you need to connect all them togethers. Currently a processor has 1 to 10 Md of them.

Author – Paul Pinault / Disk91.com



INTEGRATED CIRCUIT technology

Is a group of transistors printed on a unique wafer and already connected to create an advanced circuit like a sensor or a processor.

ICs ARE RELIABLE

No more problem to connect the different transistor, the industrial process is doing it.

ICs ARE GOING FASTER WITH A BETTER EFFICIANCY

The ability to reduce the size of the transistor inside an IC allows to use higher frequencies and a better power efficiency.

ICs SCALING DEPENDS ON ENGRAVING FINENESS

The density of transistor depends on the size of each of them. It depends on the technology ability to engrave little things. Currently 5nm is part of the best industrial performance.

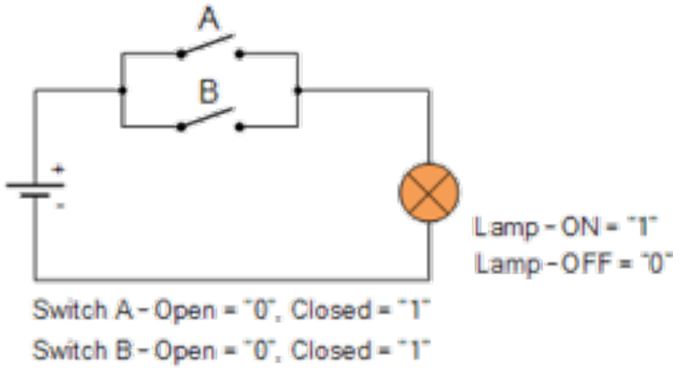
Author – Paul Pinault / Disk91.com

BOOLEAN LOGIC IS BASE OF COMPUTING

OR gate can be designed with mechanical switches or transistors in parallel.

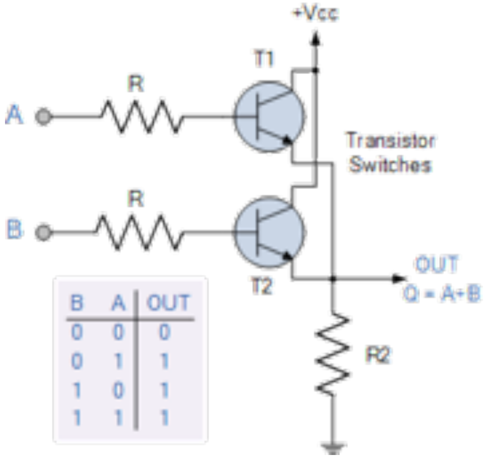
AND gate can be designed with mechanical switches or transistors in series.

Any of the logical gates can be made with transistors.

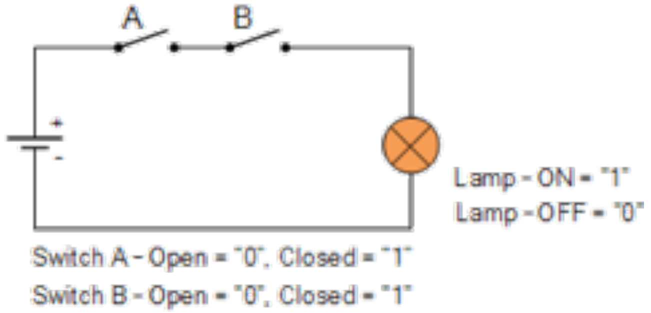


Lamp - ON = "1"
Lamp - OFF = "0"

Switch A - Open = "0", Closed = "1"
Switch B - Open = "0", Closed = "1"

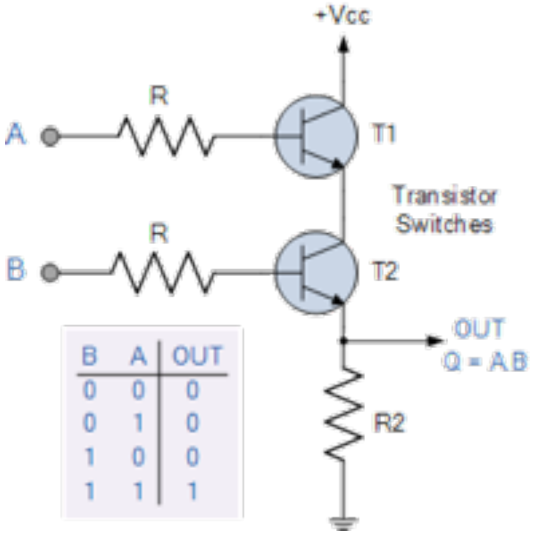


B	A	OUT
0	0	0
0	1	1
1	0	1
1	1	1



Lamp - ON = "1"
Lamp - OFF = "0"

Switch A - Open = "0", Closed = "1"
Switch B - Open = "0", Closed = "1"



B	A	OUT
0	0	0
0	1	0
1	0	0
1	1	1

Author – Paul Pinault / Disk91.com

STEPS AFTER STEPS, PROGRAMMER HAVE ADDED LAYERS TO SIMPLIFY PROGRAMMING

SOFTWARE

INTERPRETED LANGUAGES

Python, Bash, Basic, PHP, JS … These languages are executed by a program interpreting the program lines after lines calling some HIGH–LEVEL LANGUAGES functions.

HIGH LEVEL LANGUAGES

C, C++, GO, FORTAN … These languages simplify the programmer work by allowing complex operation in a single line. Compilation transforms them into assembly / machine language for being executed.

ASSEMBLY LANGUAGE

Instructions are the same as for the machine language but each of the instruction is TEXT encoded so it can be manipulated by humans.

HARDWARE

MACHINE LANGUAGE

The instruction set the micro-processor can execute. An instruction a based on micro-instruction or electrical circuits. An instruction is an OP code ; is a binary value.

MICRO INSTRUCTION

Hardware encoded advanced instruction based on the execution of multiple basic instructions.

ELECTRICAL CIRCUIT

Transistor circuits assembled to create basic instructions like mathematical operation, memory transfer…

Author – Paul Pinault / Disk91.com

OPERATING SYSTEM CREATES A HARDWARE ABSTRACTION FOR SOFTWARE

SOFTWARE

INTERPRETED LANGUAGES

Python, Bash, Basic, PHP, JS … These languages are executed by a program interpreting the program lines after lines calling some HIGH-LEVEL LANGUAGES functions.

HIGH LEVEL LANGUAGES

C, C++, GO, FORTAN … These languages simplify the programmer work by allowing complex operation in a single line. Compilation transforms them into assembly / machine language for being executed.

ASSEMBLY LANGUAGE

Instructions are the same as for the machine language but each of the instruction is TEXT encoded so it can be manipulated by humans.

OPERATING SYSTEM

HARDWARE

MACHINE LANGUAGE

The instruction set the micro-processor can execute. An instruction a based on micro-instruction or electrical circuits. An instruction is an OP code ; is a binary value.

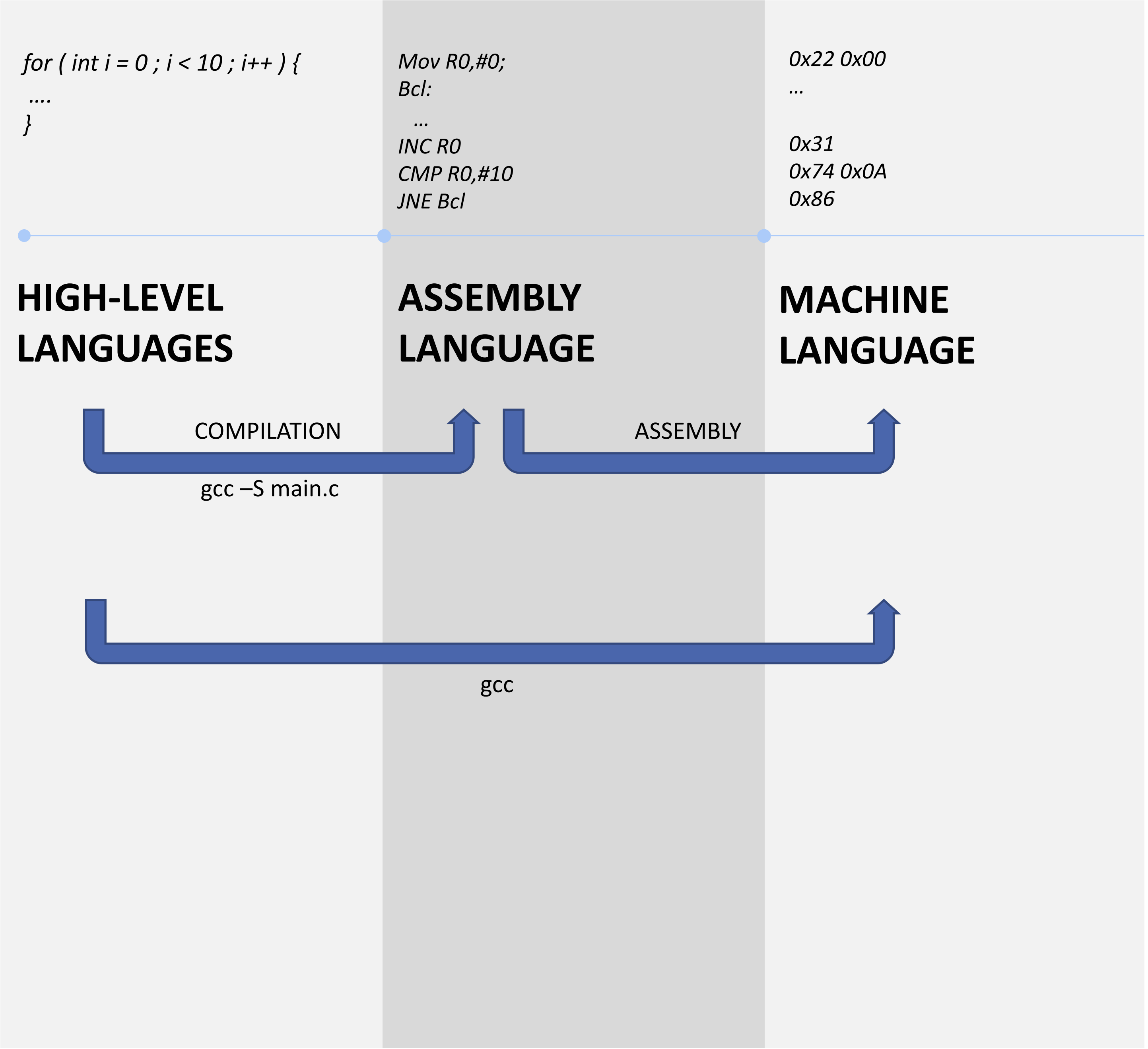
MICRO INSTRUCTION

Hardware encoded advanced instruction based on the execution of multiple basic instructions.

ELECTRICAL CIRCUIT

Transistor circuits assembled to create basic instructions like mathematical operation, memory transfer…

Compiler chain transforms a source code into an executable binary



```
for ( int i = 0 ; i < 10 ; i++ ) {
 ....
}
```

```
Mov R0,#0;
Bcl:
   ...
INC R0
CMP R0,#10
JNE Bcl
```

```
0x22 0x00
...

0x31
0x74 0x0A
0x86
```

HIGH-LEVEL LANGUAGES

ASSEMBLY LANGUAGE

MACHINE LANGUAGE

COMPILATION

gcc –S main.c

ASSEMBLY

gcc

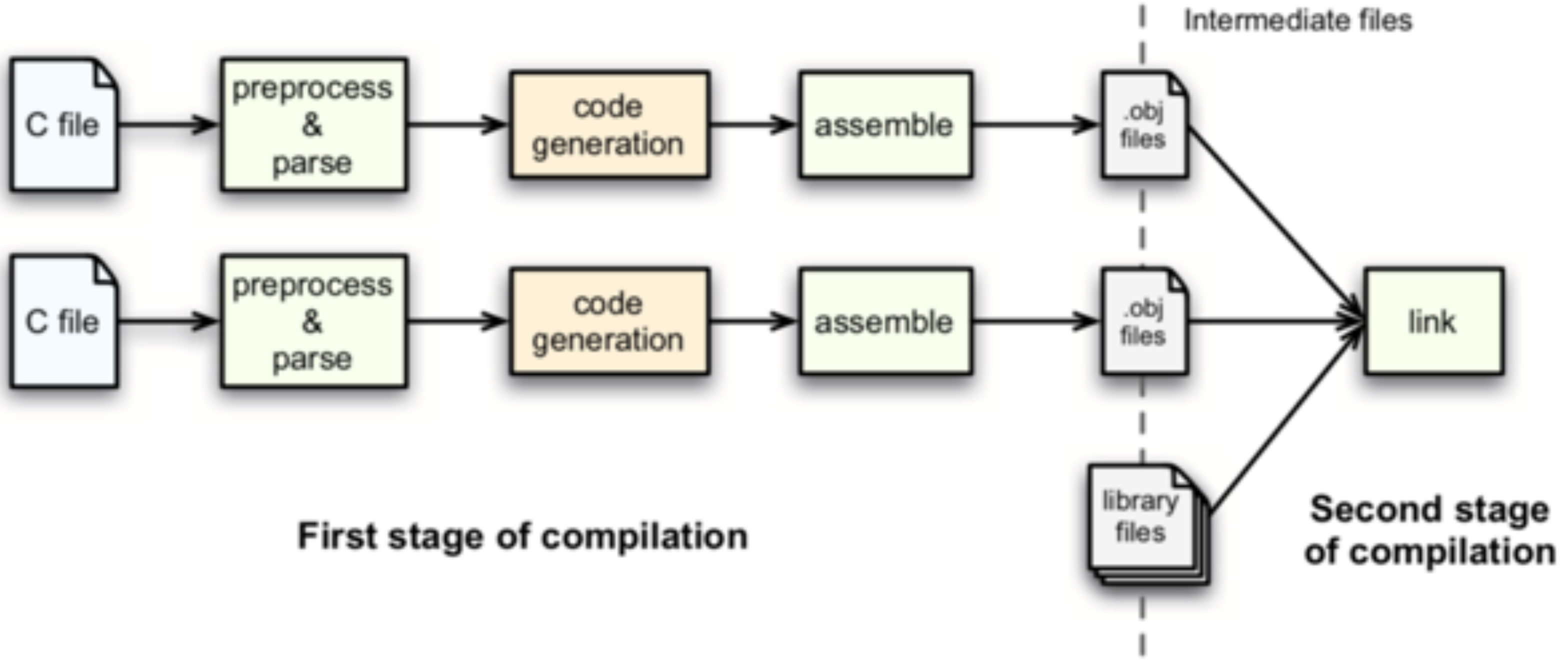
Compiler chain transforms a source code into an executable binary

When multiple files need to be assembled after the compilation phase, there is a link edition phase

1 **Preprocess & Parse**
Replace #define, verify syntax … to generate a final and valid high level source code

2 **Code generation and assembly generation**
Transform High-Level source code into language machine. At this step external references are not known.

3 **Linking**
Makes link between external references to create a single code. Place objects to the right location in memory



Author – Paul Pinault / Disk91.com

COMPUTER MAIN COMPONENTS

1 MEMORY

A memory is like an array in C, for each of the addresses you can read or write a value.

There are different types of memories in a computer:

- Persistent memory – stores data even when there is no power supply
- Volatile memory - fast memory only working when power is supplied.
- Cache memory – faster memory used to replicate parts of volatile memory to improve performance
- Registers – fastest memory blocks used to execute the computations.

Modern CPU process 3-6G instructions per seconds per core

PERSISTANT MEMORY

Unit is GB/TB
(in 2020)

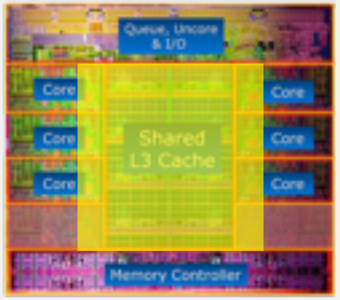
VOLATILE MEMORY

Dynamic RAM
(need to be refreshed)

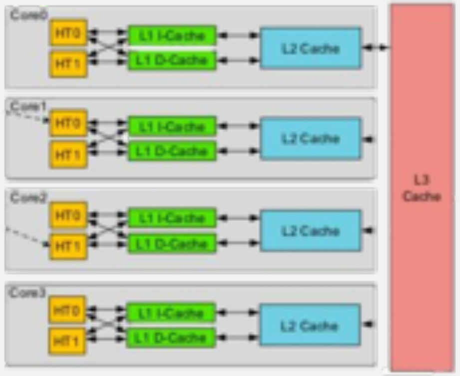
Unit is GB
(in 2020)

CACHE MEMORY

Static RAM
(no need to be refreshed)



L3 – 60 GB/s
L2 – 80 GB/s
L1 – 210 GB/s



Unit is KB / MB
(in 2020)

get more on https://www.forrestthewoods.com/blog/memory-bandwidth-napkin-math/

COMPUTER MAIN COMPONENTS

2 REGISTERS

Registers are specific, small block of memory inside the CPU core.

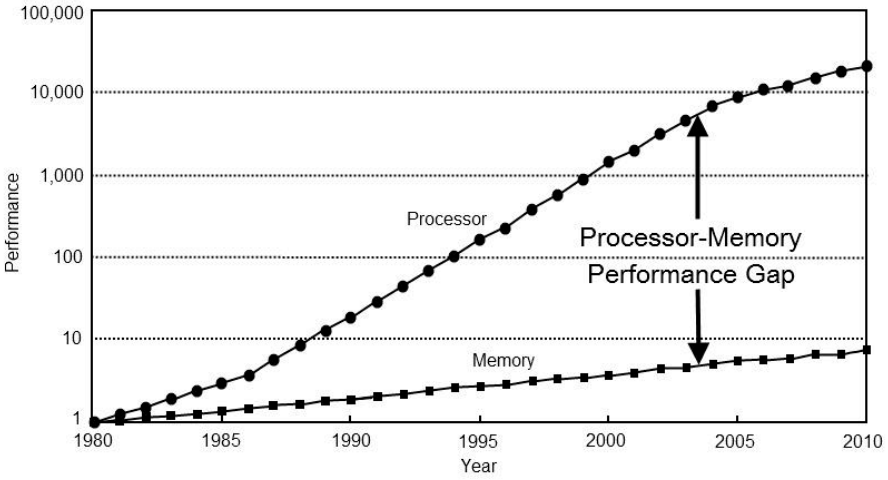
This is about **1KB** of memory per core.

This memory zone works at CPU full speed

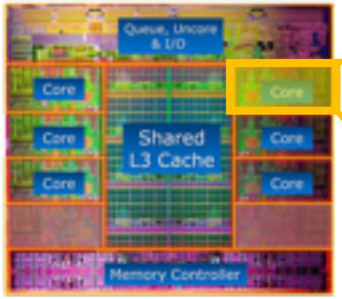
RISC CPU are making computation between 2 registers and stores the result in a register.

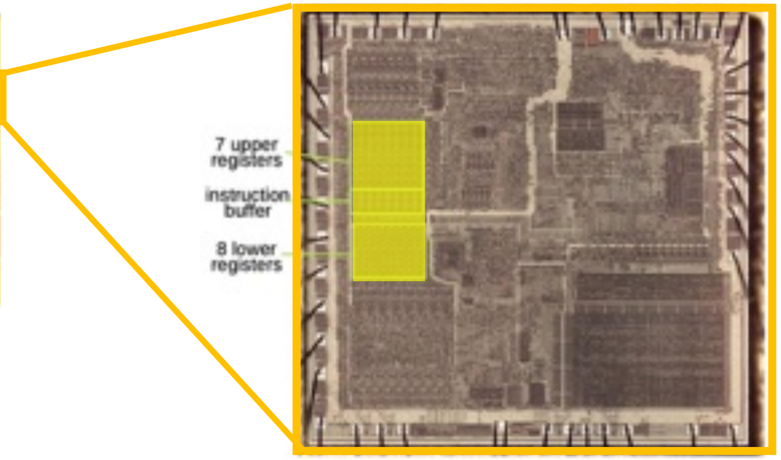
They are like local variable with the difference they are all defined and not extensible.

Why so much level of memory



Inside the CPU Core





Rq : This image and the zoomed one are totally different generations of processor.

Intel 8086
1978

COMPUTER MAIN COMPONENTS

3 Arithmetic and Logic Unit

This component is the making the computation.

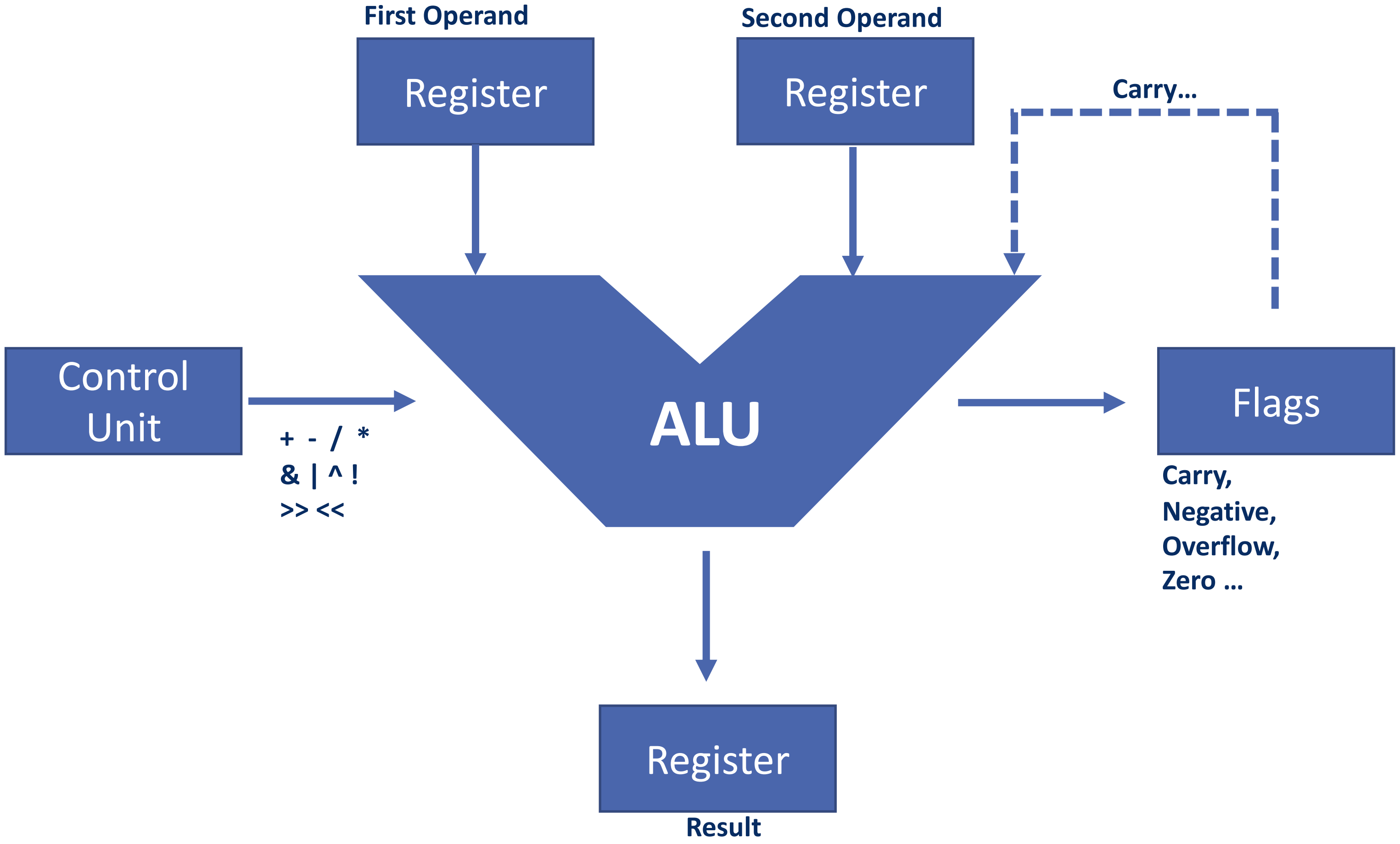
They can be arithmetical (+ - * /) or logical (And, Or, Xor, Not, Rotations …)

Two values are taken, usually, from registers. The result is also stored in a register, sometime one of the same used in input to reduce the instruction size.

In CISC processor ("Complex Instruction Set Computer"), the source and destinations can be memory.

In RISC processor ("Reduced Instruction Set Computer"), source and destination are usually only registers.

The result of the computation generates flags, they can be used in other computation of in instructions like conditional jumps.

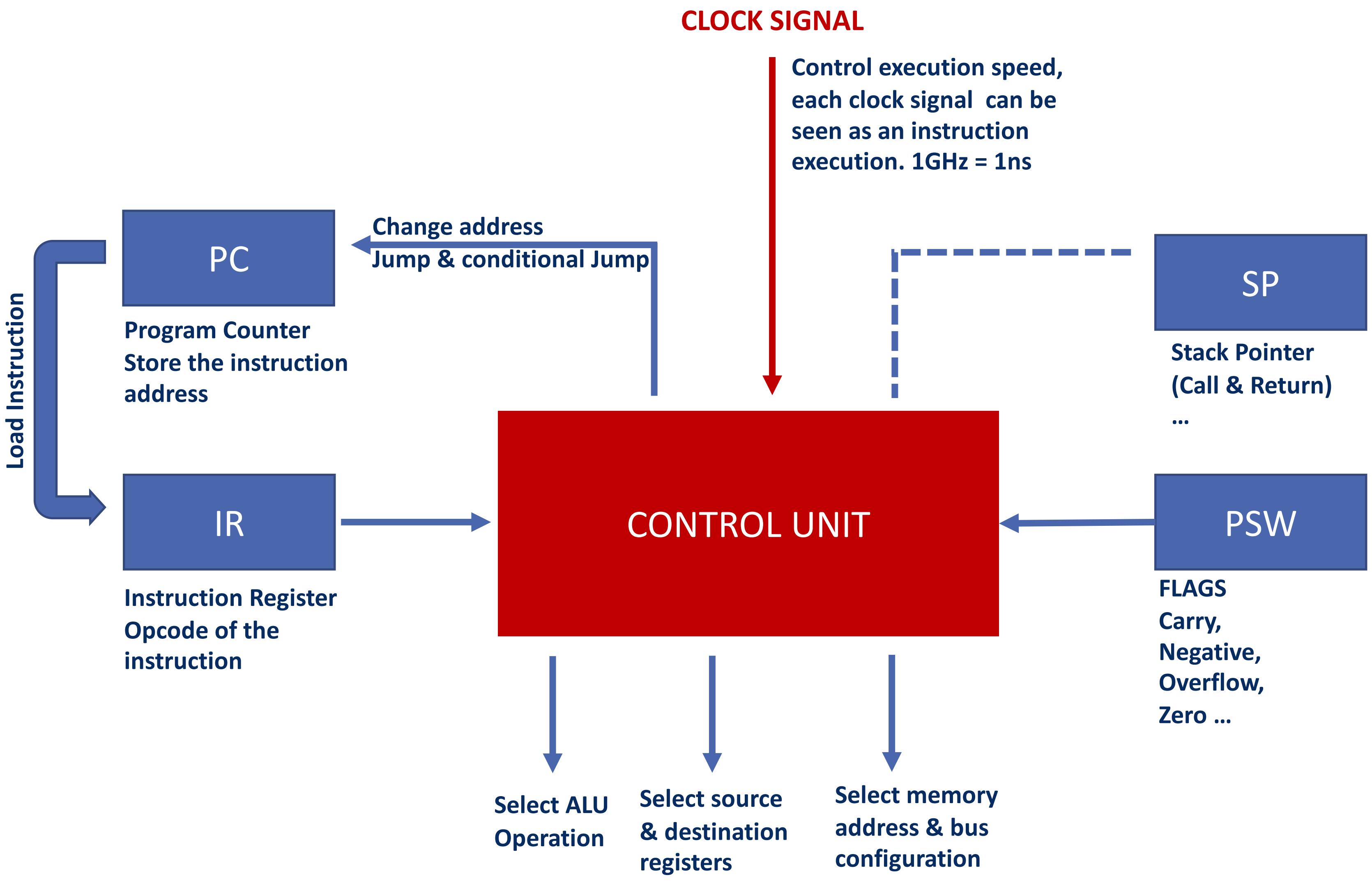


COMPUTER MAIN COMPONENTS

4 CONTROL UNIT

The control unit is the brain of the CPU, from an instruction it determines the right signals to send to the other components of the processor to perform the expected actions.

Modern Control Units are capable to run multiple instruction thread in parallel (Hyper Threading) or to dynamically reorder the instructions to optimize performance.



CLOCK SIGNAL

Control execution speed, each clock signal can be seen as an instruction execution. 1GHz = 1ns

Change address
Jump & conditional Jump

Program Counter
Store the instruction address

Instruction Register
Opcode of the instruction

Stack Pointer
(Call & Return)
…

FLAGS
Carry,
Negative,
Overflow,
Zero …

Select ALU Operation

Select source & destination registers

Select memory address & bus configuration

Author – Paul Pinault / Disk91.com

COMPUTER MAIN COMPONENTS

5 Input / Output

There are many peripherals used by a computer, some of them are internals:

- FPU (Floating Point Unit)
- MMU (Memory Management Unit)
- GPU (Graphical Processing Unit)
- AES-NI (Encryption)
- AVX (Advanced Vector Extension)
- …

There are also external peripherals

- Networks: Ethernet, WiFi, Bluetooth
- Video: HDMI, VGA…
- Extension: USB

COMPUTER MAIN COMPONENTS

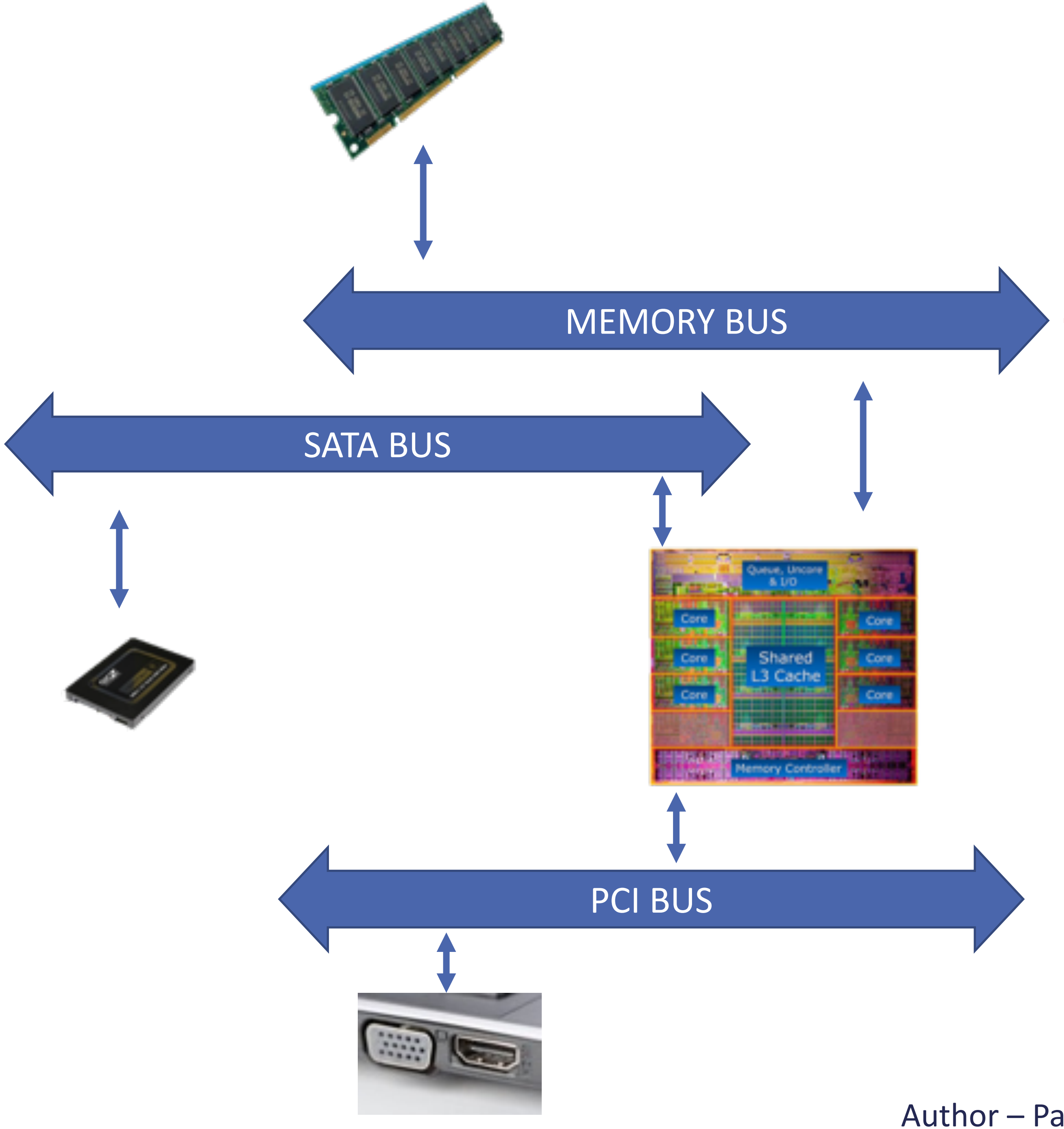
6 BUS

A bus is a group of wires used to interconnect the different components altogether. There are ADDRESS, DATA and CONTROL signals on it.

As it is not possible to pass everything within a single bus, different bus exists.

Each bus has some specificities related to the type of transported data.

Example:
- USB
- Memory BUS
- HDMI



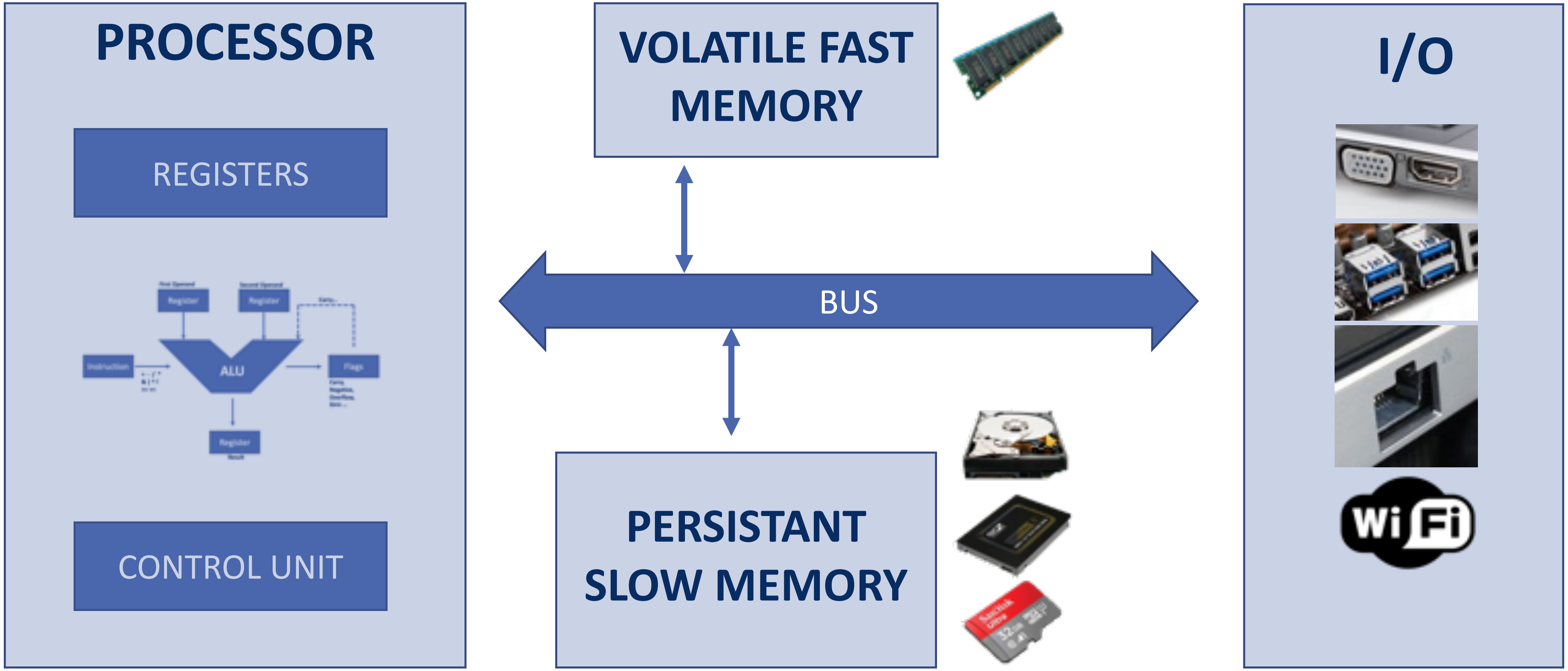
MEMORY BUS

SATA BUS

PCI BUS

Signals:
- Address (8-64bits)
- Data (8-64 bits)
- Controls
  - Read
  - Write
  - Reset
  - …

VON NEUMANN ARCHITECTURE



PROCESSOR

REGISTERS

CONTROL UNIT

VOLATILE FAST MEMORY

BUS

PERSISTANT SLOW MEMORY

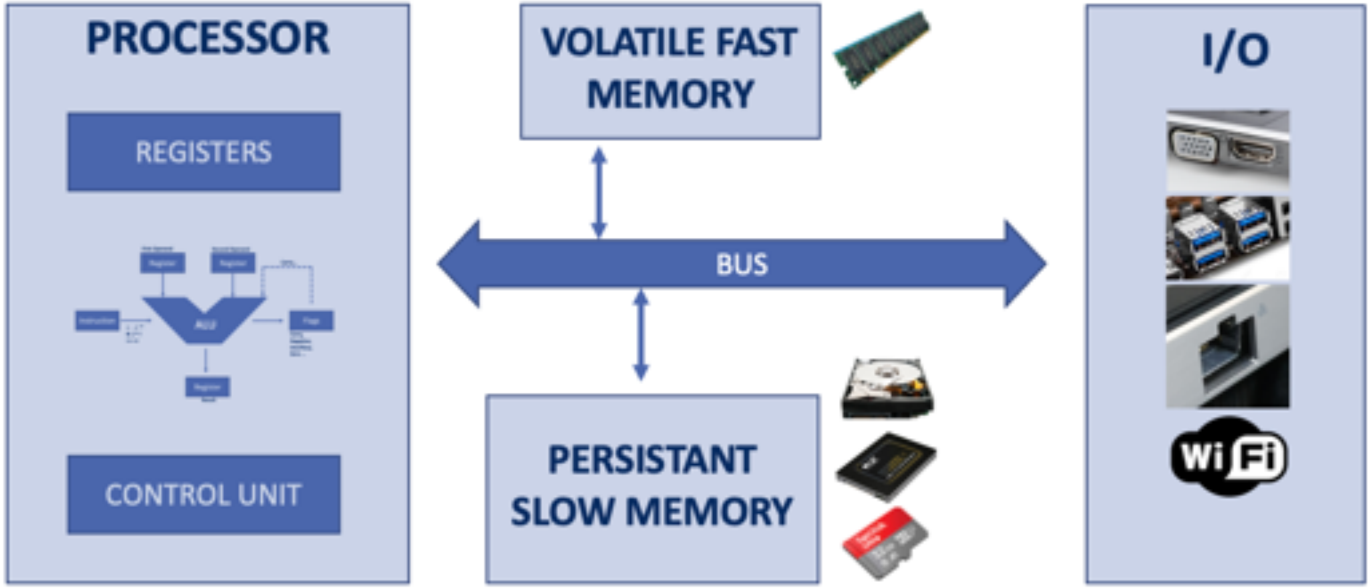
I/O

VON NEUMANN ARCHITECTURE

This architecture has been designed in 1945 and implemented in EDVAC computer.

The Intel 8086, now Core i3,i5,i7… are all based on this architecture.

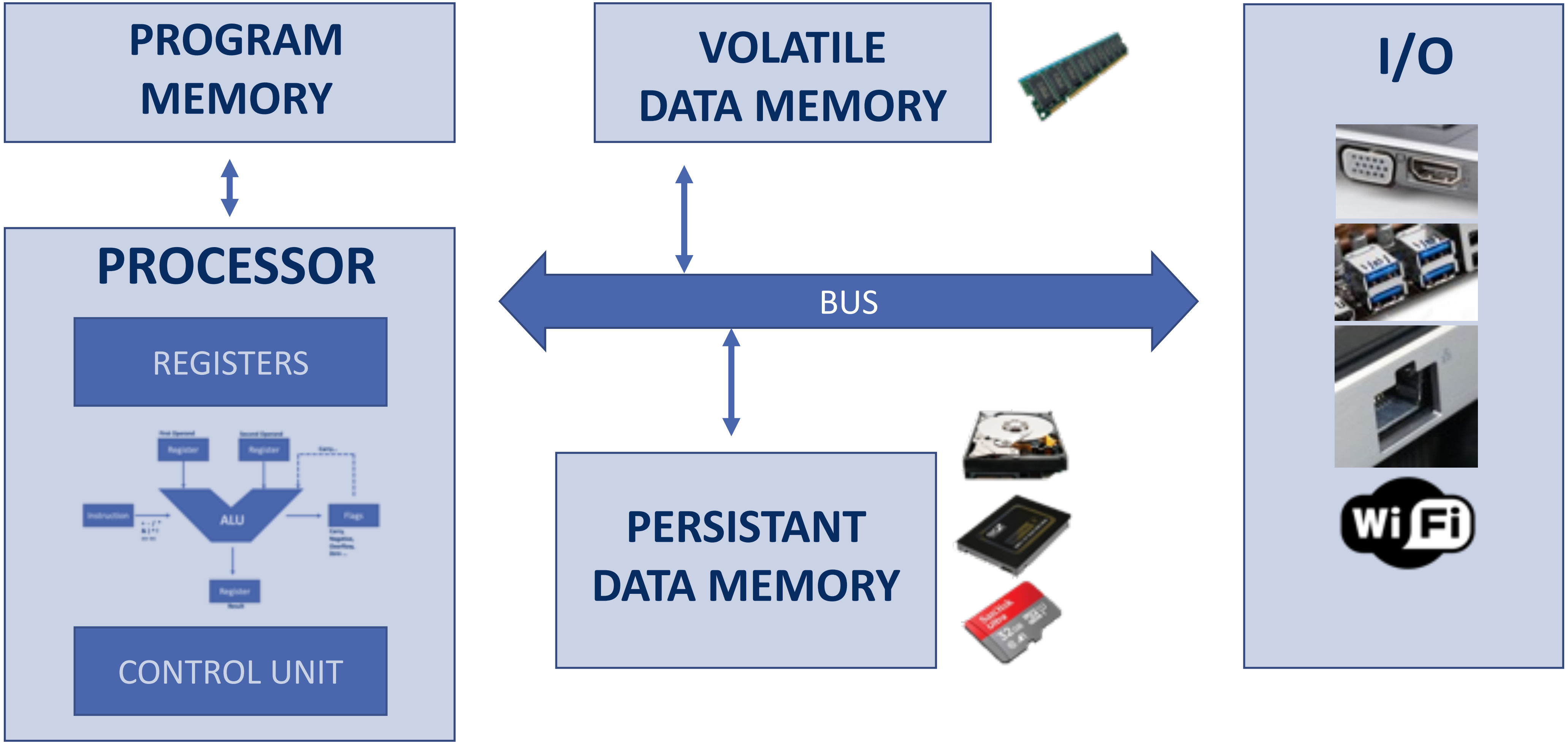
One single memory zone for program and data



Control Unit and Processing Unit are separated

System extended with I/O interfaces.

HARVARD ARCHITECTURE



PROGRAM MEMORY

PROCESSOR

REGISTERS

CONTROL UNIT

VOLATILE DATA MEMORY

BUS

PERSISTANT DATA MEMORY

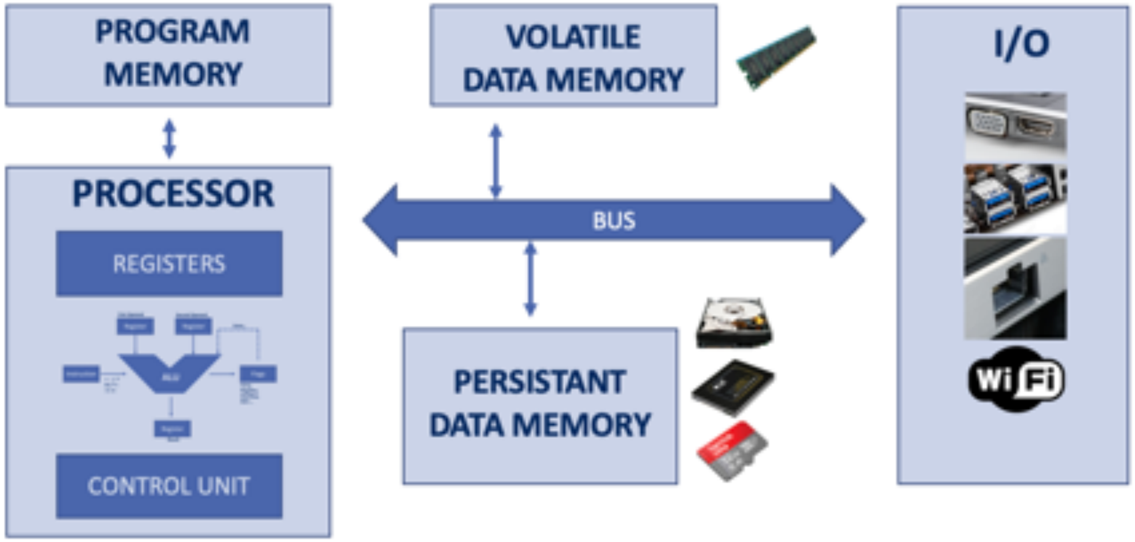
I/O

HARVARD ARCHITECTURE

This architecture has been designed in 1937. It is mostly implemented in embedded CPU.

ARDUINO Atmel MCU is based on a Harvard Architecture.

Program has is specific memory. No need to move program from persistent to volatile memory zone.



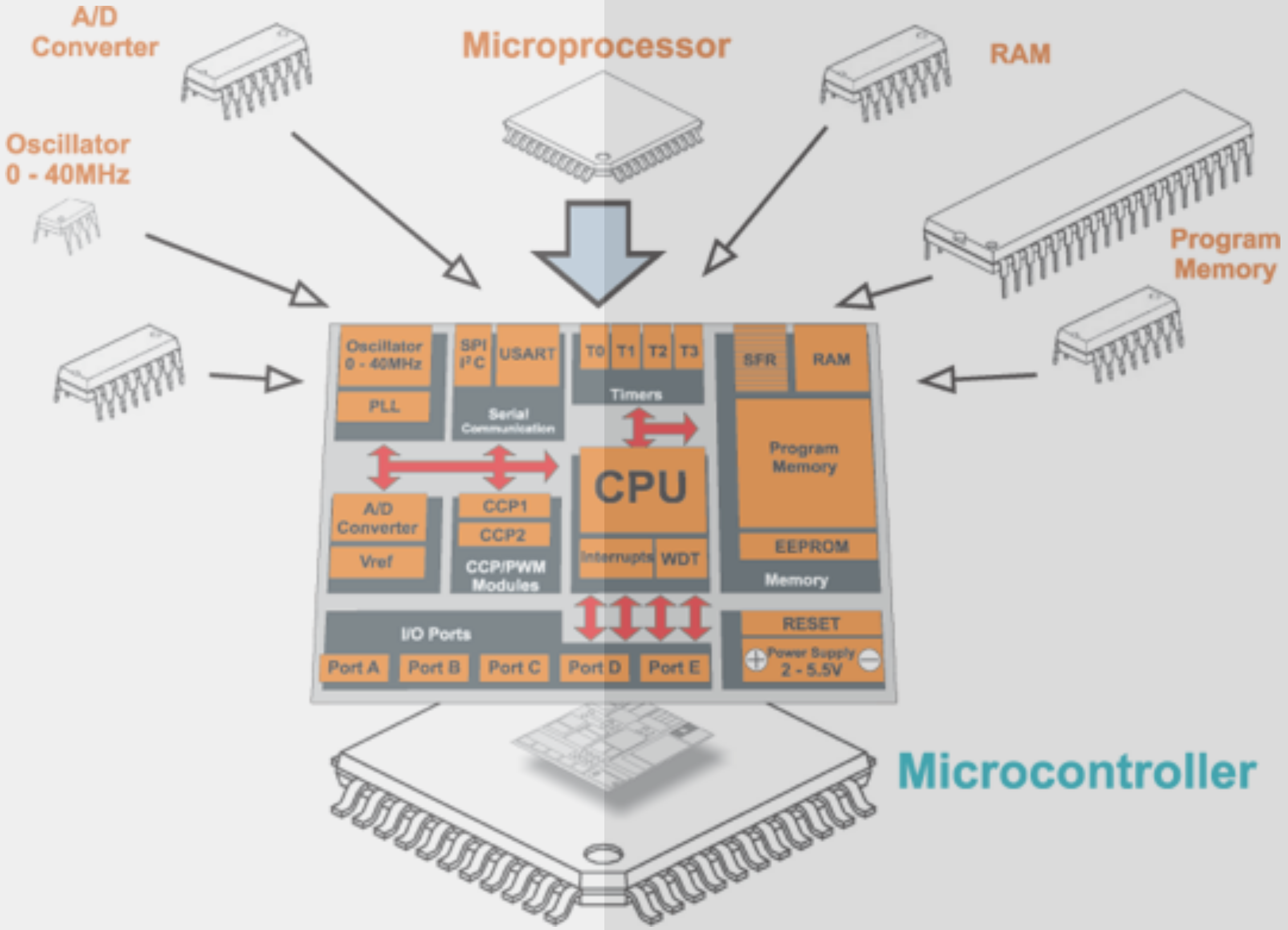
The program execution can’t modify the program itself.

Therefore, it is possible to perform a Data Read and a Program Read in a single cycle

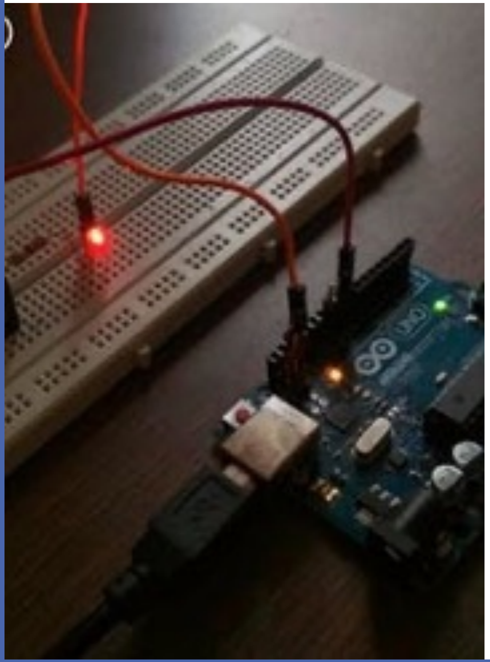
Computer architecture

CPU vs MCU

A CPU, like an Intel i7, is a computing component, it has no memory (other than cache), no IO, no storage…

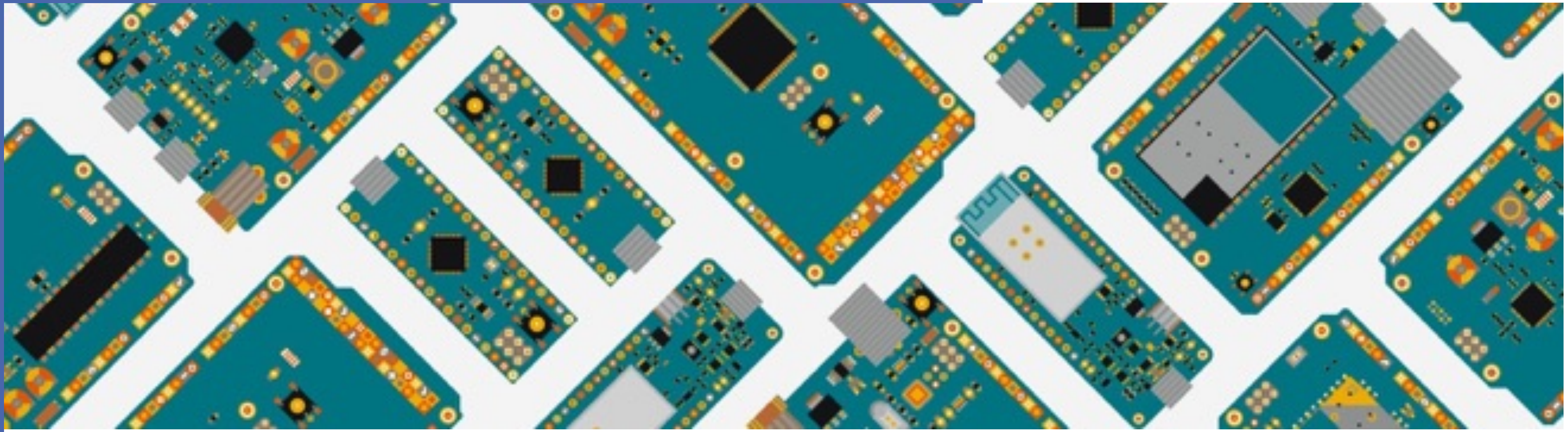


An MCU, like an Arduino, is a System On Chip, including a CPU, memory, flash, IO, communication unit, clock…

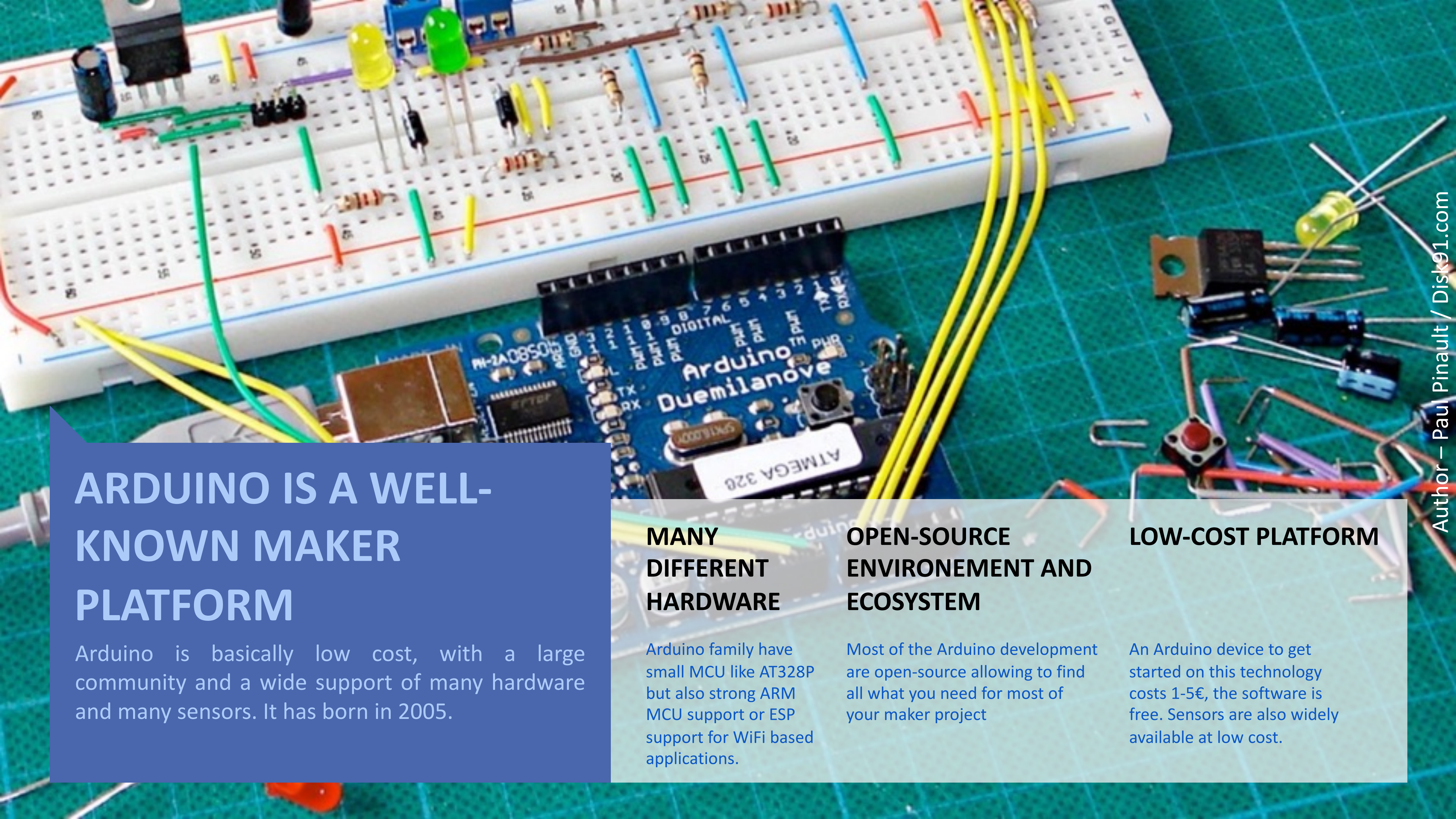


```
This example code is in the public domain.

http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press rese
void setup() {
  // initialize digital pin LED_BUILTIN as an outpu
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {$
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LE
  delay(1000);                       // wait for a
  digitalWrite(LED_BUILTIN, LOW);    // turn the LE
  delay(1000);                       // wait for a
}
```

ARDUINO PLATFORM, FOCUS ON ATMEGA 328P



Author – Paul Pinault / Disk91.com



Author – Paul Pinault / Disk91.com

ARDUINO IS A WELL-KNOWN MAKER PLATFORM

Arduino is basically low cost, with a large community and a wide support of many hardware and many sensors. It has born in 2005.

MANY DIFFERENT HARDWARE

Arduino family have small MCU like AT328P but also strong ARM MCU support or ESP support for WiFi based applications.

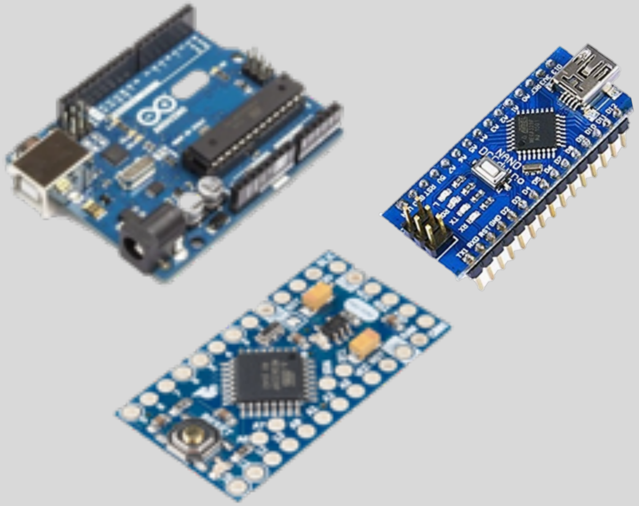
OPEN-SOURCE ENVIRONEMENT AND ECOSYSTEM

Most of the Arduino development are open-source allowing to find all what you need for most of your maker project

LOW-COST PLATFORM

An Arduino device to get started on this technology costs 1-5€, the software is free. Sensors are also widely available at low cost.

ARDUINO FAMILLY (in fact it is bigger than that… with hundreds of platforms)



ORIGINAL ARDUINO
ATMEL AT328P

FREQUENCY - 20MHz
RAM - 2KB
FLASH - 32KB
PRICE - $2

ARDUINO ZERO
ARM SAMD21 M0

FREQUENCY - 48MHz
RAM - 32KB
FLASH - 256KB
PRICE - $8-$15

ESP32 WiFi/BLE

FREQUENCY - 160MHz
RAM - 512KB
FLASH - 2-4MB
PRICE - $3-$4

RaspberryPi Pico
Dual Core ARM M0+

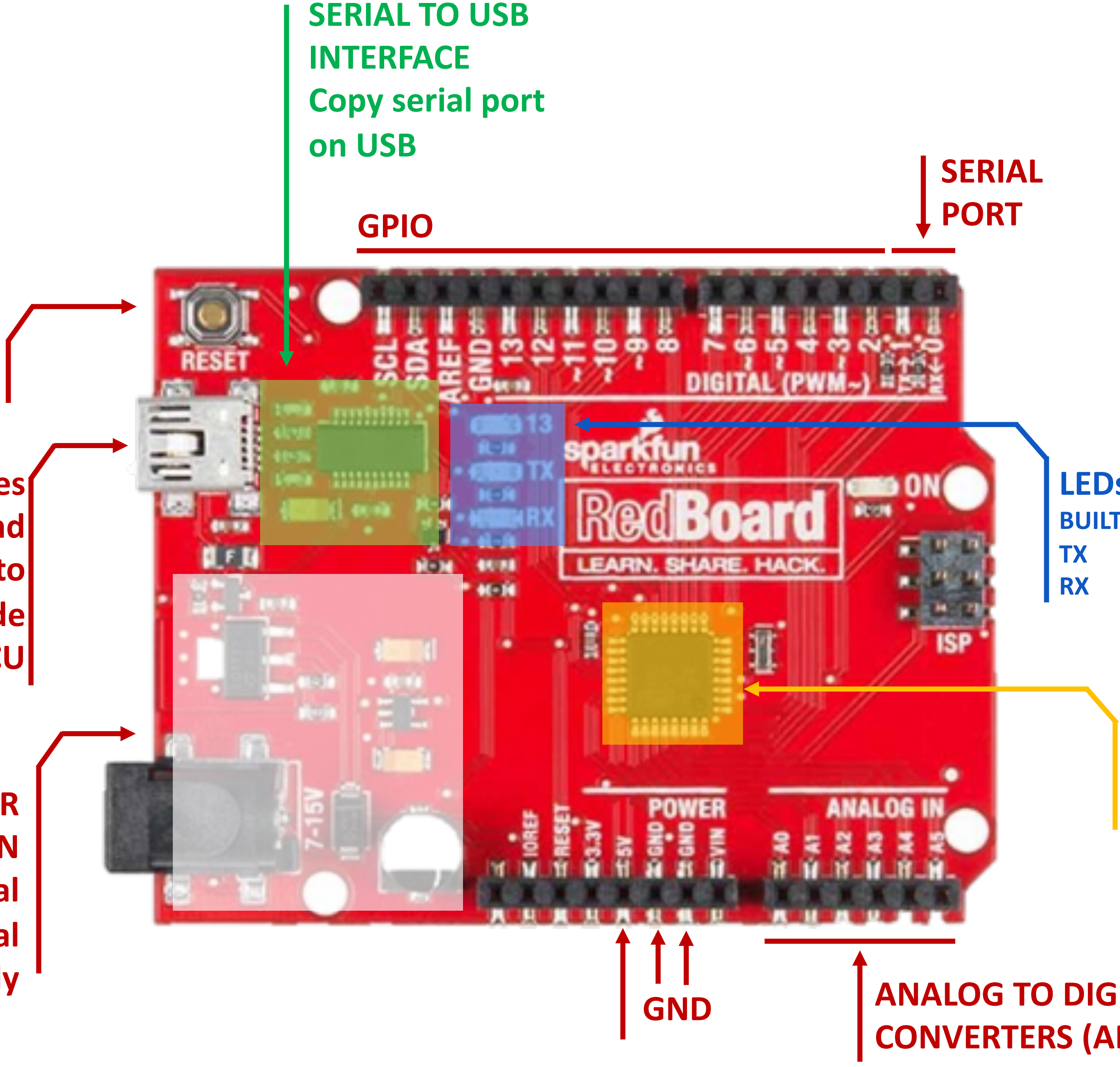
FREQUENCY - 133MHz
RAM - 264KB
FLASH - 2MB
PRICE - 4$

THE ARDUINO UNO BOARD

Here is a sparkfun board, this one is a clone of the ARDUINO UNO.

Form factor has been standardized to host HATs on top of it.

The MCU is a SMD version of the AT328P



SERIAL TO USB INTERFACE
Copy serial port on USB

GPIO

SERIAL PORT

RESET

USB provides power and allow to upload code into the MCU

LEDs
BUILTIN
TX
RX

ATMEGA 328P

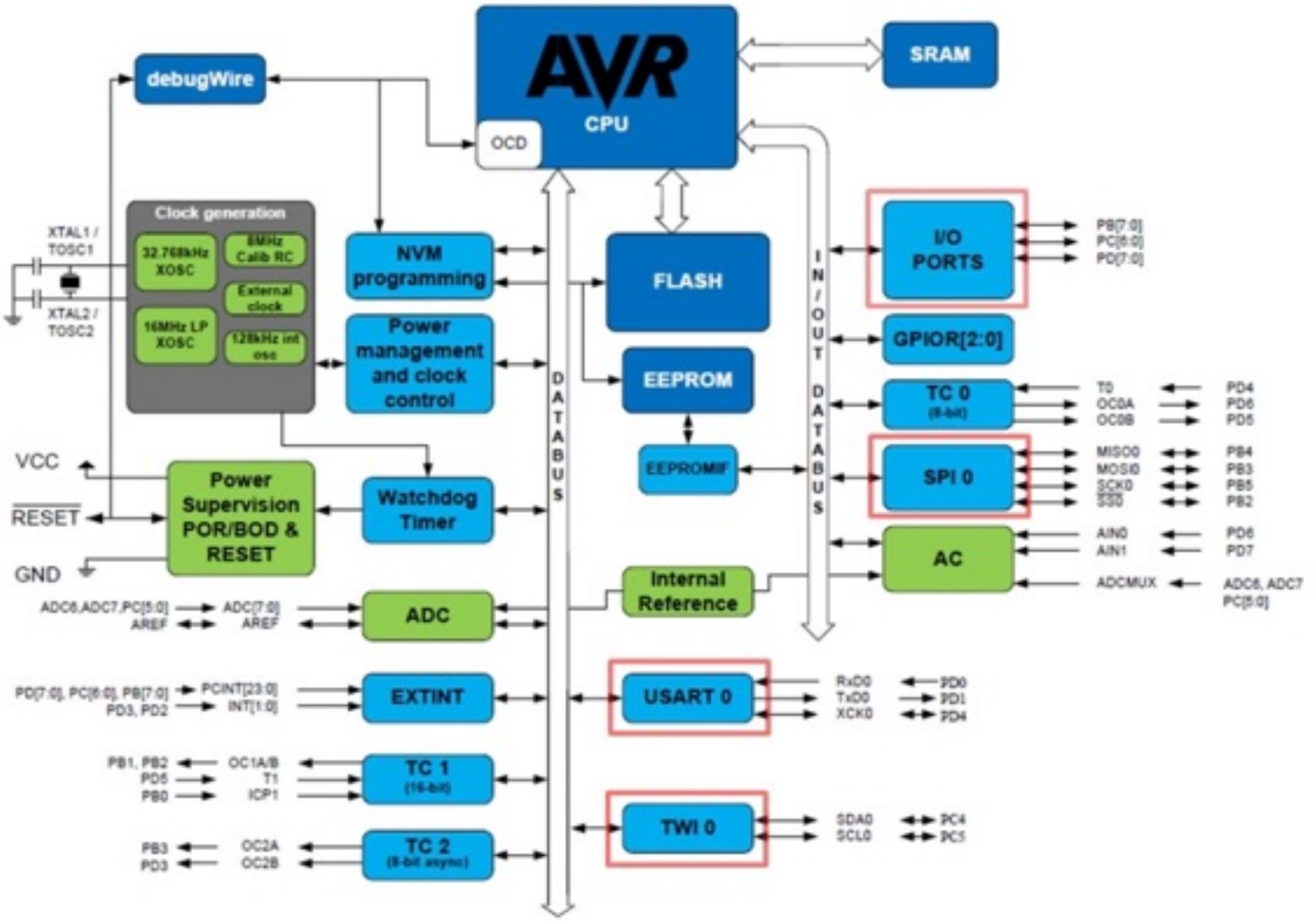
POWER REGULATION
From optional external supply

GND

5V DC

ANALOG TO DIGITAL CONVERTERS (ADC)

INSIDE THE ARDUINO MCU (AT328P)



FLASH and SRAM are separated and connected to the CPU core, typical of a HARVARD architecture.

Many I/O subsystems (USART, TWI, SPI, ADC, GPIO)

Power management for low power mode.

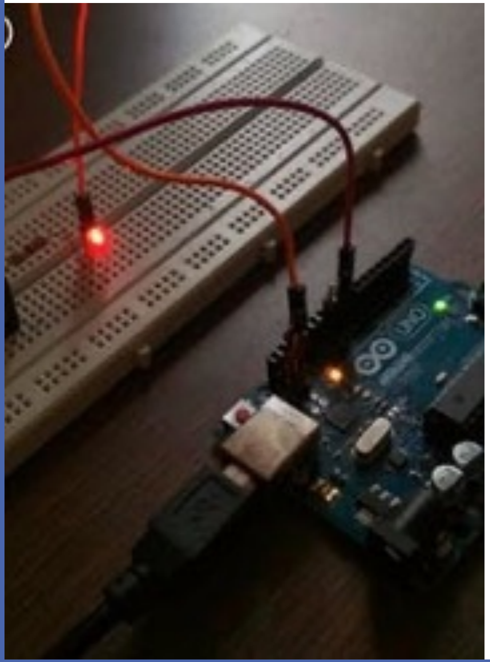
Internal clock generation

We have a typical MCU, you have no other components to add on the board to make a working system.

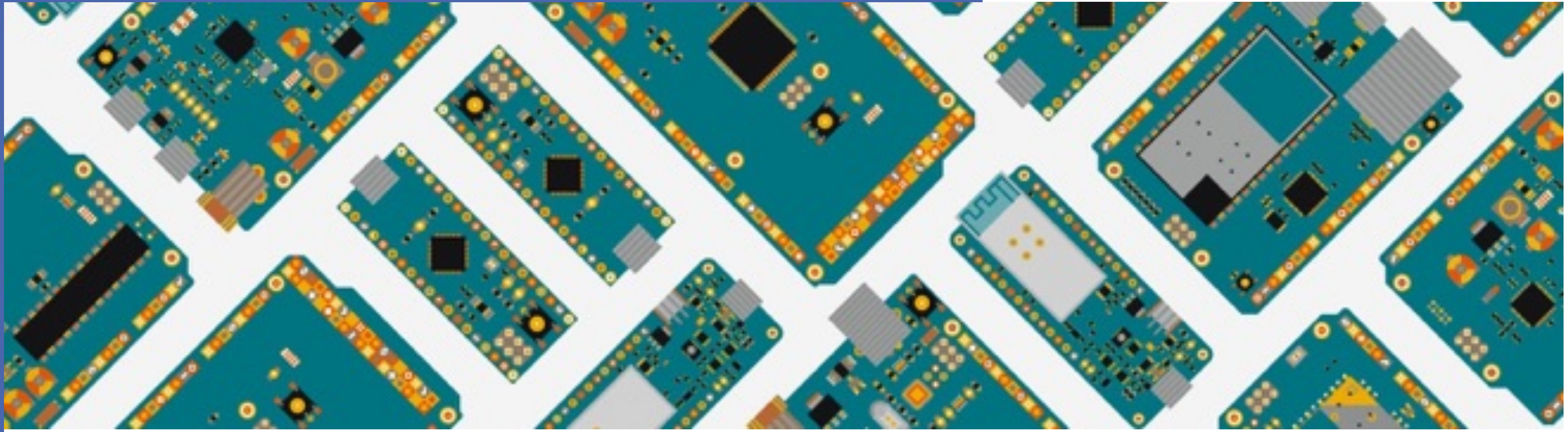
Author – Paul Pinault / Disk91.com

ARDUINO

HELLO WORLD



```
This example code is in the public domain.

http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin LED_BUILTIN as an output
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LE
  delay(1000);                       // wait for a
  digitalWrite(LED_BUILTIN, LOW);    // turn the LE
  delay(1000);                       // wait for a
}
```

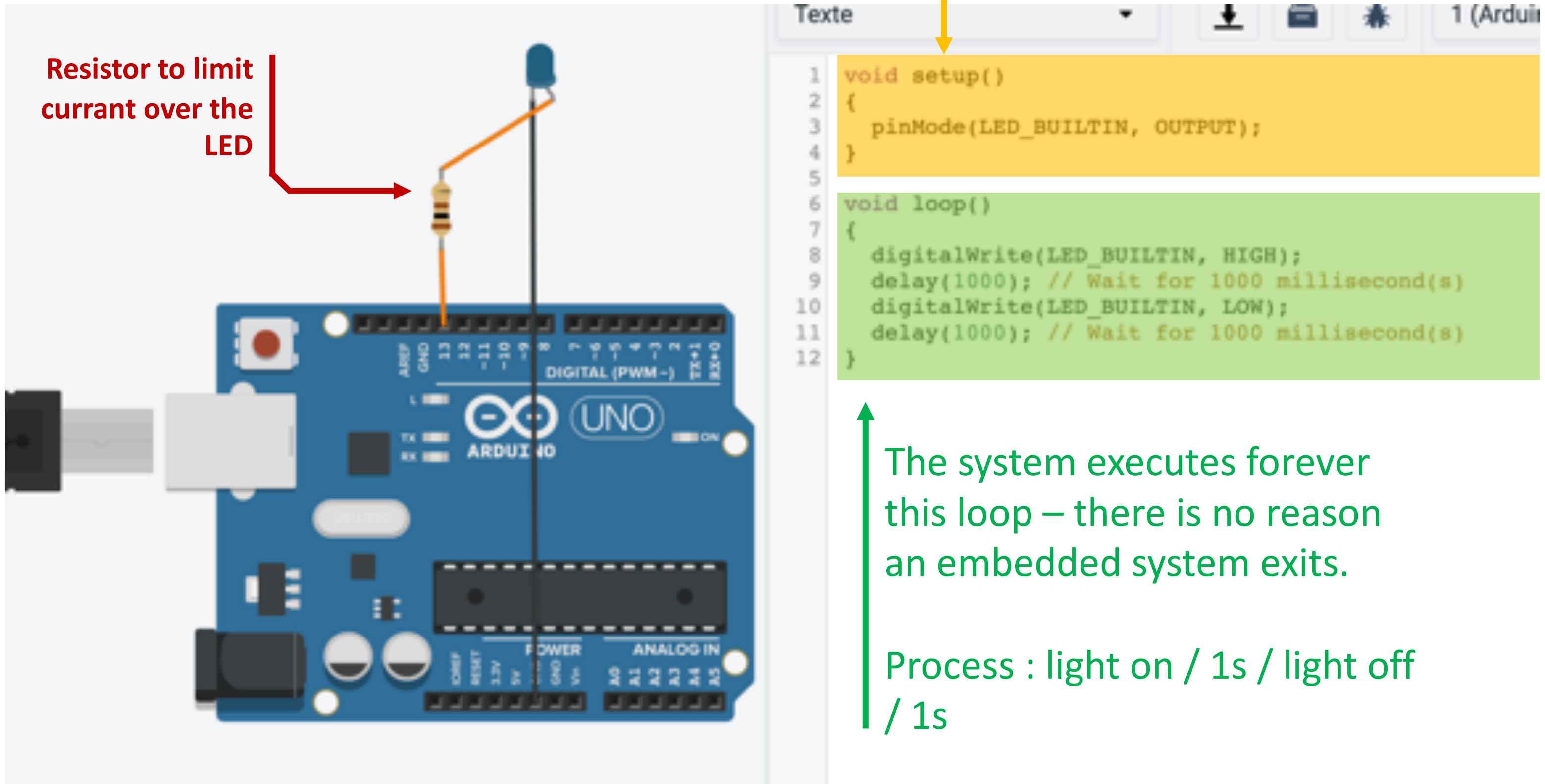


Author – Paul Pinault / Disk91.com

ARDUINO BASICS

Hello World

One Time operation, executed on power-on / reset.
We setup the GPIO as Output to deliver energy over it.



```
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000); // Wait for 1000 millisecond(s)
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000); // Wait for 1000 millisecond(s)
}
```

Resistor to limit currant over the LED

The system executes forever this loop – there is no reason an embedded system exits.

Process : light on / 1s / light off / 1s

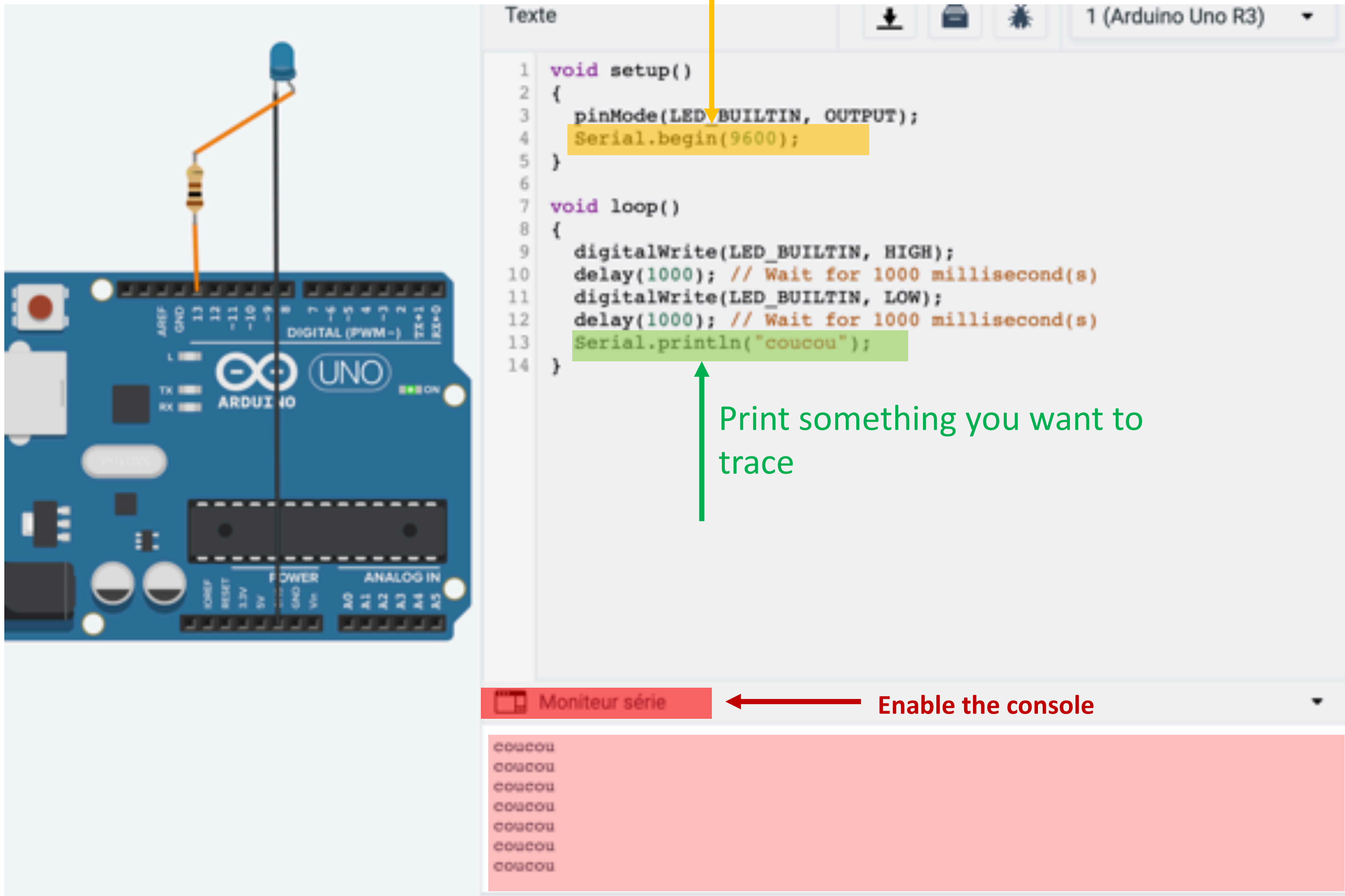
ARDUINO have a hidden main like this:

```
main() {
  setup();
  while (1) loop();
}
```

You need to setup the hardware configuration in the setup() function.

Then you can loop forever to the function you want to realize.

Debugging



```
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000); // Wait for 1000 millisecond(s)
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000); // Wait for 1000 millisecond(s)
  Serial.println("coucou");
}
```

```
coucou
coucou
coucou
coucou
coucou
coucou
coucou
```

Init the console display with a given speed (9600 bit/s)

Print something you want to trace

Enable the console

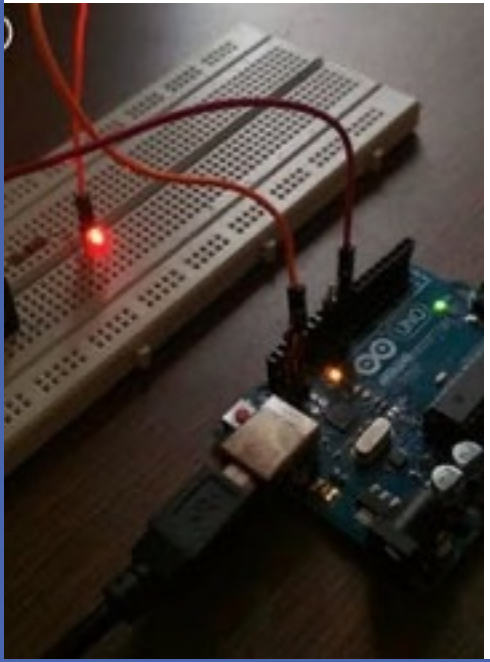
Display results

One of the ways to debug, but also used for communicating with computer or an external device is to use the serial port.

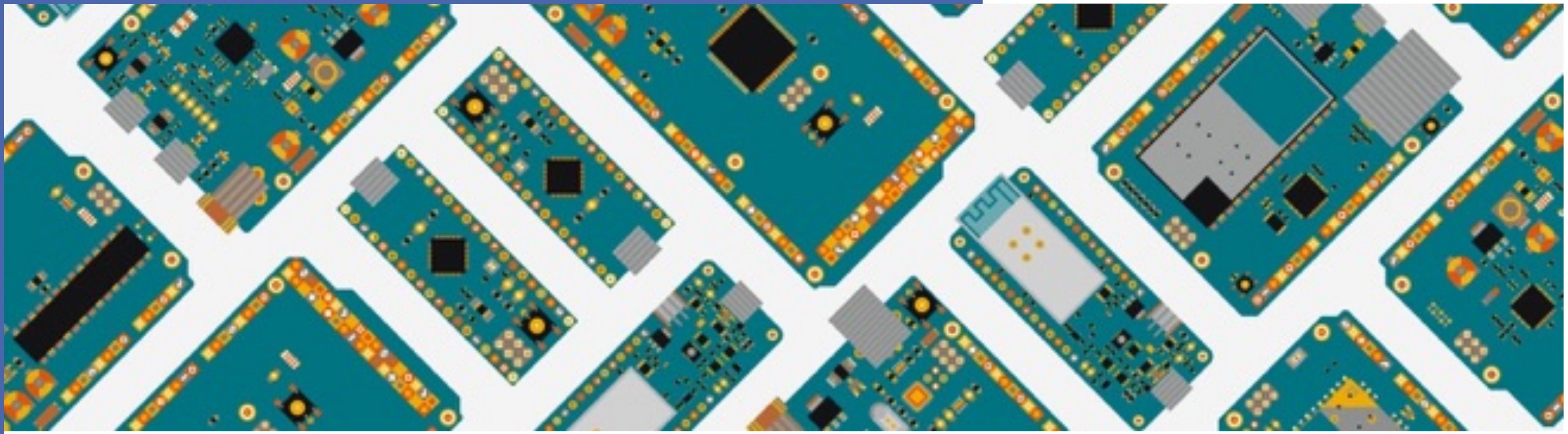
A Serial port is like an USB communication transferring data over a serial line (we will detail later). It can pass through USB but can also be used as direct link between two devices.

The Serial monitor displays what has been printed over the serial line. It can be string but also variables.

How works GPIOs ?
General Purpose Input & Output

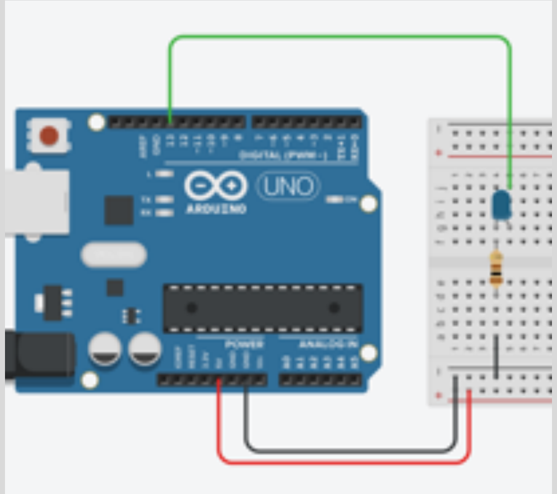


```
  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin LED_BUILTIN as an output
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {$
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED
  delay(1000);                       // wait for a s
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED
  delay(1000);                       // wait for a s
}
```



Author – Paul Pinault / Disk91.com

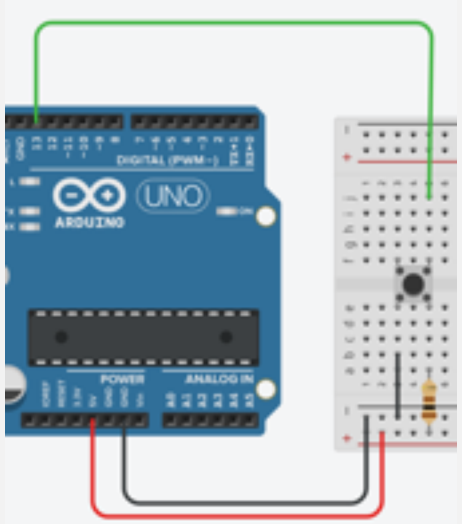
GPIO different usages



DIGITAL OUTPUT

The GPIO take a value 0 or 1 corresponding to GND or VDD.

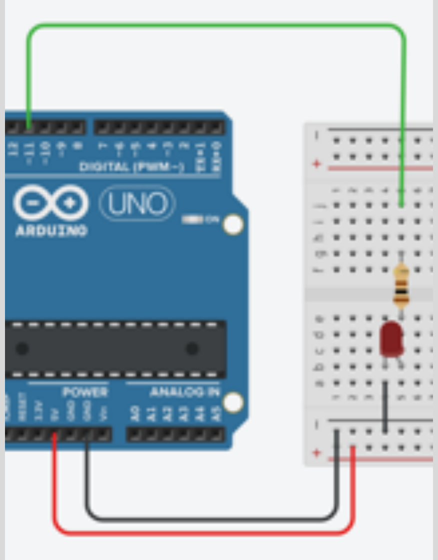
It can be used to switch a LED ON or OFF as in this example.



DIGITAL INPUT

The GPIO reads voltage GND or VDD to determiner a 0 or 1 value.

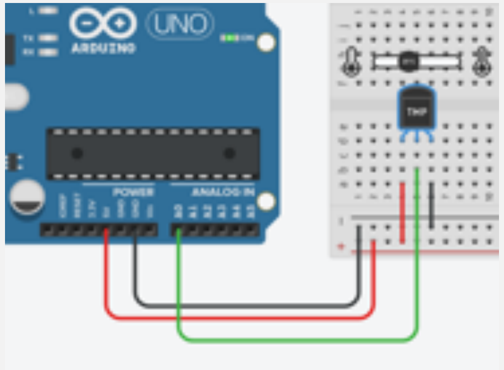
It can be used to get the status of a switch button.



ANALOG OUTPUT

The GPIO voltage is set to a given value between GND and VCC.

It can be used to modulate the light of a LED.



ANALOG INPUT

The GPIO voltage is read and converted into a decimal value.

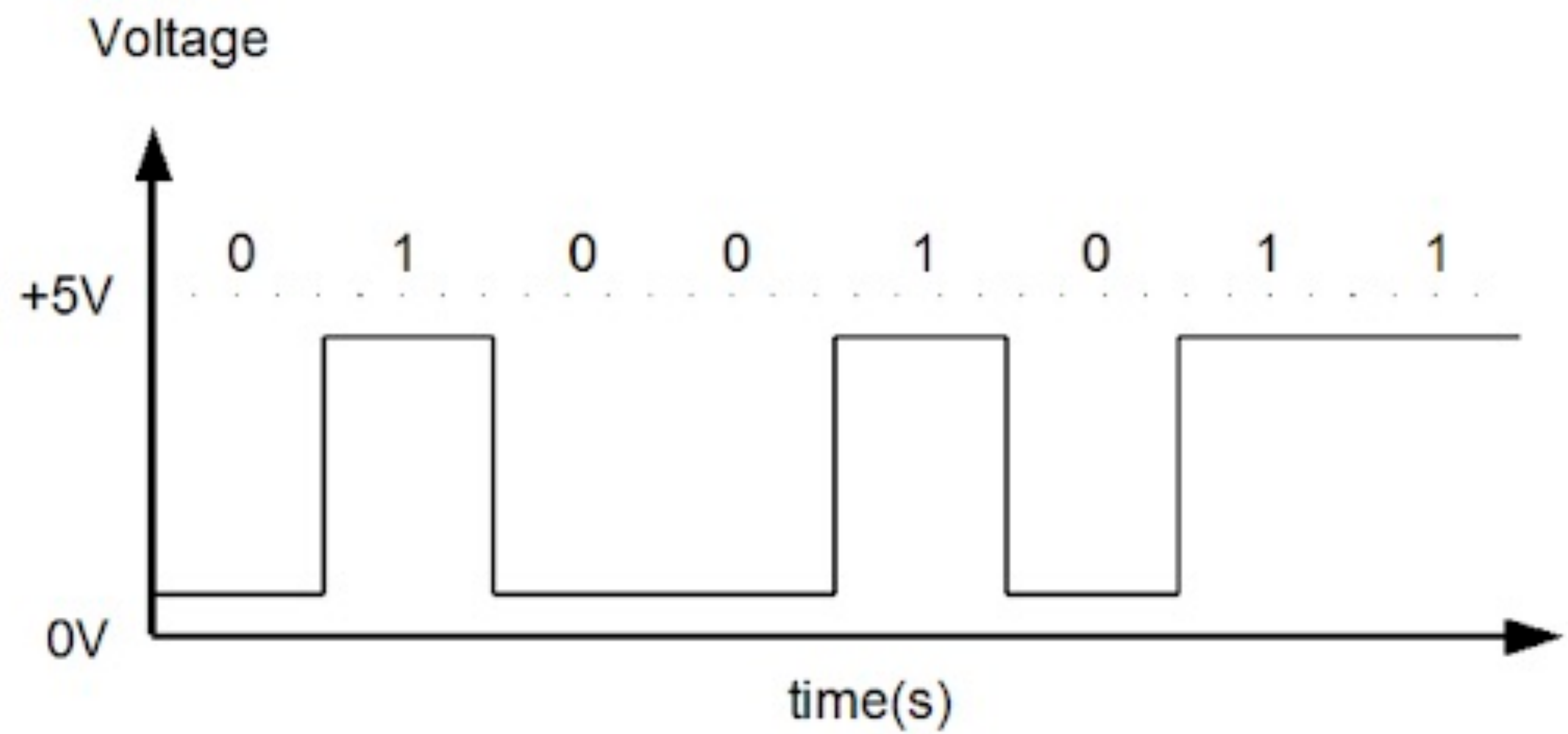
It can be used to measure the environmental Temperature.

DIGITAL SIGNAL

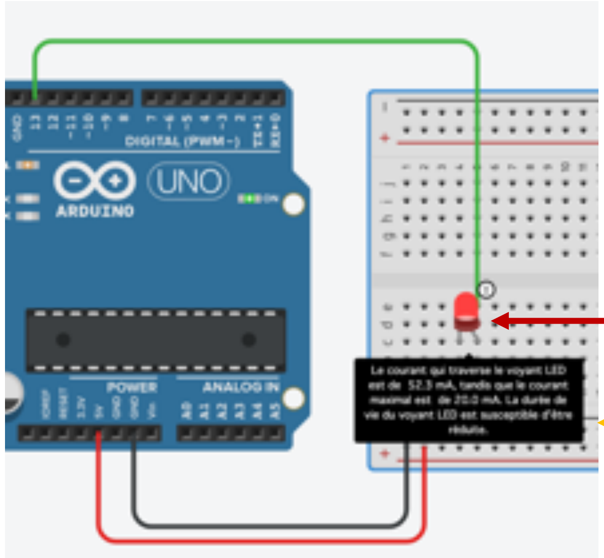
A digital signal is a 0 or 1 value. Usually, 0Volt or VDD.

VDD can have different Voltage : 3,3V most common, 5V like with AT328P (TTL)

In TTL, the 1 value starts over 2Volts when read.



DIGITAL OUTPUT



Le courant qui traverse le voyant LED est de 52.3 mA, tandis que le courant maximal est de 20.0 mA. La durée de vie du voyant LED est susceptible d'être réduite.

Without a resistor, there is no current limitation over a LED. This one can burn dur to overheat consequence of over-currant.

Even without resistor, the currant is limited by the capacity of a GPIO to deliver current.

A GPIO can deliver a certain current as a maximum. This is specified in the micro-controller Datasheet.
This is protecting the micro-controlleur from over-heating.

It is like if the GPIO has a protection resistor in series. A usual value is 100 Ohm.

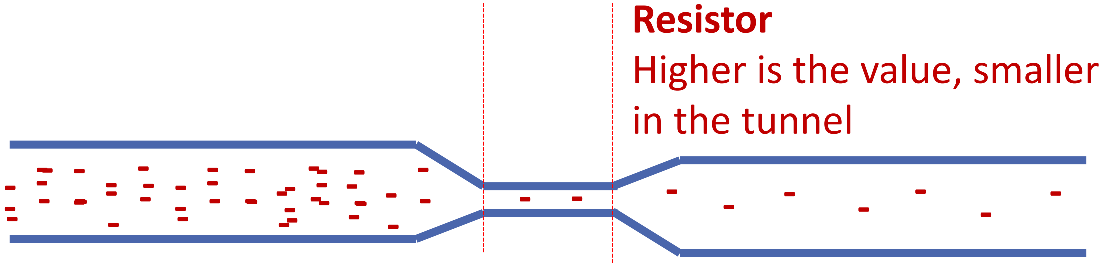
For 5V, it means I = U / R = 5 / 100 = 50mA.

Configure GPIO as a digital output:
pinMode(pin,OUTPUT);

Write a value:
digitalWrite(pin,HIGH/LOW);

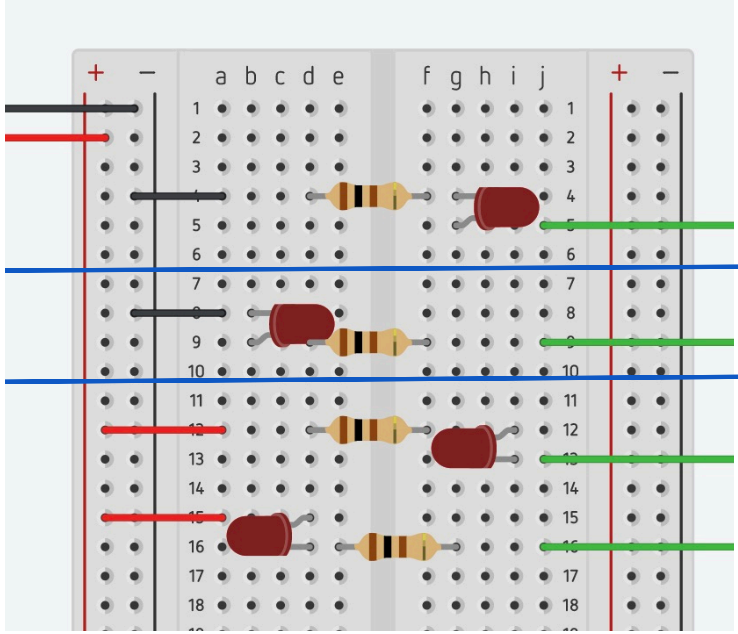
DIGITAL SIGNAL

DIGITAL OUTPUT



Resistor
Higher is the value, smaller in the tunnel

The led can ACTIVE HIGH GPIO value 1, switch the LED ON



Resistor can be after the LED

Resistor can be before the LED

The led can be ACTIVE LOW GPIO value 0, switch the LED ON

We use resistors to limit the current inside the circuit.

The resulting maximum current is I = U/R

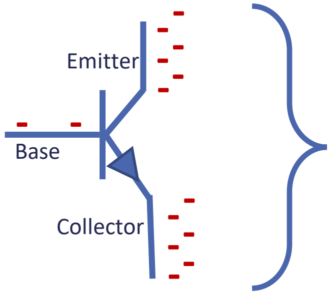
A digital signal 1 does not mean ACTIVE. In many cases we use ACTLIVE LOW logic when 0 means ACTIVE.

A non connected pin value is 1, the active ACTIVE LOW logic ensures the signal will not be active when the pad is left unconnected.

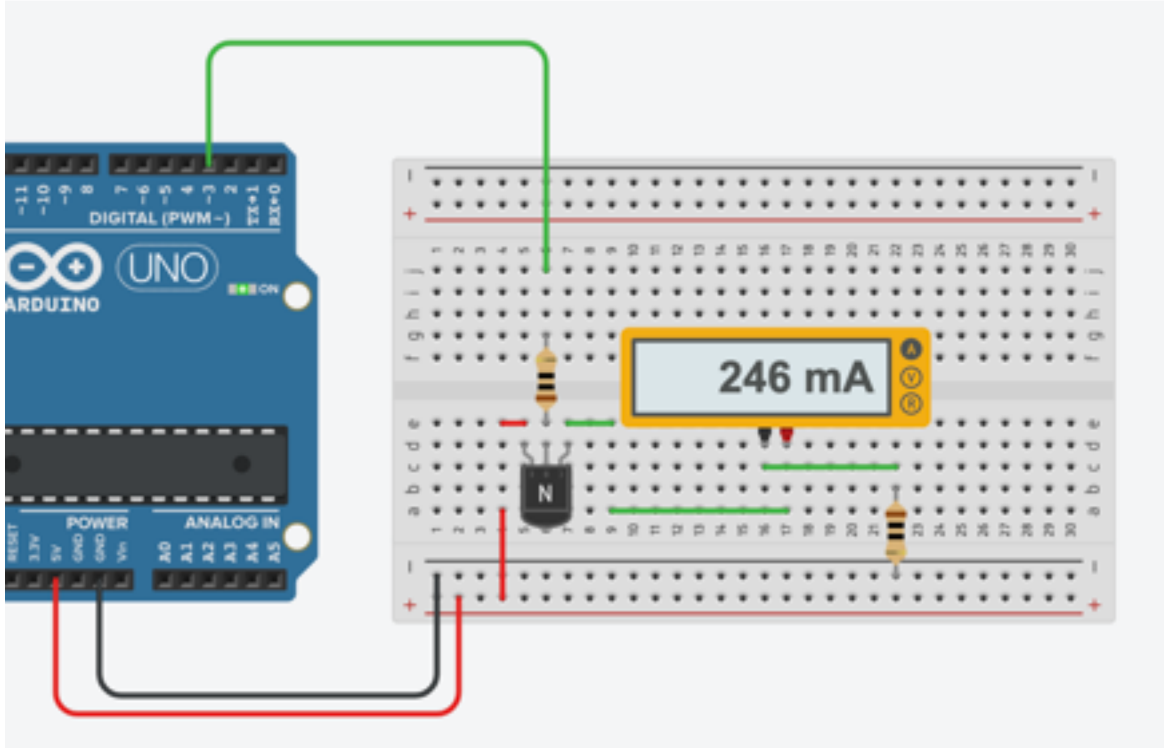
ACTIVE LOW pads are indicated with a # or _____ notation

Author – Paul Pinault / Disk91.com

DIGITAL OUTPUT



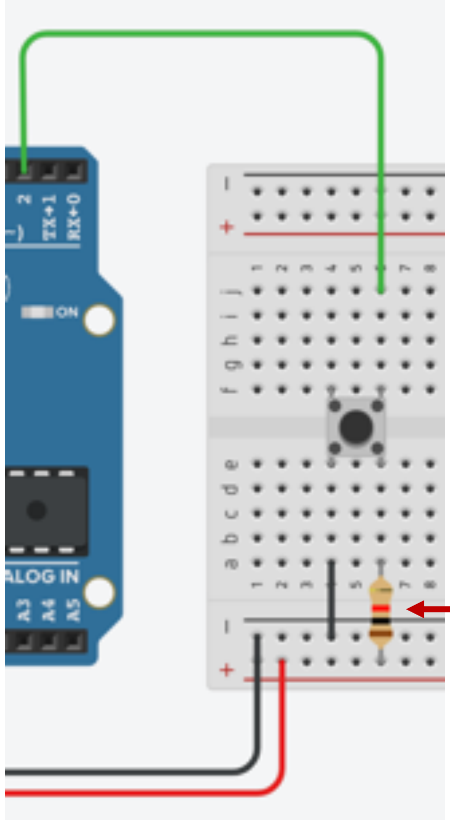
We use a transistor as a switch by saturating the Base.
This allows to have a larger current between Collector and Emitter than we can have from a GPIO



When we need more power than the GPIO can deliver, like for powering a motor we use a transistor as a switch to close a circuit with higher currant.

Make sure you will limit the currant through this circuit and make sure the source (here the Arduino board) is able to deliver the currant.

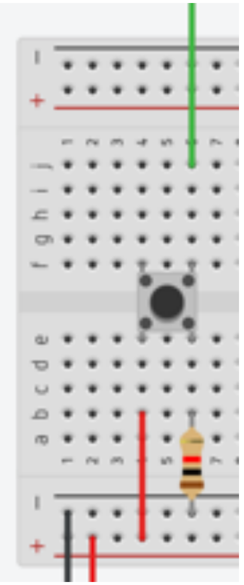
DIGITAL INPUT

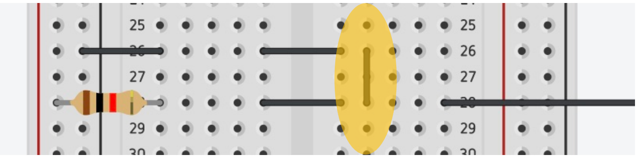


A Pull-UP resistor ensure:

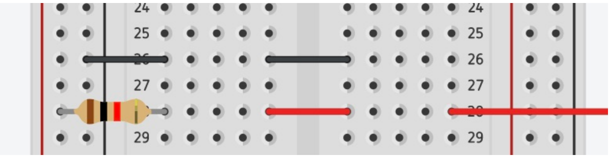
- A defined (1) value when the button is not pressed
- Not to have a shortcut when the button is pressed.

We could have a Pull-Down and ACTIVE HIGH with another configuration





Button PRESSED equivalent circuit (value 0)



Button RELEASED equivalent circuit (value 01)

A GPIO configured as an input will detect a LOW (0) value as soon as the input Voltage is 0.3 x VDD so 1.5V for 5V VDD.

A left unconnected pad will be HIGH (1).

When using a switch, we need a pull-up or pull-down to close the circuit. (it can be internal)

Configuring the pin:

pinMode(pin, INPUT);

pinMode(pin, INPUT_PULLUP);

Read a value:

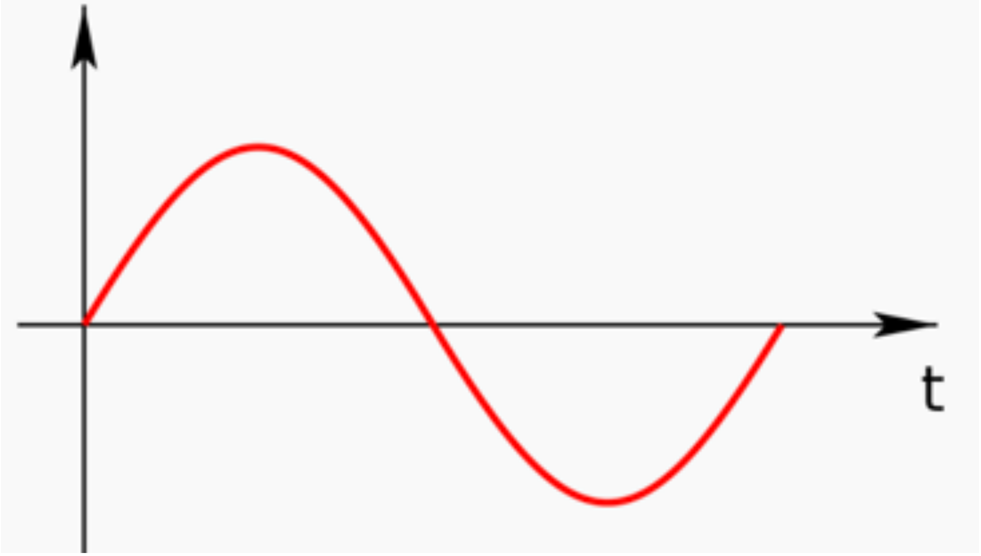
x = digitalRead(pin);

ANALOG SIGNAL

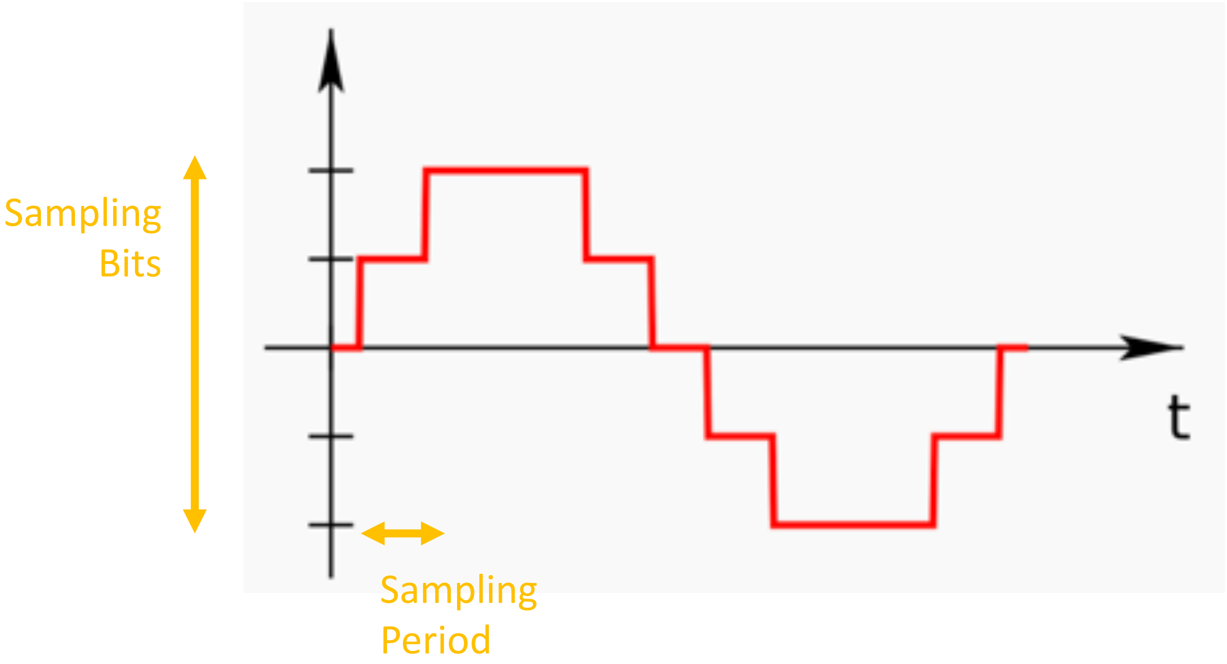
An analog signal is a value between 0V and VDD. Any value.

It can be continuous like a temperature sensor.

It can be stepped when generated or captured by a digital system.



ANALOG
Signal as it is in the reality

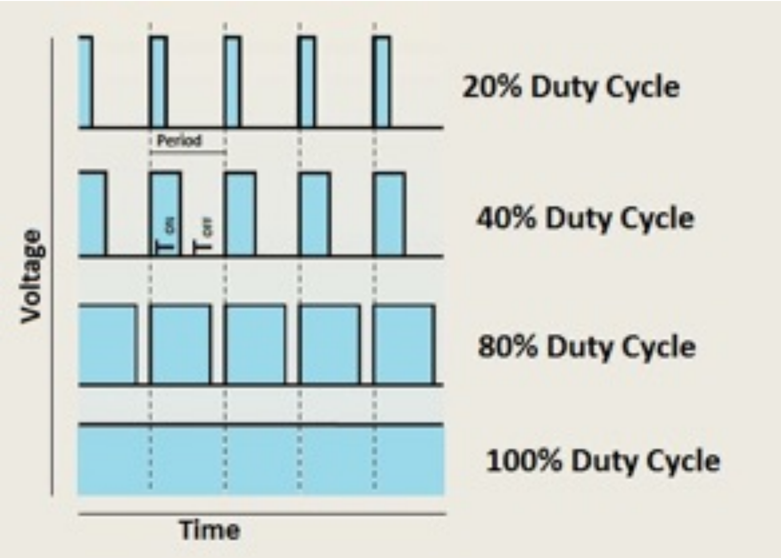


ANALOG
Signal as it is seen by a digital system

ANALOG OUTPUT

You can set values from 0 to 255 and so you have 19mV steps.
Frequency is about 500Hz

PWM – with VDD = 5V



Average – 1V

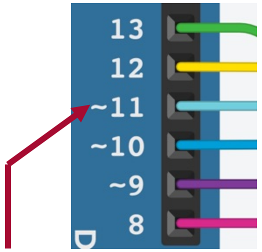
Average – 2V

Average – 4V

Average – 5V

The PWM signal generation is hardware generated. It takes no CPU time to manage this signal.

Only pins with ~ can be used as PWM / analog output



A DAC – Digital to Analog Converter can transform a decimal value into an analog value on a pin. But such a component is expensive.

Most of the MCU use PWM – Pulse Width Modulation - to simulate Analog output. But it is not, is is just averaging.

Configure GPIO as a digital output:
pinMode(pin,OUTPUT);

Write a value:
analogWrite(pin,0..255);

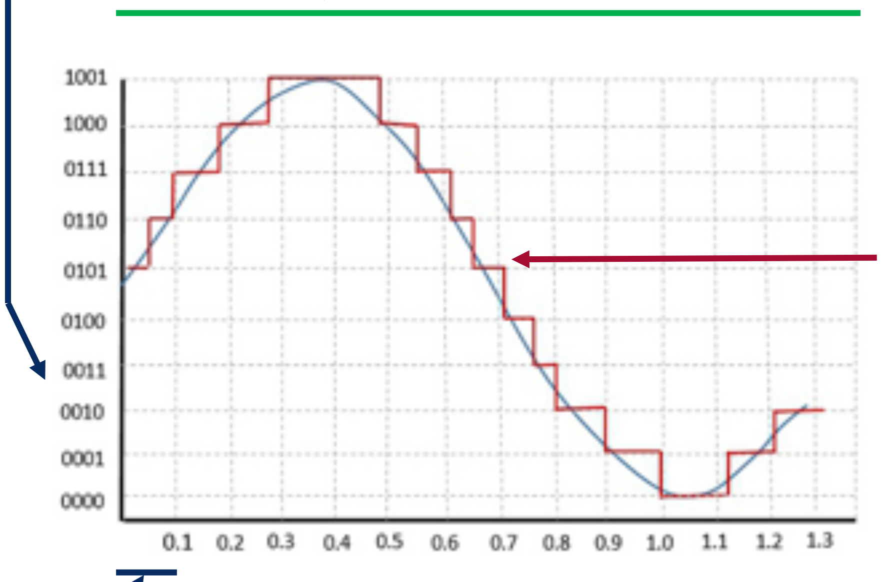
ANALOG INPUT

Conversion resolution in bits. Default is 10bits, can be 8, 12, 16 depending on hardware.

The resolution corresponds to the number of different values you can have between 0V and the reference

Reference Voltage – maximum voltage, represented by the highest possible value

A reference can be VDD, but the precision is not good, so precise voltage reference can be use like 1.1V or 2.56V … always under VDD



Sampling precision depends on resolution

Sampling Rate : number of conversions per seconds.

There is a minimum period for executing the conversion, this period depends on resolution.

Arduino UNO Resolutions:

10 bits @ 79KSpS

Arduino UNO References:

DEFAULT – VDD (5V)

INTERNAL1V1 – 1.1V

INTERNAL2V56 – 2.56V

EXTERNAL – AREF 0..5V

Conversion

(5000 / 1024) mV = 4.88mV / unit

(1100 / 1024) mV = 1.07mV / unit

(2560 / 1024) mV = 2.5mV / unit

An ADC – Analog to Digital Converter can transform an analog signal from a pin into a digital value. Such component is part of most of the MCU.

No need to configure dedicated GPIO as analog input. It is default setting for the specific pin A0,A1,A2…

Read a value:

x = analogRead(pin);

The value returned in not in V or mV. Conversion is needed.

Set Resolution:

analogReadResolution(bits)

Set Reference

analogReference(ref)

Author – Paul Pinault / Disk91.com

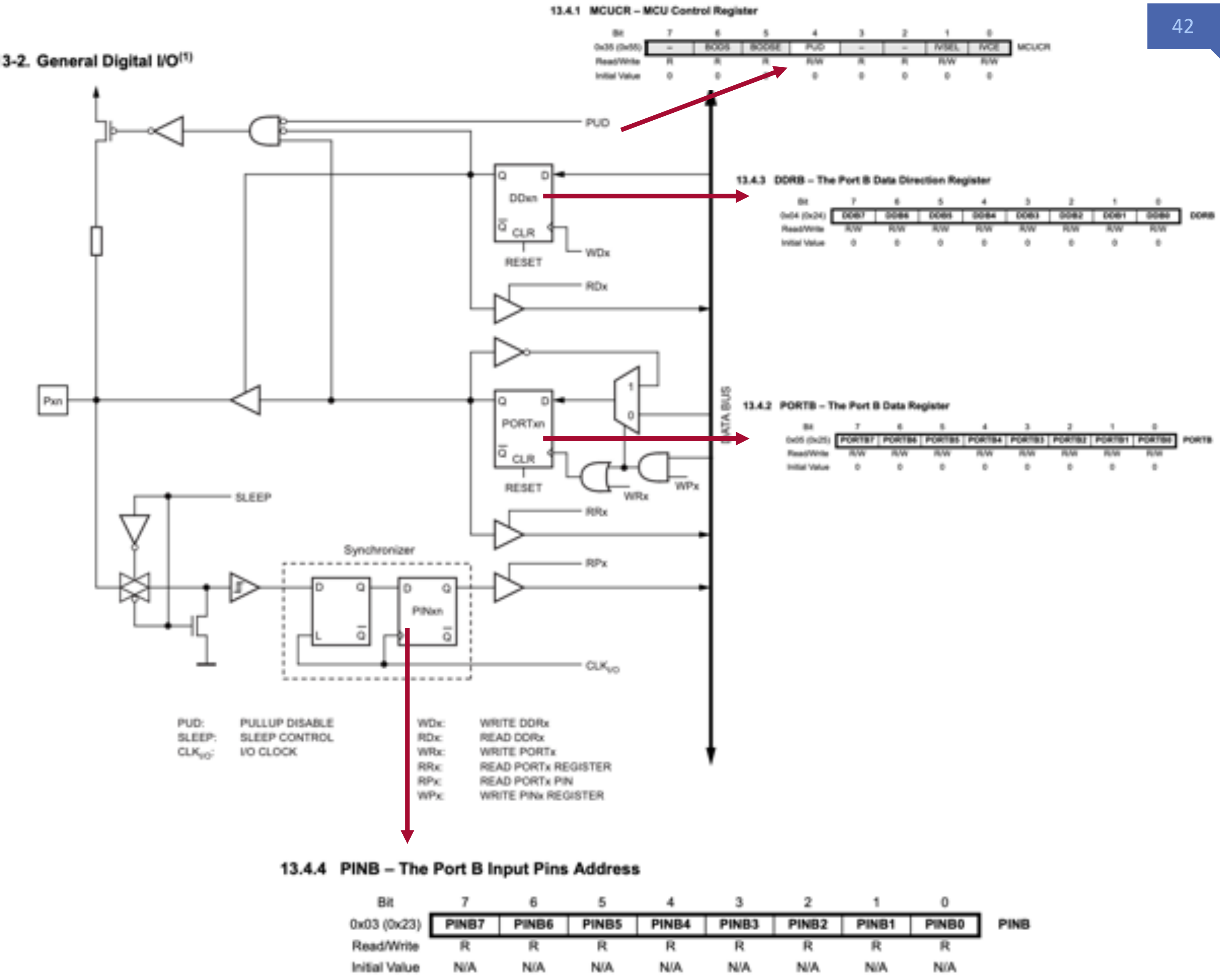
GPIO ARCHITECTURE

A GPIO is controlled by different signals to configure its expected behavior.

PUD,RRx,RPx,WPx,WRx,SLEEP,WDx,RDx…

These signals come from some special registers you can control from the software. This is basically what functions like *pinMode()* do.

Figure 13-2. General Digital I/O[1]



13.4.1 MCUCR – MCU Control Register

Bit	7	6	5	4	3	2	1	0	
0x35 (0x55)	–	BODS	BODSE	PUD	–	–	IVSEL	IVCE	MCUCR
Read/Write	R	R	R	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

13.4.3 DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

13.4.2 PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

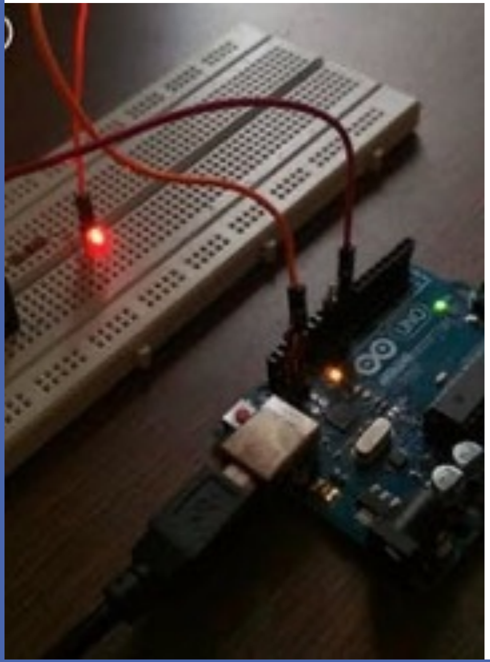
PUD: PULLUP DISABLE
SLEEP: SLEEP CONTROL
CLKI/O: I/O CLOCK

WDx: WRITE DDRx
RDx: READ DDRx
WRx: WRITE PORTx
RRx: READ PORTx REGISTER
RPx: READ PORTx PIN
WPx: WRITE PINx REGISTER

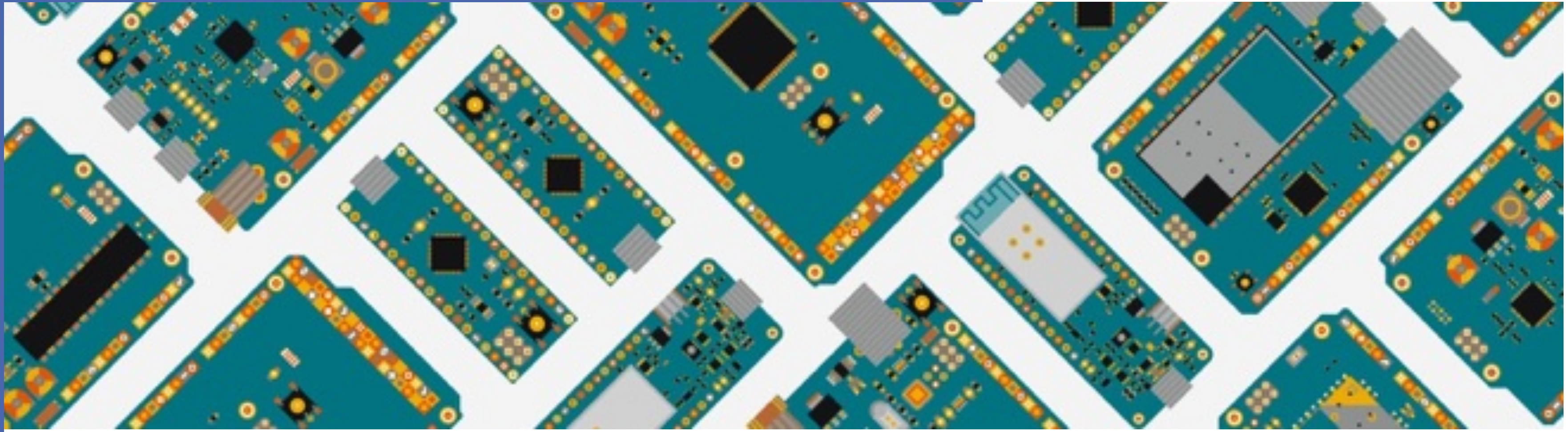
13.4.4 PINB – The Port B Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

How to manage asynchronous events ?



```
This example code is in the public domain.

http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin LED_BUILTIN as an outpu
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED
  delay(1000);                       // wait for a s
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED
  delay(1000);                       // wait for a s
}
```



Author – Paul Pinault / Disk91.com

HOW TO TAKE A SUCH PICTURE ?

You can’t predict asynchronous events; you can’t react, in the right time, on short events.

Continuously scanning while waiting for an event is very energy and CPU consuming… wasting.

A mechanism called Interrupt is solving this issue.

Interrupts explained with a hot chocolate & Nutella toast

Sequential approach

Polling approach

Chocolate could be too warm…

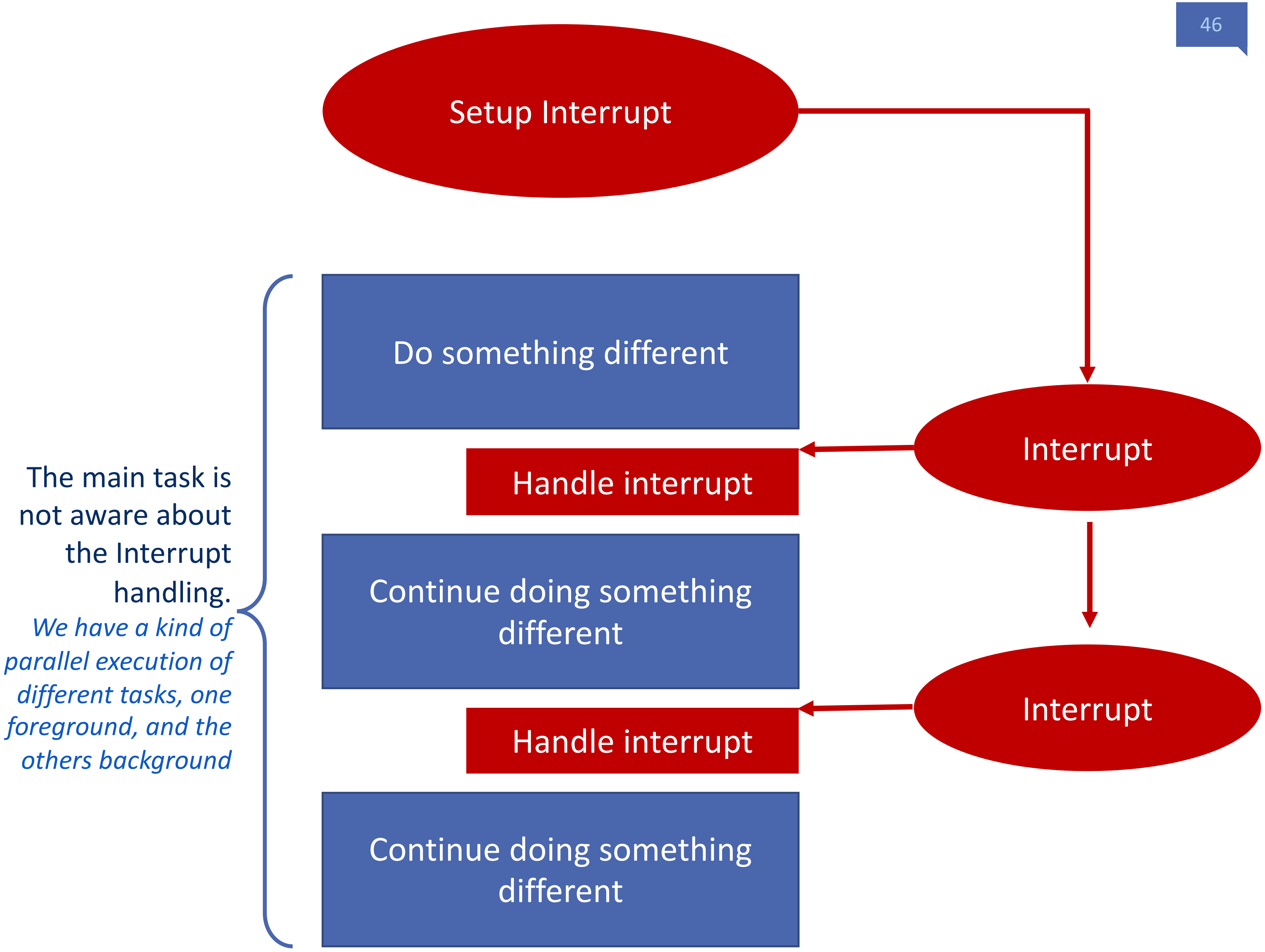
Interrupt approach

You get, at least ,more toast …

Interrupt handlers

An interrupt handler is a short piece of code to be executed on every interrupt to process it.

The main task does not know it has been stopped during the interrupt handler execution.



Setup Interrupt

Do something different

Handle interrupt

Continue doing something different

Handle interrupt

Continue doing something different

Interrupt

Interrupt

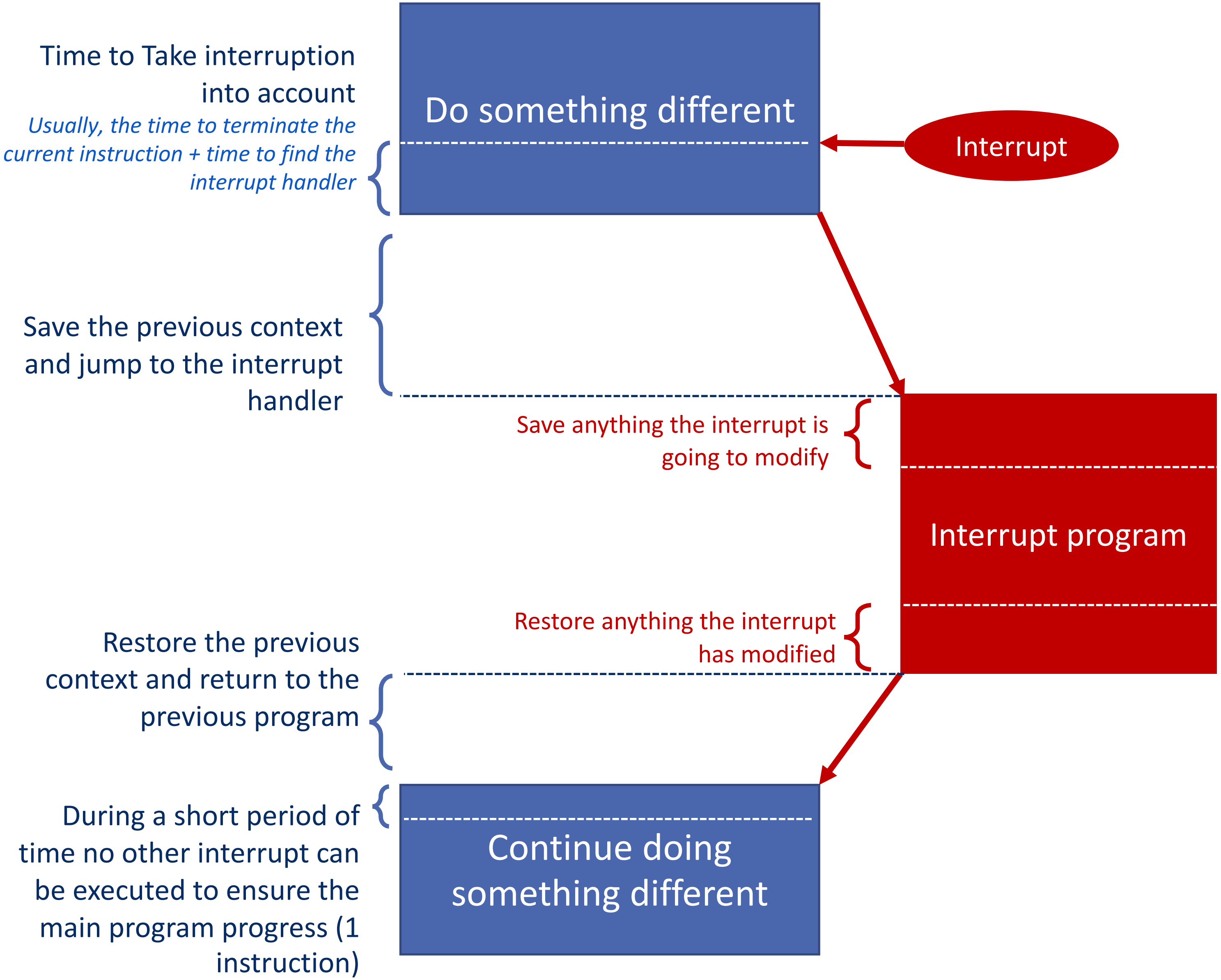
The main task is not aware about the Interrupt handling.

We have a kind of parallel execution of different tasks, one foreground, and the others background

Author – Paul Pinault / Disk91.com

Zoom on Interrupt handlers

To transparently switching from a program to an interrupt handler, at any time, the interrupt mechanism needs to save the previous context and restore it at end.



Time to Take interruption into account

Usually, the time to terminate the current instruction + time to find the interrupt handler

Do something different

Interrupt

Save the previous context and jump to the interrupt handler

Save anything the interrupt is going to modify

Interrupt program

Restore anything the interrupt has modified

Restore the previous context and return to the previous program

During a short period of time no other interrupt can be executed to ensure the main program progress (1 instruction)

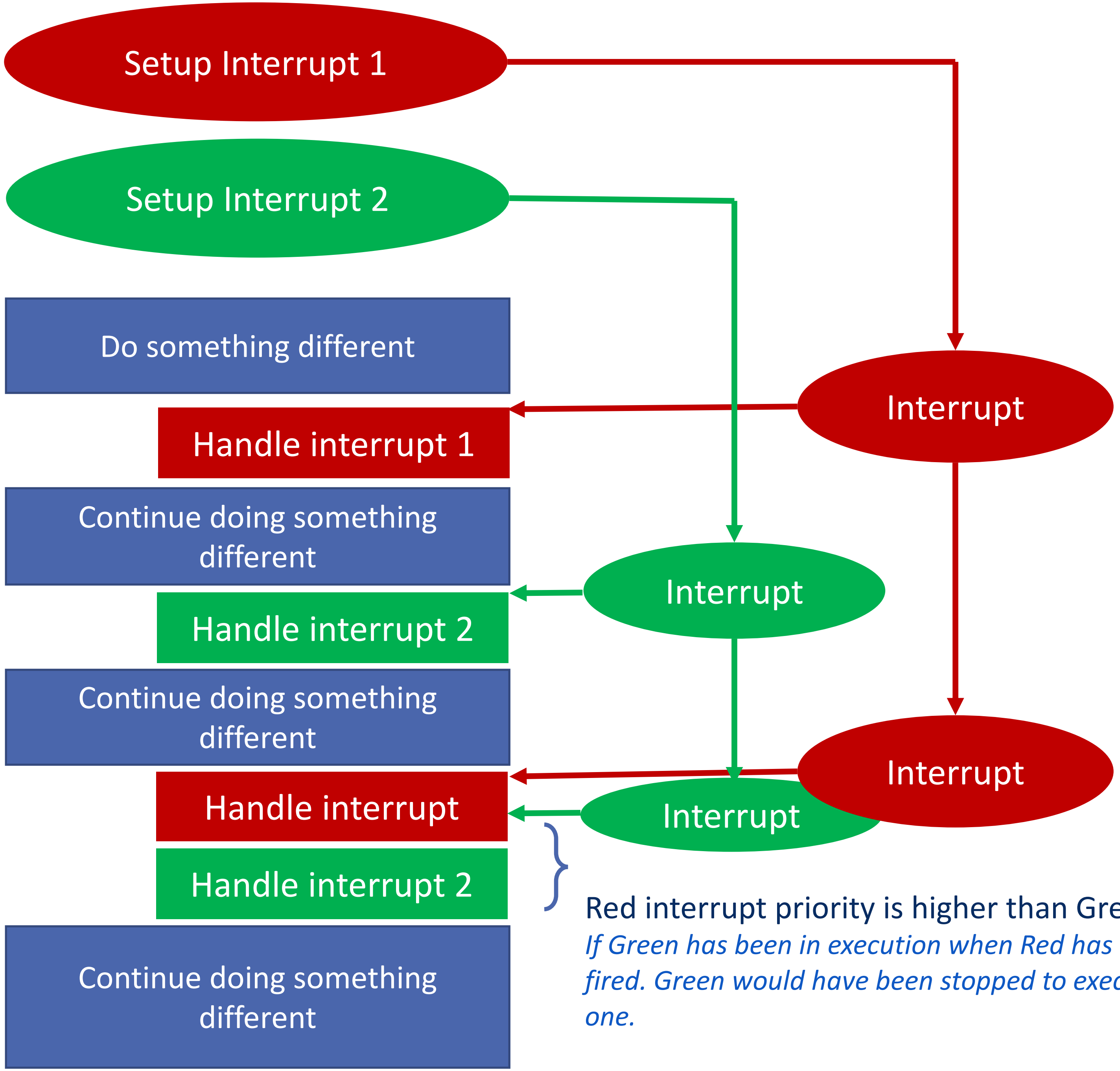
Continue doing something different

Interrupt priority

There are different types of interrupts.

All of them can be activated in parallel and conduct the execution of a handler when fired.

There are priorities to ensure the most important are not interrupted.



Setup Interrupt 1

Setup Interrupt 2

Do something different

Handle interrupt 1

Continue doing something different

Handle interrupt 2

Continue doing something different

Handle interrupt

Handle interrupt 2

Continue doing something different

Red interrupt priority is higher than Green one

If Green has been in execution when Red has been fired. Green would have been stopped to execute Red one.

Author – Paul Pinault / Disk91.com

Interrupt Sources

There are different types of interrupts.

They are corresponding to different peripherals able to process background operations and are raised when terminated.

Different classical interruptions a MCU is handling

- A pin of the MCU has its state changed

- A character has been received on the serial port
- A character has been sent on the serial port

- A given duration has been expired
- A given number of events has been seen
- A watchdog event occurred
- An ADC conversion is terminated
- Analog comparison event
- …

Interrupt Registers are allowing to manage interrupts

- Interrupt Mask Register is masking (disabling) interrupts one by one.
- Global Interrupt flag is masking all interrupts in a single change. This is needed for critical sections.
- Interrupt Flag register is indicating the status of each of the interrupt.

Interrupt Vector

Illustration of a Table of handler address

IRQ	IRQ VECTOR MEMORY ADDRESS	Address of the IRQ Handler
RESET	0x0000	0x2000
INT0	0x0002	0x2450
INT1	0x0004	0x2600
TIMER1	0x0006	0x2120
USARTRX	0x0008	0x2071

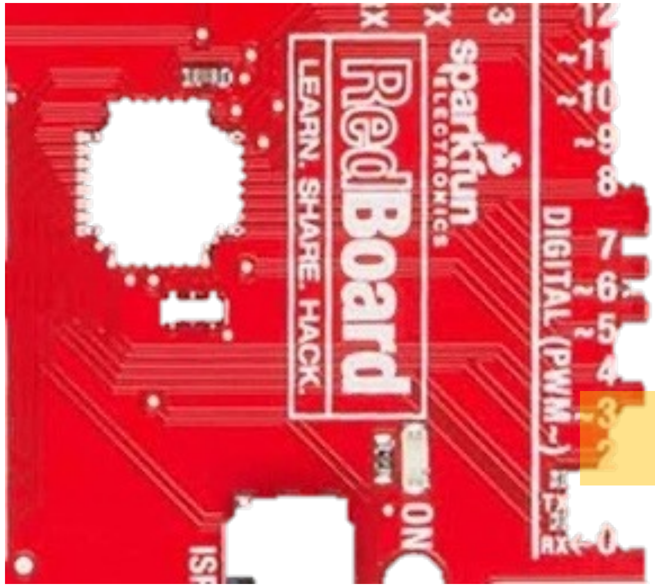
Illustration of a Table of IRQ Handler

IRQ	IRQ VECTOR MEMORY ADDRESS	Address of the IRQ Handler
RESET	0x0000	JMP 0x2000
INT0	0x0008	RET
INT1	0x0010	INC @0x150 RET
TIMER1	0x0018	JMP 0x2120
USARTRX	0x0020	RET

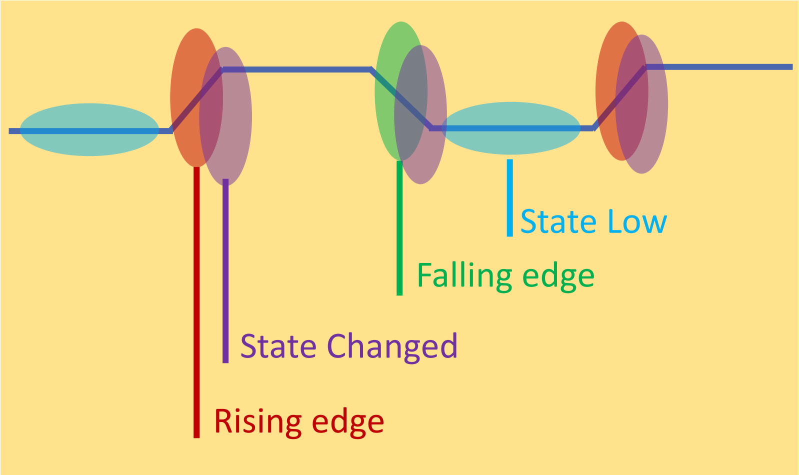
ATMEGA328P, is using a Table of IRQ Handler with 2 byte per handler so usually you have a JMP to a real IRQ handler or a RET.

INTERRUPTS IN PRACTICE

ARDUINO GPIO INTERRUPTION



Different events can cause an interrupt.
Interrupt setup will define what type of event will fire the interrupt.



Edge events are 1 shot interrupt
State low events will be raised until the state becomes high, interrupt handler can be called multiple times.

Arduino has 2 different GPIOs Interruptions:

- 2 external line interrupts connected to PIN 2 & 3 named INT0 & INT1.
- 1 GPIO changed interrupt concerning any of the pins.

Declare External line interrupt with Arduino

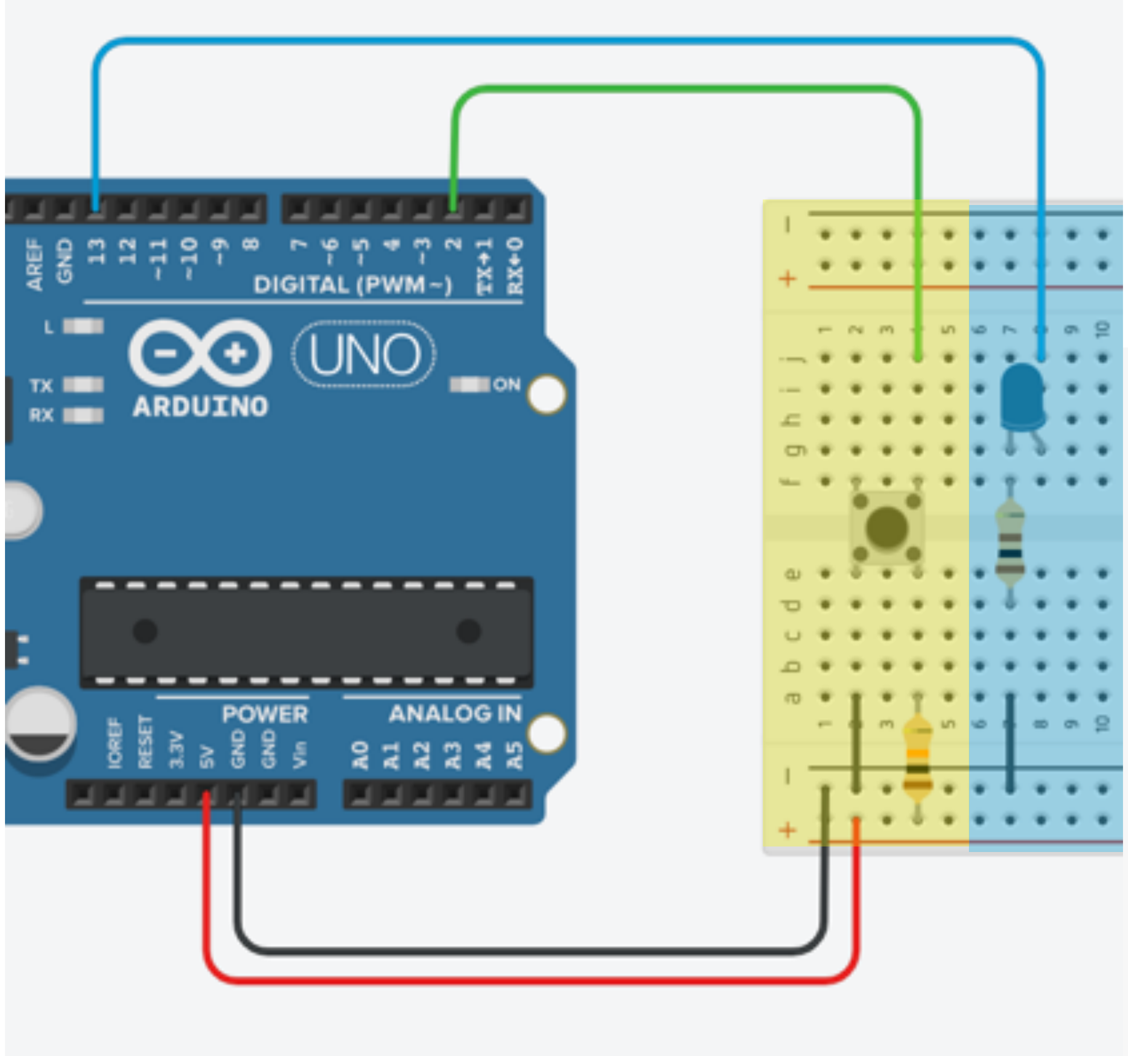
attachInterrupt(n, handler, event)
Setup a handler function to an external line interrupt for a given event LOW, CHANGE, RISING, FALLING

or
ISR(EXT_INT_0/1_vect) {…}
Interrupt handler function, what to do when the interrupt is fired. No parameter, no return value.

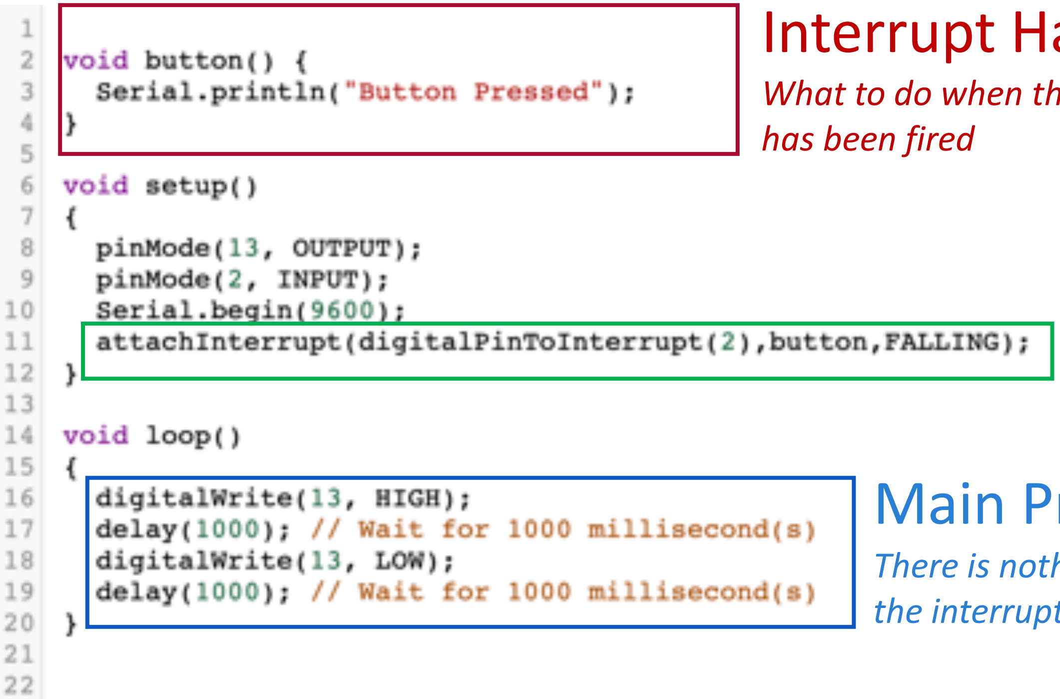
Author – Paul Pinault / Disk91.com

External interrupt

Application



The EXT interrupt can be used to detect a switch status or change immediately and take an action in parallel of other foreground actions, like here, a led blinking.



```

void button() {
  Serial.println("Button Pressed");
}

void setup()
{
  pinMode(13, OUTPUT);
  pinMode(2, INPUT);
  Serial.begin(9600);
  attachInterrupt(digitalPinToInterrupt(2),button,FALLING);
}

void loop()
{
  digitalWrite(13, HIGH);
  delay(1000); // Wait for 1000 millisecond(s)
  digitalWrite(13, LOW);
  delay(1000); // Wait for 1000 millisecond(s)
}


```

Interrupt Handler

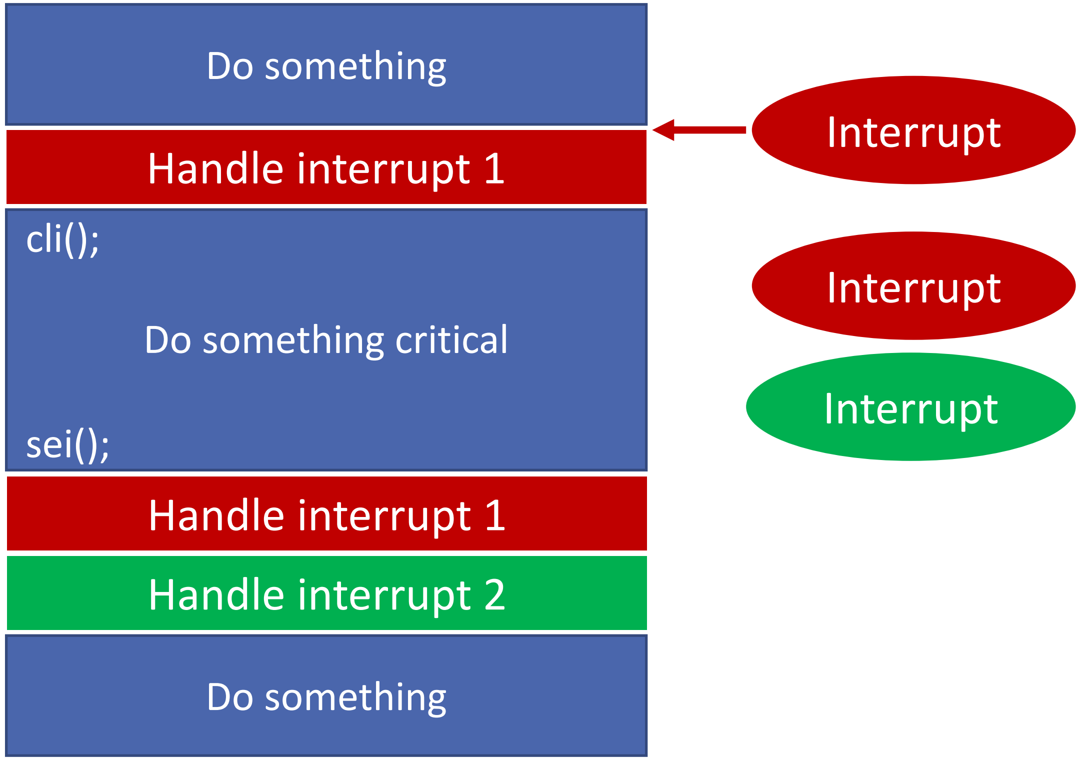
What to do when the interrupt has been fired

Interrupt Setup

Main Program

There is nothing related to the interrupt

Critical sections



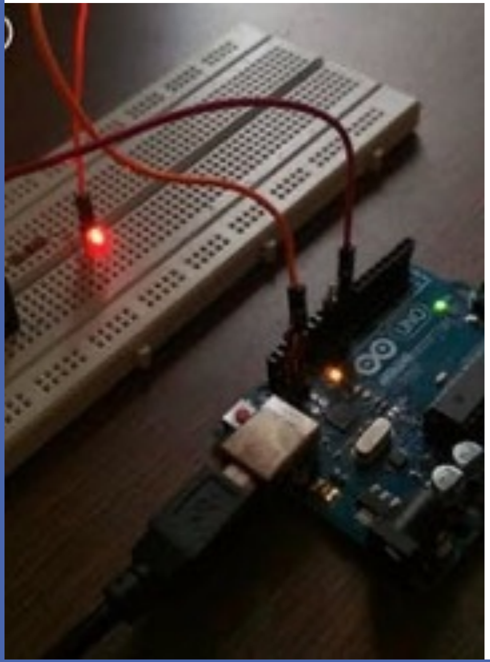
A critical section is a piece of code you do not want to be interrupted. There are multiple reasons:

- **Modify a variable shared between different process or with an interrupt handler.**
- **Critical code like emergency stop**
- **Time critical communication with**

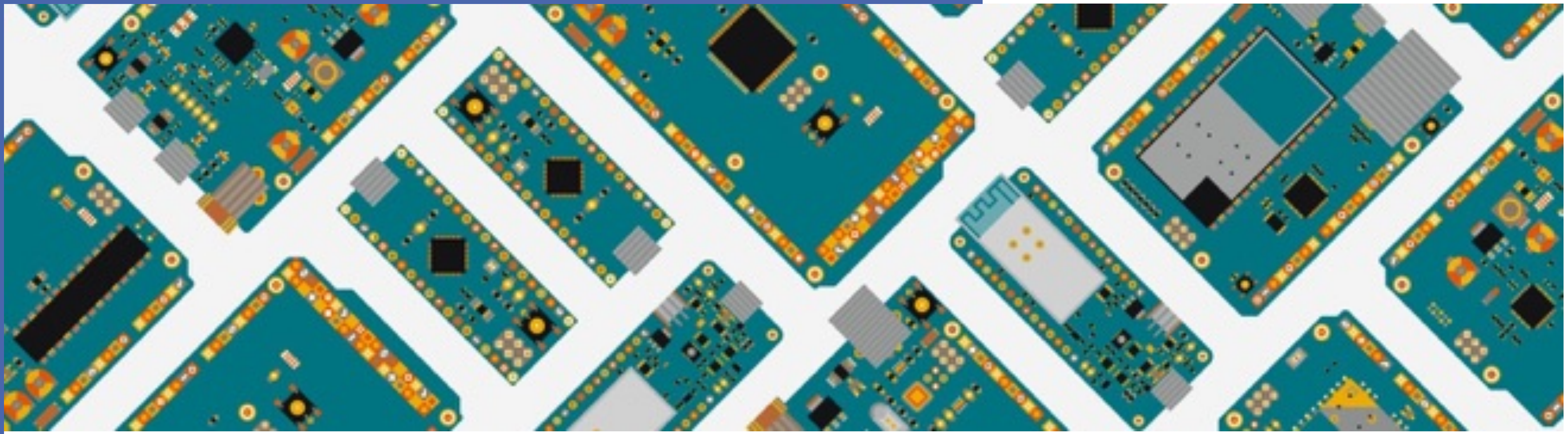
cli()
disable interrupts

sei()
Enable interrupts

How to correctly manage time in a computer system ?



```
  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin LED_BUILTIN as an output
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {$
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED
  delay(1000);                       // wait for a s
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED
  delay(1000);                       // wait for a s
}
```



Author – Paul Pinault / Disk91.com

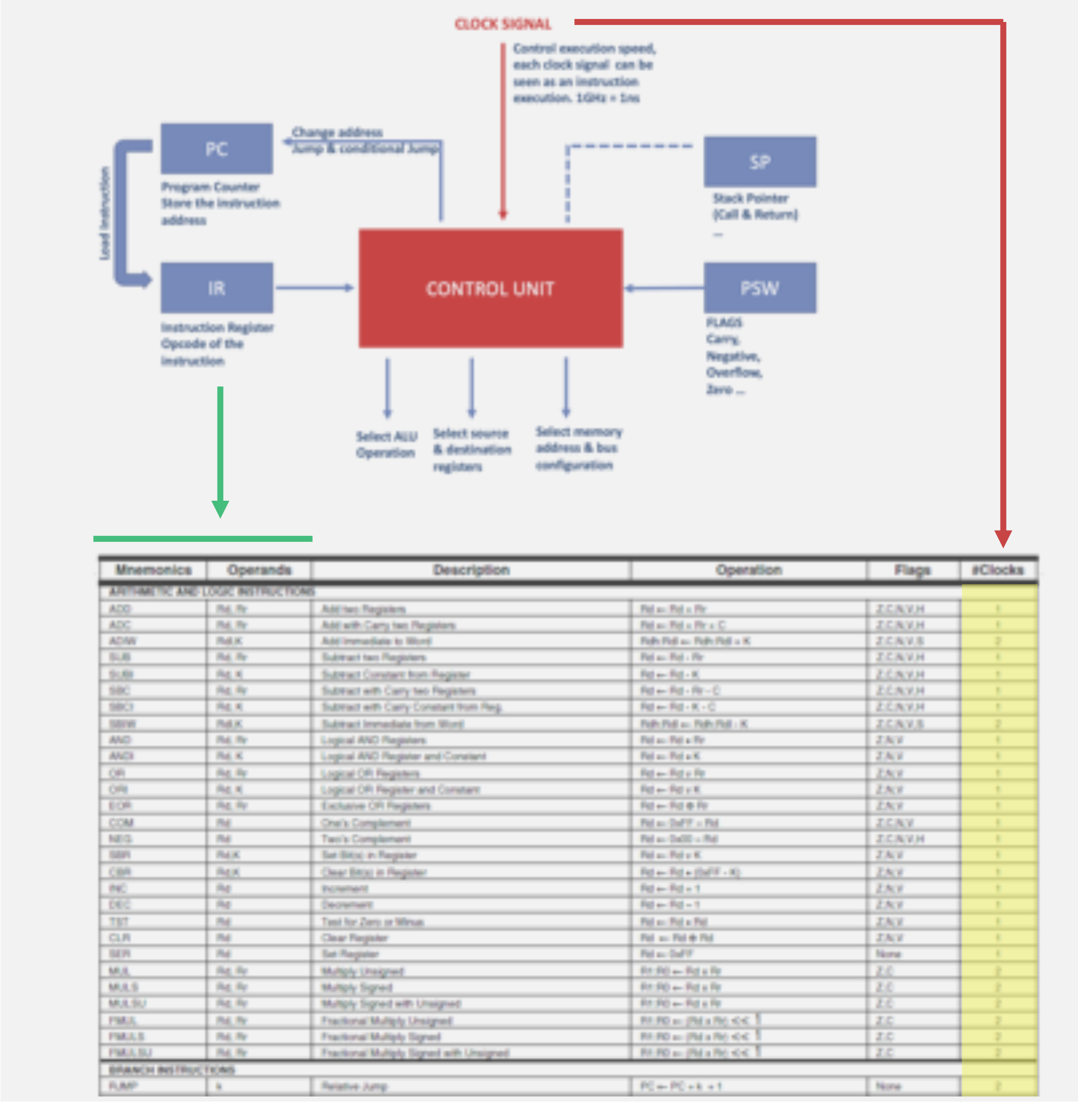
ARDUINO TIME SOURCE

The source of time can be an internal oscillator or and external Crystal source with different precision level.

- Internal Oscillator precision about 10kppm – 1%
- External Crystal Oscillator 10-50ppm

These clock source precisions are limited, but good enough for most of the applications

Manage time with instructions



```
for ( int i = 0 ; i < 250 ; i++);
```

```
                              cycle
          LDI R18,250          1
Loop:
          DEC R18              1
          NOP                  1
          BRNE Loop            2
```

Total loop cycles = 250x4 cycles

Each cycle is 1/16MHz = 62.5ns

Total loop time = 62.500 uS

C

No duration details

ASM

Duration can be precisely determined.

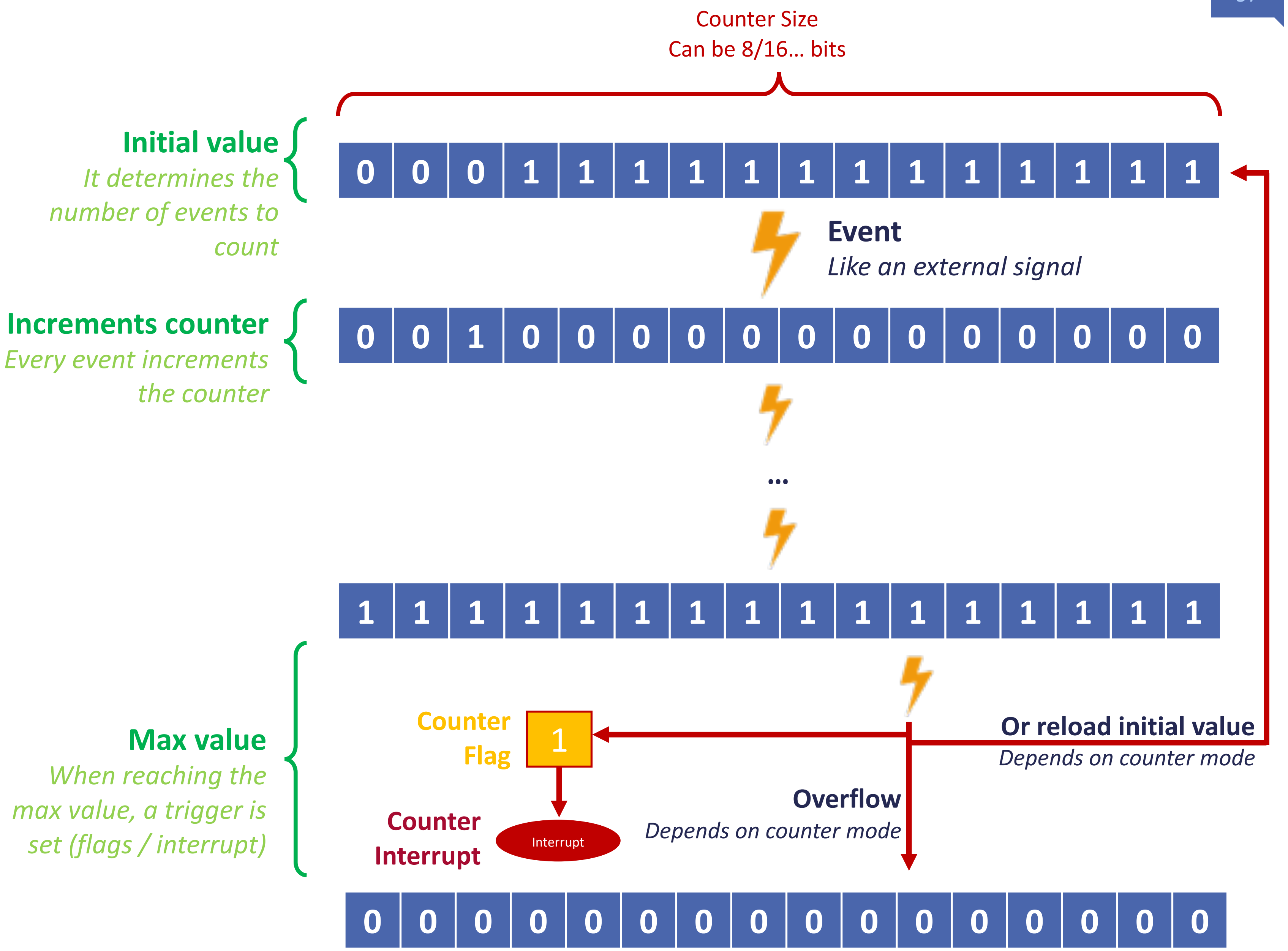
Waste CPU time spent to wait.

Real duration is determined by clock precision and potential interruption

WHAT IS A COUNTER ?

A counter is a peripheral, available in any CPU/MCU, able to count events asynchronously.

It counts events without taking CPU time (in background) and wake up the CPU when it has been programmed for.



Counter Size
Can be 8/16… bits

Initial value
It determines the number of events to count

0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1

Event
Like an external signal

Increments counter
Every event increments the counter

0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

…

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Max value
When reaching the max value, a trigger is set (flags / interrupt)

Counter Flag
1

Counter Interrupt
Interrupt

Or reload initial value
Depends on counter mode

Overflow
Depends on counter mode

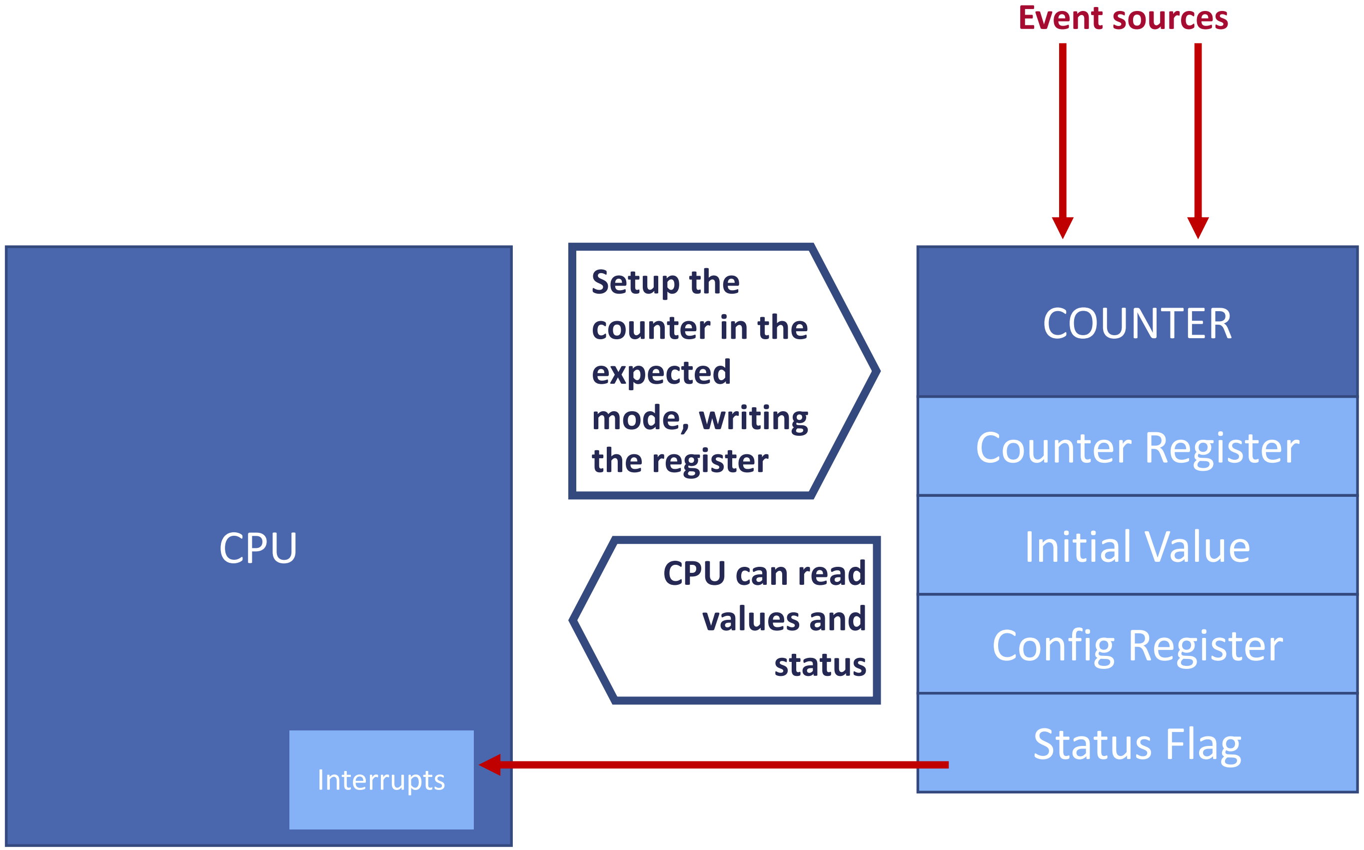
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

WHAT IS A COUNTER ?

The CPU is configuring the COUNTER writing configuration in counter’s register.

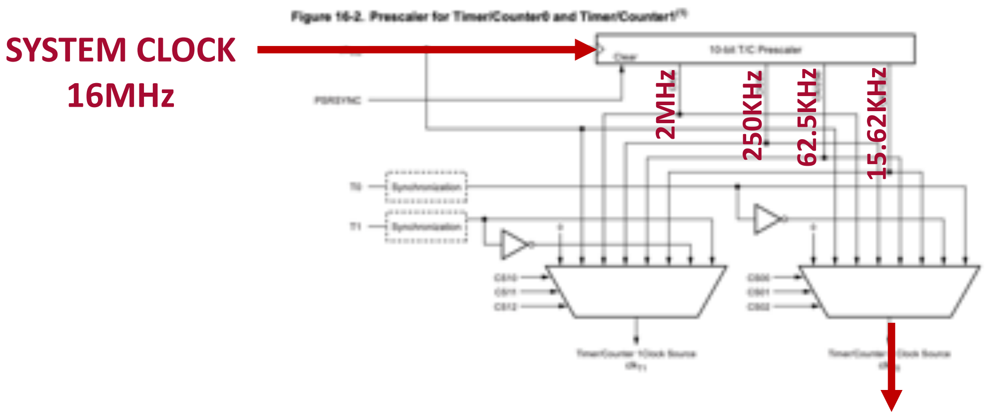
Then the CPU will do something different, until it gets triggered by the counter’s interrupt.

At any time, the CPU can read registers and counters so monitor the progress, for non interrupted usage, or basic counting purpose.



COUNTERS

TIMERS ARE COUNTERS CONNECTED ON CLOCK EVENT



CLOCK DIVIDER

COUNTER

Counter Register

Initial Value

Config Register

Status Flag

If you need to measure a longueur period, you need to use the CPU to count the counter's overflows

8bits mode
0.5us – 16ms

16bits mode
0.5us – 4.19s

A Timer is a counter. What's make a time is the source of the events.

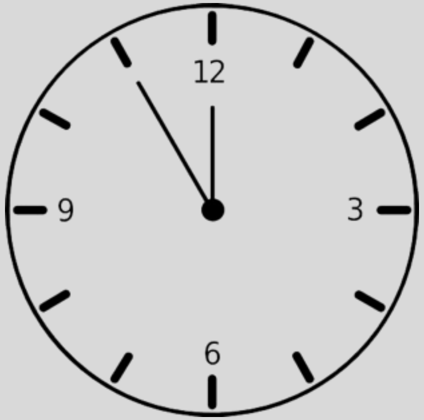
Timer are connected to a clock and therefore, will be incremented on regular basis.

Timers are counting time in background and are not impacted by workload, interruptions …

By the fact they are counters, they are setup for a given time and will generate an interrupt once that time has been reached.

You can also use them to measure time between events.

Different usages



Wait for a certain time

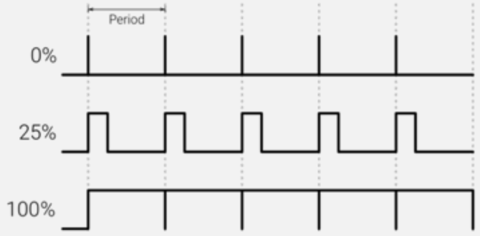
The delay() function is using a timer to get a precise pause duration. Running in background it is also a way to count time elapsed in a system, maintain a clock…

Measure time

You can start a timer at the beginning of an event and stop it at the end of it to measure the event duration precisely. The function millis() and micros() can be used for a such purpose.

Being waked-up on regular basis.

The timer can be program to generate an interrupt on regular basis. This is the starting point of any multi-tasked operating system.



Generate waveform

Timer can automatically and regularly toggle a MCU pin. This can be used to generate a PWM signal and simulate analog output as previously seen. Function analogWrite(…) is using it.

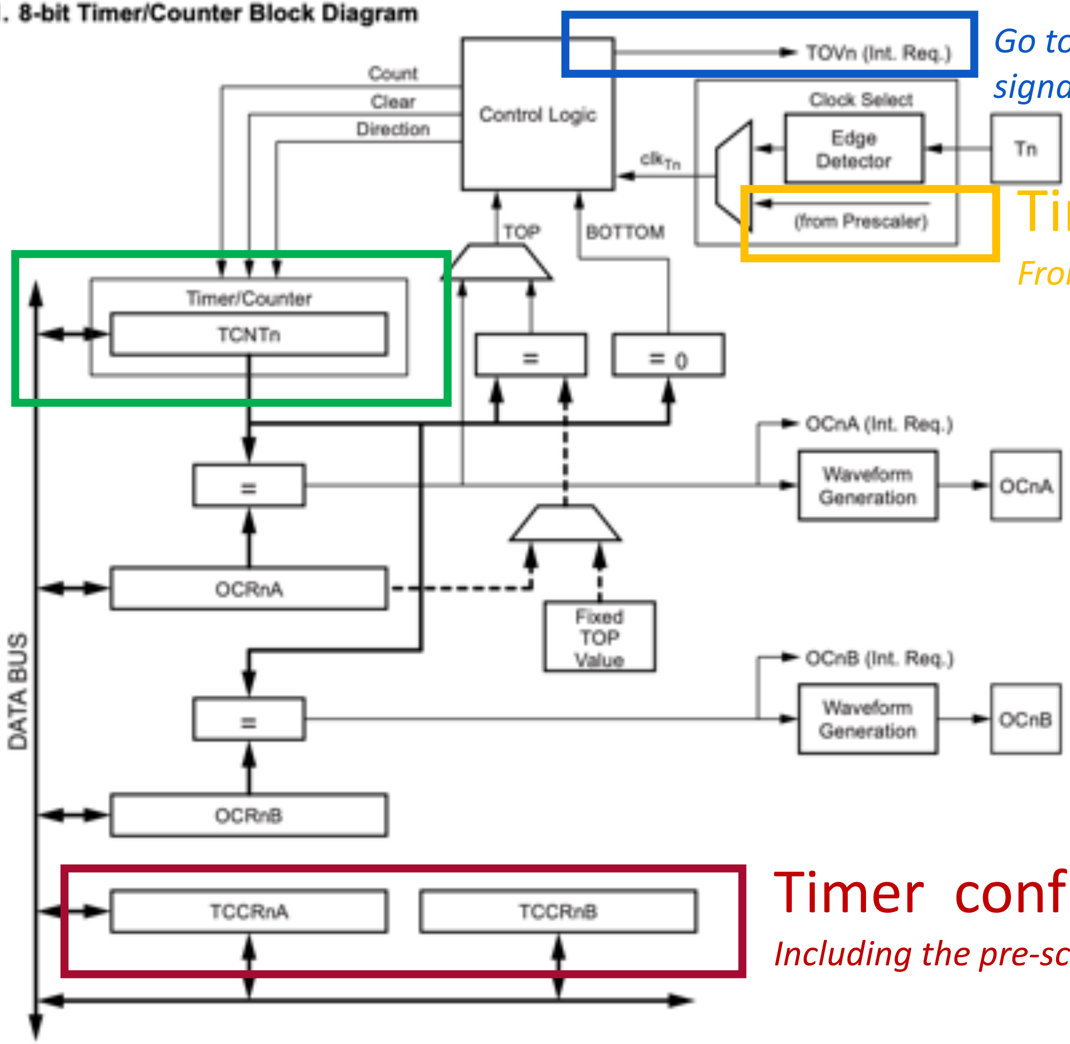
AT328P TIMERS

AT328p have

- 2x 8 bits Timers
    - Timer0 for delay(), millis()
    - Timer2 used by tone()
- 1x 16 bits Timer
    - Timer1 used by servo()

- Timer 2 can be used for our own purpose if no need of tone() – sound generation

Figure 14-1. 8-bit Timer/Counter Block Diagram



Overflow

Go to TIFR Interrupt signal

Time source

From prescaler

Timer counter

Current value of the counter

Timer configuration

Including the pre-scaler setting

Author – Paul Pinault / Disk91.com

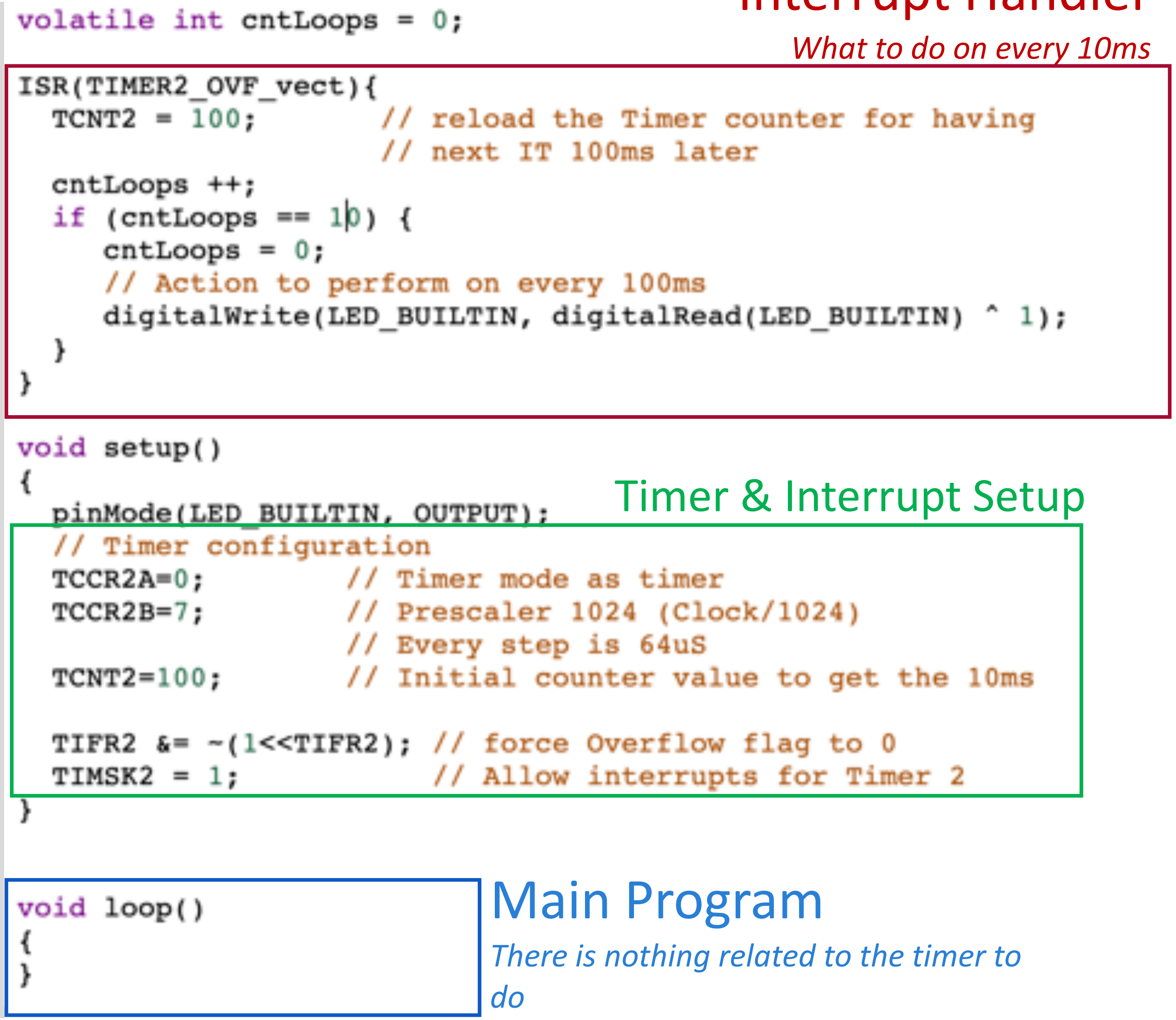
100ms periodic background action

Let's consider CLK / 1024 timer clock source:

- 1 step is 64uS
- 100 ms = 1562.5 steps – higher than 255 – it is impossible to count 100ms with this timer.
- 10 ms = 156.25 steps – we can count 10 expirations of 10ms to get 100ms.

Let's make it simple assuming 10ms = 156 steps
The counter overflow at 255 so we need to initialize the counter with value:

Cnt init = max Value – desired steps
Cnt init = 256 – 156 = 100



Interrupt Handler

What to do on every 10ms

```
volatile int cntLoops = 0;

ISR(TIMER2_OVF_vect){
  TCNT2 = 100;          // reload the Timer counter for having
                        // next IT 100ms later
  cntLoops ++;
  if (cntLoops == 10) {
     cntLoops = 0;
     // Action to perform on every 100ms
     digitalWrite(LED_BUILTIN, digitalRead(LED_BUILTIN) ^ 1);
  }
}
```

Timer & Interrupt Setup

```
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  // Timer configuration
  TCCR2A=0;           // Timer mode as timer
  TCCR2B=7;           // Prescaler 1024 (Clock/1024)
                      // Every step is 64uS
  TCNT2=100;          // Initial counter value to get the 10ms

  TIFR2 &= ~(1<<TIFR2); // force Overflow flag to 0
  TIMSK2 = 1;           // Allow interrupts for Timer 2
}
```

```
void loop()
{
}
```

Main Program

There is nothing related to the timer to do

Measure a time duration

Let’s consider CLK / 32 timer clock source:

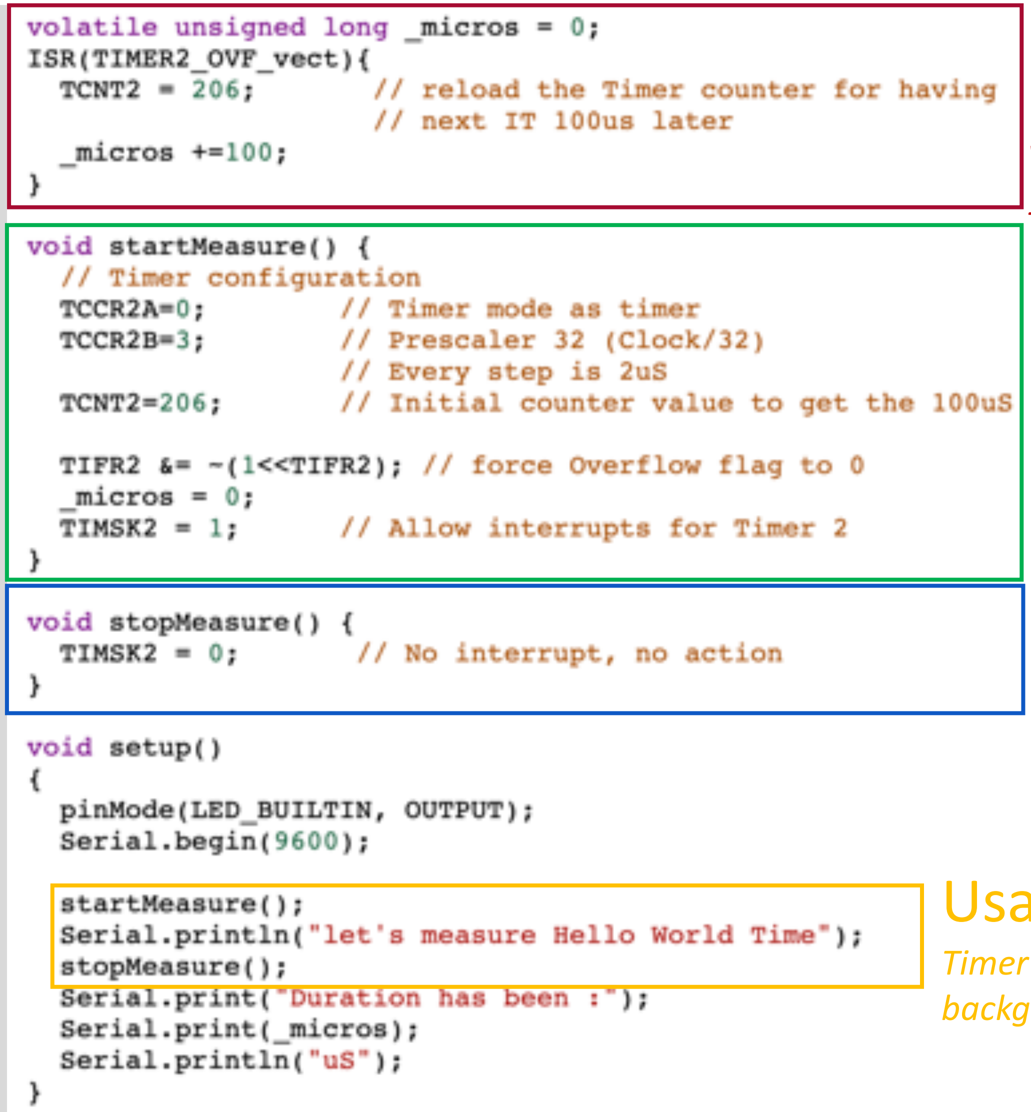
- 1 step is 2us
- every 50 steps we have 100us

We can count the 100us steps to measure a duration of something.

The precision can be adjusted by the clock division. We select the right one depending on what we want to measure.

It is important to not have too much interrupt call to not impact the measure when it concerns internal processing.

To measure external processing (like a round trip delay of an ultrasound sensor) there is no big impact.



```
volatile unsigned long _micros = 0;
ISR(TIMER2_OVF_vect){
  TCNT2 = 206;          // reload the Timer counter for having
                        // next IT 100us later
  _micros +=100;
}

void startMeasure() {
  // Timer configuration
  TCCR2A=0;            // Timer mode as timer
  TCCR2B=3;            // Prescaler 32 (Clock/32)
                       // Every step is 2uS
  TCNT2=206;           // Initial counter value to get the 100uS

  TIFR2 &= ~(1<<TIFR2); // force Overflow flag to 0
  _micros = 0;
  TIMSK2 = 1;          // Allow interrupts for Timer 2
}

void stopMeasure() {
  TIMSK2 = 0;           // No interrupt, no action
}

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);

  startMeasure();
  Serial.println("let's measure Hello World Time");
  stopMeasure();
  Serial.print("Duration has been :");
  Serial.print(_micros);
  Serial.println("uS");
}
```

Interrupt Handler

What to do on every 100uS

Timer & Interrupt Setup

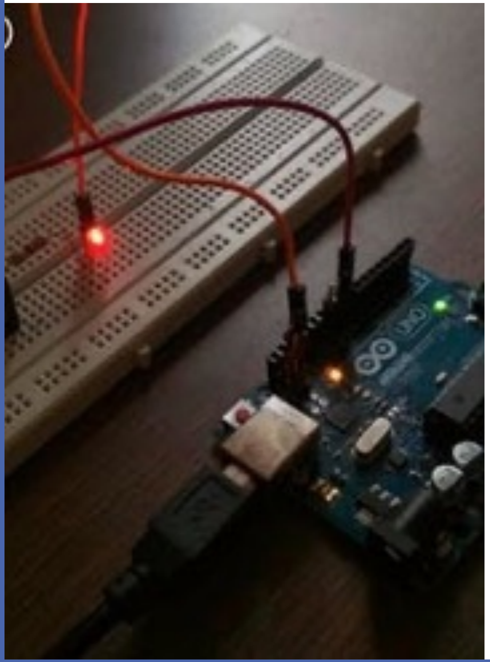
For every time we want to measure a duration

Stop Timer

When we have finished to count time

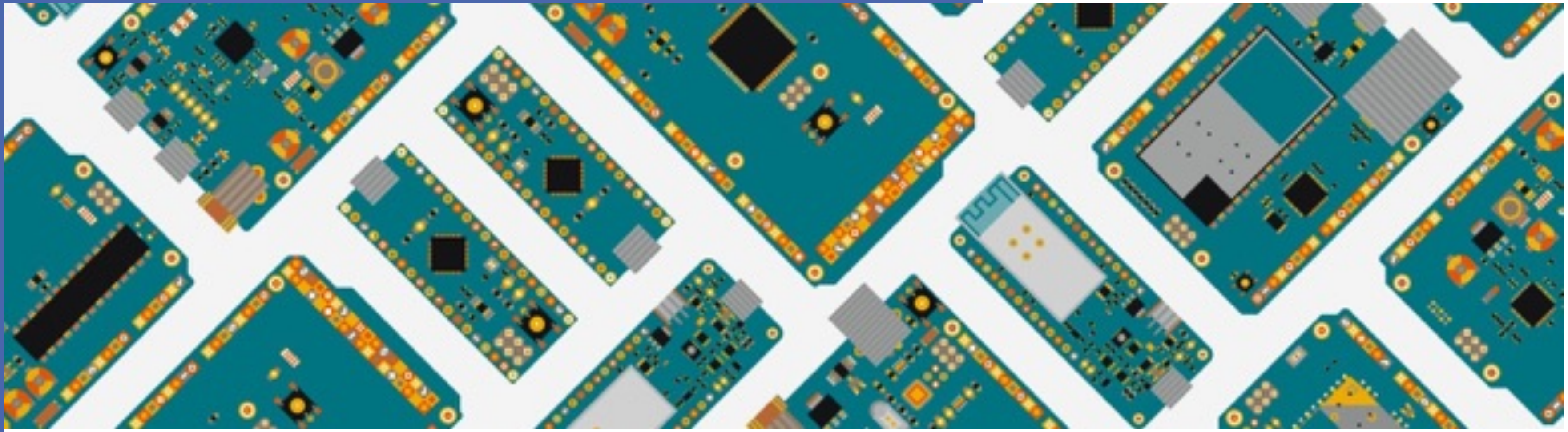
Usage

Timer counts time in background



Communication bus, multiple ways to interact between systems / with sensors

```
This example code is in the public domain.

http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin LED_BUILTIN as an output
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {$
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED
  delay(1000);                       // wait for a s
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED
  delay(1000);                       // wait for a s
}
```



Author – Paul Pinault / Disk91.com

Communication bus

Communication is a critical point and balanced different objectives:

- Simplicity & price
- Speed
- Latency
- Reliability
- Distance
- Universality

Hardware development is optimized for every use-case. Every sensor supports the most efficient communication bus. MCU must support different one.

Different usages

High speed bus, for high end communications

Desktop & server uses complex and high-speed communication buses. They are costly solution in regards of embedded systems like Arduino. We have:

- *SATA, firewire*
- *Ethernet, WiFi*
- *PCIe …*
- *4G/5G*

Universal, old school, communication bus

Most of the systems implements low-speed, old communication systems. They are serial communications:

- *UART / USART*
- *USB USART emulation*
- *IR*

Sensor optimized communication bus

More efficient communication bus have been deployed to communicate with sensors:

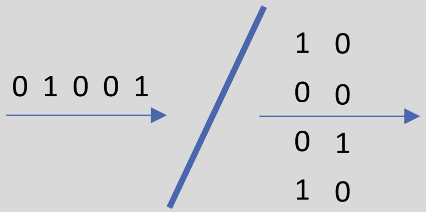
- *SPI*
- *I2C*
- *One Wire*

IoT oriented communications bus

System to system communications can rely on “network” oriented communication bus built for embedded solutions:

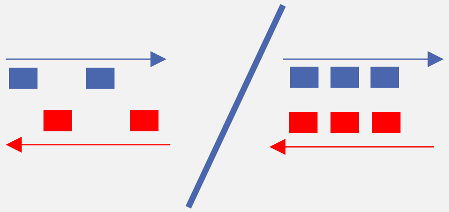
- *Bluetooth*
- *Sigfox / LoRa*
- *Local radio …*

Different parameters



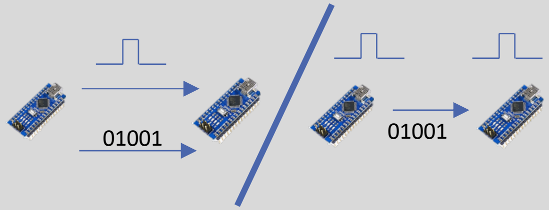
Serial vs Parallel

Data can be transmitted one bit after the other (serial communications) or by word (parallel communications).
Serial allows to have less wires and make design simpler.
Parallel is used for high-speed memory interface.
Serial is used for interfaces like UART, SATA, USB, ETHERNET… most of the communication are serial based now days.



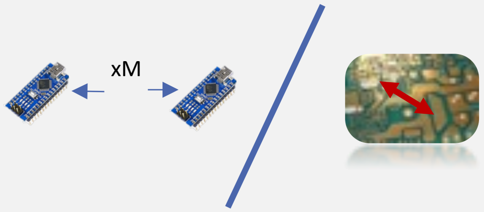
Half vs Full Duplex

Half duplex system is not able to talk and listen at the same time. Full duplex systems can do it.
Full duplex requires 1 wire for Tx and 1 wire for Rx. Even with 2 wires some systems do not have the ability to Rx and Tx in terms of computing power (less and less now days)



Synchronous vs Async

System can have a common clock for transmission, shared over a wire. This synchronization allows high speed communication.
Otherwise, the systems need to negotiate a baudrate and oversample to get in sync. The transmission rate is lower.



Long vs Short range

Communicating at 5 cm is different than communicating over long distance. The electrical consideration are different, the noise tolerance is also totally different.

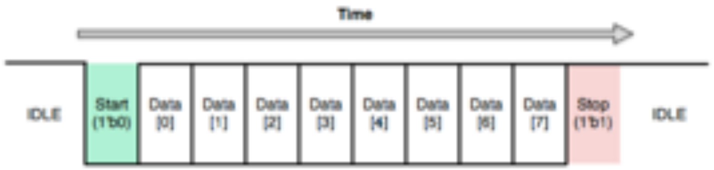
UART

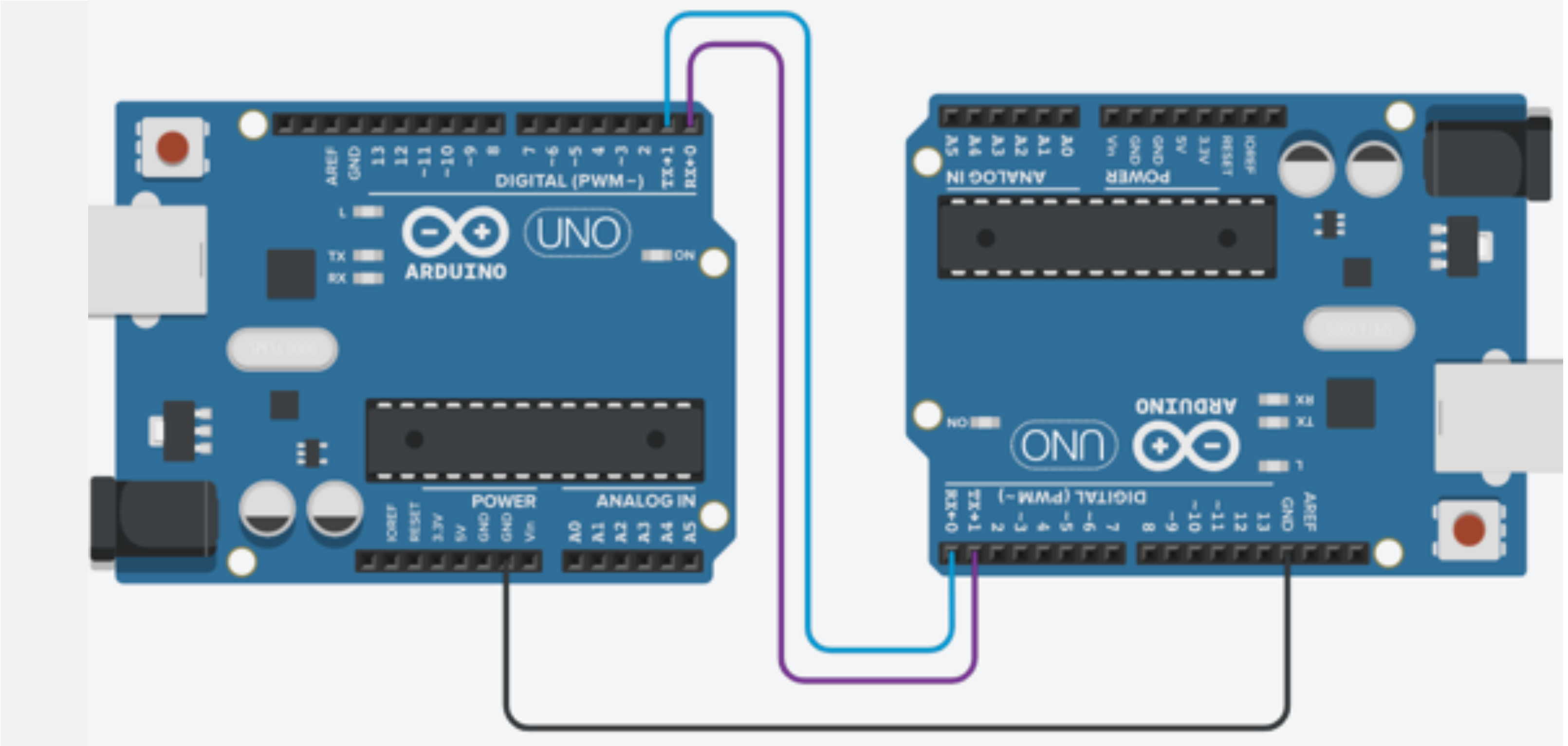
Characteristics

- Serial communication
- Asynchronous
- Usual speed 9600-115200 bps
- Half and Full duplex
- Short Range (TTL)
- Long Range (RS232)

Rx and TX are seen from the MCU point of view.

2 wires RX-TX + GND. You can also have flow control signals RTS-CTS…





```
void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.println("Hello World");
  delay(1000);
}
```

```
void setup()
{
  Serial.begin(9600);
}

void loop()
{
  if ( Serial.available() ) {
    Serial.print(Serial.read());
  }
}
```

Setup

Syncing on baudrate

Send message

On regular basis

Read message (and loop)

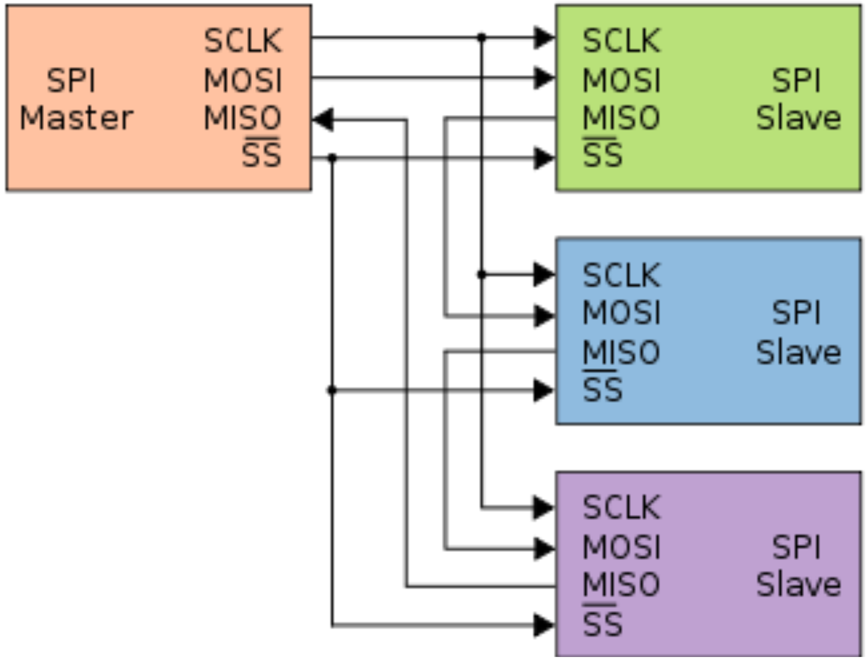
When some

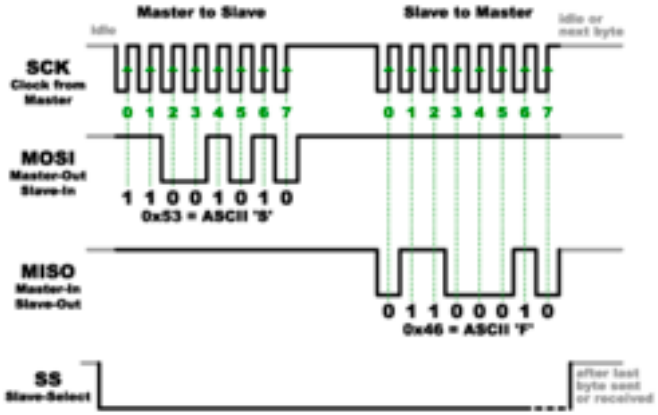
SPI

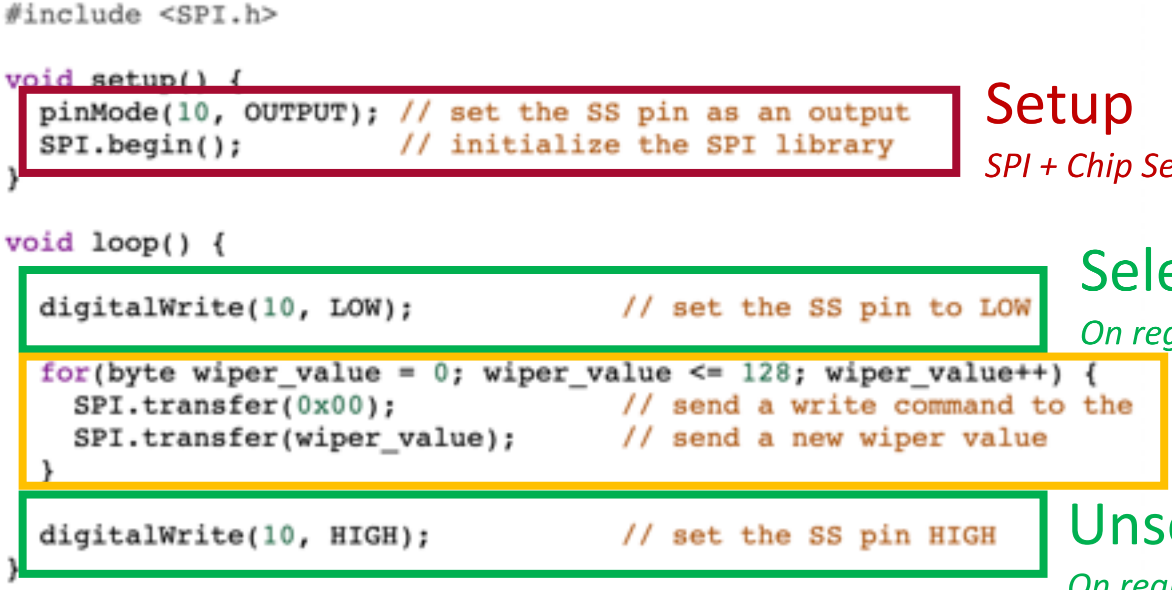
Characteristics
- Serial communication
- Synchronous
- Up to 60Mbits
- Full duplex
- Short Range

4 wires including CLK + GND
Master with multiple slaves.

Applications like SDCARDS, Flash memory, ADC, Camera lens…







```
#include <SPI.h>

void setup() {
pinMode(10, OUTPUT); // set the SS pin as an output
SPI.begin();         // initialize the SPI library
}

void loop() {
digitalWrite(10, LOW);             // set the SS pin to LOW
for(byte wiper_value = 0; wiper_value <= 128; wiper_value++) {
  SPI.transfer(0x00);              // send a write command to the
  SPI.transfer(wiper_value);       // send a new wiper value
}
digitalWrite(10, HIGH);            // set the SS pin HIGH
}
```

Setup

SPI + Chip Select pin

Select Slave

On regular basis

Write to device

COMMAND + VALUE

Unselect Slave

On regular basis

Author – Paul Pinault / Disk91.com

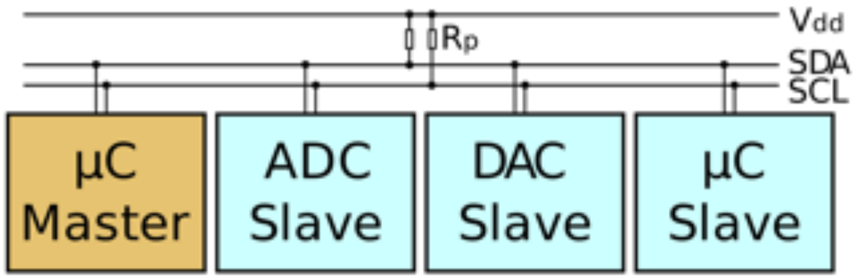
I2C

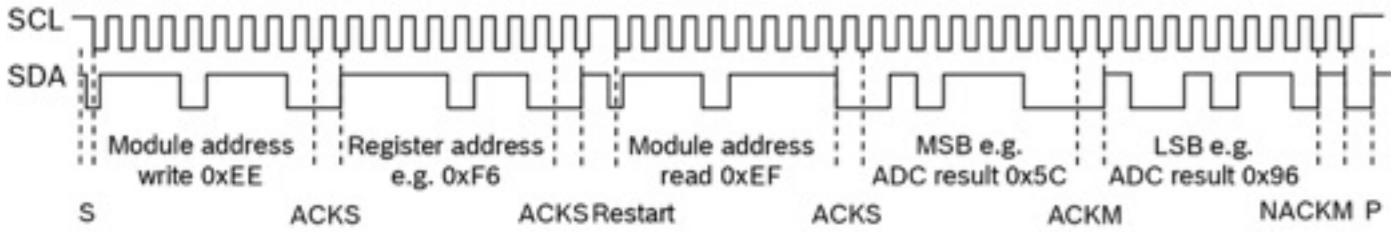
Characteristics

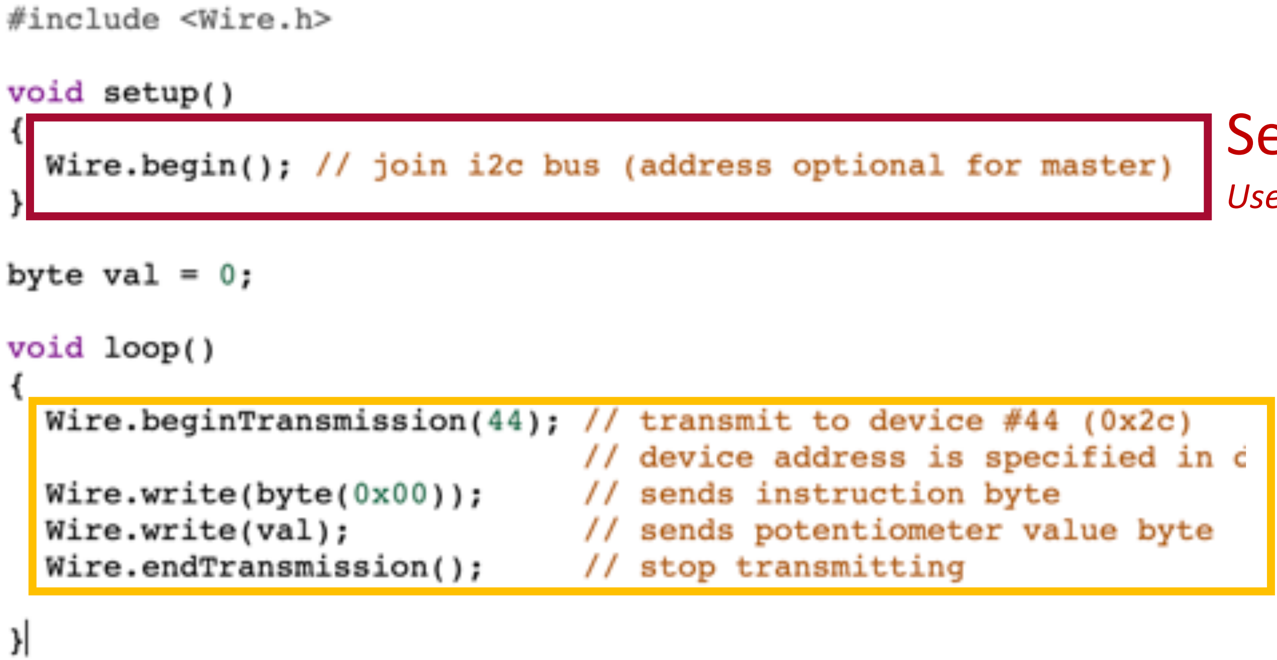
- Serial communication
- Synchronous
- 100Kbps to 5Mbps
- Half duplex
- Short Range

2 wires (SDA/SCL) + GND
Master with multiple slaves.
Each of the slaves have a hard coded 7 bits address for chip selection.

Applications like sensors (temperature, accelerometers, …)







```
#include <Wire.h>

void setup()
{
  Wire.begin(); // join i2c bus (address optional for master)
}

byte val = 0;

void loop()
{
  Wire.beginTransmission(44); // transmit to device #44 (0x2c)
                              // device address is specified in c
  Wire.write(byte(0x00));     // sends instruction byte
  Wire.write(val);            // sends potentiometer value byte
  Wire.endTransmission();     // stop transmitting

}
```

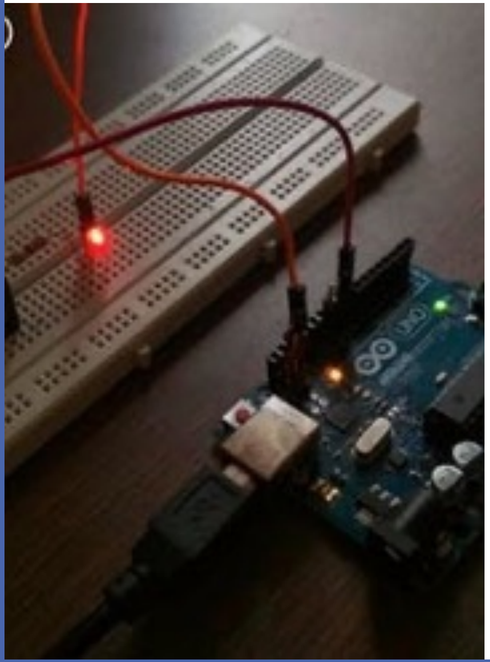
Setup

Uses specific pins

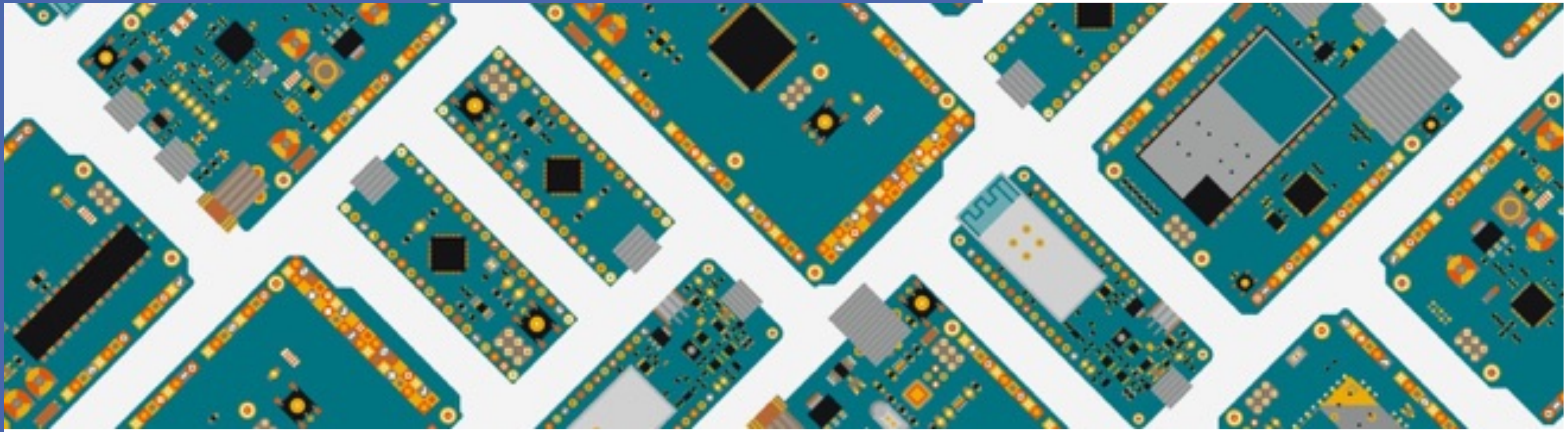
Write to device

Device address + R/W
Register address
Value to write

The STACK, or the way to manage calls and local variables at run.



```
  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press reset
void setup() {
  // initialize digital pin LED_BUILTIN as an output
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {$
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED
  delay(1000);                       // wait for a s
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED
  delay(1000);                       // wait for a s
}
```



Author – Paul Pinault / Disk91.com

SOME PROBLEMS TO SOLVE

Temporarily save data to face the limited number of register

Due to the reduce number of register available for local computation, the compiler needs some extra memory to temporarily save data. It needs a zone of memory for this the stack memory is used.

Save context when executing an interruption

We have seen the need to save the current execution context before jumping to an interrupt processing. For this we also need a memory zone. The stack in involved here also.

Being able to return at the right place at the end of a function execution

Did you even think about how the processor knows where to return at the end of a function call ? How to manage the return history in recursive executions ? Stack Is here too.

Manage local context variable (inside a function) dynamically.

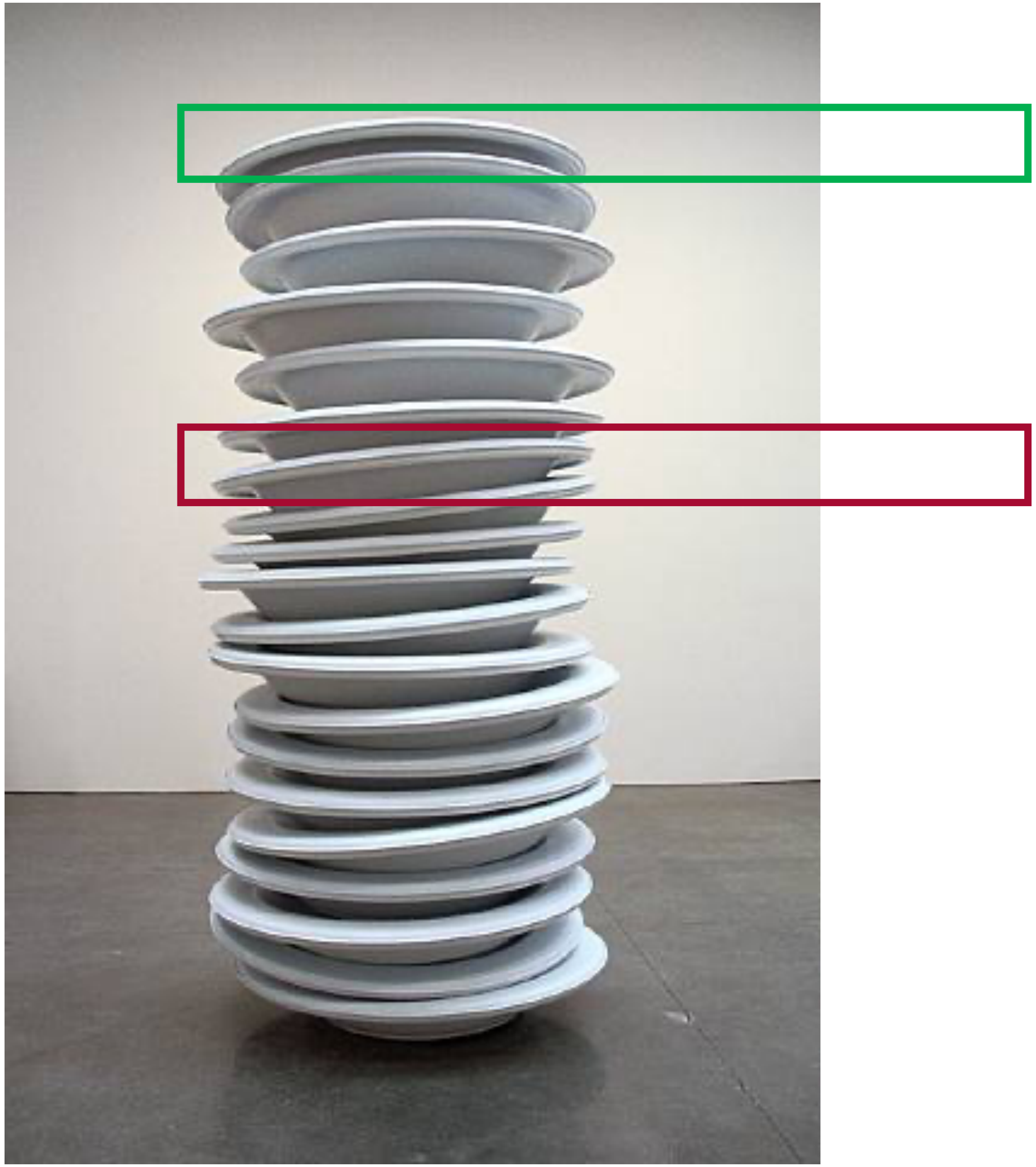
Local variable are allocated on every function call, they can't be placed in the memory in a predefined zone but need to be dynamically allocated. How to manage this in a static allocation context ? Stack !

STACK PRINCIPLE

STACK is a memory zone where the CPU / Compiler can stock temporary data.

Like the plate stack on the right, you add data on top of the memory area, and you remove data from the top of the memory only.

That way, there is no fragmentation in the memory.



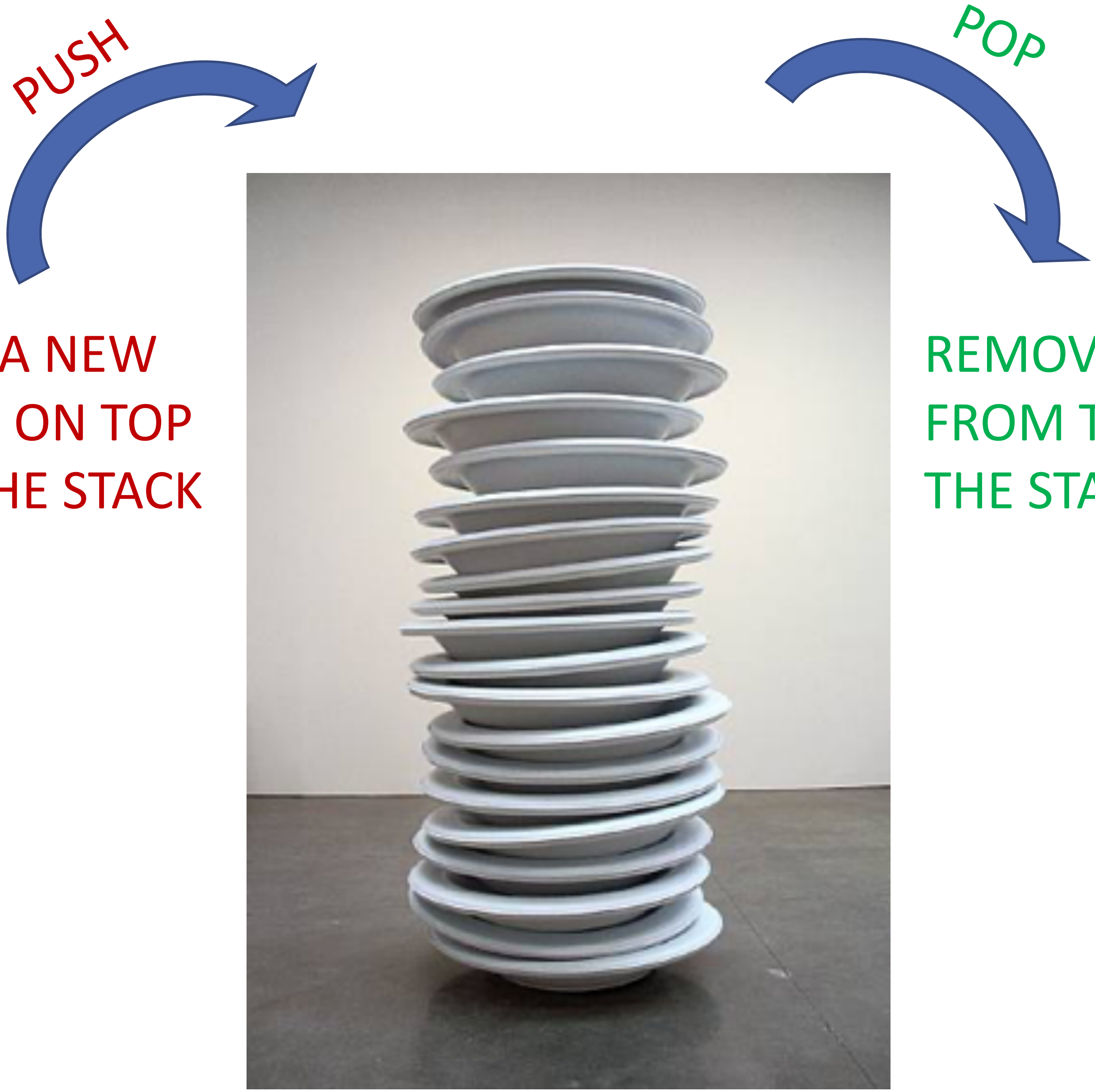
You always add / remove on top of the stack

Accessing the middle of the stack is a problem…

STACK PRINCIPLE

Two simple operations

- PUSH (add a data on top of the stack)
- POP (remove a data from top of the stack)



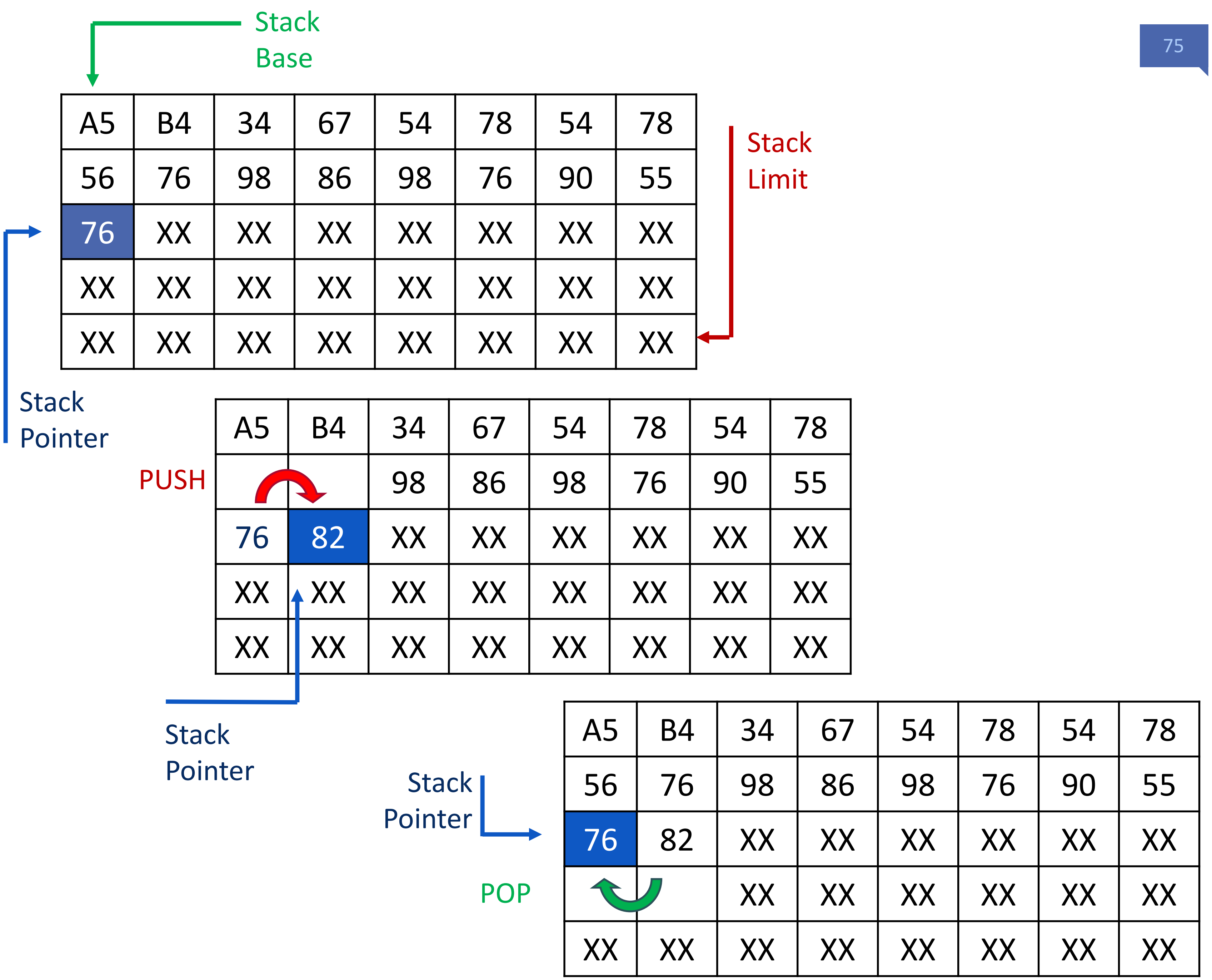
ADD A NEW DATA ON TOP OF THE STACK

REMOVE A DATA FROM THE TOP OF THE STACK

STACK PRINCIPLE

But the stack is not a stack of plate but a memory array.

- Adding a value on top of stack increment the stack pointer.
- Reaching the end of the stack will be a killer…
- Removing a value just move the stack pointer a step behind
- Stack is empty when Stack Pointer is Stack base



Stack Base

Stack Limit

Stack Pointer

A5	B4	34	67	54	78	54	78
56	76	98	86	98	76	90	55
76	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

PUSH

A5	B4	34	67	54	78	54	78
		98	86	98	76	90	55
76	82	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

Stack Pointer

Stack Pointer

POP

A5	B4	34	67	54	78	54	78
56	76	98	86	98	76	90	55
76	82	XX	XX	XX	XX	XX	XX
		XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

Author – Paul Pinault / Disk91.com

STACK IN FUNCTION EXECUTION

Stack can be use for different purpose:

- Store local variables
- Store call history
- Pass function parameters
- Pass function results

- Practically, for a faster execution, compile use a mix of register and stack for passing parameter and results.

```
int f1(int v) {
  if (v == 0) return 0;
  return f1(v-1)+v;
}

main() {
   int a = 2;
   b = f1(a);
}
```

XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

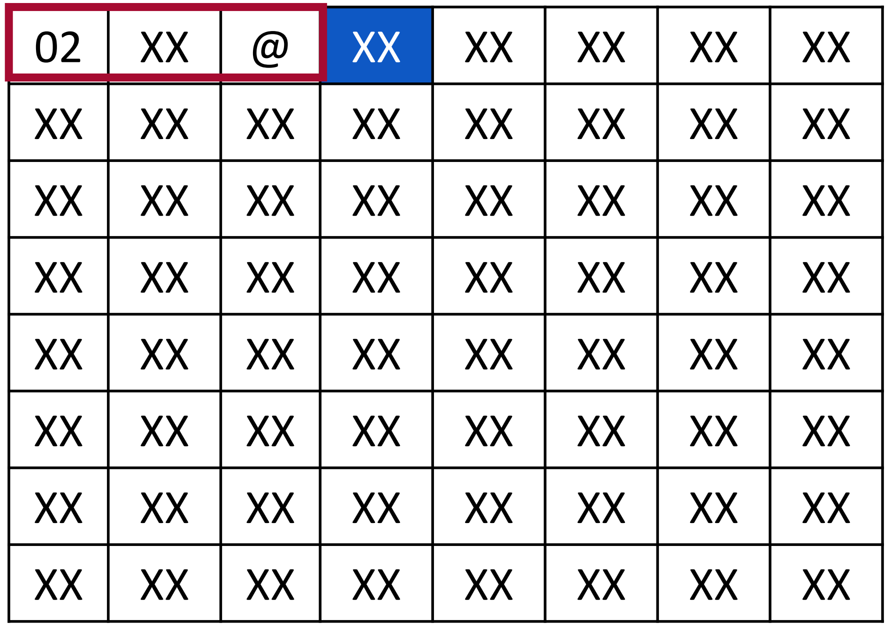
STACK IN FUNCTION EXECUTION

Here we have the call of the function f1, passing an argument (02) over the stack.

The result will be also in the stack to 1 place is reserved for it.

Then by calling the function, the address to return is pushed to the stack.

```
int f1(int v) {
  if (v == 0) return 0;
  return f1(v-1)+v;
}

main() {
   int a = 2;
   b = f1(a);
}
```



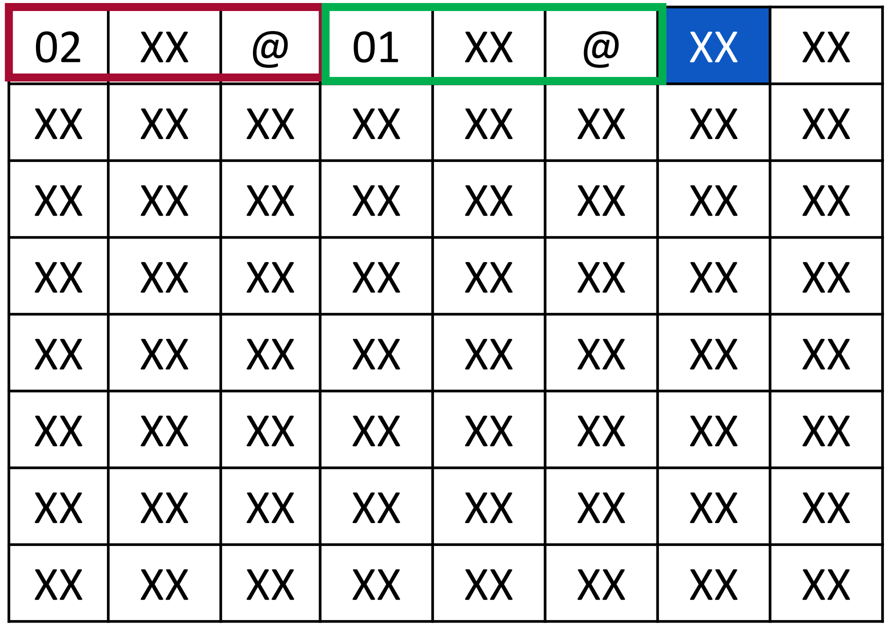
02	XX	@	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

STACK IN FUNCTION EXECUTION

Now f1 is calling itself with a new value 1.
By adding the same pattern on top of the stack we see how calls after calls the stack is filled with local function context.

The function could also have local variables included in its local context inside the stack.

```
int f1(int v) {
  if (v == 0) return 0;
  return f1(v-1)+v;
}

main() {
  int a = 2;
  b = f1(a);
}
```

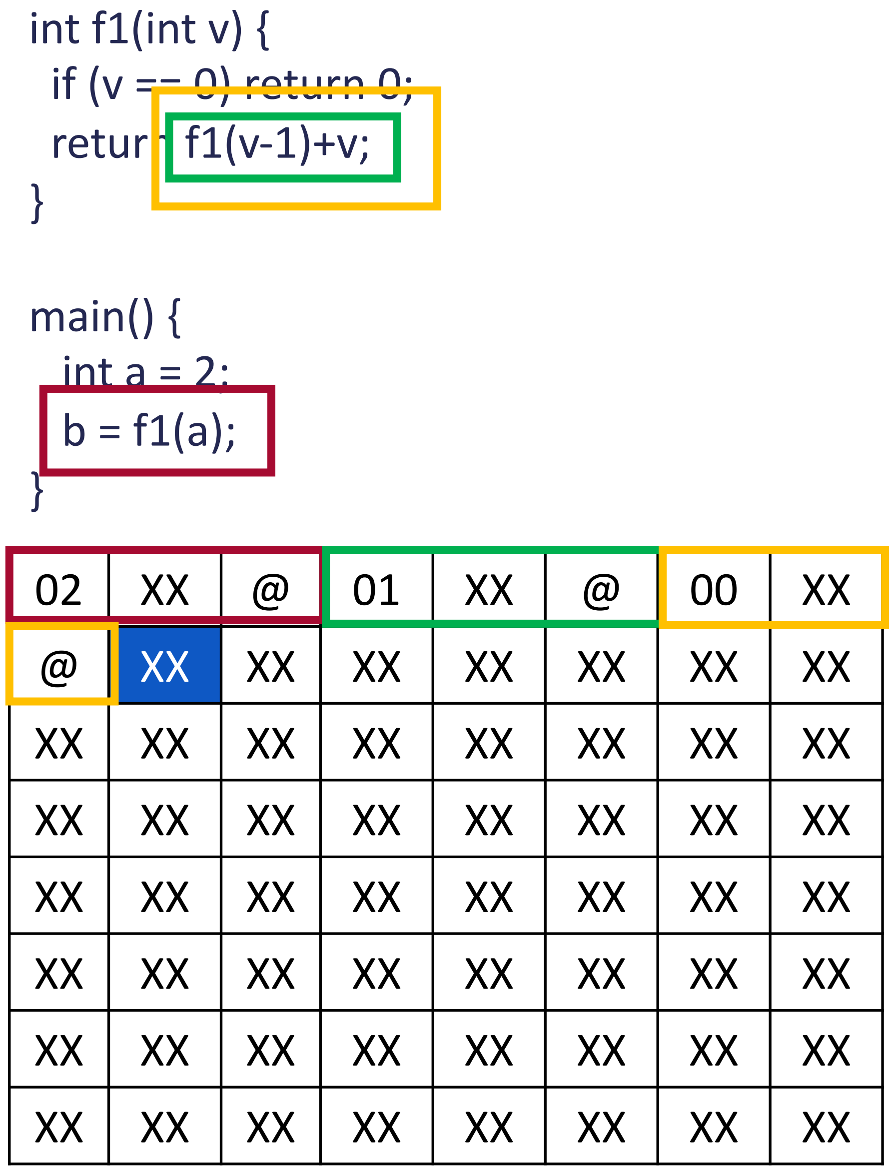


02	XX	@	01	XX	@	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

Author – Paul Pinault / Disk91.com

STACK IN FUNCTION EXECUTION

Here is the last call with the value 0 before starting to unstack the different calls.



```
int f1(int v) {
  if (v == 0) return 0;
  return f1(v-1)+v;
}

main() {
  int a = 2;
  b = f1(a);
}
```

02	XX	@	01	XX	@	00	XX
@	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

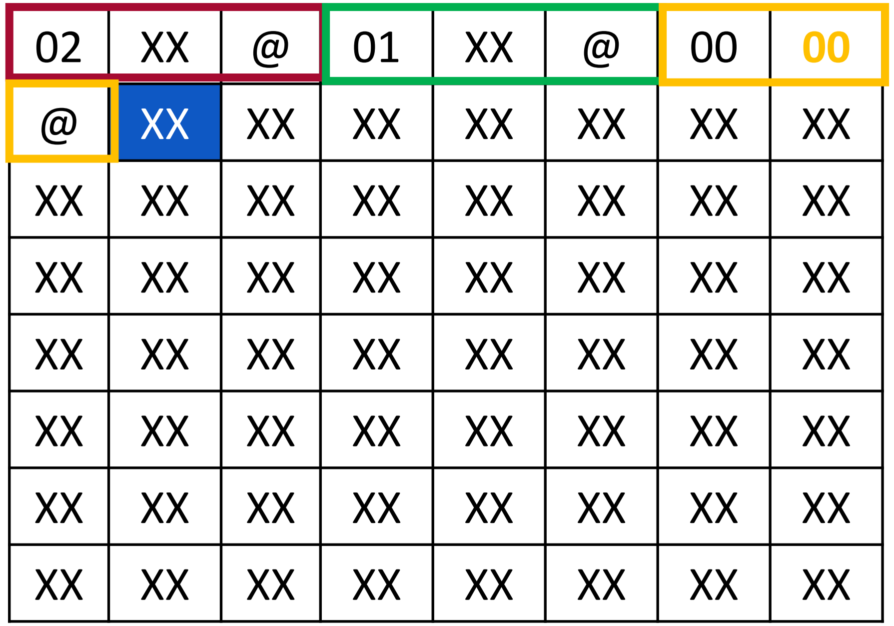
STACK IN FUNCTION EXECUTION

Now the f1(0) is ready to returned. The result value (0) is written in the Stack.

No need to be on top to be modified in fact, the function can access any of the value in its local stack context.

The saved @ allows to know where the program should jump back.

```
int f1(int v) {
  if (v == 0) return 0;
  retur  f1(v-1)+v;
}

main() {
   int a = 2;
   b = f1(a);
}
```



02	XX	@	01	XX	@	00	00
@	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

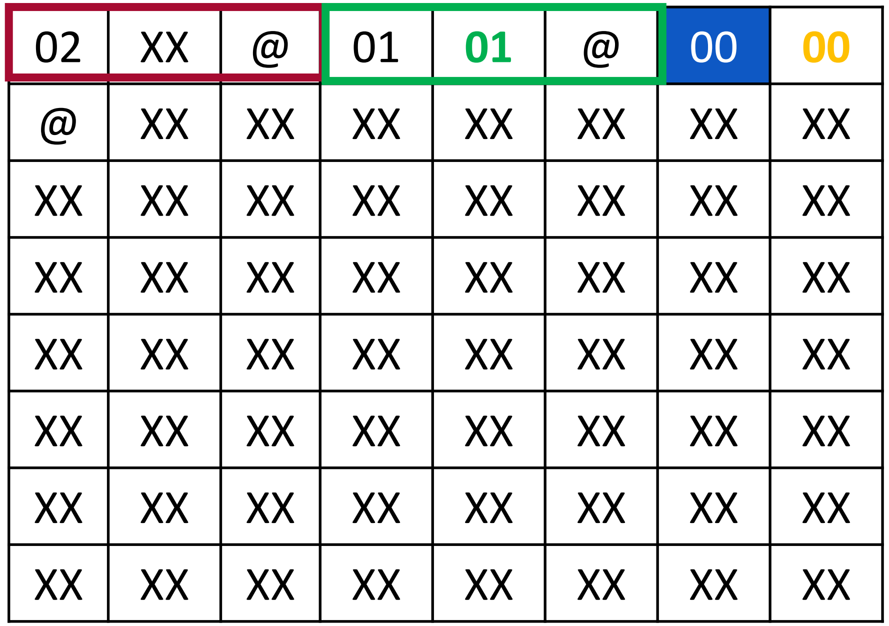
STACK IN FUNCTION EXECUTION

Function f1(1) local stack context is restored. The stack pointer is back on it.

The stack content f1(0) is not cleared. But it will be override by a future other local context.

The function can now compute the new local result and store it in the stack to pass it to f1(2)

```
int f1(int v) {
  if (v == 0) return 0;
  return f1(v-1)+v;
}

main() {
  int a = 2;
  b = f1(a);
}
```

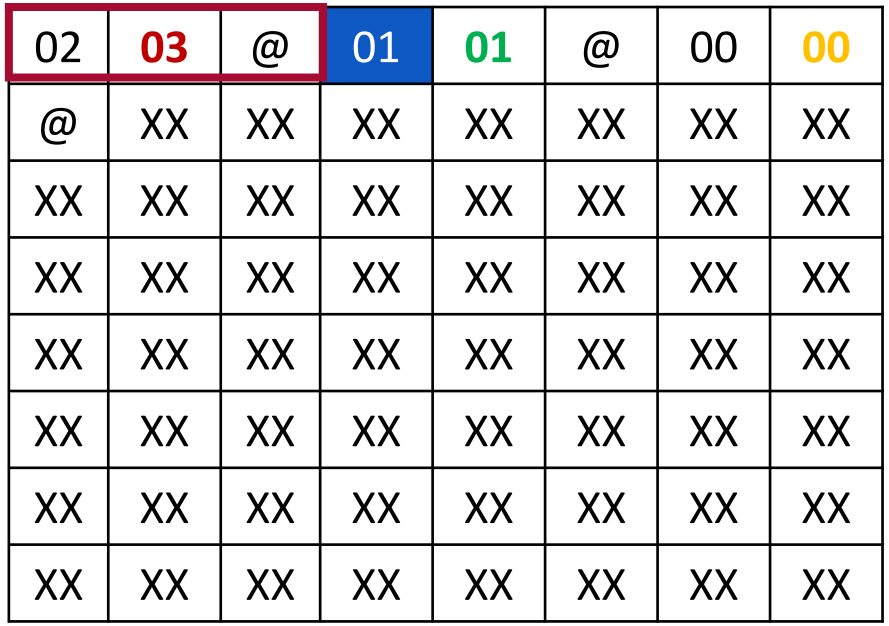


02	XX	@	01	01	@	00	00
@	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

STACK IN FUNCTION EXECUTION

The same thing append for f1(2)

```
int f1(int v) {
  if (v == 0) return 0;
  return f1(v-1)+v;
}

main() {
   int a = 2;
   b = f1(a);
}
```



02	03	@	01	01	@	00	00
@	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

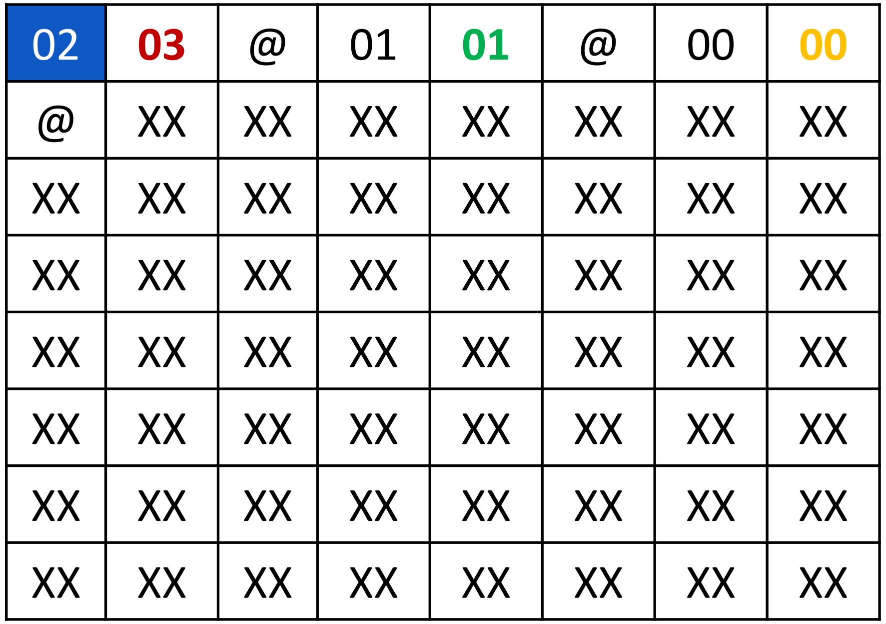
STACK IN FUNCTION EXECUTION

And finally, the Stack pointer is back to its initial state and the resulting value 03 can be retrieved.

The stack is ready for a new function call sequence …

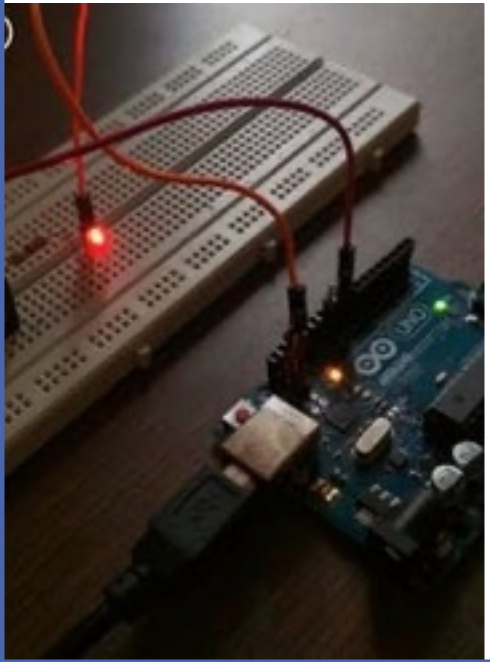
There is no memory fragmentation, local memory is dynamically allocated in stack and also globally free when terminating the function.

```
int f1(int v) {
  if (v == 0) return 0;
  return f1(v-1)+v;
}

main() {
   int a = 2;
   b = f1(a);
}
```

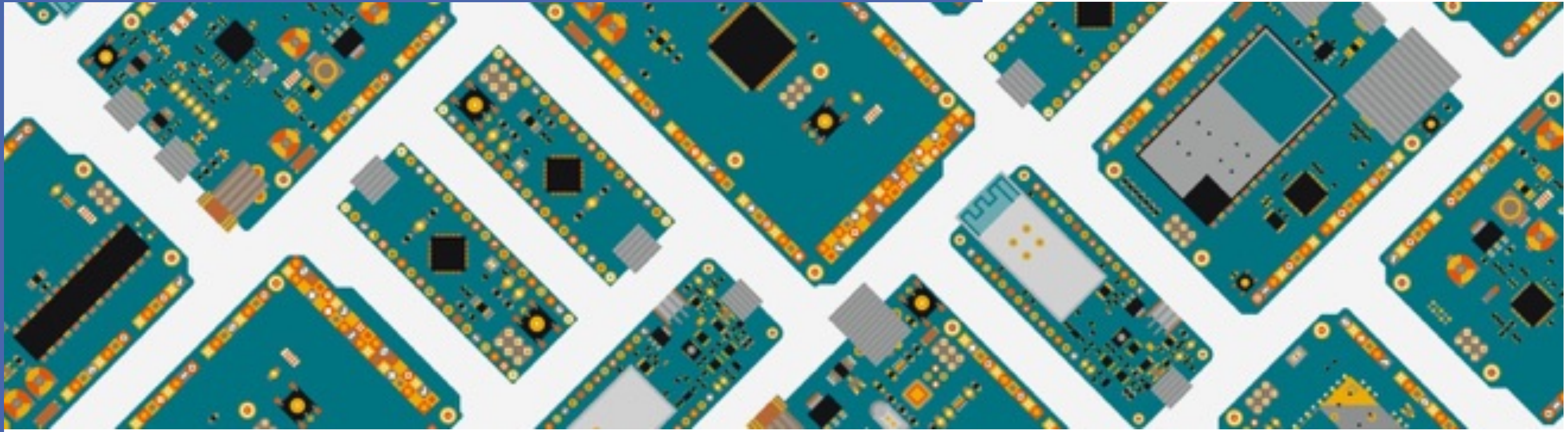


02	**03**	@	01	**01**	@	00	**00**
@	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

Modern systems architectures



```
  This example code is in the public domain.

  http://www.arduino.cc/en/Tutorial/Blink
*/

// the setup function runs once when you press rese
void setup() {
  // initialize digital pin LED_BUILTIN as an outpu
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again foreve
void loop() {$
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LE
  delay(1000);                       // wait for a
  digitalWrite(LED_BUILTIN, LOW);    // turn the LE
  delay(1000);                       // wait for a
}
```

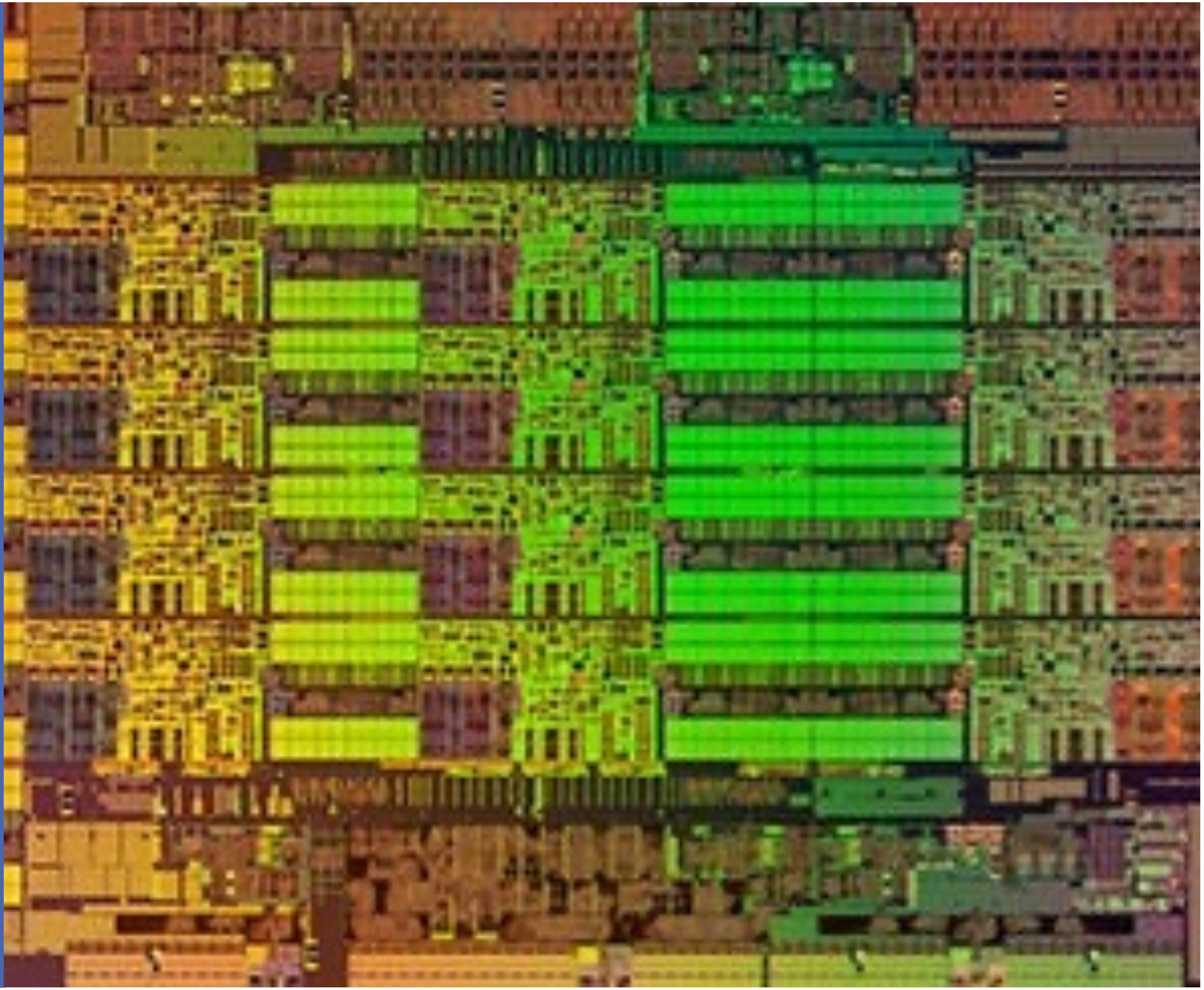


Author – Paul Pinault / Disk91.com

ARDUINO TIME SOURCE

The modern CPU architecture is integrating many features in the internal design to improve the performance:

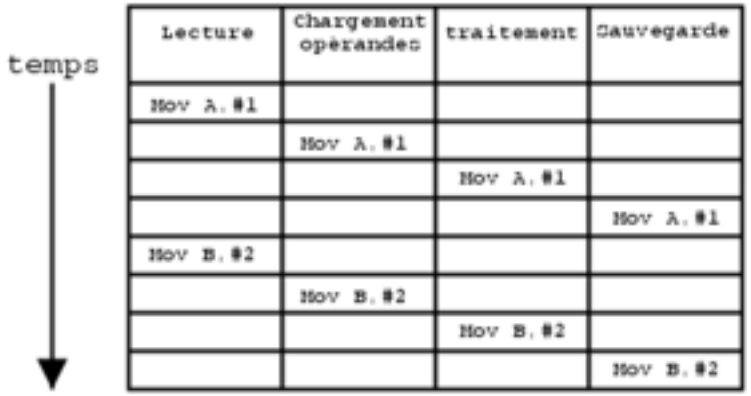
- Pipeline
- Superscalar architecture
- Hyper - Threading
- Caches
- Multi-core
- Coprocessors
- Memory Management Unit
- Virtualization
- …



Modern processor architecture

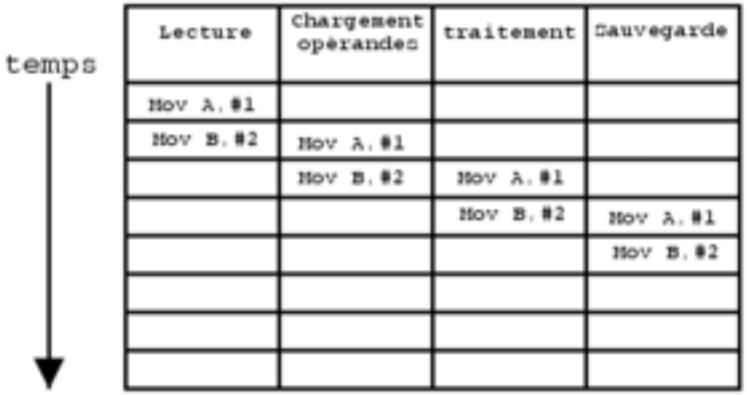
PIPELINES

```
Programme : Mov A,#1
            Mov B,#2
```



Lecture	Chargement opérandes	traitement	Sauvegarde
Mov A,#1			
	Mov A,#1		
		Mov A,#1	
			Mov A,#1
Mov B,#2			
	Mov B,#2		
		Mov B,#2	
			Mov B,#2

Exécution sans pipeline sur 8 cycles



Lecture	Chargement opérandes	traitement	Sauvegarde
Mov A,#1			
Mov B,#2	Mov A,#1		
	Mov B,#2	Mov A,#1	
		Mov B,#2	Mov A,#1
			Mov B,#2

Exécution avec pipeline sur 5 cycles

SPEED = 1 Instruction / cycle

Each of the processor instructions are split in different steps (as an example):

- Loading incoming data
- Proceed to operation
- Store the result

Each of these steps could take one clock cycle as this is the minimal period of time the CPU can manage.

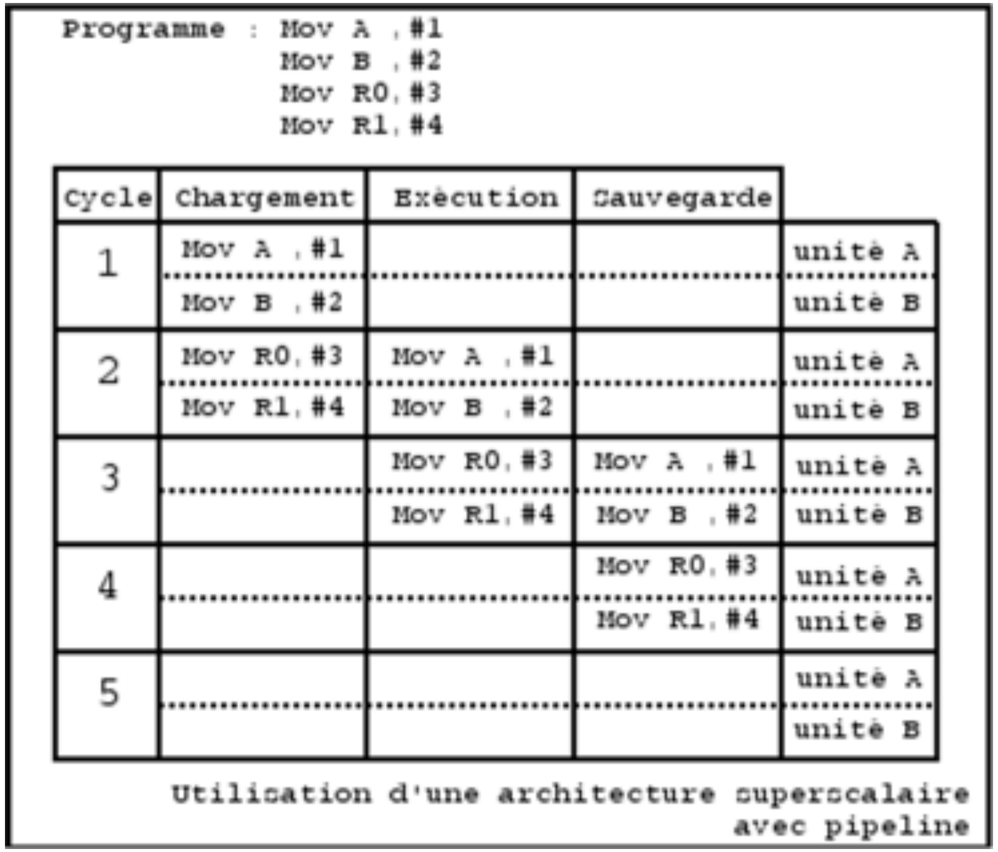
With the use of a pipeline, it is possible to run in parallel different instruction across these steps.

The pipeline throughput is 1 instruction per cycle.

This is particularly used with CISC CPU.

Author – Paul Pinault / Disk91.com

SUPERSCALAR ARCHITECTURE



```
Programme : Mov A , #1
            Mov B , #2
            Mov R0, #3
            Mov R1, #4
```

Cycle	Chargement	Exécution	Sauvegarde	
1	Mov A , #1			unité A
	Mov B , #2			unité B
2	Mov R0, #3	Mov A , #1		unité A
	Mov R1, #4	Mov B , #2		unité B
3		Mov R0, #3	Mov A , #1	unité A
		Mov R1, #4	Mov B , #2	unité B
4			Mov R0, #3	unité A
			Mov R1, #4	unité B
5				unité A
				unité B

Utilisation d'une architecture superscalaire avec pipeline

SPEED = 2 Instructions / cycle

We have, inside a single core, multiple execution units.

They are processing in parallel multiple instructions of a same program.

Basically, taking instruction 2 by two instead of one by one. We have two pipelines.

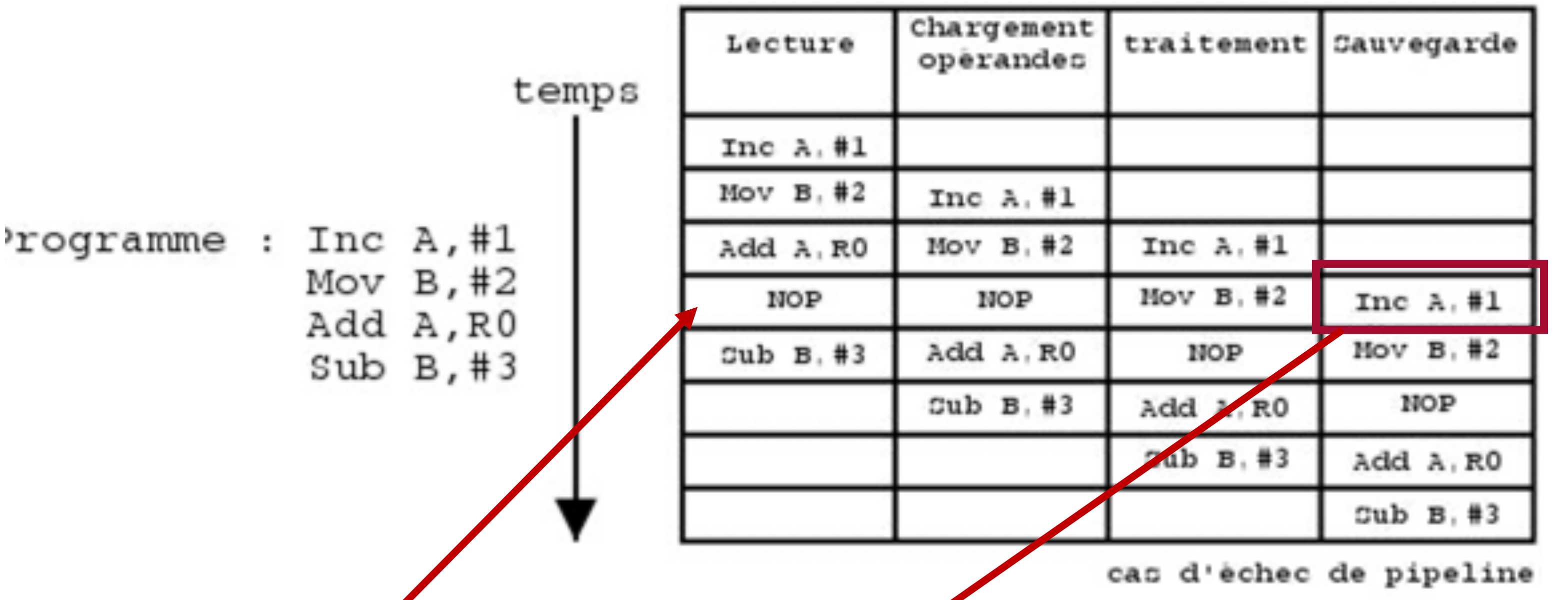
This, in theory allows to get multiple instruction to be delivered on every clock cycle.

But Pipeline is not perfect

Instruction can be in conflict and prohibit the parallel execution inside the pipeline.

In a such case a NOP is inserted.

Compilers have special algorithm for managing this. Some CPU dynamically reorder code to manage this.



Lecture	Chargement opérandes	traitement	Sauvegarde
Inc A,#1			
Mov B,#2	Inc A,#1		
Add A,R0	Mov B,#2	Inc A,#1	
NOP	NOP	Mov B,#2	Inc A,#1
Sub B,#3	Add A,R0	NOP	Mov B,#2
	Sub B,#3	Add A,R0	NOP
		Sub B,#3	Add A,R0
			Sub B,#3

cas d'échec de pipeline

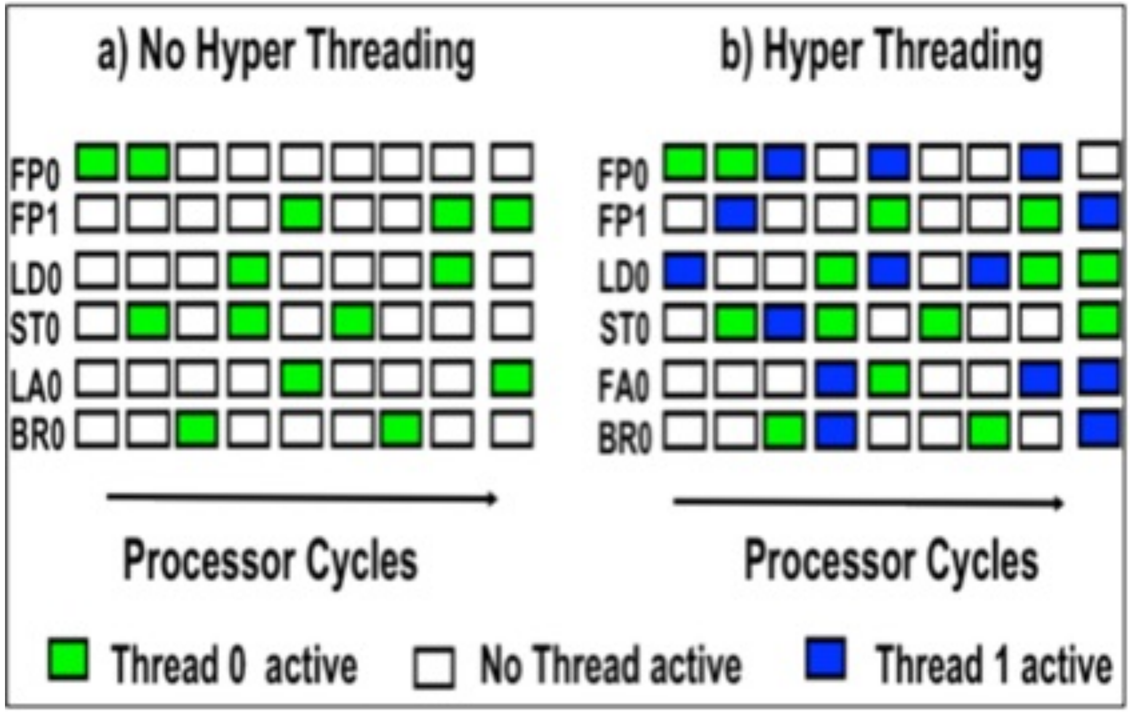
WE CAN'T LOAD "A" UNTIL THE "INC" RESULT HAS BEEN SAVED

A "NOP" is inserted a cycle has been lost

THE SAME THING HAPPEN WITH JUMPS
Modern CPU have predictive jumps.

SPEED = 0.X to 1.X Instructions / cycle

Hyper - Threading



a) No Hyper Threading

FP0	🟩	🟩	⬜	⬜	⬜	⬜	⬜	⬜	⬜
FP1	⬜	⬜	⬜	⬜	🟩	⬜	⬜	🟩	🟩
LD0	⬜	⬜	⬜	🟩	⬜	⬜	⬜	🟩	⬜
ST0	⬜	🟩	⬜	🟩	⬜	🟩	⬜	⬜	⬜
LA0	⬜	⬜	⬜	⬜	🟩	⬜	⬜	⬜	🟩
BR0	⬜	⬜	🟩	⬜	⬜	⬜	🟩	⬜	⬜

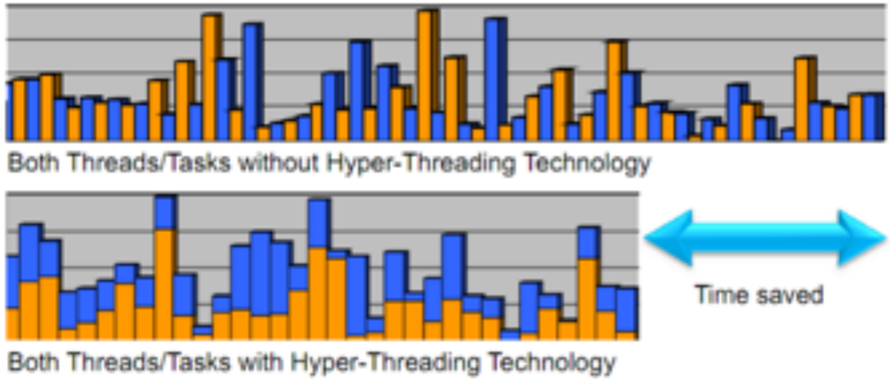
Processor Cycles

b) Hyper Threading

FP0	🟩	🟩	🟦	⬜	🟦	⬜	⬜	🟦	⬜
FP1	⬜	🟦	⬜	⬜	🟩	⬜	⬜	🟩	🟦
LD0	🟦	⬜	⬜	🟩	🟦	⬜	🟦	🟩	🟩
ST0	⬜	🟩	🟦	🟩	⬜	🟩	⬜	⬜	🟩
FA0	⬜	⬜	⬜	🟦	🟩	⬜	⬜	🟦	🟦
BR0	⬜	⬜	🟩	🟦	⬜	⬜	🟩	⬜	🟦

Processor Cycles

🟩 Thread 0 active ⬜ No Thread active 🟦 Thread 1 active



SPEED = 1.1 – 2 Instructions / cycle

As we are inserting many NOPs on the instruction flow to manage the memory and register access conflicts + not correctly predicted jumps, the idea is to replace these NOPs by instruction to execute.

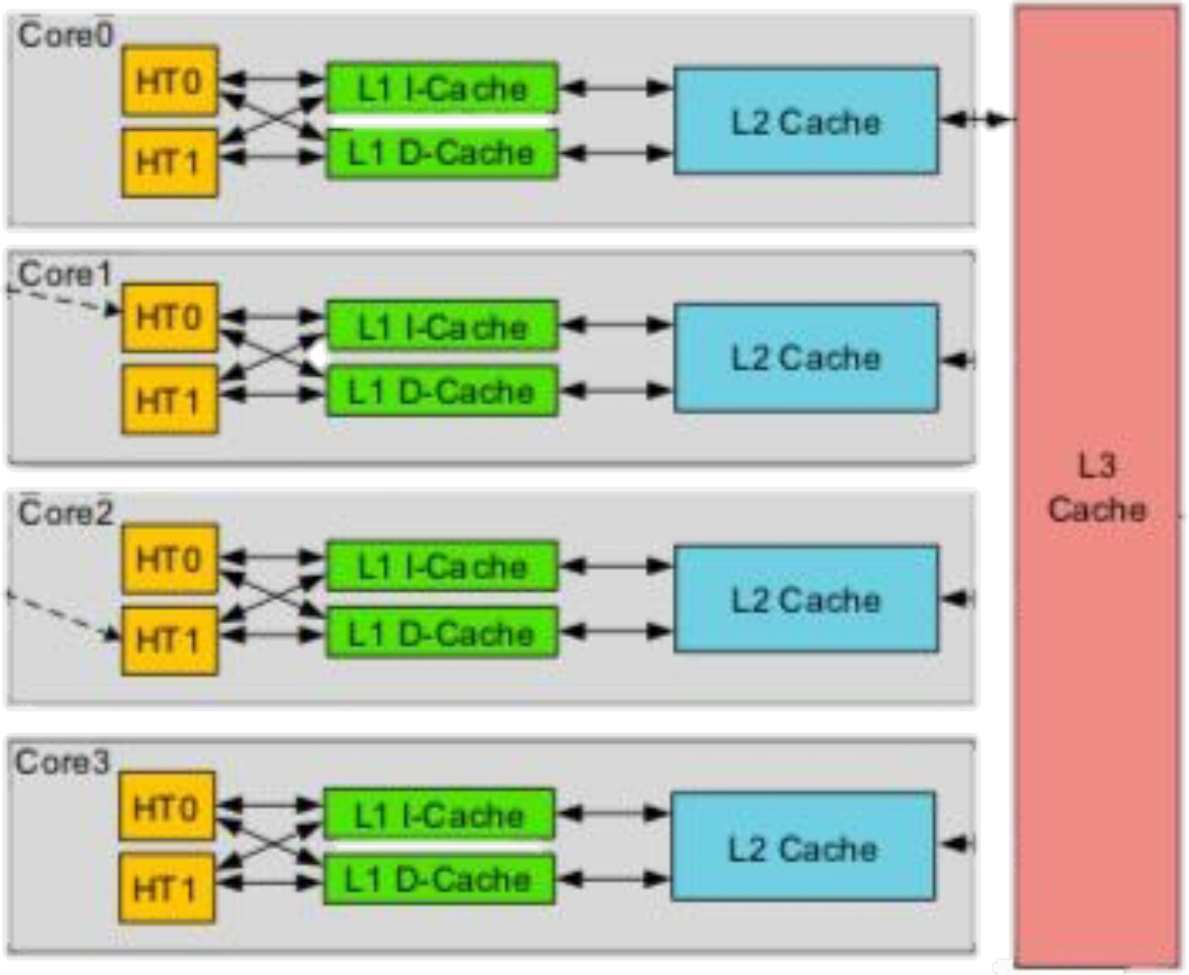
To be sure these instructions won't be in conflicts with the other one, they come from a different process or thread.

Therefore, it is hyper-threading. This is seen as a different processor even if it is composed of a single ALU.

This is corresponding to the Thread factor when you have a CPU with 2 cores / 4 threads.

Modern processor architecture

Memory Caching



SPEED = 1.1 – 2 Instructions / cycle

Memory is slow compared to processor computing capability.

Producer :
DDR4-3200 ~ 25GB/s

Consumer :
CPU = 32 core / 64 threads @ 3.2 Ghz ~ 204 GB/s

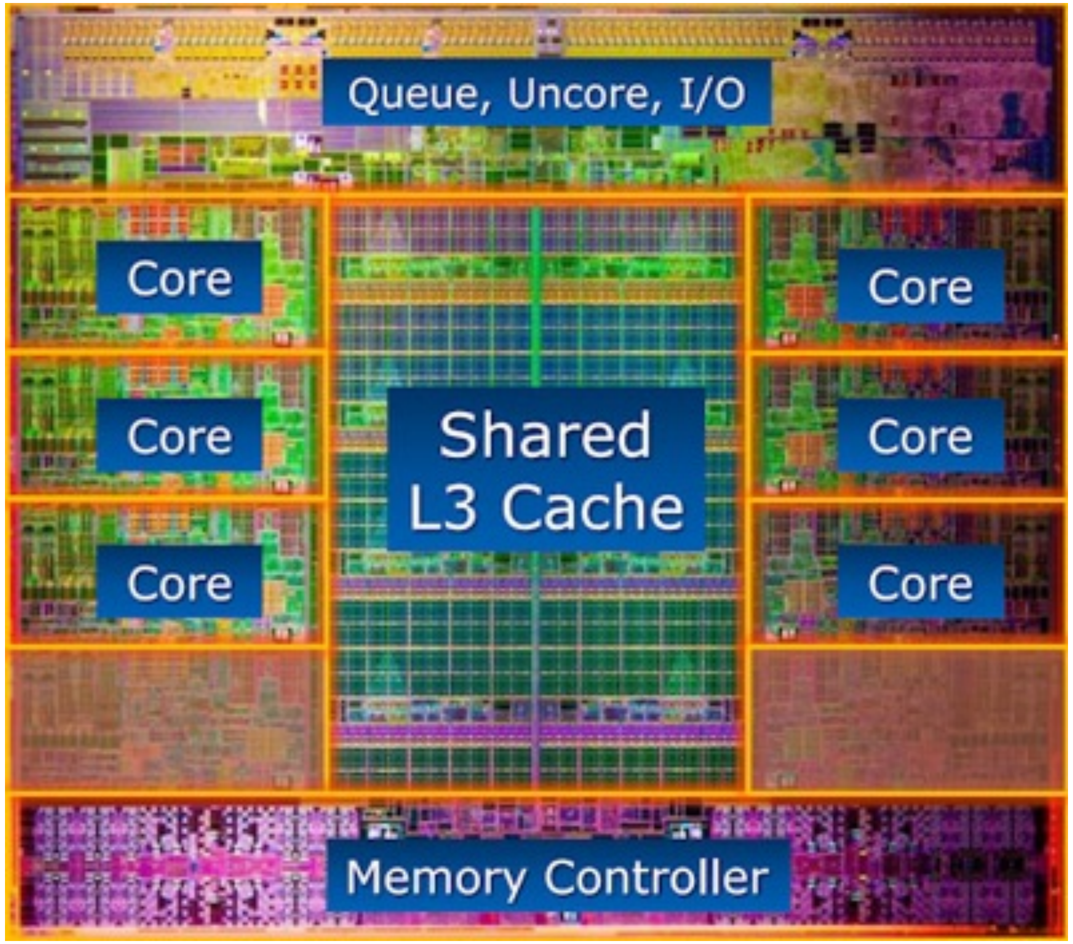
To avoid slowing down the processor computation power by 8, we use caches to store locally a data accessible in a faster way.

As a consequence, we can have cache miss and cache conflicts to manage.

Program can be optimized related to cache size.

Modern processor architecture

Multi code processor



SPEED = 64 – 128 Instructions / cycle

A processor is a big chip with a lots of pins. It is hard to get more than 4 on a single motherboard or 16 in a single servers.

In a way to be more efficient, we put altogether multiple CORE (processor) in a single Socket (chip). That way we can have 32 / 64 / … in a single socket and have 4, 8… sockets in a server for a total of:

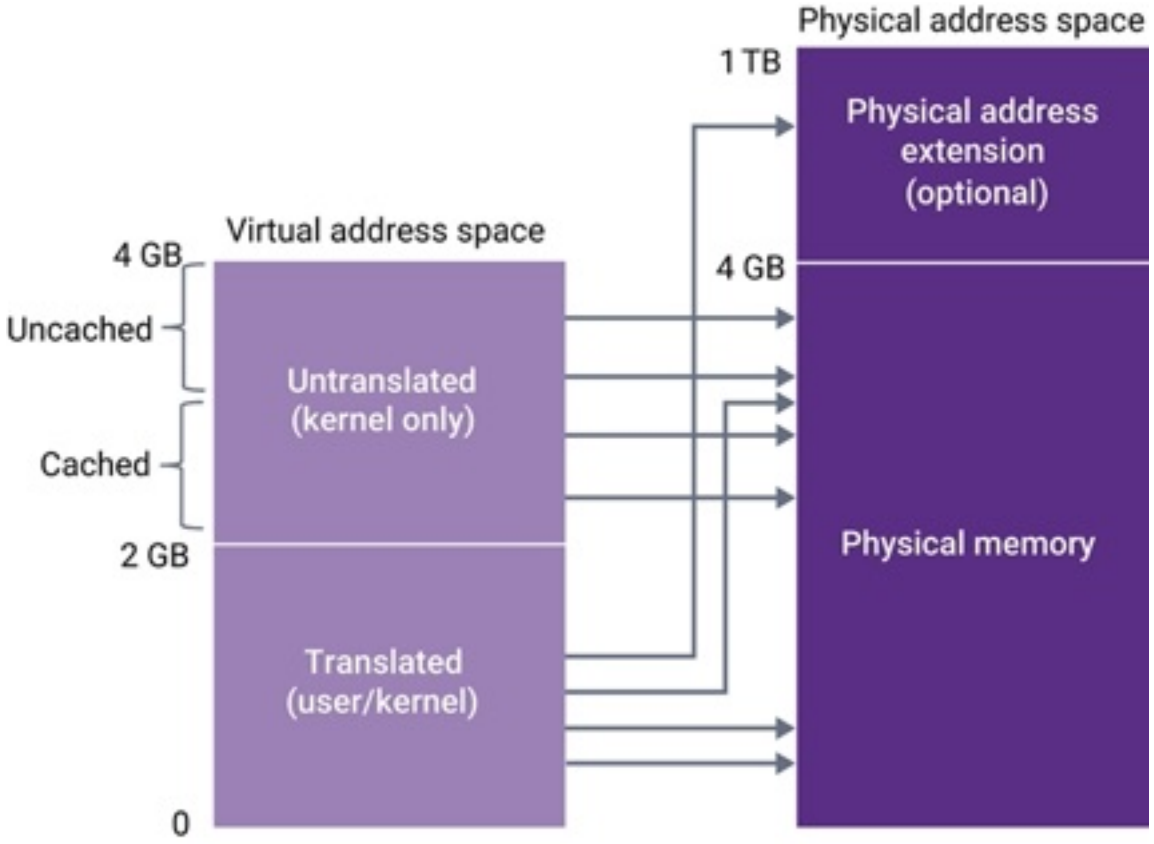
64 x 8 = 256 core / 512 Thread in a server… (or more depends on hardware and technology)

Co-Processor

Hardware dedicated to some specific function like 3D rendering and calculation, encryption, IA … allows to improve the performance by hardcoding some complex instructions in hardware.

Co-processor

Memory Management Unit (MMU)



The MMU manage the addressable memory for the MCU. Is reduce the operating system activity to manage the process memory mapping into the physical memory.

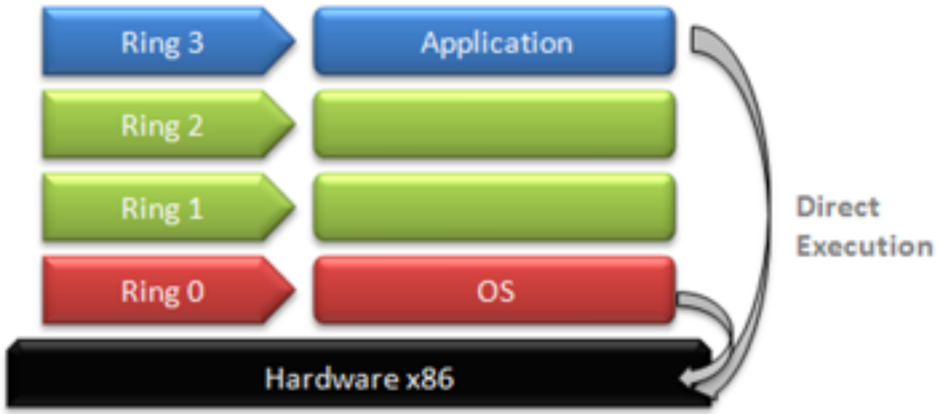
Each of the process will thing to be alone in memory and have access to the whole available and continuous memory.

The MMU also manage the access right on memory to ensure a process or a specific zone is not accessed by a wrong process.

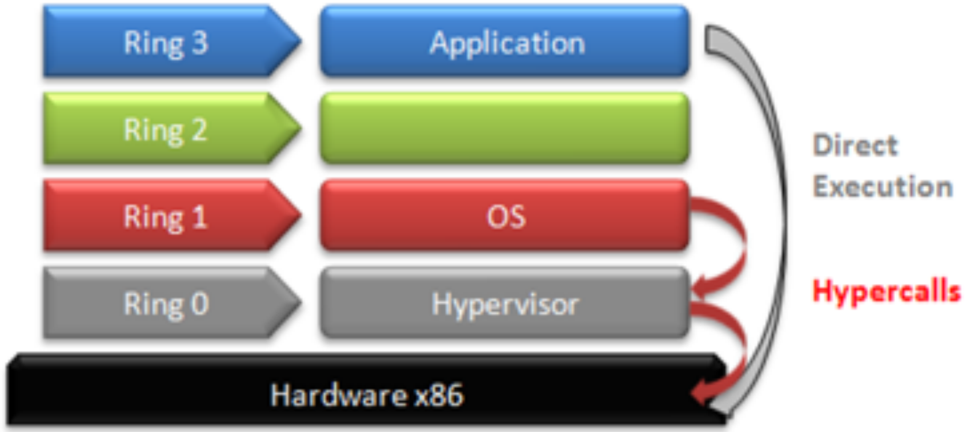
Author – Paul Pinault / Disk91.com

Co-processor

Virtualization



CLASSICAL EXECUTION OF AN OPERATING SYSTEM / APPLICATION ON A CPU



WHAT IS HAPPENING WITH A HYPERVISOR RUNNING IN SUCH SITUATION

An instruction is executed on a certain Ring level.

RING0 have access on hardware and can control what is executed on Ring 1 to 3.

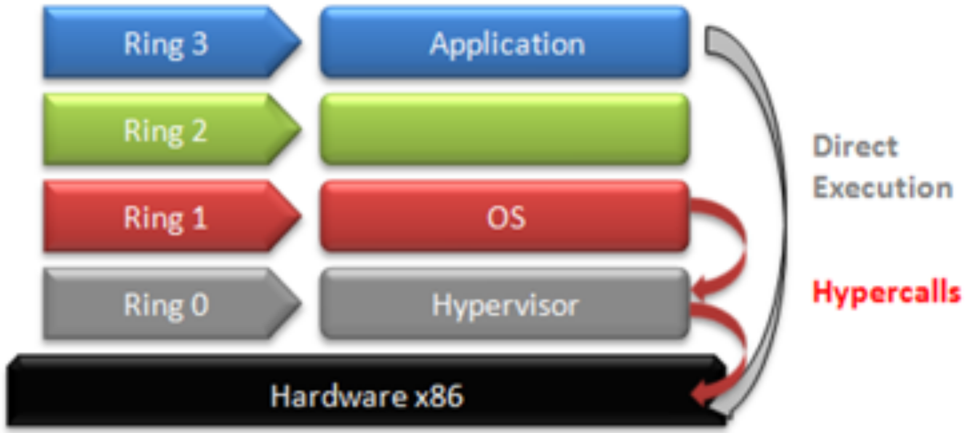
Application are running on RING3 and can’t control lower ring and hardware.

Operating systems run on ring 0, Application on Ring 3.

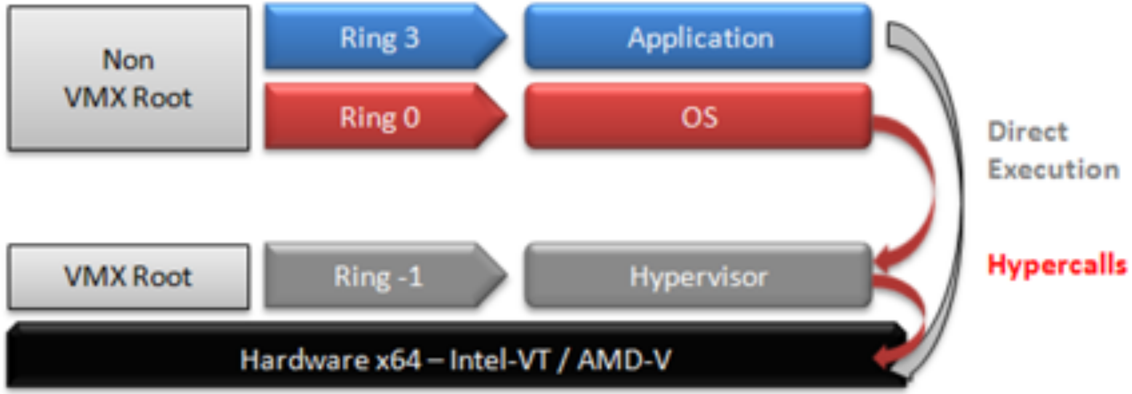
Author – Paul Pinault / Disk91.com

Co-processor

Virtualization



WHAT IS HAPPENING WITH A HYPERVISOR RUNNING IN SUCH SITUATION



WITH A CPU EXTENSION MANAGING VIRTUALIZATION

The operating system is not RING-0 and needs to manage this situation, it needs to be compiled especially for it.

Virtualization CPU extension allows to process it with a better efficiency.

A new RING -1 run the Hypervisor so the Operating system can run on RING 0.

For the operating system it is exactly as if it is running bare-metal.

The RING -1 also have some specific instructions to optimize the virtualization performance.

Author – Paul Pinault / Disk91.com

ARCHITECTURE IS PERFORMANCE

The same program running on different machine but taking benefit of the architecture evolutions

CPU / FREQ	Exécution time	Time for 1Mhz	Architecture change
386 / 25MHz	291 s	7275s	
486 / 33MHz	161 s	5313s	½ pipeline
586 / 60MHz	31 s	1860s	Pipeline + superscal.
PII / 450MHz	1,68 s	756s	
PIII / 800MHz	0,87 s	696s	
Athlon / 1.2GHz	0,43s	516s	