Chapter 4

Microchip PIC18 Architecture

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

CC BY NC SA

2021-2022

Market shares of silicon manufacturers and MCU suppliers in 2021.

2Q21 Top 10 Semiconductor Sales Leaders ($M, Including Foundries)

2Q21 Rank	1Q21 Rank	Company	Headquarters	1Q21 Total IC	1Q21 Total O-S-D	1Q21 Total Semi	2Q21 Total IC	2Q21 Total O-S-D	2Q21 Total Semi	2Q21/1Q21 % Change
1	2	Samsung	South Korea	16,152	920	17,072	19,262	1,035	20,297	19%
2	1	Intel	U.S.	18,676	0	18,676	19,304	0	19,304	3%
3	3	TSMC (1)	Taiwan	12,911	0	12,911	13,315	0	13,315	3%
4	4	SK Hynix	South Korea	7,270	358	7,628	8,762	451	9,213	21%
5	5	Micron	U.S.	6,629	0	6,629	7,681	0	7,681	16%
6	6	Qualcomm (2)	U.S.	6,281	0	6,281	6,472	0	6,472	3%
7	8	Nvidia (2)	U.S.	4,842	0	4,842	5,540	0	5,540	14%
8	7	Broadcom Inc. (2)	U.S.	4,364	485	4,849	4,400	490	4,890	1%
9	10	MediaTek (2)	Taiwan	3,849	0	3,849	4,496	0	4,496	17%
10	9	TI	U.S.	3,793	235	4,028	4,030	269	4,299	7%
—	—	Top-10 Total		84,767	1,998	86,765	93,262	2,245	95,507	10%

(1) Foundry (2) Fabless

Source: Company reports, IC Insights' *Strategic Reviews* database

Leading MCU Suppliers ($M)

2021 Rank	Company	Headquarters	2020	2021	21/20 % Chg	2021 Marketshare
1	NXP	Europe	2,980	3,795	27%	18.8%
2	Microchip	U.S.	2,872	3,584	25%	17.8%
3	Renesas	Japan	2,748	3,420	24%	17.0%
4	ST	Europe	2,506	3,374	35%	16.7%
5	Infineon	Europe	1,953	2,378	22%	11.8%

Source: Company reports, IC Insights

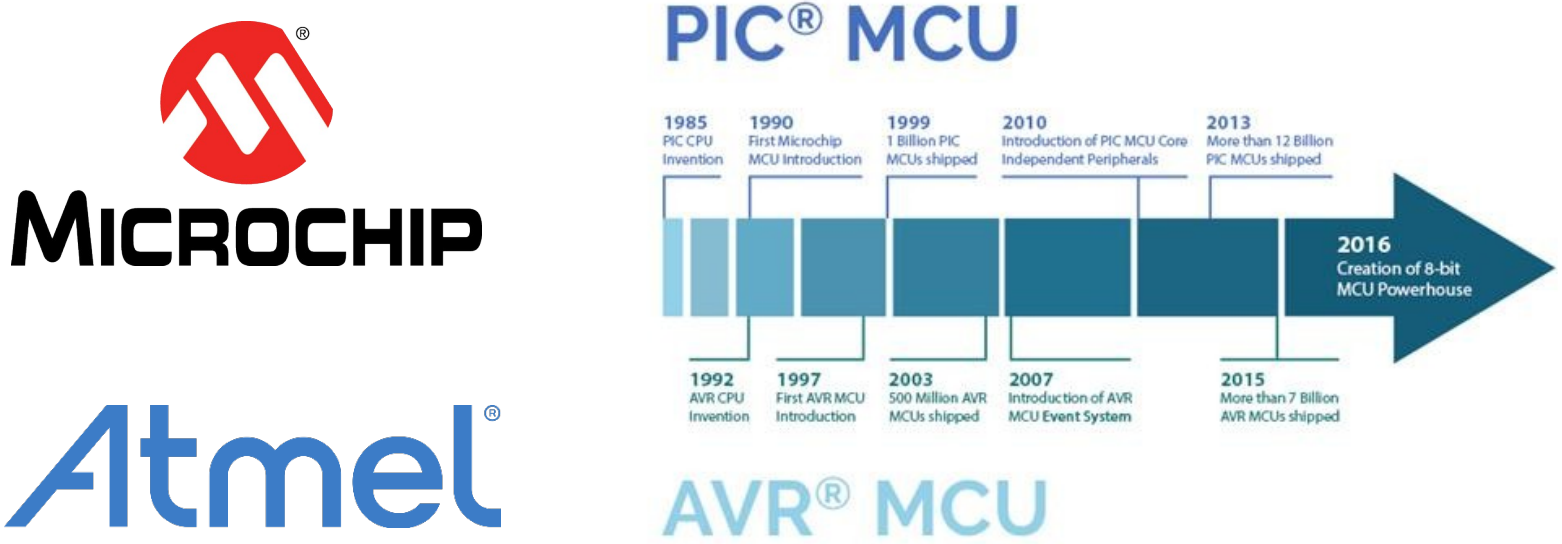
Design Only

Manufacturing Only

Technology used during labs

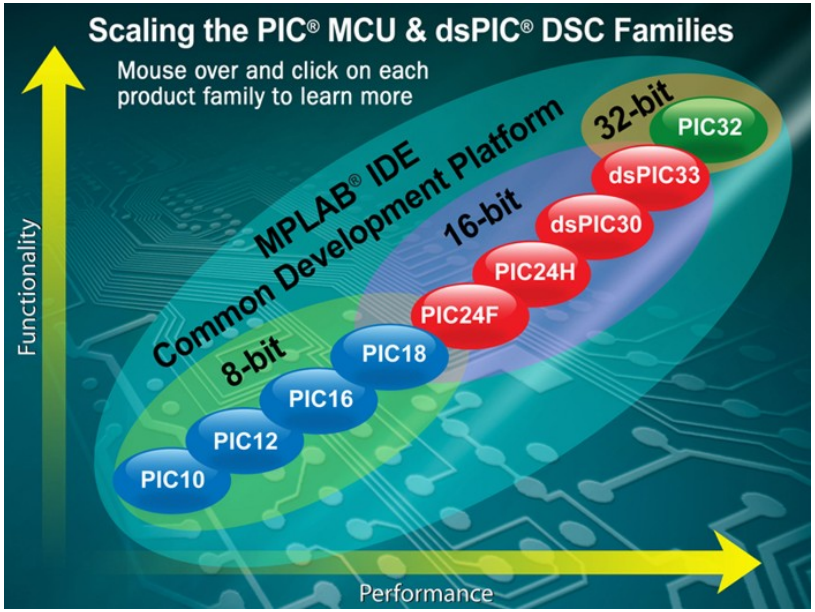
The American company Microchip is an electronic device manufacturer. Most of its turnover (fr: *chiffre d'affaires*) is due to MCUs: about 60% come from the PIC family according to Microchip ESC Filing.

In 2016 Microchip bought Atmel, its major concurrent on the 8-bit MCU market.



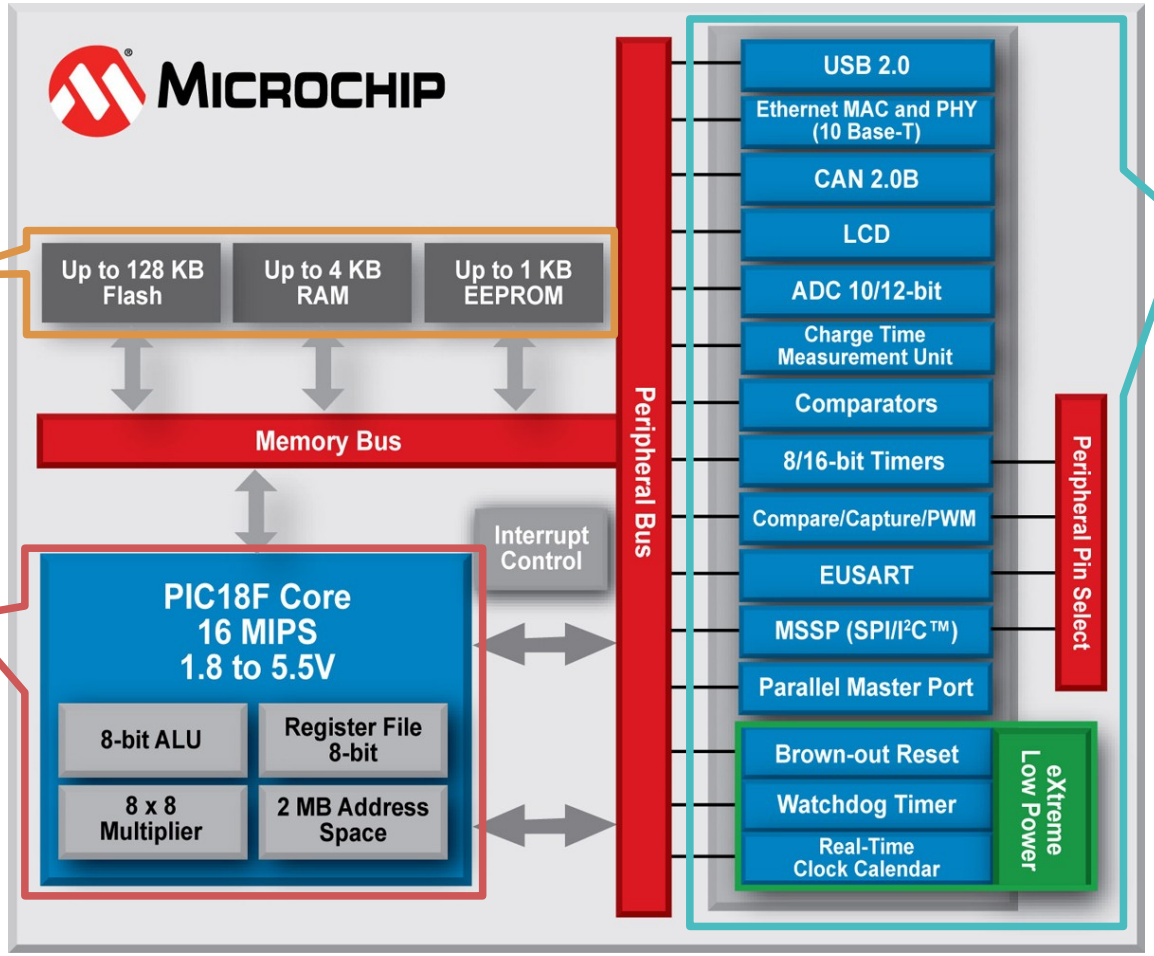
With its large range of MCU solutions, Microchip can win its clients loyalty by offering them the possibility to aim for various applications and markets.

Like many manufacturers, Microchip also supplies tools that make it easy to switch from a specific architecture to another (e.g. migration from PIC18 to PIC32).



Program & data memories

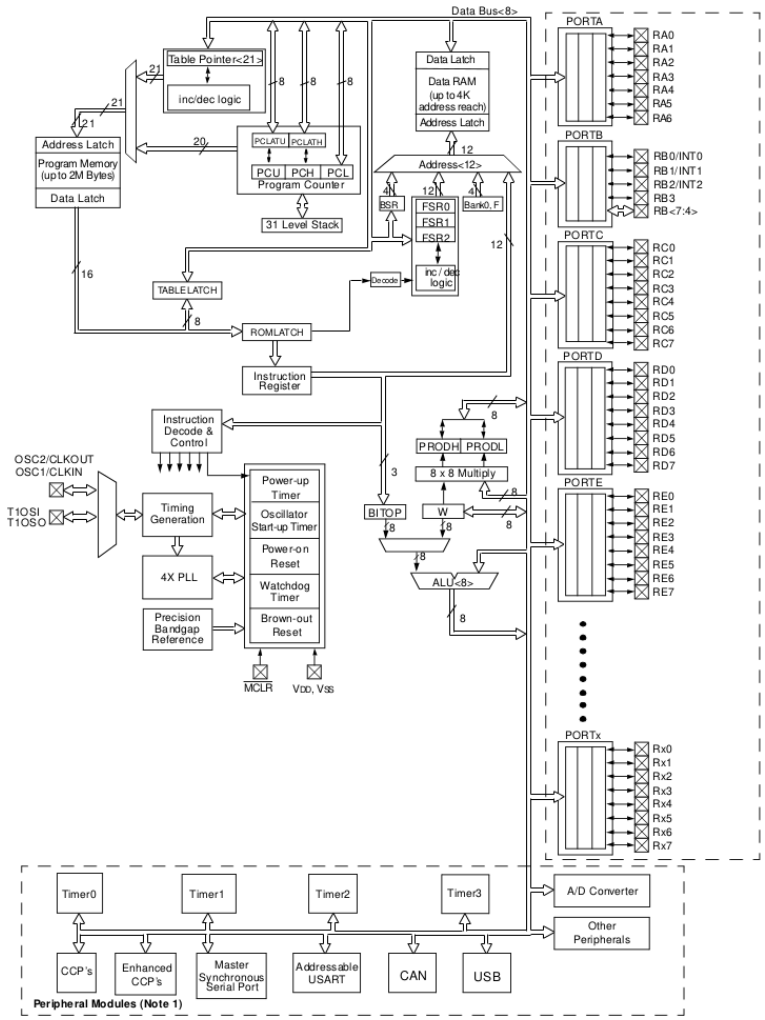
Central Processing Unit



Peripherals

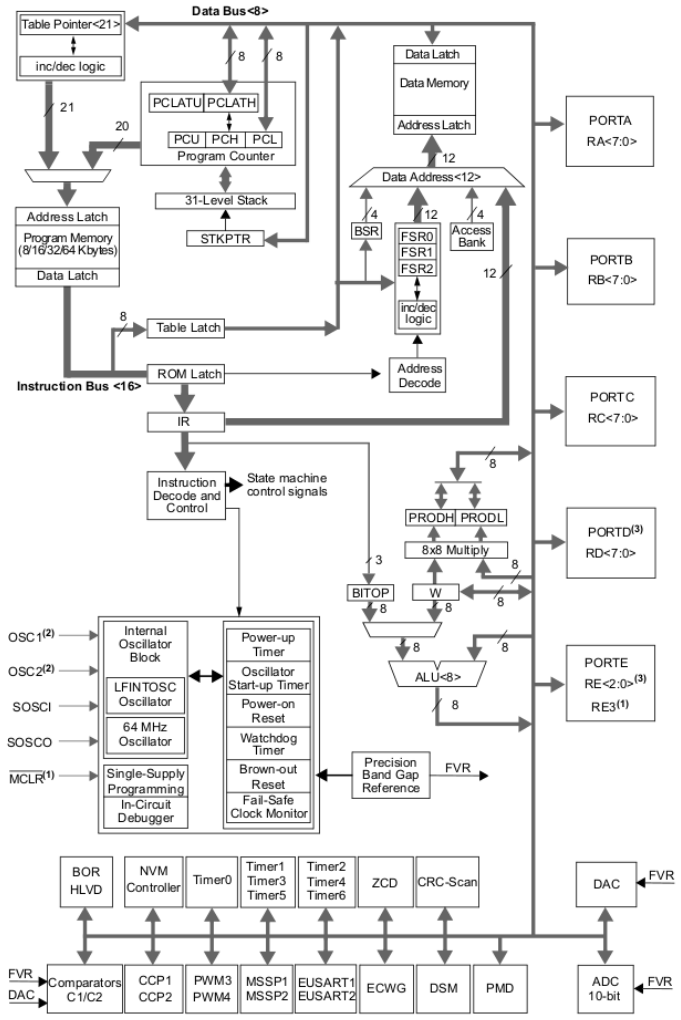
MICROCHIP PIC18 MCUs

Spot the differences!



PIC18C family (left)

PIC18F27/47K40 (right)



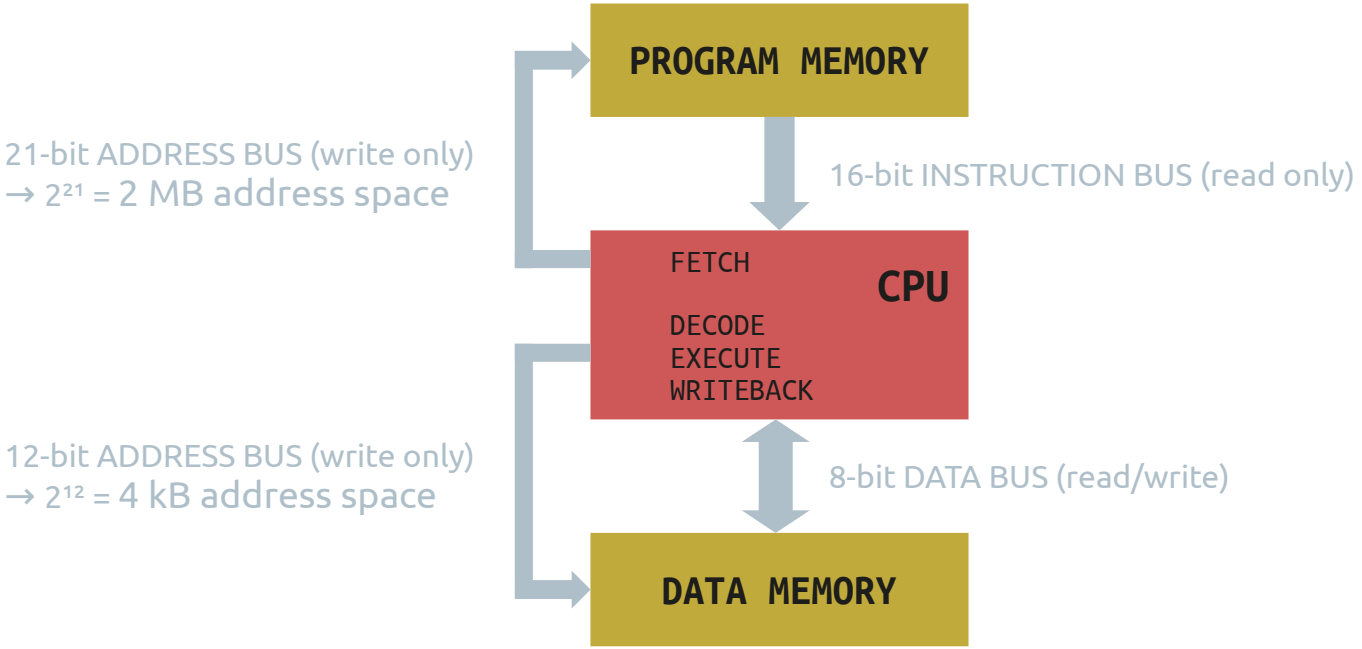
CENTRAL PROCESSING UNIT

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Architecture

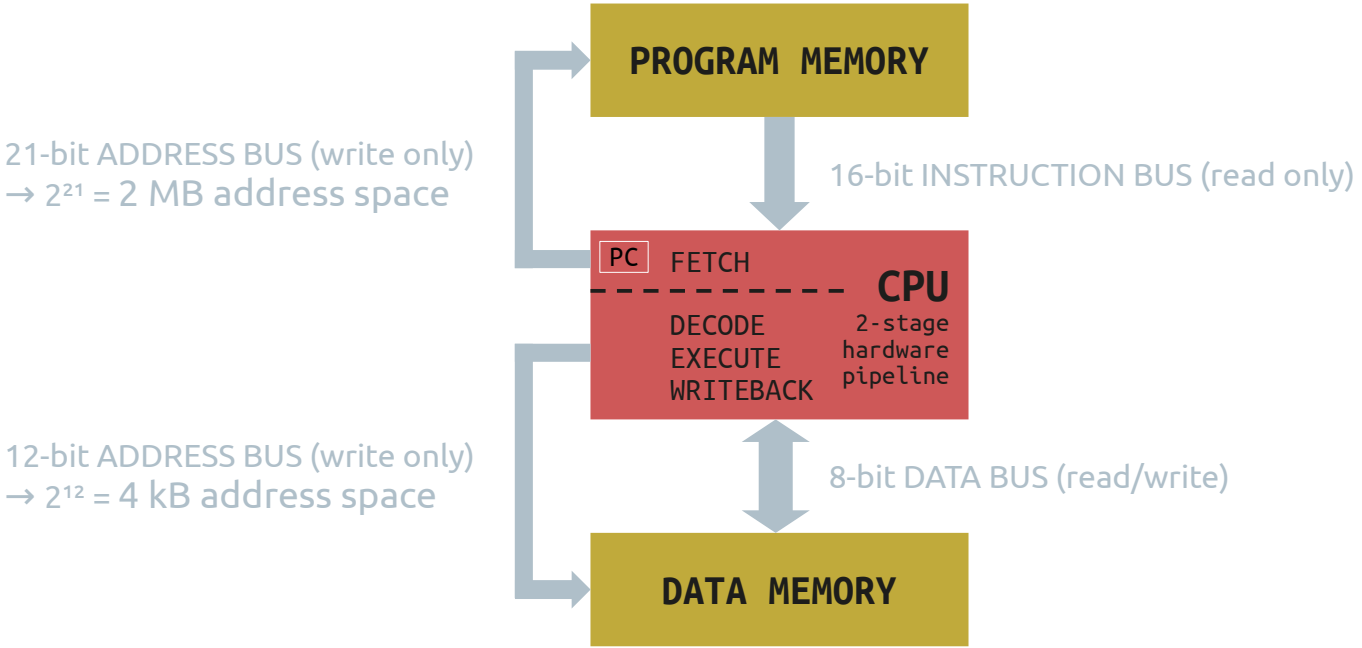
Just like the Atmel AVR, Microchip PIC18 follow a Harvard architecture: program and data memories are physically separated and their own addressing space are distinct.



Architecture

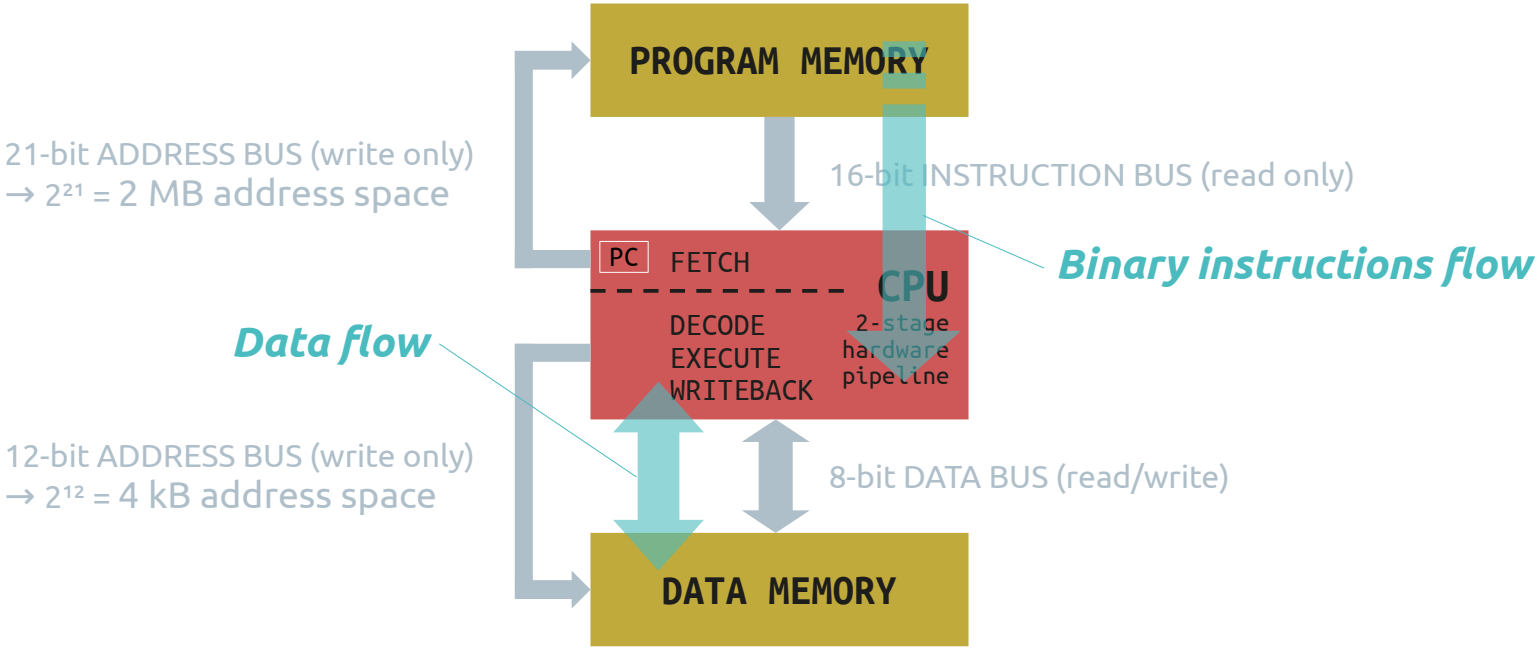
PIC18 CPUs are designed with a 2-stages hardware pipeline. They can decode-execute-store an instruction while fetching the next one from the program memory.

Max performance of 16 MIPS.



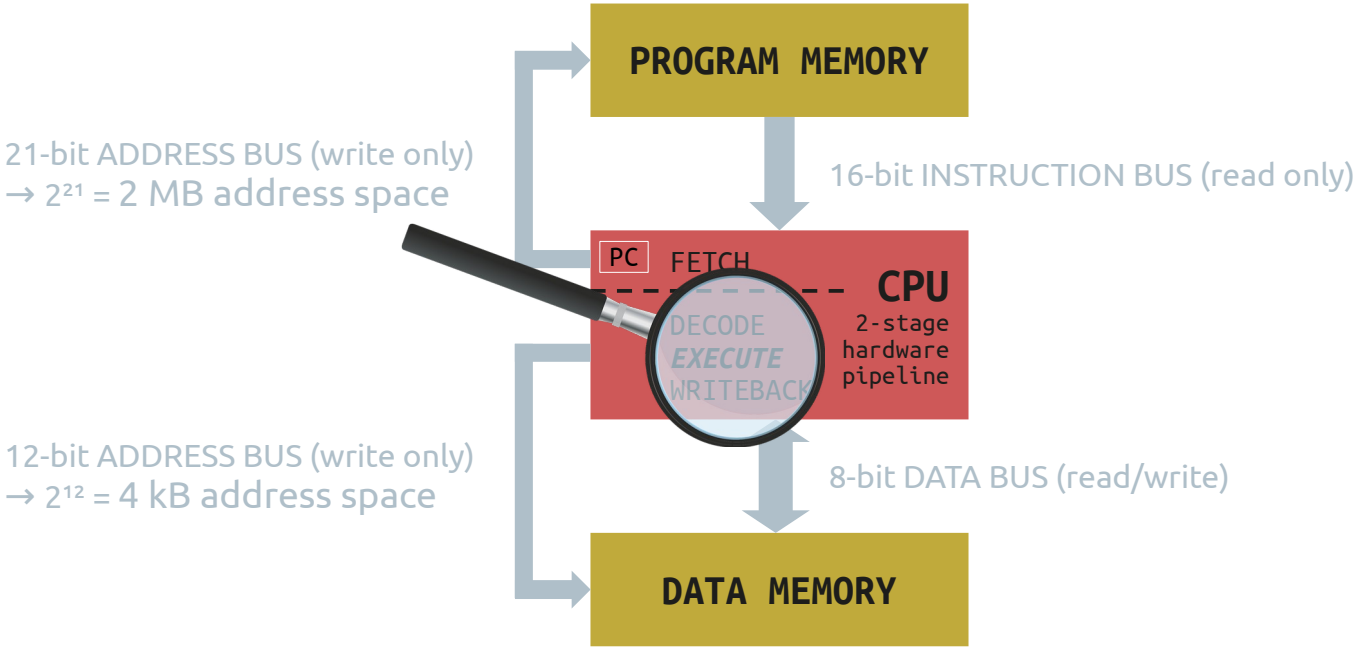
Except in sleep mode, the CPU executes a constant flow of instructions coming out of the main memory.

Some instructions ask the CPU to load or store a data from or to the data memory.



Let's dive into the EXECUTION stage to see the PIC18 Execution Units (EUs).

As the PIC18 is a 8-bit MCU, the EUs can only operate on 8-bit integer values.



Execution Unit: ALU

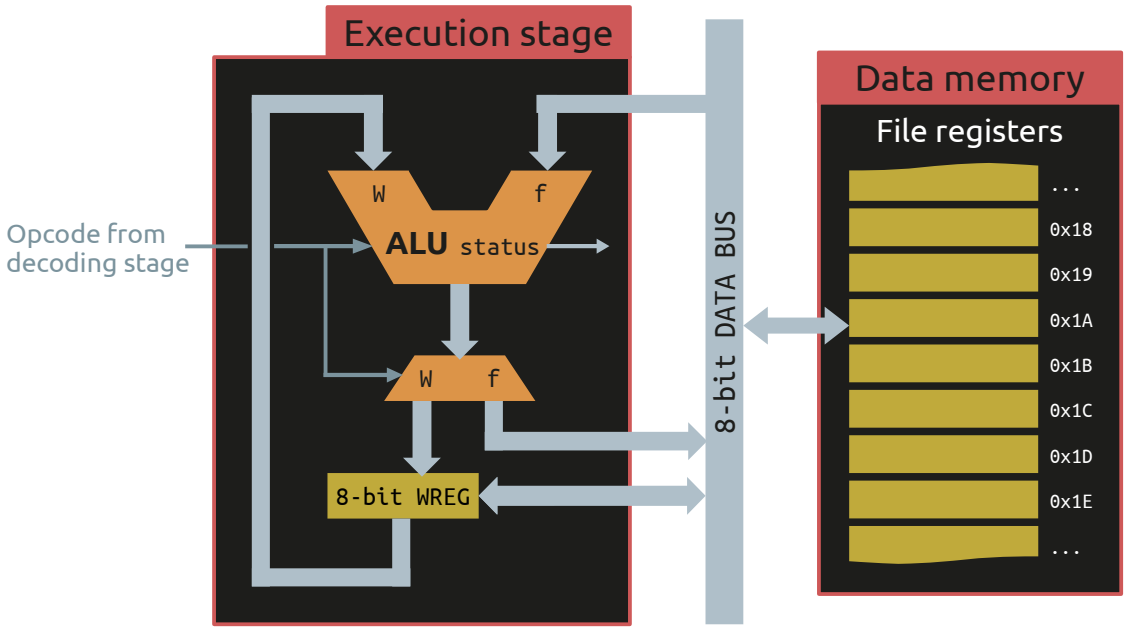
The **Arithmetic and Logic Unit (ALU)** is an execution unit in charge of arithmetic (+, -) and logic (&, |, ^, !, …) operations on 8-bit integer values.

Arithmetic operations

ADDWF
INCF
SUBWF
DECF
...

Logic operations

ANDWF
IORWF
XORWF
CLRF
SETF
...



Execution Unit: ALU

Any arithmetic or logic operation use by default a first 8-bit operand stored in the working register WREG and a second operand in the file register (data memory).

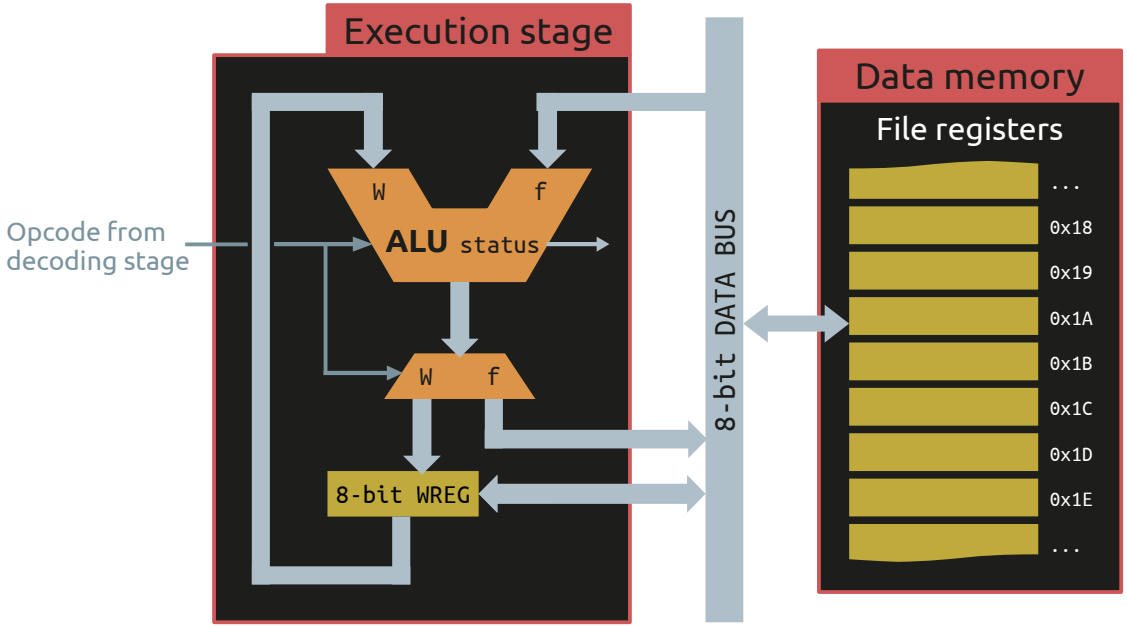
Example of an assembly language instruction and equivalent operation code (opcode).

PIC18 assembly language:

```
ADDWF f, d, a
```

16-bit opcode:

```
0010 01da ffff ffff
```



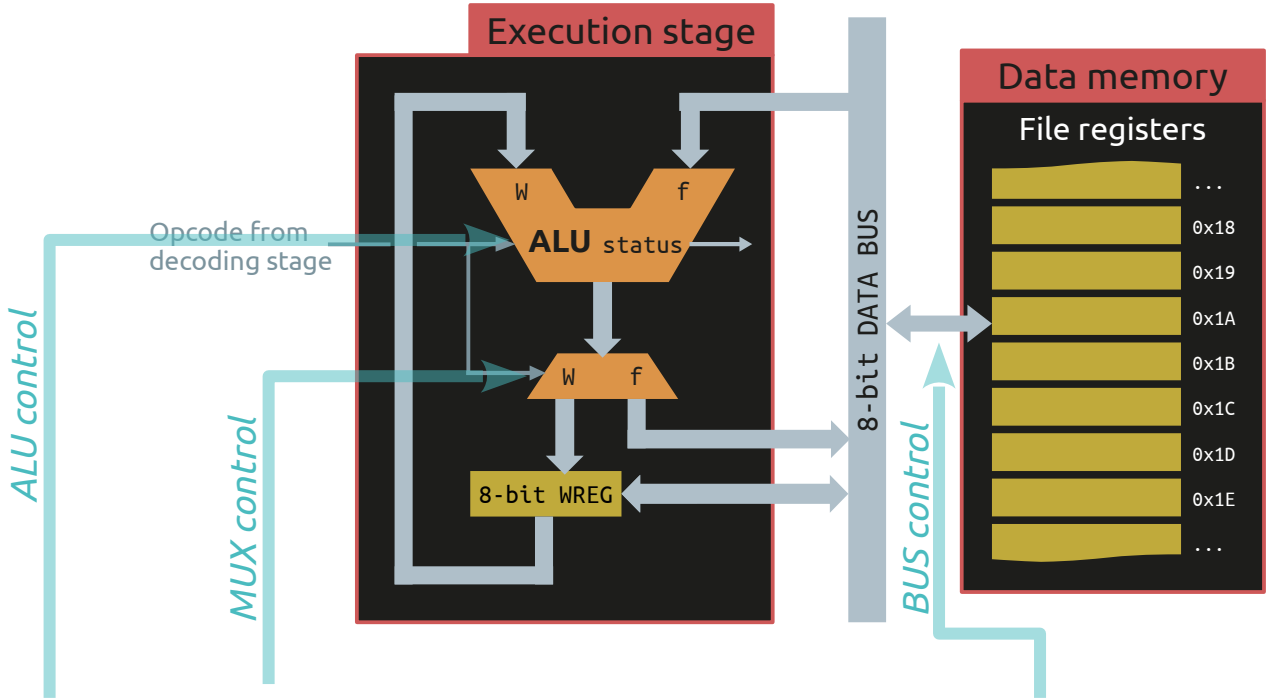
Execution Unit and Decoding Unit

PIC18 assembly language:

```
ADDWF f, d, a
```

16-bit opcode:

```
0010 01da ffff ffff
```



16-bit opcode: `0010 01` `d` `a` `ffff ffff`

Opcode
Unique for each instruction

Destination
`d=0` → Work register
`d=1` → file register

Access bank
`a=0` → Access bank
`a=1` → All banks, BSR

Source/Dest. address
8-bit,
Relative to a bank

Execution Unit: 8-bit x 8-bit integer multiplication

Numeration

```
uint8 * uint8 = uint16
 int8 *  int8 =  int15
```

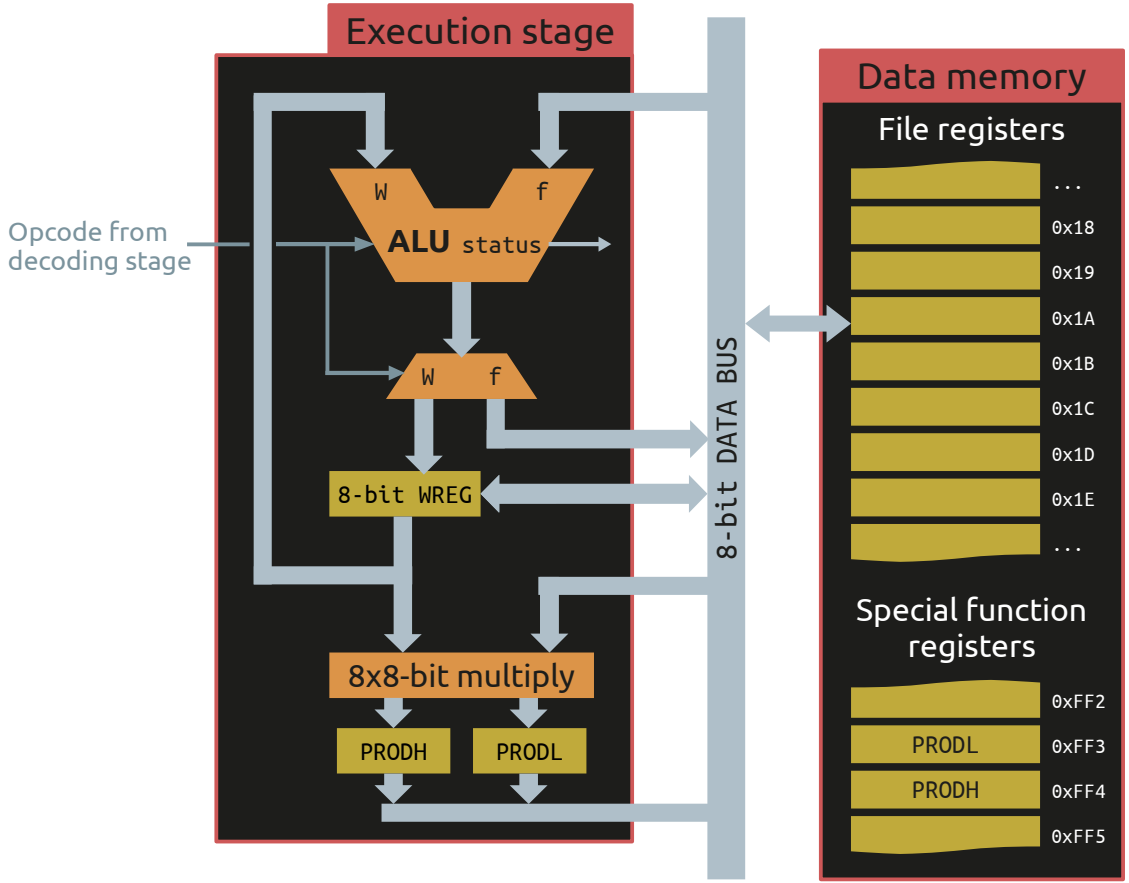
PIC18 multiply operations

MULWF (W-reg to F-reg)
MULLW (Litteral to W-Reg)

C and asm example

```
static short foo;
foo = 3*7;

…
MOVLW          3
MULLW          7
MOVFF          PRODL, <foo_L_12bit_address>
MOVFF          PRODH, <foo_H_12bit_address>

       or
MOVFF          0xFF3, <foo_L_12bit_address>
MOVFF          0xFF4, <foo_H_12bit_address>
```

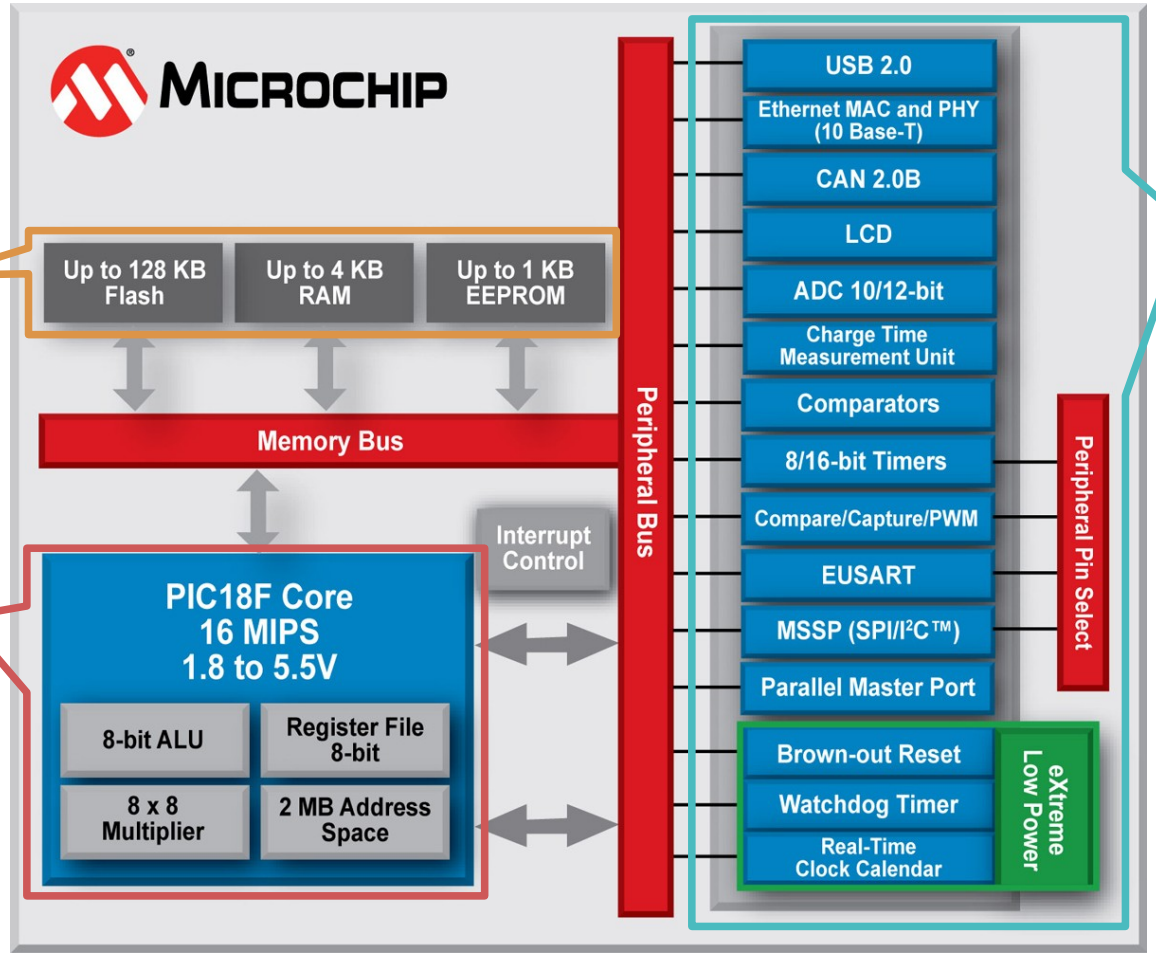
PRODL is an alias for 0xFF3, declared in a header
PRODH is an alias for 0xFF4, declared in a header



PIC18 architecture

Program & data memories

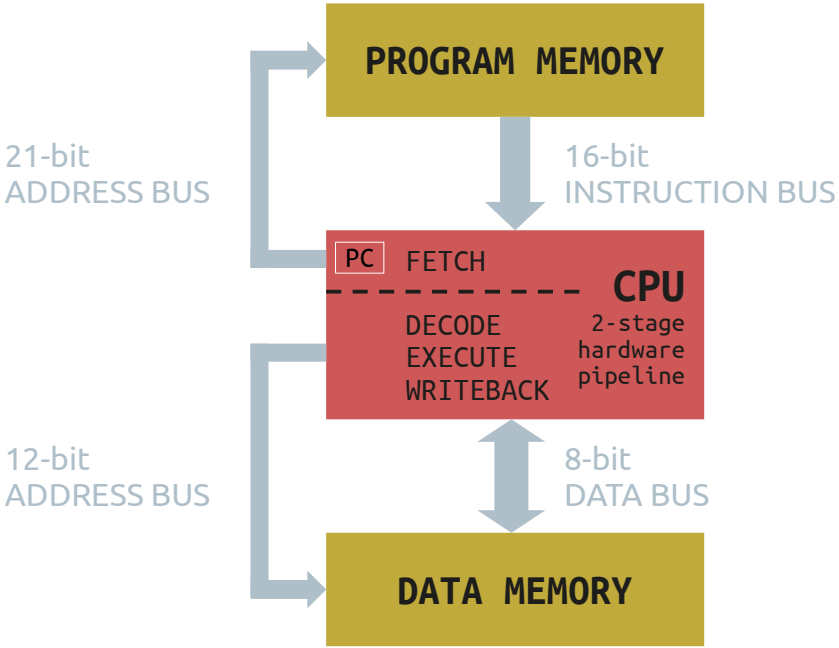
Central Processing Unit

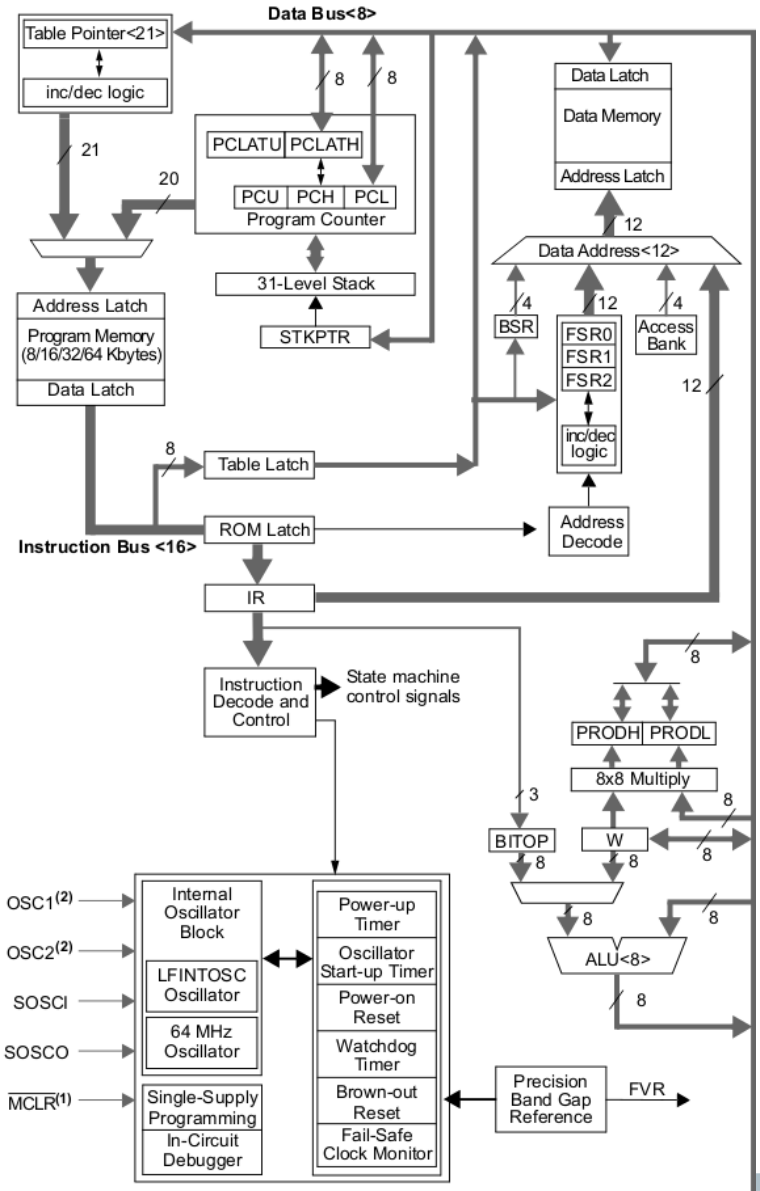


Peripherals

PIC18 CENTRAL PROCESSING UNIT

PIC18F27/47K40 CPU architecture





Find following items in this schematic

Flash memory

RAM memory

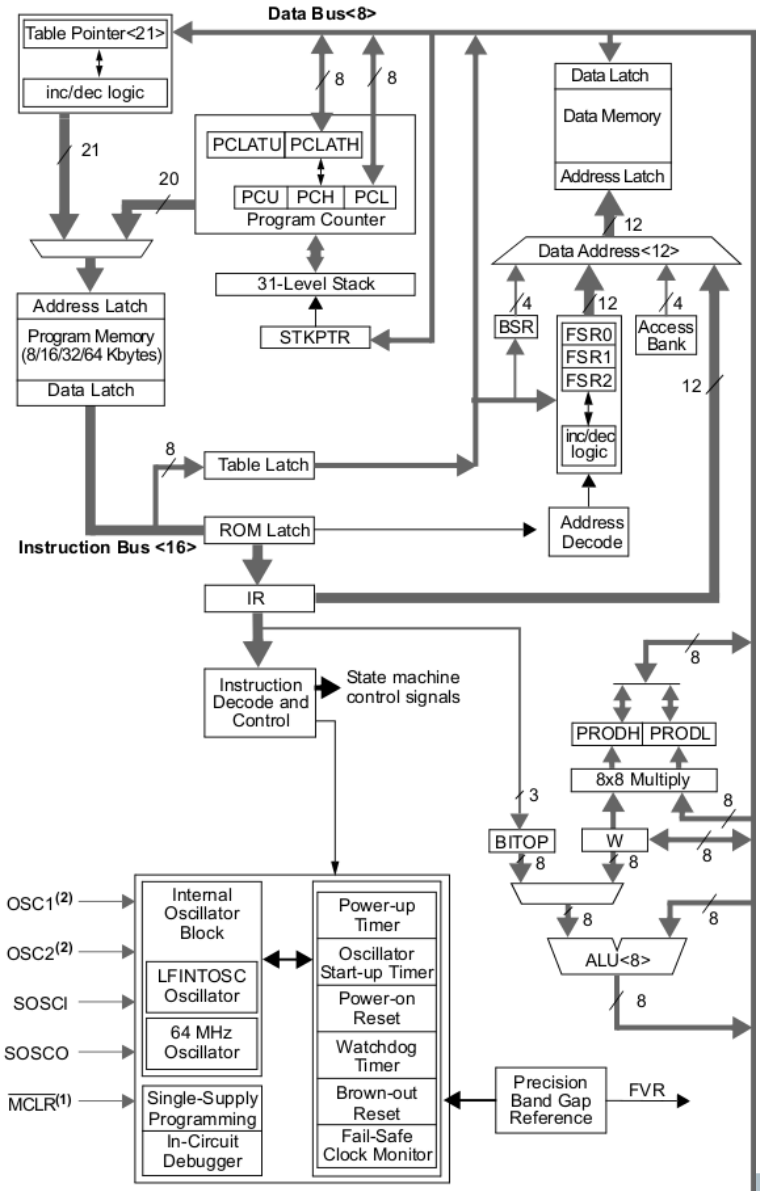
Buses

- program memory address bus
- data memory address bus
- data bus

Hardware pipeline stages

- Fetch
- Decode
- Execute (ALU, multiplier)
- Writeback

Program Counter register



Now that you know how the PIC18 CPU is made, you shall adapt your programming habits.

This CPU (like most of low-power MCUs) does not have any Floating-Point Unit. Therefore you should avoid using floats and doubles, and use integers instead.

Also as this MCU uses an 8-bit CPU you should use 8-bit integers (char C-type) as much as possible. They are usually large enough for control applications.

Finally you saw that the ALU performs simple operations. You should then avoid using advanced operators such as '/', '%', …

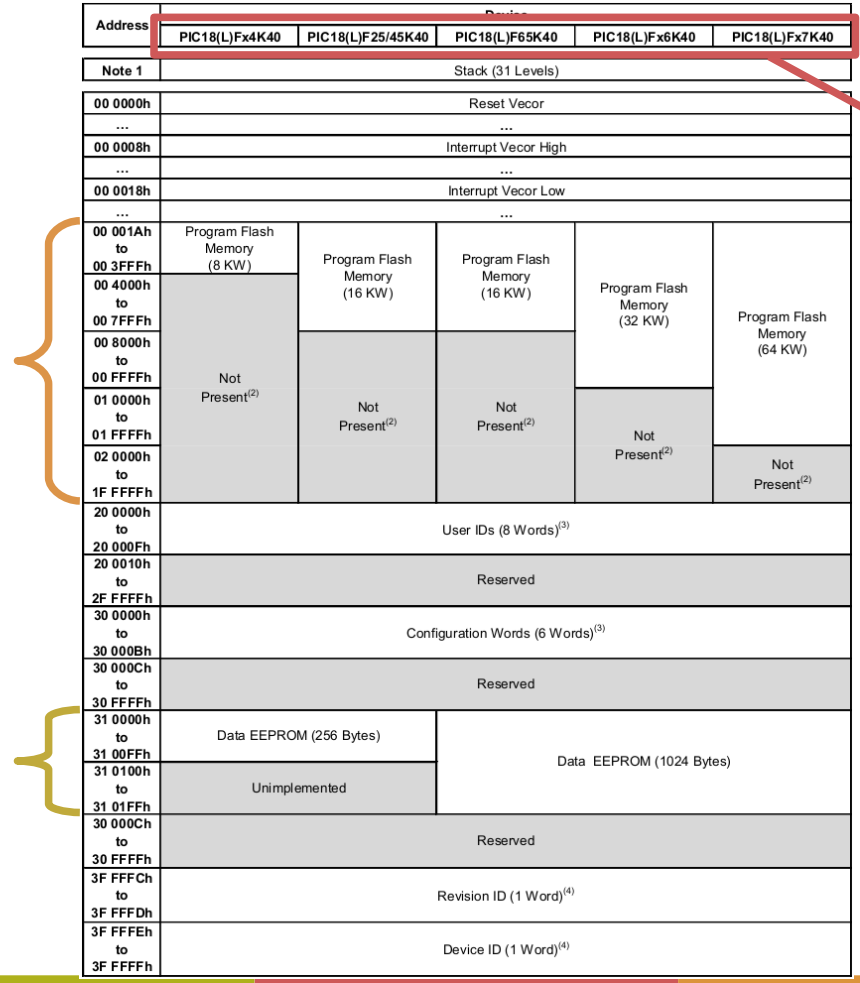
C-Type	custom typedef	Memory space	Values
char	int8	8 bits / 1 byte	-128 / 127
unsigned char	uint8	8 bits / 1 byte	0 / 255
short	int16	16 bits / 2 bytes	-32768 / +32767
unsigned short	uint16	16 bits / 2 bytes	0 / +65535
long	int32	32 bits / 4 bytes	-2G / +2G
unsigned long	uint32	32 bits / 4 bytes	0 / +4G
long long	int64	64 bits / 8 bytes	-9E / +9E
unsigned long long	uint64	64 bits / 8 bytes	0 / +18E
int		processor dependant	processor dependant
unsigned int		processor dependant	processor dependant
float		32 bits / 4 bytes (PIC/XC8)	
double		32 bits / 4 bytes (PIC/XC8)	

MEMORY

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Program and data memory map



Address	Device				
	PIC18(L)Fx4K40	PIC18(L)F25/45K40	PIC18(L)F65K40	PIC18(L)Fx6K40	PIC18(L)Fx7K40
Note 1	Stack (31 Levels)				
00 0000h	Reset Vecor				
...	...				
00 0008h	Interrupt Vecor High				
...	...				
00 0018h	Interrupt Vecor Low				
...	...				
00 001Ah to 00 3FFFh	Program Flash Memory (8 KW)	Program Flash Memory (16 KW)	Program Flash Memory (16 KW)	Program Flash Memory (32 KW)	Program Flash Memory (64 KW)
00 4000h to 00 7FFFh	Not Present[2]				
00 8000h to 00 FFFFh		Not Present[2]	Not Present[2]		
01 0000h to 01 FFFFh				Not Present[2]	
02 0000h to 1F FFFFh					Not Present[2]
20 0000h to 20 000Fh	User IDs (8 Words)[3]				
20 0010h to 2F FFFFh	Reserved				
30 0000h to 30 000Bh	Configuration Words (6 Words)[3]				
30 000Ch to 30 FFFFh	Reserved				
31 0000h to 31 00FFh	Data EEPROM (256 Bytes)		Data EEPROM (1024 Bytes)		
31 0100h to 31 01FFh	Unimplemented				
30 000Ch to 30 FFFFh	Reserved				
3F FFFCh to 3F FFFDh	Revision ID (1 Word)[4]				
3F FFFEh to 3F FFFFh	Device ID (1 Word)[4]				

Program memory

Data memory

Size of program and data memories depends on the device

PIC18F4550 program memory map

Program Counter (Fetch stage)

Hardware stack, max 31 nested function calls (saving return addresses)

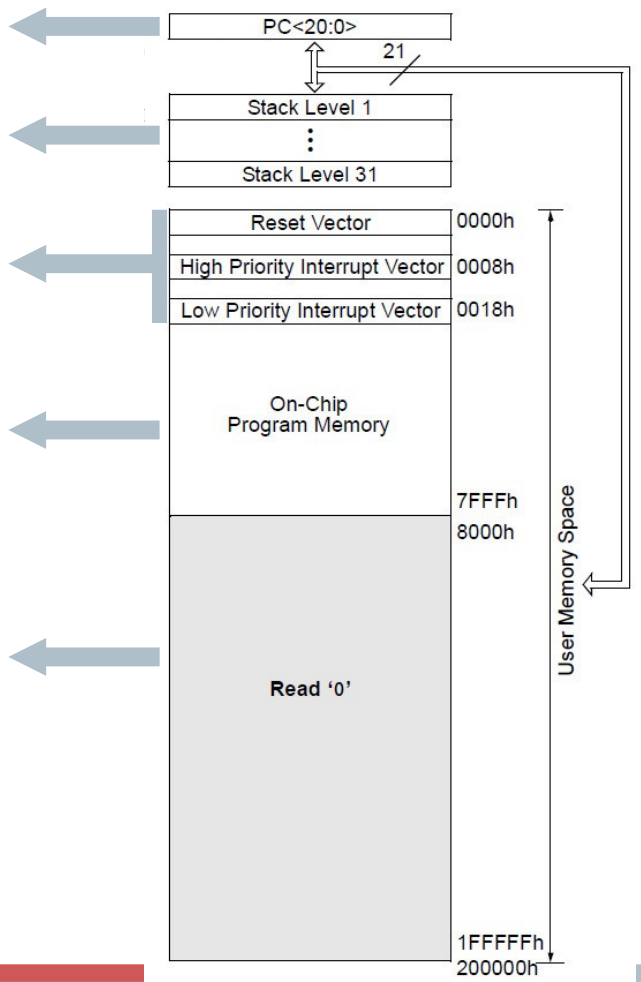
Relocatable table of interrupt vectors (redirections from IRQ to ISR)

Memory space for the embedded firmware (32 kB for the PIC18F4550 shown here)

Empty space depending on the device

Some PIC18 have up to 128 kB of program memory

Remaining space might be used for future evolutions



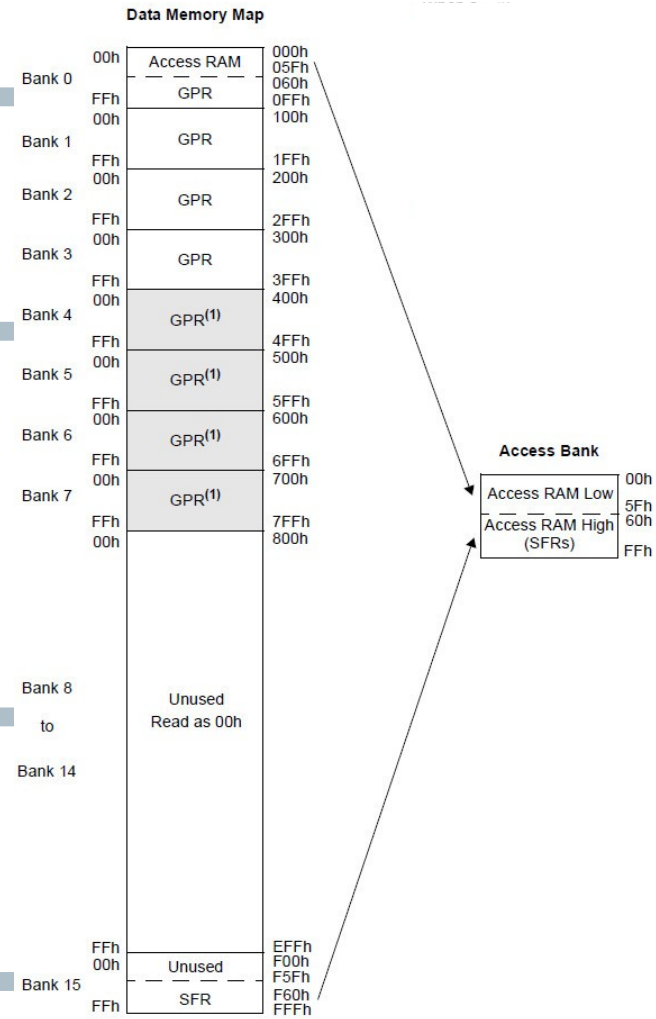
PIC18F4550 data memory map

Data memory segmented in 16 banks of 256 bytes
(see next page for selecting the working bank)

GPR (General Purpose Register file)
Registers that can be used to store any data,
available for use by all instructions.

2 kB data memory for this PIC18F4550
Up to 4kB for some other PIC18

SFR (Special Function Registers),
linked to CPU and peripherals registers



Data memory: Bank selection

The data memory is segmented in 16 banks of 256 bytes. This is a typical construction on 8-bit architectures.

The working bank is selected by configuring four bits in the BSR (Bank Select Register).

Then instructions that use a file-register operand will access to the data with 8-bit direct addressing mode.

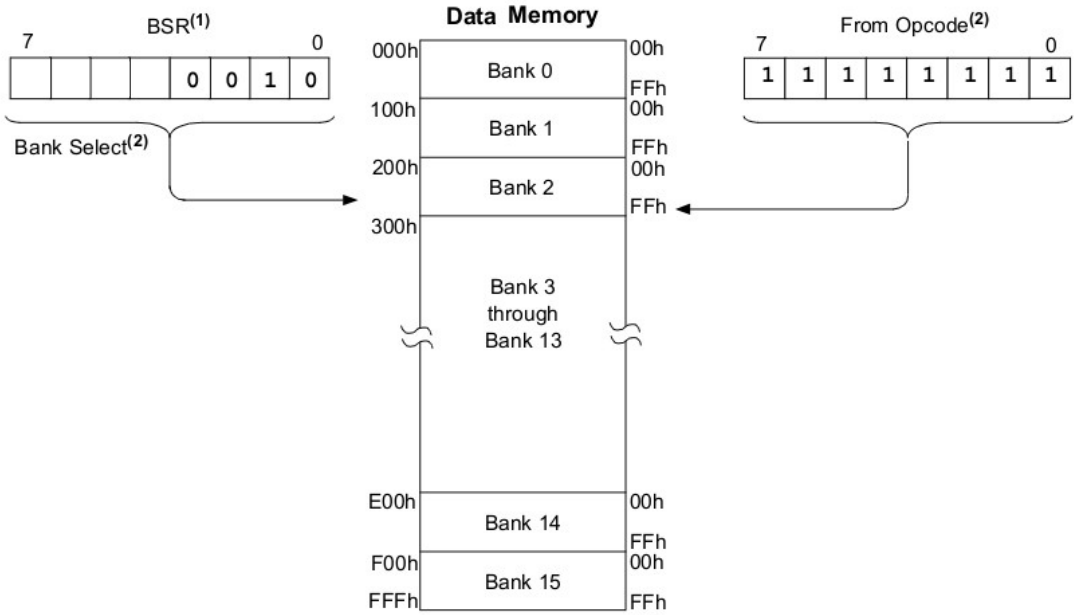
Bank select

PIC18 C language:

```
BSR = 0x02;
```

PIC18 assembly language:

```
MOVLW 0x02
MOVWF BSR
```



From the 16-bit opcode

PIC18 assembly language:

```
ADDWF f, d, a
```

16-bit opcode:

```
0010 01da ffff ffff
```

Data memory: Special Function Registers (SFR)

The SFR bank is a part of the RAM data memory where CPU and peripheral registers are mapped.

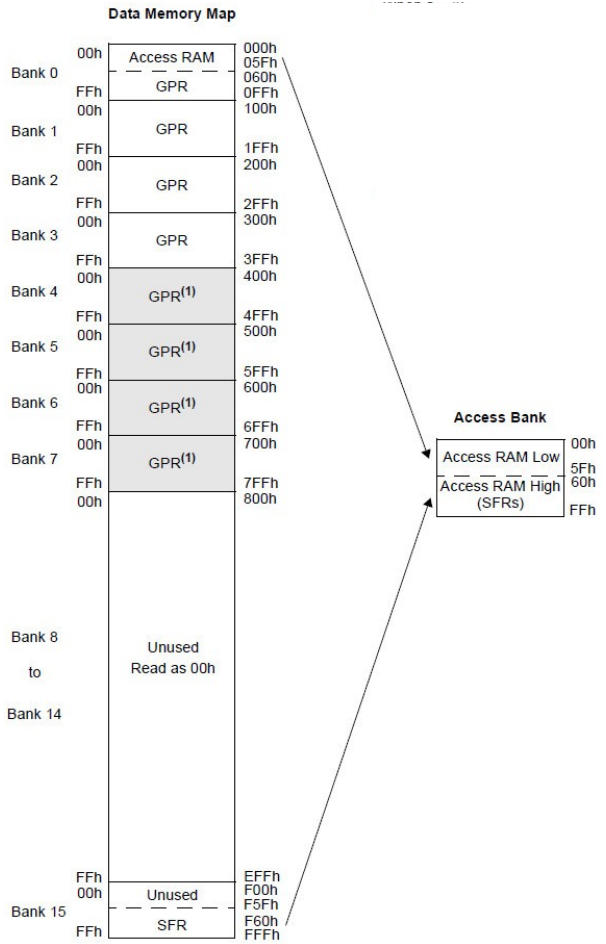
It means registers are physically outside of the memory but they are accessible with a memory address.

Core memory mapped registers

Peripheral specialised functions memory mapped registers

Address	Name	Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2(1)	FBFh	CCPR1H	F9Fh	IPR1	F7Fh	UEP15
FFEh	TOSH	FDEh	POSTINC2(1)	FBEh	CCPR1L	F9Eh	PIR1	F7Eh	UEP14
FFDh	TOSL	FDDh	POSTDEC2(1)	FBDh	CCP1CON	F9Dh	PIE1	F7Dh	UEP13
FFCh	STKPTR	FDCh	PREINC2(1)	FBCh	CCPR2H	F9Ch	—(2)	F7Ch	UEP12
FFBh	PCLATU	FDBh	PLUSW2(1)	FBBh	CCPR2L	F9Bh	OSCTUNE	F7Bh	UEP11
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—(2)	F7Ah	UEP10
FF9h	PCL	FD9h	FSR2L	FB9h	—(2)	F99h	—(2)	F79h	UEP9
FF8h	TBLPTRU	FD8h	STATUS	FB8h	BAUDCON	F98h	—(2)	F78h	UEP8
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	ECCP1DEL	F97h	—(2)	F77h	UEP7
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCP1AS	F96h	TRISE(3)	F76h	UEP6
FF5h	TABLAT	FD5h	T0CON	FB5h	CVRCON	F95h	TRISD(3)	F75h	UEP5
FF4h	PRODH	FD4h	—(2)	FB4h	CMCON	F94h	TRISC	F74h	UEP4
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB	F73h	UEP3
FF2h	INTCON	FD2h	HLVDCON	FB2h	TMR3L	F92h	TRISA	F72h	UEP2
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—(2)	F71h	UEP1
FF0h	INTCON3	FD0h	RCON	FB0h	SPBRGH	F90h	—(2)	F70h	UEP0
FEFh	INDF0(1)	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—(2)	F6Fh	UCFG
FEEh	POSTINC0(1)	FCEh	TMR1L	FAEh	RCREG	F8Eh	—(2)	F6Eh	UADDR
FEDh	POSTDEC0(1)	FCDh	T1CON	FADh	TXREG	F8Dh	LATE(3)	F6Dh	UCON
FECh	PREINC0(1)	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD(3)	F6Ch	USTAT
FEBh	PLUSW0(1)	FCBh	PR2	FABh	RCSTA	F8Bh	LATC	F6Bh	UEIE
FEAh	FSR0H	FCAh	T2CON	FAAh	—(2)	F8Ah	LATB	F6Ah	UEIR
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA	F69h	UIE
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—(2)	F68h	UIR
FE7h	INDF1(1)	FC7h	SSPSTAT	FA7h	EECON2(1)	F87h	—(2)	F67h	UFRMH
FE6h	POSTINC1(1)	FC6h	SSPCON1	FA6h	EECON1	F86h	—(2)	F66h	UFRML
FE5h	POSTDEC1(1)	FC5h	SSPCON2	FA5h	—(2)	F85h	—(2)	F65h	SPPCON(3)
FE4h	PREINC1(1)	FC4h	ADRESH	FA4h	—(2)	F84h	PORTE	F64h	SPPEPS(3)
FE3h	PLUSW1(1)	FC3h	ADRESL	FA3h	—(2)	F83h	PORTD(3)	F63h	SPPCFG(3)
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC	F62h	SPPDATA(3)
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB	F61h	—(2)
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA	F60h	—(2)

Data memory: Access bank



When executing an operation that uses direct addressing, the execution stage checks the <a> field (access bank):

```
ADDWF f, d, a
```

<a> = ‘0’ or ‘1’

If <a> = ‘0’ the CPU only sees the **access bank (top half of the bank #0 + SFR bank)**. The 4 most significant bits of the 12-bit data memory address come from the access bank register (0x0 or 0xF) → **Fast solution**.

If <a> = ‘1’ the CPU can access to all the data memory. It uses the value of the **BSR (Bank Select Register)** to generate the 4 most significant bits of the 12-bit data memory address.

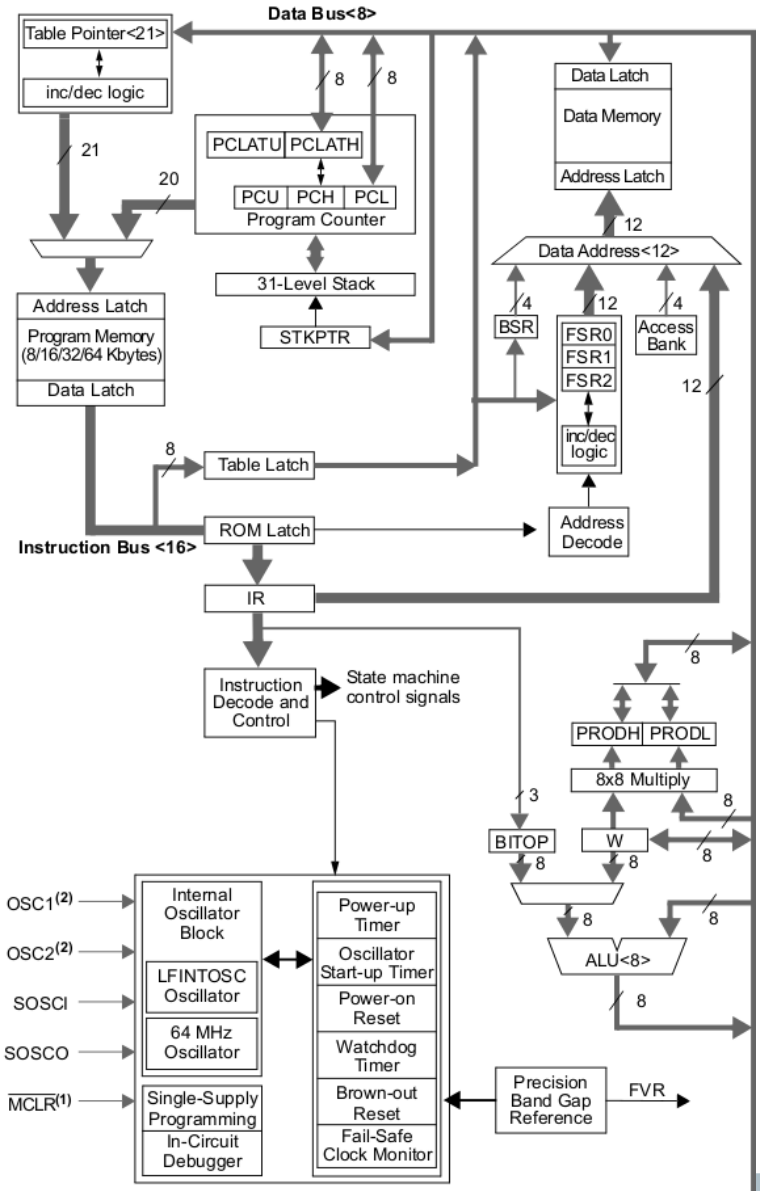
Find following items in this schematic

Data memory

BSR (Bank Select Register)

Access bank register

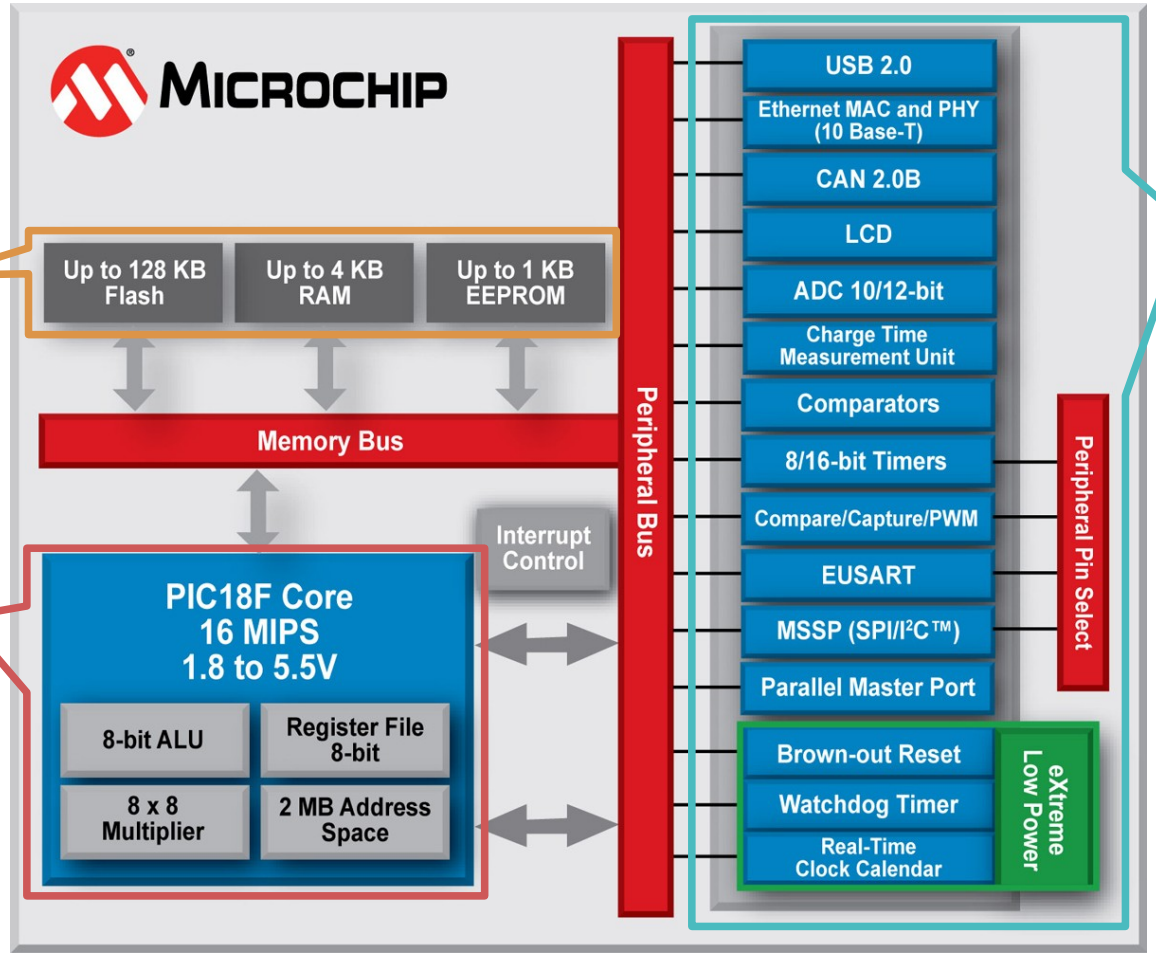
Note how they are placed relatively to each other and how the 12-bit data memory address is built.



PIC18 architecture

Program & data memories

Central Processing Unit



Peripherals

PERIPHERALS

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

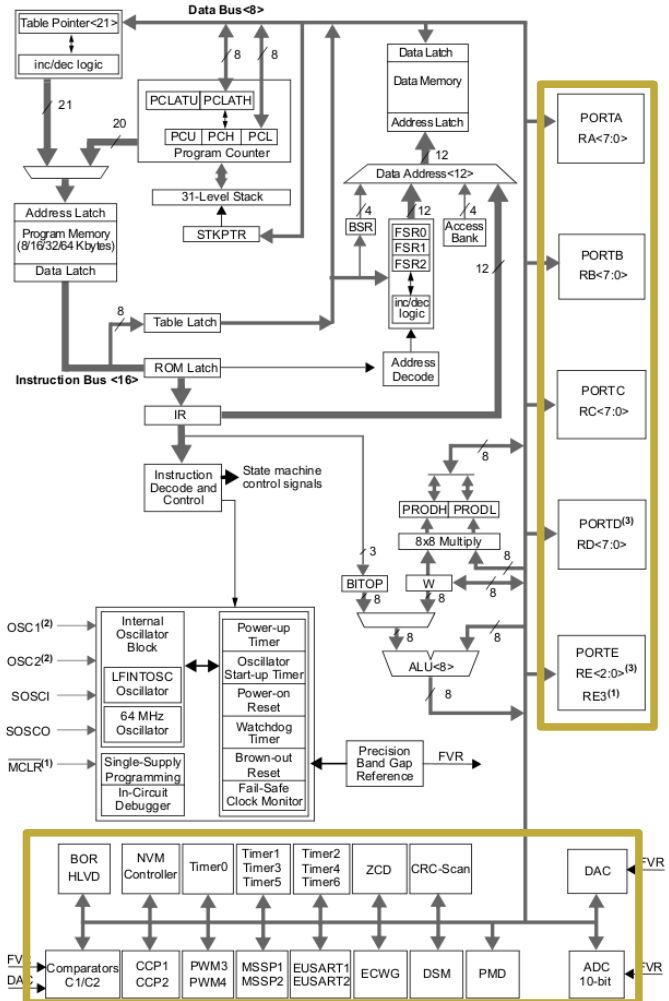
Definition

A peripheral is a hardware function that can be used to perform specific calculus and/or process some operation while letting the CPU performing something else.

For all MCUs, peripherals are physically connected to the data bus and their internal registers are mapped to the data memory.

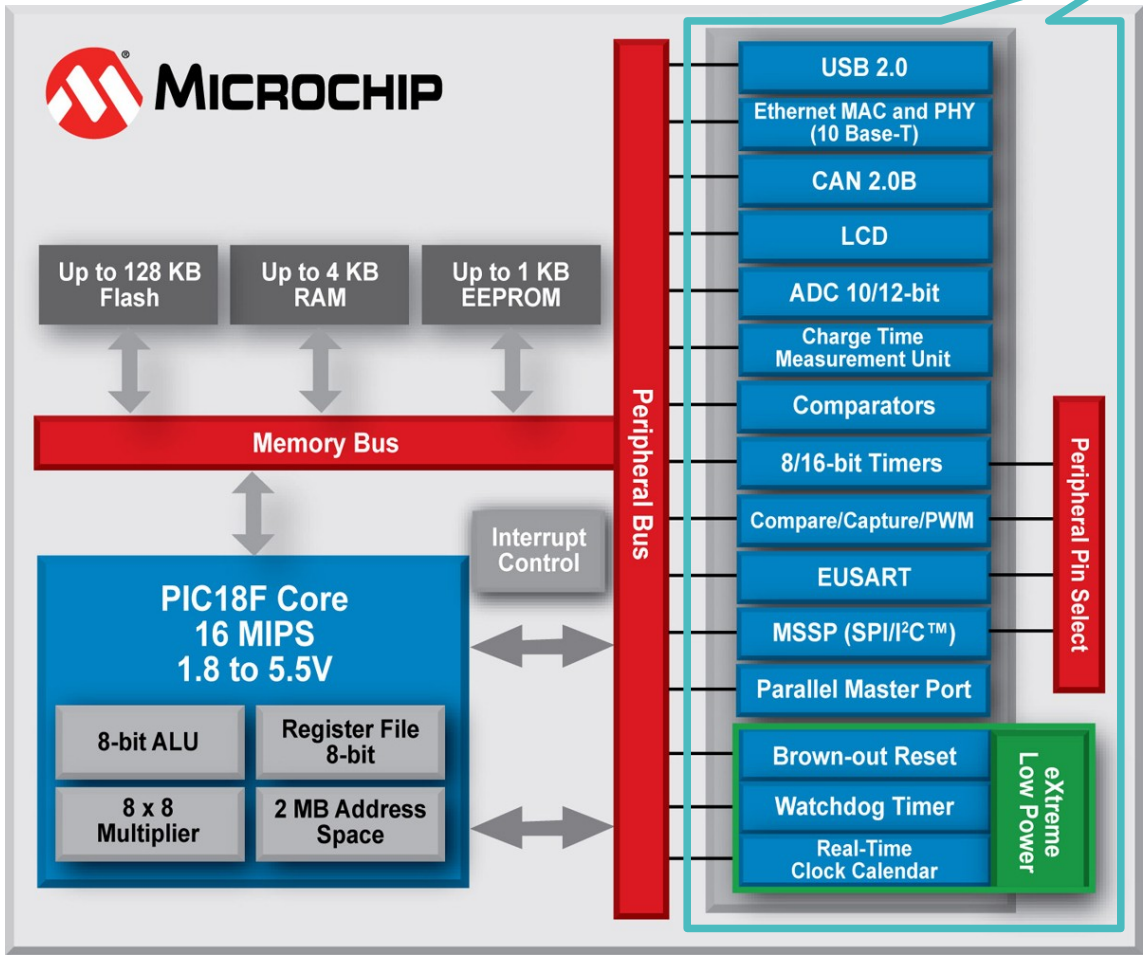
From the program point of view, accessing to a register (read or write) is the same as accessing to a data memory cell.

Example of PIC18F27/47K40 CPU and peripherals.



Hardware functions

Peripherals,
guess what they do!

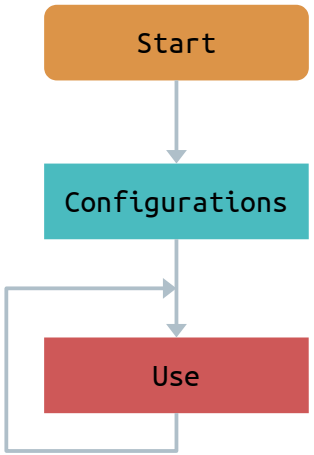


This lecture does not contain any material for using each peripheral. To know what is the role of a peripheral and how to configure it, the best material is the device datasheet.

When the processor starts (after power-on or reset), no peripheral function is configured nor activated.

The programmer must explicitly configure and activate hardware services that are needed for the application. Only then the peripherals can be used.

Most of peripherals have configuration registers that must be set once in addition to working registers that contains updated values.



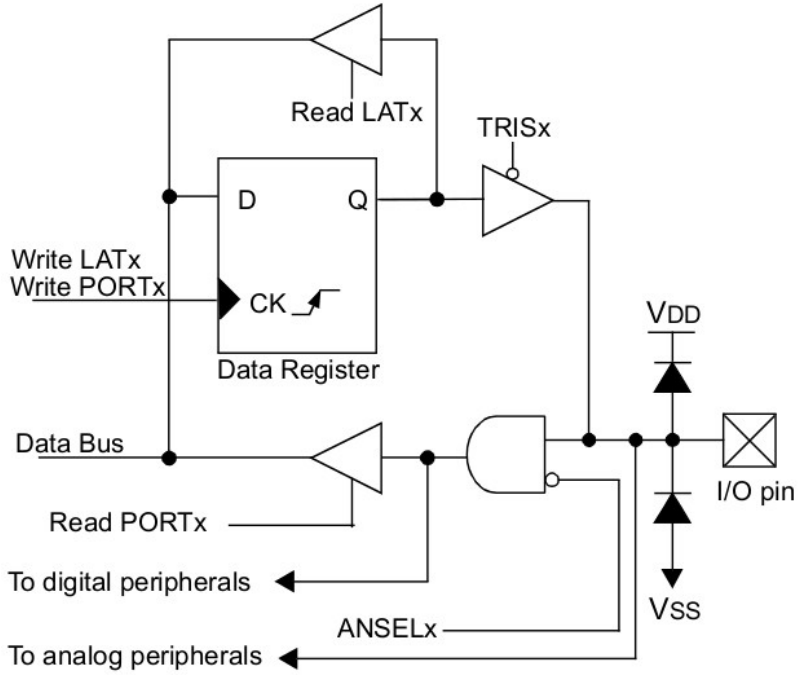

```
void main()
{

      // Peripherals configurations
      // and other initialisation
      …


      while(1) {
            // Main routine and peripherals use
            …
      }
}
```

A port is a group of 8 pins or GPIOs (General Purpose Input/Output).

They can be used as independent digital inputs and/or outputs. The PIC18F27K40 has 5 ports (port A to port E), which makes 40 independent GPIOs available.

Each port has eight registers to control the operation. These registers are:

- PORTx registers (reads the levels on the pins of the device)
- LATx registers (output latch)
- TRISx registers (data direction)
- ANSELx registers (analog select)
- WPUx registers (weak pull-up)
- INLVLx (input level control)
- SLRCONx registers (slew rate control)
- ODCONx registers (open-drain control)



Example of **GPIO (General Purpose Input/Output)** on the Curiosity HPC board.

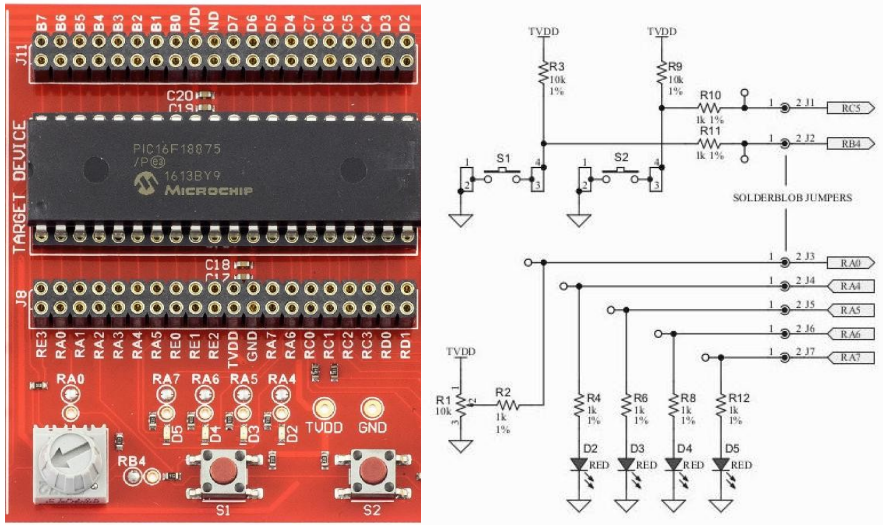
First the RA4 pin is configured as an output, then the output value is set to ‘high’. This will turn on the LED D2 on this pin.

PIC18 C program

```
//Set RA4 as an output
TRISA = 0xEF;

//Set RA4 to high level
LATD = 0x10;
```

PIC18 assembly program

```
;Set RA4 as an output
MOVLW     0xEF
MOVWF     TRISA

;Set RA4 to high level
MOVLW     0x10
MOVWF     LATA
```

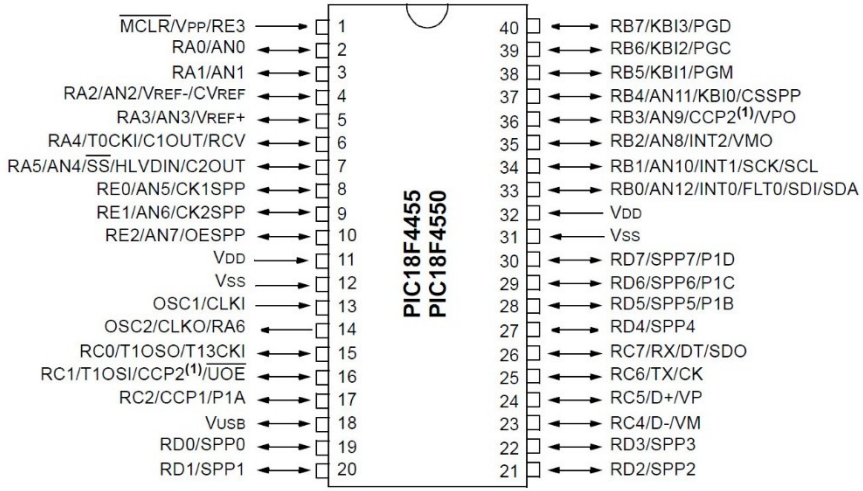


RA4 *= pin 4 of port A (also bit 4 of registers associated to port A).*
TRISx *= register that contains the direction of the 8 pins of the port x (‘0’ = Output, ‘1’ = Input).*
LATx *= register that contains the output value for pins configured as outputs in port x.*

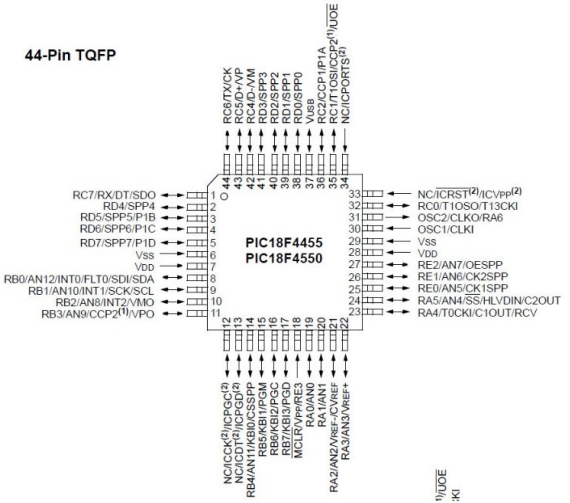
MCUs usually have more peripherals than they can really handle.

In fact, many peripherals are connected to the same pins. This implies that some peripherals cannot be used at the very same time.

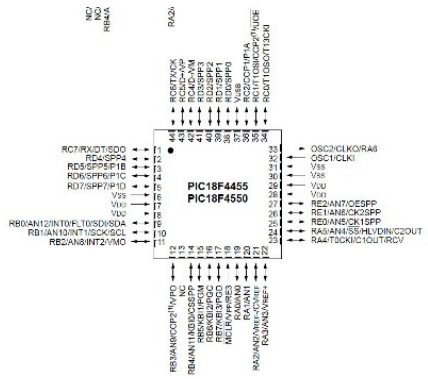
DIP package



TQFP package

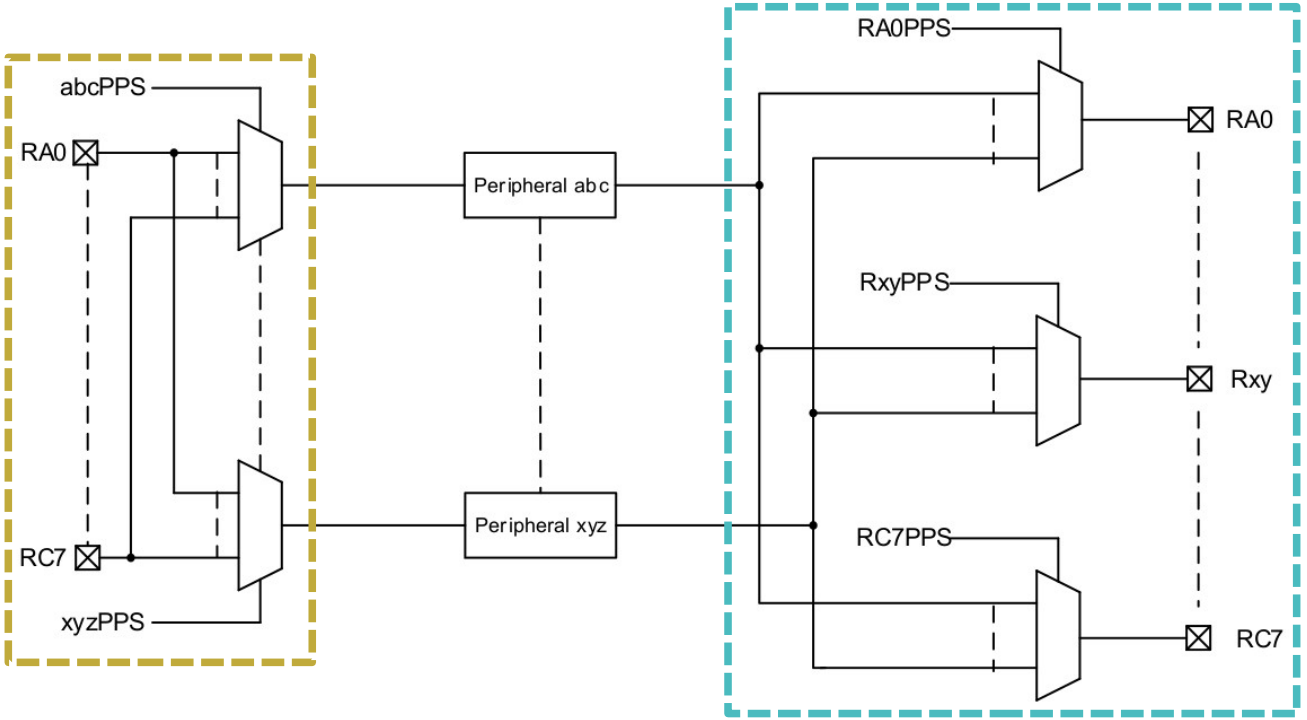


QFN package



As many input and output pins can be used by several peripherals, connections between pins and peripherals must be set using the Peripheral Pin Select (PPS) module.

Input pin redirections



Peripheral outputs redirections

Reference clock module

Like all other MCUs, the PIC18 family offers configuration and system hardening services.

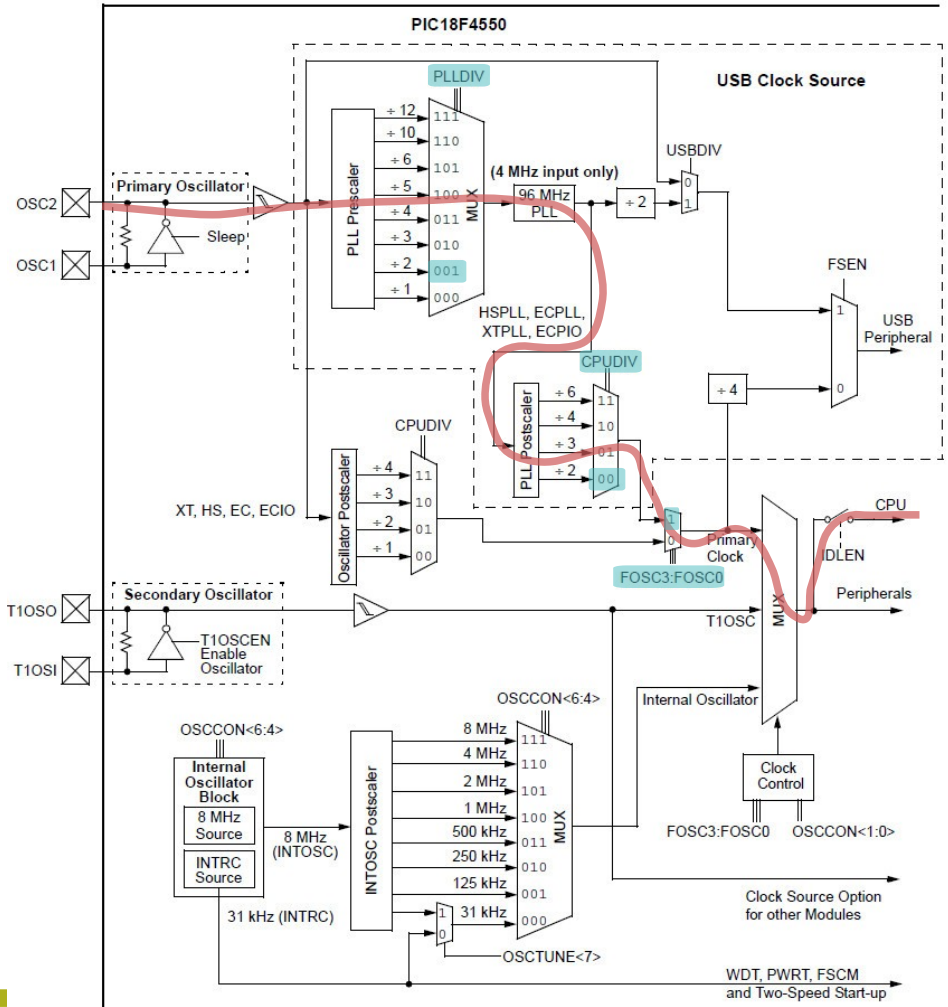
“Hardening” an application consists in making it less sensitive to its environment (power supply fluctuations, …) or even handle unexpected situations (watchdog, …).

Any program on PIC18 must start with this configuration:

```
/* CPU specific features configuration */
#pragma config PLLDIV=2, CPUDIV=OSC1_PLL2, FOSC=HSPLL_HS
#pragma config BOR=OFF, WDT=OFF, MCLRE=ON, LVP = OFF
```

Reference clock module

External clock
(e.g. 8 MHz cristal)

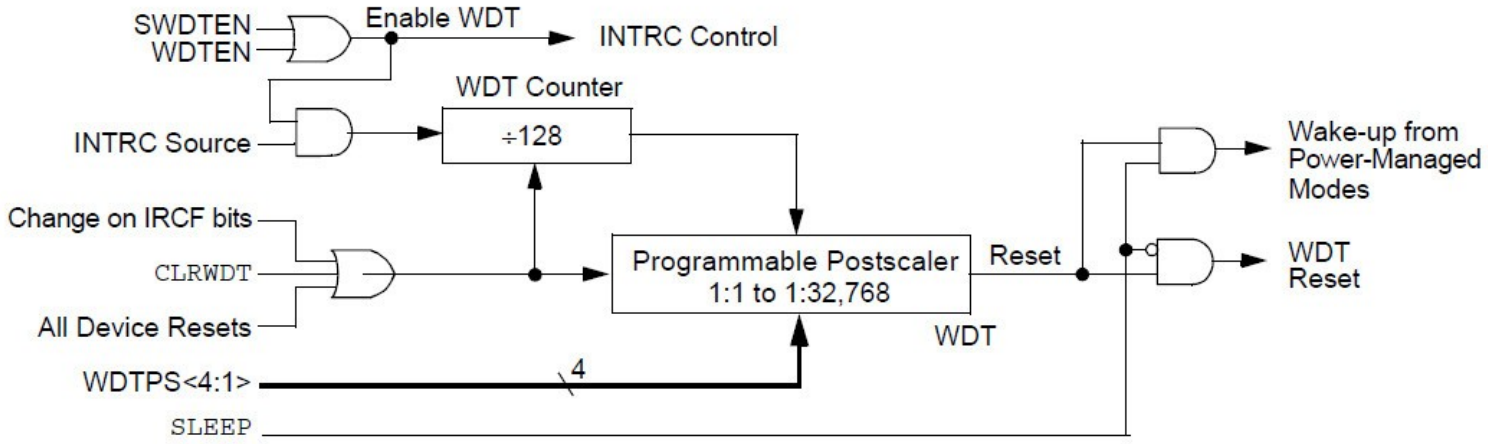



```
#pragma config
PLLDIV=2,
CPUDIV=OSC1_PLL2,
FOSC=HSPLL_HS
```

The watchdog is an application agent.

In normal circumstances, the application resets the watchdog timer at regular intervals.

When the application remains stuck for a long time, the watchdog timer is not reset and will eventually overflow. The watchdog will force the application to reboot by resetting the CPU.



INTERRUPTS

Event

Interrupt Flag (IF)

Interrupt Request (IRQ)

Interrupt Service Routine (ISR)

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Interrupt flag

Once a peripheral has been configured, it becomes autonomous.

When its job is done (counting, converting, …) or **when a special event occurs** (message received, switch pressed, …), **the peripheral will raise a Interrupt Flag (IF)**.

A interrupt flag is a bit in a register, each flag corresponding to a precise event.

Peripheral Interrupt Request (Flag) Register 0

Bit	7	6	5	4	3	2	1	0
			TMR0IF	IOCIF		INT2IF	INT1IF	INT0IF
Access			R/W	R		R/W	R/W	R/W
Reset			0	0		0	0	0

Bit 5 – TMR0IF Timer0 Interrupt Flag bit(1)

Value	Description
1	TMR0 register has overflowed (must be cleared by software)
0	TMR0 register has not overflowed

Bit 4 – IOCIF Interrupt-on-Change Flag bit(1,2)

Value	Description
1	IOC event has occurred (must be cleared by software)
0	IOC event has not occurred

Bits 0, 1, 2 – INTxIF External Interrupt 'x' Flag bit(1,3)

Value	Description
1	External Interrupt 'x' has occurred
0	External Interrupt 'x' has not occurred

Interrupt Service Routine (ISR)

However some events require immediate attention. Plus, using interrupt flags the same way as classical variables (e.g. testing them in "`if`" statements) is inefficient.

That is why an **interrupt mechanism** has been designed. This is the hardware way of stopping the normal execution flow of the CPU. It will then switch to the execution of a function dedicated to the event: the **ISR (Interrupt Service Routine)**.

```
// Doing the classical way

main() {

      // do something
      ...
      ...
      ...
      ...
      while(1){
            if( event_A )
            ...
            if( event_B )
            ...
            if( event_C )
      }
}
```

Event C occurs here

Event C is processed here

```
// Using interrupt mechanism

void event_A_ISR()
{ ... }

void event_B_ISR()
{ ... }

void event_C_ISR()
{ ... }

main() {
      // do something
      ...
      ...
      ...
      ...
}
```

Event C occurs here

Immediate processing

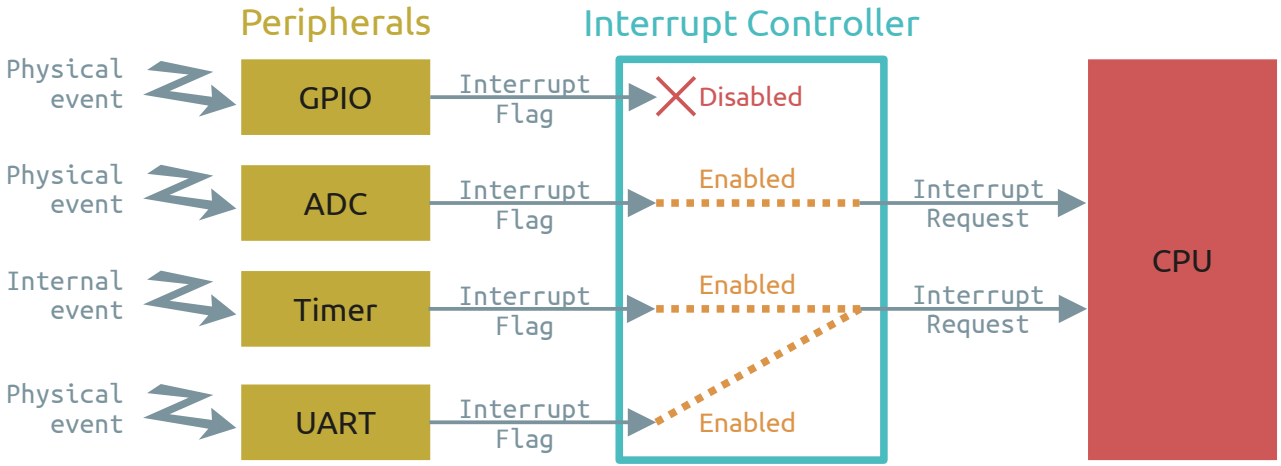
Main program execution is paused

Interrupt Request (IRQ)

All interrupts are disabled by default: it is the programmer work to decide and explicitly tell which event is worthy enough to lead to an interrupt.

To do so, the programmer must configure the **Interrupt controller**.

This circuit will turn selected interrupt flags into **Interrupt Requests (IRQ)**, that will cause the CPU to stop its current work.



Interrupt flag
Bit in a register that ***indicates*** that an event has occurred.

Interrupt request
Physical wire connected to the CPU that ***makes it switch to another function***.

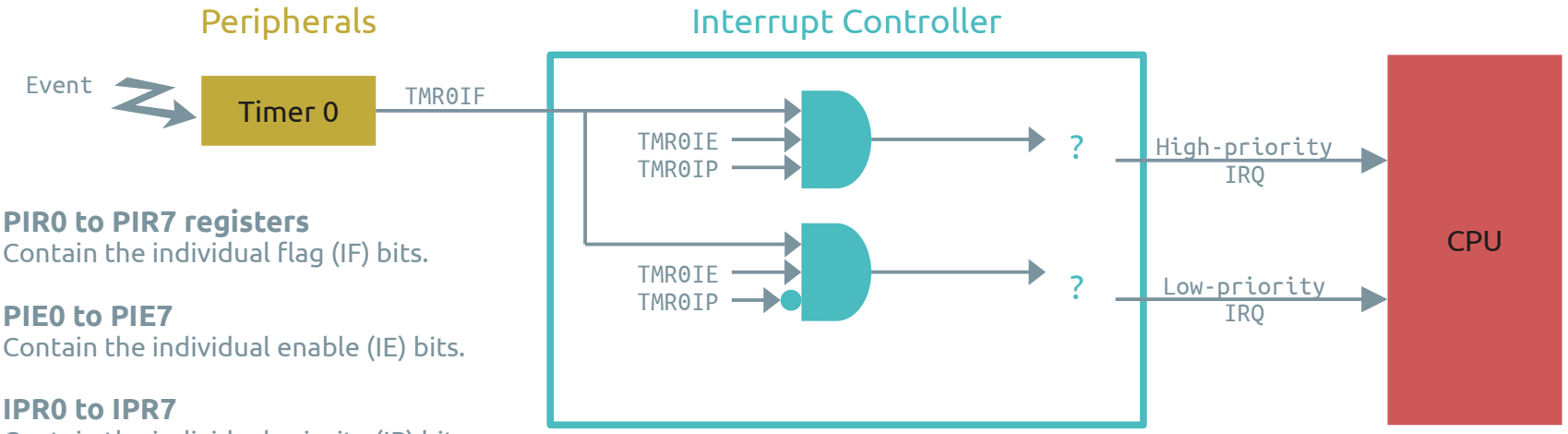
Notes
Many interrupt flags can be linked onto the same IRQ signal.
A specific IRQ corresponds to a specific ISR.

Interrupt controller

The **interrupt controller** consists of AND and OR gates. This circuitry independently let or prevent interrupt flags becoming interrupt requests.

To do so, one must configure the registers of the interrupt controller. Like the PIC18, most MCUs need to configure three bits for each interrupt:

the Interrupt Flag (IF) bit, the Interrupt Enable (IE) bit, the Interrupt Priority (IP) bit.



PIR0 to PIR7 registers
Contain the individual flag (IF) bits.

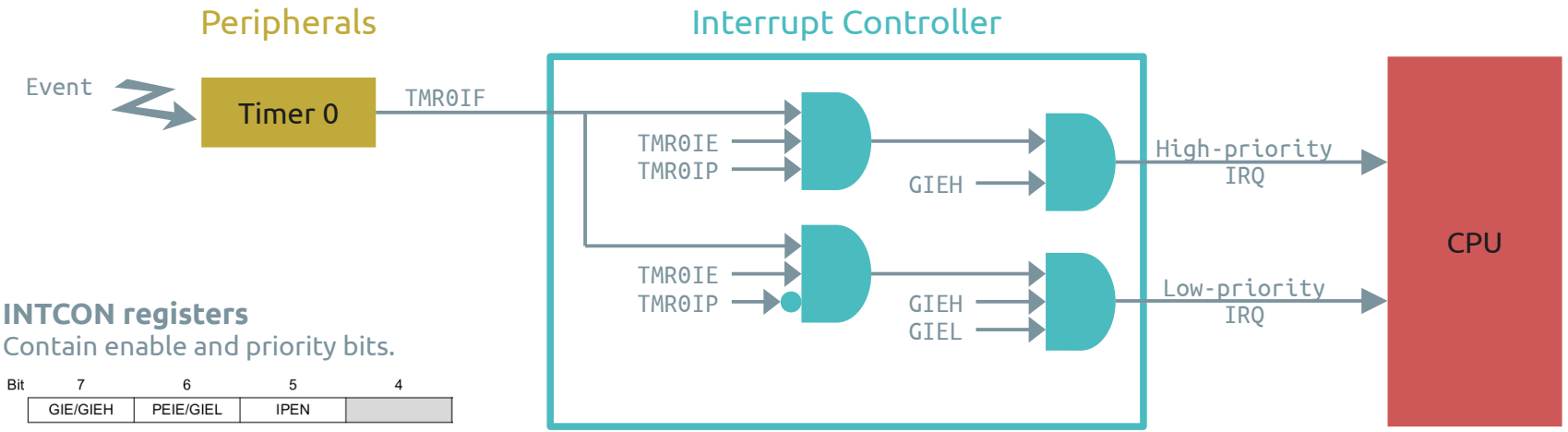
PIE0 to PIE7
Contain the individual enable (IE) bits.

IPR0 to IPR7
Contain the individual priority (IP) bits.

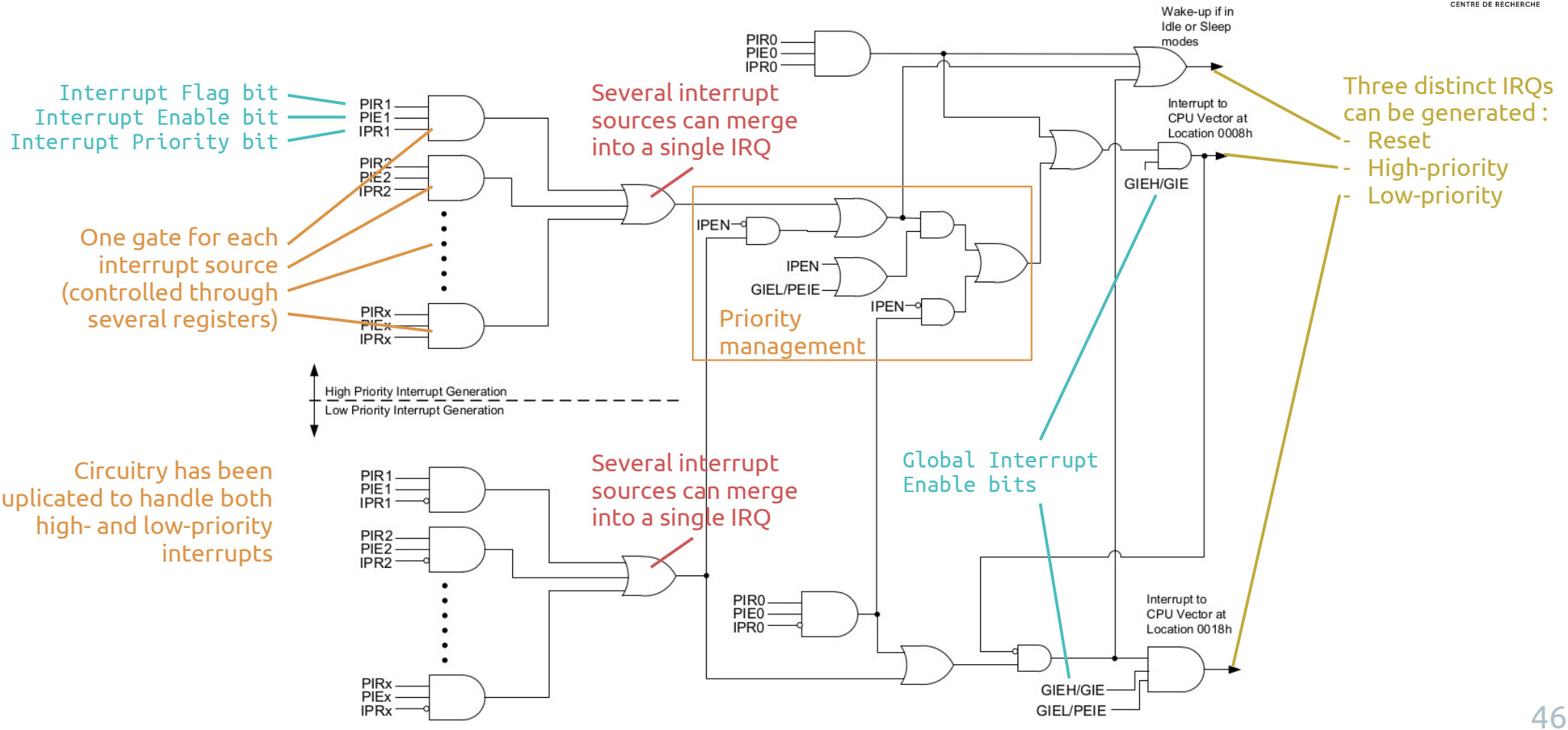
Interrupt controller

After configuring the interrupt controller for every single interrupt source, one last parameter should allow the CPU to see IRQ signals.

Most MCUs have a Global Interrupt Enable (GIE) bit to do so. The PIC18 has two of them: GIEH and GIEL for respectively high- and low-priority interrupts.



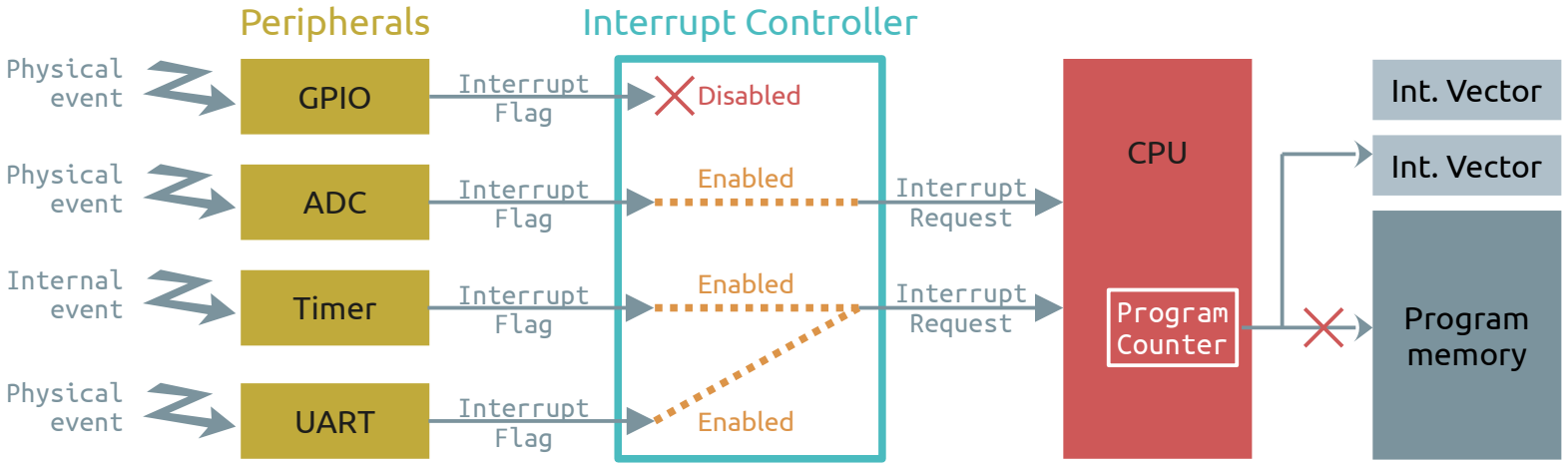
Interrupt controller: PIC18F27/47K40 interrupt controller logic

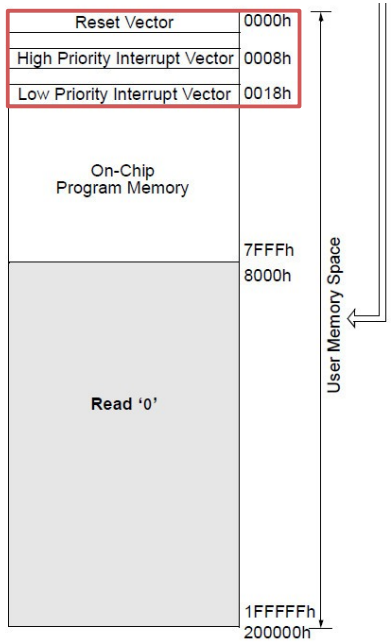


Interrupt Service Routine (ISR)

Interrupt Service Routines (ISR) are functions called by the **Interrupt vectors**, which are specific parts of the Flash memory.

When the CPU sees an **Interrupt Request (IRQ)**, it pauses its main program and branches automatically to the corresponding interrupt vector, causing the execution of the Interrupt Service Routine.





Interrupt Service Routine (ISR)

ISRs are not proper functions: they should not be called from the main program.

They are automatically called whenever the CPU sees the IRQ signals (this is called “event-driven programming”, fr: *programmation évènementielle*)

As ISRs are called at unpredictable moments, it is not possible to pass arguments to the ISR functions.

In order to exchange information between the main program and the ISRs, global variables can be used.

But remember global variables are shared resources and be used very carefully!

Interrupt Service Routine (ISR)

Here is an example of how writing an ISR using the Microchip XC8 toolchain.

```
 /*  ISR - high level Interrupt Service Routine  */
void interrupt high_priority high_isr(void) {

     if( PIR0bits.TMR0IF ) {
          PIR0bits.TMR0IF = 0;
          ...
     }

     if( PIR1bits.ADIF ) {
          PIR1bits.ADIF = 0;
          ...
     }
}


/* program entry point */
void main(void) {

     timer0_init();
     interrupt_enable();

     while(1) {
          /* user application scheduler */
     }
}
```

ISR for all high-priority interrupt sources

Check if Timer 0 is the source of this interrupt

Clear the Timer 0 interrupt flag, and proceed to what it should do

Check if ADC is the source of this interrupt

Clear the ADC interrupt flag, then proceed to what it should do

Configure interrupt for Timer 0

Enable interrupt for the CPU

Main routine (main application function)

Interrupt vector

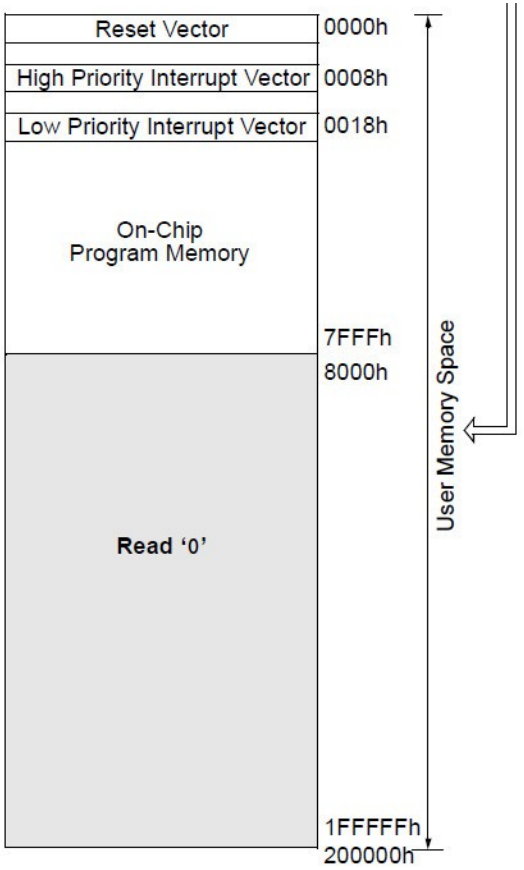
Interrupt vectors are very small areas in the Flash memory.

→ 16 bytes for the high-priority interrupt vector

→ 2 bytes for the low-priority interrupt vector

In fact, those areas are too small to contain the ISR but they are large enough to contain `CALL` or `GOTO` instructions, which will actually call the ISR (see next page).

Technically ISR are stored in the program memory, but they should be accessed only through interrupt vectors.



Context switch

As an ISR can be called at anytime, it will break the **context** that the main program is using (values of W-reg, STATUS, BSR for the PIC18). All registers must be saved in order to return to prior values when the ISR has ended.

The PIC18 toolchain will usually implement a hardware context backup for the high-priority ISR and a software context backup for the low-priority ISR.

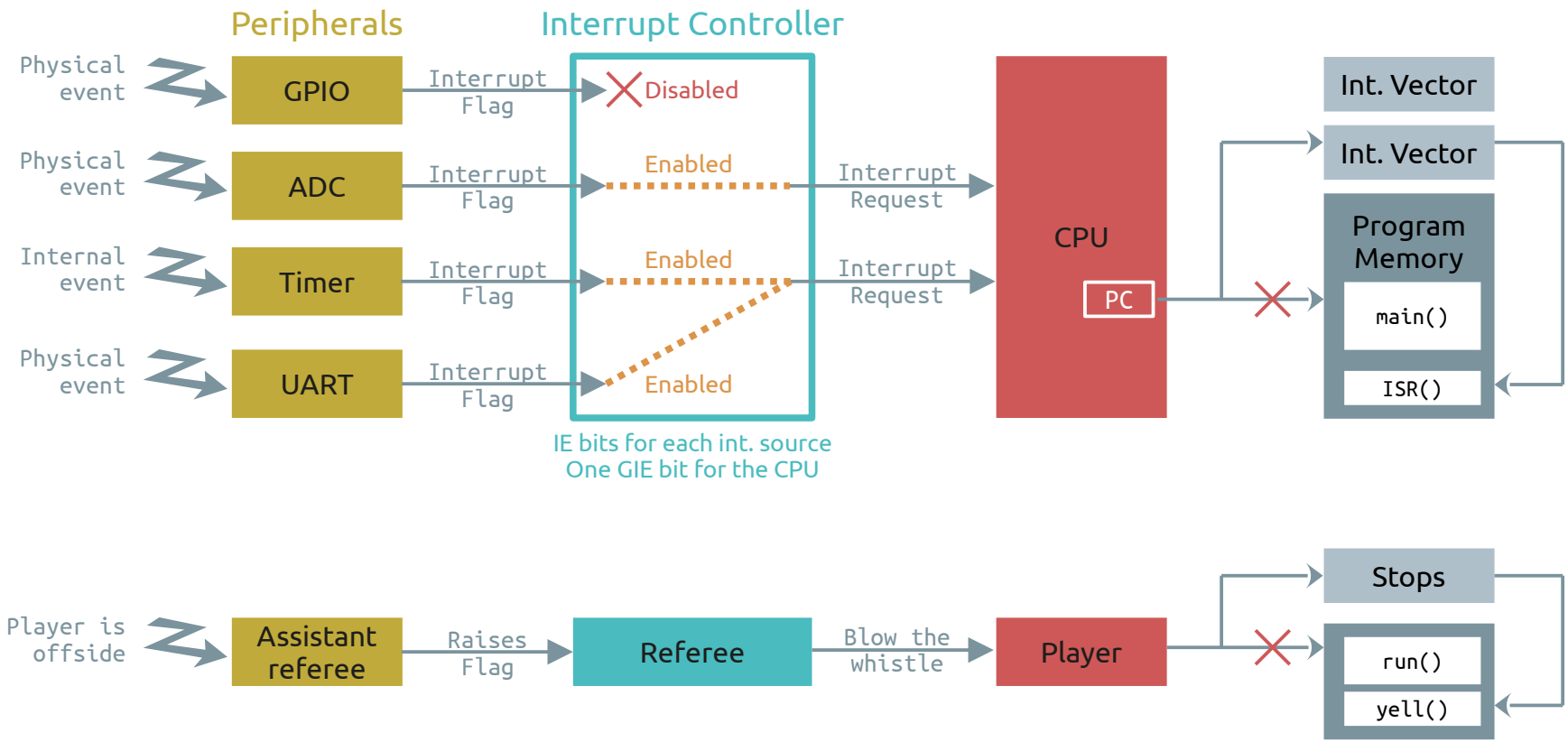
Hardware context backup (CPU shadow registers)

```
high_vector:
      CALL  high_isr, 1

high_isr:
      ; user program - ISR processing
      RETFIE      1
```

Hardware context backup (Flash memory)

```
low_vector:
      GOTO    low_isr

low_isr:
      MOVWF   wreg_tmp
      MOVFF   STATUS,     status_tmp
      MOVFF   BSR,        bsr_tmp
      ; user program - ISR processing
      MOVFF   bsr_tmp,    BSR
      MOVFF   status_tmp, STATUS
      MOVF    wreg_tmp,   W
      RETFIE
```

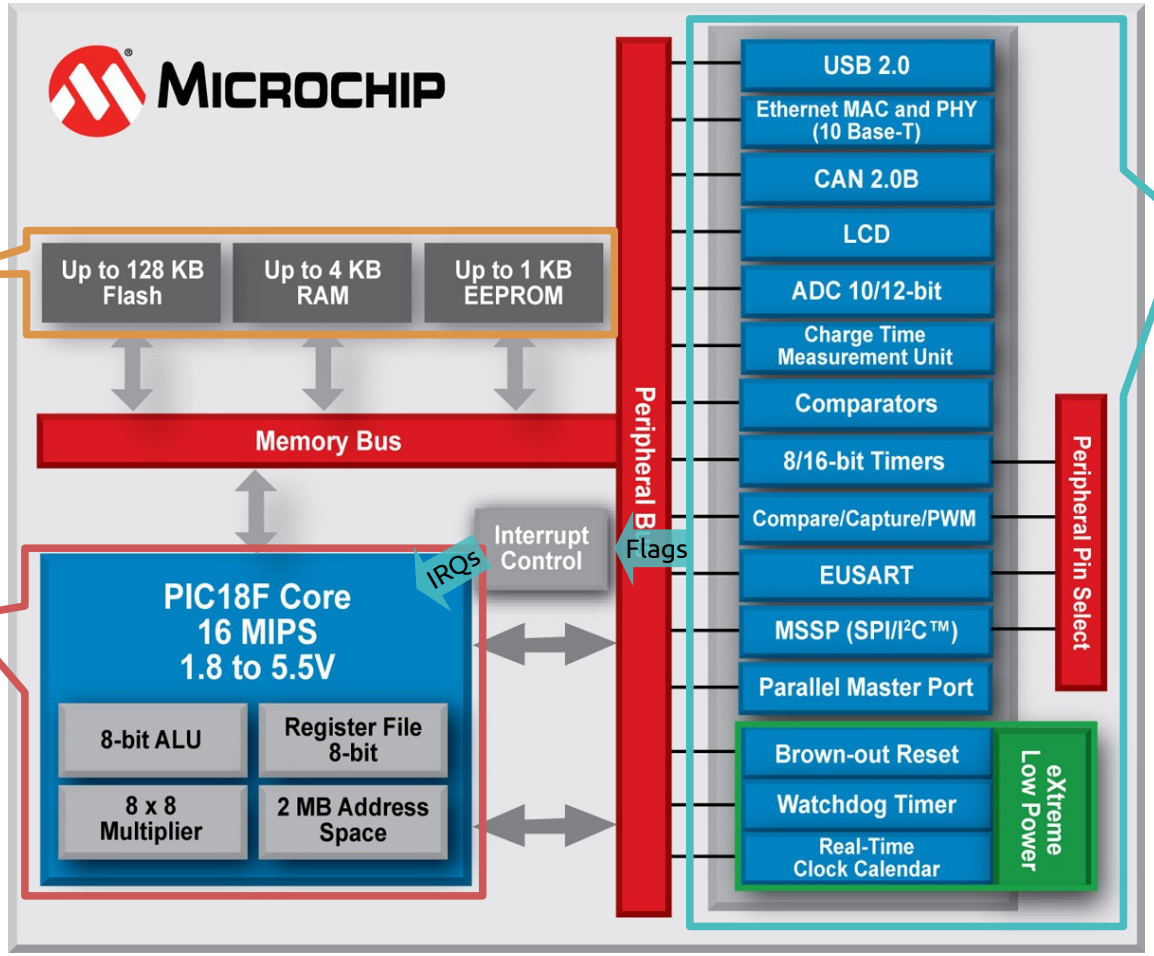
Summary


Peripherals
Interrupt Controller
Physical event
GPIO
Interrupt Flag
Disabled
Physical event
ADC
Interrupt Flag
Enabled
Interrupt Request
Internal event
Timer
Interrupt Flag
Enabled
Interrupt Request
Physical event
UART
Interrupt Flag
Enabled
IE bits for each int. source
One GIE bit for the CPU
CPU
PC
Int. Vector
Int. Vector
Program Memory
main()
ISR()
Player is offside
Assistant referee
Raises Flag
Referee
Blow the whistle
Player
Stops
run()
yell()

PIC18 architecture

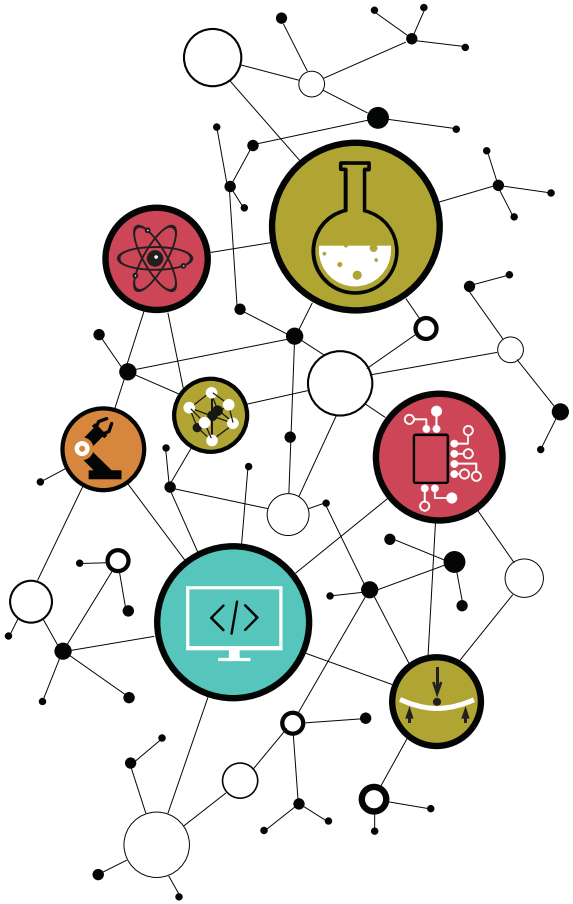
Program & data memories

Central Processing Unit



Peripherals

CONTACT



Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

With the precious help of:

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