Chapitre 5

Architecture Microchip PIC18

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Part de marchés des compagnies semiconducteurs et fabricants de MCU en 2016.

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%

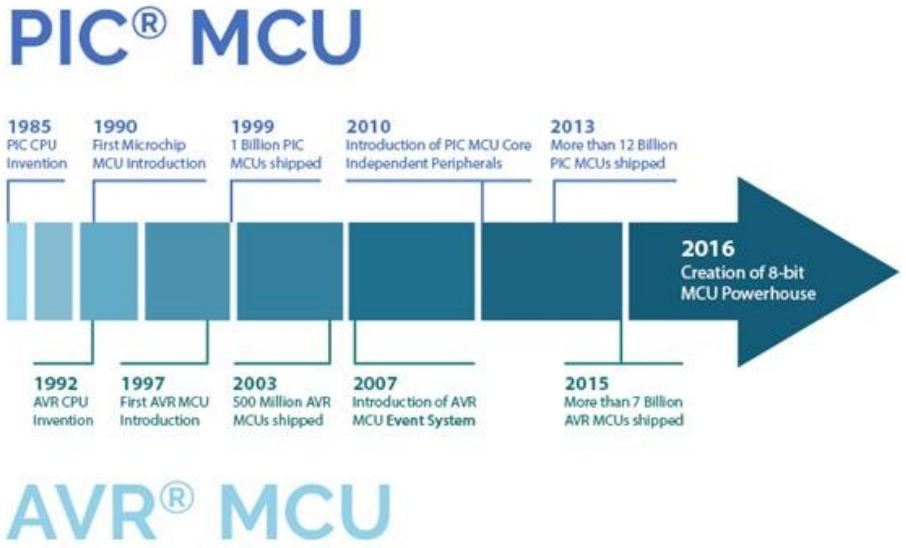
Source: Company reports, IC Insights

L'américain Microchip est un fabricant de composants électroniques La plus grande partie de son chiffre d'affaires est liée aux MCU. D'après Microchip ESC Filing, 60 % de ses revenus sont générés par la famille PIC.

En 2016 Microchip rachète Atmel, plus grand concurrent sur le marché des MCU 8-bits.

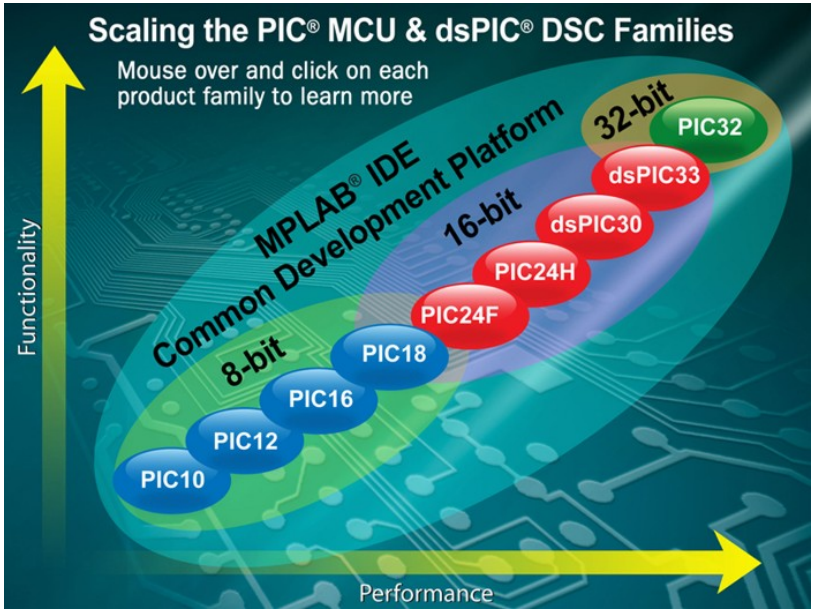
MICROCHIP

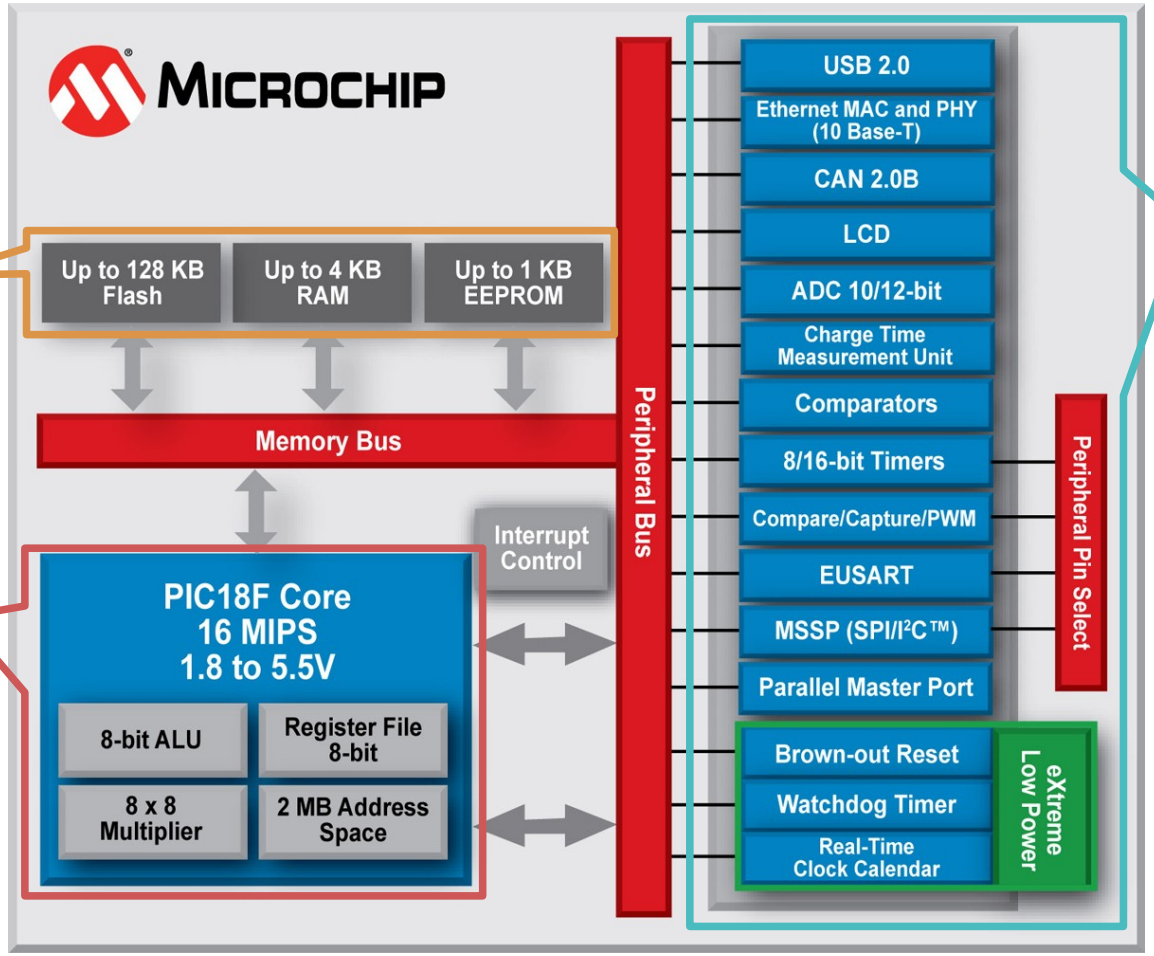
Atmel



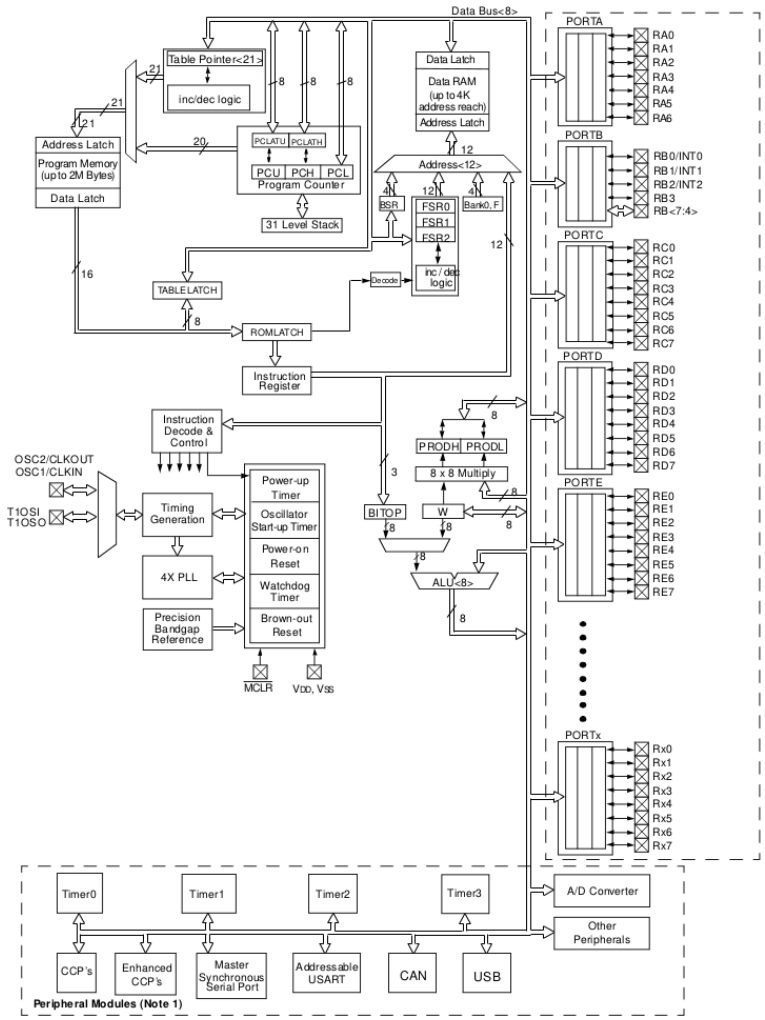
Avec sa large gamme de micro-contrôleurs, Microchip peut s'assurer de la fidélité des clients en leur donnant la possibilité de viser de nombreux marchés et applications.

Comme n'importe quel fabricant, Microchip fournit également des outils permettant la transition d'une architecture vers une autre (par ex. migration de PIC18 vers PIC32).



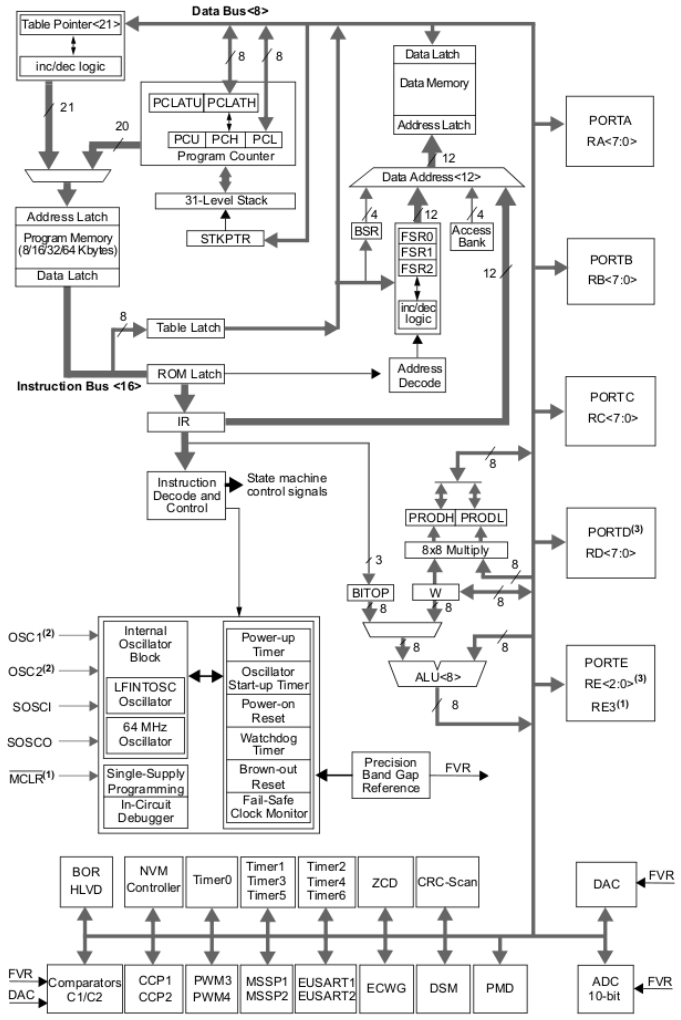

Program & data memories
Central Processing Unit
Peripherals
MICROCHIP
Up to 128 KB Flash
Up to 4 KB RAM
Up to 1 KB EEPROM
Memory Bus
Interrupt Control
PIC18F Core
16 MIPS
1.8 to 5.5V
8-bit ALU
Register File 8-bit
8 x 8 Multiplier
2 MB Address Space
Peripheral Bus
USB 2.0
Ethernet MAC and PHY (10 Base-T)
CAN 2.0B
LCD
ADC 10/12-bit
Charge Time Measurement Unit
Comparators
8/16-bit Timers
Compare/Capture/PWM
EUSART
MSSP (SPI/I²C™)
Parallel Master Port
Peripheral Pin Select
Brown-out Reset
Watchdog Timer
Real-Time Clock Calendar
eXtreme Low Power

Quelles sont les différences ?



PIC18C family (left)

PIC18F27/47K40 (right)

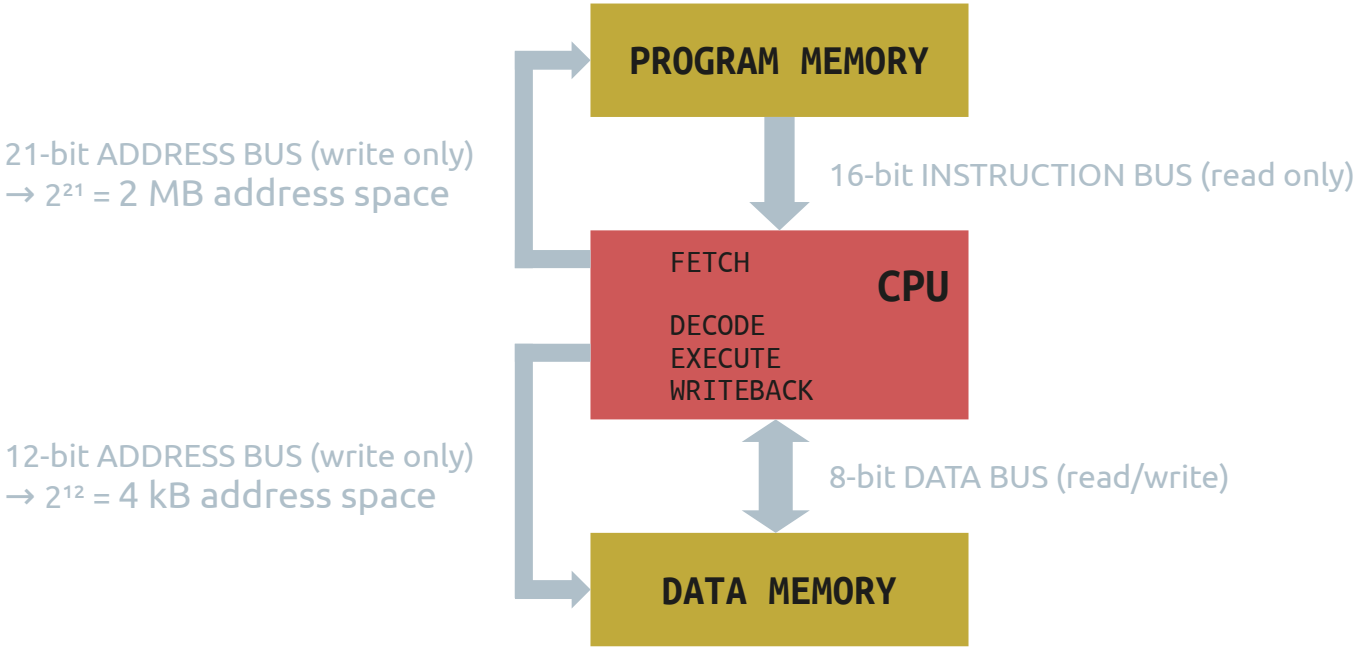


CENTRAL PROCESSING UNIT

ENSI CAEN

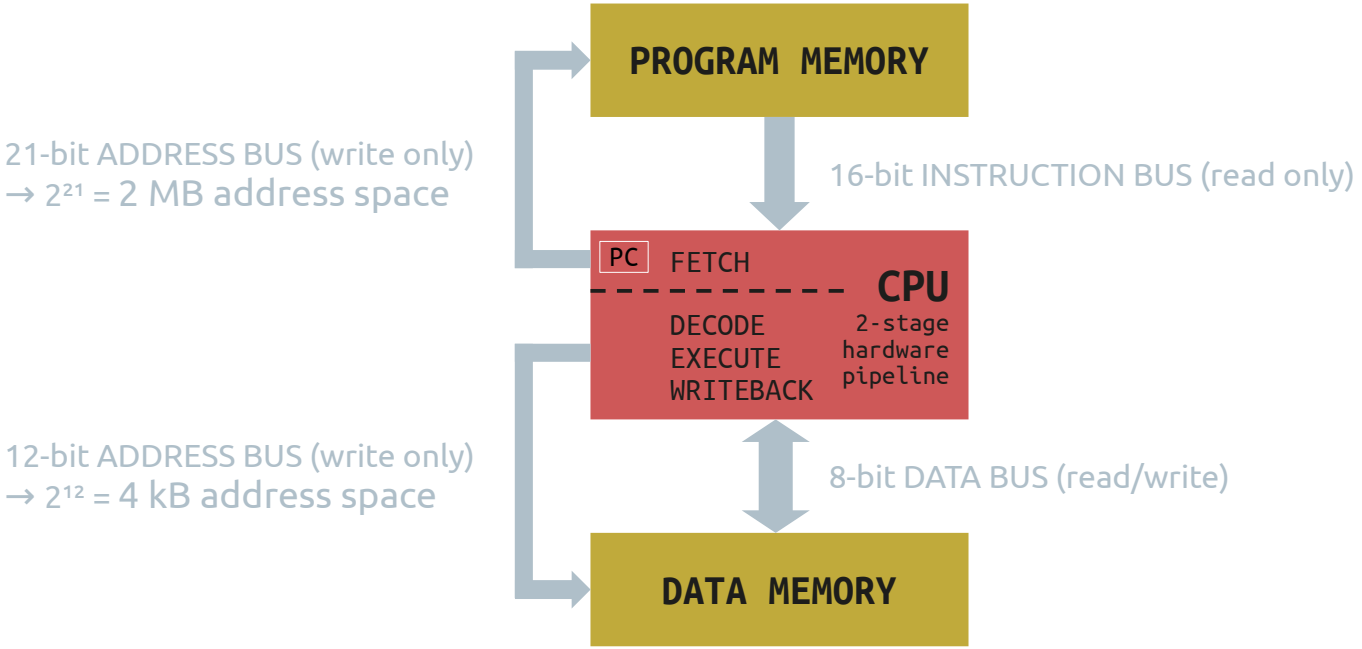
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Tout comme l'AVR d'Atmel, le PIC18 de Microchip suit une architecture Harvard : les mémoires programme et données sont physiquement séparées et leurs espaces d'adressage sont distincts.



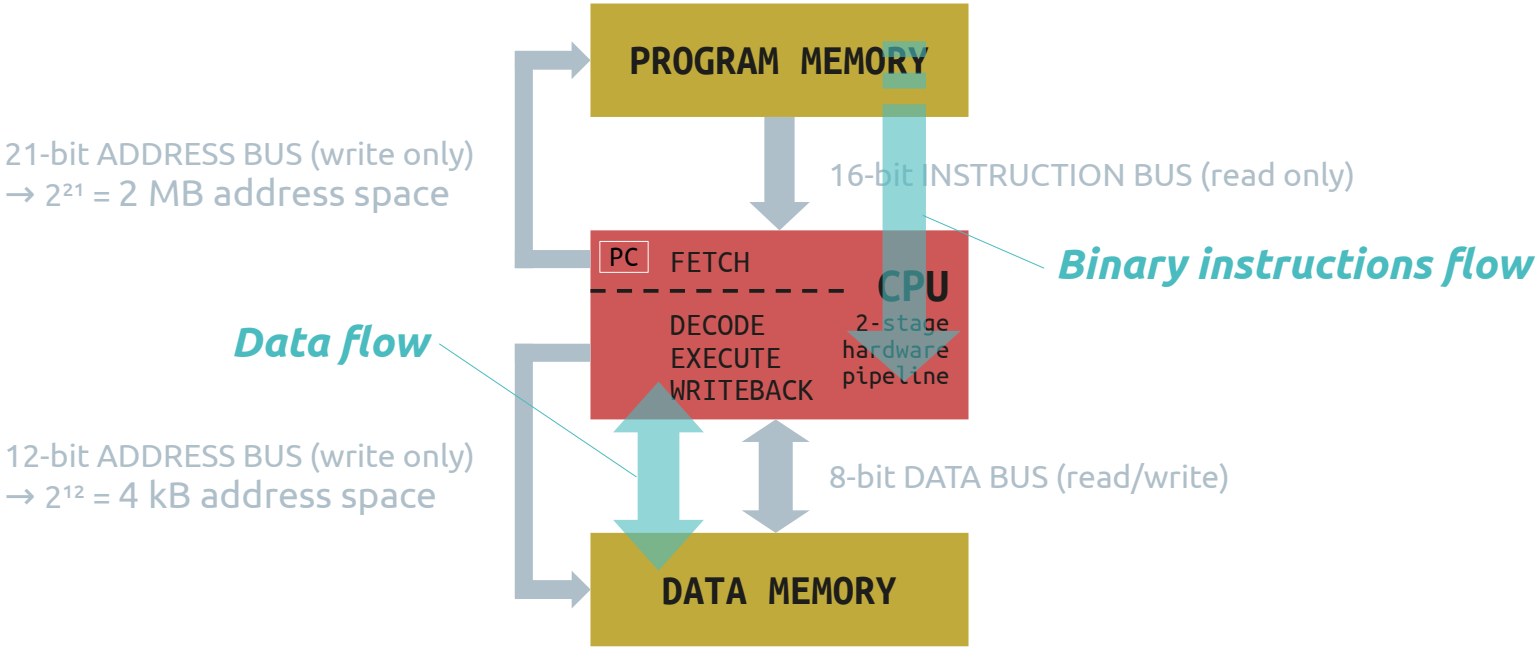
Les CPU des PIC18 possède un pipeline matériel à 2 étages. Ils peuvent décoder-exécuter-sauver une instruction tout en cherchant la suivante en mémoire programme.

Max performance = 16 MIPS.



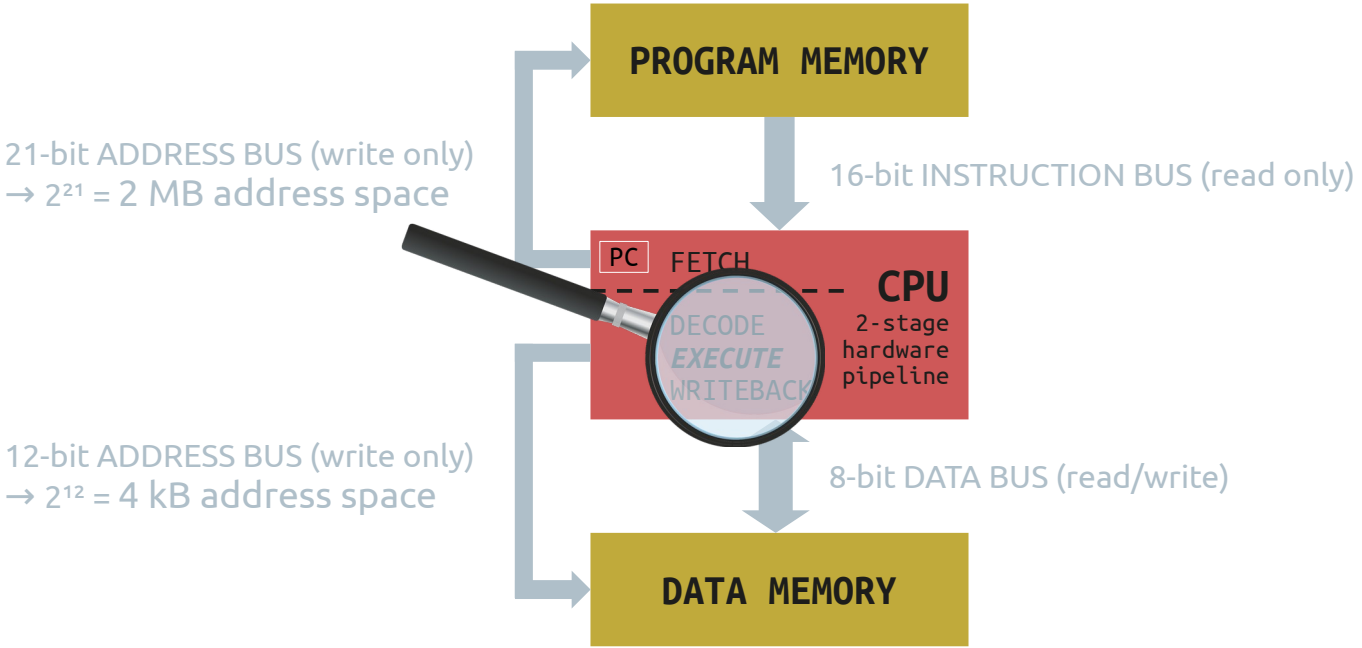
À part en mode veille, le CPU exécute un flot constant d'instructions en provenance de la mémoire programme.

Certaines demandent au CPU de lire ou écrire une donnée dans la mémoire données.



Concentrons-nous sur l'étage d'exécution pour voir les *Execution Units* (EUs) du PIC18.

Le PIC18 étant un processeur 8-bits, il ne peut gérer que des données sur 8 bits.



Execution Unit: ALU

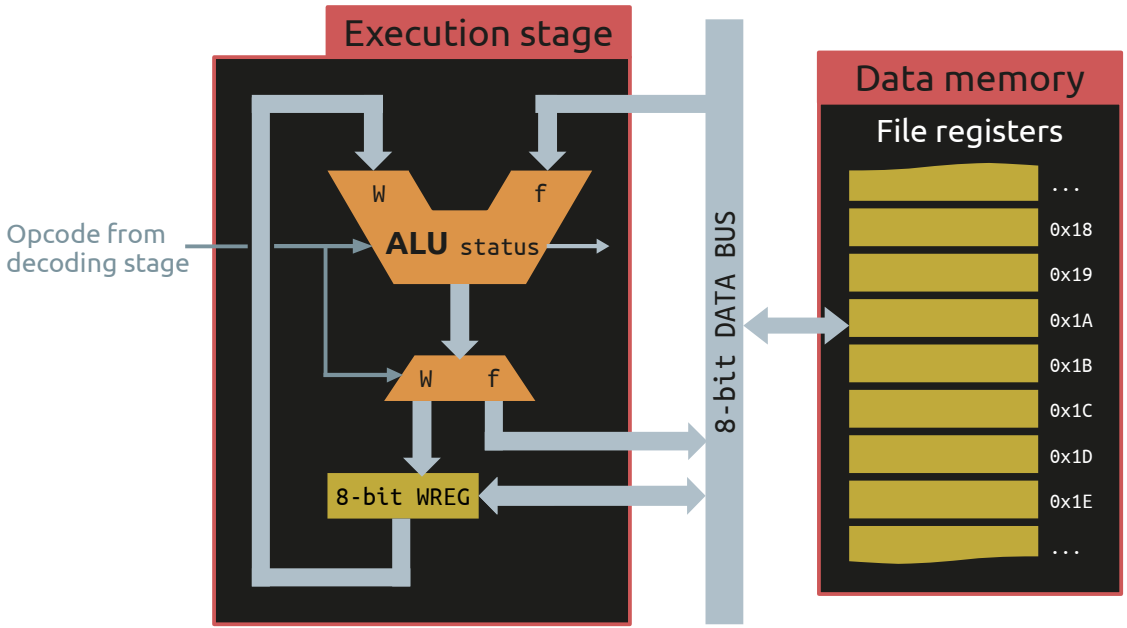
L'***Arithmetic and Logic Unit* (ALU)** est l'unité d'exécution en charge des opérations arithmétiques (+, -) et logiques (&, |, ^, !, …) sur des entiers 8 bits.

Arithmetic operations

ADDWF
INCF
SUBWF
DECF
...

Logic operations

ANDWF
IORWF
XORWF
CLRF
SETF
...



Execution Unit: ALU

Pour les opérations à deux opérandes, une seule donnée peut provenir de la mémoire donnée, tandis que l'autre provient d'un registre interne au CPU (WREG, *Working reg*).

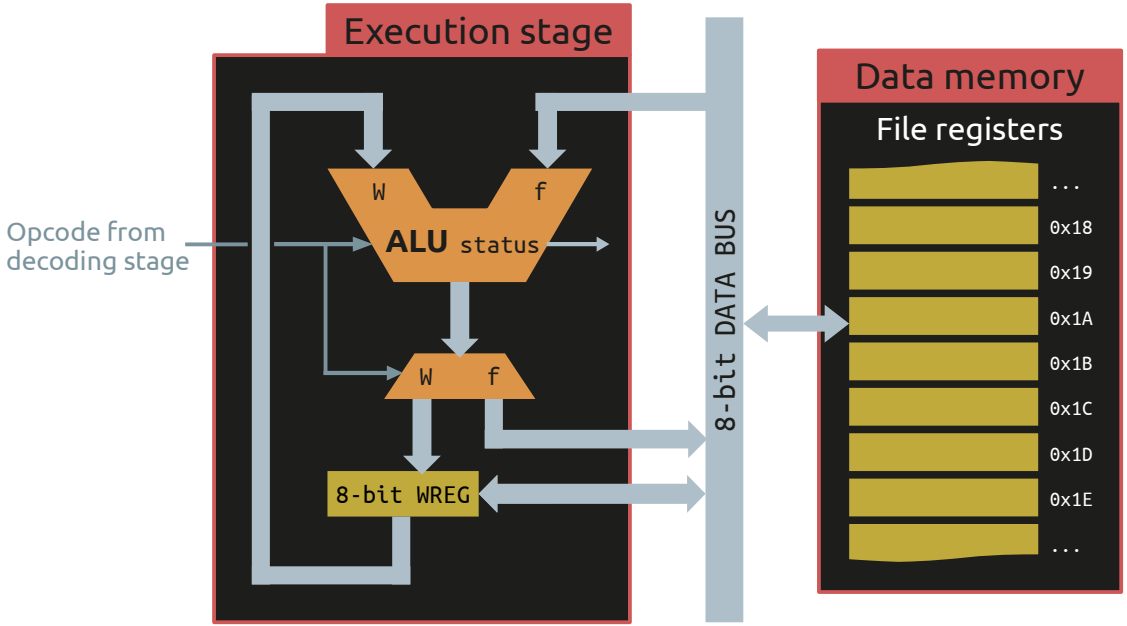
Exemple d'une instruction en assembleur et son équivalent en code binaire.

PIC18 assembly language:

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

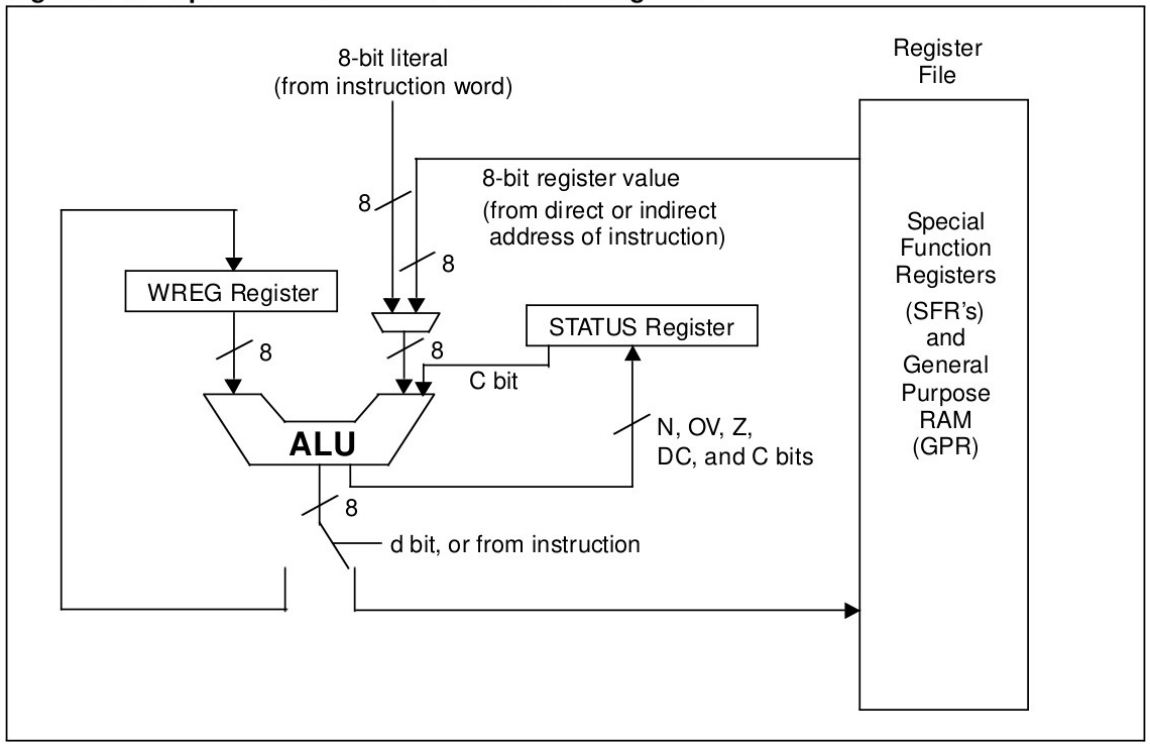
16-bit opcode:

```
0010 01da ffff ffff
```



Autre point de vue, celui du PIC18C Family Reference Manual.

Figure 5-4: Operation of the ALU and WREG Register



PICmicro devices contain an 8-bit ALU and an 8-bit working register (WREG). The ALU is a general purpose arithmetic and logical unit. It performs arithmetic and Boolean functions between the data in the working register and any register file.

The ALU is 8-bits wide and is capable of addition, subtraction, multiplication, shift and logical operations.

In two-operand instructions, typically one operand is the working register (WREG register). The other operand is a file register or an immediate constant. In single operand instructions, the operand is either the WREG register or a file register.

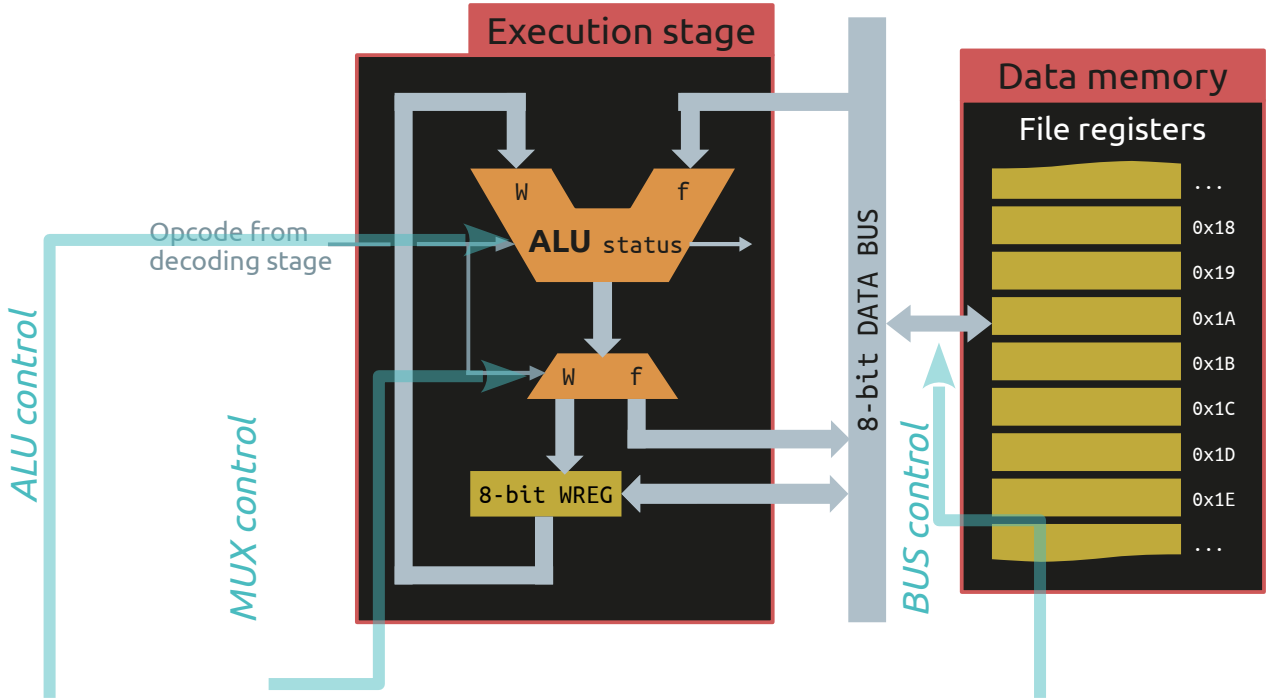
Depending on the instruction executed, the ALU may affect the values of the Carry (C), Digit Carry (DC), Zero (Z), Overflow (OV), and Negative (N) bits in the STATUS register.

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

Numération

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

Instructions de multiplication

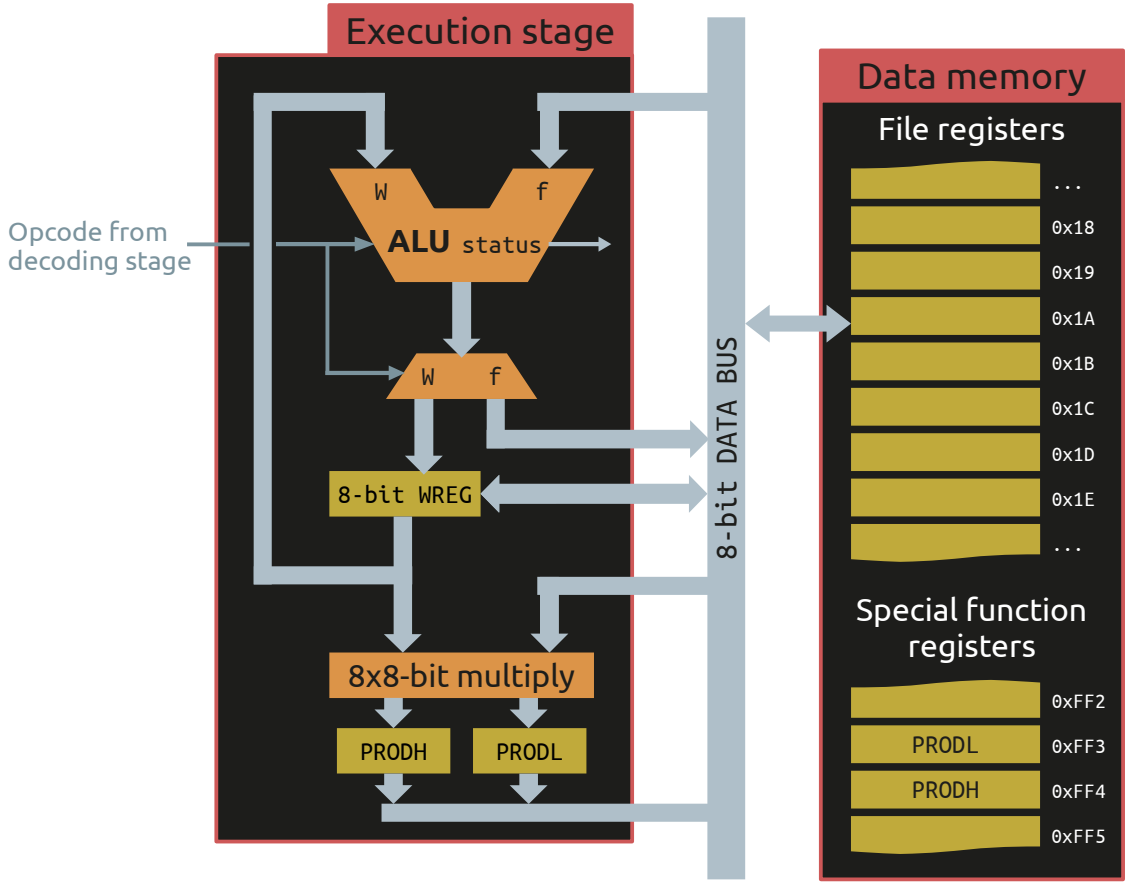
MULWF (W-reg to F-reg)
MULLW (Litteral to W-Reg)

Exemple en C et assembleur

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

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

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

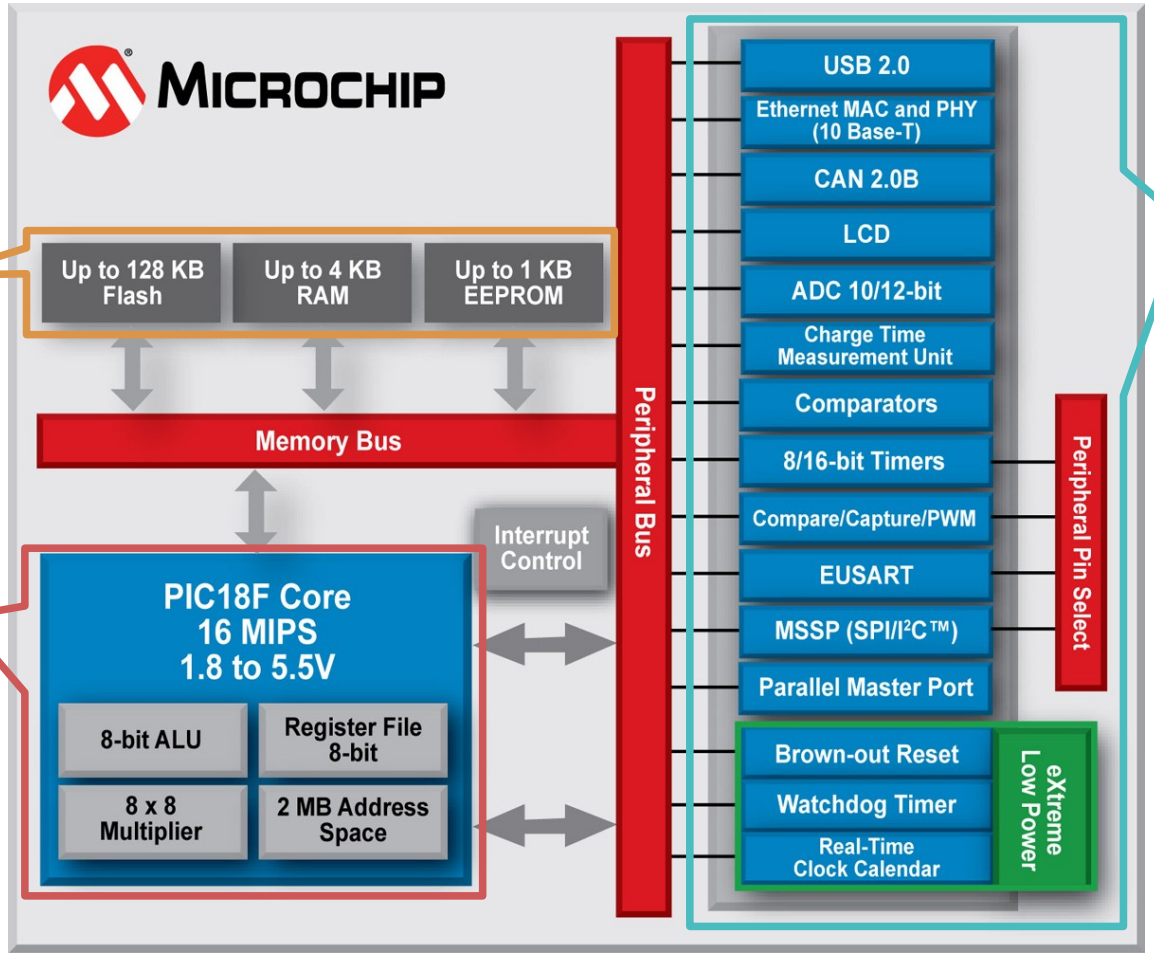
PRODL est un alias pour 0xFF3, déclaré dans un header
PRODH est un alias pour 0xFF4, déclaré dans un header



Architecture PIC18

Program & data memories

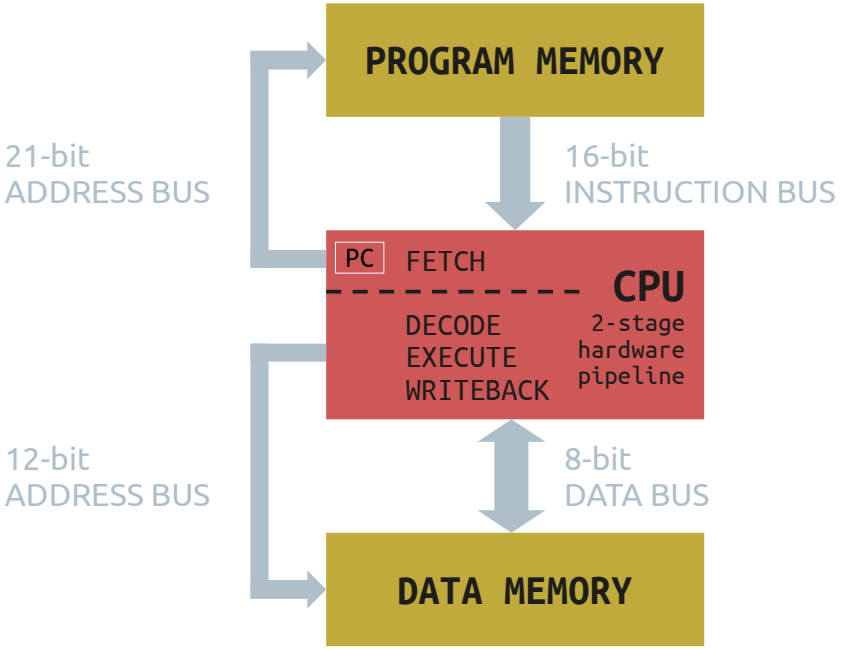
Central Processing Unit

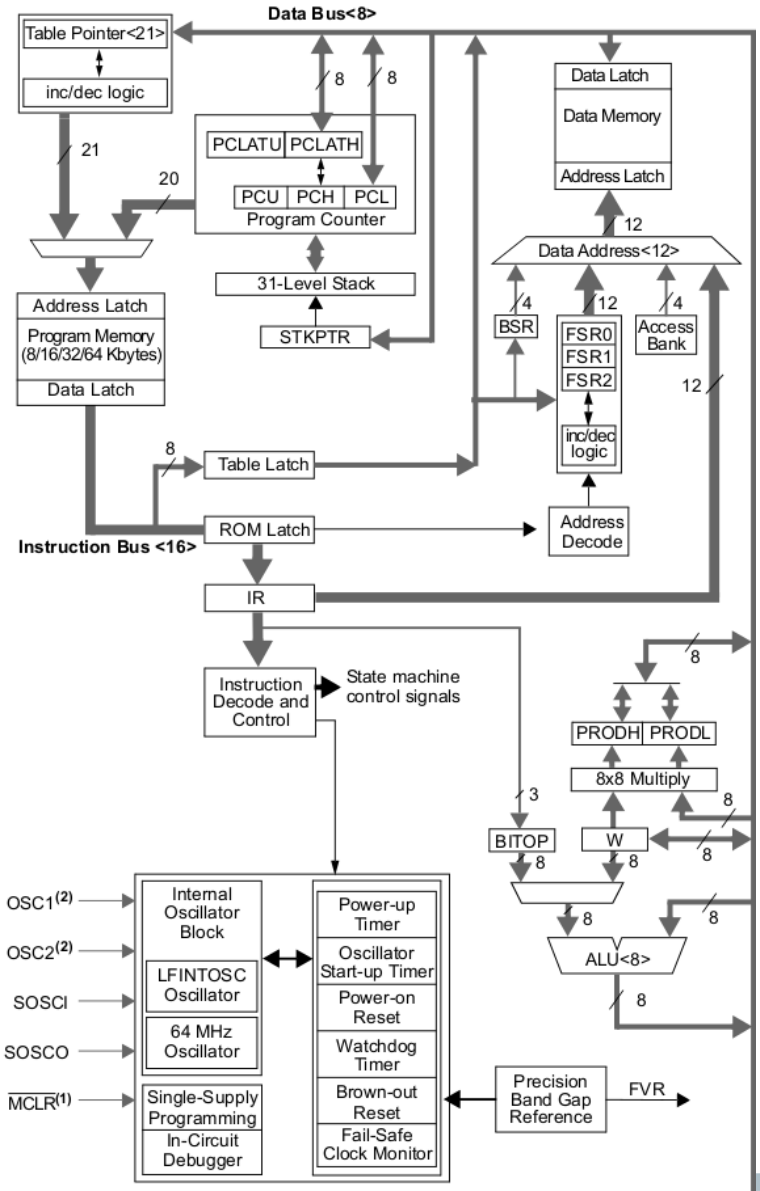


Peripherals

PIC18 CENTRAL PROCESSING UNIT

Architecture du CPU PIC18F27/47K40





Architecture du CPU PIC18F27/47K40

Trouvez ces composants dans le schéma

Flash memory

RAM memory

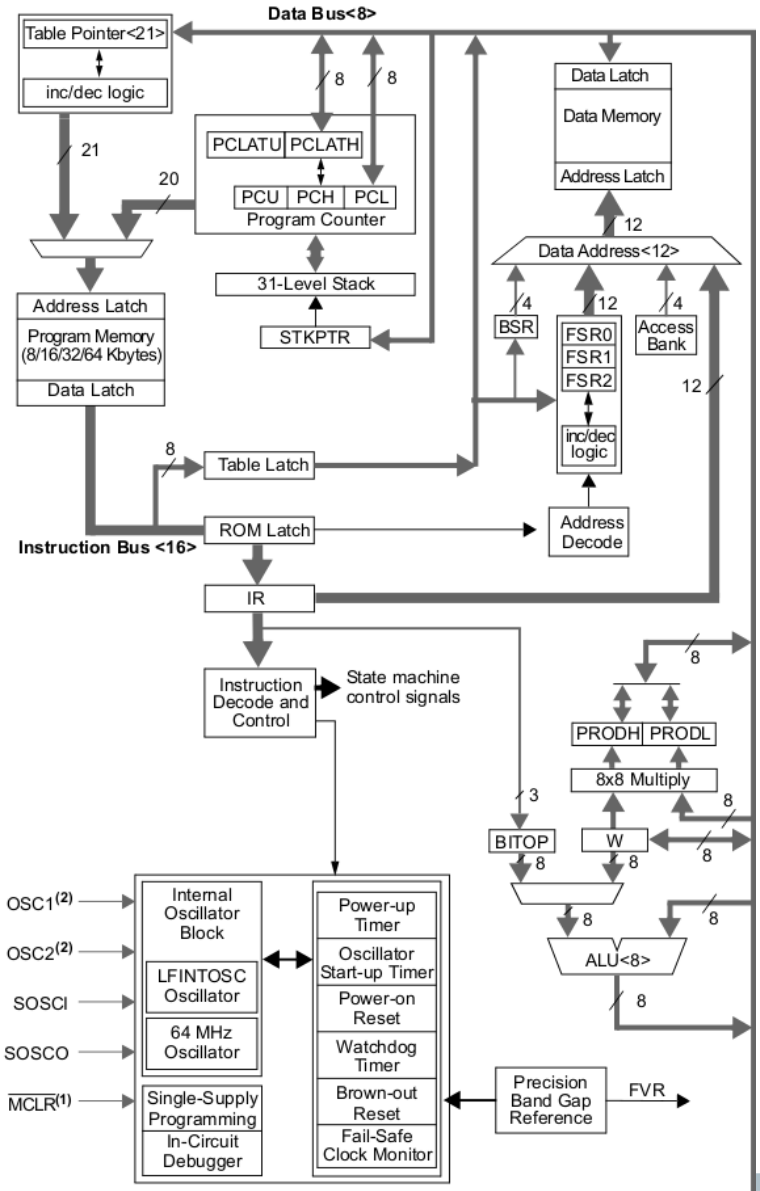
Buses

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

Hardware pipeline stages

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

Program Counter register



Avec les MCU, il faut adapter ses habitudes de codage à l'architecture cible.

Ce CPU (comme la plupart des *MCU low-power*) n'a pas de *Floating-Point Unit*. Il faut donc éviter l'utilisation de *float* et *double* et utiliser des entiers à la place.

Aussi, comme il s'agit d'un CPU 8-bits il faut utiliser autant que possible des entiers sur 8 bits (type char en C). Cela suffit généralement pour des applications de supervision.

Enfin, le jeu d'instruction nous renseigne sur les opération nativement réalisées par le CPU (+, *, …). Ici il faudra éviter d'utiliser des opérations complexes telles que '/', '%', …

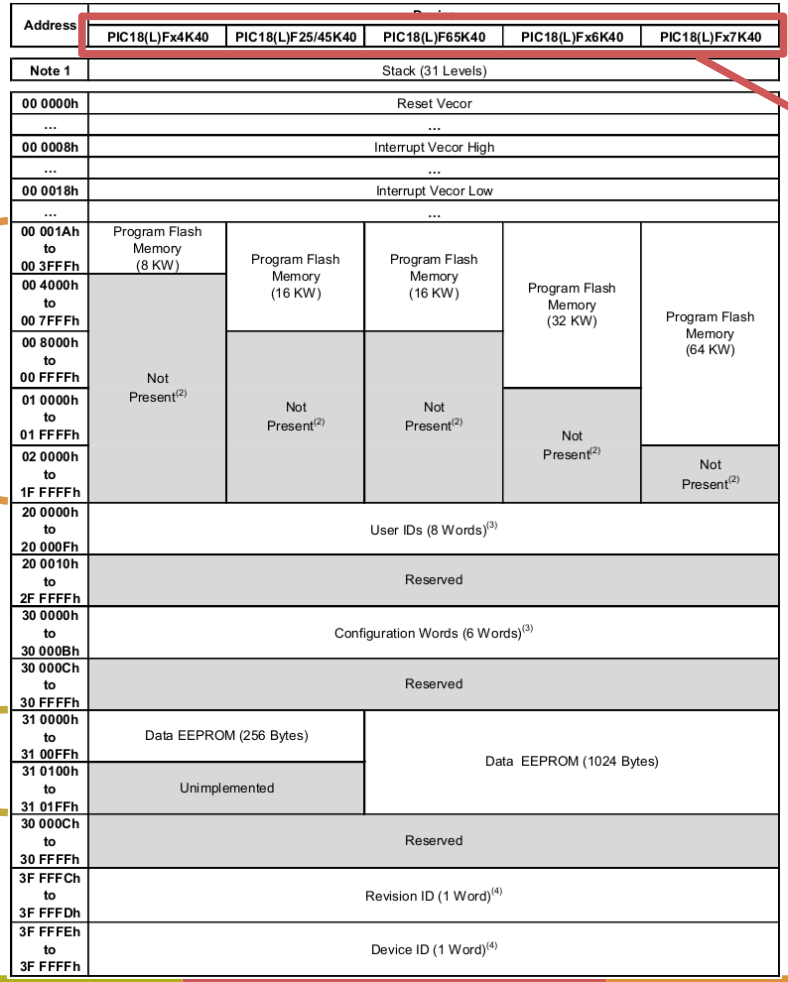
C-Type	custom typedef	Memory space	Values
char	int8	8 bits / 1 byte	-128 / 127
unsigned char	uint8	8 bits / 1 byte	0 / 255
short	int16	16 bits / 2 bytes	-32768 / +32767
unsigned short	uint16	16 bits / 2 bytes	0 / +65535
long	int32	32 bits / 4 bytes	-2G / +2G
unsigned long	uint32	32 bits / 4 bytes	0 / +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)	

MÉMOIRE

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Plan d'adressage des mémoires programme et données



Address	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

La taille de la mémoire programme dépend de la référence du PIC18

Mémoire programme du PIC18F4550

Program Counter (pour l'étage de *Fetch*)

Pile matérielle, max 31 appels imbriqués (sauvegarde l'adresse de retour)

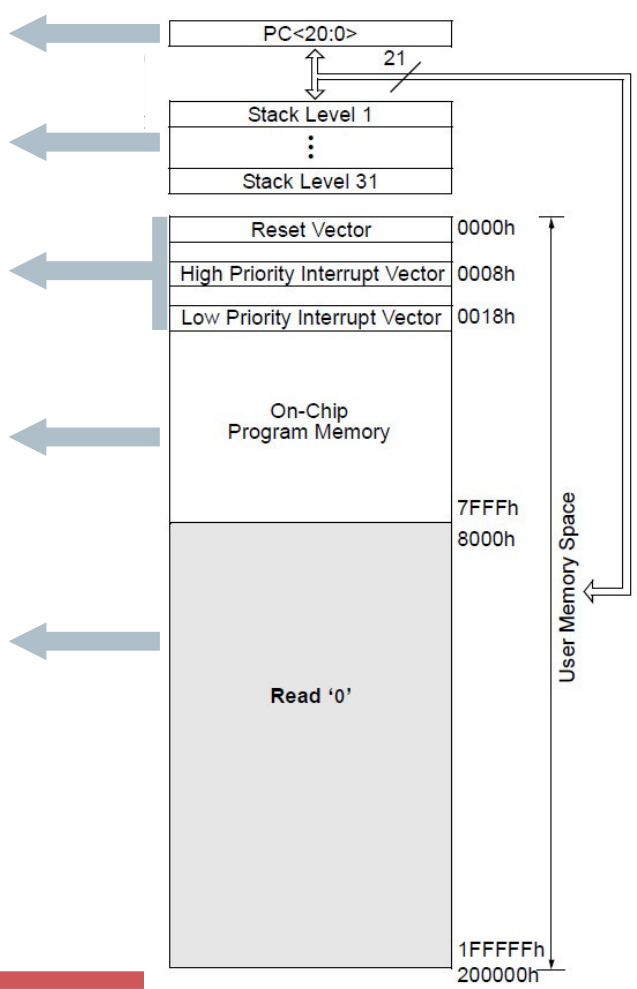
Table des vecteurs d'interruption (redirection des IRQ vers les ISR)

Espace mémoire pour le logiciel embarqué (32 kB pour le PIC18F4550 présenté ici)

Espace vide, dépend de la référence du MCU

Certains PIC18 ont 128 kB de mémoire programme

L'espace restant peut être réservé pour de futures évolutions



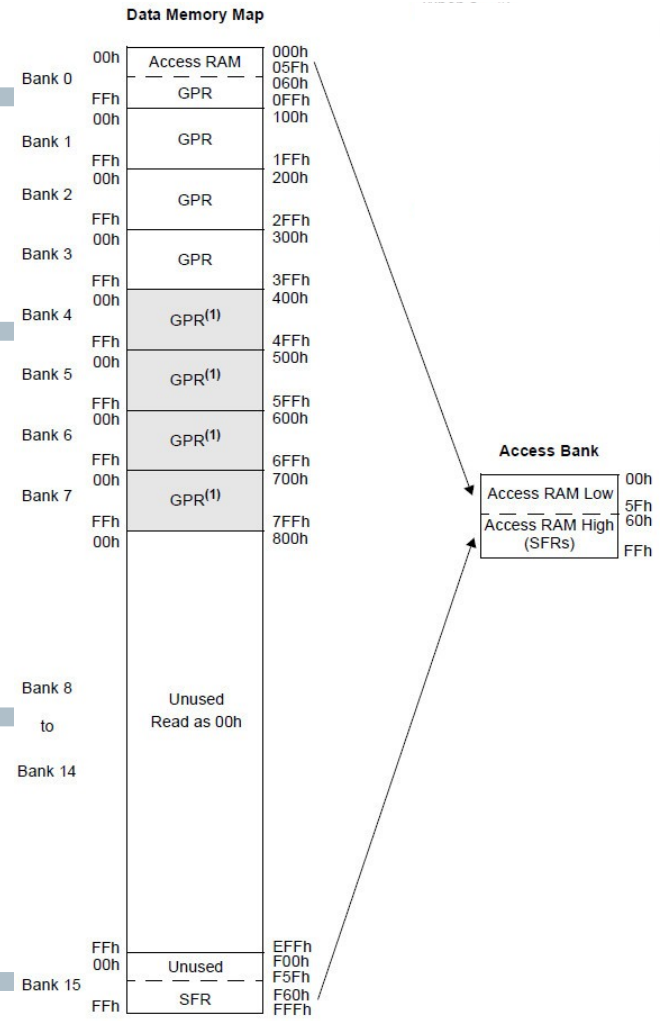
Mémoire données du PIC18F4550

La mémoire de données est segmentée en 16 banques de 256 octets

GPR (*General Purpose Register file*)
Registres qui peuvent contenir n'importe quelle donnée, utilisable par toutes les instructions.

2 ko de mémoire données pour ce PIC18F4550
Jusqu'à 4 ko pour d'autres PIC18

SFR (*Special Function Registers*),
Liés au CPU et aux registres des périphériques



Mémoire données : sélection de la banque

La mémoire données est segmentée en 16 banques de 256 octets. Cette construction est classique sur les architectures 8-bits.

La banque de travail est sélectionnée grâce à 4 bits dans le BSR (*Bank Select Register*).

L'instruction utilisera 8 bits pour indiquer la donnée à traiter dans cette banque (*direct addressing mode*).

Bank select

PIC18 C language:

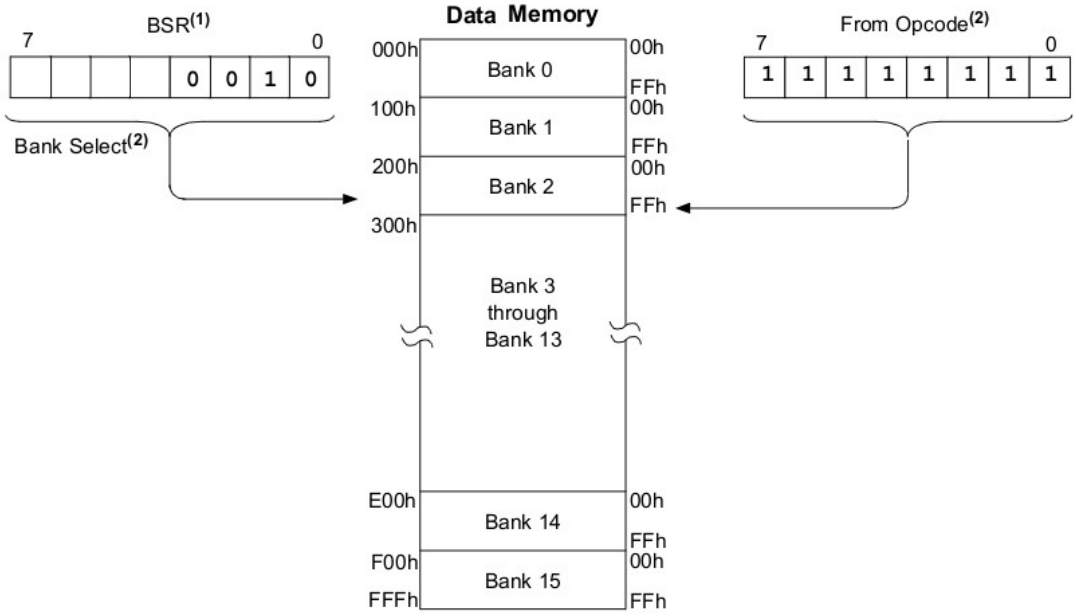
```
BSR = 0x02;
```

PIC18 assembly language:

```
MOVLW 0x02
MOVWF BSR
```

Ou bien utiliser l'instruction dédiée :

```
MOVLB 0x02
```



From the 16-bit opcode

PIC18 assembly language:

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

16-bit opcode:

```
0010 01da ffff ffff
```

Mémoire données : *Special Function Registers* (SFR)

La banque SFR est une partie de la mémoire données (moitié inférieure de la *bank 15*).

Les registres des périphériques y sont mappés.

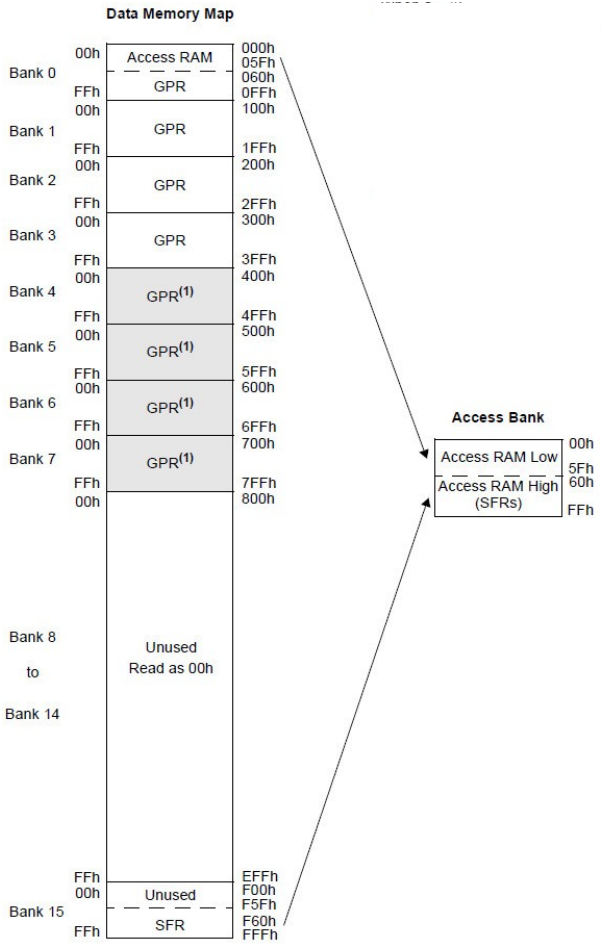
Ça signifie que tous les registres sont accessibles en RAM, même s'ils sont physiquement implantés auprès de leur périphérique.

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)

Mémoire données : *access bank*



Lorsqu'un instruction cherche à accéder à une donnée en mémoire, l'étage d'exécution regarde la valeur du bit <a>

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

<a> = '0' or '1'

Si <a> = '0', le CPU ne voit que l'***access bank (top half of the bank #0 + SFR bank)***. Les 4 bits de poids fort de l'adresse 12-bit proviennent de l'*access bank register* (0x0 ou 0xF) → **Solution rapide**.

Si <a> = '1', le CPU peut avoir accès à toute la mémoire données. Il utilise la valeur stockée dans le **BSR (*Bank Select Register*)** pour générer les 4 bits de poids fort pour construire l'adresse 12-bits de la mémoire données.

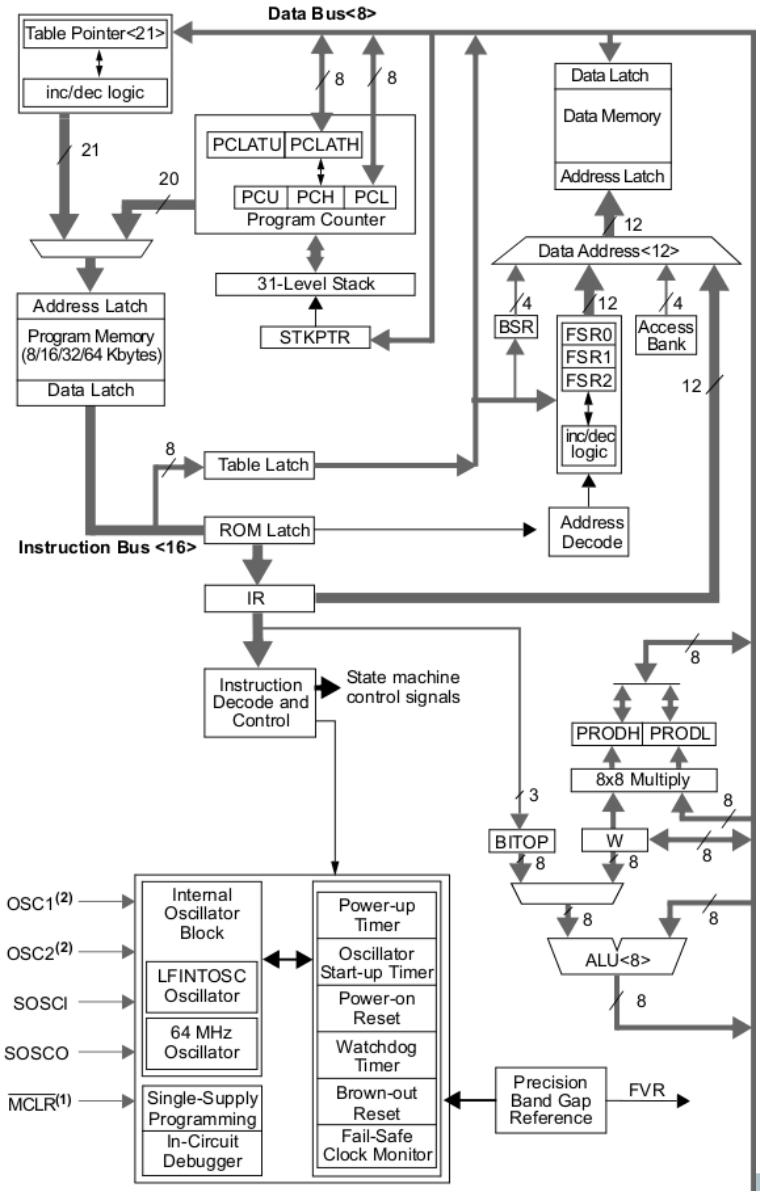
Retrouvez dans le schéma

Mémoire de donnée

BSR (*Bank Select Register*)

Access bank register

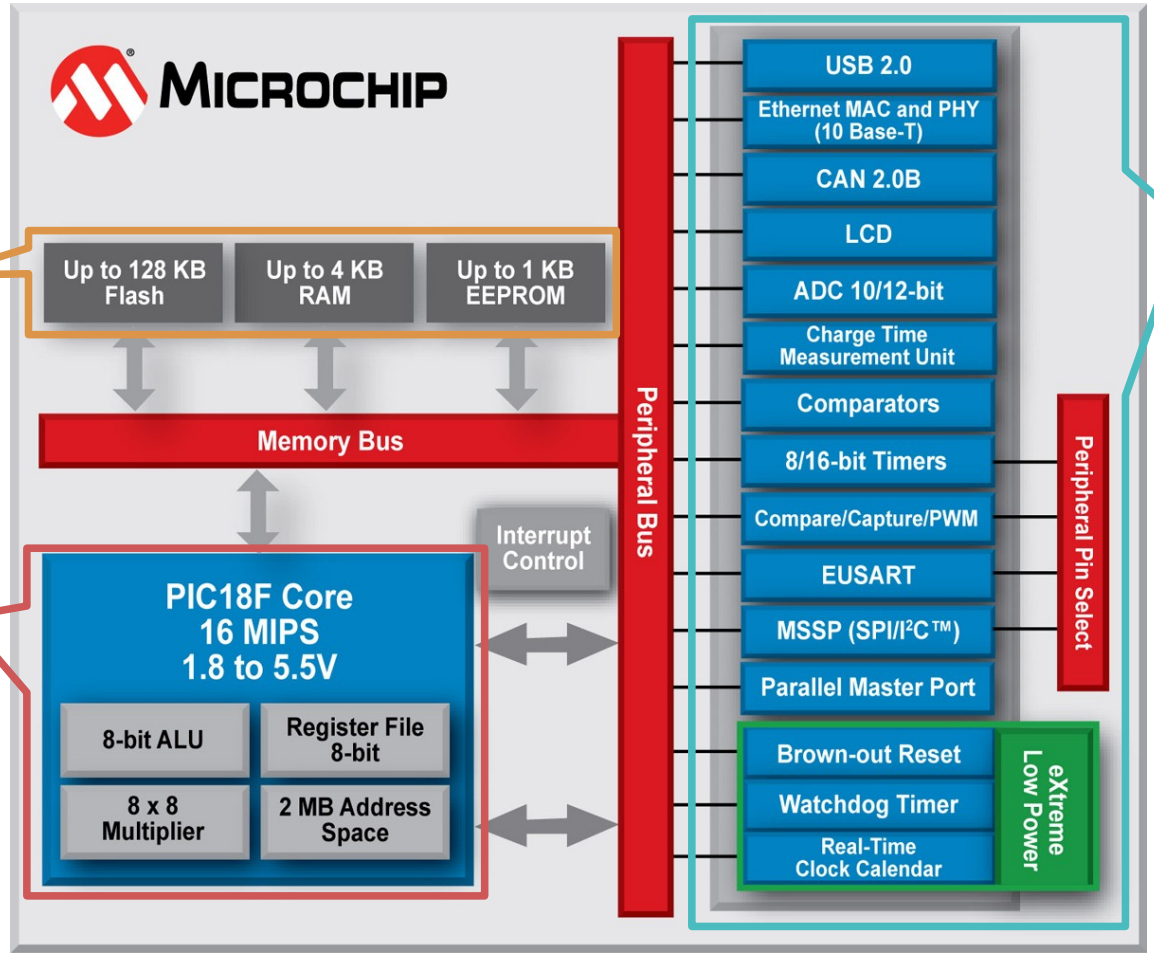
Observez comment ces éléments sont placés et comment ils permettent de générer des adresses 12-bits.



Architecture PIC18

Program & data memories

Central Processing Unit



Peripherals

PÉRIPHÉRIQUES

ENSI CAEN

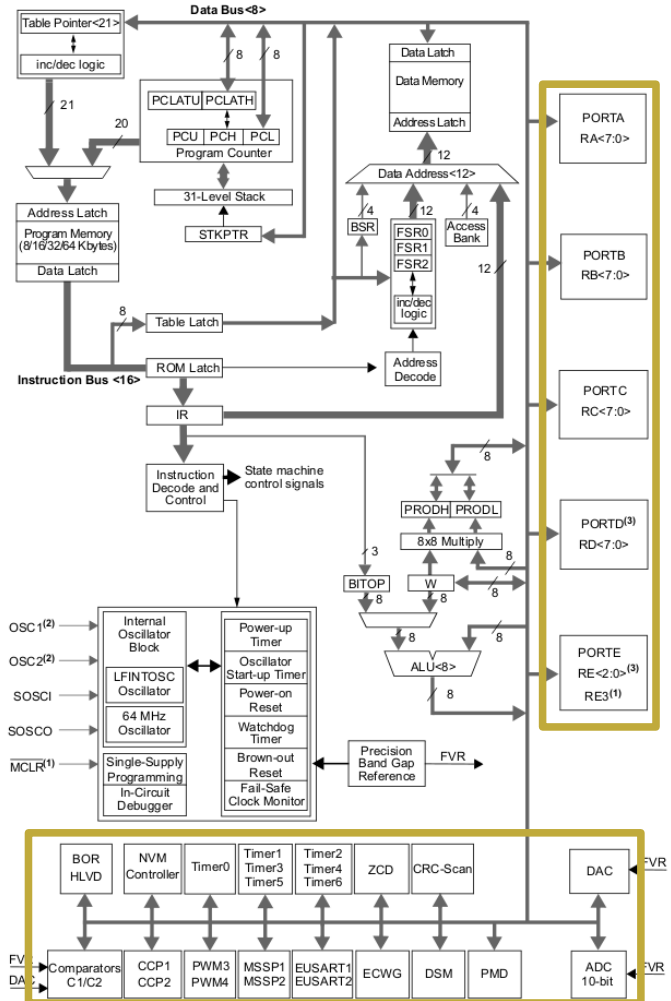
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Un périphérique est une fonction matérielle qui peut être utilisée pour réaliser un traitement spécifique et ainsi alléger la charge du CPU, qui pourra exécuter autre chose pendant ce temps.

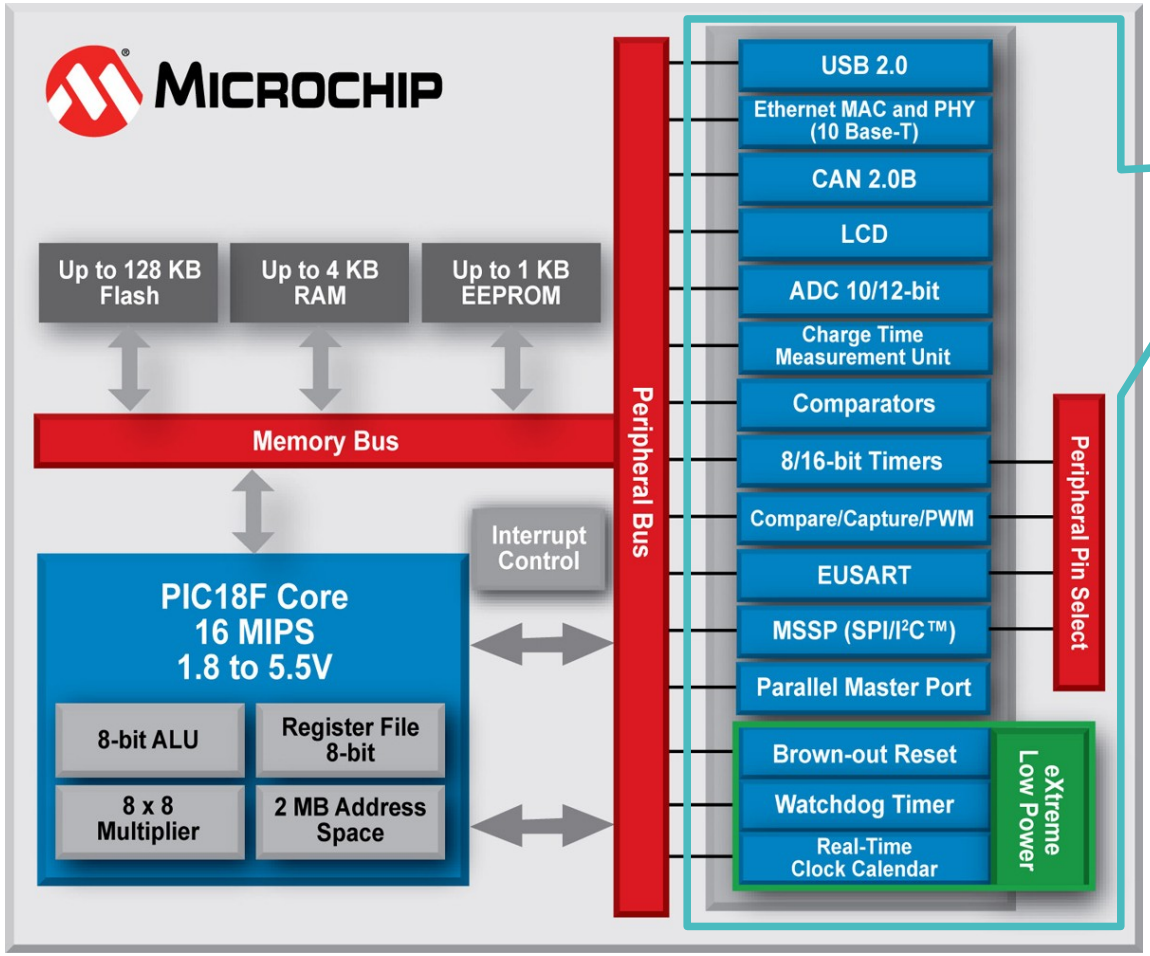
Pour tout micro-contrôleur, les périphériques sont reliés au bus de données et leurs registres internes sont mappés vers la mémoire de données.

Du point de vue du programme, accéder à un registre (lecture ou écriture) revient à accéder à un simple élément de la mémoire.

Exemple du PIC18F27/47K40 CPU et de ses périphériques.



Fonctions matérielles



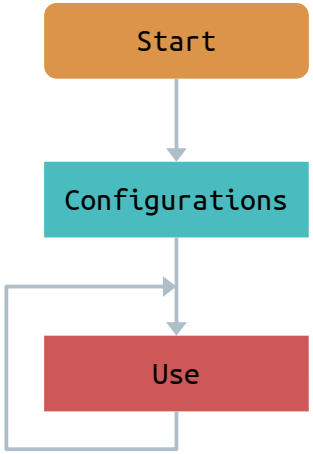
Périphériques, que font-ils ?

Ce cours ne contient aucune information sur l'utilisation d'un périphérique particulier. Les datasheets des MCU fournissent toutes les informations nécessaires à ce propos et constituent le meilleur moyen de se former.

Quand un processeur démarre (ou *reset*), aucune fonction périphérique n'est activée.

Le développeur doit explicitement configurer et activer les services matériels nécessaires à l'application. Seulement ensuite les périphériques peuvent être utilisés.

La plupart des périphériques ont des registres de configuration à initialiser une fois, en plus de registres de travail qui contiennent les valeurs à jour en temps-réel.



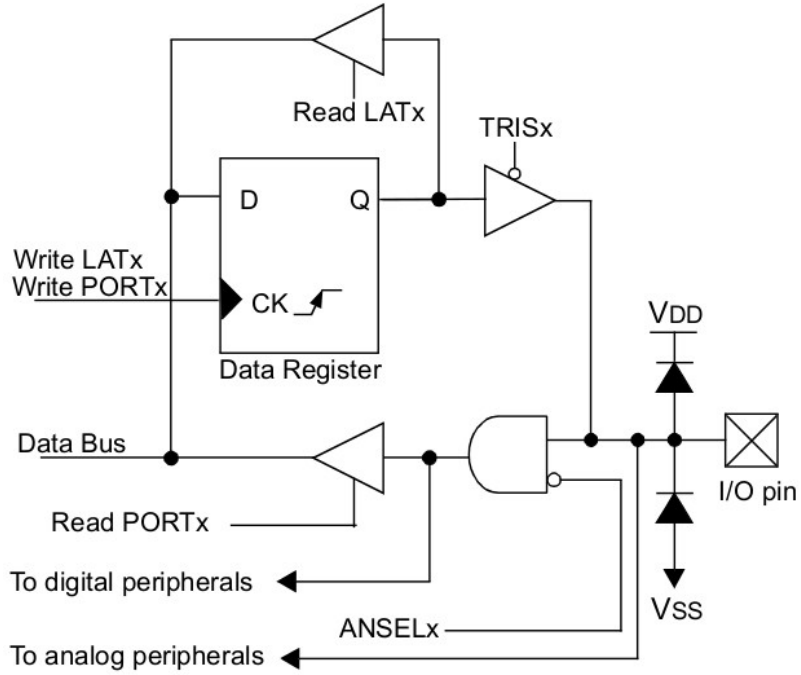

```
void main()
{

      // Peripherals configurations
      // and other initialisation
      …


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

Un port est un groupe de 8 broches, appelées GPIO (*General Purpose Input/Output*).

Ces broches sont utilisées en entrée ou en sortie, indépendamment des autres. Le PIC18F27K40 a 5 ports (port A à port E), ce qui lui confère 40 broches GPIO.

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)



Exemple de **GPIO (*General Purpose Input/Output*)** sur la carte Olimex PIC-USB-4550.

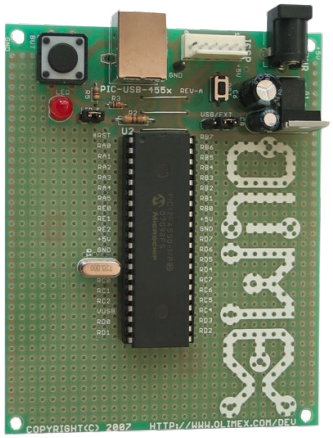
D'abord la broche RD3 est configurée en sortie, puis la valeur de sortie est mise à l'état haut ('high'). Ceci a pour effet d'allumer la LED connectée à cette broche.

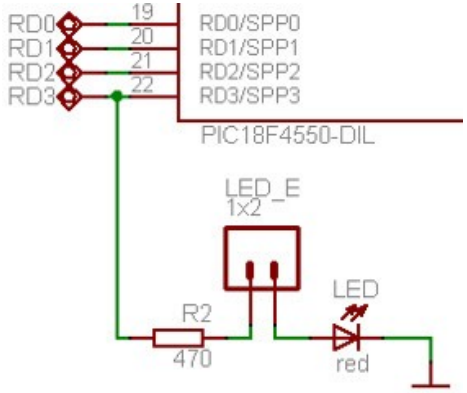
PIC18 C program

```
//Set RD3 as an output
TRISD = 0xF7;

//Set RD3 to high level
LATD = 0x04;
```

PIC18 assembly program

```
;Set RD3 as an output
MOVLW     0xF7
MOVWF     TRISD

;Set RD3 to high level
MOVLW     0x04
MOVWF     LATD
```



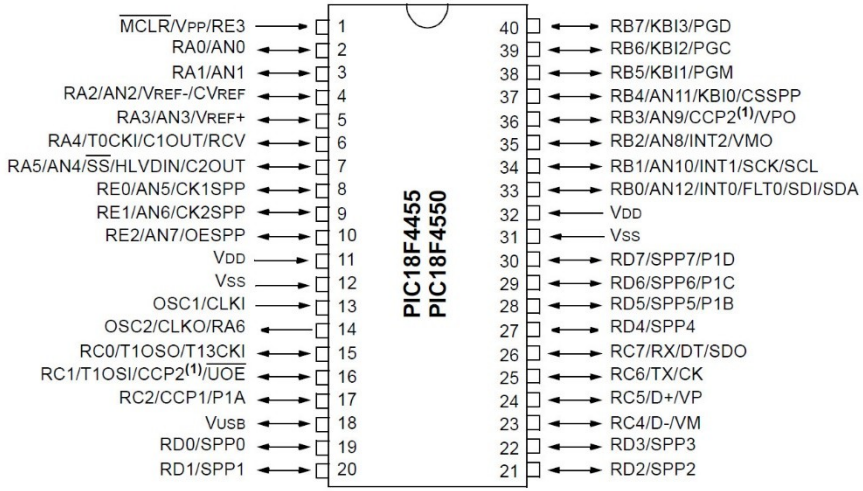


RD3 = pin 3 of port D (also bit 3 of registers associated to port D).
TRISx = register that contains the direction of the 8 pins of the port x ('0' = Output, '1' = Input).
LATx = register that contains the output value for pins configured as outputs in port x.

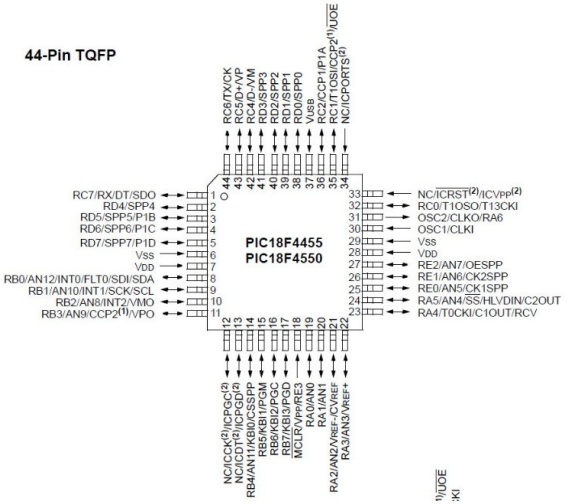
Les MCU intègrent en réalité plus de périphériques qu'ils ne sont capables de gérer à la fois.

En fait, plusieurs périphériques internes sont reliés aux mêmes broches. Ceci implique que tous les périphériques ne peuvent pas être utilisés en même temps.

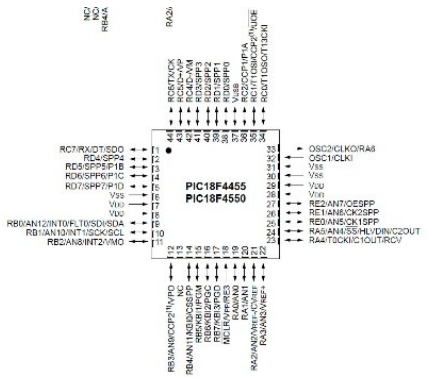
DIP package



TQFP package

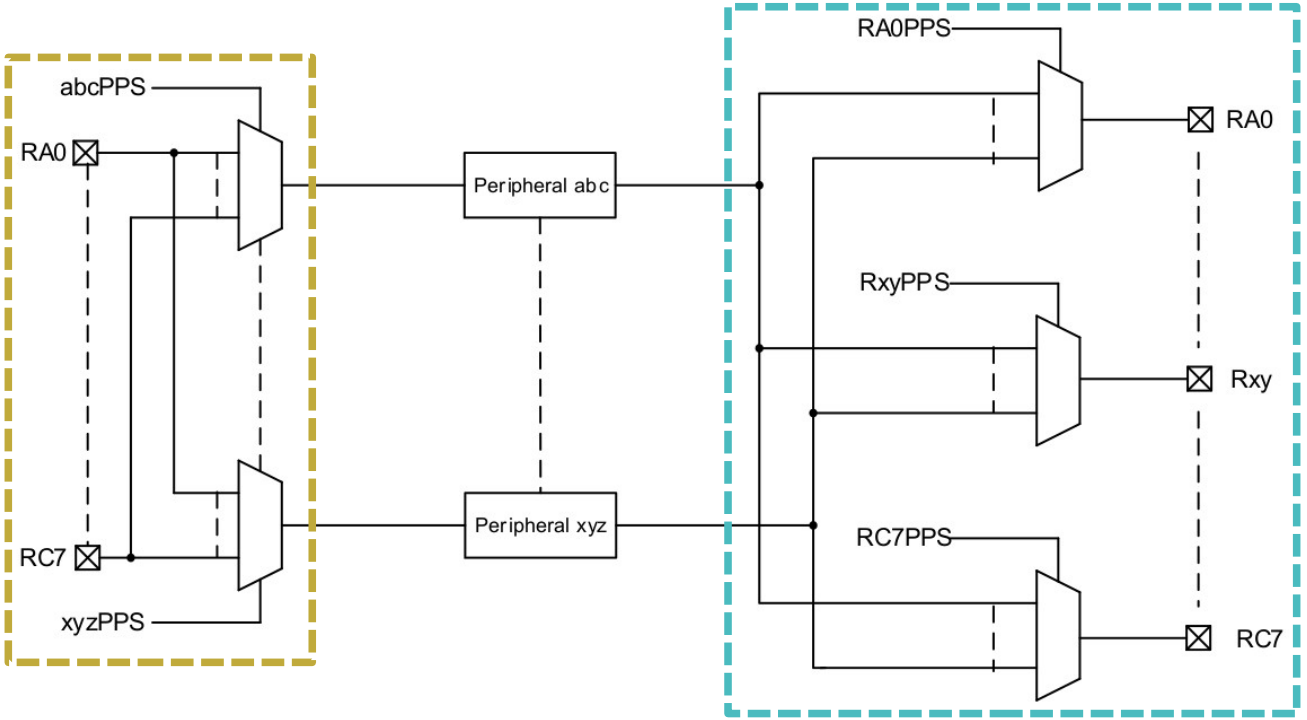


QFN package



Comme une broche peut être reliée à plusieurs périphériques (ou inversement), les connexions doivent être imposées via le module *Peripheral Pin Select* (PPS).

Input pin redirections

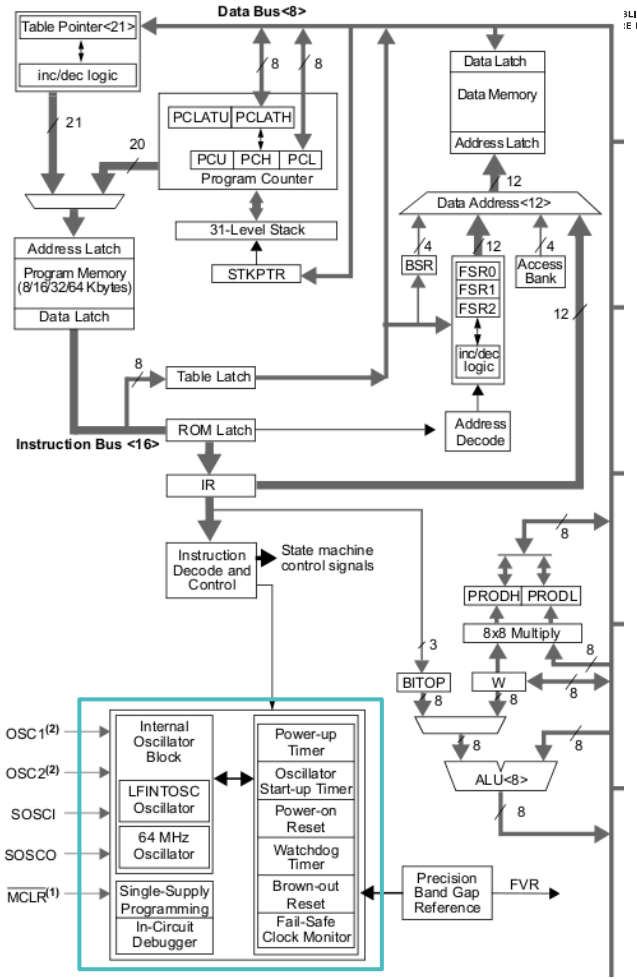


Peripheral outputs redirections

D'autres constituants du PIC18 ne sont pas des périphériques à proprement parler, puisqu'ils ne fournissent pas de services au CPU mais son plutôt garants de son bon fonctionnement.

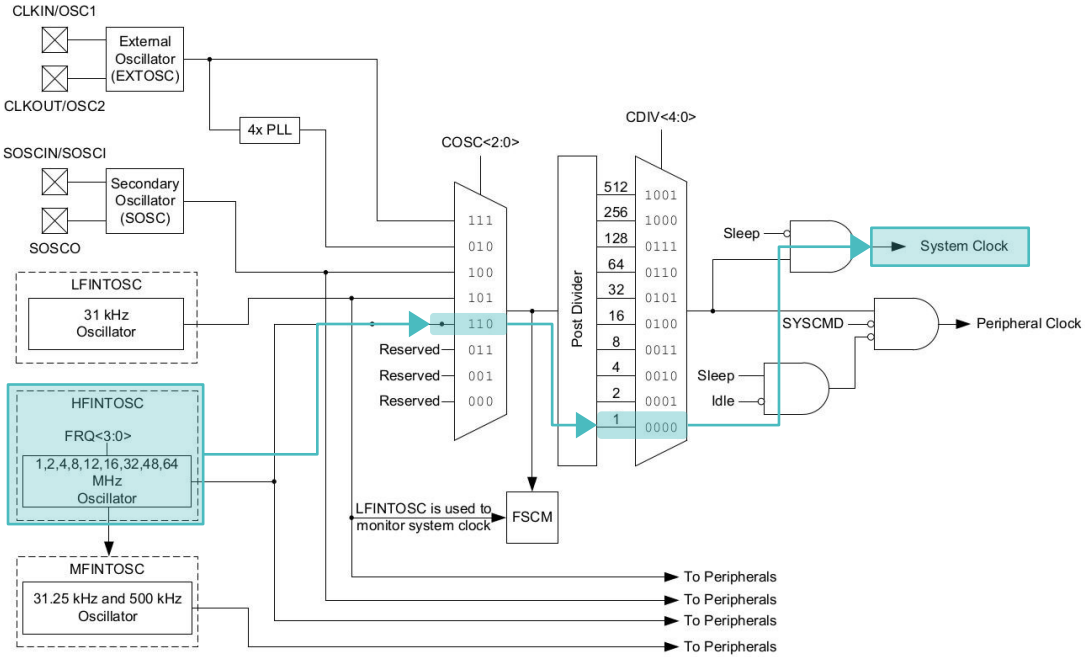
Comme tout MCU, la gamme PIC18 offre des services de configuration de de fiabilisation du système.

Rendre robuste une application consiste à la rendre moins sensible à son environnement (variations de tension, ...) voire même gérer des situations imprévues.



Horloge de référence

La référence d'horloge du système peut être réalisée par résonateur externe (quartz) ou interne (MEMS, RC, etc). Les résonateurs externes offrent une meilleure précision (dérive de quelques ppm) mais nécessitent un composant supplémentaire sur la carte.



Configuration en C PIC18

```
#pragma config FEXTOSC = OFF
#pragma config RSTOSC = HFINTOSC_64MHZ
#pragma config MCLRE = EXTMCLR
#pragma config DEBUG = OFF
```

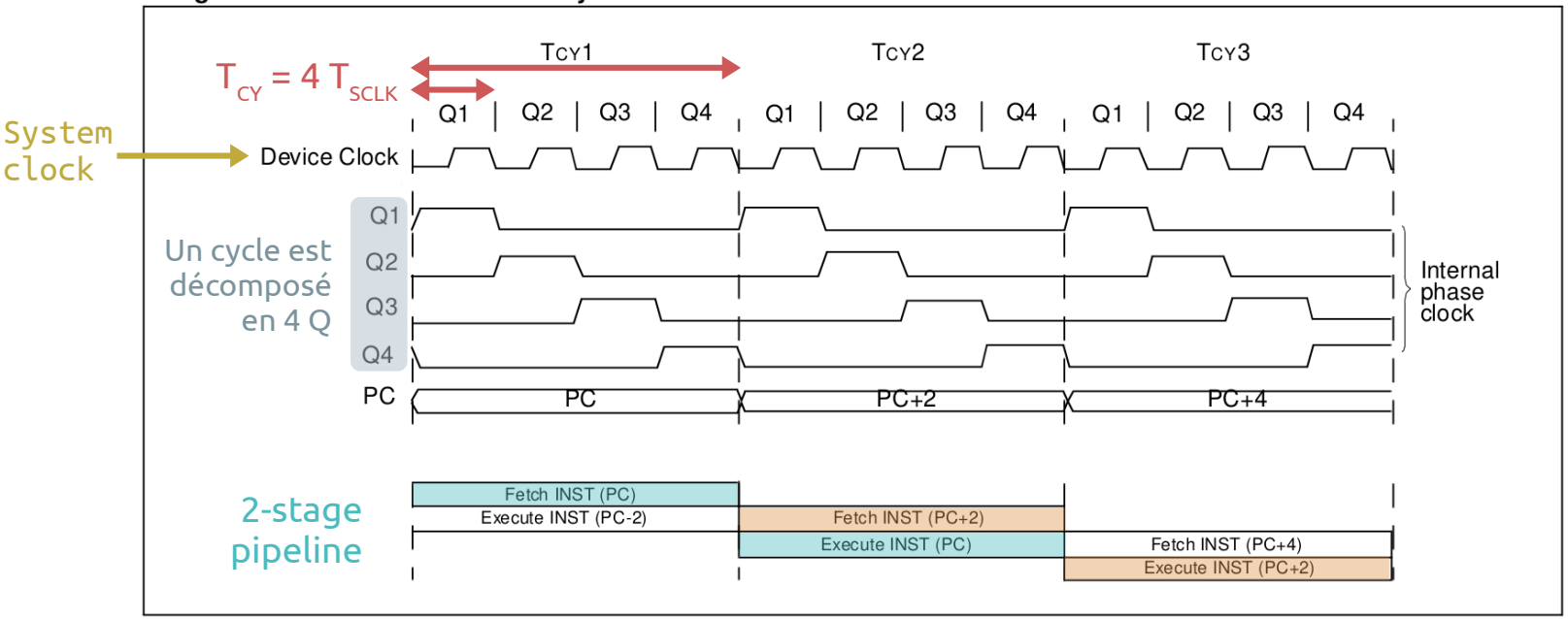
Configuration en assembleur PIC18

```
CONFIG FEXTOSC = OFF
CONFIG RSTOSC = HFINTOSC_64MHZ
CONFIG MCLRE = EXTMCLR
CONFIG DEBUG = OFF
```

Notons que l'horloge système (de période T_{SCLK}) cadence le CPU, mais celui-ci a besoin de quatre coups d'horloge pour produire une instruction.

Le temps de cycle d'une instruction T_{CY} vaut donc $4 \times T_{SCLK}$.

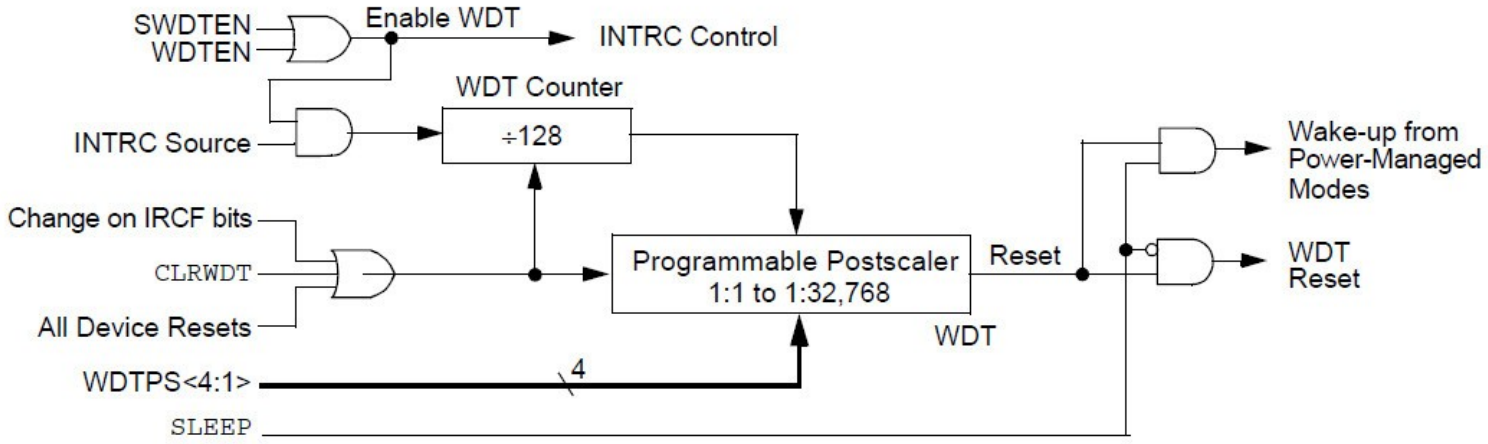
Figure 4-3: Clock/Instruction Cycle



Le *watchdog* (chien de garde) est un agent applicatif.

Il est implémenté sous forme d'un compteur de temps qui s'incrémente, et qui dans des circonstances normales est remis à 0 à intervalles réguliers (instruction `CLRWDT`).

Si l'application reste bloquée trop longtemps, le *watchdog timer* n'est pas réinitialisé et il finira par déborder. Le débordement du *watchdog timer* entraînera un *reset* du CPU.



Les micro-contrôleurs PIC18 ont deux modes de veille qui permettent de réduire la consommation du MCU. L'application doit explicitement passer en mode veille avec l'instruction SLEEP.

Dans ces deux modes, l'horloge du CPU est désactivée (pipeline à l'arrêt) et aucun instruction n'est exécutée. Les valeurs des registres et de la RAM sont conservées.

Dans le mode *Idle*, les horloges des périphériques sont toujours actives, tandis qu'en mode *Sleep* les périphériques sont également à l'arrêt.

Le processeur se réveillera avec une interruption, un reset ou un watchdog.

INTERRUPTIONS

Évènements

Interrupt Flag (IF)

Interrupt Request (IRQ)

Interrupt Service Routine (ISR)

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Interrupt flag

Une fois configuré, un périphérique travaille en autonomie.

Quand sa tâche est réalisée (compter, convertir, ...) ou **quand il détecte un évènement particulier** (message reçu, ...), **le périphérique lève un *Interrupt Flag* (IF)**.

Un *interrupt flag* est un bit dans un registre, chacun correspondant à un évènement.

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)

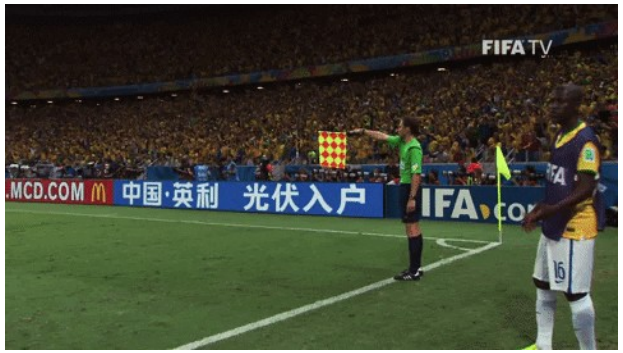
Value	Description
1	TMR0 register has overflowed (must be cleared by software)
0	TMR0 register has not overflowed

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)

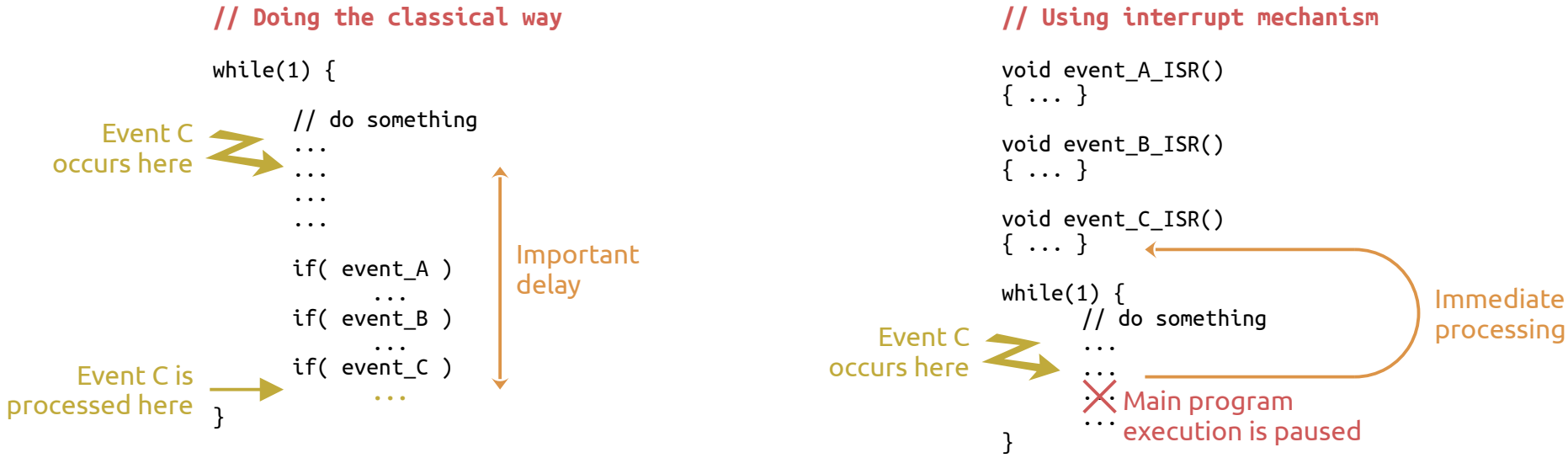
Value	Description
1	External Interrupt 'x' has occurred
0	External Interrupt 'x' has not occurred



Interrupt Service Routine (ISR)

Parfois un évènement requiert une attention immédiate, il faudrait donc surveiller les *flags* en permanence. Mais les tester (if) comme de simples variables est inefficace.

Le mécanisme d'interruption vient pallier ce problème. Il s'agit d'un processus matériel qui interrompt le fil normal normal de l'exécution, pour basculer vers l'exécution d'une fonction dédiée au traitement de l'évènement : l'**ISR *(Interrupt Service Routine)***.

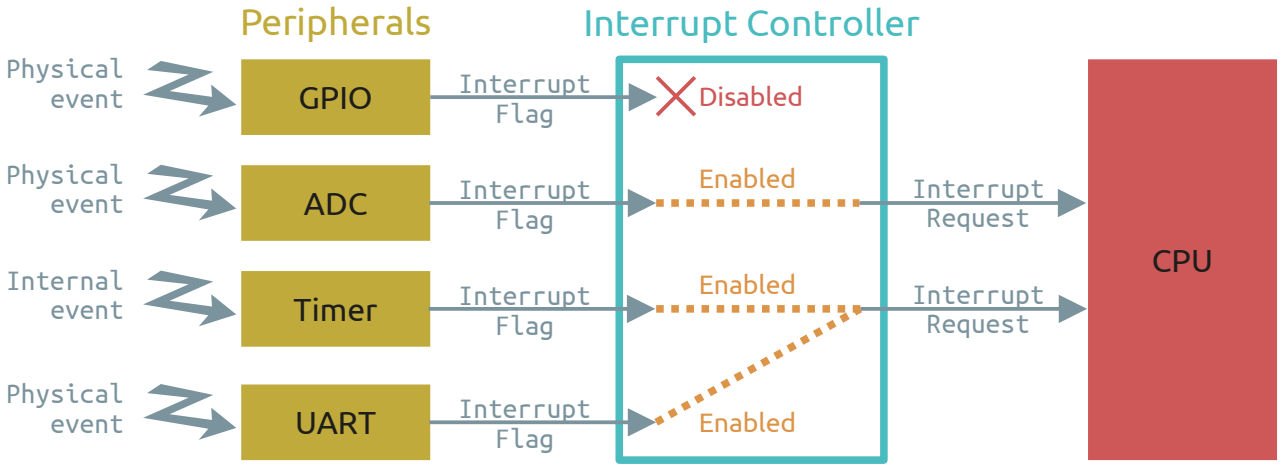


Interrupt Request (IRQ)

Toutes les interruptions sont désactivées par défaut : c'est au développeur de définir explicitement les évènement suffisamment importants pour interrompre le CPU.

Cela implique de configurer le **contrôleur d'interruptions** ou ***Interrupt controller***.

Celui-ci viendra selon sa configuration interrompre ou propager les *flags* pour les transformer en ***Interrupt Requests* (IRQ)**, qui stopperont le CPU.



Interrupt flag
Bit dans un registre qui ***indique*** qu'un évènement s'est produit.

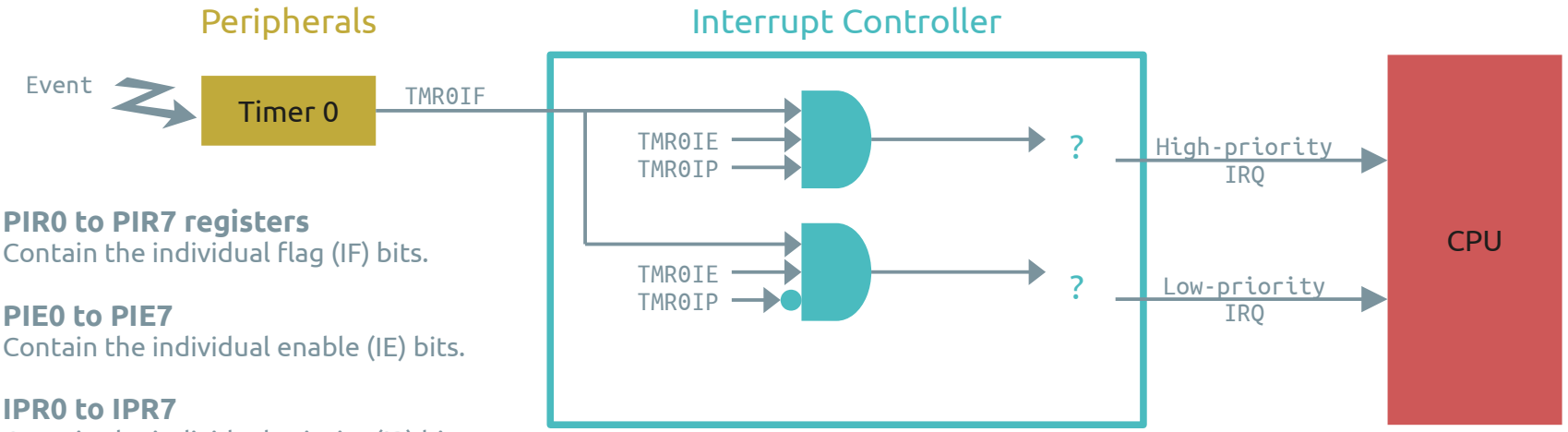
Interrupt request
Signal (fil) physiquement relié au CPU et qui ***le fait basculer vers une autre fonction***.

Notes
Plusieurs drapeaux peuvent être branchés vers le même signal IRQ.
Une IRQ correspond précisément à une ISR.

L'**interrupt controller** est fait de portes AND et OR. Les bits en entrée de ces portes permet d'arrêter ou de poursuivre la propagation de l'interrupt flags jusqu'à l'IRQ.

Comme de nombreux micro-contrôleurs, le PIC18 s'appuie sur trois bits de configuration pour chaque source d'interruption :

Le bit *Interrupt Flag* (IF), le bit *Interrupt Enable* (IE), le bit *Interrupt Priority* (IP).



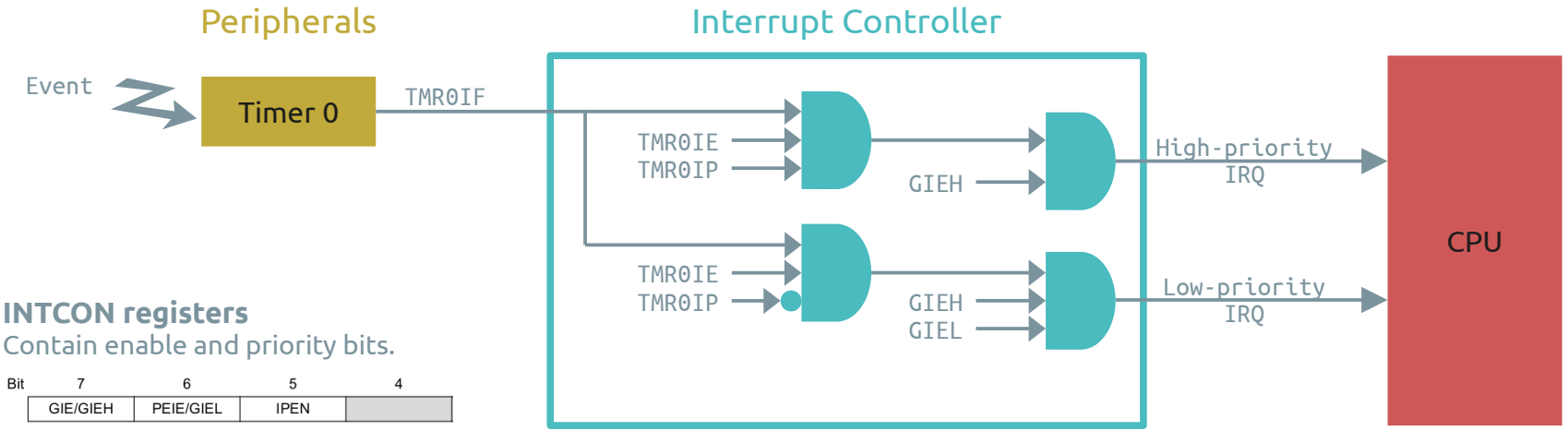
PIR0 to PIR7 registers
Contain the individual flag (IF) bits.

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

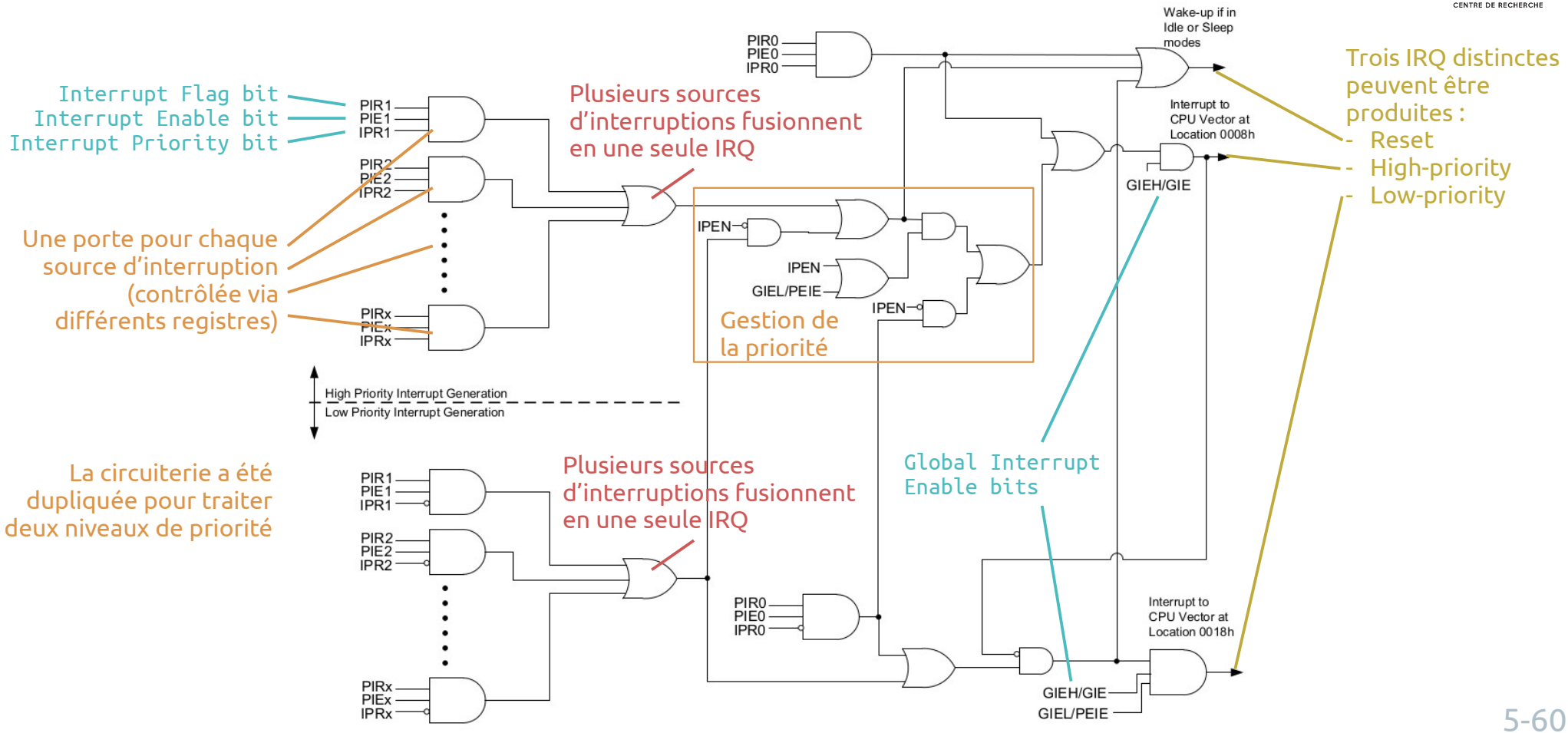
IPR0 to IPR7
Contain the individual priority (IP) bits.

Après avoir configuré l'*interrupt controller* et ainsi autorisé les interruptions depuis les périphériques souhaités uniquement, il reste à donner l'autorisation au CPU de s'arrêter pour traiter ces requêtes.

La plupart des MCU ont un bit *Global Interrupt Enable* (GIE) pour remplir ce rôle. Le PIC18 en a même deux : GIEH et GIEL pour les interruptions haute et basse priorité.



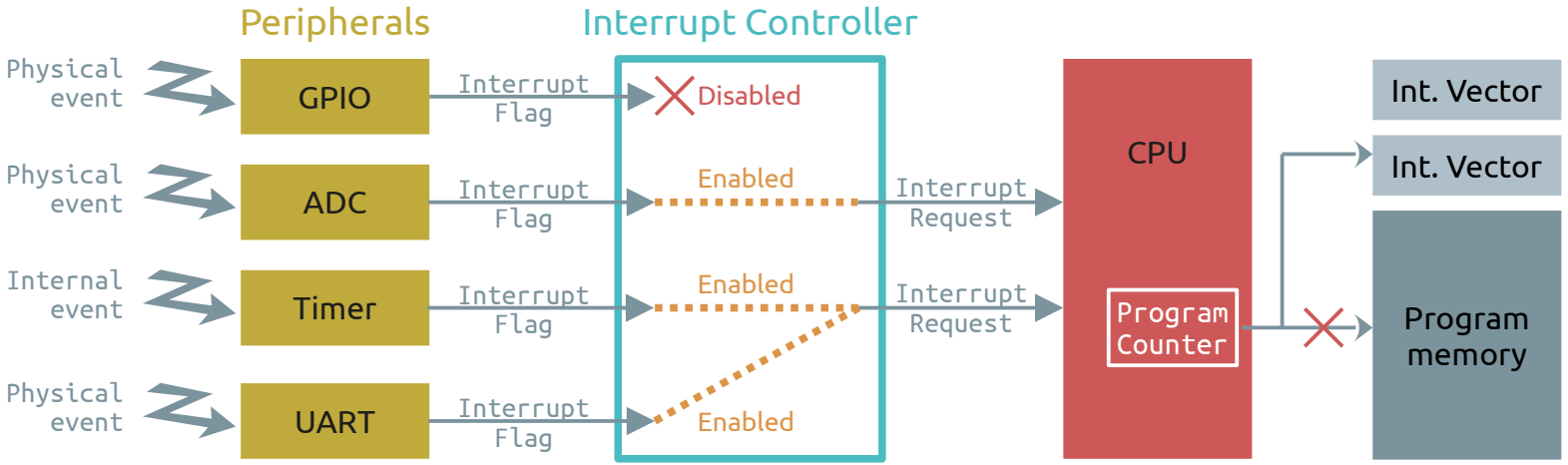
Interrupt controller: logique du contrôleur d'interruption du PIC18F27/47K40

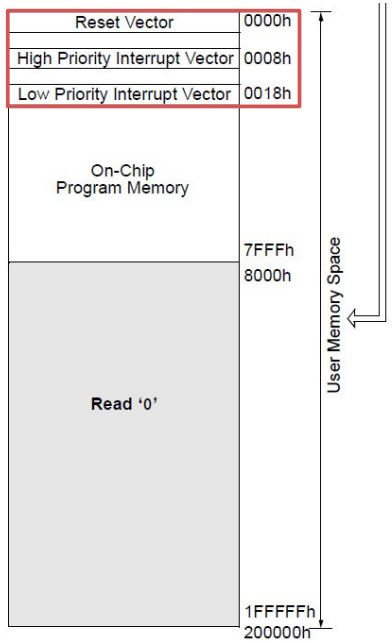


Interrupt Service Routine (ISR)

Les *Interrupt Service Routines* (ISR) sont des fonctions appelées par les ***Interrupt vectors***, qui sont dans une zone précise de la mémoire Flash.

Quand le CPU voit une ***Interrupt Request* (IRQ)**, le *Program Counter* se branche automatiquement sur le vecteur d'interruption correspondant, provoquant ainsi l'appel de la routine d'interruption.





Les ISR ne sont pas des fonctions normales :

elles ne doivent pas être appelées depuis le programme principal.

Elles sont automatiquement appelées quand le CPU voit une IRQ. Ce paradigme de programmation s'appelle « programmation évènementielle » (*event-driven progr.*).

Comme une ISR est appelée à un moment inconnu, il est impossible de lui passer des arguments, ni obtenir une valeur de retour.

Pour échanger des informations entre l'application principale et les ISR, nous utilisons des variables globales. Mais attention, les variables globales sont ressources partagées et doivent être employées avec précaution !

Voici un exemple d'écriture d'ISR en PIC18

```
 /*  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 appelée pour toute interruption de priorité haute

On vérifie si le Timer 0 est la source de l'IRQ

On efface alors son flag, puis on traite l'évènement

On vérifie si l'ADC0 est la source de l'IRQ

On efface alors son flag, puis on traite l'évènement

Configure le timer 0 (y compris autoriser son interruption)

Autorise les interruptions pour le CPU

Routine principale

Interrupt vector

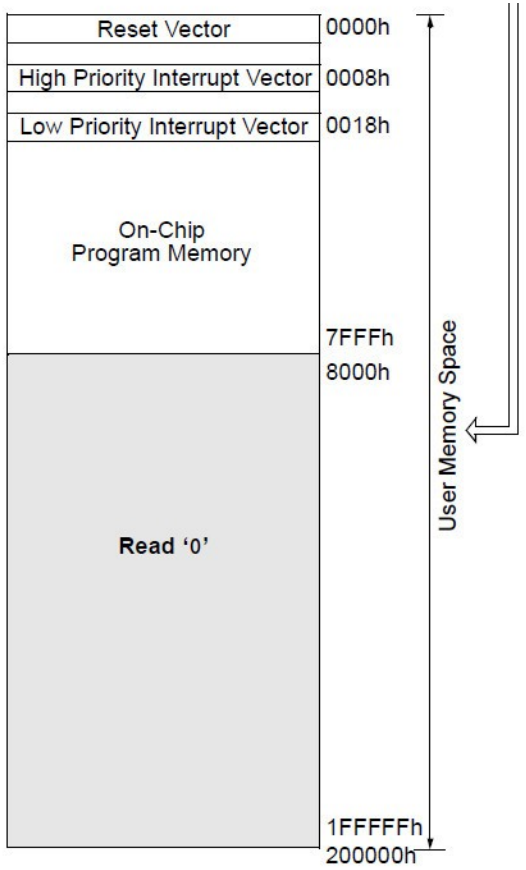
Les vecteurs d'interruption (*Interrupt vectors)* sont de très petites zones en mémoire programme.

→ 16 octets pour le vecteur d'interruption haute priorité

→ 2 octets pour le vecteur d'interruption basse priorité

Ces espaces sont trop petits pour contenir le code de l'ISR, mais assez grand pour contenir une instruction `CALL` ou `GOTO` pour appeler cette ISR (voir page suivante).

Les fonctions ISR sont donc techniquement stockées dans la mémoire programme, mais ne doivent être accédées que via les vecteurs d'interruption.



Changement de contexte

Pouvant être appelée n'importe quand et ayant son propre traitement à effectuer, une ISR écrasera le **contexte** de la fonction interrompue (valeurs de `W-reg`, `STATUS`, `BSR`). L'ISR devra donc se charger au départ de sauvegarder le contexte de la fonction interrompue, afin de lui restituer en fin d'ISR.

La chaîne de compilation XC8 implémente une sauvegarde matérielle du contexte pour les interruptions haute priorité et une sauvegarde logicielle pour les basse priorité.

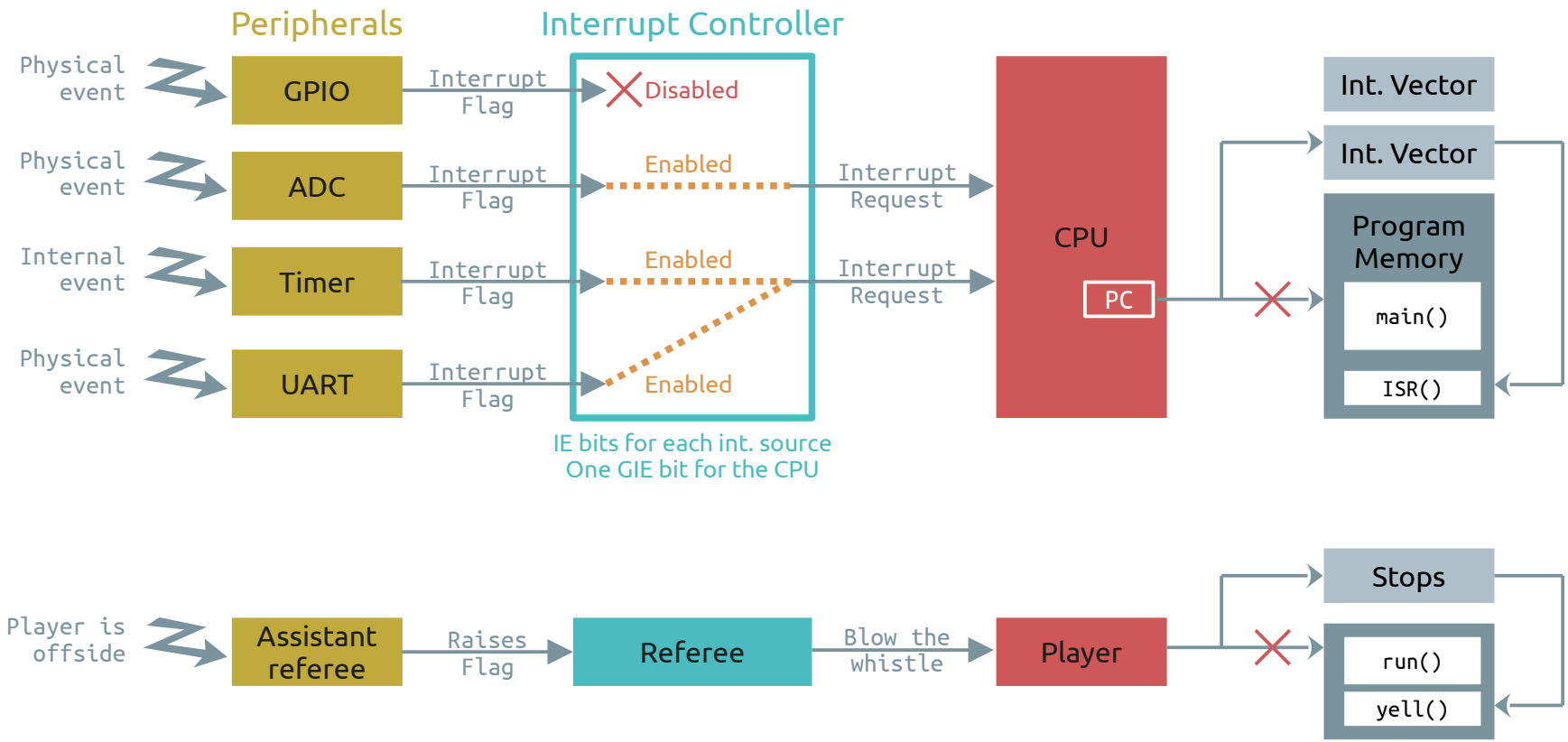
Hardware context backup (CPU shadow registers)

```
high_vector:
      CALL  high_isr, 1

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

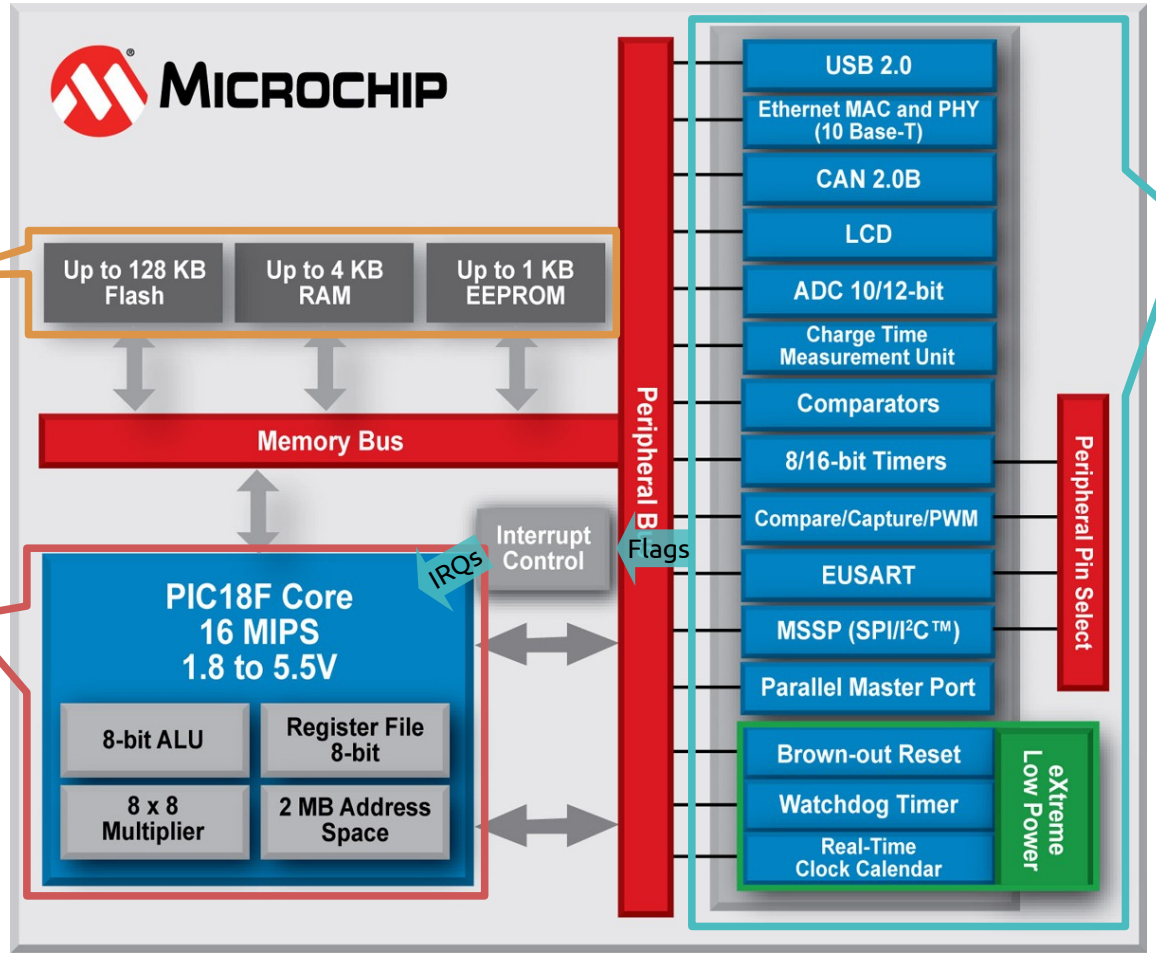
Software context backup (Flash memory)

```
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
```

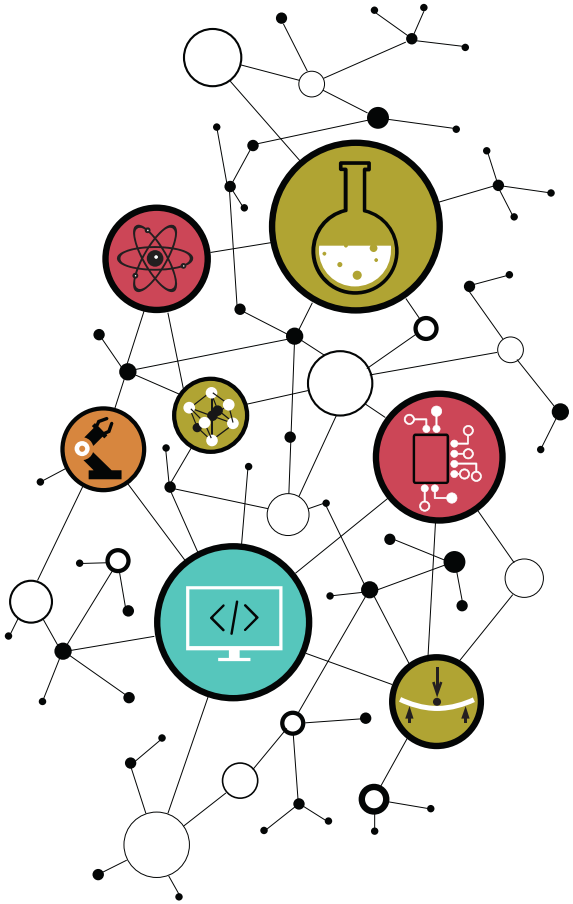
En résumé


Peripherals
Interrupt Controller
Physical event
GPIO
Interrupt Flag
Disabled
Physical event
ADC
Interrupt Flag
Enabled
Internal event
Timer
Interrupt Flag
Enabled
Physical event
UART
Interrupt Flag
Enabled
IE bits for each int. source
One GIE bit for the CPU
Interrupt Request
Interrupt Request
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



CONTACT



Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)