Informatique Industrielle

Cours Master SIS

Micro-contrôleurs Microchip

Intervenants :

Marc Allain - marc.allain@fresnel.fr

Julien Marot - julien.marot@fresnel.fr

Coordonnées

- **Marc Allain** [Maître de conférence]
  marc.allain@fresnel.fr
  Equipe physique et traitement d'image,
  Institut Fresnel, bureau 215

- **Julien Marot** [Maître de conférence]
  julien.marot@fresnel.fr
  Equipe Groupe Signaux Multidimensionnels
  Institut Fresnel, bureau 237

Note : l*es intervenants sont sur le domaine Universitaire de St-Jérôme.*

Organisation de l'enseignement

Contenu horaire :

- **20 h de cours + 10 h de TD** [Julien Marot] (≃10x 3h)

Présentation de l'informatique industrielle, des systèmes micro-programmés (architecture, principes généraux, ...). Étude d'un micro-contrôleur Microchip PIC 18F4520. Programmation en langage Assembleur et langage C.

- **35 h de travaux pratiques** [Allain & Marot] (12x 4h)

Mise en pratique des connaissances sur la carte de démonstration PICDEM2 plus. Utilisation du micro-contrôleur Microchip PIC 18F4520.

Merci d'être à l'heure en cours / TP !

Contrôle des connaissances

Vous êtes principalement évalués sur la base des TP (15 points/20)

(1) Avant toute chose, vous devez rédiger un **algorigramme**,

(2) les programmes écrits doivent être **commentés**,

(3) **vérification des programmes** en simulation et sur carte d'essai,

(4) **chaque étudiant sera noté individuellement** ; nous évaluerons la participation de chacun au sein d'un binôme constitué.

Examen (5 points/20) : contrôle des connaissances avec poly de cours, sans calculatrice, sur les notions vues en cours (exercices inclus) et en TP.

« Boite à outils »

- **Les différentes bases de numérotation**
  (*binaire, octal, décimal, hexadécimal*)

- **Conversions et opérations sur les nombres binaires**

- **Notions d'électronique numérique**
  (*fonctions logiques combinatoires et séquentielles*)

- **Notion de programmation**
  (*algorigramme, concept de variable, fonction*, etc.)

Objectifs du cours

L'objectif de ce cours est de vous rendre capable de choisir, de programmer, d'utiliser un micro-contrôleur et plus généralement de vous transmettre une culture des systèmes micro-programmés.

Non dédié à un microcontrôleur

- *Notions d'architecture* [des systèmes micro-programmés]
- *Éléments constitutifs* [d'un système micro-programmé]
- *Fonctionnement* [d'un système micro-programmé]
- *Éléments de choix* [d'un système micro-programmé]

Dédié à un microcontrôleur

- Connaissances des différents types d'instruction
- Notion d'interruption
- Programmation en Assembleur
- Programmation en langage C

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées au microcontrôleur (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

L'informatique industrielle

« L'informatique industrielle est une branche de l'informatique appliquée qui couvre l'ensemble des techniques de conception et de programmation, de systèmes informatisés à vocation industrielle, qui ne sont pas des ordinateurs. »
(Source : Wikipédia)

Source : Ascom S.A.

L'informatique industrielle

Domaines d'applications :

Alarme, automobile, aviation, instrumentation, médicale, téléphonie mobile, terminaux de paiement pour carte bancaire ...

Image fournie par Microchip

L'informatique industrielle

Applications :

- Automates, robotique,
- Mesures de grandeurs physiques,
- Systèmes temps-réel,
- Systèmes embarqués.

Source : Ascom S.A.

Les différents systèmes programmables

- **Les circuits spécialisés ou ASIC (Application Specific Integrated Circuit) :**

  Les circuits spécialisés sont des circuits spécialisés dès leur conception pour une application donnée.

  Exemples : DSP (*Digital Signal Processing*), co-processeur arithmétique, processeur 3-D, contrôleur de bus, ...

Source : Texas Instruments

Source : NVidia

Avantages :

- Très rapide
- Consommation moindre
- Optimisé pour une application

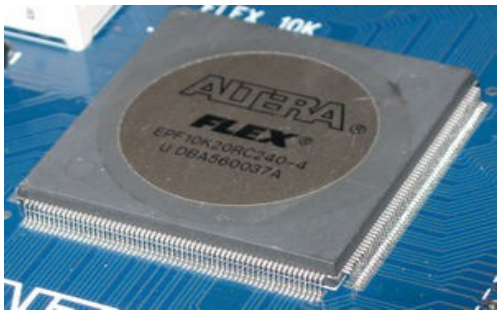
Inconvénients :

- Faible modularité
- Possibilité d'évolution limité
- Coût

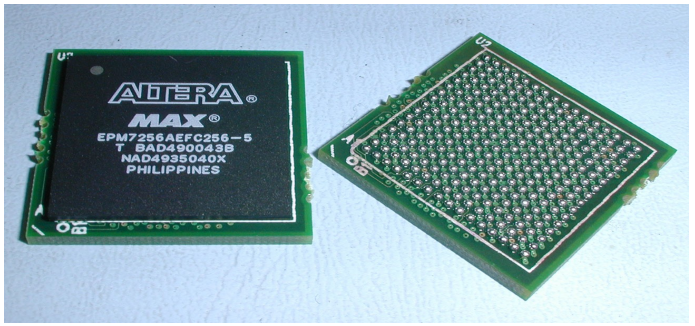
Les différents systèmes programmables

Les systèmes en logique programmée et/ou en logique programmable sont connus sous la désignation de PLD (*programmable logic device, circuit logique programmable*)

- **FPGA** (*field-programmable gate array, réseau de portes programmables in-situ*),
- **PAL** (*programmable array logic, réseau logique programmable*),
- ...



Source : Altera



Source : Altera

« Un circuit logique programmable, ou réseau logique programmable, est un circuit intégré logique qui peut être reprogrammé après sa fabrication. Il est composé de nombreuses cellules logiques élémentaires pouvant être librement assemblé.» (Wikipédia)

Avantages :

- Forte modularité
- Rapidité

Inconvénients :

- Mise en oeuvre plus complexe
- Coûts de développement élevé

Les différents systèmes programmables

- **Les systèmes micro-programmés :**

Les micro-contrôleurs sont typiquement des systèmes micro-programmés.

Micro-contrôleur Microchip
PIC16F690 en boîtier DIL20

Un **micro-contrôleur** est un :

« Circuit intégré comprenant essentiellement un microprocesseur, ses mémoires, et des éléments personnalisés selon l'application. » (Arrêté français du 14 septembre 1990 relatif à la terminologie des composants électroniques.)

Un micro-contrôleur contient un microprocesseur.

Avantages :

- Mise en oeuvre simple
- Coûts de développement réduits

Inconvénients :

- Plus lent
- Utilisation sous optimale

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Deux types de processeurs

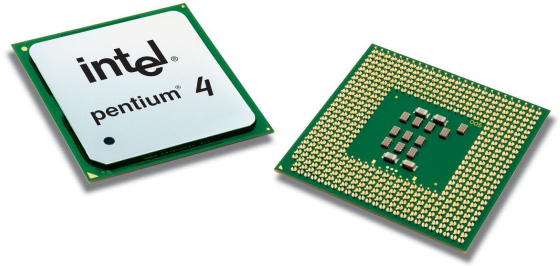
- **CISC** : *Complex Instruction Set Computer*

  Grand nombre d'instructions,
  Type de processeur le plus répandu

- **RISC** : *Reduced Instruction Set Computer*

  Nombre d'instructions réduit
  (sélection des instructions pour une exécution plus rapide)
  Décodage des instructions plus rapide

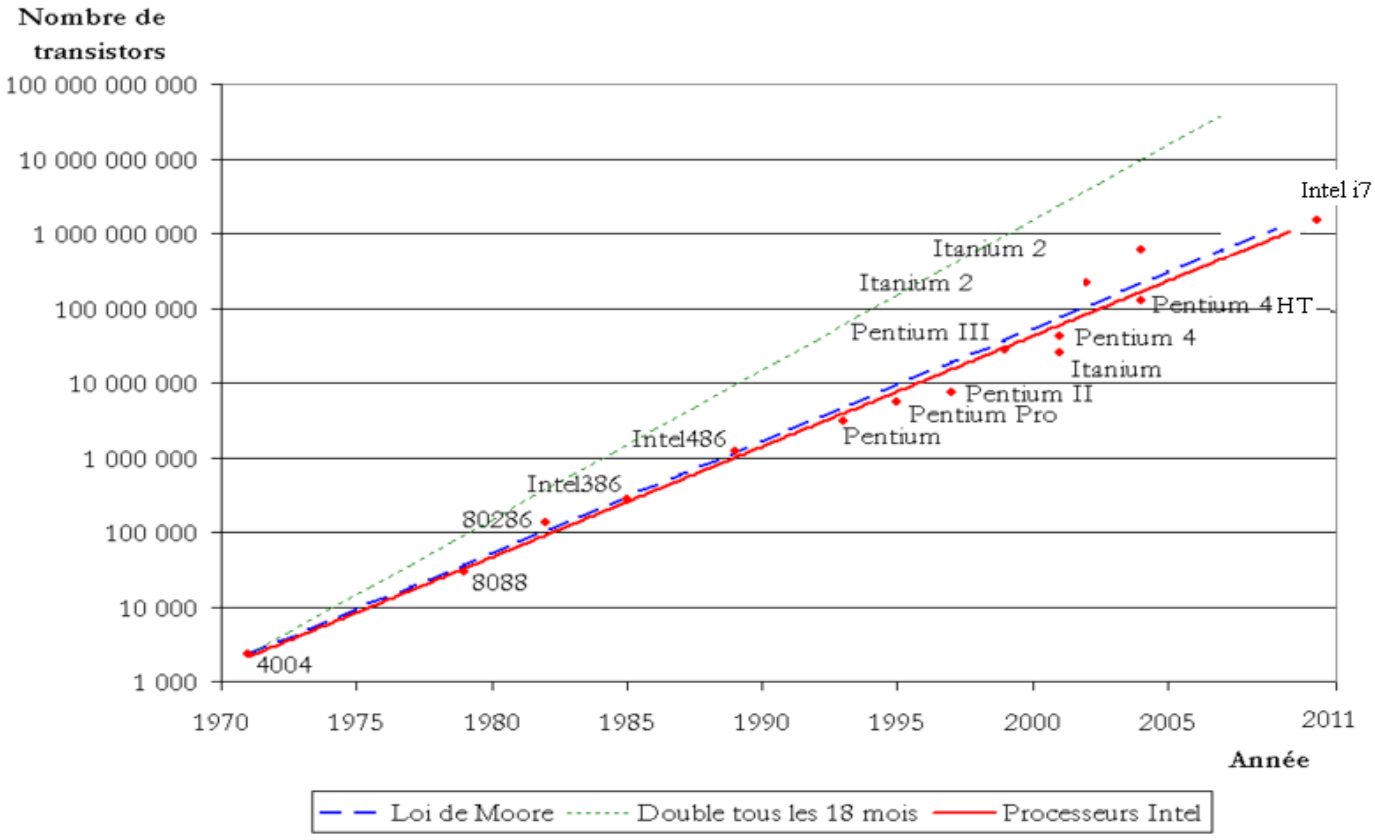
Évolution et Loi de Moore



Source : Intel

Intel Pentium 4 Northwood C (2002)
architecture interne 32 bits
fréquence d'horloge 2,4/3,4 Ghz
(bus processeur : 200Mhz)
plus de 42 millions de transistors, gravés en 0,13 µm
450 MIPS

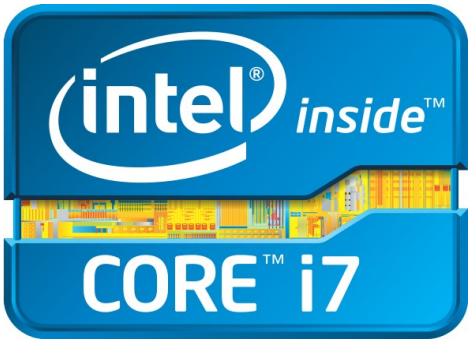
Source : Intel



Source : Wikipédia

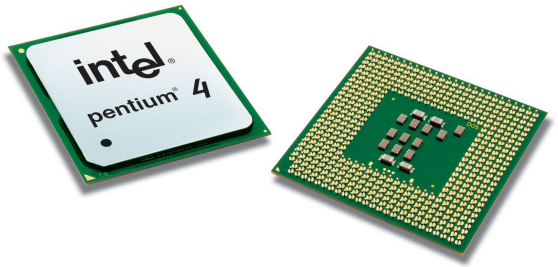
Intel 8086 (1978)
architecture interne 16 bits
bus 16 bits
fréquence d'horloge 4,77/10 Mhz
39 000 transistors, gravés en 3µm
0,33/0,75 MIPS

Évolution et Loi de Moore



Intel Core i7 Gulftown (2011)
architecture interne 64 bits
4/6 coeurs
fréquence d'horloge 3,46 Ghz
Fréquence de bus: 3,2 GHz
Fréquence de transfert des données 25.6 Gb/sec.
1,17 Milliards de transistors, gravés en 32nm
6000 MIPS

« The wall » :
limite industrielle et physique,
20 nm
→
performance / Watt consommé



Intel Pentium 4 Northwood C (2002)
architecture interne 32 bits
fréquence d'horloge 2,4/3,4 Ghz
Fréquence de bus: 0,2 GHz
plus de 42 millions de transistors, gravés en 0,13 µm
450 MIPS

Source : Wikipédia

Les structures des systèmes micro-programmés

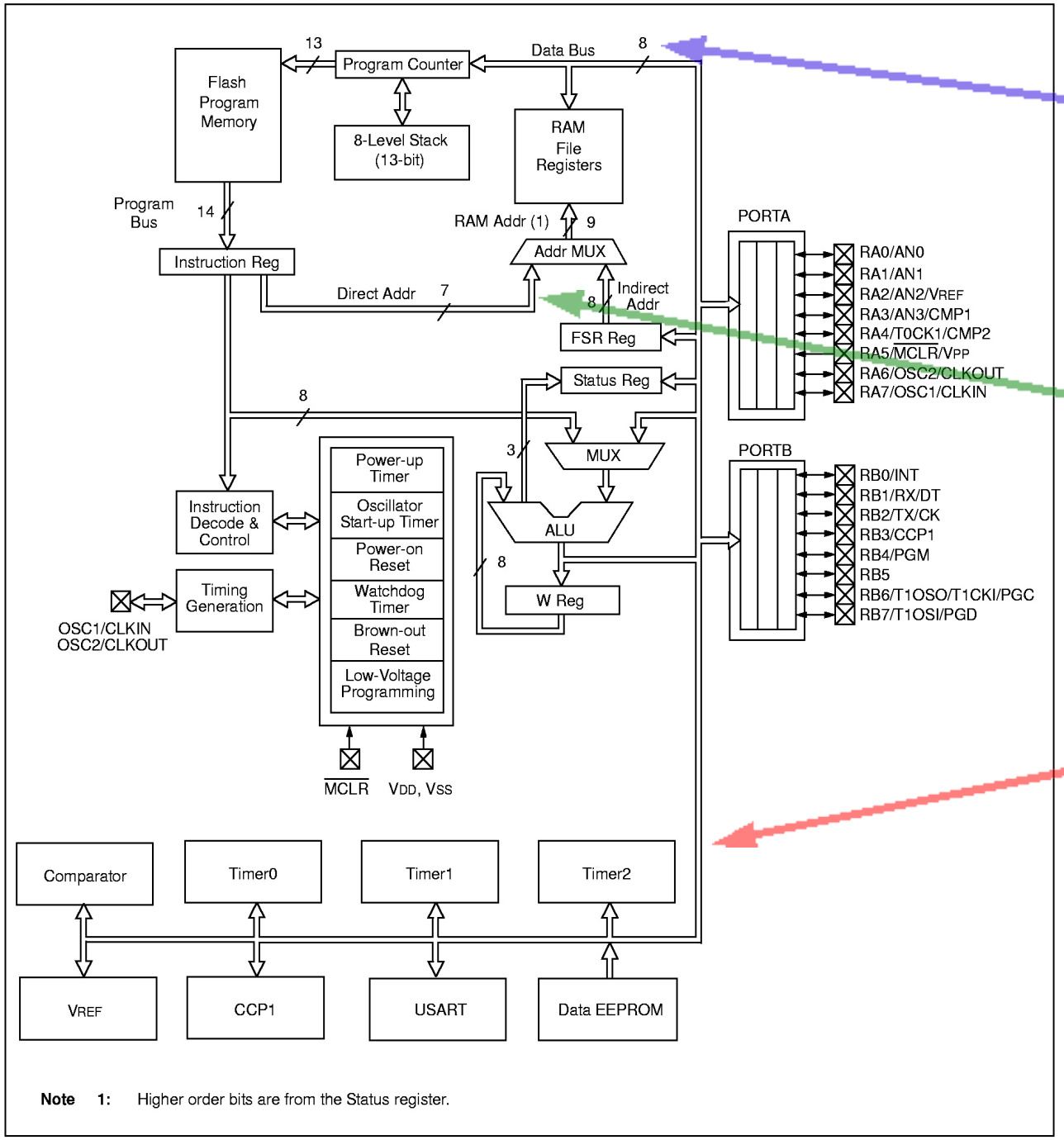
- **Les différents bus d'un système micro-programmés**

« Un bus est un jeu de lignes partagées pour l'échange de mots numériques. »
(Traité de l'électronique, Paul Horowitz & Winfield Hill)

Définition : Un bus permet de faire transiter (liaison série/parallèle) des informations codées en binaire entre deux points. Typiquement les informations sont regroupés en mots : octet (8 bits), word (16 bits) ou double word (32 bits).

Caractéristiques d'un bus:
- nombres de lignes,
- fréquence de transfert.

FIGURE 3-1: BLOCK DIAGRAM



Note 1: Higher order bits are from the Status register.

- **« Largeur du bus »**

8

- **Unidirectionnel**

- **Bidirectionnel**

Issu de la documentation technique du PIC16F628

Structures des systèmes micro-programmés

Il existe 3 Types de bus :

- **Bus de données** : permet de transférer entre composants des données,
  ex. : *résultat d'une opération, valeur d'une variable, etc.*

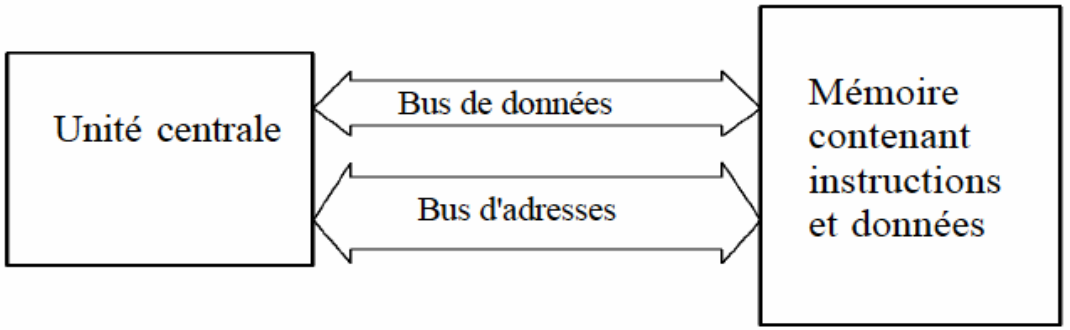
- **Bus d'adresses** : permet de transférer entre composants des adresses,
  ex. : *adresse d'une case mémoire, etc.*

- **Bus de contrôle** : permet l'échange entre les composants d'informations de contrôle [bus rarement représenté sur les schémas].
  ex. : *périphérique prêt/occupé, erreur/exécution réussie, etc.*

Définition : Une **adresse** est un nombre binaire qui indique un emplacement dans une zone mémoire

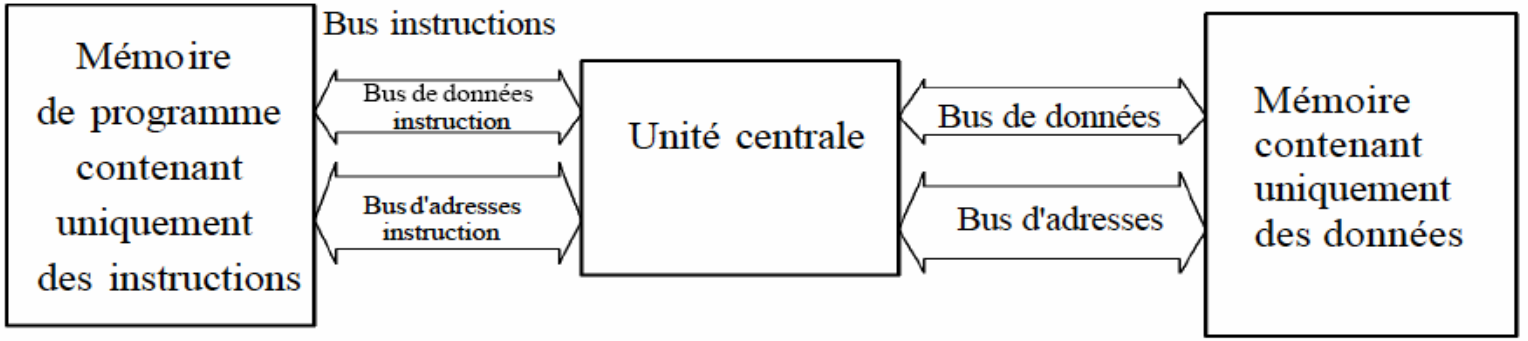
Structures des systèmes micro-programmés

- **Structure de Von Neumann**



Extraits du cours intitulé « Les systèmes micro-programmés »

- **Structure de Harvard**



La différence se situe au niveau de la séparation ou non des mémoires programmes et données. La structure de Harvard permet de transférer données et instruction simultanément, ce qui permet un gain de performances.

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

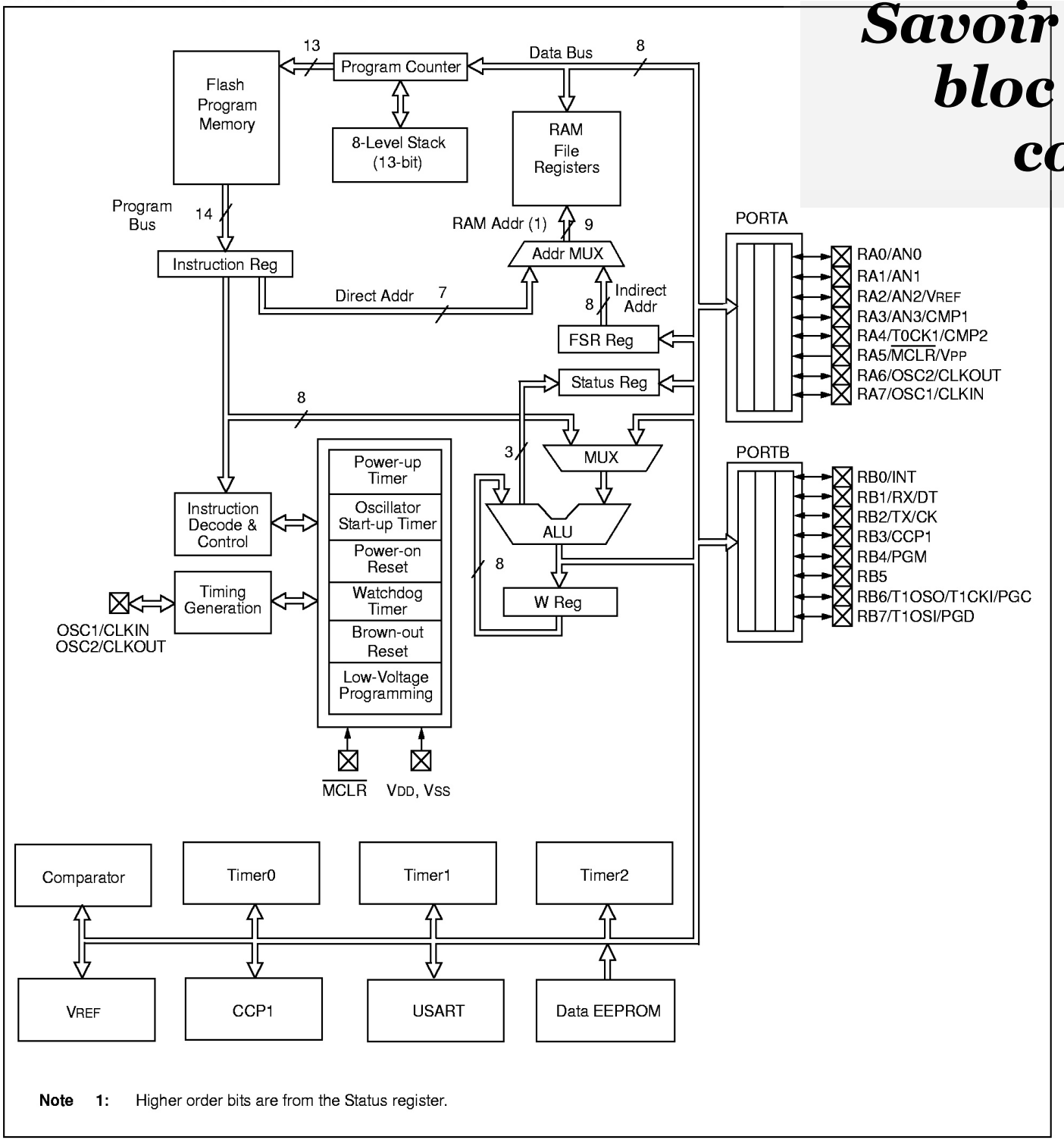
Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

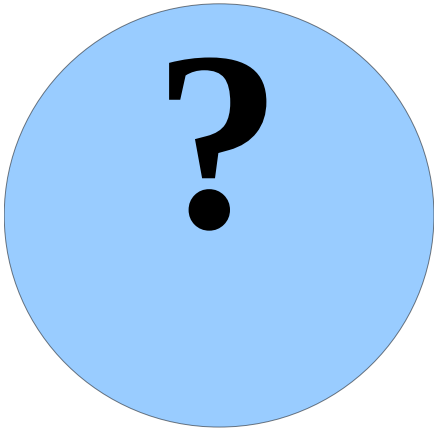
Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Savoir lire le schéma bloc d'un micro-contrôleur

FIGURE 3-1: BLOCK DIAGRAM

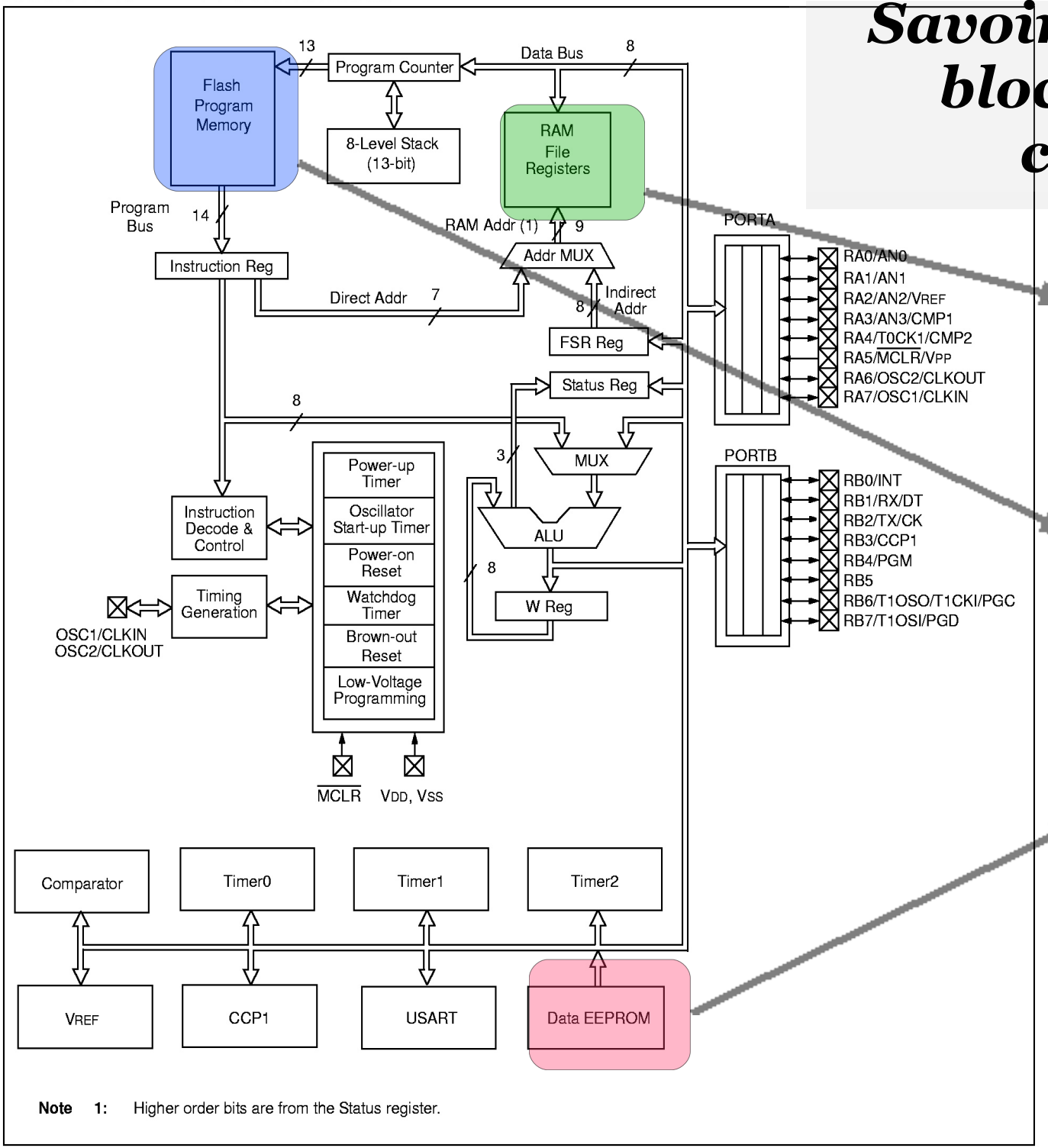


Note 1: Higher order bits are from the Status register.



Issu de la documentation technique du PIC16F628

FIGURE 3-1: BLOCK DIAGRAM



Note 1: Higher order bits are from the Status register.

Savoir lire le schéma bloc d'un micro-contrôleur

Les mémoires :

RAM (Random Access Mem.)

mémoire rapide qui permet de stocker temporairement des données.

ROM (Read Only Memory)

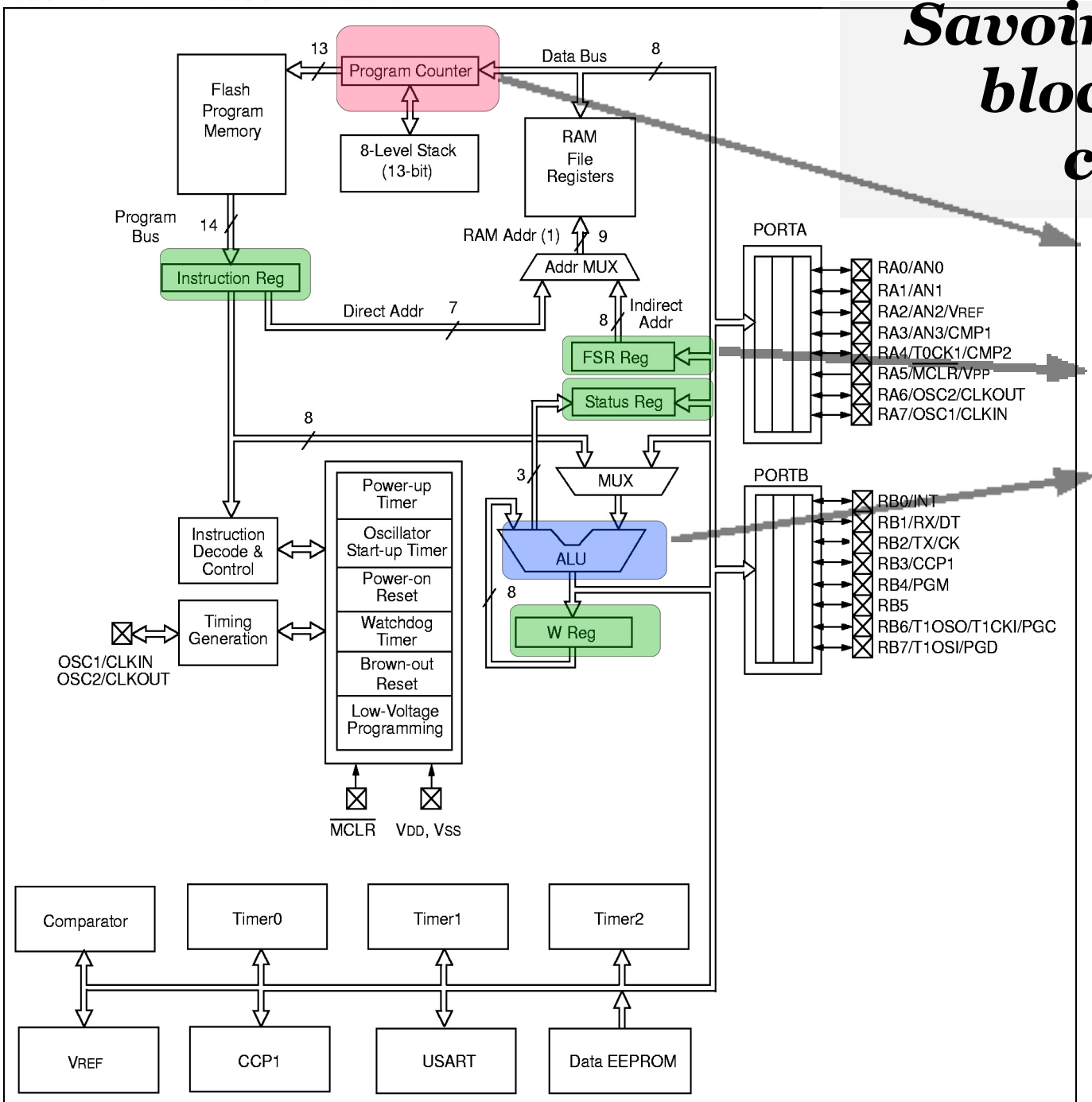
mémoire à lecture seule, programmée à vie.

EEPROM

(Elec. Erasable Programmable Read Only Memory)

mémoire lente qui permet de stocker des données même après coupure de l'alim.

FIGURE 3-1: BLOCK DIAGRAM

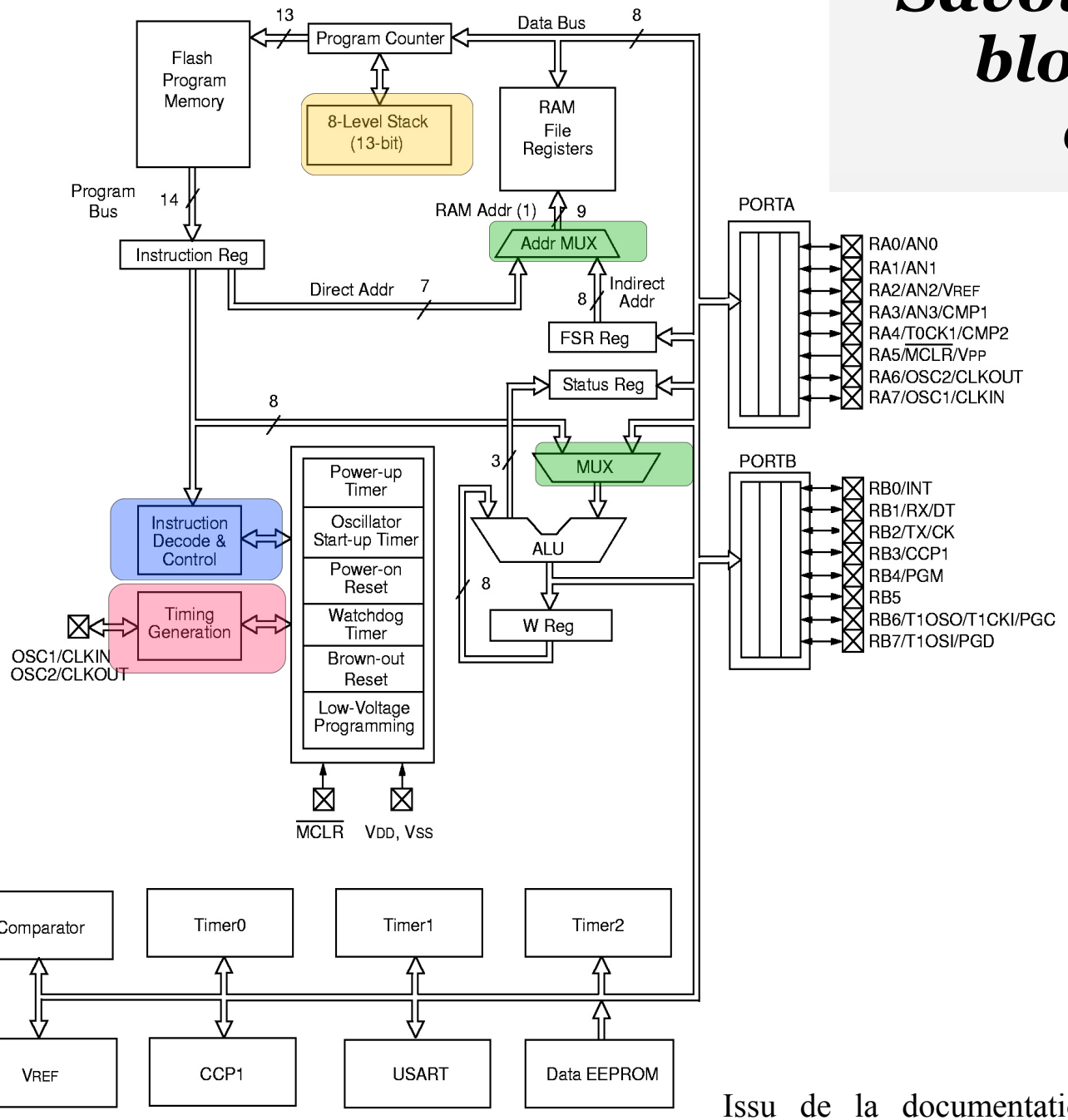


Note 1: Higher order bits are from the Status register.

Savoir lire le schéma bloc d'un micro-contrôleur

- **PC** (Program Counter)
- **Registre**(case mémoire)
- **ALU**
- **Multiplexeur**
- **Décodeur d'instructions**
- **horloge**
- **Stack (pile)**
  - *LIFO (Last In First Out)*
  - *FIFO (First In First Out)*

FIGURE 3-1: BLOCK DIAGRAM



Note 1: Higher order bits are from the Status register.

Savoir lire le schéma bloc d'un micro-contrôleur

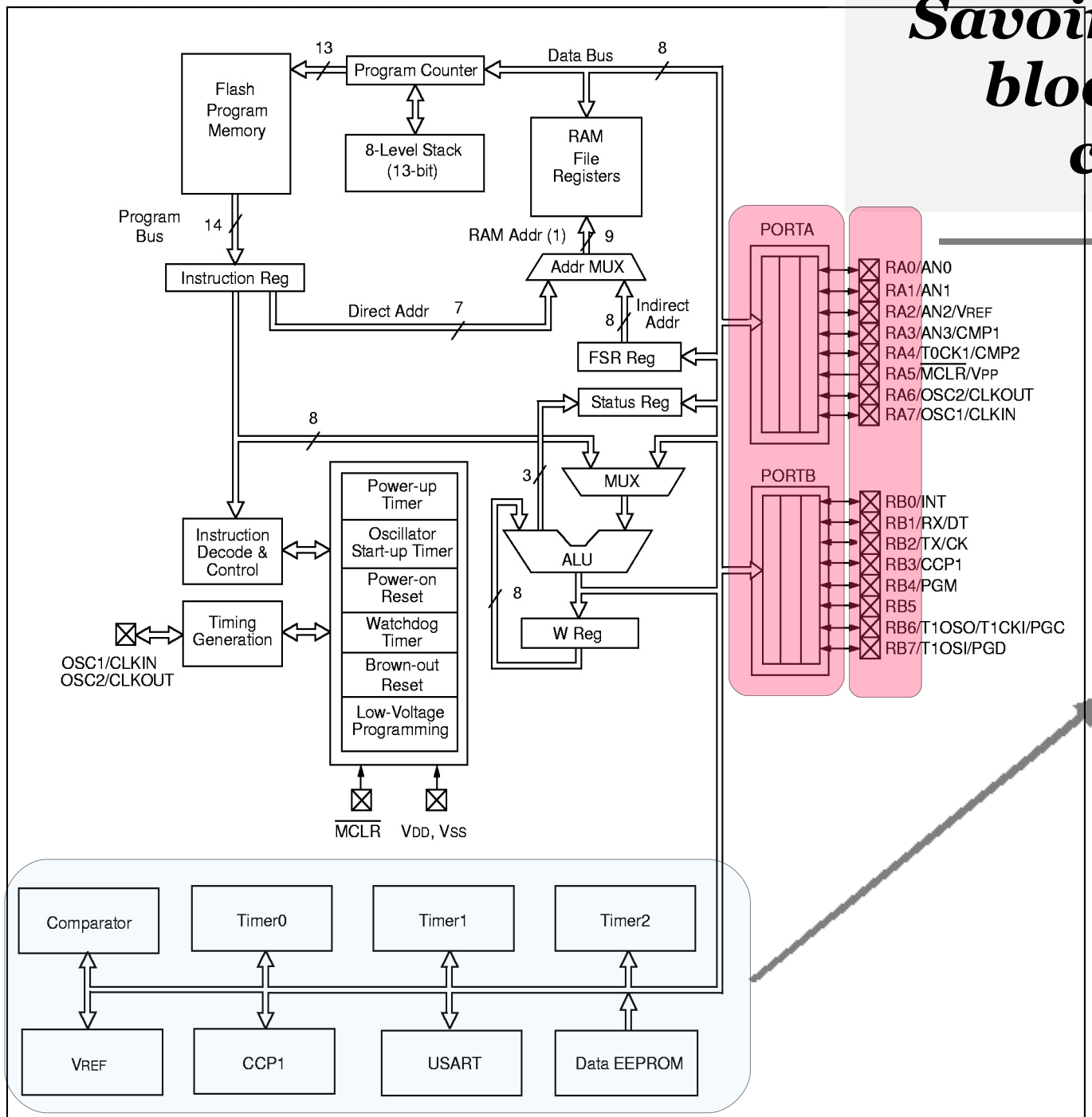
- **Registre** (case mémoire)
- **ALU**
- **PC** (Program Counter)
- **Multiplexeur**
- **Décodeur d'instructions**
- **horloge**
- **Stack (pile)**

LIFO (Last In First Out)
FIFO (First In First Out)

Issu de la documentation technique du PIC16F628

Savoir lire le schéma bloc d'un micro-contrôleur

FIGURE 3-1: BLOCK DIAGRAM



Note 1: Higher order bits are from the Status register.

- **Ports d'entrées/sorties**
- **USART** (Universal Synchronous Asynch. ReceiverTransmitter) *interface de communication série,*
- **CCP** (Capture/Compare/PWM) *Modulation en largeur d'impulsions*
- **Timer**
- **Comparateur**
- **CAN/CNA**
- **Référence de tension**
- **Module HF**
- **Liaison USB, ...**

Les éléments de choix

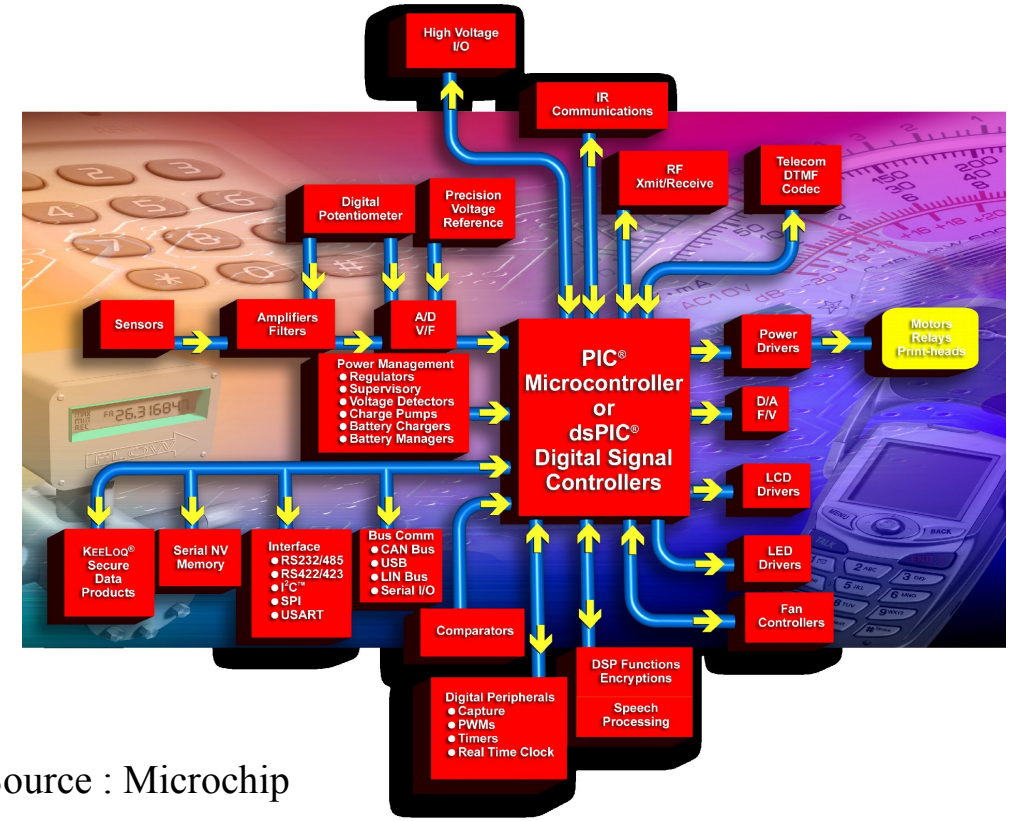
Architecture :
- ALU (8, 16, 32, 64 bits)
- Structure du processeur (Harvard, Von Neumann)
- Type de processeur (RISC, CISC)
- Taille des mémoires programme et donnée
- Nombre de ports d'entrée/sortie

Fonctionnalités :
- Fonctions *analogiques* : CAN, CNA, Comparateur, ...
- Fonctions de *timing* : Timer, Watchdog, ...
- Fonctions de *communication* : UART (Communication série), USB, I2C, ...
- Facilité de programmation : In-Circuit Serial Programming, Self Programming, ...

Mise en oeuvre, maintenance :
- Coût de développement : outils de développement, formation, ...
- Suivi du micro-contrôleur : production suivie, disponibilité, composant obsolète, ...

Caractéristiques électriques :
- Fréquence d'horloge
- Tensions d'alimentation
- Consommation d'énergie, modes faible consommation d'énergie, ...

Caractéristiques physiques :
- Type de boîtier : DIL, PLCC, ...



Source : Microchip

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Binaire, octal, décimal et hexadécimal

On rappelle tout d'abord les différentes bases qui nous seront utiles :

le **binaire** (base 2) est constitué de 2 chiffres :

0, 1

l'**octal** (base 8), est constitué de 8 chiffres :

0, 1, 2, 3, 4, 5, 6, 7

le **décimal** (base 10), est constitué de 10 chiffres :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

l'**hexadécimal** (base 16), est constitué de 16 chiffres :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Remarque : pour connaître la base associée à un nombre, on le note entre parenthèse avec en indice une lettre b,o,d ou h selon qu'il s'agit d'un codage binaire, octal, décimal ou hexadécimal. Par exemple, $(1001)_b$, $(3F1)_h$ ou $(128)_d$.

Codes pondérés

Dans une base donnée, le nombre s'exprime comme une *somme pondérée*. Par exemple, le nombre 128 **décimale** (base 10) est constitué de 3 chiffres :

- le chiffre 8 est affecté du poids de 1 (unités)
- le chiffre 2 est affecté du poids de 10 (dizaines) « Un Zéro »
- le chiffre 1 est affecté du poids de 100 (centaines) « Un Zéro Zéro»

Le nombre peut donc s'écrire

$$1 \times 100 + 2 \times 10 + 8 \times 1 = (\mathbf{128})_{\mathbf{d}}$$

Codes pondérés

Le nombre 10 **binaire** (base 2) est constitué de 2 chiffres :

- le chiffre 0 est affecté du poids de 1
- le chiffre 1 est affecté du poids de 10 (exprimé en base 2)

Le nombre peut donc s'écrire

$$1 \times 10 + 0 \times 1 = (\mathbf{10})_b$$

Remarque : le nombre 10 **binaire** *ne s'exprime pas* dix car ceci sous-entend que le nombre est exprimé en décimal...

Codes pondérés

Dans une base donnée, le nombre s'exprime comme une *somme pondérée*. Par exemple, le nombre 1F8 **hexadécimal** (base 16) est constitué de 3 chiffres :

- le chiffre 8 est affecté du poids de 1 (unités)
- le chiffre F est affecté du poids de 10 (dizaines)
- le chiffre 1 est affecté du poids de 100 (centaines)

Le nombre peut donc s'écrire

$$1 \times 100 + F \times 10 + 8 \times 1 = (\mathbf{1F8})_h$$

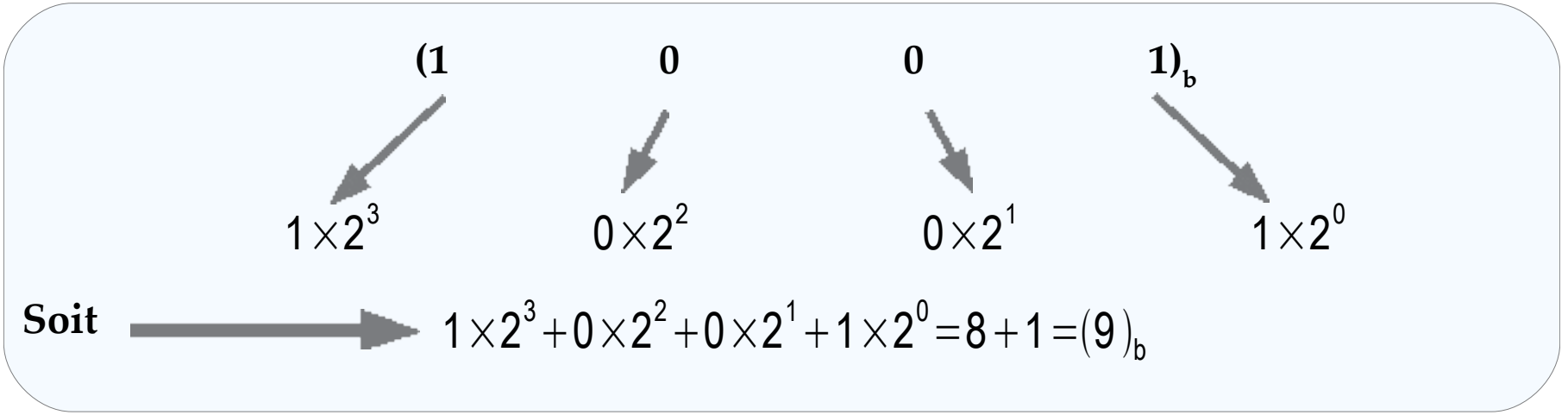
Conversion binaire-hexadécimal : le codage hexadecimal a été créé afin d'alléger l'exploitation des nombres binaires. Il permet en particulier une **conversion simple** par *regroupement des bits par 4 en partant de la droite*, chaque paquet étant alors simple à convertir :

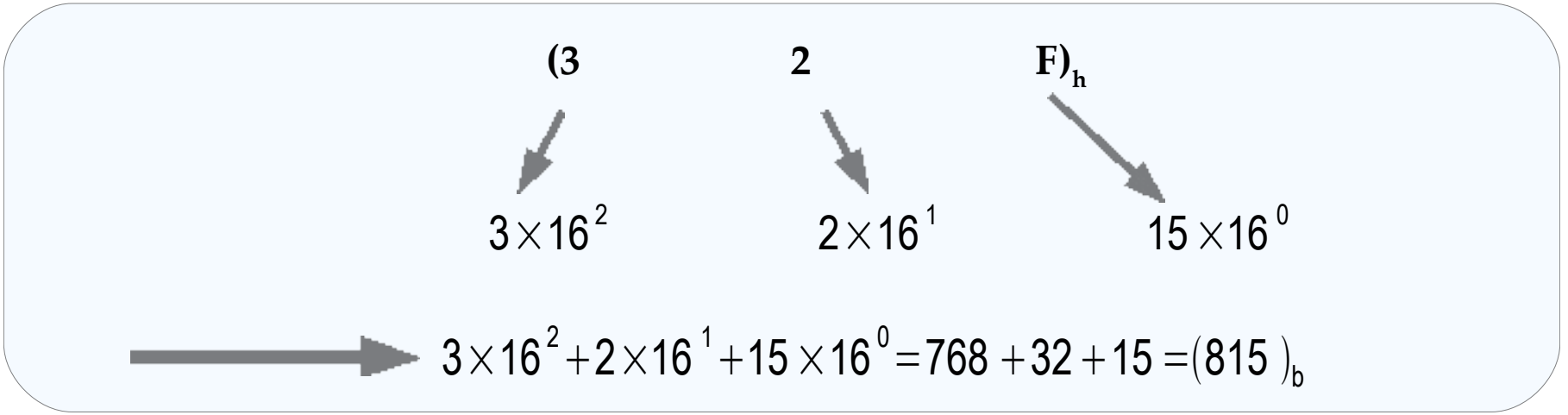
$$\underline{0\,0\,0\,1} \quad \underline{1\,1\,1\,1} \quad \underline{1\,0\,0\,0}$$

$$\mathbf{1} \qquad \mathbf{F} \qquad \mathbf{8} \qquad = \quad (\mathbf{1F8})_h$$

Conversion

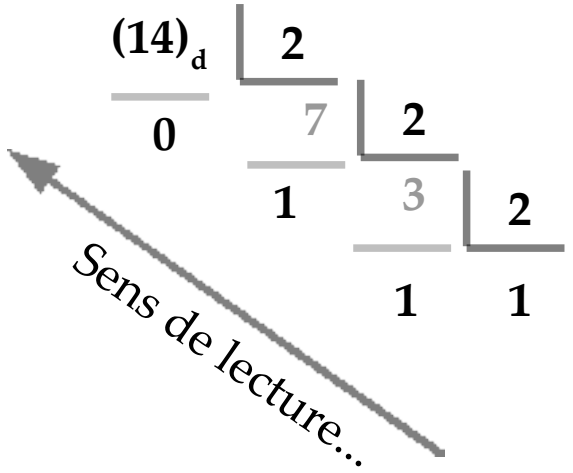
Conversion en décimal : développement en somme de puissances de la base.





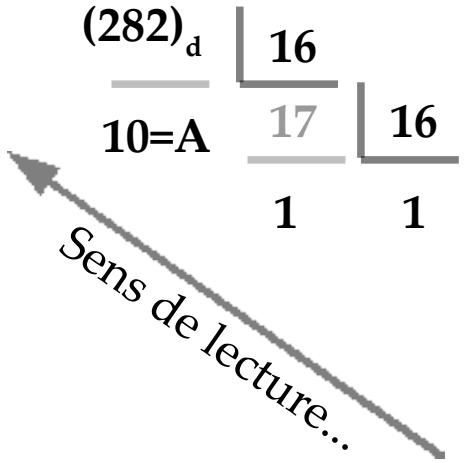
Conversion décimal ➡ binaire : division par 2 successives...

$(14)_d = (1\ 1\ 1\ 0)_b$



Conversion décimal ➡ hexadécimal : division par 16 successives...

$(282)_d = (11A)_h$



Binaire, octal, décimal et hexadécimal

Exercices 1 : convertir en décimal les chiffres binaires suivants :

$(111)_b$, $(1010)_b$, $(1001\ 1110)_b$

Exercices 2 : convertir en binaire les chiffres décimaux suivants :

8, 12, 256, 1023

Exercices 3 : convertir en hexadécimal les chiffres binaires suivants :

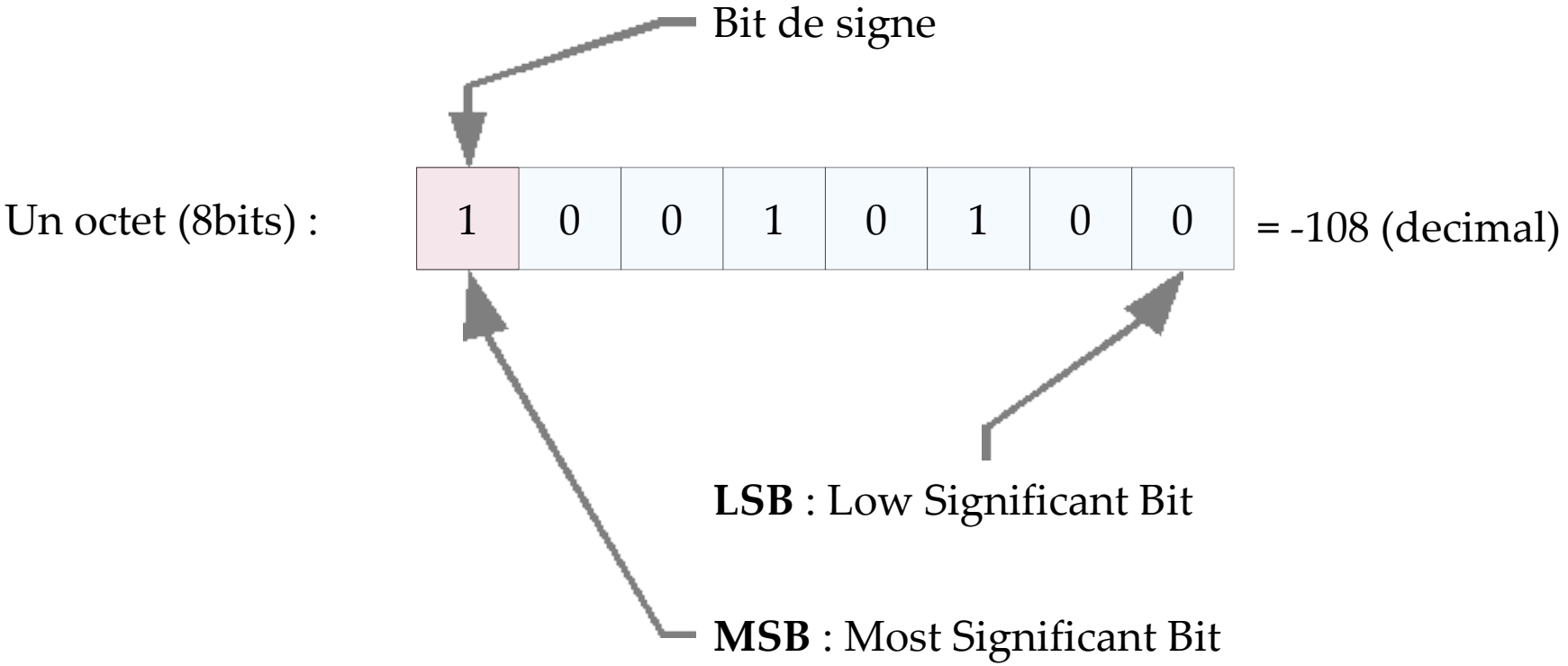
$(111)_b$, $(1010)_b$, $(1001\ 1110)_b$

Exercices 4 : convertir en hexadécimal les chiffres décimaux suivants :

8, 12, 67, 256, 1023, 12341

Binaire, octal, décimal et hexadécimal

Pour indiquer le **signe** d'un nombre binaire, on ajoute un bit en tête du nombre.
On peut ainsi coder les entier relatifs et les nombres réels.



Un octet (8 bits)
Un mot ou word (16 bits)
Un double mot ou double word (32 bits)

Opérations arithmétiques binaires

Les techniques de calcul des opérations arithmétiques *peuvent être transposées* du décimal au binaire.

- **Addition** **:** V = A + B, ***Exemple*** : $(0110)_b + (0101)_b = (1011)_b$
- **Multiplication** : V = A x B, ***Exemple*** **:** $(0110)_b . (0101)_b = (011110)_b$
- **Soustraction** **:** V = A – B,

Pour calculer V, on calcule la somme entre A et le *complément à deux* de B

Exemple : $(0110)_b – (0101)_b = (0001)_b$

NOTEZ BIEN QUE...

Une multiplication (division) par 2 correspond à un décalage à gauche (à droite).

Complément à deux : remplacer les un par des zéros (et vice-versa), puis ajouter 1.

Exemple : (1011) donne $(0100)_b + (1)_b = (0101)_b$

Exercices 1 : effectuez les additions binaires suivantes

Addition en binaire	Vérification
+ 1 0 0 1 1 0	

Addition en binaire	Vérification
+ 1 0 0 1 0 1 1 1 0 0 1 1	

Addition en binaire	Vérification
+ 1 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 0 1	

Exercices 2 :

Soustraction en binaire	Vérification
- 1 0 1 0 1 0	

Soustraction en binaire	Vérification
- 1 1 1 1 0 1 1 1 0 0 1 1	

Soustraction en binaire	Vérification
- 1 1 0 0 0 0 1 0 0 1 1 1	

Exercices 3 :

Multiplication en binaire	Vérification
x 1 0 0 1 1 0	

Multiplication en binaire	Vérification
x 1 0 0 1 0 1 1 0 1	

Multiplication en binaire	Vérification
x 1 1 0 1 0 1 1 0 0 1 0	

Exercices 4 : divisez (multipliez) par deux $(0100)_b$, $(1000101)_b$, $(3F)_h$, $(FF)_h$.

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Instructions

Un **jeu d'instruction** est un ensemble d'opérations directement réalisables sur un système micro-programmé donné.

Par exemple : le PIC18F4520 (RISC) possède un jeu d'instructions composé de 75 instructions. L'exécution d'une instruction peut nécessiter un ou plusieurs **cycles d'horloges** suivant la complexité de l'instruction.

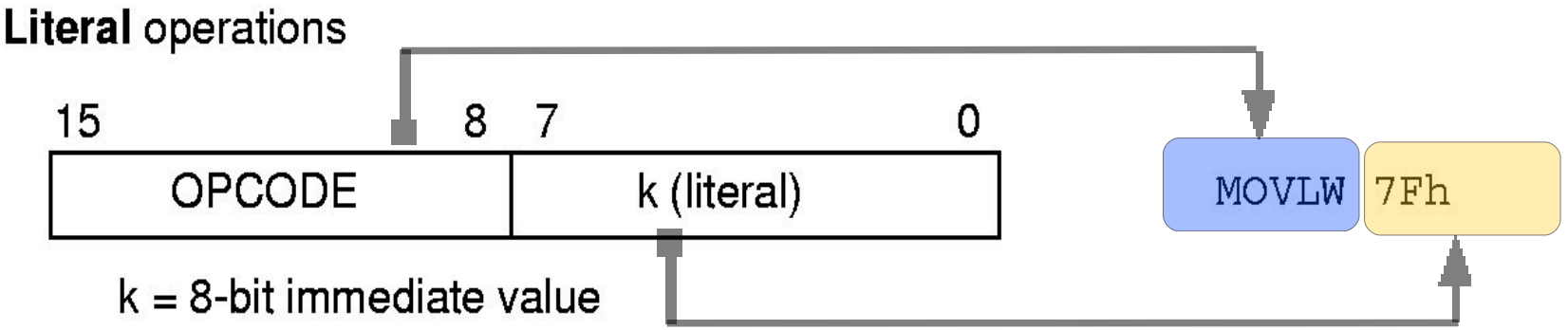
NOTE : Un **cycle d'horloge** correspond à une période de **l'horloge** (signal de référence temporelle). La **fréquence d'horloge** est le nombre de cycles effectués par une horloge en une seconde.

Instructions

Une instruction est composée au minimum de deux parties:

Instruction = OPCODE + opérande(s)

OPCODE (Operation CODE) : partie d'une instruction qui précise quelle opération doit être réalisée

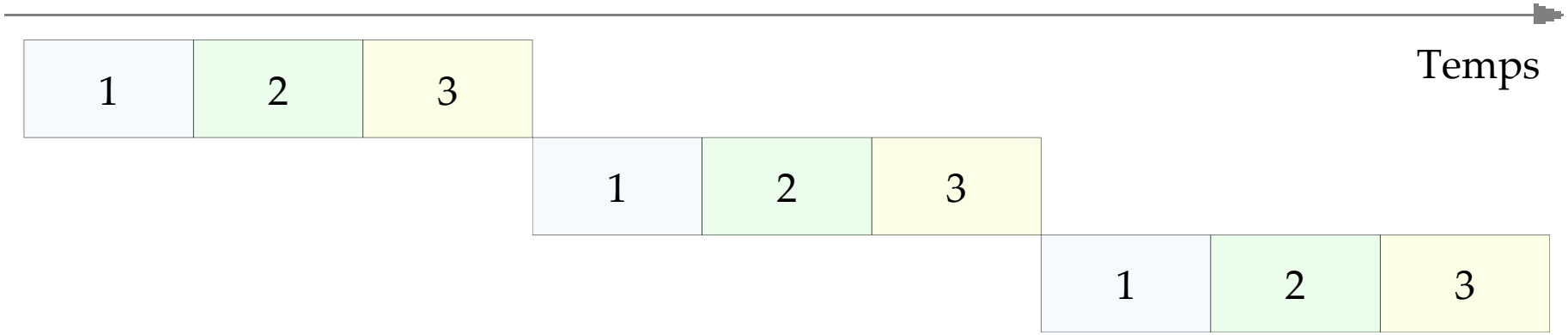


Extrait du datasheet (documentation technique) du PIC18F4520.

Pipeline et flot d'instructions

3 étapes pour l'exécution d'une instruction :

- Lecture de l'instruction (1)
- Décodage de l'instruction (2)
- Exécution de l'instruction (3)



Création d'un pipeline => permet une exécution plus rapide des instructions

Les différents modes d'adressage

La nature et le nombre d'opérandes qui constituent une instruction déterminent le **mode d'adressage** de l'instruction. *On distingue 4 modes d'adressage principaux.*

L'adressage inhérent : il n'y a pas d'opérande !

ex : NOP, RESET, CLRWDT ;

Description de l'instruction RESET extraite de la notice technique (le « datasheet ») du PIC 18F4520 [micro-contrôleur utilisé en TP].

ATTENTION : Un nombre important d'information utiles figure sur ces fiches...

RESET	Reset
Syntax:	RESET
Operands:	None
Operation:	Reset all registers and flags that are affected by a $\overline{\text{MCLR}}$ Reset.
Status Affected:	All
Encoding:	0000 0000 1111 1111
Description:	This instruction provides a way to execute a $\overline{\text{MCLR}}$ Reset in software.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Start Reset	No operation	No operation

Example: RESET

After Instruction
Registers = Reset Value
Flags* = Reset Value

Les différents modes d'adressage

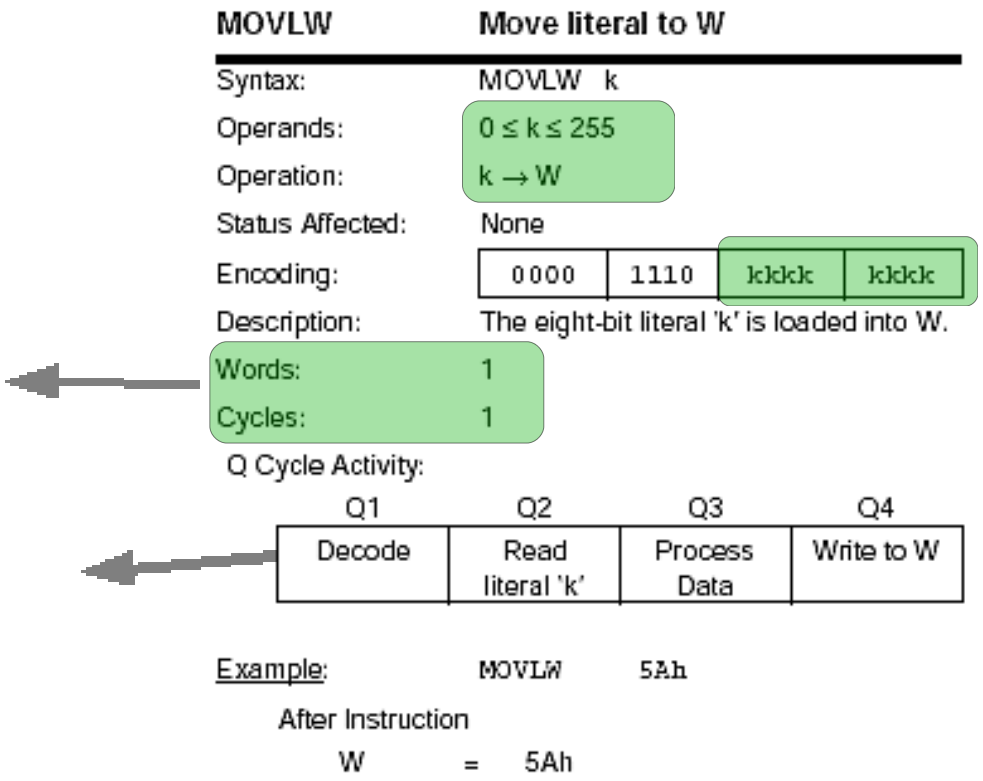
La nature et le nombre d'opérandes qui constituent une instruction détermine le **mode d'adressage** de l'instruction. *On distingue 4 modes d'adressage principaux.*

L'adressage immédiat : l'opérande est une valeur

ex : MOVLW 5Ah ;

Nombres de cycles nécessaires à l'exécution

Exécution de l'instruction (pipeline à 4 niveaux)



Les différents modes d'adressage

La nature et le nombre d'opérandes qui constituent une instruction détermine le **mode d'adressage** de l'instruction. *On distingue 4 modes d'adressage principaux.*

L'adressage direct (étendu) : l'opérande est l'adresse (bits de poids faibles de l'adresse complète) de la donnée dans la page mémoire active.

ex : ADDWF 000Fh,

En **mode direct étendu** : on transmet l'adresse complète

```
Example:        ANDWF    REG,  0,  0
    Before Instruction
        W       =   17h
        REG     =   C2h
    After Instruction
        W       =   02h
        REG     =   C2h
```

ANDWF — **AND W with f**

Syntax:	ANDWF f {,d {,a}}
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(W) .AND. (f) → dest
Status Affected:	N, Z
Encoding:	0001 | 01da | ffff | ffff
Description:	The contents of W are AND'ed with register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Les différents modes d'adressage

La nature et le nombre d'opérandes qui constituent une instruction détermine le **mode d'adressage** de l'instruction. *On distingue 4 modes d'adressage principaux.*

- **l'adressage indirect (indexé) :** l'opérande est l'adresse d'un registre qui contient l'adresse de la donnée.

- En **mode indirect indexé**, on ajoute un décalage par rapport à l'adresse.

NOTE : Il existe de nombreux autres modes d'adressage (ex. implicite, inhérent, relatif) : leur nombre varie en fonction du constructeur et du micro-contrôleur !

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Logique combinatoire et séquentielle

La compréhension du fonctionnement d'un microcontrôleur s'appuie sur des connaissances élémentaires de logique combinatoire et séquentielle.

- **un système est dit combinatoire** si l'état (logique) des sorties ne dépend que de l'état (logique) présent appliqué à ses entrées.

- **un système est dit séquentiel** si l'état (logique) de la sortie du système **à l'instant t** dépend de l'état (logique) présent appliqué aux entrées **et** des états de la sortie dans le passé.

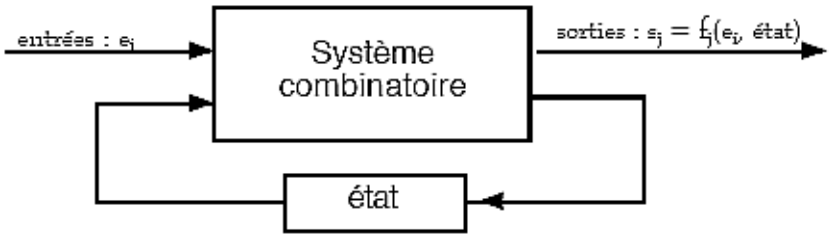


Table de vérité

Considérons tout d'abord le cas de la logique combinatoire à 1 sortie (le cas à plusieurs sorties n'est pas très différent). Pour connaître l'état du système aux divers combinaisons logiques des entrées on construit la **table de vérité** qui exprime la valeur de la sortie *s* en fonction de toutes les configurations possible des entrées binaires ($\mathbf{E_i}$), cf. ci-dessous.

Table de vérité à 2 entrées :

E_1	E_2	s
0	0	x
0	1	x
1	0	x
1	1	x

Table de vérité à 3 entrées :

E_1	E_2	E_3	s
0	0	0	x
0	0	1	x
0	1	0	x
0	1	1	x
1	0	0	x
1	0	1	x
1	1	0	x
1	1	1	x

On notera que pour une fonction logique à *une seule variable d'entrée*, il existe 2^2=4 combinaisons de sorties.

E_1	F0	F1	F2	F3
0	0	1	0	1
1	0	0	1	1

De même, pour deux variables d'entrées, il existe 2^4=16 combinaisons de sorties.

E_1	E_2	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FI0	F11	F12	F13	F14	F15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Opérateurs élémentaires

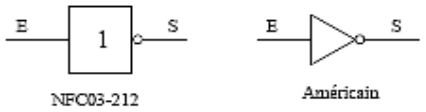
Dans ces configurations, on extrait typiquement *6 fonctions logiques d'intérêt* que sont les opérateurs **NON** (une entrée), **ET**, **OU**, **ET-NON**, **OU-NON**, et **OU-EXCLUSIF** (deux entrées).

NON

Table de vérité :

E	S
0	1
1	0

Symboles :

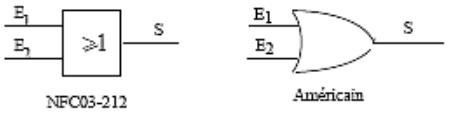


OU

Table de vérité :

E_1	E_2	S
0	0	0
0	1	1
1	0	1
1	1	1

Symboles :

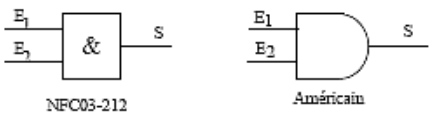


ET

Table de vérité :

E_1	E_2	S
0	0	0
0	1	0
1	0	0
1	1	1

Symboles :

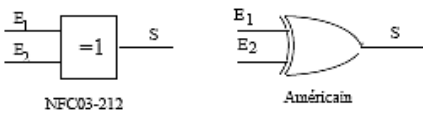


OU-EX

Table de vérité :

E_1	E_2	S
0	0	0
0	1	1
1	0	1
1	1	0

Symbole :

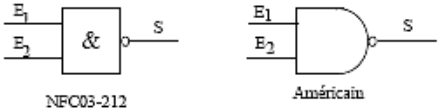


ET-NON

Table de vérité :

E_1	E_2	S
0	0	1
0	1	1
1	0	1
1	1	0

Symboles :

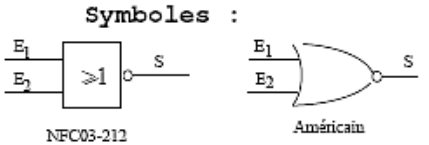


OU-NON

Table de vérité :

E_1	E_2	S
0	0	1
0	1	0
1	0	0
1	1	0

Symboles :



Les opérateurs ET-NON et OU-NON forment un groupe complet, c.à.d. que toute fonction logique complexe peut être construite sur la base de l'une de ces fonctions élémentaires.

Algèbre de BOOLE

Les opérateurs logiques élémentaires permettent la construction d'une algèbre dite « **algèbre de Boole** ». Ainsi, si on considère deux entrées binaires *A* et *B*, on adopte alors la convention suivante pour construire des équations logiques :

NON	$\overline{A}$	**ET-NON,**	$\overline{A \cdot B} = \overline{A} + \overline{B}$ $\overline{A + B} = \overline{A} \cdot \overline{B}$
ET	$A \cdot B$	**OU-NON,**	$\overline{A + B}$
OU,	$A + B$	**OU-EXCLUSIF**	$A \oplus B$

Les différentes opérations bénéficient des propriétés suivantes

Associativité **:**

$$(A \cdot B) \cdot C = A \cdot B \cdot C$$
$$(A + B) + C = A + B + C$$
$$(A \oplus B) \oplus C = A \oplus B \oplus C$$

Commutativité **:**

$$A \cdot B = B \cdot A \qquad \overline{A \cdot B} = \overline{B \cdot A}$$
$$A + B = B + A \qquad \overline{A + B} = \overline{B + A}$$
$$A \oplus B = B \oplus A$$

Distributivité **:**

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$
$$A \cdot (B + C) = A \cdot B + A \cdot C$$

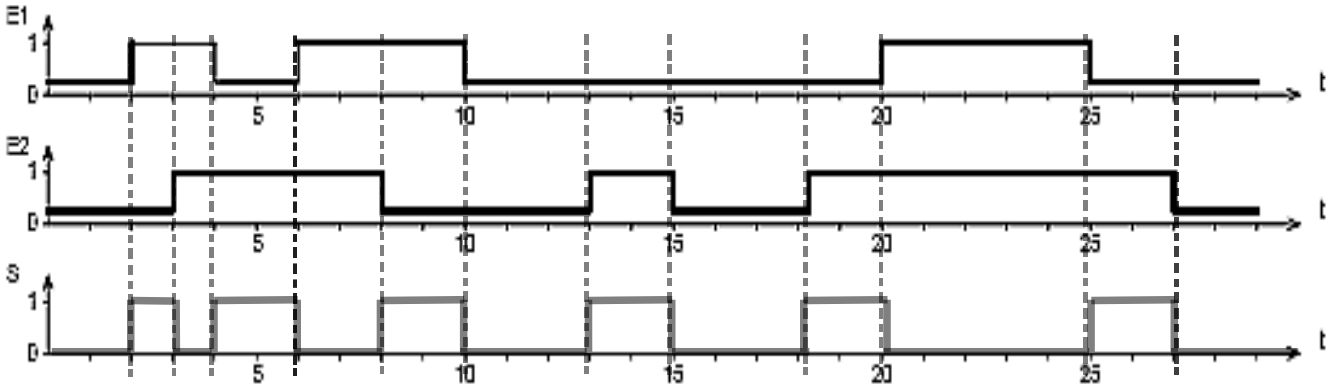
Lois de De Morgan **:**

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$
$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

Le chronogramme

Dans les microcontrôleurs, les états du système changent en fonction d'une **base de temps** qui est l'horloge. Ceci conduit naturellement à introduire les **chronogrammes** comme outil d'analyse des états logiques d'un système.

Le chronogramme a pour objet de tracer l'état binaire de la (des) sortie(s) en fonction de l'évolution au cours du temps de l'état des entrées. Ceci est illustré ci-dessous.

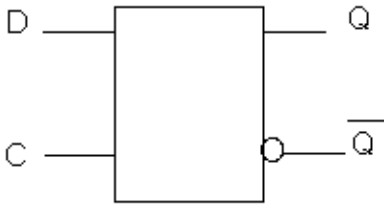


Les bascules asynchrones

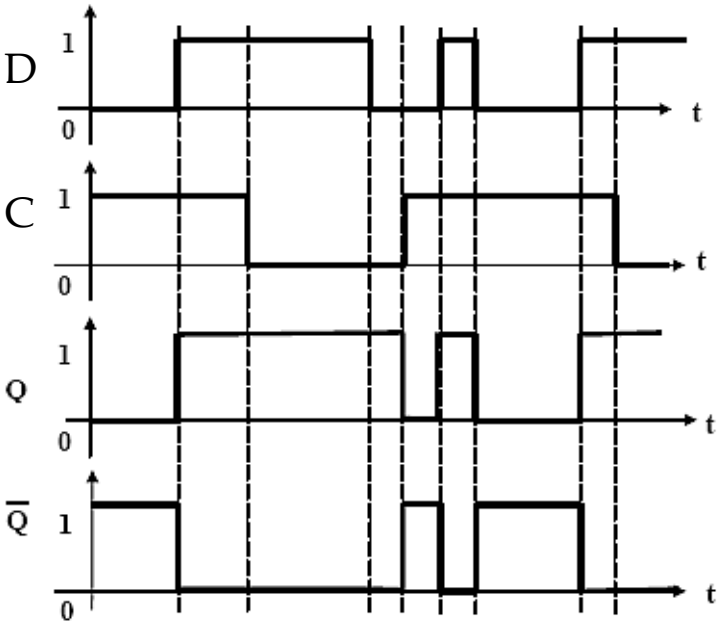
Pour l'essentiel, une bascule asynchrone est une fonction « mémoire » qui est commandée. Ce type de fonction est notamment utilisé pour créer des registres du microcontrôleur.

Le verrou D (Latch D)

Le verrou D (ou bascule D asynchrone) est très répandu : elle copie en sortie l'état de l'entrée D uniquement si sa commande C est active ; dans le cas contraire, l'état en sortie Q est celui précédent, cf. chronogramme.

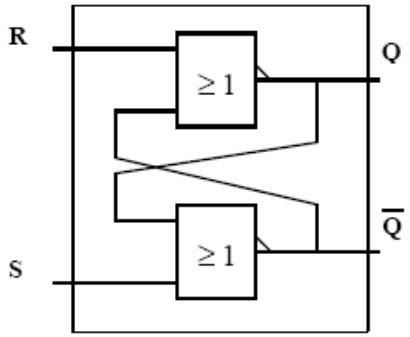


Entrées		Sorties	
C	D	Qn+1	$\overline{Qn+1}$
0	X	Qn	$\overline{Qn}$
1	0	0	1
1	1	1	0

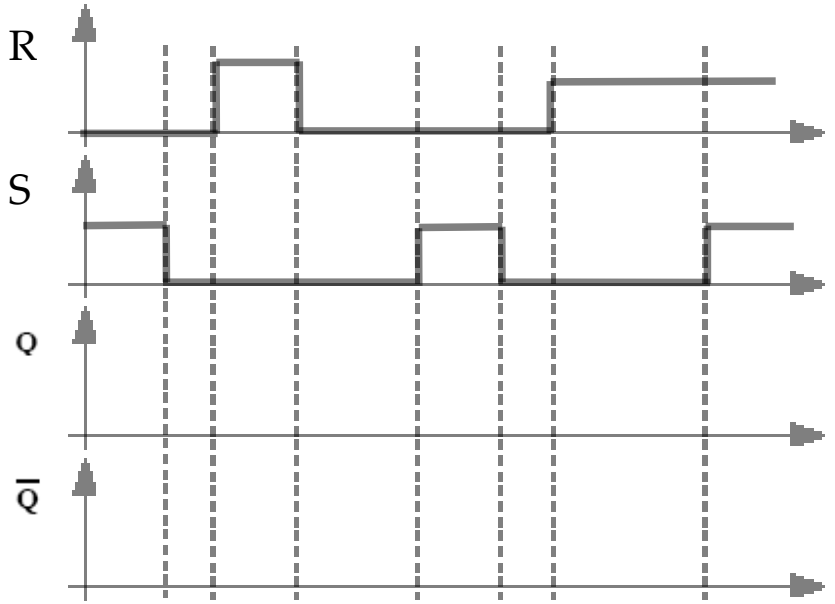


Exercice : étude de la bascule RS

Écrire la table de vérité et compléter le chronogramme pour le verrou RS ci-dessous.

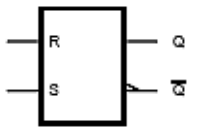


entrée	entrée	sortie	sortie
R	**S**	**Qn**	$\overline{\mathbf{Qn}}$
0	0		
0	1		
1	0		
1	1		



2. LE VERROU DE TYPE RS.

Norme européenne



Entrée R : Quand cette donnée prend l'état interne 1, un état 0 est mémorisé par le verrou. Quand elle est à l'état interne 0, elle n'a aucun effet.

Entrée S : Quand cette donnée prend l'état interne 1, un état 1 est mémorisé par le verrou. Quand elle est à l'état interne 0, elle n'a aucun effet.

Remarque : L'effet de la combinaison R = S = 1 est précisé par le constructeur.

R : Reset ou forçage à l'état 0 de la sortie.

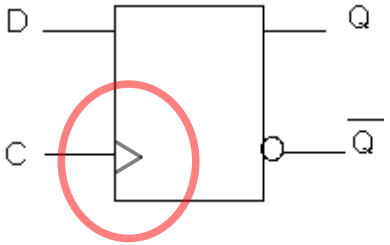
S : Set ou forçage à l'état 1 de la sortie.

Les bascules synchrones

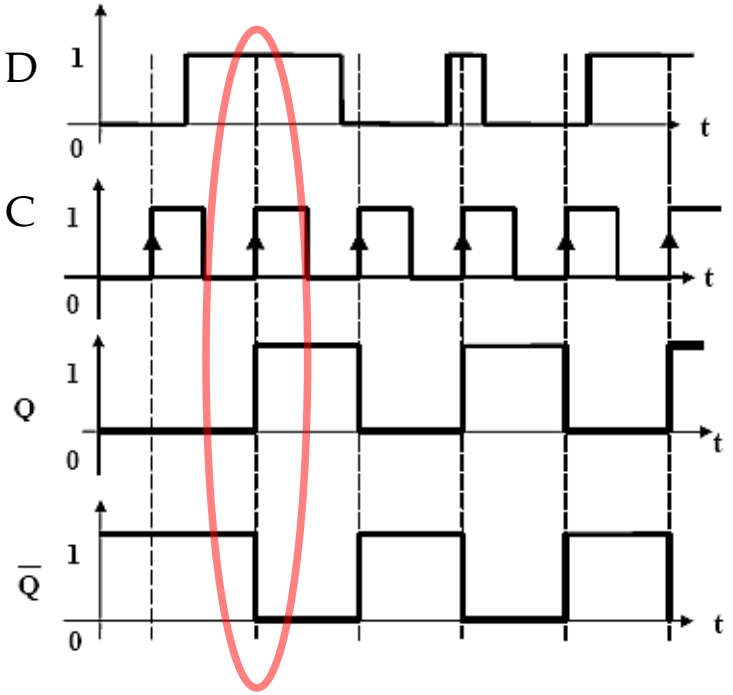
Une bascule synchrone est une bascule qui ne change d'état que sur front montant ou descendant appliqué sur son entrée de commande. Ce type de bascule est à la base du fonctionnement du microcontrôleur.

La Bascule D (Flip-Flop D)

C'est la version *synchrone* du verrou D !

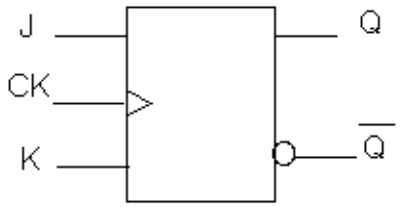


Entrées		Sorties	
C	D	Qn+1	$\overline{Qn+1}$
0	X	Qn	$\overline{Qn}$
↑	0	0	1
↑	1	1	0

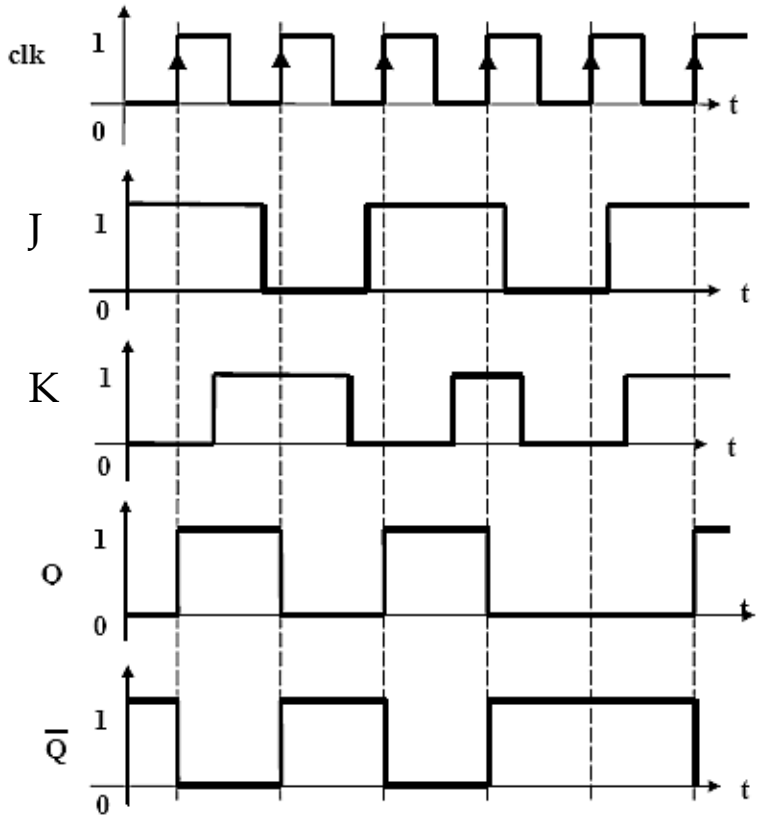


Les bascules synchrones

La Bascule JK (*Jump-Knock out*)



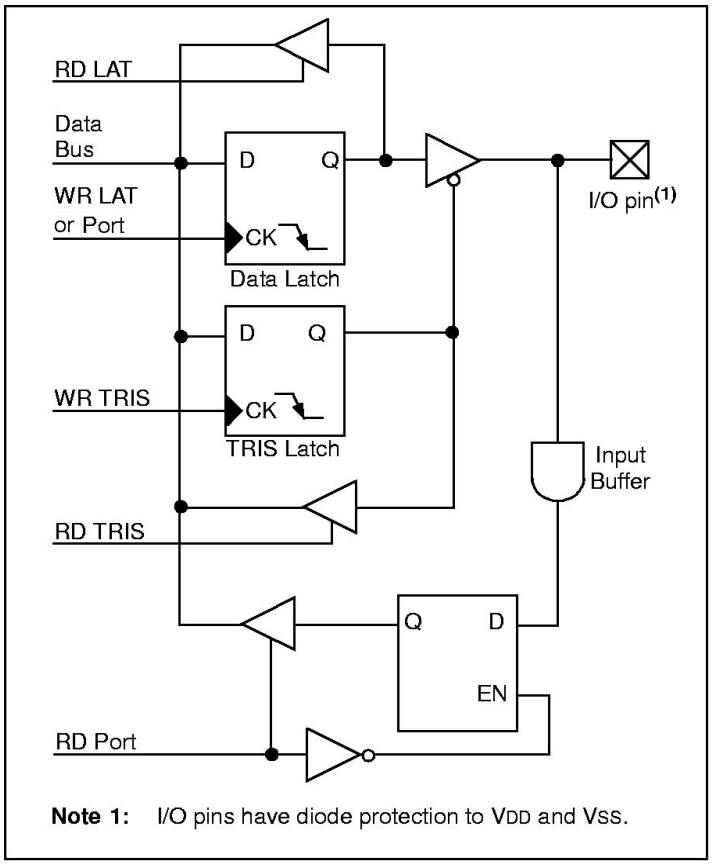
Entrées			Sorties	
CK	J	K	Qn+1	$\overline{Qn+1}$
0	X	X	Qn	$\overline{Qn}$
1	X	X	Qn	$\overline{Qn}$
↓	X	X	Qn	$\overline{Qn}$
↑	0	0	Qn	$\overline{Qn}$
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	$\overline{Qn}$	Qn



Structures des ports d'entrées/sorties

Un port d'entrées/sorties est par définition un port bidirectionnel.

De fait, il est donc nécessaire de configurer **la direction** du port (*in* ou *out*). Dans le microcontrôleur, des registres spécifiques sont dédiés à la gestion de ces ports...



EXAMPLE 10-3: INITIALIZING PORTC

```
CLRF    PORTC   ; Initialize PORTC by
                ; clearing output
                ; data latches
CLRF    LATC    ; Alternate method
                ; to clear output
                ; data latches
MOVLW   0CFh    ; Value used to
                ; initialize data
                ; direction
MOVWF   TRISC   ; Set RC<3:0> as inputs
                ; RC<5:4> as outputs
                ; RC<7:6> as inputs
```

Extrait de la documentation technique du PIC18F4520 de Microchip.

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

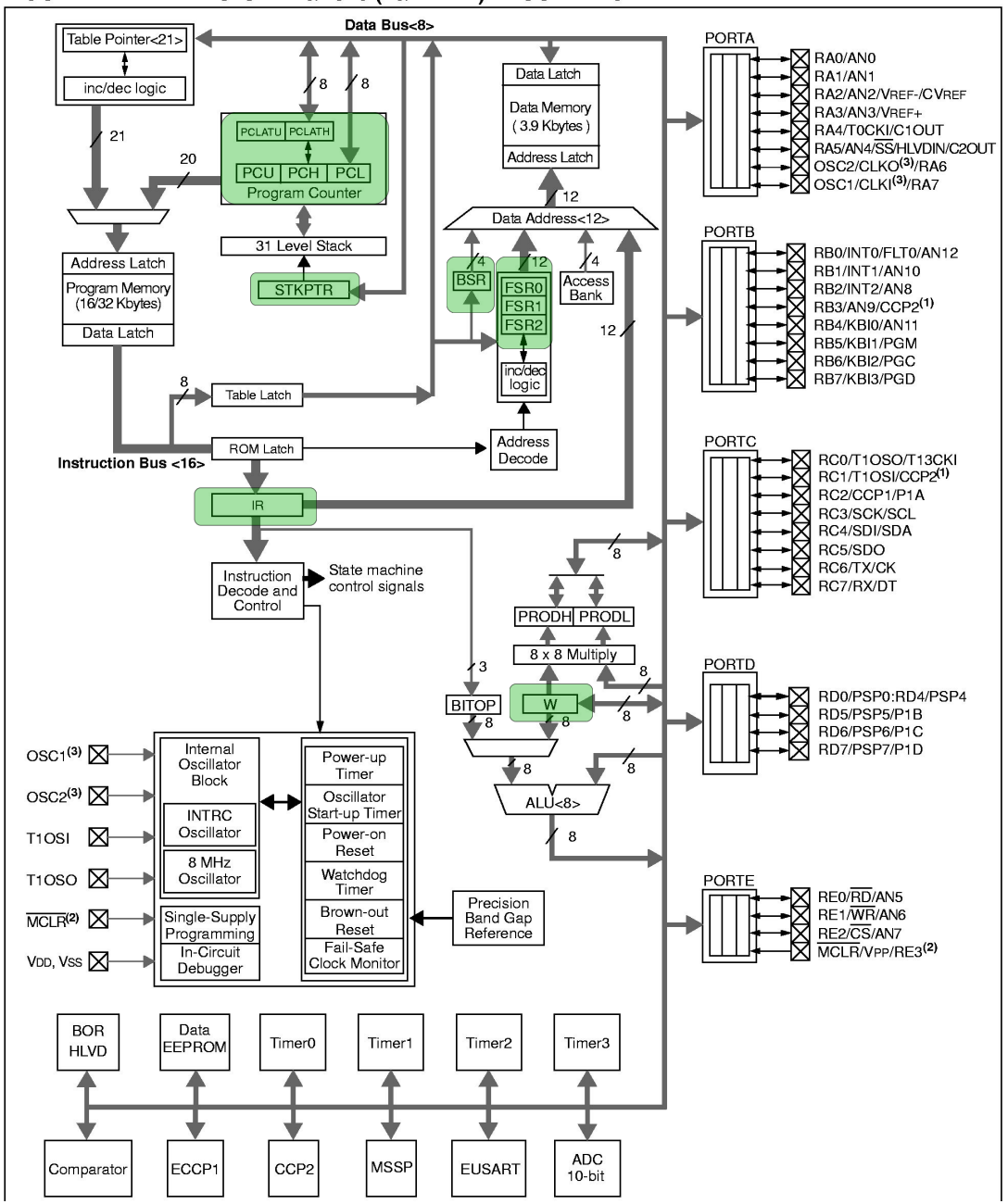
Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

FIGURE 1-2: PIC18F4420/4520 (40/44-PIN) BLOCK DIAGRAM



Les registres

Un **registre** 8 bits est synonyme d'un ensemble de 8 cases mémoire. De nombreux registres sont utilisés pour gérer le microcontrôleur.

Le registre W (accumulateur)

Le compteur programme (PC)

Le registre d'état (Flags)

Les registres de configuration :

les registres de directions pour les ports d'entrées/sorties (TRIS, SFR), les registres de gestion des interruptions, de gestion de la mémoire (BSR, GPR, *etc.*)

ATTENTION : tous les registres du microcontrôleur ne sont pas représentés sur le schéma...

Par exemple : le registre d'état

Le registre d'état (Status Register) contient des bits d'informations sur les opérations arithmétiques menées par l'ALU (ex., le dépassement de format après avoir demandé l'addition de deux valeurs 8 bits).

STATUS REGISTER

U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC	C
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **N:** Negative bit
This bit is used for signed arithmetic (2's complement). It indicates whether the result was negative (ALU MSB = 1).
1 = Result was negative
0 = Result was positive

bit 3 **OV:** Overflow bit
This bit is used for signed arithmetic (2's complement). It indicates an overflow of the 7-bit magnitude which causes the sign bit (bit 7 of the result) to change state.
1 = Overflow occurred for signed arithmetic (in this arithmetic operation)
0 = No overflow occurred

bit 2 **Z:** Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit Carry/$\overline{\text{borrow}}$ bit
For ADDWF, ADDLW, SUBLW and SUBWF instructions:
1 = A carry-out from the 4th low-order bit of the result occurred
0 = No carry-out from the 4th low-order bit of the result

Note: For $\overline{\text{borrow}}$, the polarity is reversed. A subtraction is executed by adding the 2's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either bit 4 or bit 3 of the source register.

bit 0 **C:** Carry/$\overline{\text{borrow}}$ bit
For ADDWF, ADDLW, SUBLW and SUBWF instructions:
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

Phase de démarrage du micro-contrôleur

Suite à une opération de remise à zéro (RESET), le micro-contrôleur effectue une phase de démarrage :

1/ RESET : il peut être déclenché par la mise sous tension du micro-contrôleur, la réception d'un signal sur la broche RESET du micro-contrôleur, une instruction de RESET, ...

2/ Initialisation du micro-contrôleur : le micro-contrôleur effectue une temporisation afin de garantir la stabilité des signaux d'horloge.

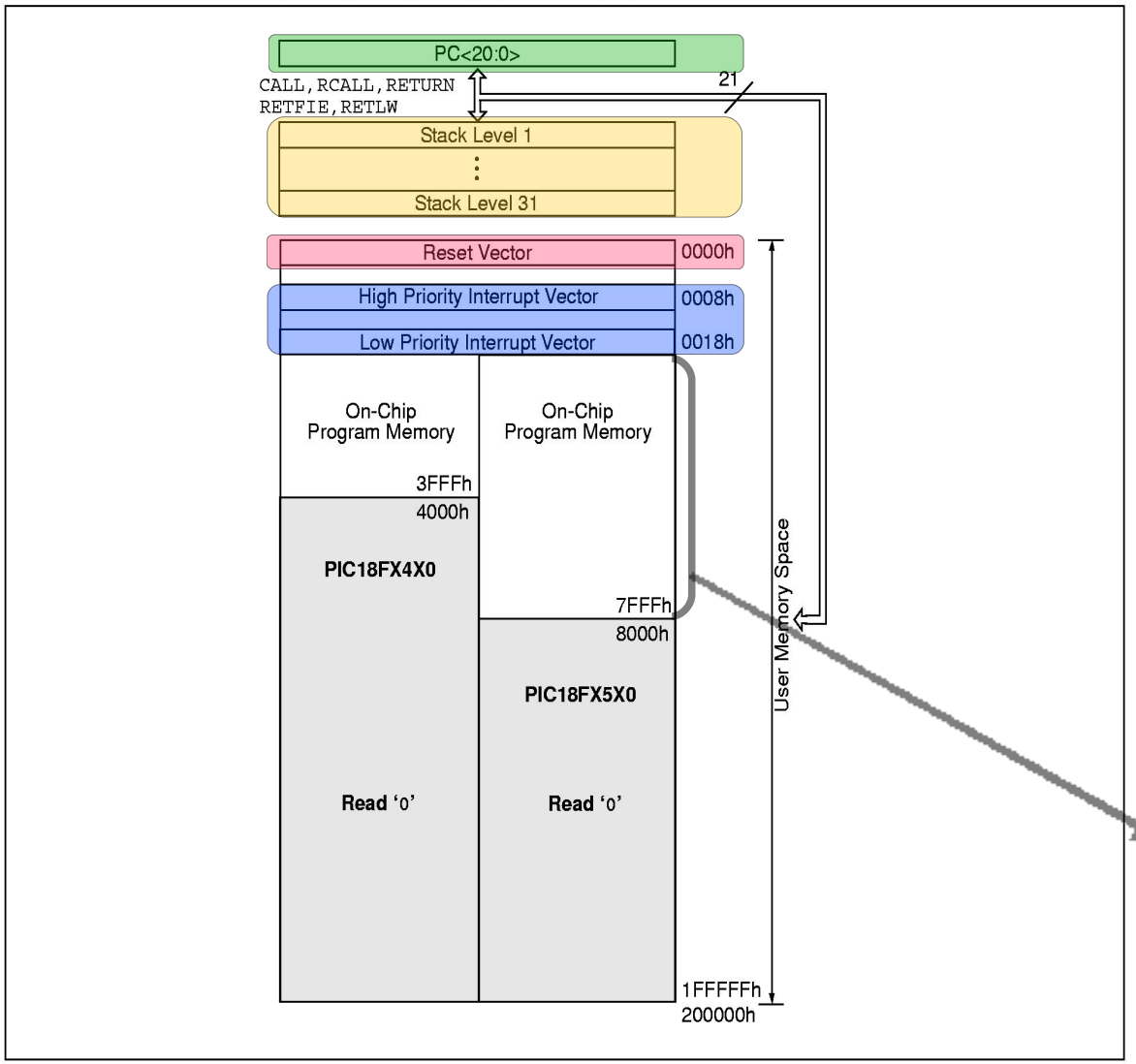
3/ Effacement des registres : le micro-contrôleur efface le contenu des registres (variable en fonction du « mode de RESET » que vous effectuez).

4/ Lecture du vecteur RESET
Le micro-contrôleur lit l'adresse du programme principal dans la mémoire programme.

5/ Début de l'exécution du programme principal.

Organisation de la mémoire programme

FIGURE 5-1: PROGRAM MEMORY MAP AND STACK FOR PIC18F2420/2520/4420/4520 DEVICES



Compteur de programme (PC)

le compteur de programme

Pile (Stack)

une pile pour gérer les appels programmes et les interruptions

Vecteur Reset

pointeur vers l'adresse mémoire du début du programme principal

Vecteurs d'interruption

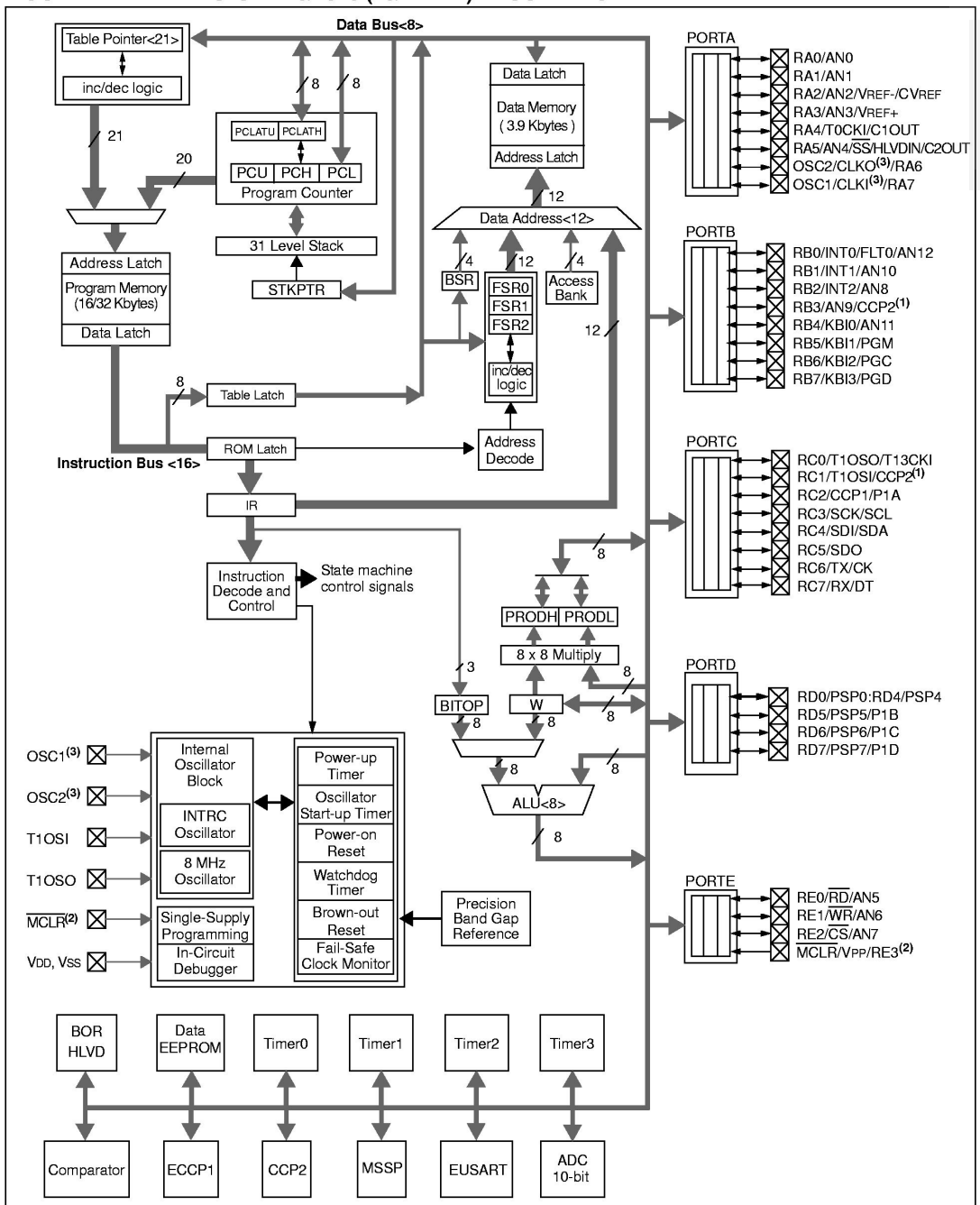
pointeur vers l'adresse mémoire du programme à exécuter en cas d'interruptions

Mémoire programme

zone mémoire réservée au stockage des programmes écrits par l'utilisateur

Remarque : Un **pointeur** est une variable contenant une adresse mémoire.

FIGURE 1-2: PIC18F4420/4520 (40/44-PIN) BLOCK DIAGRAM



Note 1: CCP2 is multiplexed with RC1 when configuration bit CCP2MX is set, or RB3 when CCP2MX is not set.

2: RE3 is only available when $\overline{MCLR}$ functionality is disabled.

3: OSC1/CLKI and OSC2/CLKO are only available in select oscillator modes and when these pins are not being used as digital I/O. Refer to **Section 2.0 "Oscillator Configurations"** for additional information.

Exécution d'une instruction

Adressage inhérent

L'instruction ne comporte pas d'opérande et agit implicitement sur un registre.

Exemples : SLEEP, RESET, NOP

Adressage immédiat

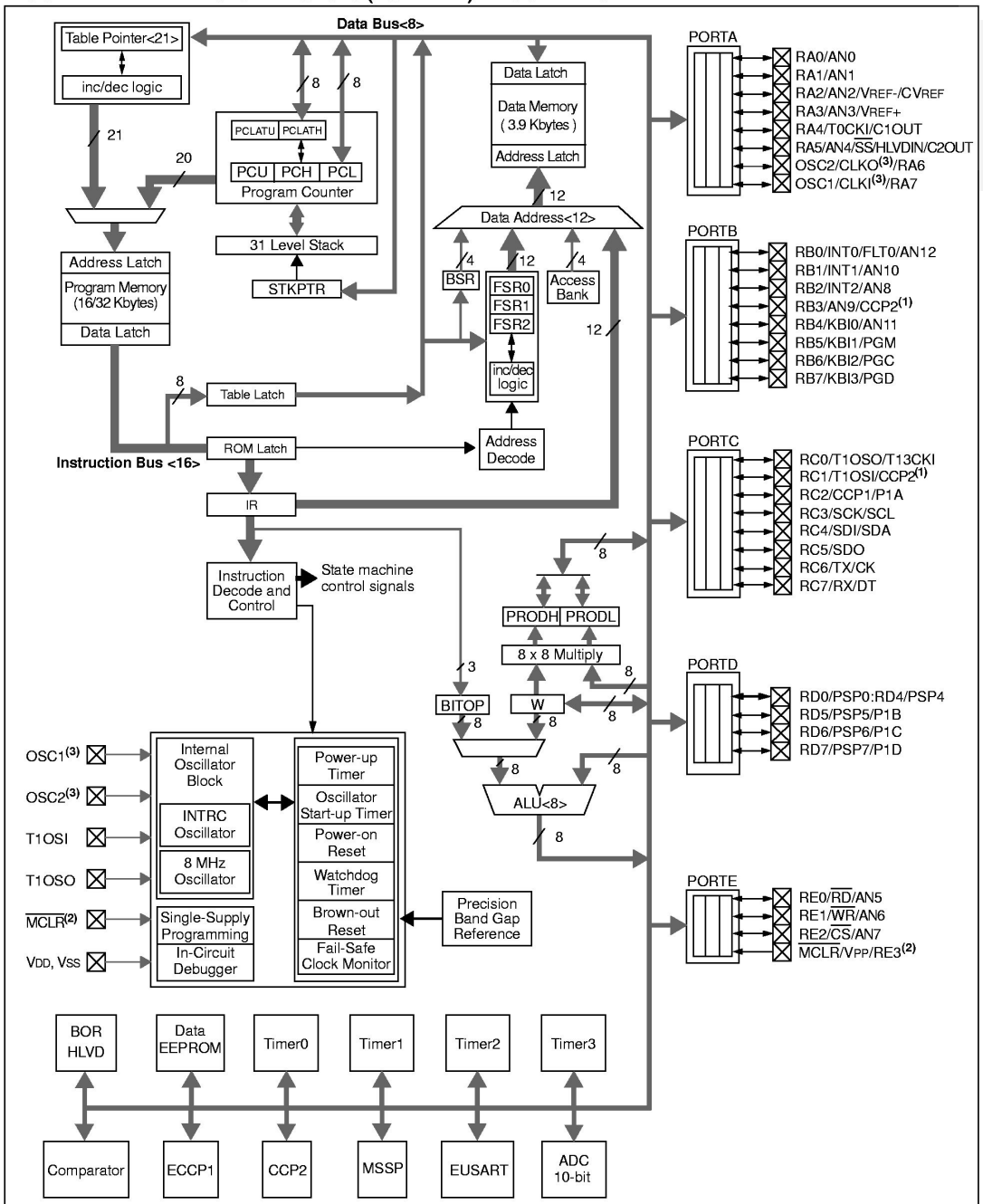
L'instruction comporte une opérande et agit explicitement sur un registre

Exemples : ADDLW, MOVLW

Déroulement:

(1) Le compteur programme indique l'adresse de l'instruction suivante dans la mémoire programme. (2) L'instruction est lue et stockée dans le registre d'instruction. (3) Puis elle est décodée par le module de décodage et de contrôle des instructions. (4) Finalement elle est exécutée.

FIGURE 1-2: PIC18F4420/4520 (40/44-PIN) BLOCK DIAGRAM



Note 1: CCP2 is multiplexed with RC1 when configuration bit CCP2MX is set, or RB3 when CCP2MX is not set.

2: RE3 is only available when MCLR functionality is disabled.

3: OSC1/CLKI and OSC2/CLKO are only available in select oscillator modes and when these pins are not being used as digital I/O. Refer to **Section 2.0 "Oscillator Configurations"** for additional information.

Exécution d'une instruction

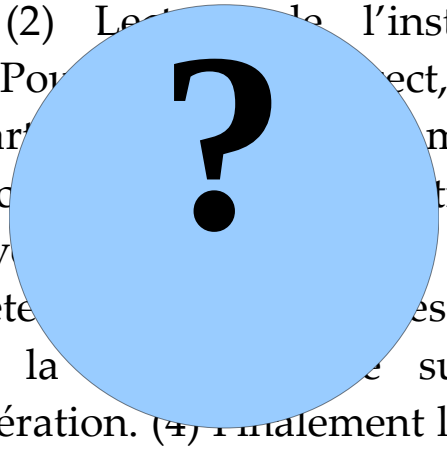
Adressage direct (étendu)

L'instruction comporte une opérande qui indique l'adresse mémoire sur laquelle s'effectue l'opération.

Exemples : CLRF (direct), MOVFF (étendu)

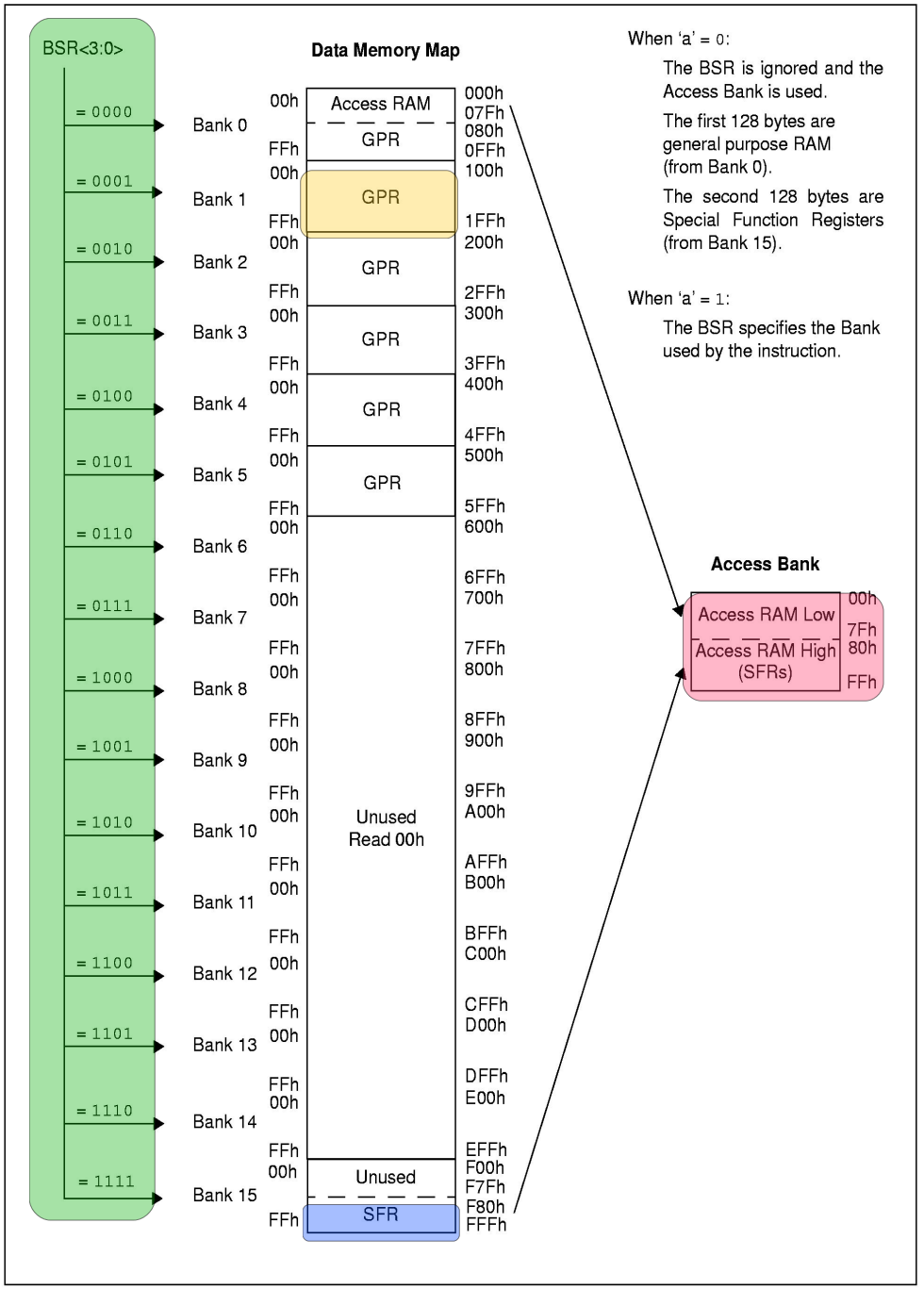
Déroulement:

(1) Lecture de l'instruction dans la mémoire programme à l'adresse pointée par le compteur programme. (2) Le[illegible]de l'instruction et décodage. (3) Pou[illegible]ect, l'opérande constitue la par[illegible] mémoire sur laquelle s'effec[illegible]tie haute est complétée av[illegible] (3') Pour l'adressage éte[illegible]est l'adresse complète de la [illegible] sur laquelle s'effectue l'opération. (4) Finalement l'instruction est exécutée sur la case mémoire pointée.



Organisation de la mémoire données

FIGURE 5-6: DATA MEMORY MAP FOR PIC18F2520/4520 DEVICES



BSR (*Bank Select Register*)

Permet de pré-sélectionner la page pour un accès mémoire plus rapide.
=> notion de **pagination de la mémoire**

GPR (*General Purpose Registers*)

Espaces mémoires qui permet le stockage de données temporaires (variable, ...)

Access Bank

pointeurs vers des zones mémoires

SFR (*Special Function Registers*)

Registres de contrôle et d'état pour les périphériques (notamment...)

Pagination de la mémoire

« La pagination de la mémoire consiste à diviser la mémoire en blocs (pages) de longueur fixe. » (Source : Comment Ça Marche)

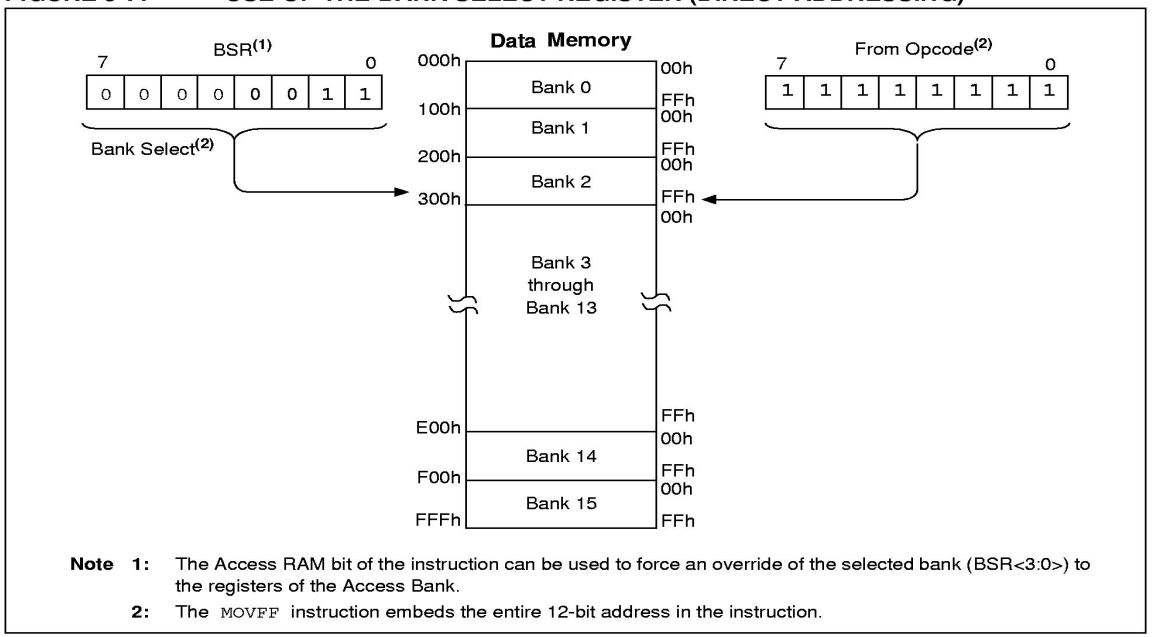
Une adresse mémoire est alors divisée en deux parties :

Partie haute	Partie Basse

Dans le cas d'une instruction avec adressage direct, on transmet seulement la partie basse de l'adresse. Le micro-contrôleur utilise le registre BSR pour compléter l'adresse.

Attention !! En adressage direct, on doit s'assurer que l'on travaille **dans la bonne page mémoire**.

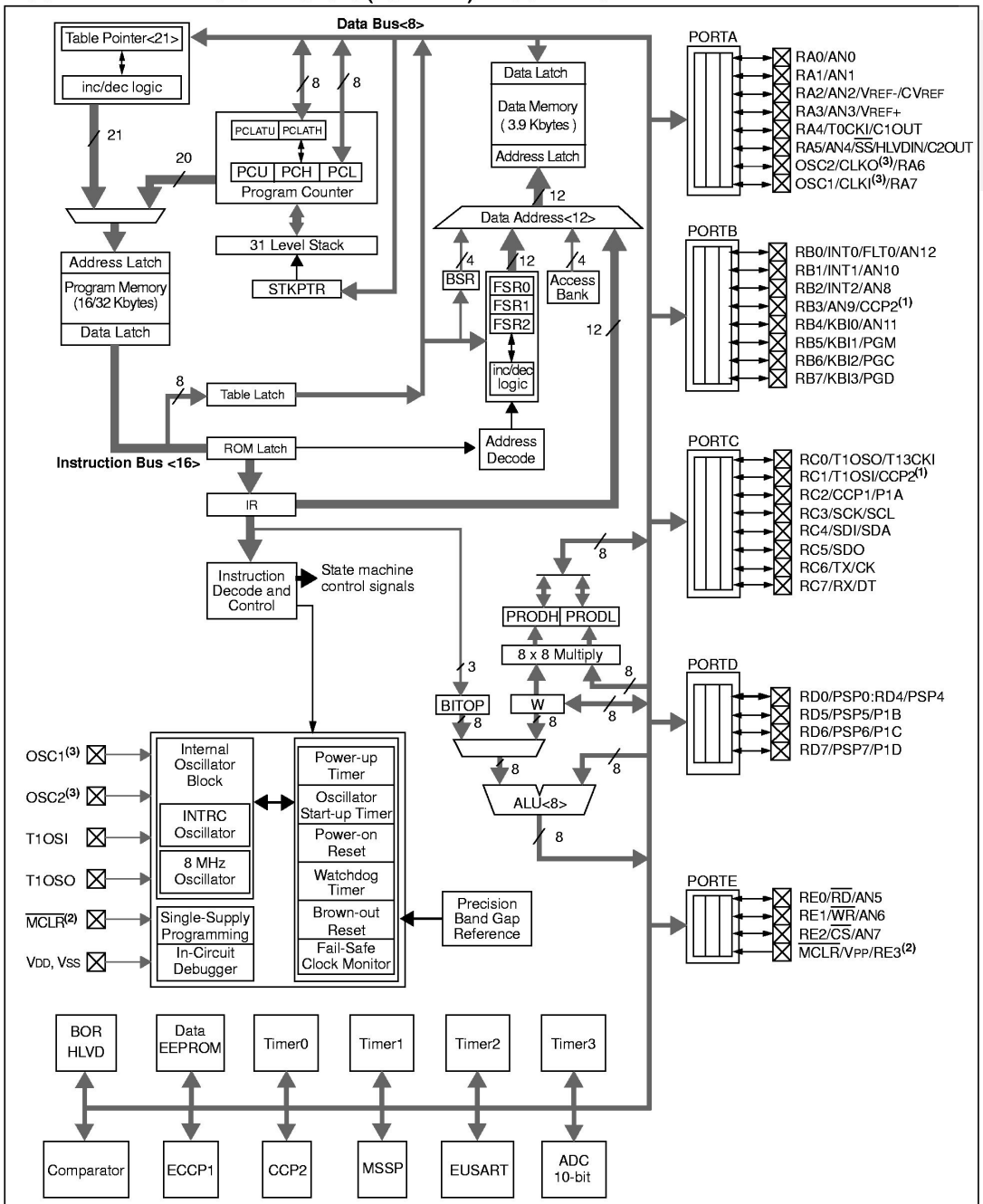
FIGURE 5-7: USE OF THE BANK SELECT REGISTER (DIRECT ADDRESSING)



Note 1: The Access RAM bit of the instruction can be used to force an override of the selected bank (BSR<3:0>) to the registers of the Access Bank.

2: The `MOVFF` instruction embeds the entire 12-bit address in the instruction.

FIGURE 1-2: PIC18F4420/4520 (40/44-PIN) BLOCK DIAGRAM



Note 1: CCP2 is multiplexed with RC1 when configuration bit CCP2MX is set, or RB3 when CCP2MX is not set.

2: RE3 is only available when MCLR functionality is disabled.

3: OSC1/CLKI and OSC2/CLKO are only available in select oscillator modes and when these pins are not being used as digital I/O. Refer to **Section 2.0 "Oscillator Configurations"** for additional information.

Exécution d'une instruction

Adressage direct (étendu)

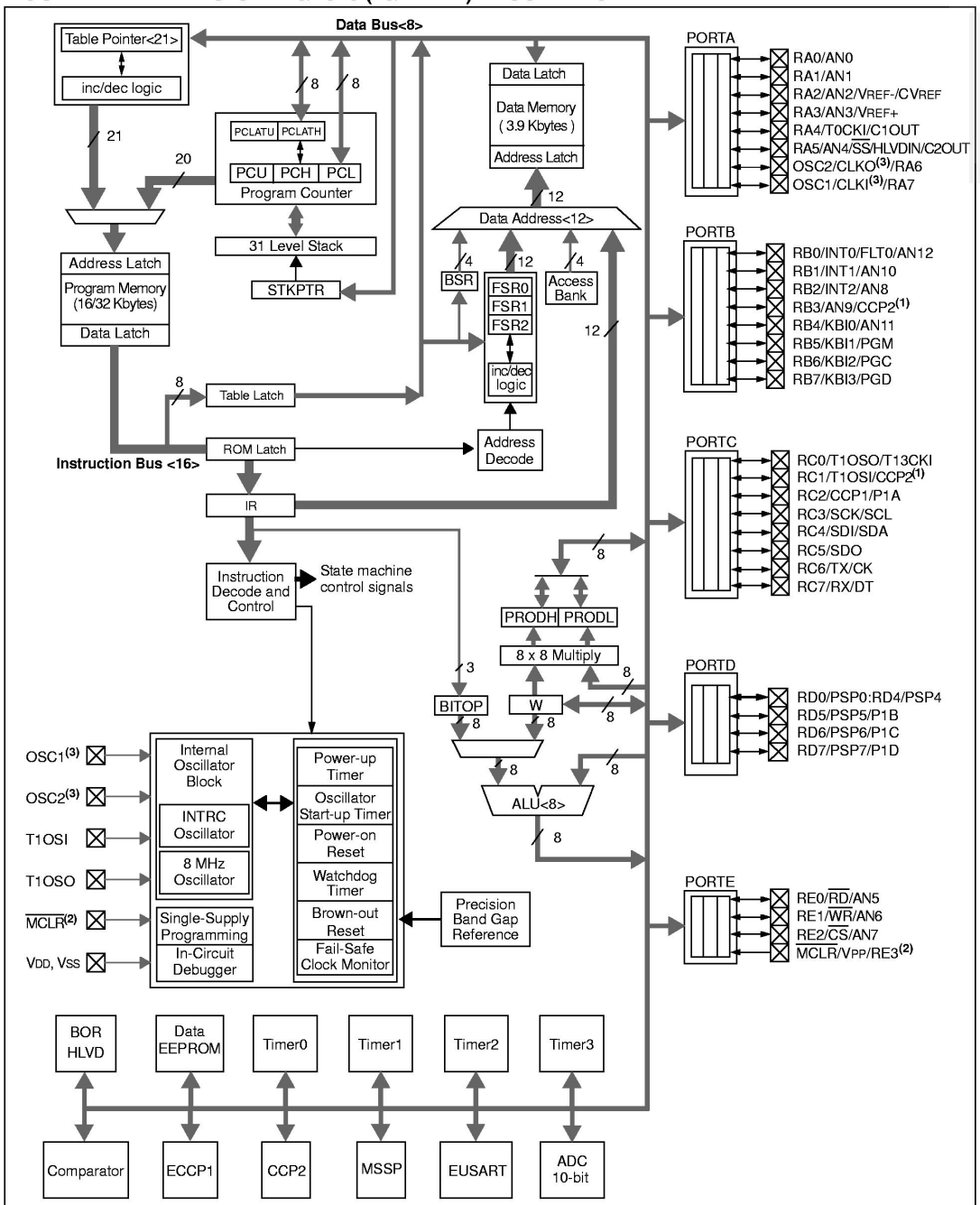
L'instruction comporte une opérande qui indique l'adresse mémoire sur laquelle s'effectue l'opération.

Exemples : CLRF (direct), MOVFF (étendu)

Déroulement:

(1) Lecture de l'instruction dans la mémoire programme à l'adresse pointée par le compteur programme. (2) Lecture de l'instruction et décodage. (3) Pour l'adressage direct, l'opérande constitue la partie basse de l'adresse mémoire sur laquelle s'effectue l'opération, la partie haute est complétée avec le registre BSR. (3') Pour l'adressage étendu, l'opérande est l'adresse complète de la case mémoire sur laquelle s'effectue l'opération. (4) Finalement l'instruction est exécutée sur la case mémoire pointée.

FIGURE 1-2: PIC18F4420/4520 (40/44-PIN) BLOCK DIAGRAM



Note 1: CCP2 is multiplexed with RC1 when configuration bit CCP2MX is set, or RB3 when CCP2MX is not set.

2: RE3 is only available when $\overline{MCLR}$ functionality is disabled.

3: OSC1/CLKI and OSC2/CLKO are only available in select oscillator modes and when these pins are not being used as digital I/O. Refer to **Section 2.0 "Oscillator Configurations"** for additional information.

Exécution d'une instruction

Adressage indirect (indexé)

L'instruction comporte une opérande indiquant un pointeur, c.à.d. une adresse de la case mémoire sur laquelle s'effectue l'opération.

Exemples : ADDWF, INDF1, 1

Déroulement:

(1) Lecture de l'instruction dans la mémoire programme à l'adresse pointée par le compteur programme. (2) Lecture de l'instruction et décodage. (3) La valeur de l'opérande indique le pointeur à utiliser. (4) La valeur pointée est lue (avec un éventuel décalage en mémoire). (5) Finalement l'instruction est exécutée sur la valeur pointée.

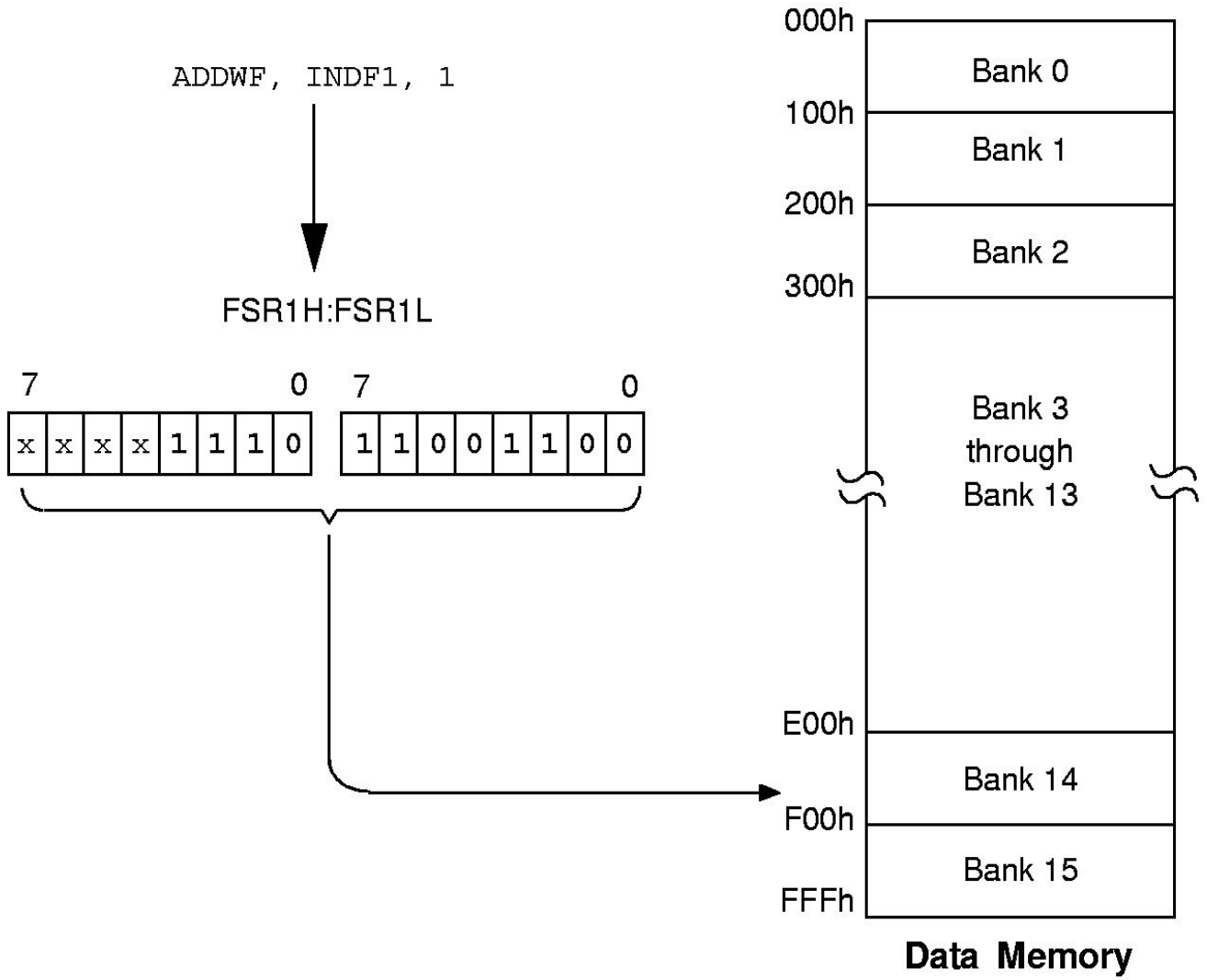
FIGURE 5-8: INDIRECT ADDRESSING

Using an instruction with one of the indirect addressing registers as the operand....

...uses the 12-bit address stored in the FSR pair associated with that register....

...to determine the data memory location to be used in that operation.

In this case, the FSR1 pair contains ECCh. This means the contents of location ECCh will be added to that of the W register and stored back in ECCh.



Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Conception d'un système embarqué

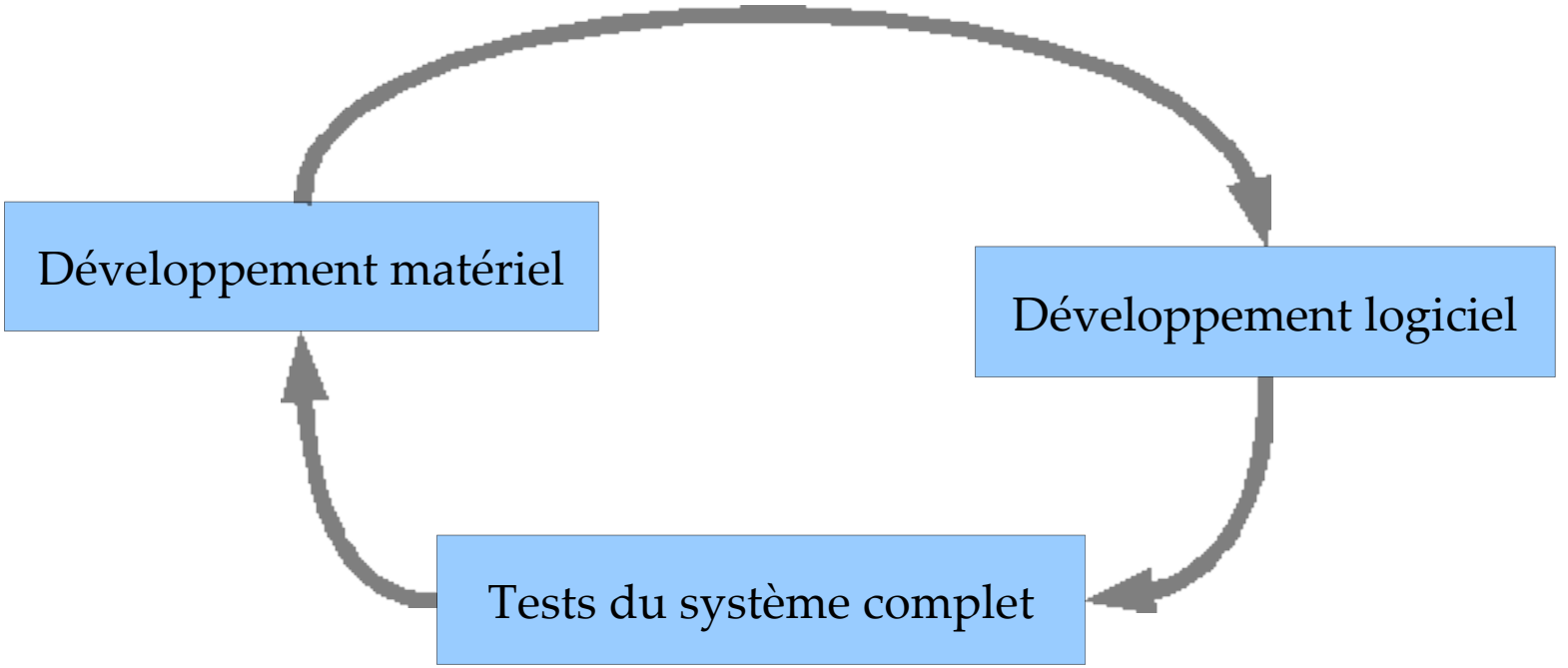
Formellement, la conception d'un système embarqué basé sur un microcontrôleur peut être décomposée en 3 étapes distinctes.

(1) **Le développement matériel** s'appuie sur un *cahier des charges*, *c.à.d.* la définition des fonctionnalités et des performances du système. Cette étape doit permettre de spécifier les *caractéristiques* du microcontrôleur, de ses périphériques et de l'électronique associée.

(2) **Le développement logiciel** s'appuie sur l'étape précédente pour construire un algorigramme, puis le code qui va être testé. Cette étape requiert que vous choisissiez le langage (assembleur et/ou évolué) que vous utiliserez sur des bases *objectives*, par exemple de manière à optimiser le temps de développement, la facilité de maintenance, le nombre d'opérations, *etc.*

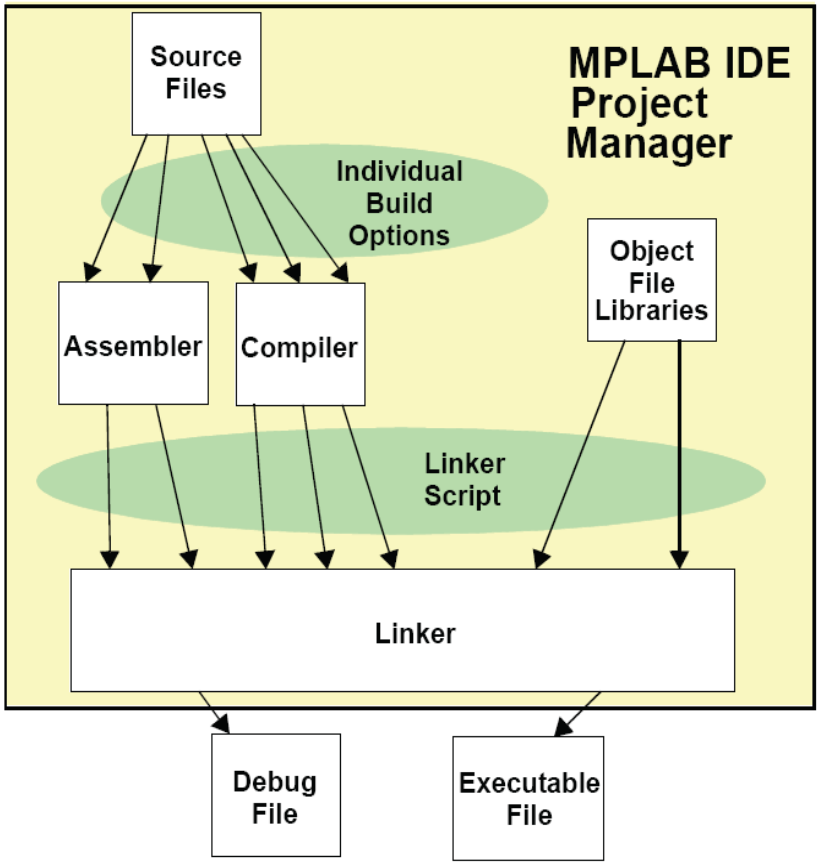
(3) **La phase de test** doit être menée pour vérifier que le cahier des charges initial est bien rempli. Cette phase de test « finale » n'empêche pas d'avoir mené des tests séparés lors des phases de développement matériel et logiciel.

Chacune de ces trois étapes précédente peut être relativement complexe et mobiliser des moyens financiers et humains conséquents. Par ailleurs, le test à une étape peut remettre en cause les choix fait à une étape précédente : en pratique, on est donc plutôt confronté à un **cycle de conception** plutôt qu'à un enchaînement parfaitement séquentiel !



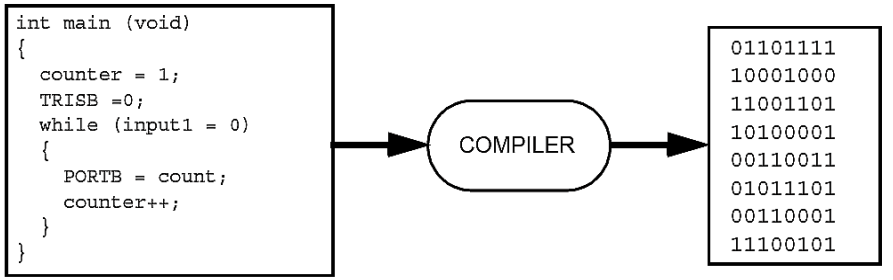
Le développement du logiciel

La construction d'un code machine exécutable s'appuie sur un certain nombre de composantes (fichiers sources, librairies) qui suivent le diagramme organisationnel ci-dessous.



(1) **Les fichiers sources** écrits dans un langage assembleur et/ou évolué doivent permettent au système embarqué d'effectuer les tâches requises.

(2) **Le compilateur et/ou l'assembleur** a pour rôle de convertir les instructions des sources en langage machine.

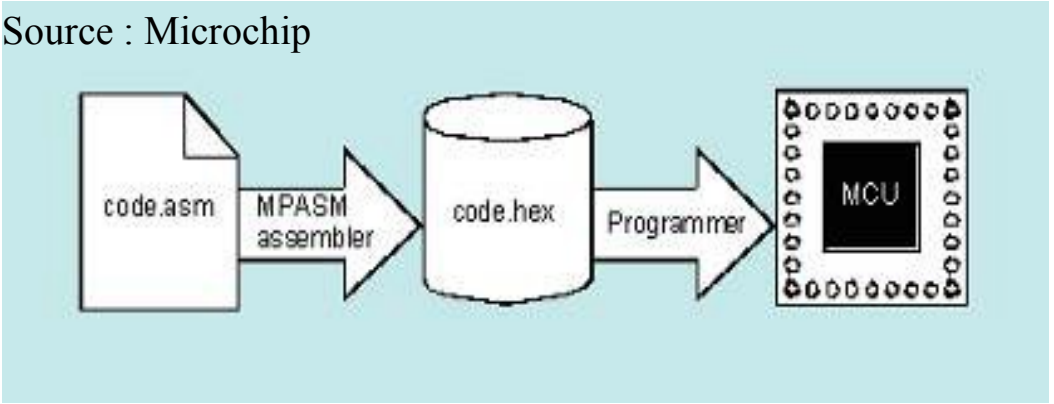


(3) **L'éditeur de lien** permet de construire un exécutable à partir des objets issus soit des sources soit de librairies pré-existantes.

La programmation en Assembleur

Le langage Assembleur (abrégé ASM) **est un langage de programmation de bas-niveau**, qui fait la correspondance entre des instructions en langage machine (mots binaires) et des symboles appelés *mnémoniques* plus simples à utiliser.

Source : Microchip



Le langage Assembleur est un langage compilé, c'est à dire :

1. L'utilisateur écrit son programme en langage Assembleur. Ce fichier est assemblé pour traduire le programme en langage machine (avec éventuellement des améliorations).

2. Le programme en langage machine est alors utilisé pour programmer le micro-contrôleur, c.à.d. qu'il est transféré dans la mémoire (programme) pour être exécuté.

Les types d'instructions en Assembleur

A. Les instructions propres au micro-contrôleur :

- Les instructions de transfert : movlw, movf, ...
- Les instructions arithmétiques : decf, addwf, ...
- Les instructions logiques : xorlw, andlw, ...
- Les instructions de branchement : bz (*branch if zero*), bra (*branch always*), ...

B. Les instructions pré-processeur permettent au programmeur de donner des indications au compilateur, elles sont destinées au PC et non pas au micro-contrôleur !

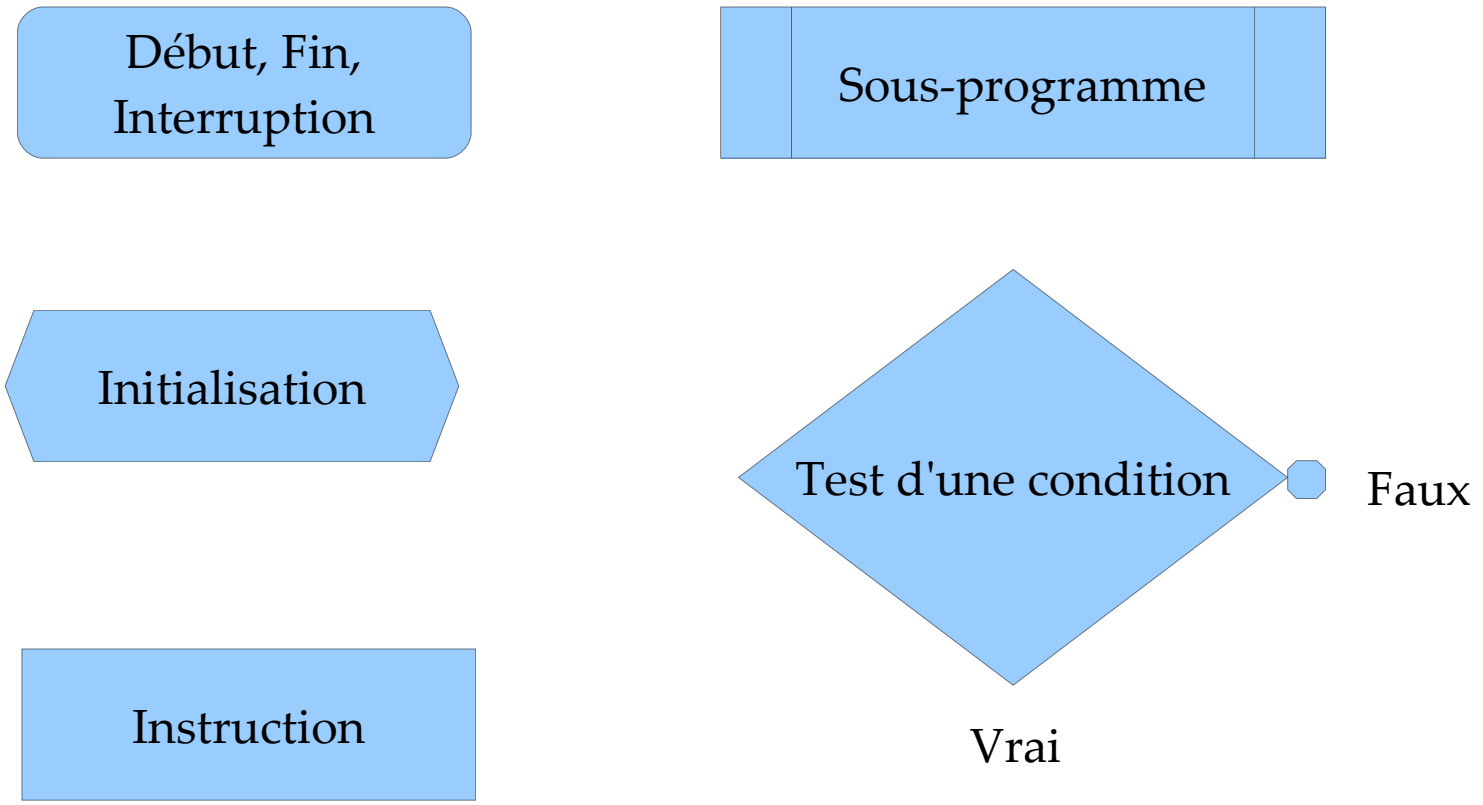
Il existe différents types d'instruction pré-processeur :

- les instructions de contrôle : org = début du programme, end = fin du programme, *etc.* ;
- les instructions conditionnelles : if, else, endif, *etc.* ;
- les instructions relatives aux données : res = réservation d'espace mémoire, *etc.* ;
- les instructions pour les macros

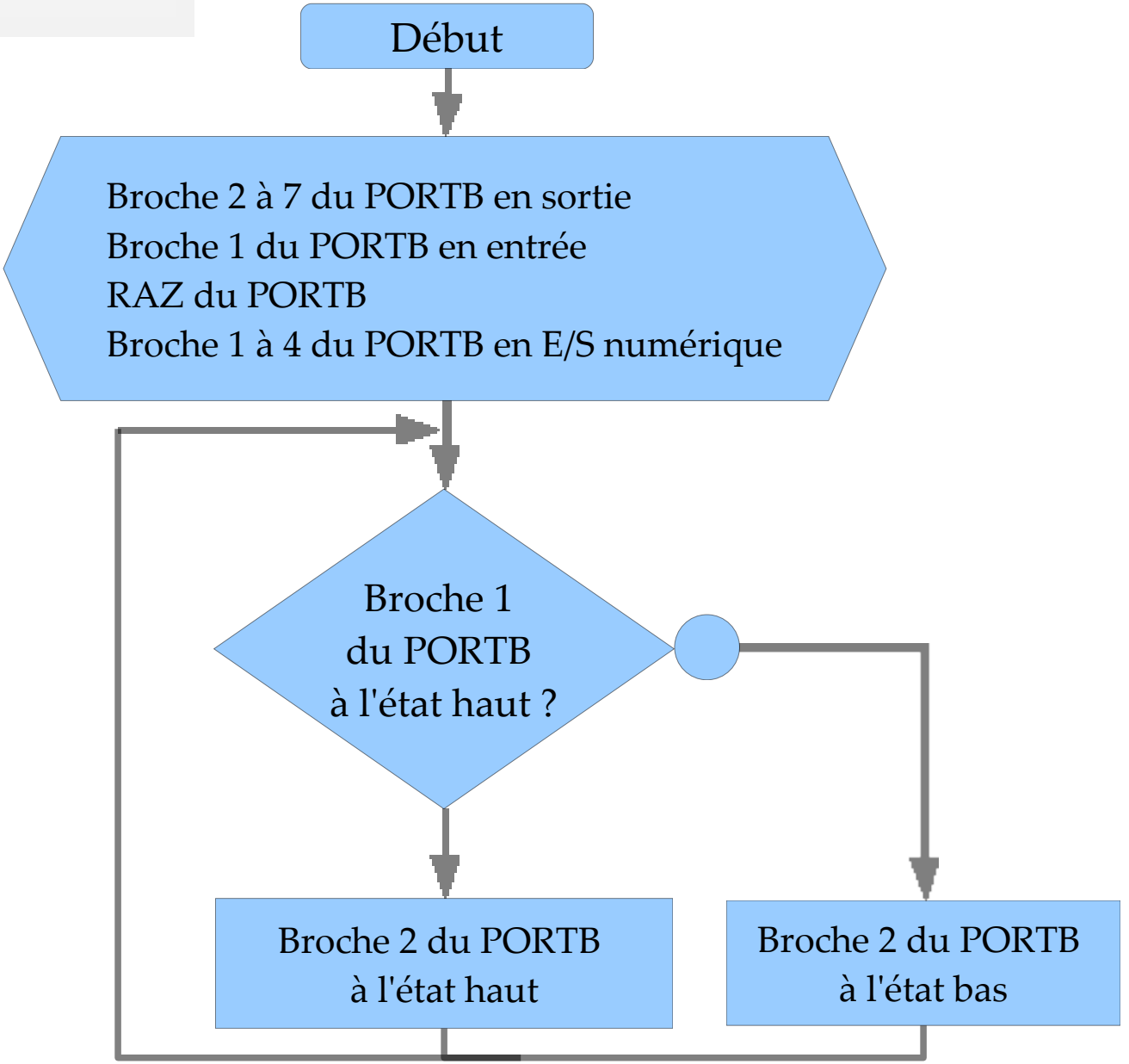
Algorigrammes

La description du programme par un algorigramme permet de :

- **gagner en efficacité** lors de la phase de codage du programme,
- d'**optimiser la structure** du programme,
- de **clarifier le fonctionnement** du programme,
- **le rendre compréhensible** à une personne extérieure.



Premier programme en assembleur



Structure d'un programme en assembleur

Dans un programme en assembleur, on peut distinguer une **partie préliminaire** qui est systématique c.à.d. qui ne change pas d'un programme à l'autre.

Début

```
;
; Filename : premier_programme.asm
;
; Description :Recopie de l'état de la broche 1 du PORTB
; sur la broche 2 du PORTB
;
; Author:   Eric Magraner
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2006/07

list p=18f4520
; Définition du micro-contrôleur utilisé

#include <p18f4510.inc>
; Définitions des emplacements mémoires des registres

; La configuration du micro-contrôleur est définie avec MPLAB
; (Logiciel de développement Microchip)
```

La première partie concerne l'en-tête qui définit, le plus clairement possible, la fonction du programme ainsi que divers informations permettant de gérer l'historique du code (auteur, date d'écritures et de modifications, numéro de version, *etc.*)

```
; Filename : premier_programme.asm
;
; Description : Recopie de l'état de la broche 1 du PORTB
; sur la broche 2 du PORTB
;
; Author:   Eric Magraner
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2006/07
```

Début

```
list p=18f4520
; Définition du micro-contrôleur utilisé

#include <p18f4510.inc>
; Définitions des emplacements mémoires des registres
; et configurations matérielles par défaut

#include <MA_CONFIG.inc>
; Modification des configurations matérielles par défaut
```

La déclaration du micro-contrôleur permet au compilateur de générer un code machine qui soit compréhensible pour le microcontrôleur que vous souhaitez programmer.

```
;
; Filename : premier_programme.asm
;
; Description : Recopie de l'état de la broche 1 du PORTB
; sur la broche 2 du PORTB
;
; Author:   Eric Magraner
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2006/07
```

Début

```
list p=18f4520
; Définition du micro-contrôleur utilisé
```

```
#include <p18f4510.inc>
; Définitions des emplacements mémoires des registres
; et configurations matérielles par défaut

#include <MA_CONFIG.inc>
; Modification des configurations matérielles par défaut
```

Une directive au pré-processeur demande l'inclusion d'un fichier de définition spécifique au microcontrôleur qui définit certaines configurations matérielles par défaut et permet de simplifier l'écriture des programmes, cf. transparent suivant.

```
; Filename : premier_programme.asm
;
; Description : Recopie de l'état de la broche 1 du PORTB
; sur la broche 2 du PORTB
;
; Author:   Eric Magraner
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2006/07

list p=18f4520
; Définition du micro-contrôleur utilisé
```

Début

```
#include <p18f4510.inc>
; Définitions des emplacements mémoires des registres
; et configurations matérielles par défaut
```

```
#include <MA_CONFIG.inc>
; Modification des configurations matérielles par défaut
```

Extrait du fichier p18f4510.inc de définitions propre au micro-contrôleur

```
;----- Register Files
PORTA              EQU  H'0F80'
PORTB              EQU  H'0F81'
PORTC              EQU  H'0F82'
PORTD              EQU  H'0F83'
PORTE              EQU  H'0F84'
LATA               EQU  H'0F89'
LATB               EQU  H'0F8A'
LATC               EQU  H'0F8B'
LATD               EQU  H'0F8C'
LATE               EQU  H'0F8D'
  .                  .
  .                  .
  .                  .
```

Une directive au pré-processeur supplémentaire peut être spécifiée de manière à modifier la configuration par défaut établie dans p18f4510.inc.

Note : pour savoir comment modifier ces configurations, il faut aller voir le fichier p18f4510.inc.

```
; Filename : premier_programme.asm
;
; Description :Recopie de l'état de la broche 1 du PORTB
; sur la broche 2 du PORTB
;
; Author:   Eric Magraner
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2006/07

list p=18f4520
; Définition du micro-contrôleur utilisé

#include <p18f4510.inc>
; Définitions des emplacements mémoires des registres
; et configurations matérielles par défaut

; #include <MA_CONFIG.inc>
; Modification des configurations matérielles par défaut
```

Début

Exemple de ce que pourrait être le fichier MA_CONFIG.inc

```
;----- Utilisation de l'oscillateur
;----- en mode haute vitesse
CONFIG OSC = HS
```

On peut distinguer ensuite une **seconde partie** qui correspond à la configuration des éléments du microcontrôleur qui entrent *directement* en jeu dans la fonction réalisée...

La première opération consiste systématiquement à **initialiser le vecteur RESET**. Notez que cette étape n'a pas été notée dans l'algorigramme (ce qui pourrait être considéré comme une lacune...).

Broche 2 à 7 du PORTB en sortie
Broche 1 du PORTB en entrée
RAZ du PORTB
Broche 1 à 4 du PORTB en E/S numérique

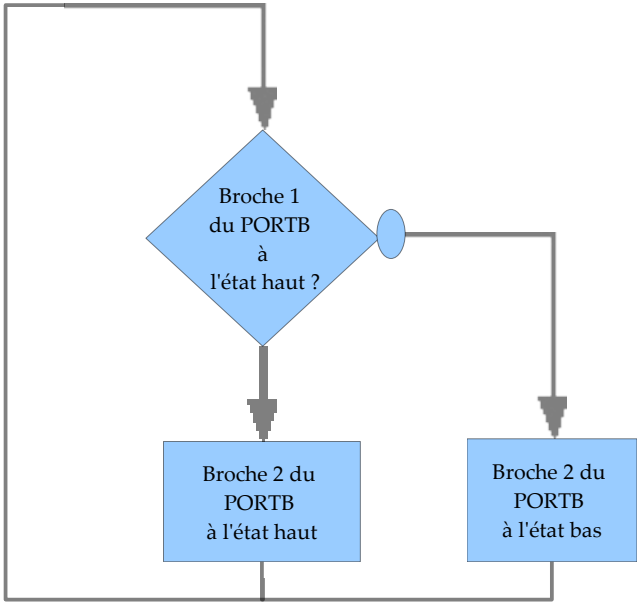
```
        org    h'0000'          ; initialisation du vecteur RESET
        goto init

init    clrf   PORTB
        movlw  b'00000001'
        movwf  TRISB            ; Configuration de la direction
                                ; du PORTB. Broche 1 en entrée.
                                ; Broche 2 à 8 en sortie
        clrf   LATB
        movlw  0Fh
        movwf  ADCON1           ; Configuration des broches 1 à 4
                                ; du PORTB en E/S numérique
```

La seconde opération correspond à la **configuration du PORT B** telle que décrite par l'algorigramme... Cette configuration est directement fourni par le *Datasheet* du PIC 18F4520...

Broche 2 à 7 du PORTB en sortie
Broche 1 du PORTB en entrée
RAZ du PORTB
Broche 1 à 4 du PORTB en E/S numérique

```
        org    h'0000'          ; initialisation du vecteur RESET
        goto init

init    clrf   PORTB
        movlw  b'00000001'
        movwf  TRISB            ; Configuration de la direction
                                ; du PORTB. Broche 1 en entrée.
                                ; Broche 2 à 8 en sortie
        clrf   LATB
        movlw  0Fh
        movwf  ADCON1           ; Configuration des broches 1 à 4
                                ; du PORTB en E/S numérique
```

La troisième partie du programme est dédiée à la réalisation de la fonction principale, c.à.d. la boucle et le test.




```
boucle      btfss   PORTB,0     ; Broche 1 du PORTB à l'état haut ?
                                ; saute l'instruction suivante si
                                ; état haut
            goto    eteindre
allumer     bsf     PORTB,1     ; Broche 2 du PORTB à l'état haut
            goto    boucle
eteindre    bcf     PORTB,1     ; Broche 2 du PORTB à l'état bas
            goto    boucle

            END
```

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Les interruptions

« Une interruption est un arrêt temporaire de l'exécution normale d'un programme informatique par le microprocesseur afin d'exécuter un autre programme (appelé routine d'interruption).

Les interruptions matérielles sont utilisées lorsqu'il est nécessaire de pouvoir réagir en temps réel à un événement asynchrone, ou bien, de manière plus générale, afin d'économiser le temps d'exécution lié à une boucle de consultation (polling loop).» (Source : *Wikipédia*)

Une interruption peut avoir différentes sources : périphérique d'entrée/sortie, *timer*, *watchdog* (*cf.* explications plus loin), ...

Les interruptions sont utilisées pour avertir le micro-contrôleur quand une condition est remplie. En utilisant les interruptions, on évite que le micro-contrôleur reste en attente inutilement (*pooling-loop*), elles permettent de gérer les événements asynchrones.

Les interruptions

Les interruptions sont, en général, contrôlées par 3 bits :

- **Un bit de flag**

indique qu'une interruption a été déclenchée et indique la source.

- **Un bit de validation**

permet à l'utilisateur d'activer ou non une interruption.

- **Un bit de priorité**

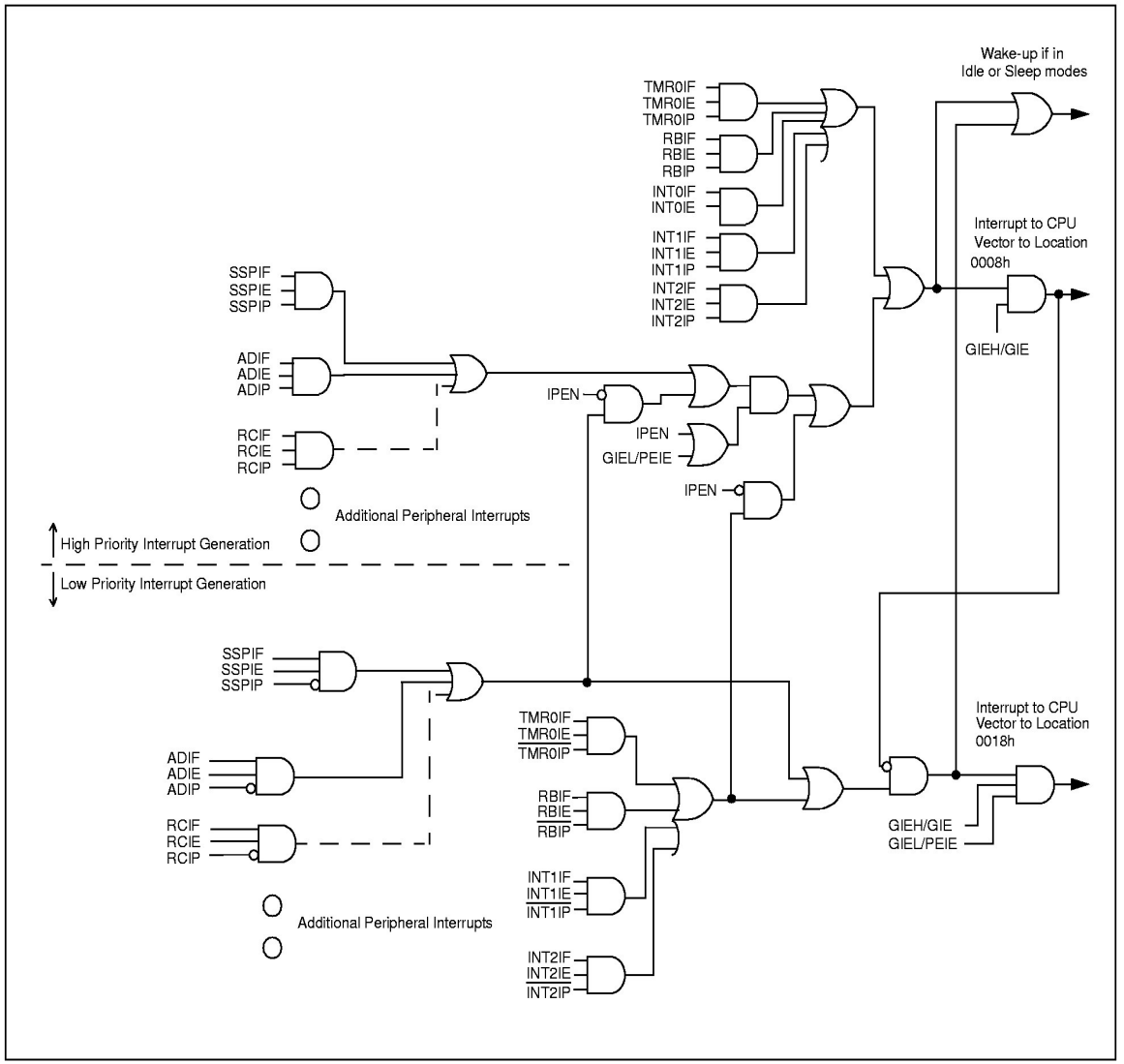
permet de sélectionner la priorité (haute/basse) de l'interruption.

Gestion des priorités :

Il existe des interruptions de priorité hautes et basses. À chaque type de priorité correspond un **vecteur d'interruption** et donc potentiellement une gestion différente des interruptions suivant leur priorité.

Schéma de la logique d'interruption

FIGURE 9-1: PIC18 INTERRUPT LOGIC



Ce schéma permet de comprendre

- le fonctionnement de la logique d'interruption,

- la priorité accordée à une interruption,

- la configuration de la logique à mettre en place pour l'application souhaitée

On notera notamment que si une interruption de haute priorité est en concurrence avec une interruption de basse priorité, l'interruption de haute priorité « prend la main ».

Déroulement d'une interruption

(1). **Réception de l'interruption :** le micro-contrôleur reçoit une interruption.

(2). **Sauvegarde des données (sauvergarde du contexte) :** le micro-contrôleur sauve une partie variable (en fonction du type d'interruption) de son état interne dans la pile, notamment l'adresse dans la mémoire programme où le micro-contrôleur s'est arrêté.

(3). **Lecture de l'adresse du vecteur d'interruption et chargement dans le PC.**

(4). **Exécution de la routine d'interruption,**

> **Attention !!** L'utilisateur doit penser à effectuer une **sauvegarde de données** du programme principal pour ne pas les effacer pendant la routine d'interruption et également à **supprimer le flag d'interruption** qui a déclenché l'interruption.

(5). **Rétablissement des données :** le micro-contrôleur rétablit les données stockées dans la pile.

(6). **Le micro-contrôleur reprend son fonctionnement normal...**

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

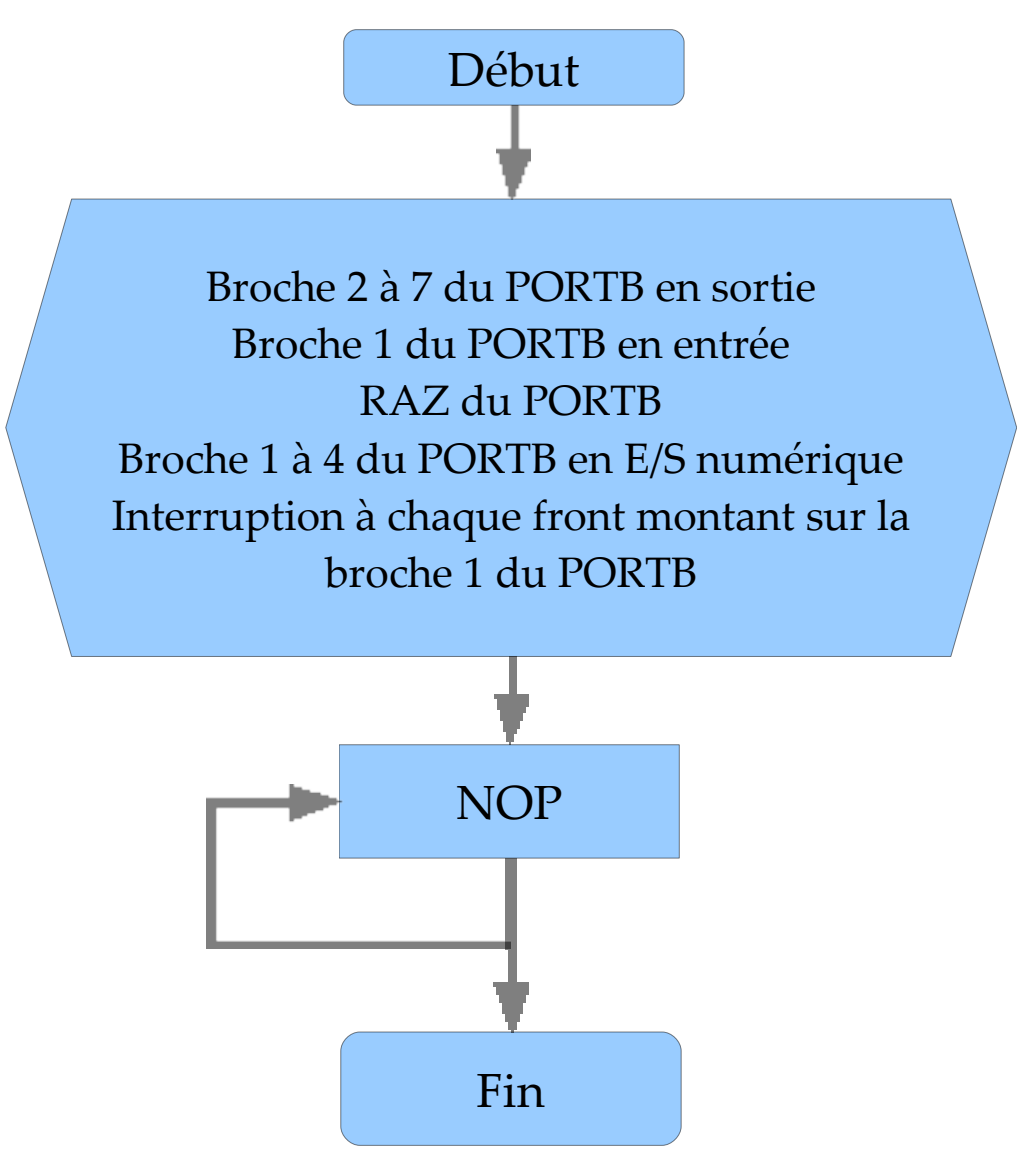
Présentation des interruptions

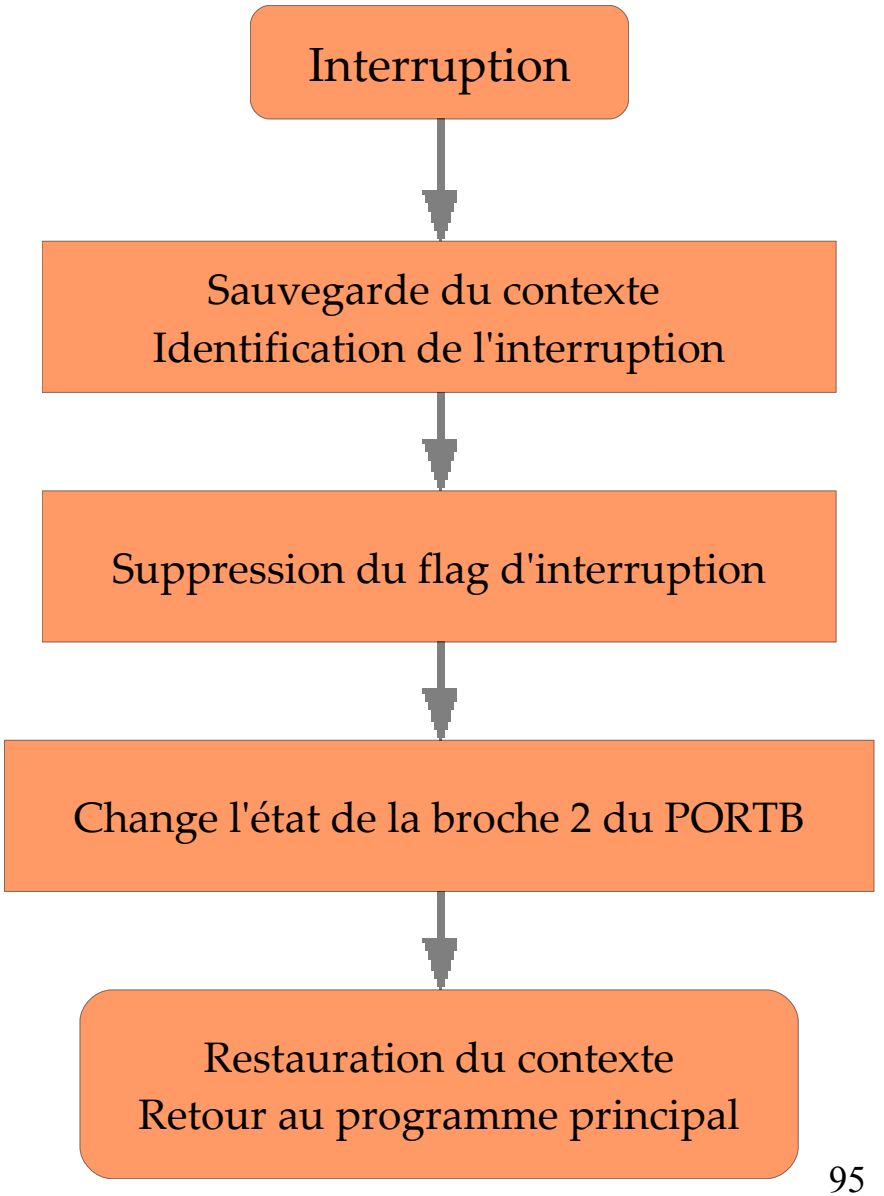
Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Premier programme avec interruption





Le début d'un programme en assembleur, avec interruption, reste très proche de celui d'une version sans interruption. On peut tout de même remarquer des directives de **réservation d'emplacements mémoire** en prévision de la sauvegarde du contexte lors de l'interruption.

```
; Filename : premier_programme_interruption.asm
; Change l'état de la broche 2 du PORTB à chaque front
; montant sur la broche 1 du PORTB (gestion par interruption
; Author:      Eric Magraner
; Company:     Université Paul Cézanne
; Revision:    1.00
; Date:        2006/07

list p=18f4520
; Définition du micro-contrôleur utilisé
```

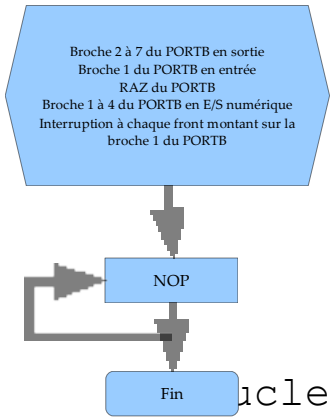
Début

```
#include <p18f4510.inc>
; Définitions des emplacements mémoires des registres
; et configurations matérielles par défaut

#include <MA_CONFIG.inc>
; Modification des configurations matérielles par défaut
```

```
W_TEMP          RES 1   ; Réservation d'un octet en mémoire
STATUS_TEMPRES 1   ; Réservation d'un octet en mémoire
BSR_TEMP        RES 1   ; Réservation d'un octet en mémoire
```

Du code du programme principal, on distingue les étapes classiques d'initialisation du vecteur RESET et du PORT B. On note aussi les parties propres aux interruptions : **initialisation du vecteur et du registre d'INTERRUPTION**.

```
org     h'0000'             ; Init. du vecteur RESET
        goto    init

        org     h'0008'     ; Init. du vecteur INTERRUPTION
        goto    routine_interruption

init    clrf    PORTB
        movlw   b'00000001'
        movwf   TRISB           ; Config. de la dir. du PORTB
        clrf    LATB
        movlw   0Fh
        movwf   ADCON1      ; Broche 1à4 du PORTB en E/S num.
        movlw   b'10010000' ; 0x90 -> w
        movwf   INTCON      ; w -> INTCON (Init. du registre d'interrup.)
```




```
ucle nop
        goto    boucle

        END
```

Le registre d'interruption INTCON permet, *d'une part* d'activer les interruptions (bit 7), et *d'autre part* d'activer le mode interruption externes INT0 (bit 4). Dans ce cas, l'interruption sera détectée sur la broche 0 du port B (*cf. datasheet*).

REGISTER 9-1: INTCON REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

bit 7 GIE/GIEH: Global Interrupt Enable bit
When IPEN = 0:
1 = Enables all unmasked interrupts
0 = Disables all interrupts
When IPEN = 1:
1 = Enables all high priority interrupts
0 = Disables all interrupts

bit 6 PEIE/GIEL: Peripheral Interrupt Enable bit
When IPEN = 0:
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts
When IPEN = 1:
1 = Enables all low priority peripheral interrupts
0 = Disables all low priority peripheral interrupts

bit 5 TMR0IE: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 overflow interrupt
0 = Disables the TMR0 overflow interrupt

bit 4 INT0IE: INT0 External Interrupt Enable bit
1 = Enables the INT0 external interrupt
0 = Disables the INT0 external interrupt

bit 3 RBIE: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 TMR0IF: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 INT0IF: INT0 External Interrupt Flag bit
1 = The INT0 external interrupt occurred (must be cleared in software)
0 = The INT0 external interrupt did not occur

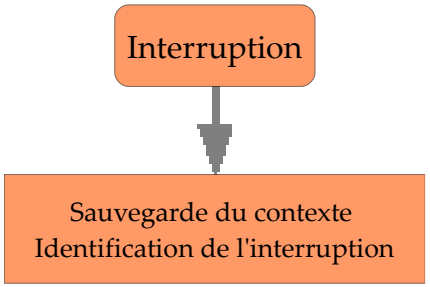
bit 0 RBIF: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Note: A mismatch condition will continue to set this bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared.

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

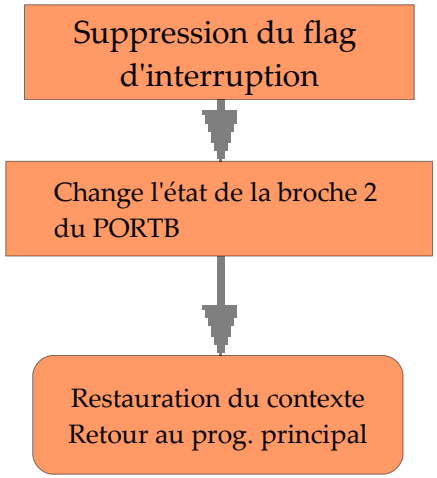
Le déclenchement d'une interruption conduit le microcontrôleur à sauver l'adresse de l'instruction courante dans la pile, puis à charger le vecteur d'interruption dans le PC.

Dès lors, il est **systématiquement** nécessaire de (1) **sauvegarder le contexte** et (2) **identifier l'origine de l'interruption**.



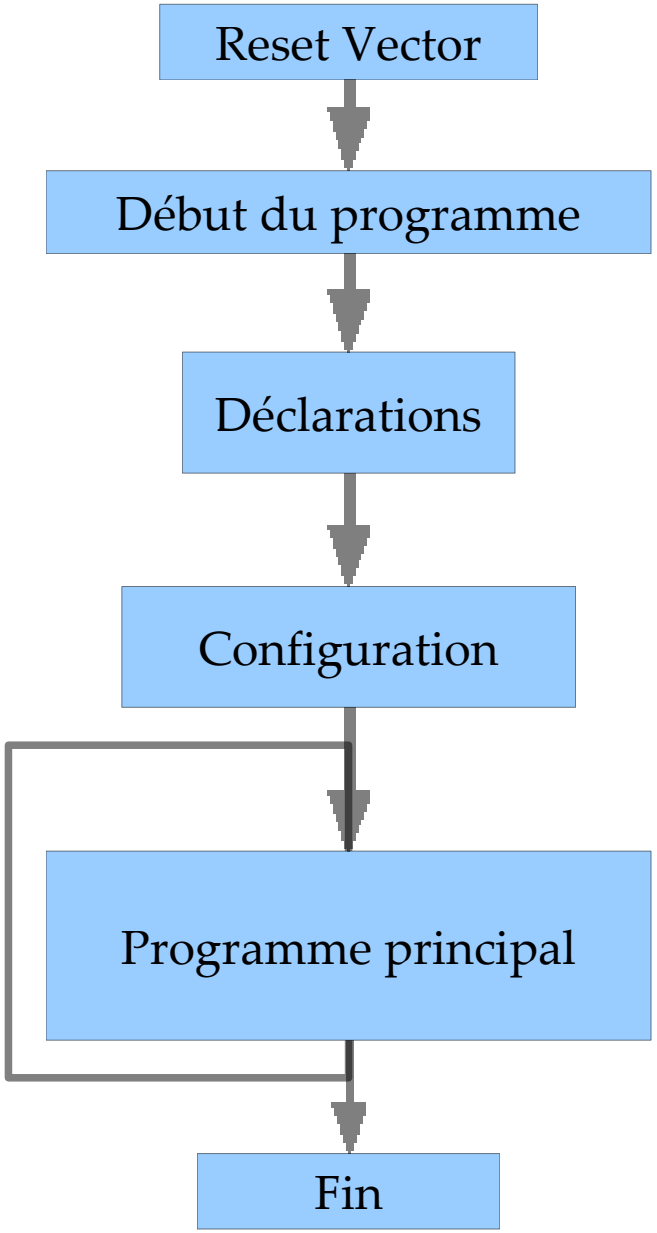

```
routine_interruption

    ; Sauvegarde du contexte
    movwf   W_TEMP                  ; Sauvegarde de W
    movff   STATUS, STATUS_TEMP     ; Sauvegarde de STATUS
    movff   BSR, BSR_TEMP           ; Sauvegarde de BSR

    ; identification de l'origine de l'interruption
    btfsc   INTCON,1
    goto    interruption_INT0
    bra     restauration_contexte
```

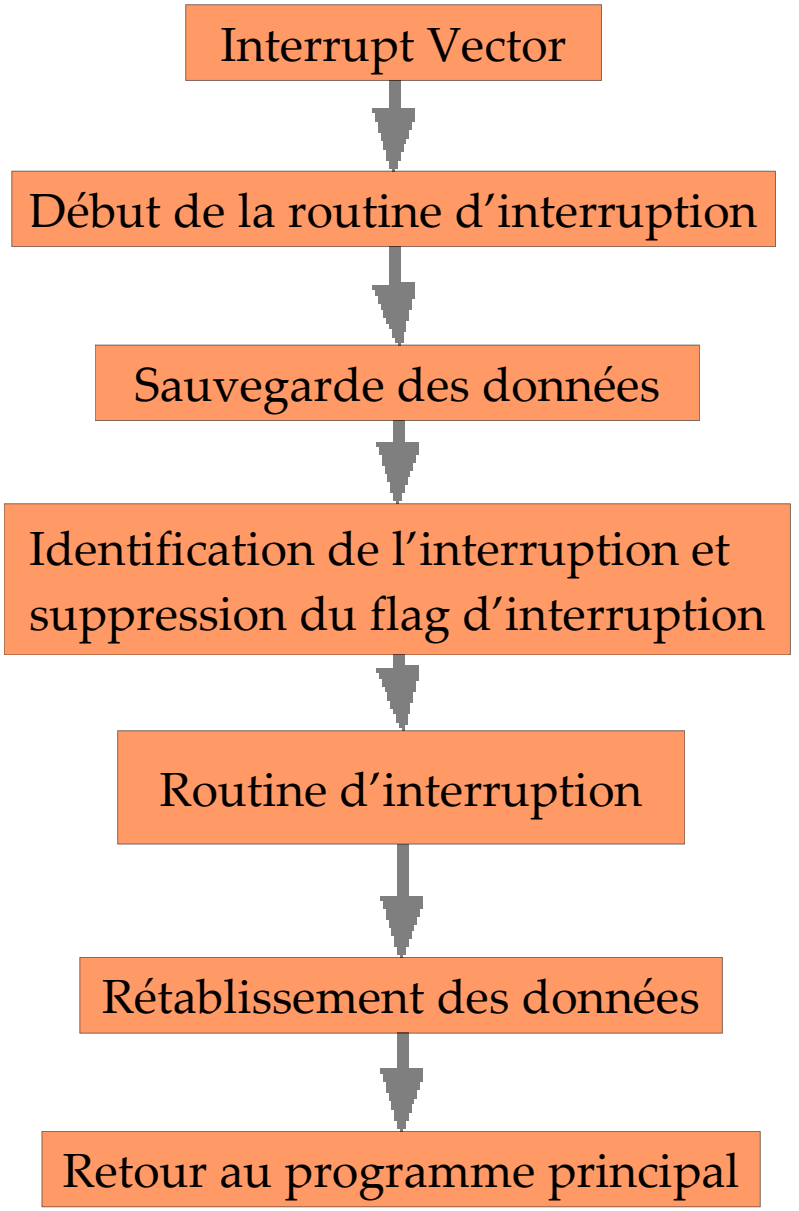
Il faut ensuite **systématiquement** (3) mettre à **zéro le bit d'interruption** puis, (4) **exécuter la fonction** pour laquelle l'interruption a été prévue, et enfin (4) faire la **restauration du contexte** (5) et retourner au programme principal.




```
interruption_INT0

bcf     INTCON,1    ; Suppression du flag d'interruption

movlw   0x02        ; 0x02 -> w
xorwf   PORTB       ; w xor PORTB -> PORTB
goto    restauration_contexte

  ;   Restauration du contexte
restauration_contexte
movff   BSR_TEMP, BSR           ; Restauration de BSR
movff   W_TEMP, W               ; Restauration de W
movff   STATUS_TEMP, STATUS     ; Restauration de STATUS

retfie
```

Structure générale d'un programme





Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le microcontrôleur / spécificité pour le PIC 18F4520

Fonctions intégrées du PIC 18F4520

Les microcontrôleurs intégrent des fonctionnalités qu'il est souvent utile de connaître pour gagner du temps de développement. Par exemple, le PIC18F4520 intègre les « modules » suivants :

- **Les compteurs** ..... *Timer*
- Les modules « Capture Compare PWM » ..... *CCP*
- Les comparateurs ..... *Comparator*
- Les modules de conversion analogique/numérique ..... *CAN/CNA*
- *Les chiens-de-garde* ..... *Watchdog*
- *Les différents modes de gestion de l'alimentation*

Une présentation complète des différentes fonctions sort du cadre de ce cours. Nous nous limiterons à ici à la présentation du module TIMER que nous utiliserons en TP...

La fonctionnalité « Timer »

Les timers sont des registres incrémentés à chaque réalisation d'un événement, la valeur de ces registres pouvant être pré-positionnée à une valeur initiale.

Les événements qui *commandent* l'incrémentation sont

- un cycle d'horloge, c'est la fonction « *timer* » ;
- un front montant sur une broche en entrée, c'est la fonction « *counter* ».

Il en découle que le module *timer* peut remplir les fonctions suivantes,

- **Utilisation « timer » :** permet de fournir une *référence temporelle* à partir de l'horloge du micro-contrôleur, notamment dans le cadre d'applications temps réel.
- **Utilisation « counter » :** sert à compter un *nombre d'événements asynchrones* sur une broche d'entrée du micro-contrôleur.

Illustration par un exemple simple...

Cahier des charges :

On cherche à utiliser le « module timer » du microcontrôleur pour faire clignoter une LED connectée sur le port RB1. La période est fixée à une fréquence de 1 Hz.

Une méthode générale...

(1) Lire dans la documentation (*data-sheet*) la section traitant du module.

(2) Déduisez-en les registres à configurer lors de la phase d'initialisation.

(3) Construisez l'algorigramme préalable à l'écriture du programme.

(4) Écrivez le programme assembleur, testez-le et débuggez-le...

Extrait du data-sheet du PIC18F4520, p. 123-125

11.0 TIMER0 MODULE

The Timer0 module incorporates the following features:

- Software selectable operation as a timer or counter in both 8-bit or 16-bit modes
- Readable and writable registers
- Dedicated 8-bit, software programmable prescaler
- Selectable clock source (internal or external)
- Edge select for external clock
- Interrupt-on-overflow

The T0CON register (Register 11-1) controls all aspects of the module's operation, including the prescale selection. It is both readable and writable.

A simplified block diagram of the Timer0 module in 8-bit mode is shown in Figure 11-1. Figure 11-2 shows a simplified block diagram of the Timer0 module in 16-bit mode.

REGISTER 11-1: T0CON: TIMER0 CONTROL REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0

bit 7 **TMR0ON:** Timer0 On/Off Control bit
1 = Enables Timer0
0 = Stops Timer0

bit 6 **T08BIT**: Timer0 8-bit/16-bit Control bit
1 = Timer0 is configured as an 8-bit timer/counter
0 = Timer0 is configured as a 16-bit timer/counter

bit 5 **T0CS**: Timer0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKO)

bit 4 **T0SE**: Timer0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA**: Timer0 Prescaler Assignment bit
1 = TImer0 prescaler is NOT assigned. Timer0 clock input bypasses prescaler.
0 = Timer0 prescaler is assigned. Timer0 clock input comes from prescaler output.

bit 2-0 **T0PS2:T0PS0**: Timer0 Prescaler Select bits
111 = 1:256 prescale value
110 = 1:128 prescale value
101 = 1:64 prescale value
100 = 1:32 prescale value
011 = 1:16 prescale value
010 = 1:8 prescale value
001 = 1:4 prescale value
000 = 1:2 prescale value

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

11.1 Timer0 Operation

Timer0 can operate as either a timer or a counter; the mode is selected with the T0CS bit (T0CON<5>). In Timer mode (T0CS = 0), the module increments on every clock by default unless a different prescaler value is selected (see **Section 11.3 "Prescaler"**). If the TMR0 register is written to, the increment is inhibited for the following two instruction cycles. The user can work around this by writing an adjusted value to the TMR0 register.

The Counter mode is selected by setting the T0CS bit (= 1). In this mode, Timer0 increments either on every rising or falling edge of pin RA4/T0CKI. The incrementing edge is determined by the Timer0 Source Edge Select bit, T0SE (T0CON<4>); clearing this bit selects the rising edge. Restrictions on the external clock input are discussed below.

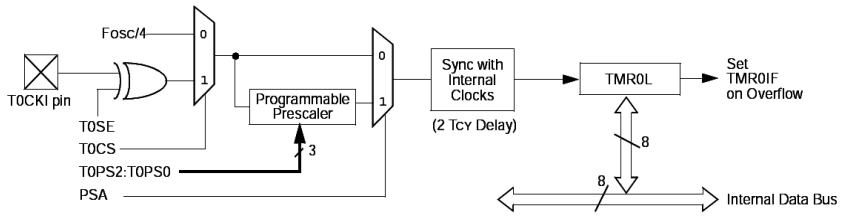
An external clock source can be used to drive Timer0; however, it must meet certain requirements to ensure that the external clock can be synchronized with the internal phase clock (TOSC). There is a delay between synchronization and the onset of incrementing the timer/counter.

11.2 Timer0 Reads and Writes in 16-Bit Mode

TMR0H is not the actual high byte of Timer0 in 16-bit mode; it is actually a buffered version of the real high byte of Timer0 which is not directly readable nor writable (refer to Figure 11-2). TMR0H is updated with the contents of the high byte of Timer0 during a read of TMR0L. This provides the ability to read all 16 bits of Timer0 without having to verify that the read of the high and low byte were valid, due to a rollover between successive reads of the high and low byte.

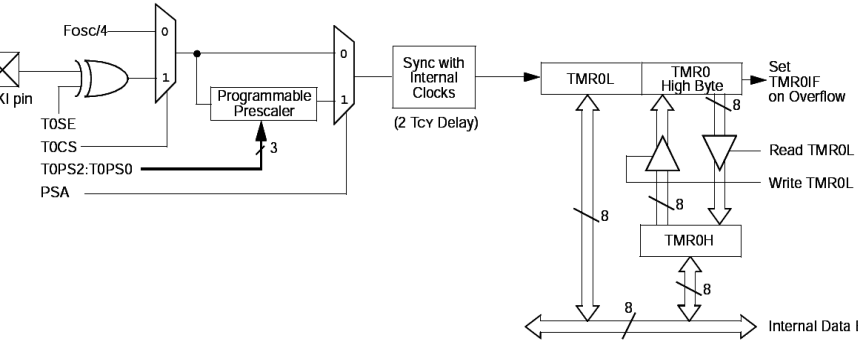
Similarly, a write to the high byte of Timer0 must also take place through the TMR0H Buffer register. The high byte is updated with the contents of TMR0H when a write occurs to TMR0L. This allows all 16 bits of Timer0 to be updated at once.

FIGURE 11-1: TIMER0 BLOCK DIAGRAM (8-BIT MODE)



Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)



Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

Extrait du data-sheet du PIC18F4520, p. 123-125

11.3 Prescaler

An 8-bit counter is available as a prescaler for the Timer0 module. The prescaler is not directly readable or writable; its value is set by the PSA and T0PS2:T0PS0 bits (T0CON<3:0>) which determine the prescaler assignment and prescale ratio.

Clearing the PSA bit assigns the prescaler to the Timer0 module. When it is assigned, prescale values from 1:2 through 1:256 in power-of-2 increments are selectable.

When assigned to the Timer0 module, all instructions writing to the TMR0 register (e.g., `CLRF TMR0`, `MOVWF TMR0`, `BSF TMR0`, etc.) clear the prescaler count.

Note: Writing to TMR0 when the prescaler is assigned to Timer0 will clear the prescaler count but will not change the prescaler assignment.

11.3.1 SWITCHING PRESCALER ASSIGNMENT

The prescaler assignment is fully under software control and can be changed "on-the-fly" during program execution.

11.4 Timer0 Interrupt

The TMR0 interrupt is generated when the TMR0 register overflows from FFh to 00h in 8-bit mode, or from FFFFh to 0000h in 16-bit mode. This overflow sets the TMR0IF flag bit. The interrupt can be masked by clearing the TMR0IE bit (INTCON<5>). Before re-enabling the interrupt, the TMR0IF bit must be cleared in software by the Interrupt Service Routine.

Since Timer0 is shut down in Sleep mode, the TMR0 interrupt cannot awaken the processor from Sleep.

TABLE 11-1: REGISTERS ASSOCIATED WITH TIMER0

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset Values on page
TMR0L	Timer0 Register, Low Byte								50
TMR0H	Timer0 Register, High Byte								50
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	49
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	50
TRISA	RA7(1)	RA6(1)	RA5	RA4	RA3	RA2	RA1	RA0	52

Legend: Shaded cells are not used by Timer0.

Note 1: PORTA<7:6> and their direction bits are individually configured as port pins based on various primary oscillator modes. When disabled, these bits read as '0'.

Questions :

(1) Expliquez comment fonctionne le module TIMER0 ? Comment peut-on l'utiliser pour faire basculer la sortie RB1 toute les 0.5 seconde ?

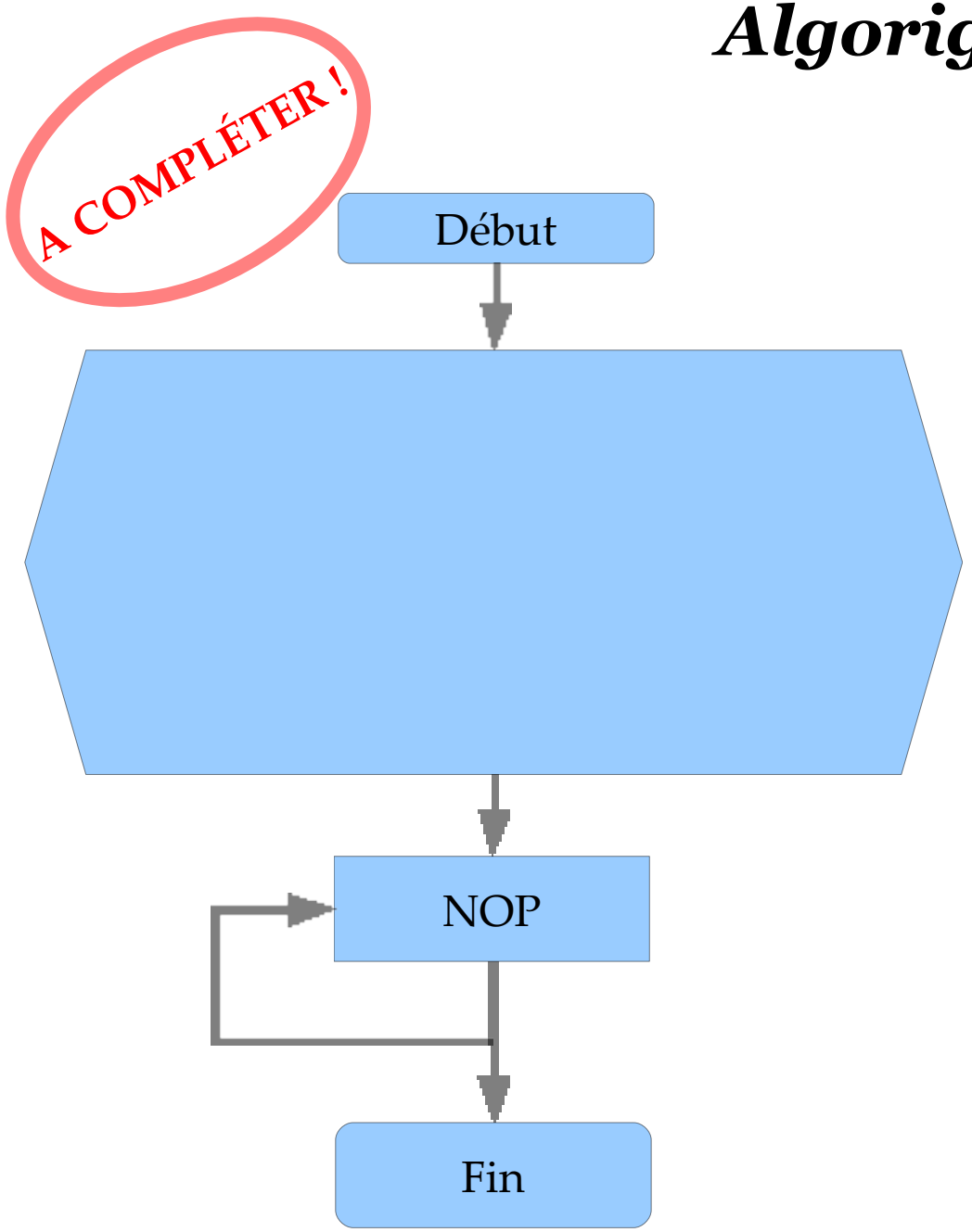
(2) Donnez les valeurs d'initialisation des différentes registres pour rentrer dans le cahier des charges.

(3) Construisez l'algorigramme préalable à l'écriture du microcode.

(4) Finalement, écrivez le programme en assembleur.

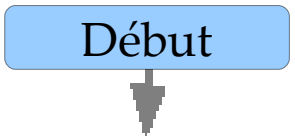
(5) Évaluez l'erreur sur la période associée au temps d'exécution du code et modifiez les registres en conséquence.

Algorigramme



Programme assembleur


```
; Filename : timer0.asm
;
; Description : Génération d'un signal carré sur la
; broche 1 du PORTB par utilisation du module TIMER0
;
; Author:
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2007/09
```



Programme assembleur [suite]

Capture, Compare, PWM

Les modules CCPM possède trois modes de fonctionnement :

- **capture**

  Le mode *capture* déclenche une action si un événement pré-déterminé apparaît (ex : changement d'état sur une broche). Utilisé avec les timers, ce module peut compter les temps d'arrivées.

- **compare**

  Le mode *compare* effectue une comparaison permanente entre le contenu d'un timer et une valeur donnée pour déclencher une action si ces contenus sont égaux.

- **Pulse width modulation (PWM)**

  Le mode PWM génère un signal rectangulaire de fréquence et de rapport cyclique choisis par l'utilisateur.

Illustration par un exemple simple…

Cahier des charges :

On cherche à utiliser le « module CCP » du microcontrôleur pour générer un signal rectangulaire sur la broche RC2. La période du PWM est fixée à une fréquence de 600 Hz et le rapport cyclique à 0,5.

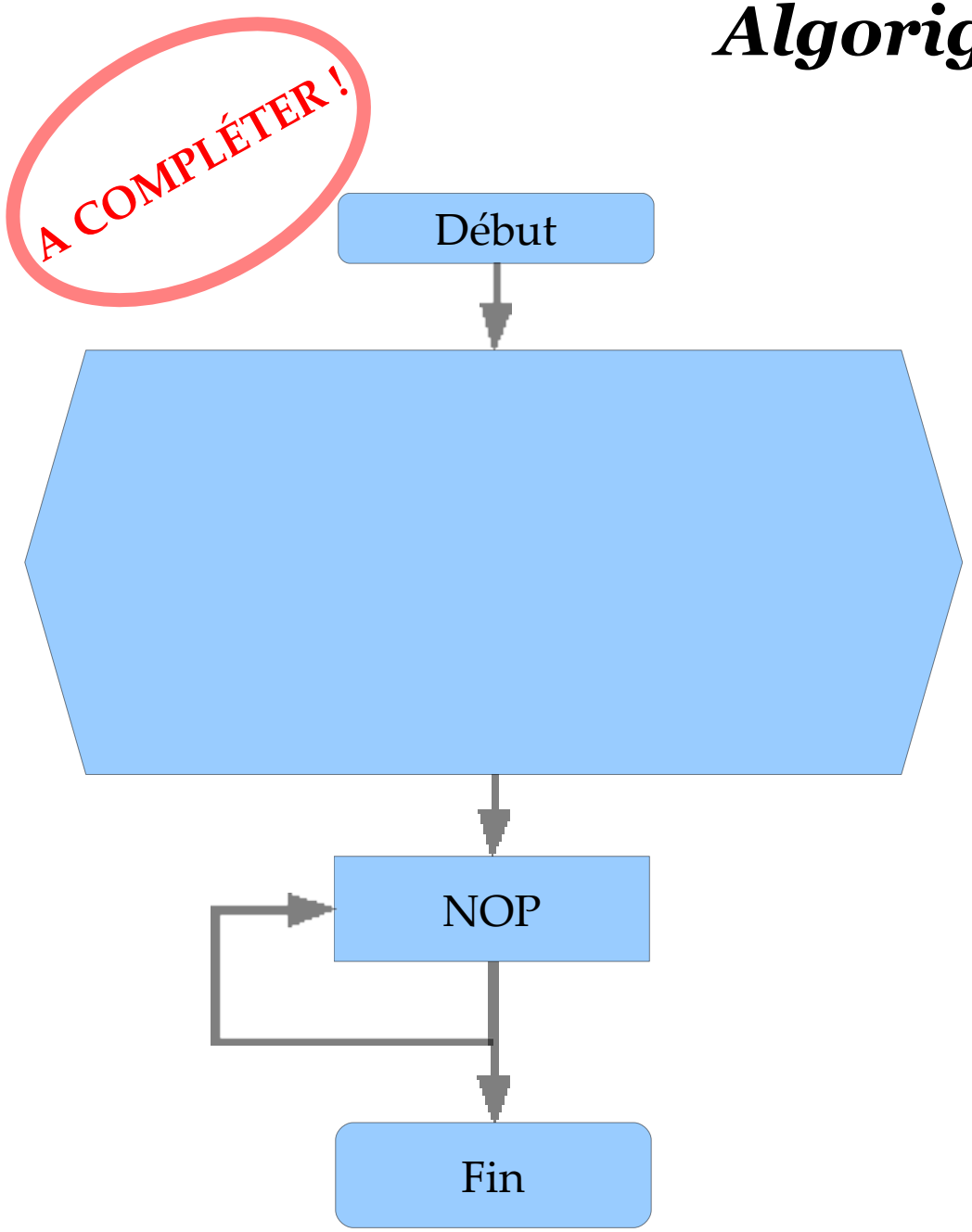
Questions :

(1) Expliquez comment les modules CCP1 et TIMER2 fonctionnent ensemble pour produire un signal rectangulaire de *période* et de *rapport cyclique* donné.

(2) Identifiez les registres à initialiser et les valeurs associées.

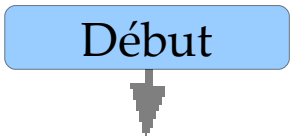
(3) Construisez l'algorigramme et écrivez le programme assembleur.

Algorigramme



Programme assembleur


```
; Filename : pwm0.asm
;
; Description : Génération d'un signal carré sur la
; broche 2 du PORTC par utilisation du module PWM
;
; Author:
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2007/09
```



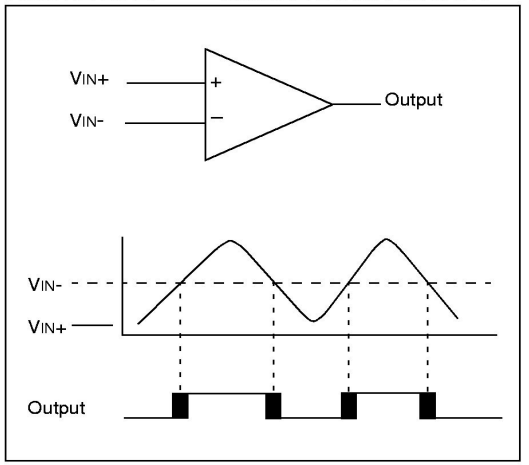
Les comparateurs

Les comparateurs permettent de comparer le signal analogique présent sur un broche du micro-contrôleur à une *valeur de référence*.

Cette valeur de référence peut être

- **soit un signal *analogique*** dont on fait l'acquisition sur une autre broche du micro-contrôleur (*convertisseur analogique - numérique*),
- **soit une *tension de référence*** générée en interne par le micro-contrôleur à l'aide du module de génération de tension de référence.

FIGURE 20-2: SINGLE COMPARATOR



Ce principe de fonctionnement décrit ci-contre permet typiquement d'effectuer une commande de type tout-ou-rien (TOR).

Illustration par un exemple simple…

Cahier des charges :

On reprend le cahier des charges posé pour faire clignoter un LED branché sur RB0 (*cf.,* illustration du module TIMER p.XX) mais on cherche maintenant à ajuster la fréquence de clignotement en fonction d'une tension présentée sur la broche RA3 du microcontrôleur. La fréquence sera de 300Hz si la tension est de inférieure à $V_{DD}/2$ et de 600 Hz si elle est supérieure à $V_{DD}/2$.

Questions :

(1) On utilise une référence interne de tension pour générer la valeur de $V_{DD}/2$. Expliquez comment le *module de tension de référence* peut être configuré pour cela. Expliquez ensuite comment le module de comparaison peut être utilisé pour réaliser le cahier des charges.

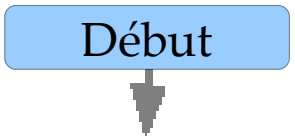
(2) Identifiez les registres à initialiser et les valeurs associées.

(3) Construisez l'algorigramme et écrivez le programme assembleur.

Algorigramme

Programme assembleur


```
; Filename : comparator.asm
;
; Description : Génération d'un signal carré sur la
; broche RB0 par TIMER0 et ajustement de la fréquence en
; fonction d'une comparaison à une tension de référence.
;
; Author:
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2007/09
```



Programme assembleur [suite]

Module de conversion analogique/numérique

Le module de conversion analogique/numérique permet de convertir le signal analogique présent sur une broche du micro-contrôleur en un signal numérique.

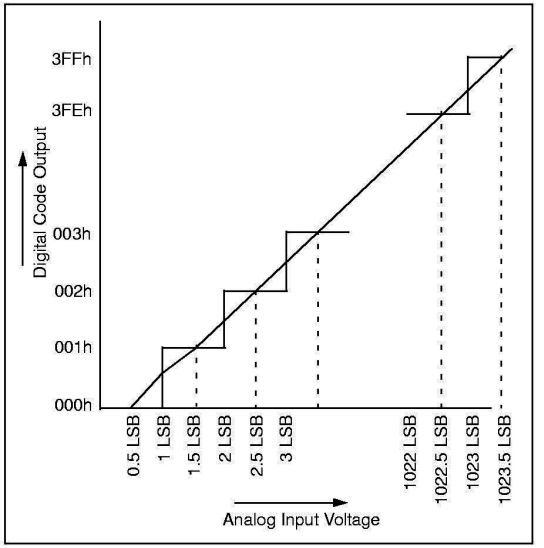
Les paramètres à prendre en compte pour la numérisation d'un signal sont

- **la pleine échelle** du module de conversion A/N
  Indique la plage de tension admissible en entrée du module.

- **la dynamique**
  Indique le nombre de bits utilisés pour coder une valeur analogique en numérique.

- **la fréquence d'échantillonnage**

FIGURE 19-2: A/D TRANSFER FUNCTION



Notes : (1) la pleine échelle et la dynamique permettent de calculer la *résolution en tension* du module de conversion A/N ; (2) la fréquence d'échantillonnage doit respecter le (fameux) « *théorème de Shannon* » ; cf cours de traitement du signal.

Illustration par un exemple simple…

Cahier des charges :

On reprend le cahier des charges posé pour générer un signal rectangulaire avec le module PWM mais on cherche maintenant à ajuster en continu la fréquence de manière à ce qu'elle soit proportionnelle à la tension présentée sur la broche RA0. Pour permettre l'ajustement en continu de la tension, on déclenchera la conversion par interruption du TIMER0 automatiquement tous les 1/100 secondes.

Questions :

(1) Lisez la documentation et expliquez le fonctionnement du CAN sur la base de la figure 19-1. Reprenez la relation entre la valeur du registre PR2 qui gère la fréquence et calculez les valeurs extrêmes de fréquence atteignables. Déduisez de ce qui précède la valeur du *prescaler* pour obtenir des fréquences prises entre 244 Hz et 60 Khz. La relation entre la conversion et la fréquence est-elle linéaire ? Sachant que le CAN est un convertisseur 10 bits et que PR2 est un registre 8 bits, quels sont les bits du CAN que vous allez utiliser ? Quel sera l'inconvénient éventuel ?

Illustration par un exemple simple…

Questions :

(1) Identifiez les registres à initialiser et les valeurs associées.

(2) Construisez l'algorigramme et écrivez le programme assembleur.

Algorigramme

Programme assembleur

A COMPLÉTER !

```
; Filename : CAN.asm
;
; Description : Génération d'un signal carré sur la
; par le module PWM avec ajustement de la fréquence en
; continu par conversion AN sur broche AN0.
;
; Author:
; Company:  Universite Paul Cezanne
; Revision: 1.00
; Date:   2007/09
```

Début

Programme assembleur [suite]

Le « chien de garde » (Watchdog)

Une **watchdog** (WDT) est un dispositif de protection pour éviter que le micro-contrôleur ne se bloque. Une watchdog effectue un redémarrage du système (RESET) si une action définie n'est pas effectuée dans un délai donné.

Concrètement, l'utilisateur affecte une valeur à un registre (*Watchdog Postscaler*), qui définit une durée temporelle (**timeout**). Périodiquement le micro-contrôleur va incrémenter un registre (*Watchdog counter*). Si ce registre est plein (*overflow*), le micro-contrôleur effectue un re-démarrage.

Pour que le micro-contrôleur ne redémarre pas, le programme doit périodiquement ré-initialiser le registre (Watchdog counter).

Les différents modes de fonctionnement

En plus du mode de fonctionnement par défaut (*Primary Run Mode*), les micro-contrôleurs possèdent de nombreux autres modes de fonctionnement. L'existence de ces modes vise principalement à réduire la consommation d'énergie qui est une contrainte forte pour les systèmes embarqués.

On notera principalement trois modes de fonctionnement :

- **run mode ---** mode de fonctionnement par défaut du micro-contrôleur, toutes les fonctions sont activées, *la consommation d'énergie est maximale*.

- **sleep mode ---** le micro-contrôleur est placé en mode sommeil, la consommation d'énergie est minimale, *le micro-contrôleur peut-être réveillé par une interruption*,

- **IDLE mode ---** le processeur du micro-contrôleur est arrêté, plus aucune instruction n'est exécutée, l'utilisateur peut choisir de désactiver des fonctions du micro-contrôleur afin de diminuer la consommation d'énergie. *Les fonctions activées restantes fonctionnent normalement et peuvent réveiller le micro-contrôleur par une interruption*.

Plan

Présentation de l'informatique industrielle et des systèmes micro-programmés

Architecture des micro-contrôleurs

Présentation des différents éléments d'un micro-contrôleur, éléments de choix

Rappels sur les nombres binaires et les différents codages

Les instructions

Rappels sur la logique combinatoire et séquentielle

Étude du fonctionnement d'un micro-contrôleur : le PIC 18F4520

Programmation en Assembleur -- Rappel sur les algorigrammes

Présentation des interruptions

Étude d'un programme en Assembleur avec gestion des interruptions

Présentation de fonctions intégrées (timer, PWM, etc.)

Presentation du langage C pour le µ-contrôleur / spécificité PIC 18F4520

Langage C & microcontrôleur

De plus en plus, les programmes pour micro-contrôleur sont écrits en langage C. Ce langage permet de développer rapidement des programmes, des bibliothèques qui pourront être ensuite utilisées sur différentes machines.

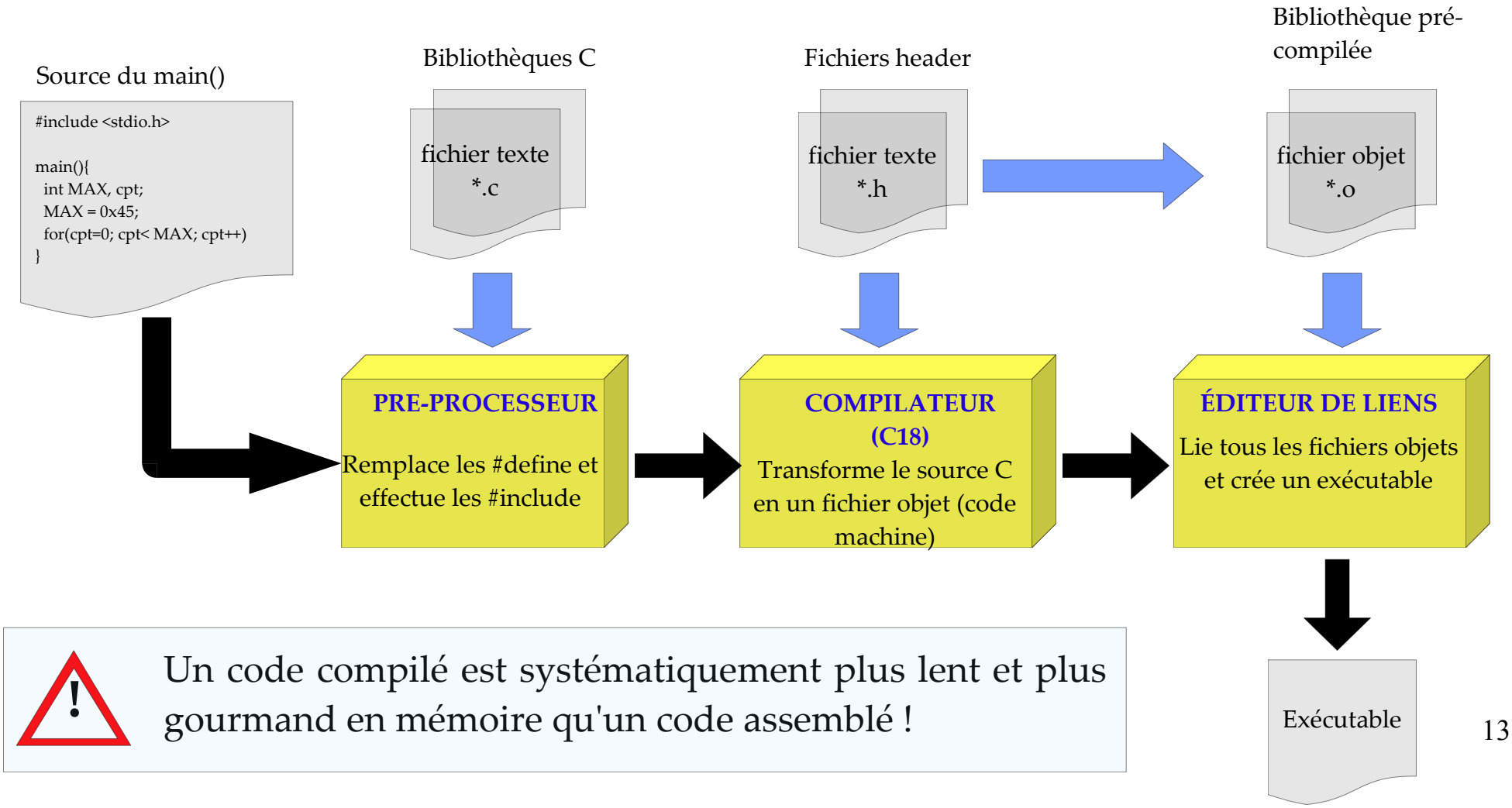
Pourquoi un langage tel que le C ?

- **Universel** : il n'est pas dédié à une application !
- **Moderne** : structuré, déclaratif, récursif, avec structures de contrôle et de déclaration.
- **Proche de la machine** : manipulations de bits, pointeurs, possibilité d'incorporer de l'assembleur, etc.
- **Portable** : le même code peut être utilisé sur plusieurs machines [*il faut toutefois faire attention à ne pas créer des fonctions spécifiques à une machine*].
- **Extensible** : il est possible de créer des bibliothèques ou d'en incorporer.

Documentation Microchip : « *MPLAB C18 C COMPILER USER'S GUIDE (DS51288J)* »

Construction d'un exécutable (1)

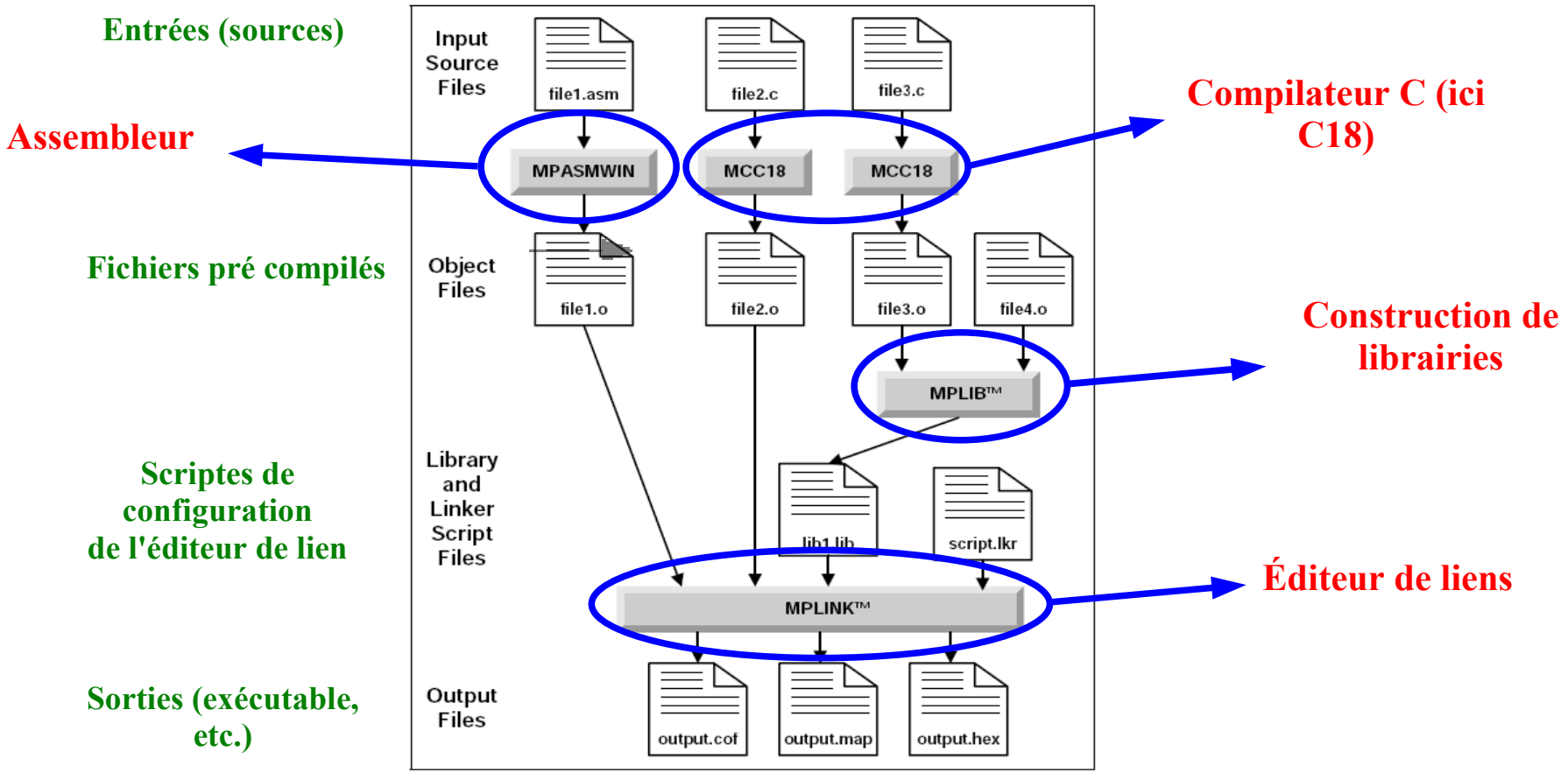
Alors que l'assembleur fait une conversion *directe* de mnémoniques en langage machine, le compilateur C doit construire le code machine à partir d'une syntaxe de plus haut niveau. Le recours à des *librairies pré-compilées* est permis par l'éditeur de lien qui construit un exécutable à partir des différents *fichiers objets*.



Un code compilé est systématiquement plus lent et plus gourmand en mémoire qu'un code assemblé !

Construction d'un exécutable (2)

Assembleur et compilateur peuvent néanmoins être utilisés pour construire un seul exécutable, *cf.* ci-dessous. Là encore, l'éditeur de lien s'occupe de construire le code machine exécutable à partir des différents fichiers objets.



Vous avez accès à la majorité des fonctionnalités du C de la norme ANSI C. Vous pouvez donc écrire quelque chose comme

```
vitesse = 0x27
```

pour faire *l'affectation d'une variable* codée sur 8 bits, ou encore

```
if(vitesse == limite)
```

pour effectuer un *test d'égalité* entre deux variables *entières* codées sur un nombre de bits appropriés.

Plus précisément, vous avez accès (*cf.* cours de C++)

- aux déclaration de *variables, constantes, tableaux, pointeurs, structures, etc.*
- aux opérateurs *arithmétiques* et *logiques*,
- aux opérateurs d'*affectation, incrémentation* et *décrémentation*,
- aux opérateurs de contrôle de flux (*test* et *boucles*),
- à la *conversion de type*,
- à l'*intégration de routines en assembleur*.

Par contre, les instructions gérant les *entrées* et *sorties* (écran, clavier, disque, *etc.*) n'ont pas toujours de sens dans ce type de contexte ; cf. TP.

Un premier programme en C

```
//================================
// Filename: PremierProgramme.C
//================================
// Author:    marc ALLAIN
// Company:   Universite Paul Cezanne
// Revision: 1.00
// Date:    2006/07
//================================

#include <p18f4520.h>
#define duree 10000
#pragma config WDT = OFF

// Déclaration des variables globales
char c;
float pht;

// Prototypes des fonctions
void tempo(unsigned int count);

// Programme Principal
void main()
{
      PORTB = 0x00;
      TRISB = 0x00;
      while(1){
        PORTB++;
        tempo(duree);
      }
}

tempo(unsigned int count){
      while(count--);
}
```

Fichier des en-têtes de fonction pré-compilées spécifiques au microcontrôleur considéré. Contient notamment les équivalences nom & adresses.

Déclaration d'un alias au préprocesseur (i.e., duree sera systématiquement remplacé par 10000).

Directive pour adresser des emplacement spécifiques de la mémoire programme ou données (cf. plus loin).

Déclaration de variables globales

Déclaration d'une fonction

Affectation de registres du microcontrôleur ; PORTB et TRISB sont déclarés dans p18f4520.h

Les opérateurs du langage C

Opérateurs *arithmétiques* et *relationnels* :

Opérateur	Traduction	Exemple	Résultat
`+`	Addition	`x + y`	l'addition de x et y
`-`	Soustraction	`x - y`	la soustraction de x et y
`*`	Produit	`x * y`	la multiplication de x et y
`/`	Division	`x / y`	le quotient de x et y
`%`	Reste	`x % y`	Reste de la division euclidienne de x par y
`+`(unaire)	Signe positif	`+x`	la valeur de x
`-`(unaire)	Signe négatif	`-x`	la négation arithmétique de x
`++`(unaire)	Incrément	`x++` ou `++x`	x est incrementé ($x = x + 1$). L'opérateur préfixe `++x` (resp. suffixe `x++`) incrémente x avant (resp. après) de l'évaluer
`--`(unaire)	Decrément	`x--` ou `--x`	x est décrementé ($x = x - 1$). L'opérateur préfixe `--x` (resp. suffixe `x--`) décrémente x avant (resp. après) de l'évaluer

Opérateur	Traduction	Exemple	Résultat
`<`	inférieur	`x < y`	1 si x est inférieur à y
`<=`	inférieur ou égal	`x <= y`	1 si x est inférieur ou égal à y
`>`	supérieur	`x > y`	1 si x est supérieur à y
`>=`	supérieur ou égal	`x >= y`	1 si x est supérieur ou égal à y
`==`	égalité	`x == y`	1 si x est égal à y
`!=`	non inégalité	`x != y`	1 si x est différent de y

Opérateurs *logiques* et *de manipulation de bits* :

Opérateur	Traduction	Exemple	Résultat (pour chaque position de bit)
`&`	ET bit à bit	`x & y`	1 si les bits de x et y valent 1
`|`	OU bit à bit	`x | y`	1 si le bit de x et/ou de y vaut 1
`^`	XOR bit à bit	`x ^ y`	1 si le bit de x ou de y vaut 1
`~`	NON bit à bit	`~x`	1 si le bit de x est 0
`<<`	décalage à gauche	`x << y`	décale chaque bit de x de y positions vers la gauche
`>>`	sécalage à droite	`x >> y`	décale chaque bit de x de y positions vers la droite

Opérateur	Traduction	Exemple	Résultat
`&&`	ET logique	`x && y`	1 si x et y sont différents de 0
`||`	OU logique	`x || y`	1 si x et/ou y sont différents de 0
`!`	NON logique	`!x`	1 si x est égal à 0. Dans tous les autres cas, 0 est renvoyé.

Opérateurs *d'accès à la mémoire, etc.*

Opérateur	Traduction	Exemple	Résultat
`&`	Adresse de	`&x`	l'adresse mémoire de x
`*`	Indirection	`*p`	l'objet (ou la fonction) pointée par p
`[ ]`	Elément de tableau	`t[i]`	L'équivalent de `*(x+i)`, l'élément d'indice i dans le tableau t
`.`	Membre d'une structure ou d'une union	`s.x`	le membre x dans la structure ou l'union s
`->`	Membre d'une structure ou d'une union	`p->x`	le membre x dans la structure ou l'union pointée par p

Opérateur	Traduction	Exemple	Résultat
`()`	Appel de fonction	`f(x,y)`	Exécute la fonction f avec les arguments x et y
`(type)`	cast	`(long)x`	la valeur de x avec le type spécifié
`sizeof`	taille en bits	`sizeof(x)`	nombre de bits occupé par x
`? :`	Evaluation conditionnelle	`x?:y:z`	si x est différent de 0, alors y sinon z
`,`	séquencement	`x,y`	Evalue x puis y

Types de données

Le tableau ci-dessous présente les types de variables supportés par le compilateur C18 ainsi que leur format de codage.

Type	Size	Minimum	Maximum
char(1,2)	8 bits	-128	127
signed char	8 bits	-128	127
unsigned char	8 bits	0	255
int	16 bits	-32,768	32,767
unsigned int	16 bits	0	65,535
short	16 bits	-32,768	32,767
unsigned short	16 bits	0	65,535
short long	24 bits	-8,388,608	8,388,607
unsigned short long	24 bits	0	16,777,215
long	32 bits	-2,147,483,648	2,147,483,647
unsigned long	32 bits	0	4,294,967,295

Le format est celui du *bus de donnée* pour les types char et unsigned char. L'utilisation de variables « plus grandes » est néanmoins permise. Par exemple, une déclaration de la forme

```
#pragma idata test=0x0200
long l=0xAABBCCDD;
```

conduit au stockage mémoire suivant

Address	*0x0200*	*0x0201*	*0x0202*	*0x0203*
Content	0xDD	0xCC	0xBB	0xAA

Structures & champs de bits

Les structures sont des types composites dans lesquels des variables de types distincts peuvent cohabiter...

```
struct prof {
        char nom[30];
        char prenom[30];
        char labo[30];
        int tel;
        int HETD;
};
```

```
int main(){
        struct prof DEP_SDM[10];

        DEP_SDM[0].tel = 2878;
        strcpy(DEP_SDM[0].nom,"allain");
        strcpy(DEP_SDM[0].prenom,"marc");
        ...
}
```

Les structures « champs de bits » permettent d'accéder explicitement à des sous-ensembles d'une variable : le premier champ correspond au bit 0 et le nom de l'élément est suivi par le nombre de bits associé.

```
struct {
  unsigned RB0:1;
  unsigned RB1:1;
  unsigned RB2:1;
  unsigned RB3:1;
  unsigned GROUPE:3;
  unsigned RA7:1;
}PORTBbits;
```

Exemple : on accède à la patte RB0 par une instruction PORTAbits.RA0 = 1;

Unions

Dans une union, les champs partagent la même adresse. Cette construction permet d'utiliser des dénominations différentes pour adresser les même bits. *cf.* le fichier de header.

```
extern volatile near union {
 struct {
       unsigned RE0:1;    unsigned RE1:1;
       unsigned RE2:1;    unsigned RE3:1;
 };
 struct {
       unsigned RD:1;    unsigned WR:1;
       unsigned CS:1;    unsigned MCLR:1;
 };
 struct {
       unsigned NOT_RD:1;    unsigned NOT_WR:1;
       unsigned NOT_CS:1;    unsigned NOT_MCLR:1;
 };
 struct {
  unsigned :3;    unsigned VPP:1;
 };
 struct {
  unsigned AN5:1;    unsigned AN6:1;    unsigned AN7:1;
 };
} PORTEbits;
```

Exemple : PORTEbits.RE0 et PORTE.NOT_RD partage la même adresse.

De l'assembleur dans du C ?

C18 permet l'utilisation d'instructions en assembleur presque comme si vous utilisiez MPASM. Ces instructions doivent être dans un « bloc » délimité par _asm et _endasm.

```
_asm                                    /* User assembly code */
        MOVLW 10                        // Move decimal 10 to count
        MOVWF count, 0
        start:                  /* Loop until count is 0 */
        DECFSZ count, 1, 0
        GOTO done
        BRA start
        done:
_endasm
```

Cet assembleur diffère néanmoins de MPASM sur les poinst suivant :

- pas les directives,
- commentaires rédigés dans la syntaxe du C,
- pas de valeur par défaut, *i.e.*, les arguments d'une instruction doivent être spécifiés,
- la convention de notation en hexadecimal est 0xNN,
- les étiquettes doivent comporter un « : »,
- pas d'adressage indexé.

La conversion de type (cast)

Une conversion de type intervient lorsqu'un opérateur doit agir sur des opérandes de types différents. En général, les opérandes sont alors converties selon la règle suivante :

L'opérande la plus petite est convertit dans le type de l'opérande la plus grande.

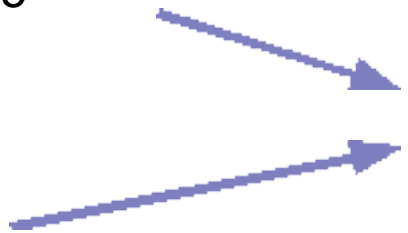
char < int < long < float < double

Il est aussi possible de forcer une conversion de type :

```
float ecart, distance    = 11;
int tronque, nbp = 5;

ecart          = distance / (float) (nbp – 1);
tronque  = (int) ecart;
```

ecart=2.25

ecart=2



Modificateurs de type

Associé à chacun des types, le compilateur C18 supporte les *modificateurs de type* classiques qui sont

const : *une constante stockée en ROM qui ne pourra pas être modifiée.*
ex. const int MAX = 10000;

volatile : *une variable stockée en RAM ou en pile qui peut être modifiée* (défaut).
ex. unsigned char PORTB = 0xF0;

En plus de ces modificateurs, C18 en introduit deux autres pour contrôler les accès aux différentes formes de mémoire...

ram : *une donnée placée dans la mémoire donnée RAM* (défaut)
ex. unsigned char PORTB = 0xF0;

rom : *une donnée stockée dans la mémoire programme ROM*
ex. rom const int MAX = 10000;

Variables globales, locales...

La programmation en langage C se base entièrement sur des fonctions dont la principale est main(). La déclaration de variable peut être faite à l'*intérieur* ou à l'*extérieur* d'une fonction avec des effets distincts (notion de *classes des stockage*).

A l'extérieur d'une fonction, une déclaration est **globale** : la variable est visible pour toutes les fonctions et l'adresse allouée est fixe.

ex.
```
#include <pic18f4520.h>
int MAX;
main(){...}
```

A l'intérieur d'une fonction, la variable n'est pas visible de l'extérieur et les mots clés suivant détermine la manière dont la variable est stockée :

`static` : *variable stockée en RAM à une adresse fixe.*

ex.
```
void function(){
static int MAX;...}
```

auto : *une variable stockée en pile (défaut).*

ex. void function(){
int MAX;...}

register : *la variable est allouée dans un registre de travail.*

Note : *(i) correspondance obligatoire entre format de la variable et format du registre ; (ii) inutile pour le PIC qui ne possède qu'un registre de travail.*

En plus de ces classes de stockage standard, C18 en introduit la classe suivante pour réduire l'occupation mémoire des variables allouées lors de l'écriture de fonction :

overlay : *l'adresse est statiquement allouée dans la mémoire donnée RAM mais elle peut être affectée à une autre variable déclarée dans une autre fonction.*

ex. void function1(){
overlay int x = 5; return x;}

void function2(){
overlay int y = 2; return y;}

Les directives « pragma »

En dépit d'une grande souplesse et d'une bonne proximité avec la couche matérielle, la norme ANSI C ne permet pas de contrôler les adresses mémoires où sont placées des variables ou des instructions. Pour pallier ce problème, C18 introduit les directives #pragma qui permettent d'accéder spécifiquement à des adresses de la mémoire donnée et programme ainsi que de configurer le microcontrôleur. Ainsi la directive

```
#pragma config WDT = OFF
```

permet de désactiver la fonctionnalité « watch-dog » du microcontrôleur ; la liste des fonctionnalités et de leur valeurs est disponible dans la documentation DS51537.

Cette directive permet de placer un morceau de code à une adresse quelconque de la mémoire programme. Par exemple,

```
#pragma code  mon_prog = 0x@ // après : placement imposé dans la mémoire

void mon_prog(void){
   instructions;
}

#pragma code                          // après : placement libre dans la mémoire
```


permet de placer `instructions;` à l'adresse `0x@` de la mémoire programme.

Les directives « pragma » (2)

Ceci permet en particulier de configurer les d'interruptions : une première directive permet d'*initialiser le vecteur d'interruption* (haute ou basse) et alors qu'une seconde directive identifie l'adresse mémoire de cette fonction :

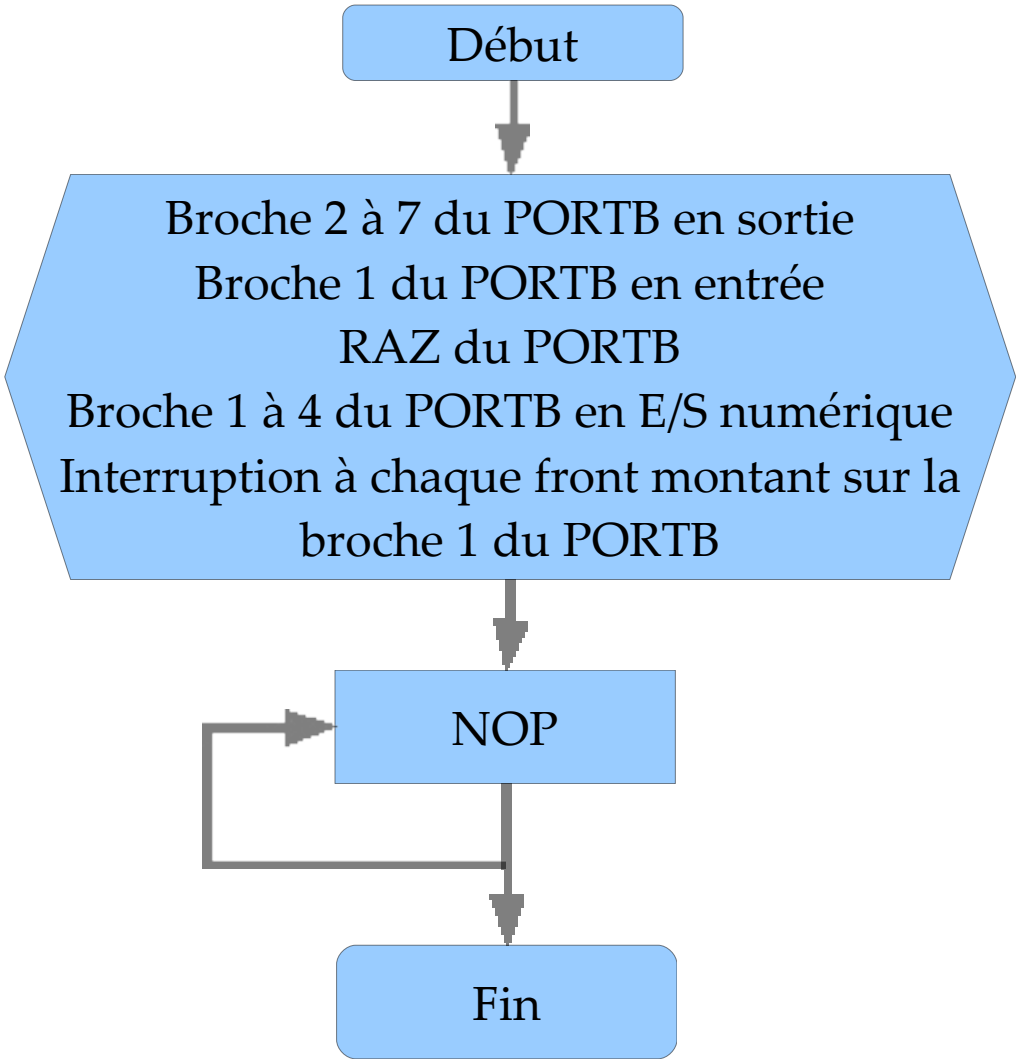
```
#pragma code vecteur_low = 0x18

void vecteur_low(){
        _asm goto routine_int_low _endasm
};

#pragma code


#pragma interruptlow routine_int_low
void routine_int_low(){
...};
```

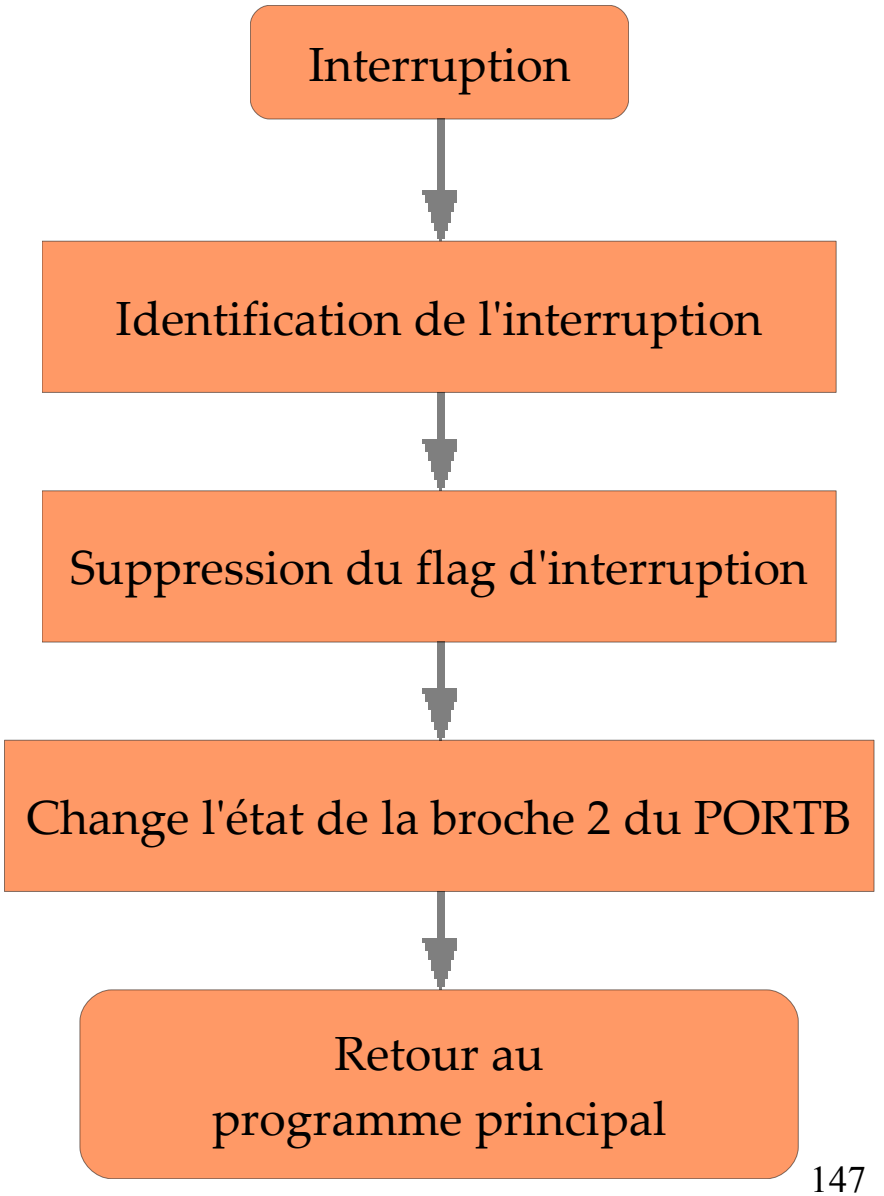
Quelques remarques : (**1**) une fonction d'interruption ne possède ni entrée ni sortie ; (**2**) une sauvegarde minimale du contexte est assurée automatiquement et peut être complétée en modifiant la seconde directive comme suit

```
#pragma interruptlow routine_int_low save = ma_variable
```

où ma_variable est une variable **globale** à sauver.

Gestion des interruptions (1)





Gestion des interruptions (2)

```
//=======================================
// Filename: premier_programme_C.C
//=======================================
// Author:   Marc Allain
// Company:  Universite Paul Cezanne
// Revision: 1.00
// Date:   2007/10
//=======================================

#include <p18f4520.h>
#pragma config WDT = OFF

// Prototypes des fonctions
void Vecteur_interruption(void);
void Routine_gestion_interruption(void);

// Programme Principal
void main()
{
     PORTB = 0;
     TRISB = 0x01;
     LATB = 0;
     ADCON1 = 0x0F;
     INTCON = 0x90;
     while(1){
     }
}
```

Gestion des interruptions (3)

```
// Indique l'adresse mémoire ou débute la fonction

#pragma code Vecteur_interruption = 0x08
void Vecteur_interruption (void)
{
  _asm
    goto Routine_gestion_interruption
  _endasm
}
#pragma code // Placement libre du programme dans la mémoire

// Routine de gestion des interruptions

#pragma interrupt Routine_gestion_interruption
void Routine_gestion_interruption()
{
     // Vérification que l'interruption est déclenchée par RB0
     if (INTCONbits.INT0IF)
     {
     // Suppression du flag d'interruption
     INTCONbits.INT0IF = 0;
     // Change l'état de RB1
     PORTB ^= 0x02;
     }
}
```

Quelques astuces/pièges en langage C

- **Utiliser le passage des arguments par adresse**

La fonction ne travaille pas sur une copie de la valeur de l'argument effectif mais directement sur celui-ci (gain de place en mémoire).

- **Les fonctions récursives**

Elles sont à éviter pour la programmation des systèmes embarqués à cause d'un débordement de pile matériel. En effet, on rappelle que sur un micro-contrôleur, le nombre d'appels de fonctions imbriqués est limité (8 pour PIC).