

Architecture des microcontrôleurs

l'unité de contrôle

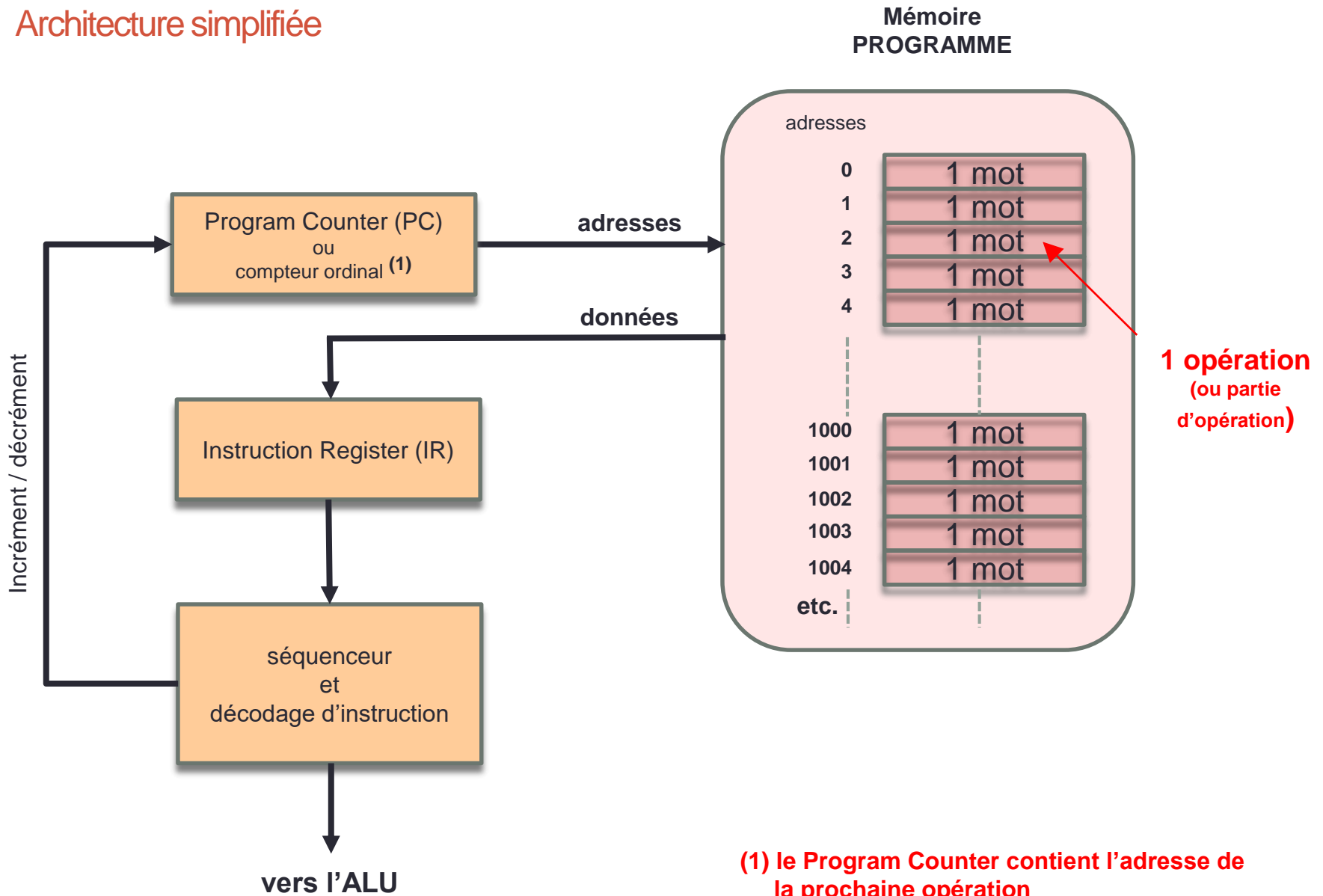
Sommaire

1. L'unité de contrôle
2. Les ressources et le jeu d'instructions
3. Les modes d'adressage
4. Les routines
5. Les interruptions

1. L'unité de contrôle

- L'unité de contrôle permet de séquencer correctement les opérations contenues dans un programme.
- Elle permet de dérouler les opérations dans le bon ordre, selon ce qui est écrit dans le programme.
- Elle permet de présenter à l'ALU les informations nécessaires à l'exécution d'une opération :
 - la nature de l'opération à réaliser (**décodage d'instruction**),
 - l'endroit où se trouvent les opérandes (entrées) de l'opération (**décodage d'adresse**),
 - l'endroit où placer le résultat (sortie) de l'opération.

Architecture simplifiée



Cas du PIC18

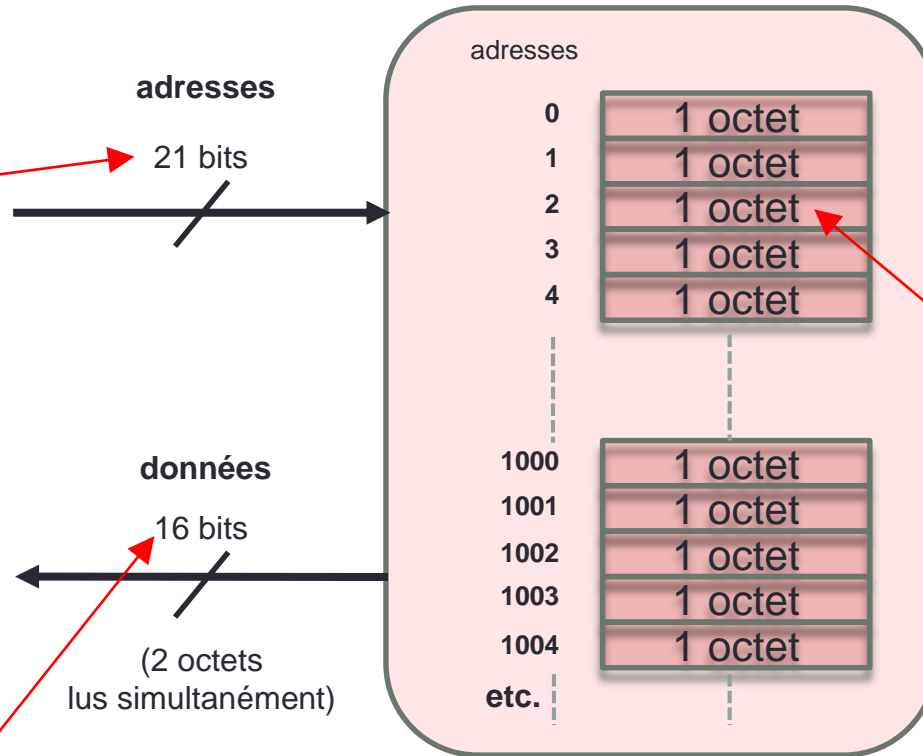
le Program Counter
contient une adresse sur
21 bits
⇒
2²¹ adresses au maximum
(2 Megabytes max)

dans la pratique, la taille
physiquement disponible est
plus petite
(ex: 32 kBytes pour le
PIC18F45K20)

1 opération est codée sur
2 octets (dans la plupart
des cas)
⇒
l'unité de contrôle
récupère simultanément 2
« cases mémoire »

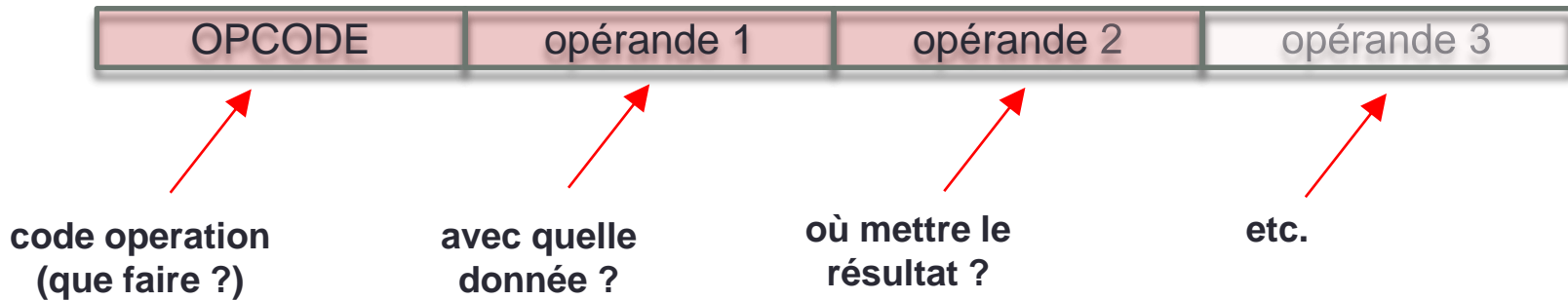
- l'octet de poids faible d'une opération est toujours à une adresse paire
- le Program Counter s'incrémente de 2 entre 2 opérations successives

Mémoire PROGRAMME (Flash)



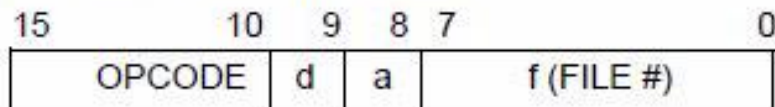
1 octet
=
1 1/2 opération

Structure typique d'une opération



Exemple d'opération pour un PIC18: la plupart des opérations sont codées sur 2 octets.

Byte-oriented file register operations



d = 0 for result destination to be WREG register
d = 1 for result destination to be file register (f)
a = 0 to force Access Bank
a = 1 for BSR to select bank
f = 8-bit file register address

Example Instruction

ADDWF MYREG, W, B

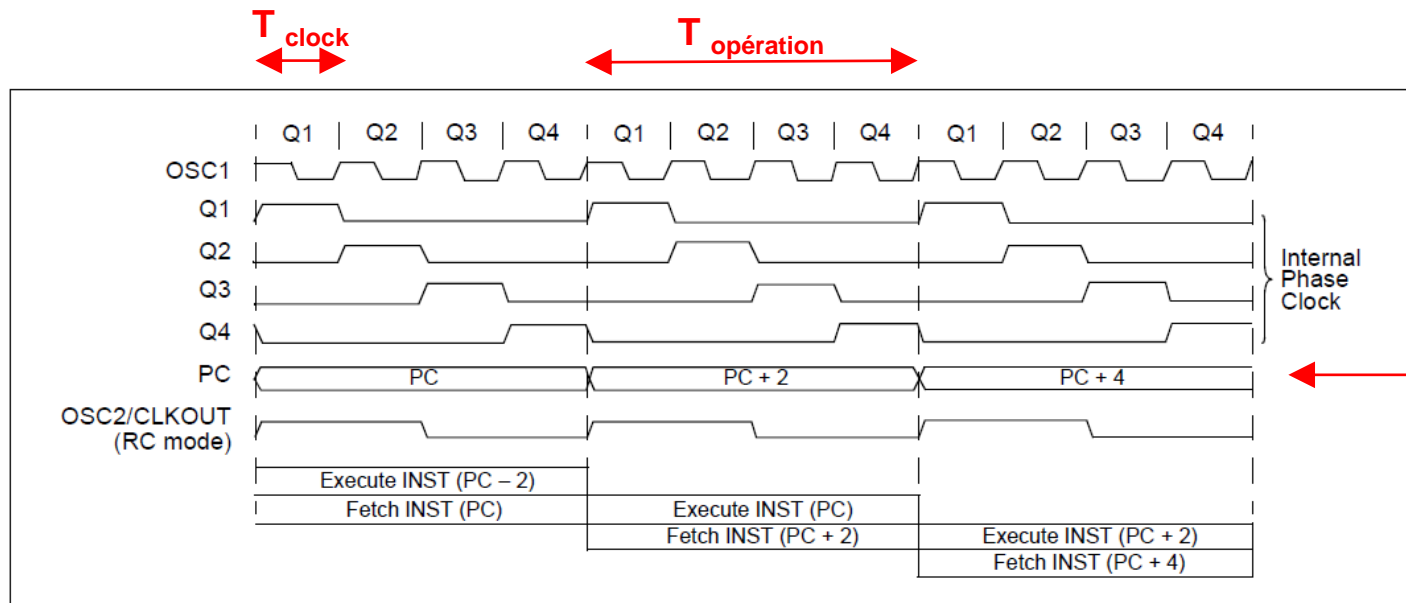
cf.datasheet PIC18F45K20 page 317

Déroulement d'une opération

Une opération se déroule typiquement en 4 cycles machine:

1. décodage de l'opération (c'est-à-dire du mot qui a été placé dans le registre d'instruction (IR))
2. lecture en mémoire de la donnée à traiter (le cas échéant)
3. exécution de l'opération
4. écriture dans la mémoire du résultat de l'opération

$$\Rightarrow \text{temps d'exécution d'une opération élémentaire} = \frac{4}{f_{\text{clock}}}$$

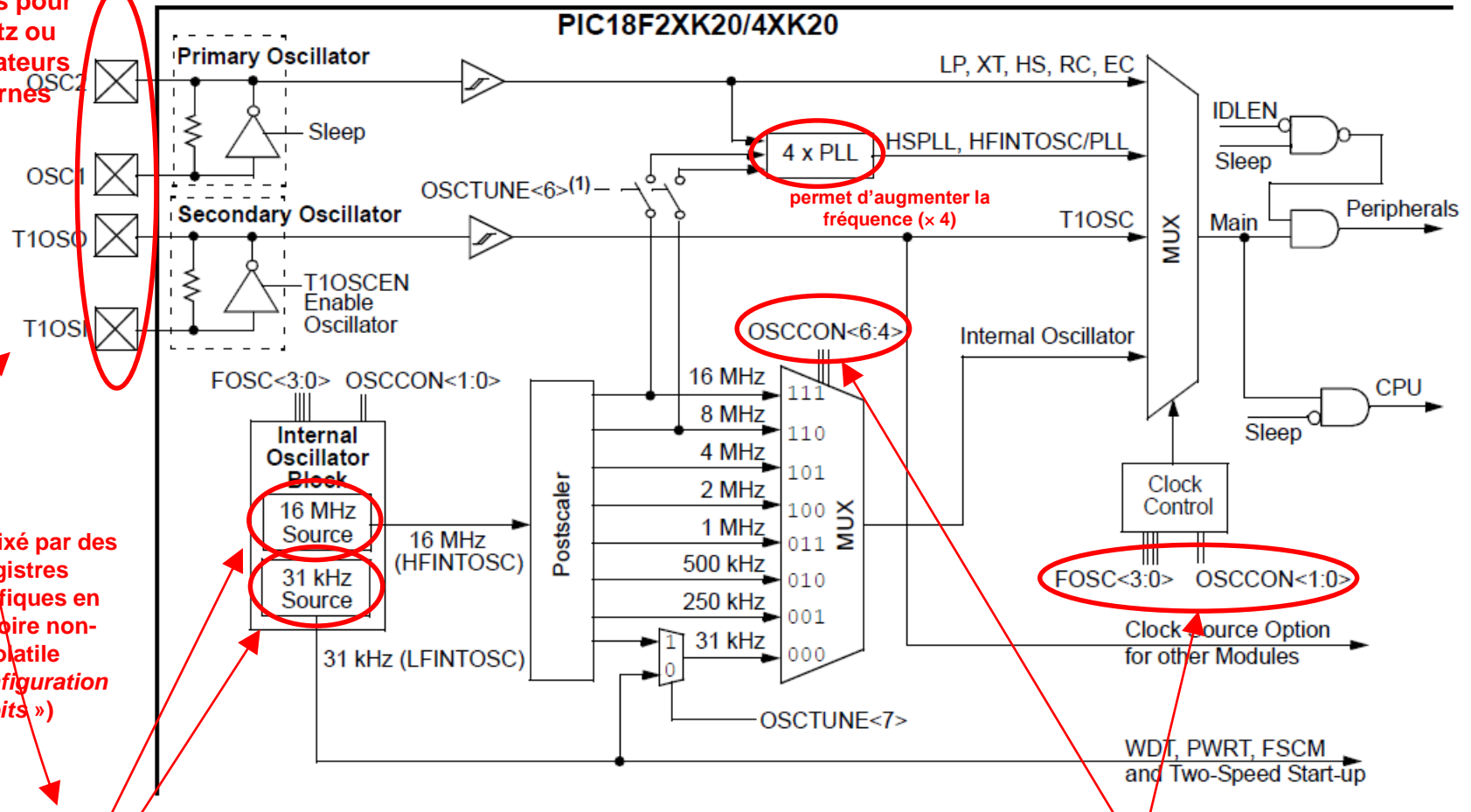


rappel:
le Program Counter
contient l'adresse de
la **prochaine**
opération

cf. datasheet PIC18F45K20 page 69

Exemple du PIC18F45K20: circuit d'horloge

branchements prévus pour quartz ou oscillateurs externes

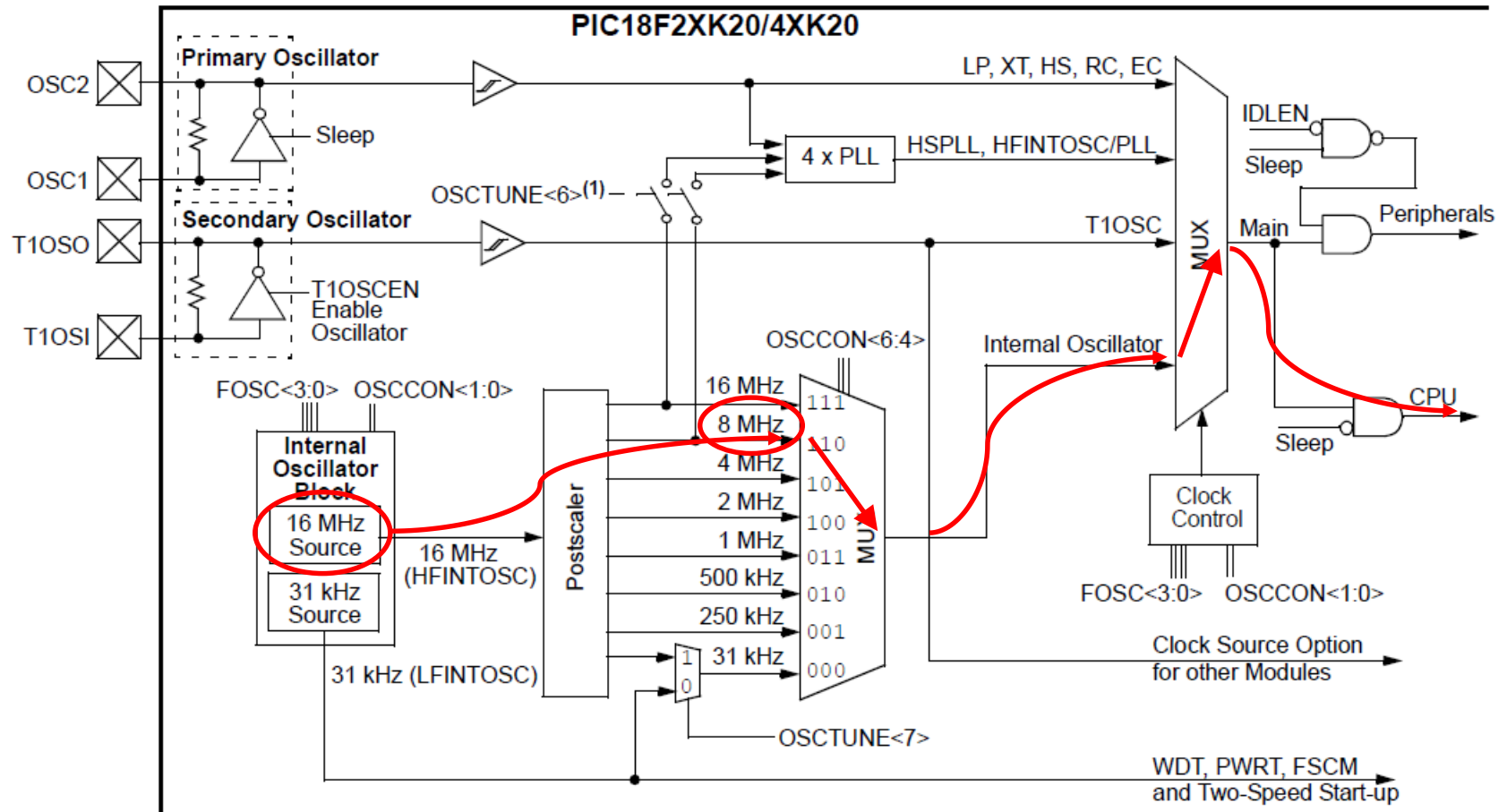


choix fixé par des registres spécifiques en mémoire non-volatile (« configuration bits »)

oscillateurs internes

configuration fixée par des registres en RAM dédiés

Pour faire fonctionner la CPU à 8 MHz, sans oscillateur ou quartz externe:



2. Les ressources et le jeu d'instructions

Ressources de base nécessaires pour constituer un ensemble opérationnel: (exemple du PIC18F45K20)

- Mémoire programme: non volatile (interne pour μ C, externe pour μ P) (32 kBytes)
- Mémoire données: volatile / RAM (interne pour μ C, externe pour μ P) (1536 Bytes)
- Optionnellement, mémoire données non volatile: EEPROM (Electrically Erasable Programmable Read-Only Memory) (256 Bytes)
- ALU + jeu d'instructions + modes d'adressage (8 bits - \approx 70 instructions - littéral / direct / indirect)
- Accumulateur(s) (1 accumulateur)
- Registre d'état (C - DC - Z - OV - N)
- Compteur ordinal (PC) (valeur sur 21 bits)
- Pile (stack) et pointeur de pile (stack pointer) (stack: valeur sur 21 bits - stack pointer: valeur sur 5 bits)
- Périphériques (GPIOs, timers, ADC, etc.) + registres de configurations

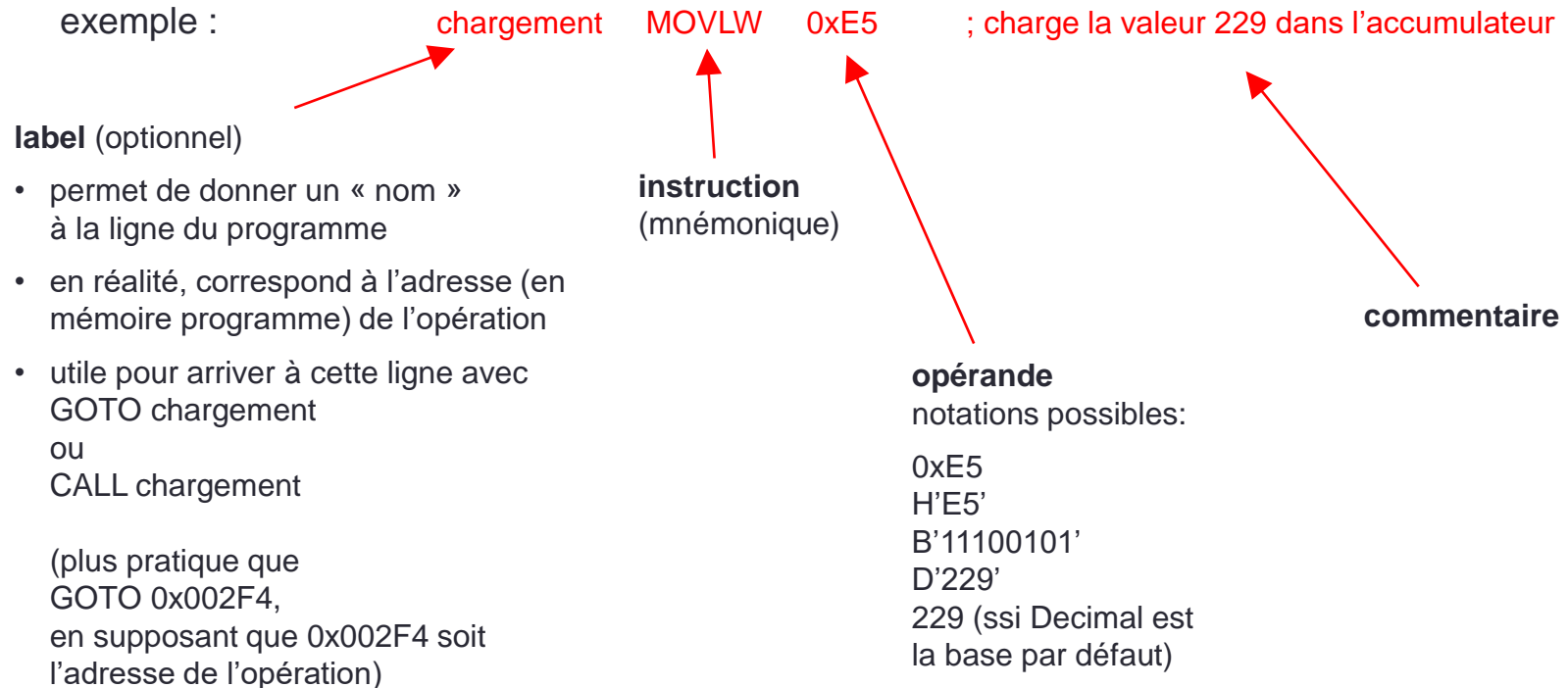
Jeu d'instructions sur un PIC18F:

≈ 70 instructions réparties en 4 groupes principaux

- **Byte-oriented** instructions: pour manipuler (lire / écrire) un octet
ex: ADDWF, MOVWF, INCF, ...
- **Bit-oriented** instructions : pour manipuler (tester / modifier) un bit d'un octet
ex: BSF, BTFSC, ...
- **Literal** instructions : lorsque l'instruction donne directement la valeur d'un octet
ex: MOVLW, ADDLW, ...
- **Control** instructions : par exemple pour gérer le déroulement d'un programme
ex: CALL, GOTO, NOP, ...

Les instructions constituent le « dictionnaire de base » permettant d'écrire un programme.


exemple :



Au début du programme, les variables doivent être déclarées (nom + longueur).

Exemple pour déclarer des variables, stockées en RAM

directive
indiquant que
les variables
qui suivent
sont en RAM

 **UDATA** ; ou **UDATA_ACS**


var1 RES 1 ; variable sur 1 octet
var2 RES 2 ; variable sur 2 octets

A la compilation, des emplacements en RAM (registres) seront attribués aux variables.
(Ce n'est pas la personne écrivant le programme qui choisit les adresses !)

- ☞ À ne pas confondre avec les symboles.
(Juste un mot remplacé par un autre pour plus de lisibilité du programme.
Cela ne « consomme » aucune case mémoire.)

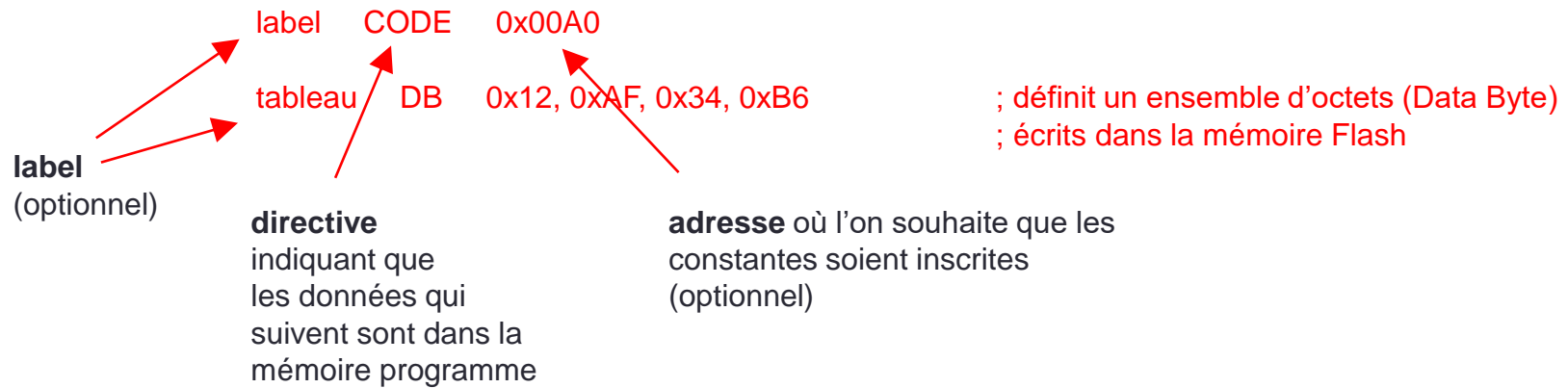
Exemple:

directive
indiquant que
l'on déclare un
symbole

 **tempo EQU 0xF0**
...

MOVLW tempo ; charge la valeur 0xF0 dans l'accumulateur

Exemple pour déclarer des constantes, stockées dans la mémoire Flash



3. Les modes d'adressage

Les modes d'adressage permettent d'indiquer où se trouve l'opérande utilisé par une opération.

Par exemple:

Additionner W et A



où se trouve A ?

- Adressage **inhérent** : l'opération ne nécessite pas d'opérande
- Adressage **littéral** (ou **immédiat**) : l'opération donne la valeur de l'opérande
- Adressage **direct** : l'opération donne l'adresse de l'opérande
- Adressage **indirect** : l'opération indique où se trouve l'adresse de l'opérande

3.1 Adressage inhérent

L'opération ne nécessite pas d'opérande.

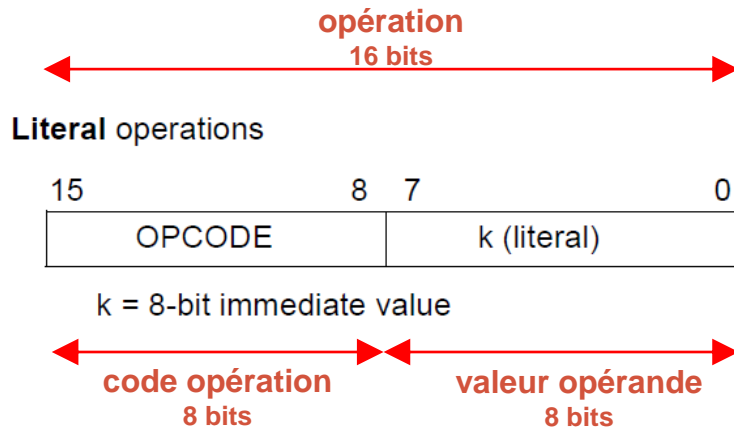
remarque: ici, le terme « adressage » est un abus de langage (on ne fournit pas d'adresse)

Par exemple:

NOP	; no operation
RESET	; software device reset
SLEEP	; go into standby mode

3.2 Adressage littéral (ou immédiat)

L'opération contient la valeur de l'opérande. (« adressage » = abus de langage)



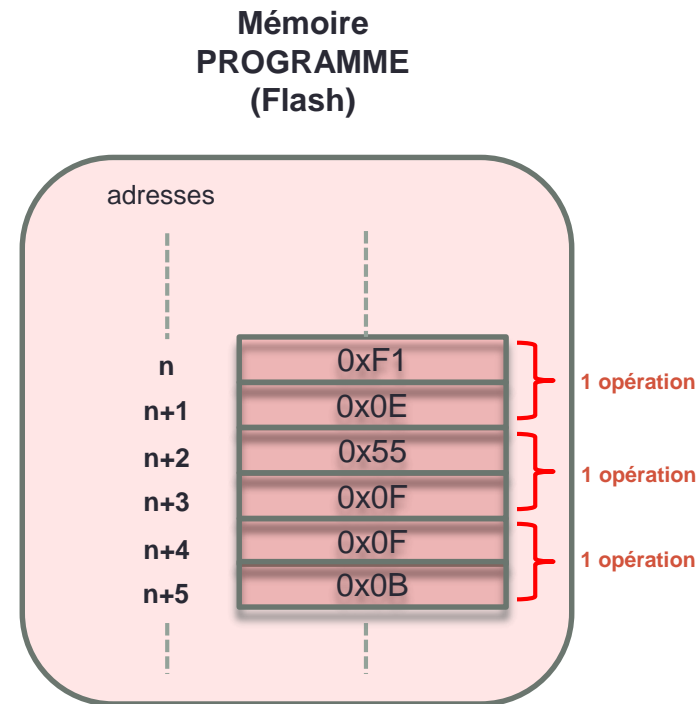
Par exemple:

MOVLW 0xF1 ; 0xF1 → W

ADDLW 0x55 ; W + 0x55 → W

ANDLW 0x0F ; W AND 0x0F → W

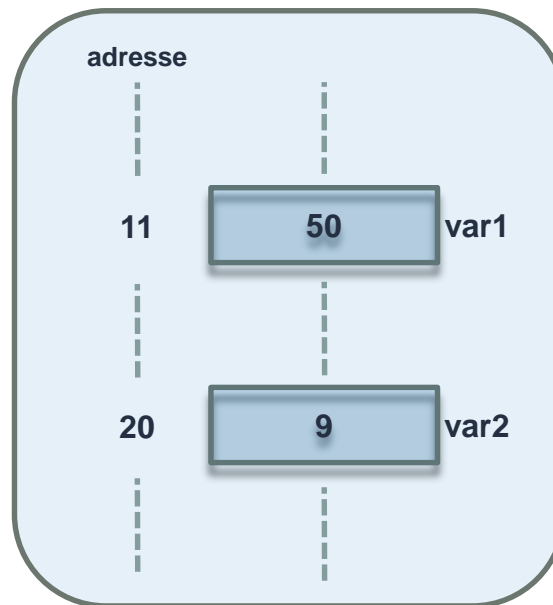
compilation



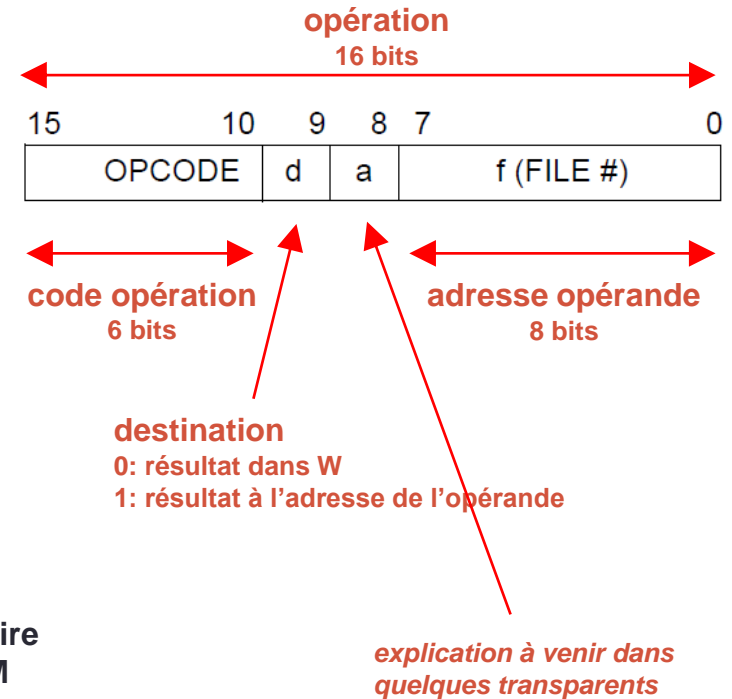
3.3 Adressage direct

L'opération contient l'adresse de l'opérande.

Par exemple:



Mémoire RAM



MOVF var1,0 ; valeur de var1 → W

MOVWF var2 ; W → var2
; var2 contient maintenant la même valeur que W

compilation

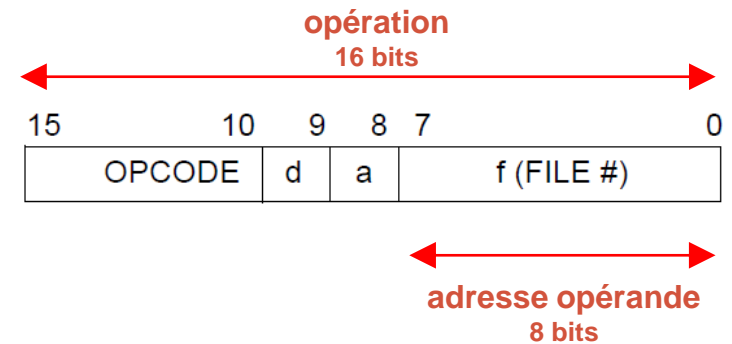


MOVF 11,0

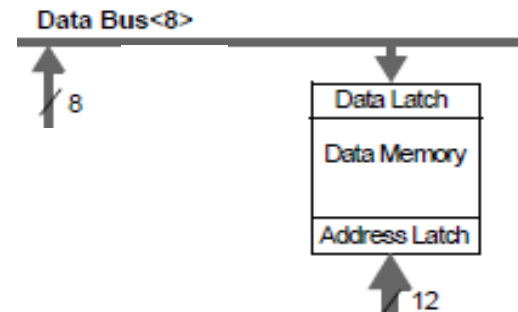
MOVWF 20

Adressage direct: particularité du PIC18

L'opération fournit l'adresse de l'opérande sur **8 bits**.



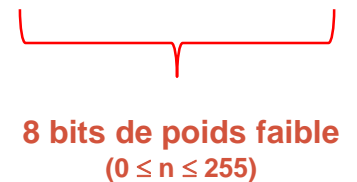
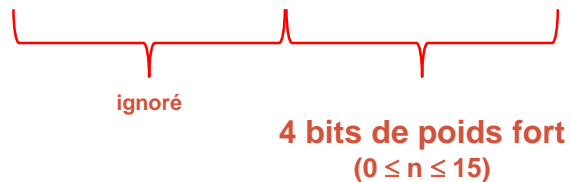
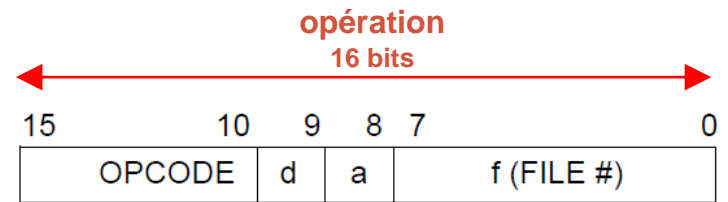
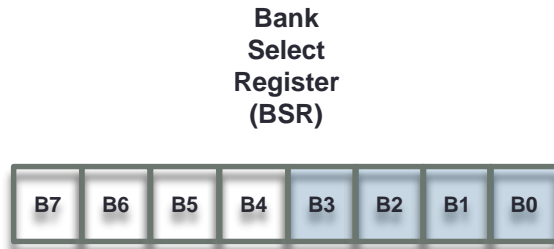
Mais l'adresse complète est sur **12 bits**.



⇒ Il manque 4 bits pour constituer l'adresse complète

Bank Select Register:

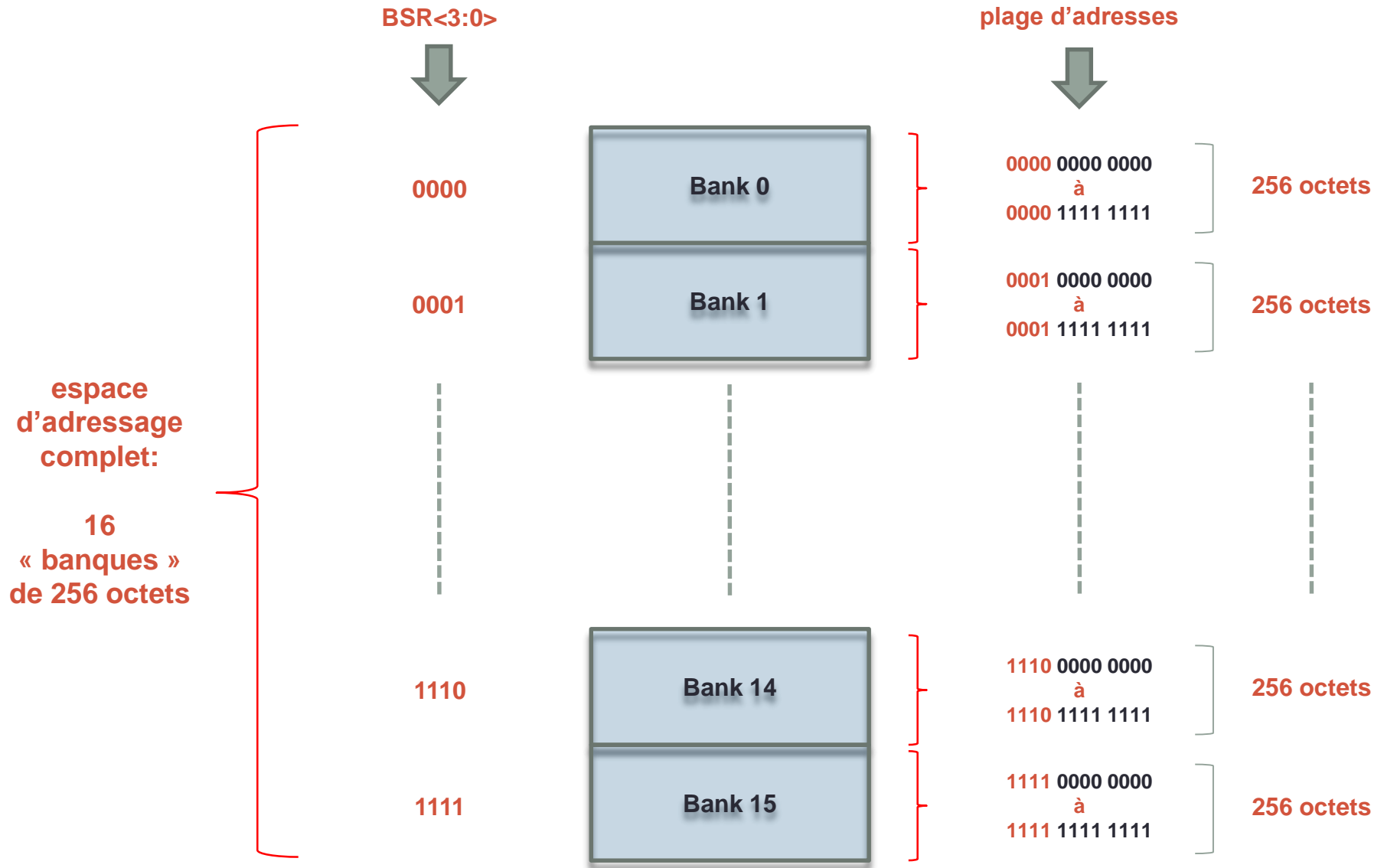
registre (en RAM) dédié contenant les 4 bits de poids fort de l'adresse



adresse complète
sur 12 bits

(16 x 256 possibilités: 4 kBytes adressables)

RAM Memory Map:





Le registre BSR est en registre en RAM « comme les autres ».

⇒ son contenu doit être mis à jour par les opérations du programme.



« Je n'ai pas besoin de plus de 256 octets pour les variables de mon programme.

⇒ Si toutes mes variables sont dans la **banque 0**, je n'ai pas besoin d'utiliser les autres banques.

⇒ il suffit d'initialiser le registre BSR à 0x00 une fois pour toutes et je suis tranquille pour la suite. »



Hélas, les registres permettant de configurer le microcontrôleur (fréquence d'horloge, ports d'entrées / sorties, périphériques, etc.) se trouvent presque tous dans la **banque 15** !

- ⇒ Il faut obligatoirement travailler avec la banque 0 et la banque 15, voire également les autres banques si besoin.
- ⇒ BSR doit être mis à jour dès que nécessaire, par les lignes de code du programme.

Heureusement, la situation n'est pas si grave:

Pour accéder à des variables situées dans la 1^{ère} « **moitié** » de la **banque 0** ou la 2^{ème} « **moitié** » de la **banque 15**, le PIC18 n'a pas besoin d'utiliser l'information contenue dans le registre BSR.

Le microcontrôleur est capable dans ce cas de reconstruire l'adresse complète (sur 12 bits), à partir des 8 bits de poids faible fournis avec l'instruction .

Il va compléter de lui-même les 4 bits de poids fort.

4 bits de poids fort
de l'adresse



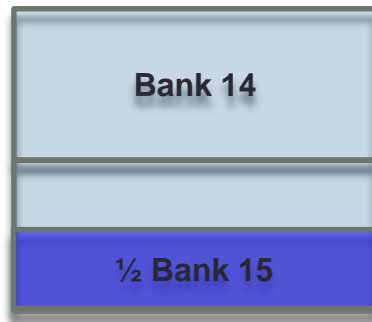
0000

0001



1110

1111

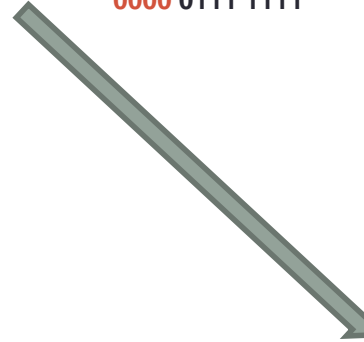


plage d'adresses

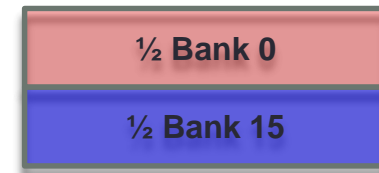


0000 0000 0000
à
0000 0111 1111

128 octets
(0 à 127)



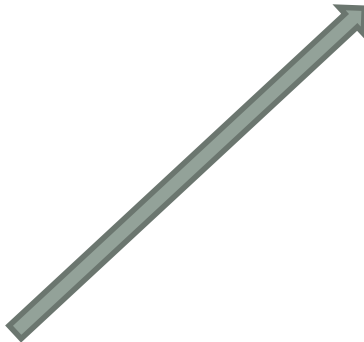
Access Bank

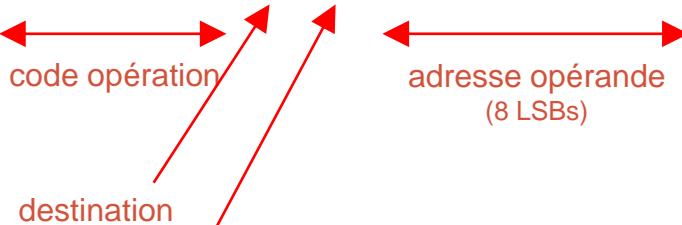
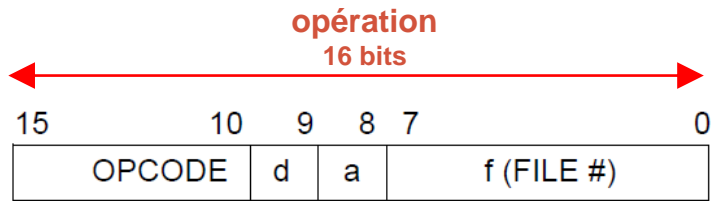


256 octets

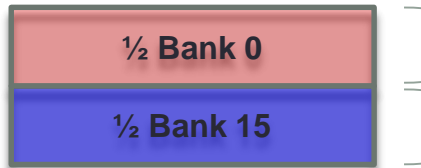
1111 1000 0000
à
1111 1111 1111

128 octets
(128 à 255)





a = 0: **BSR est ignoré**
 le µC accède à l'Access Bank



≈ 128 octets

≈ 128 octets

pour les variables générales

(à déclarer grâce à la directive **UDATA_ACS**
 plutôt que **UDATA** – cf. page 13)

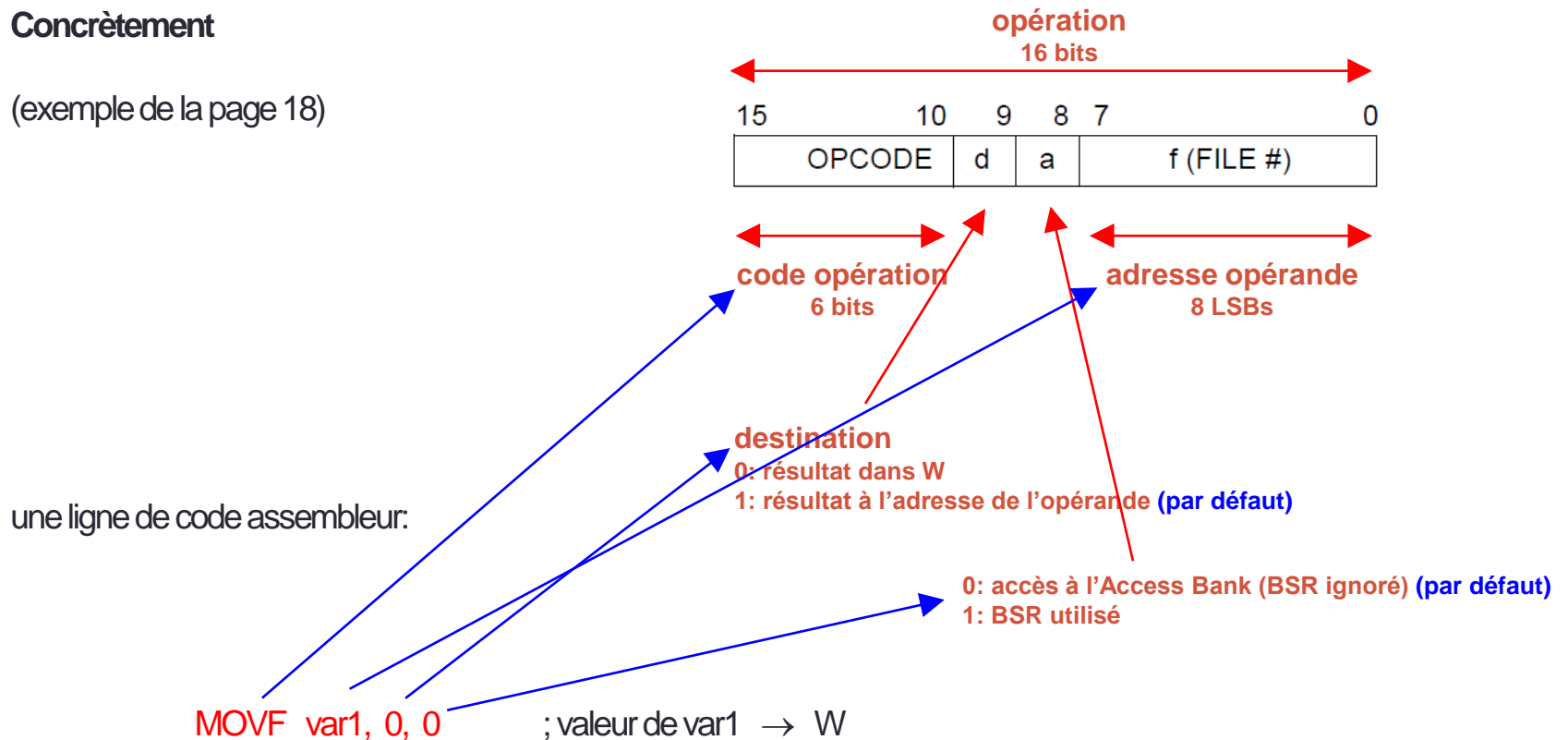
pour les registres servant à configurer le
 matériel (horloge, timers, etc.)

(**S**pecial **F**unction **R**egisters)

a = 1: **BSR est lu par le µC pour former**
 l'adresse du registre à accéder

Concrètement

(exemple de la page 18)



var1 ayant préalablement été déclarée de la manière suivante:

```
UDATA_ACS
```

```
var1 RES 1 ; variable sur 1 octet, se situant dans l'Access Bank (1ère moitié de la Bank 0)
```

3.4 Adressage indirect

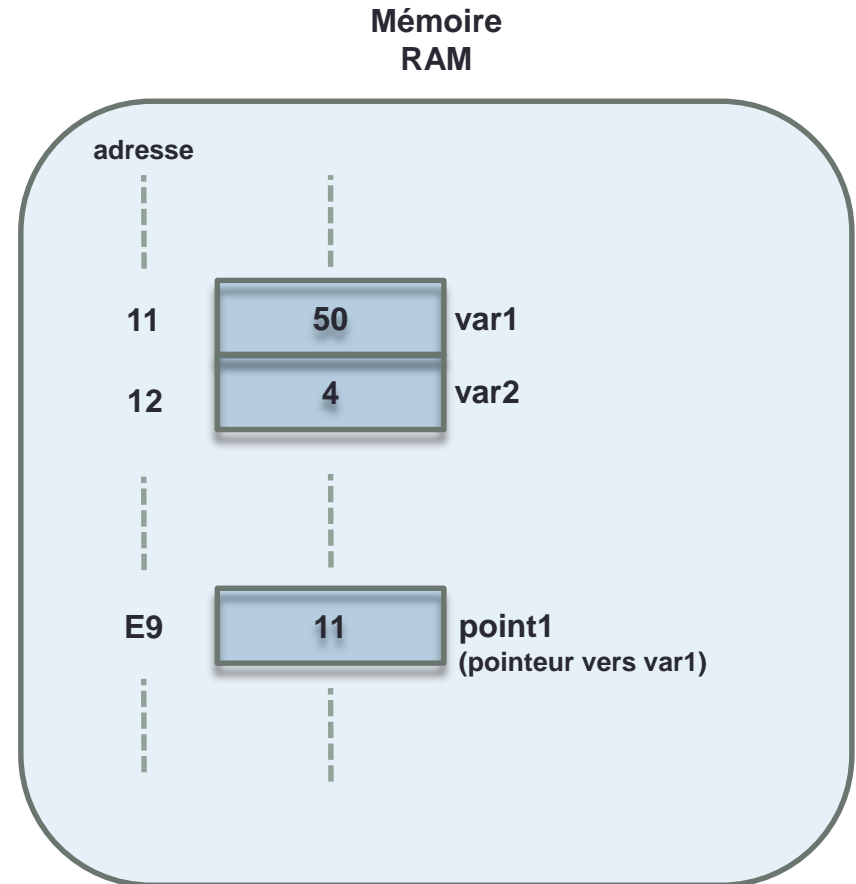
L'opération ne contient pas l'adresse de l'opérande.

Elle indique par contre où la trouver: dans un pointeur.

Par exemple:

- **var1** est un registre en RAM.
Son type est « variable ».
Sa valeur est 50.
Son adresse est 11.
- **point1** est un registre en RAM.
Son type est « pointeur ».
Sa valeur est 11.
(C'est une adresse !)
Son adresse est E9.
(Peu importe !)

⇒ utile pour travailler avec des tableaux
(En incrémentant par exemple le contenu de point1,
on accède à la variable stockée juste derrière var1.)



Pour rappel: utilisation des pointeurs en C

```
uint8_t var1 = 50;           // on déclare une variable de type
                              // entier non-signé sur 8 bits,
                              // et on initialise sa valeur à 50

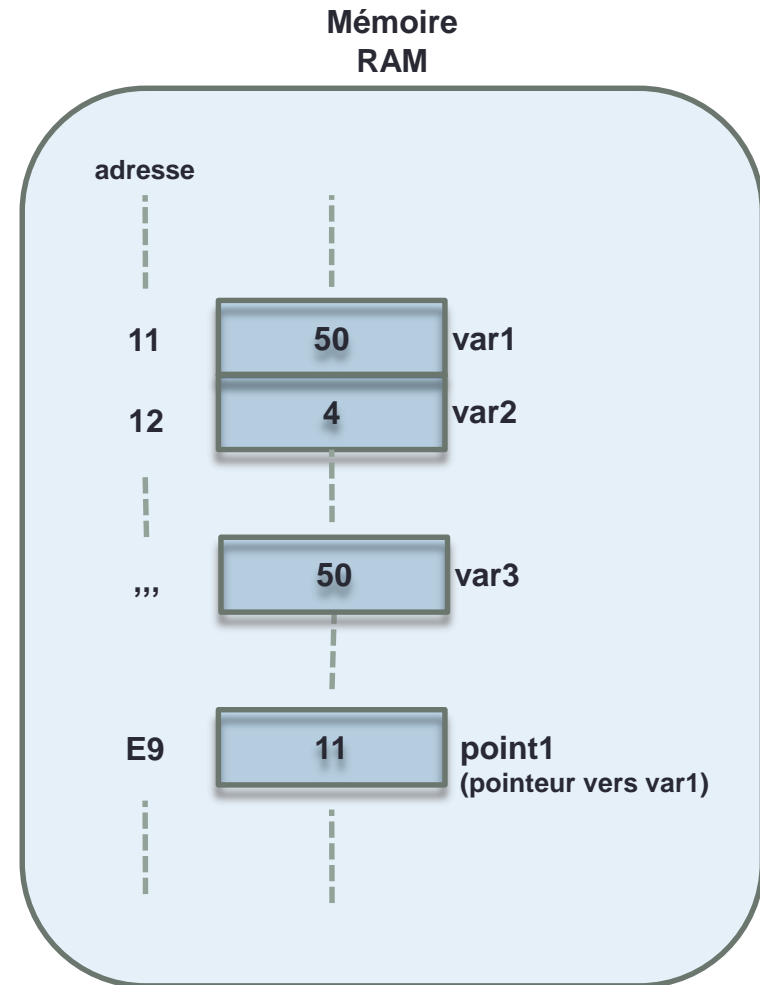
uint8_t var2 = 4;
uint8_t var3;

uint8_t *point1;             // on déclare une variable de type
                              // pointeur: elle contiendra l'adresse
                              // d'une variable de type uint8_t

var3 = var1;                 // => valeur de var3 = valeur de var1

point1 = &var1;              // => valeur de point1 = adresse de var1

var3 = *point1;              // => valeur de var3 = valeur de la variable
                              // dont l'adresse est dans le pointeur
                              // (même chose que var3 = var1)
```



Adressage indirect: utilisation avec PIC18

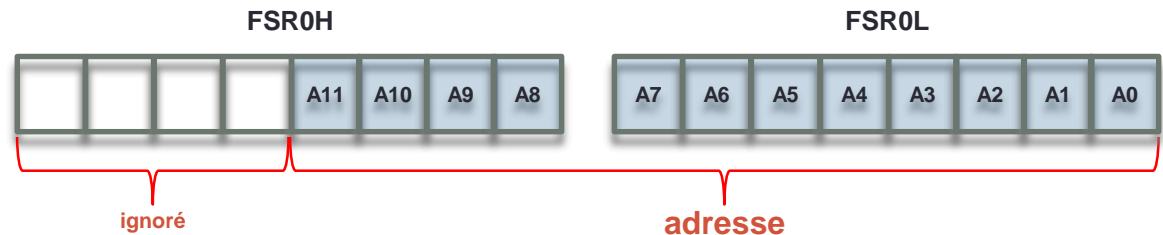
3 emplacements dédiés, prévus pour contenir une adresse de variable en RAM.

rappel: l'adresse de la RAM est sur 12 bits
⇒ il faut 2 octets (donc 2 registres) pour pouvoir la stocker

FSR0H et FSR0L
FSR1H et FSR1L
FSR3H et FSR3L } 3 paires de registres (chacune peut contenir une adresse de RAM)

Pour indiquer dans une opération que l'on souhaite accéder à une variable dont l'adresse est stockée dans **FSR0H : FSR0L**, utiliser le registre **INDF0**.
(Idem pour **INDF1** et **INDF2**.)

Exemple:



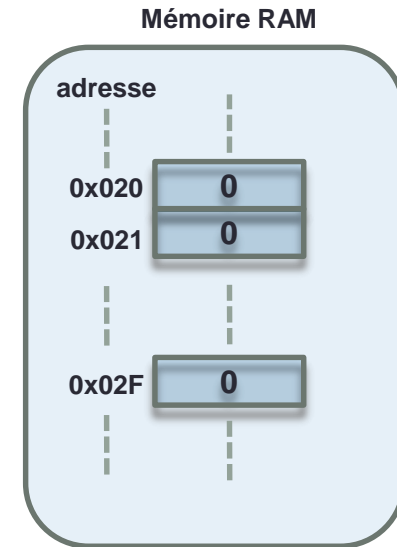
MOVF INDF0, 0 ; lit la variable dont l'adresse est contenue dans FSR0H : FSR0L (ici, pour la recopier dans W)



Inutile de préciser la Bank, puisque l'adresse complète (12 bits) est renseignée.
La variable peut sans problème se trouver n'importe où dans l'espace d'adressage.

Exemple: mise à 0 de la RAM de l'adresse 0x020 à 0x02F

```
CLRF    FSR0H, 0
MOVLW   0x20
MOVWF   FSR0L, 0
```



```
boucle  CLRF    INDF0      ; met à 0 (Clear) le registre dont l'adresse est dans le pointeur FSR0
        INCF    FSR0L, 1    ; incrémente (ajoute 1) au contenu de FSR0L
        BTFSS   FSR0L, 4, 0 ; teste le bit (Bit Test) n°4 du registre (File) FSR0L,
                           ; s'il vaut 1, saute l'opération suivante (Skip if Set)

        GOTO    boucle

suite   ...
        ...
```

Sur le PIC18, il est possible en une seule opération de modifier la valeur du pointeur ET de l'utiliser.

Au lieu d'utiliser **INDF0**, utiliser

	POSTDEC0	(post decrement)
ou	POSTINC0	(post increment)
ou	PREINC0	(pre increment)
ou	PLUSW0	(additionne au contenu de FSR0 la valeur signée de W, FSR0 et W n'étant pas modifiés)

(Idem pour INDF1 et INDF2)

Application à l'exemple précédent:

boucle	CLRF	POSTINC0
	BTFSS	FSR0L, 4, 0
	GOTO	boucle
suite	...	
	...	

Remarque: l'adressage indirect en utilisant les registres FSR0, INDF0, etc, est bien indiqué pour lire ou écrire des tableaux stockés en **RAM**.

Si l'on souhaite lire des tableaux de valeurs stockées en **ROM** (mémoire Flash), on peut utiliser la méthode « **COMPUTED GOTO** »:

le tableau est conçu comme une routine qui retourne une valeur constante dans l'accumulateur.

(cf . page 68 de la datasheet du PIC18F45K20)

...

MOVLW 0x01

MOVWF PCLATH ; charge PCLATH à 0x01 en prévision de ce qui suit

MOVF offset, W ; charge dans W une valeur correspondant au n° de la ligne à lire

CALL lire_tableau ; appelle la routine « lire_tableau »

ORG 0x0100

; directive indiquant au compilateur que les lignes suivantes doivent

; être écrites dans la mémoire Flash à partir de l'adresse 0x0100

lire_tableau ADDWF PCL ; modifie le contenu du Program Counter, en y ajoutant le contenu de W

RETLW 0x12

; sort de la routine avec la valeur 0x12 dans l'accumulateur

RETLW 0x56

RETLW 0x3F

4. Les routines

Une routine (ou sous-programme) permet d'écrire une seule fois une tâche répétitive, qui sera exécutée plusieurs fois (à intervalle régulier ou non) lors de l'exécution d'un programme principal.

L'exécution d'une routine se déroule selon la séquence suivante:

1. **interruption** du déroulement du programme principal
2. « **saut** » au début de la routine, à une adresse définie (« branchement »)

CALL ma_routine

- ⇒ Il ne faut pas exécuter tout de suite l'opération qui suit (dont l'adresse se trouve déjà dans le Program Counter). Il faudra l'exécuter plus tard...
- ⇒ Il faut remplacer l'adresse qui était dans le PC par celle de « ma_routine ».
- ⇒ L'adresse qui était dans le PC doit être sauvegardée (dans la pile (stack)): elle indique où reprendre le programme qui a été interrompu.

3. **exécution** de la routine

3. **retour** ensuite au programme principal

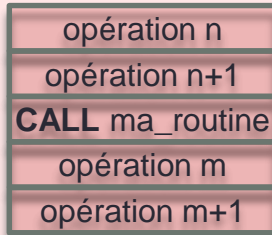
RETURN

- ⇒ L'adresse de retour, sauvegardée en haut de la pile, est rechargée dans le PC.

mémoire PROGRAMME

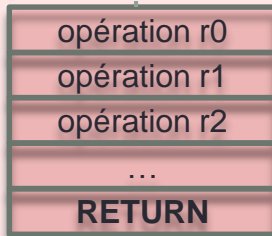
adresse

0x1020
0x1022
0x1024
0x1028
0x102A



ma_routine

0x12FE
0x1300



- Le pointeur de pile est incrémenté pour indiquer où se trouve le haut de la pile (Top-of-Stack).
- Le contenu du PC est sauvegardé en haut de la pile (PUSH).
- Le contenu du PC est remplacé par l'adresse de « ma_routine ».

- L'adresse de retour qui se trouve en haut de la pile est rechargée dans le PC (POP).
- Le pointeur de pile est décrémenté pour mettre à jour la position Top-of-Stack.

pile (stack)

adresse

31

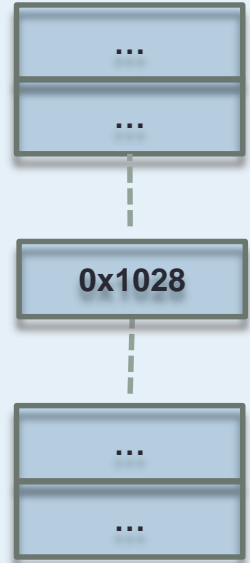
30

5

2

1

0



pointeur de pile (stack pointer)

5

Routines: particularités du PIC18

- La pile a une taille finie: 31 registres de 21 bits.
(Chaque registre peut contenir une adresse de la mémoire programme.)
- Le pointeur de pile est un registre 5 bits.
(Il contient une adresse comprise entre 0 et 31.)
- La pile ne peut pas contenir plus de 31 adresses
⇒ au-delà de 31 appels de routines imbriqués, l'adresse de retour ne peut pas être sauvegardée
« *stack overflow* ».

Solution (compliquée) : gérer la pile de façon logicielle
(transférer le contenu de la pile dans la RAM, et manipuler le stack pointer de façon logicielle)

- Il peut être utile de sauvegarder des registres importants lors de l'appel d'une routine, et de les restaurer en sortant (« *sauvegarde & restauration de contexte* »).
Si ces registres sont modifiés dans la routine, cela permet de revenir au programme principal avec leurs valeurs de départ.

CALL ma-routine, 1 ; W, STATUS et BSR sont sauvegardés dans des registres dédiés

RETURN 1 ; W, STATUS et BSR sont restaurés

5. Les interruptions

Une interruption est un événement **asynchrone**, matériel (le plus souvent) ou logiciel (plus rarement), qui interrompt la séquence normale du programme, afin d'exécuter un sous-programme qui traite l'événement.

Contrairement à une routine classique appelée de manière déterministe par une ligne de code (CALL ma_routine), une routine d'interruption, ou **ISR (Interrupt Service Routine)**, est déclenchée par un événement pouvant se produire n'importe quand (**IRQ – Interrupt ReQuest**).

Exemple:



message
reçu



2 méthodes possibles pour gérer un événement asynchrone:

- **logicielle** : scrutation (ou Polling)

Une boucle du programme teste régulièrement si l'événement attendu s'est produit.

- ⇒ la CPU est occupée par cette tâche
(la plupart du temps pour tester un événement qui n'est pas encore arrivé),
- ⇒ l'événement n'est pas forcément pris en compte immédiatement
(il faut attendre de passer par l'instruction qui fait le test).



- **matérielle**: interruption

Un mécanisme interrompt le programme dès que l'événement attendu s'est produit.

- ⇒ la CPU peut faire autre chose,
- ⇒ l'événement est pris en compte immédiatement.



Une routine classique peut être située n'importe où dans la mémoire programme:
l'opération qui l'appelle (CALL ma_routine) fournit l'adresse du début.

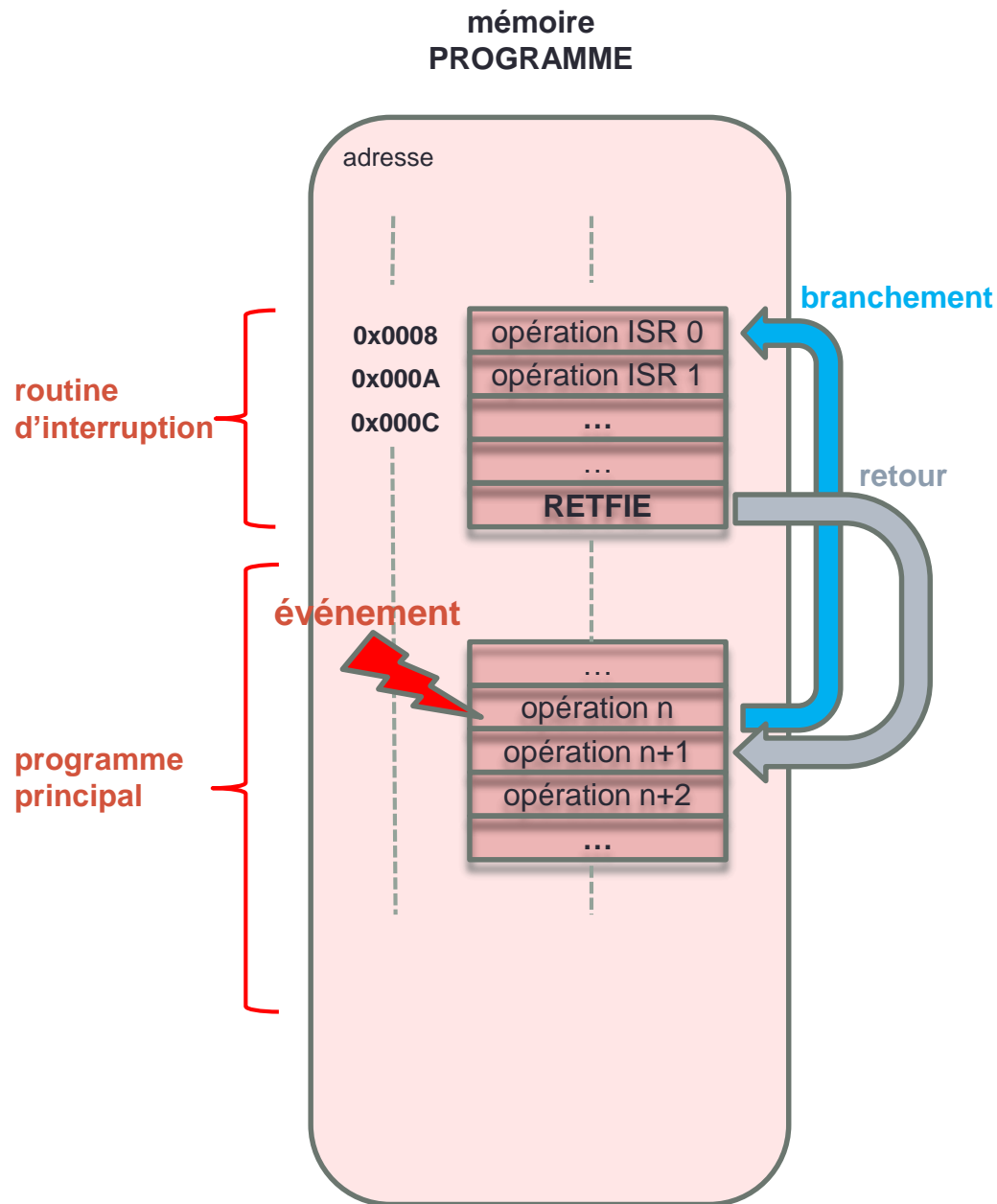
A contrario, une routine d'interruption n'est pas appelée par une ligne de programme.
⇒ Son adresse de début doit être **connue d'avance** par le microcontrôleur.

Sur un PIC18, les sources possibles d'interruption sont (liste non exhaustive):

- | | |
|---|--|
| changement d'état sur une entrée | • External interrupt from the INT, INT1, and INT2 pins |
| | • Change on RB7:RB4 pins |
| limite de temps atteinte | • TMR0 Overflow |
| | • TMR1 Overflow |
| | • TMR2 Overflow |
| | • TMR3 Overflow |
| | • USART Interrupts |
| | - Receive buffer full |
| | - Transmit buffer empty |
| réception d'un message sur un périphérique de transmission / réception (ou fin de transmission) | • SSP Interrupt |
| | • SSP I ² C bus collision interrupt |
| conversion analogique / numérique terminée | • A/D conversion complete |
| | • CCP interrupt |
| tension d'alimentation trop basse | • LVD Interrupt |
| | • Parallel Slave Port |
| | • CAN interrupts |

- Chaque source d'interruption peut être activée ou désactivée (masquée) individuellement.
- Une routine d'interruption ne peut pas être interrompue.
(Sauf par une interruption plus prioritaire.)
- Selon les microcontrôleurs / microprocesseurs, il peut exister plusieurs niveaux de priorité.
Sur un PIC18, il n'existe que 2 niveaux de priorité (haute ou basse).
- Selon les microcontrôleurs / microprocesseurs, l'adresse de début d'une routine d'interruption peut être modifiée ou non.
Sur un PIC18, elle est fixe:

0x00008	(priorité haute)
0x00018	(priorité basse)
- Selon les microcontrôleurs / microprocesseurs, des sources d'interruption (IRQ) différentes peuvent déclencher des routines d'interruption (ISR) différentes.
Sur un PIC18, toutes les IRQ de même priorité déclenchent la même ISR.



1. Un **FLAG** (un bit dédié dans un registre dédié) est levé (le plus souvent: mis à 1) pour indiquer que l'événement s'est produit.

Remarque: C'est une opération matérielle.

Le FLAG est levé, que l'événement soit activé ou non comme source d'interruption.

2. Les interruptions sont désactivées (sauf celles qui sont plus prioritaires).

Remarque: C'est une opération matérielle.

Sur un PIC18, c'est le bit **GIE** qui est mis à 0 (**Global Interrupt Enable**: un bit dédié dans un registre dédié).

3. L'opération en cours se termine.

4. Le contenu du Program Counter est sauvegardé en haut de la pile.

5. Le contenu du PC est remplacé par l'adresse de début de la routine d'interruption.

Remarque: Pour un PIC18, cette adresse vaut **0x08** ou **0x18**.

6. La routine d'interruption s'exécute (comme n'importe quel programme), jusqu'à sa dernière instruction.

Remarque: Sur un PIC18, cette instruction est **RETFIE**.

7. L'adresse de retour est récupérée dans la pile, et rechargée dans le PC.

8. Les interruptions sont réactivées.

Remarque: Sur un PIC18, le bit **GIE** est remis à 1.

- Les mécanismes détaillés à la page précédente s'exécutent de façon matérielle.

(Ce n'est pas le programme qui doit les prendre en charge.)



- Généralement, les flags d'interruption doivent être effacés de façon logicielle, dans la routine d'interruption, juste avant d'en sortir (avant l'instruction **RETFIE**).

Si ce n'est pas fait, la routine d'interruption va boucler infiniment.

- S'il existe une seule routine d'interruption (cas du PIC18), mais plusieurs sources d'interruption possibles, il est possible de savoir quel événement a déclenché l'ISR en testant au début de la routine les flags d'interruption.

En cas d'événements simultanés, la priorité peut être gérée selon l'ordre dans lequel les flags sont testés.

- Si un événement déclencheur intervient alors qu'une ISR est déjà en cours d'exécution, l'événement n'est pas perdu. Il est juste mis en attente (« pending exception »).

(Le flag d'interruption correspondant ayant été levé, une nouvelle interruption sera déclenchée dès que l'ISR en cours sera terminée.)

- Certains registres importants (ex: accumulateur) risquent fort d'être modifiés dans l'ISR.
 - ⇒ Cela peut avoir des conséquences désastreuses pour le programme qui s'est fait interrompre.
C'est notamment le cas si l'interruption est déclenchée au milieu d'un calcul « non atomique »
(qui nécessite plusieurs instructions).
 - ⇒ Il faut « sauvegarder et restaurer le contexte ».

Sur un PIC18, lors d'une interruption de priorité haute, les registres W, STATUS et BSR sont sauvegardés dans des registres dédiés (« shadow registers »).

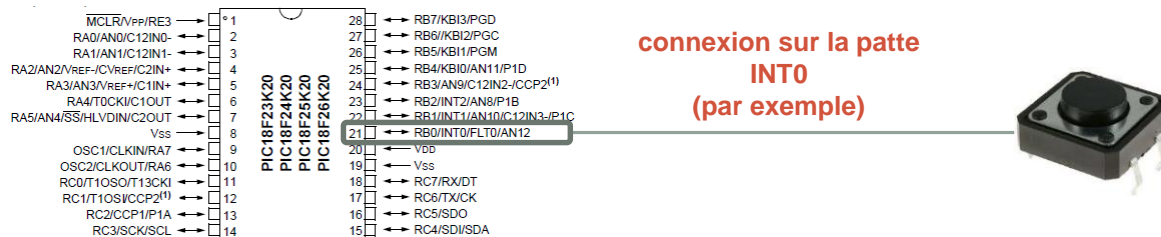
« Sauvegarde du contexte »

Pour les récupérer à la fin de l'ISR, et recopier dans les registres leurs valeurs initiales, il faut utiliser l'instruction

RETFIE 1

« Restauration du contexte »

Exemple: configuration du PIC18 pour déclencher une interruption lors de l'appui sur un bouton



1. Activer l'interruption sur la patte INT0.
⇒ Mettre à 1 le bit **INT0IE** (**INT0** External Interrupt **E**nable).

REGISTER 9-1: INTCON: INTERRUPT CONTROL 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

2. Préciser si le déclenchement se fait sur front montant ou descendant.
⇒ Fixer le bit **INTEDG0** (**INT0** External Interrupt **EDG**e Select) selon ce qui est précisé dans la datasheet.

REGISTER 9-2: INTCON2: INTERRUPT CONTROL 2 REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	U-0	R/W-1	U-0	R/W-1
RBPU	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP
bit 7							bit 0

3. Vérifier que le flag d'interruption INT0IF (**INT0** Interrupt **F**lag) est bien à 0.
☞ Il ne faudra pas oublier de le remettre à 0 dans la routine d'interruption.

REGISTER 9-1: INTCON: INTERRUPT CONTROL 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

4. Activer globalement les interruptions.
⇒ Mettre à 1 le bit **GIE** (**G**lobal Interrupt **E**nable).

REGISTER 9-1: INTCON: INTERRUPT CONTROL 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