Cours - Microinformatique (MicroInfo)

Cours créé par Cédric Bornand, Pierre Favrat et Bertrand Hochet

HEIG-VD mars 2023
rev. 1.1

Microinformatique - HEIG-VD

Informations générales

Professeur	Cédric Bornand
Email	cedric.bornand@heig-vd.ch
Téléphone	024 557 27 75
Bureau	Y-Parc, CH-1400 Yverdon-les-Bains, bureau C0.03

Professeur	Pierre Favrat
Email	pierre.favrat@heig-vd.ch
Téléphone	024 557 27 77
Bureau	Y-Parc, CH-1400 Yverdon-les-Bains, bureau C0.03

Professeur	Bertrand Hochet
Email	bertrand.hochet@heig-vd.ch
Téléphone	024 557 27 68
Bureau	Y-Parc, CH-1400 Yverdon-les-Bains, bureau C0.03

HISTORIQUE - MODIFICATIONS MAJEURES

VERSION	DATE	DESCRIPTION
V 0.0	février 2014	Premier draft
V 0.1	19 février 2016	Version initiale (PFT)
V 0.2	19 mars 2016	Added chapters 5,6,11 (BHT)
V 0.3	22 mars 2016	Added chapters 7,8 (PFT, BHT)
V 0.4	2 février 2017	Added chapters 3,4 (PFT)
V 1.0	15 mars 2017	First release (PFT)
V 1.1	15 mars 2023	Few modifications (CBD))

Table des matières

Liste des tableaux

Chapitre 1

Introduction

La microinformatique est le sous-domaine de l'informatique proche du matériel. Elle s'exprime à l'aide de langages dit de bas niveau tels que l'assembleur et le C. On utilise la microinformatique dans les systèmes embarqués ou dans les ordinateurs lorsque l'on a des contraintes de performances, de coût ou de consommation.

1.1 Place du cours MicroInfo dans le cursus de l'ingénieur

Le cours de microinformatique est un cours technologique basé sur des microprocesseurs qui sont le composant principal des microcontrôleurs. Il se situe parmi les autres branches du génie électrique dans la partie de mise en oeuvre (fig. 1.1).

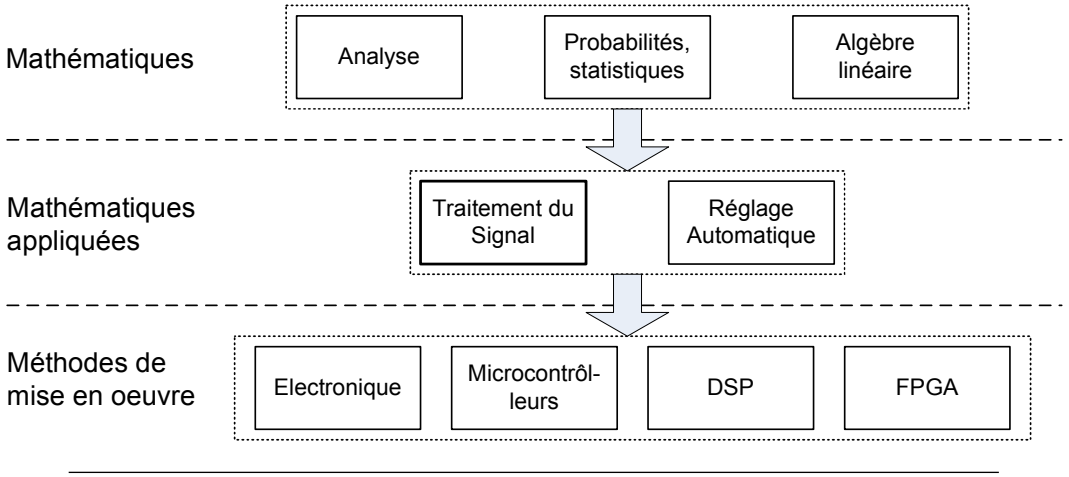

FIGURE 1.1 – Situation du cours MicroInfo dans le cursus de l'ingénieur

1.2 Définitions : processeur, microprocesseur, microcontrôleur et système-on-chip

Un processeur est un système qui permet l'exécution d'opérations élémentaires telles que des opérations arithmétiques (additions, soustractions, multiplication et division), des opérations logiques (OR, AND, XOR), des tests (égal à, plus petit que, etc.) et des déplacements de données.

Le microprocesseur est un système micro-électronique, aussi appelé circuit intégré ou plus communément "chip" ou "puce", permettant l'exécution d'opérations élémentaires. Contrairement au processeur qui est un concept, le microprocesseur est un composant que l'on place sur un circuit imprimé.

Un microcontrôleur est un système micro-électronique contenant un microprocesseur et des périphériques. Les périphériques essentiels sont les minuteries (ou "timers"), les interfaces de communication série et les mémoires (données et instructions).

Le système-on-chip est un ensemble, différent du microcontrôleur, qui inclut tous les composants électroniques d'un ordinateur à part les mémoires. Il se présente sous la forme d'un circuit intégré. La mémoire

de données peut dans certain cas être assemblée en dessus du chip pour réduire la taille du système.

1.3 Éléments de systèmes logiques

Pour pouvoir configurer un microcontrôleur de manière efficace, nous avons besoins de quelques éléments de systèmes logiques.

1.3.1 Définitions

Etat logique : chacune des 2 valeurs que peut prendre une variable logique
Variable logique : grandeur qui ne peut prendre que les 2 états logiques
Fonction logique : variable logique qui dépend d'autres variables
Table de vérité : Liste des valeurs de sortie en fonction des combinaisons des entrées

Variables		Sortie
A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 1.1 – Exemple de table de vérité à deux variables (fonction ET)

La table de vérité permet de déterminer l'équation logique. Dans l'exemple ci dessus nous avons représenté la fonction A ET B = S. Pour construire une table de vérité on procède de la façon suivante :

1. on attribue une colonne pour chaque variable ou entrée (A B C ...)
2. on attribue une colonne pour chaque résultat ou sortie
3. on liste toutes les combinaisons des variables d'entrée (2^n combinaisons)
4. on calcule le résultat pour chaque ligne

1.3.2 Fonctions logiques

Si le nombre de variables d'entrée n'est pas trop grand, on peut faire l'inventaire exhaustif de toutes les fonctions imaginables (Table 1.2).

Dans le tableau 1.2, on peut repérer trois fonctions particulières :

1. inversion ;
2. fonction dans laquelle une seule configuration des entrées donne 1 ;
3. fonction dans laquelle une seule configuration des entrées donne 0.

1.3.3 Opérateurs élémentaires

Il y a trois fonctions de base qui permettent de créer toutes les autres fonctions logiques :

NON (NOT)	$\overline{A}$	not A
ET (AND)	$A \cdot B$	A and B
OU (OR)	$A + B$	A or B

ET et OU sont duals, c-à-d qu'il ont des comportements semblables, mais avec les variables inverses. On peut nommer d'autres fonctions de la table 1.2 par convenance :

NON ET (NAND)	$\overline{A \cdot B}$	not(A and B)
NON OU (NOR)	$\overline{A + B}$	not(A or B)
OU exclusif (XOR)	$A \oplus B$	A xor B
NON OU exclusif (XNOR)	$\overline{A \oplus B}$	not(A xor B)

Variables		Fonctions															
A	B	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	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

$$
\begin{array}{lcllcll}
F0 & = & 0 & F15 & = & \overline{F0} & = & 1 \\
F1 & = & A \cdot B & F14 & = & \overline{F1} & = & \overline{A \cdot B} \\
F2 & = & A \cdot \overline{B} & F13 & = & \overline{F2} & = & \overline{A} + B \\
F3 & = & A & F12 & = & \overline{F3} & = & \overline{A} \\
F4 & = & \overline{A} \cdot B & F11 & = & \overline{F4} & = & A + \overline{B} \\
F5 & = & B & F10 & = & \overline{F5} & = & \overline{B} \\
F6 & = & A \oplus B & F9 & = & \overline{F6} & = & \overline{A \oplus B} \\
F7 & = & A + B & F8 & = & \overline{F7} & = & \overline{A + B}
\end{array}
$$

TABLE 1.2 – Résultats possibles de fonctions à deux variables

Ces quatre fonctions supplémentaires peuvent toutes s'exprimer en fonction des trois fonctions de base. Pour les fonctions de plus de 2 variables on peut se ramener à une fonction de 2 variables par substitution. Ceci pour montrer que toute fonction logique peut se ramener aux trois fonctions de base.

1.3.4 Postulats de l'algèbre de Boole

De l'hypothèse que l'inverse d'une variable ne peut jamais avoir la même valeur que la variable, on trouve :

$$
\begin{array}{c}
A + \overline{A} = 1 \\
A \cdot \overline{A} = 0
\end{array}
\tag{1.1}
$$

Ce sont les postulats de l'algèbre de Boole.

1.3.5 Théorèmes

Grâce aux postulats nous pouvons poser une liste de théorèmes qui vont nous permettre de simplifier des équations logiques :

1	$\overline{\overline{A}}$	involution not(not A) = A
2	$A + 0 = A$	
3	$A \cdot 0 = 0$	
4	$A + 1 = 1$	
5	$A \cdot 1 = A$	
6	$A + A = A$	idempotence de l'opérateur OU
7	$A \cdot A = A$	idempotence de l'opérateur ET
8	$A + B = B + A$	commutativité
9	$A \cdot B = B \cdot A$	commutativité
10	$A + (B + C) = (A + B) + C = A + B + C$	associativité
11	$A \cdot (B \cdot C) = (A \cdot B) \cdot C = A \cdot B \cdot C$	associativité
12	$A + B \cdot C = (A + B) \cdot (A + C)$	distributivité
13	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	distributivité
14	$A \cdot B + \overline{A} \cdot B = B$	absorption
15	$\overline{A + B} = \overline{A} \cdot \overline{B}$	Théorème de De Morgan
16	$\overline{A \cdot B} = \overline{A} + \overline{B}$	Théorème de De Morgan
17	$(A \cdot B) + (\overline{A} \cdot C) + (B \cdot C) = (A \cdot B) + (\overline{A} \cdot C)$	Théorème du Consensus
18	$(A + B) \cdot (\overline{A} + C) \cdot (B + C) = (A + B) \cdot (\overline{A} + C)$	Théorème du Consensus

1.3.6 Ecriture canonique

Définitions :

- mintermes : les lignes de la table de vérité qui donnent 1 comme résultat par fonction ET
- maxtermes : les lignes de la table de vérité qui donnent 0 comme résultat par fonction OU

Par convention, on note m_x le minterme tel que x est le nombre décimal égal au code binaire formé par les variables. Par exemple, m_5 est le minterme correspondant à la configuration [101] des variables d'entrée. Par convention, on note M_x le maxterme tel que x est le nombre décimal égal au code binaire formé par les variables. Par exemple, M_2 est le maxterme correspondant à la configuration [10] des variables d'entrée. Une conséquence importante est que les mintermes dépendent de l'ordre dans lequel les variables sont considérées.

Pour déterminer la forme canonique d'une fonction logique dont on connait la table de vérité on peut faire la somme des mintermes :

$$F = m_1 + m_2 + m_3 + m_6 + m_7$$

A	B	C	F	
0	0	0	0	
0	0	1	1	$\overline{A} \cdot \overline{B} \cdot C = 1$ OU
0	1	0	1	$\overline{A} \cdot B \cdot \overline{C} = 1$ OU
0	1	1	1	$\overline{A} \cdot B \cdot C = 1$ OU
1	0	0	0	
1	0	1	0	
1	1	0	1	$A \cdot B \cdot \overline{C} = 1$ OU
1	1	1	1	$A \cdot B \cdot C = 1$

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

On peut aussi utiliser l'opération duale qui est le produit des maxtermes :

$$F = M_0 \cdot M_4 \cdot M_5$$

A	B	C	F	
0	0	0	0	$A+B+C$ exclut cette ligne
0	0	1	1	
0	1	0	1	
0	1	1	1	
1	0	0	0	$\overline{A}+B+C$ exclut cette ligne
1	0	1	0	$\overline{A}+B+\overline{C}$ exclut cette ligne
1	1	0	1	
1	1	1	1	

$$F = (A+B+C)\cdot(\overline{A}+B+C)\cdot(\overline{A}+B+\overline{C})$$

Il est aisé de montrer que la somme des mintermes est égale au produit des maxtermes. Ceci montre encore une fois la dualité des opérations logiques. On constate aussi que le nombre de maxtermes est plus petit (3) que celui des mintermes (5). Il est donc plus simple d'utiliser le produit des maxtermes dans ce cas ci.

Le nom minterme vient du fait que dans une somme de produits, seulement un terme de la somme doit être à 1 pour donner un résultat de 1. Au contraire, dans un produit de sommes, il faut que toutes les sommes soient à 1 et donc on l'appelle maxterme.

La forme canonique est rarement optimale, il faut donc la simplifier si on cherche à optimiser l'implémentation de la fonction. Il existe des méthodes manuelles et des méthodes algorithmiques. On utilisera la méthode des tables de Karnaugh pour une réduction manuelle si le nombre de variables est limité. Au delà de 4 variables on a intérêt à utiliser une méthode algorithmique telle que celle de Quine-McCluskey.

1.3.7 Opérateurs logiques en C

La programmation des microcontrôleurs s'effectue principalement en C (2016). Nous avons donc besoin de savoir comment les opérations logiques sont représentées dans ce langage. En C, on différentie la logique sur 1 bit appelée "bit à bit" de celle sur un mot complet (8-16-32 bits). En effet, les deux types produisent des résultats différents car le FAUX sur un mot est bien '00000000' par exemple mais le VRAI comprend toutes les autres valeurs.

Opérateur	Signification	Notation C	Utilisation typique
AND	ET	&	Forçage à 0 d'un bit dans un registre
OR	OU	|	Forçage à 1 d'un bit dans un registre
XOR	OU exclusif	^	Inversion d'un bit dans un registre
NOT	inversion	~	Inversion de tous les bits d'un registre

Table 1.3 – Opérateurs logiques bit à bit en C

Exemple :

```
char REG;
REG &= 0x7F; //force à 0 le bit no 7
```

Opérateur	Signification	Notation C	Utilisation typique
AND	ET	&&	Test logique
OR	OU	||	Test logique
NOT	négation	!	Test logique
	équivalence	==	Test logique

Table 1.4 – Opérateurs logiques booléens en C

Exemple :

```
if (REG == 0x0F)
    printf("REG est égal à 0x0F\n");
```

Les opérateurs bit à bit s'utilisent souvent pour la configuration de registres dans les microcontrôleurs, exemple :

```
REG = P1IN & (BIT5 | BIT2);
```

Les opérateurs booléens quand à eux s'utilisent surtout dans la simplification d'expressions, exemple :

```
if (!(!x && !y)) est équivalent à if (x || y)
```

Chapitre 2

GPIO (General Purpose Input Output)

Le GPIO (General Purpose Input/Output) est une broche d'entrée ou sortie à usage générique. Les GPIOs sont groupées en ports de 8bits ou 16bits. Au sein d'un port ont peut accéder à chaque broche indépendamment en utilisant un masque (voir ğ2.2).

Les GPIOs sont configurés é l'aide de registres qui ont la dimension d'un port (8 ou 16 bits). Un port est donc juste un groupe de GPIOs qui permet une écriture simplifiée du code.

2.1 Matériel

Les GPIOs sont présents sur tous les microcontrôleurs en nombre plus ou moins grand selon le nombre de broches disponibles sur le boîtier. On choisit souvent un microcontrôleur en fonction du nombre de GPIO disponibles.

Le fonctionnement matériel du GPIO est décrit à la fig. 2.1.

Il contient trois parties principales :

- Le circuit de sortie
- Le circuit d'entrée
- Un ou plusieurs commutateurs pour connecter la broche

Au delà du circuit de base décrit auparavant, il existe d'autres fonctions optionnelles dans les GPIO comme :

- synchronisation avec un signal ou une horloge
- circuit pour traiter des interruptions (voir ch. 4)
- convertisseur de niveaux comme par exemple 3.3V - 5V
- configuration de la puissance de l'amplificateur de sortie

2.1.1 Input

Dans la partie "entrée" du GPIO on trouve un tampon qui mémorise la valeur logique présente à l'entrée ainsi qu'un bloc pour sélectionner une résistance de pull-up ou pull-down. Ce bloc est très important car il est nécessaire pour éviter qu'une entrée se retrouve flottante. Si une entrée est laissée non-connectée (flottante) elle peut alors prendre n'importe quelle valeur et même changer de valeur aléatoirement au cours du temps. Pour éviter ce phénomène on doit utiliser une résistance placée selon le schéma suivant fig. 2.2.

La valeur de cette résistance est choisie de manière optimale entre deux facteurs : i) le courant consommé VCC-GND qui doit être le plus faible possible quand l'interrupteur est fermé et ii) La vitesse à laquelle la résistance décharge la capacité d'entrée du circuit intégré CMOS lorsque l'interrupteur est ouvert. Les valeurs typiques se situent entre $10k\Omega$ et $100k\Omega$. Mais attention sous 3.3V, ça consommera lorsque l'interrupteur est fermé, entre $330\mu A$ et $33\mu A$ ce qui n'est pas négligeable !

Dans le cas particulier du MSP430F5529 la valeur typique des résistances de pull est de $35k\Omega$ ce qui provoque un courant de $86\mu A$ sous 3V. Il est donc avantageux de choisir un interrupteur poussoir plutôt que commutant pour minimiser le temps total de consommation. On peut aussi réduire la tension d'alimentation à 1.8V pour réduire le courant mais attention à la réduction de vitesse du microcontrôleur.

2.1.2 Output

Dans la partie "sortie" du GPIO on trouve un tampon qui mémorise la valeur logique que l'on veut sortir sur la broche ainsi qu'un amplificateur pour permettre au circuit de commuter la charge de sortie rapidement. Cette charge est souvent de type capacitive avec une valeur comprise entre 2pF et 10pF typiquement. On peut donc calculer le courant nécessaire pour charger cette capacité à 90% dans un temps voulu. Ensuite on peut ajuster le courant, si disponible dans le GPIO, pour atteindre la vitesse recherchée.

2.2 Logiciel

Pour programmer les GPIOs on utilise des registres organisés en ports. Par exemple, dans la nomenclature Texas Instrument [réf], le port1 contient huit GPIO et on les nomme P1.0 à P1.7. On peut aussi utiliser le groupement par seize où la notation devient PA, PB, etc. PA est équivalent aux ports P0 et P1 regroupés alors que PB est équivalent aux ports P2 et P3.

2.2.1 Programmation par masque

Comme les GPIO sont groupés en paquets de 8 ou 16 du point de vue logiciel, il faut un moyen pour programmer chaque GPIO individuellement. Cela signifie que si on modifie un GPIO, il ne faut surtout pas affecter les autres du même groupe. Un moyen d'arriver à ceci est d'utiliser un masque qui représente le bit désiré. Par exemple si on désire sélectionner le bit 3 on utilisera le masque BIT3 = 00001000 sur 8 bits (La numérotation commence é zéro). Si on désire sélectionner le bit 3 sur 16 bits alors le masque sera BIT3 = 0000000000001000.

Soit un registre (REG) de 8 bits servant à configurer 8 broches d'un microcontrôleur (fig 2.3).

Pour mettre à 1 le bit 0 sans changer les bits 1 à 7 on peut utiliser la fonction OU ; REG = REG OU BIT0 avec BIT0 = 00000001 :

```
REG = REG | BIT0  //en C avec BIT0 = 1
REG |= BIT0       //en C simplifié
```

On appelle cette technique le masquage. Pour mettre un bit à 0, il faut utiliser une autre fonction logique car le OU logique ne fait rien avec un zéro. Il est intéressant de résoudre ce problème par soi-même pour trouver la solution. On cherche donc à mettre un bit à 0 sans changer l'état logique des autres bits du port. Pour trouver la solution on peut partir d'un exemple et procéder par élimination :

```
REG = 00000011    //on désire mettre le premier bit à zéro sans influencer
    les autres
\\test avec OU :
REG = REG | BIT0  //ne fait rien donc ce n'est pas la solution
\\test avec ET :
REG = REG & BITO  //REG = 00000001 FAUX ça ne marche pas non plus
```

Mais si on avait inversé BIT0 alors ça aurait marché :

```
REG = REG & ~BITO   //REG = 00000010 JUSTE
REG &= ~BIT0         //en C simplifié
```

On constate donc qu'il faut utiliser la fonction duale mais avec une inversion préalable. Ceci convient pour l'écriture dans un registre mais qu'en est-il de la lecture ? Comment pourrait-on lire un GPIO parmi tous les bits d'un port ?

La réponse est similaire au test d'un bit dans un mot en langage C. Prenons comme exemple la lecture, ou test, du BIT0 dans REG et procédons par élimination. Il faut trouver la fonction désirée avec par exemple REG = 00000011 et BIT0 = 00000001 :

```
//essai avec OU avec REG = 00000011 et BIT0 = 00000001 :
TEST_BIT0 = REG | BIT0  //TEST_BIT0 = 00000011 FAUX
//et si REG = 00000010
TEST_BIT0 = REG | BIT0  //TEST_BIT0 = 00000011 FAUX

//Il est clair que ça ne marche pas avec OU essayons avec ET :
TEST_BIT0 = REG & BIT0  //TEST_BIT0 = 00000001 JUSTE
//et si REG = 00000010
TEST_BIT0 = REG & BIT0  //TEST_BIT0 = 00000000 JUSTE
```

Donc le test ou lecture d'une valeur s'effectue avec l'opérateur logique ET. Dans le cas d'un test en C, on doit se rappeler que le résultat est VRAI pour toute valeur de TEST_BITx différente de zéro. Il est faux si et seulement si TEST_BITx = 0. Ceci provient du fait que la variable C la plus petite est de 8 bits. Il n'y a pas de variable de 1 bit en C.

2.2.2 écriture dans un registre

Soit le registre REG d'une longueur de 8 bits. Pour écrire la valeur logique 1 dans le registre on utilise la fonction OU entre le registre et le masque du bit voulu. Pour écrire un 0 logique on utilise la fonction ET entre le registre et le masque du bit voulu inversé, exemples :

```
//Mise à 1 du bit zéro :
REG = REG | BIT0    //en C avec BIT0 = 1
REG |= BIT0         //en C simplifié

//Mise à 0 du bit zéro :
REG = REG & ~BITO   //REG = 00000010 JUSTE
REG &= ~BIT0         //en C simplifié
```

Pour la mise à 1 ou 0 simultanée de plusieurs bits on utilise le OU pour sommer les masques et créer un masque contenant les deux bits. Exemple : BIT3 | BIT0 = 00001001 :

```
//Mise à 1 du bit zéro et trois simultanément :
REG |= BIT0 | BIT3

//Mise à 0 du bit zéro et trois simultanément :
REG &= ~BIT0 | BIT3
```

2.2.3 Lecture d'un registre

Pour lire un registre on compare le registre avec le masque du bit voulu en utilisant la fonction ET :

```
//Lecture du BIT3 uniquement :
TEST_REG = REG & BIT3; // BIT3=00001000
```

Le résultat de cette affectation sera donc 0 si le BIT3 est à zéro et 8 si le BIT3 est à 1. Pour lire plusieurs bits à la fois on peut, comme pour l'écriture, créer un masque combiné. Exemple, on veut lire le bit 5 et le bit 2 simultanément :

```
//Lecture du BIT5 et du BIT2 :
TEST_REG = REG & (BIT5 | BIT2); // BIT5 | BIT2 = 00100100
```

2.3 Configuration des GPIO

Avant de pouvoir utiliser un GPIO il faut le configurer. Cela inclut principalement :

- Le multiplexeur d'entrée
- La fonction entrée ou sortie
- La résistance de pull-up ou pull-down lorsqu'on est en entrée

Pour les configurer on doit trouver leurs adresses physiques qui les identifient à l'intérieur du circuit intégré. Ces adresses peuvent être trouvées dans la documentation du microcontrôleur mais souvent elles sont fournies sous forme de constantes dans une librairie qu'on peut inclure dans le code C.

```
#include <msp430.h>

#define P1IN    (PAIN_L)        /* Port 1 Input */
#define P1OUT   (PAOUT_L)       /* Port 1 Output */
#define P1DIR   (PADIR_L)       /* Port 1 Direction */
#define P1REN   (PAREN_L)       /* Port 1 Resistor Enable */
#define P1DS    (PADS_L)        /* Port 1 Drive Strength */
#define P1SEL   (PASEL_L)       /* Port 1 Selection */
```

Exemple de configuration du port P1.2 et P1.5 en sortie :

```
P1SEL &= ~BIT5 & ~BIT2;  // Reset P1.2 et P1.5 to select GPIO
P1DIR |=  BIT5 | BIT2;   // Set P1.2 et P1.5 to output direction
//Option : ajouter le P1DS pour changer le courant de sortie

//Puis écriture sur les sorties P1.5 et P1.2 d'un 1 logique :
P1OUT |= BIT5 | BIT2;
```

Exemple de configuration du port P1.2 et P1.5 en entrée :

```
P1SEL &= ~BIT5 & ~BIT2;  // Reset P1.2 et P1.5 to select GPIO
P1DIR &= ~BIT5 & ~BIT2;  // Reset P1.2 et P1.5 to input direction
//Option : ajouter les résistances de pull
P1OUT |= BIT5 | BIT2;    // Sélection de la résistance
P1REN |= BIT5 | BIT2;    // Activation de la résistance

//Puis lecture de l'entrée P1.5 et P1.2 :
TEST_REG = P1IN & (BIT5 | BIT2);
```

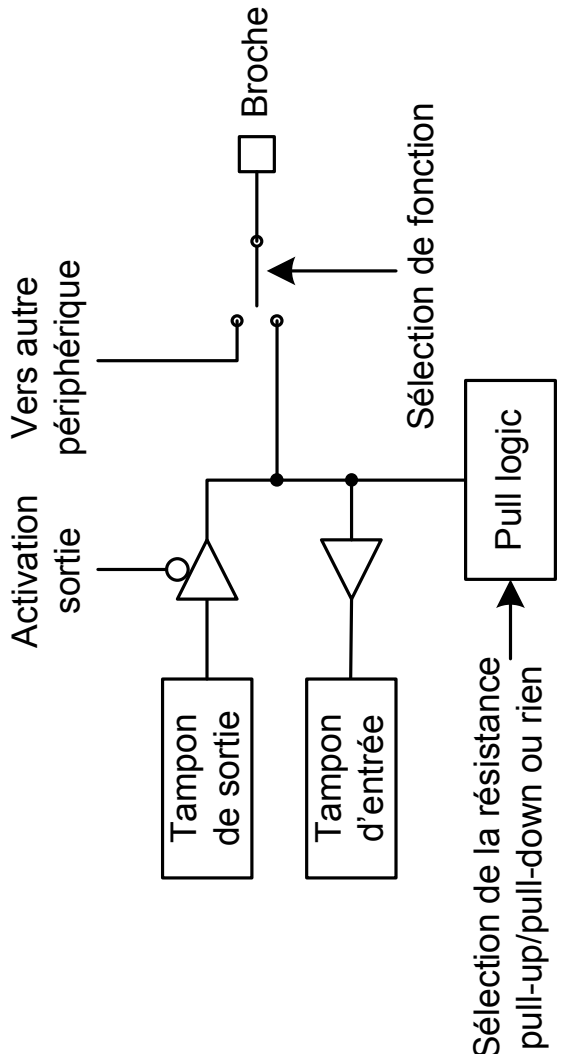

FIGURE 2.1 – Description matériel d'un GPIO

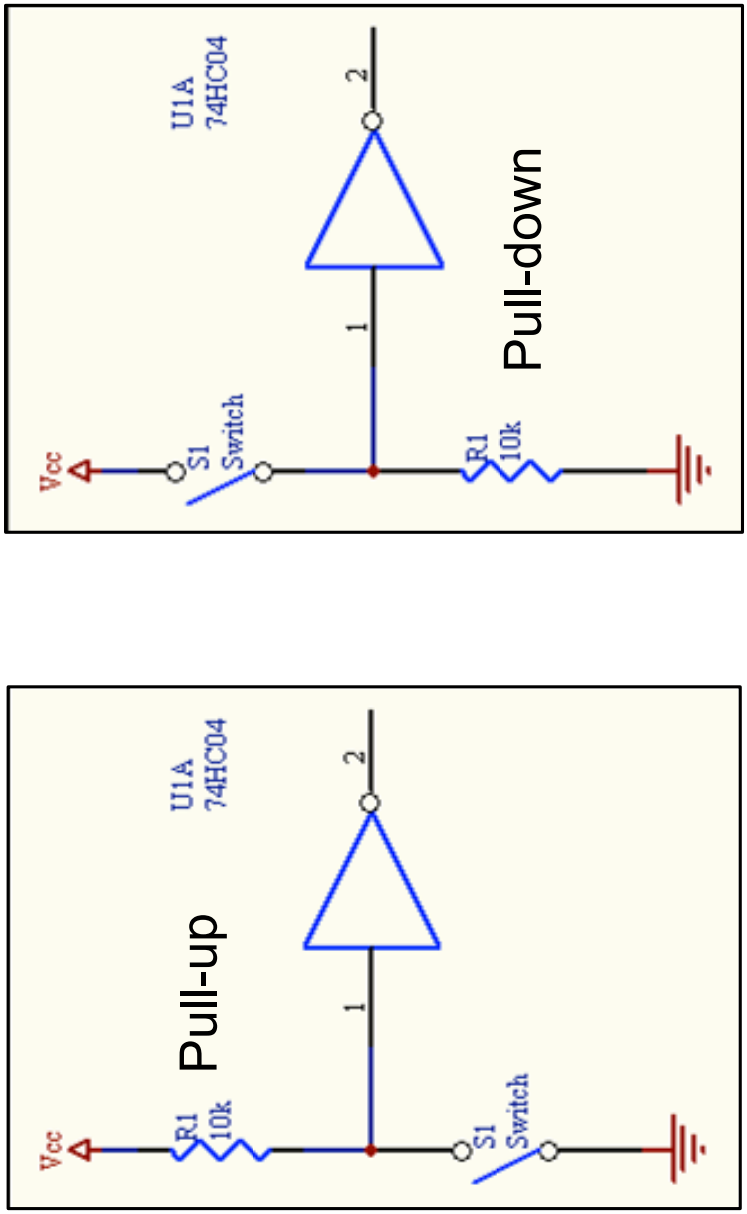

FIGURE 2.2 – Résistances de pull-up et pull-down

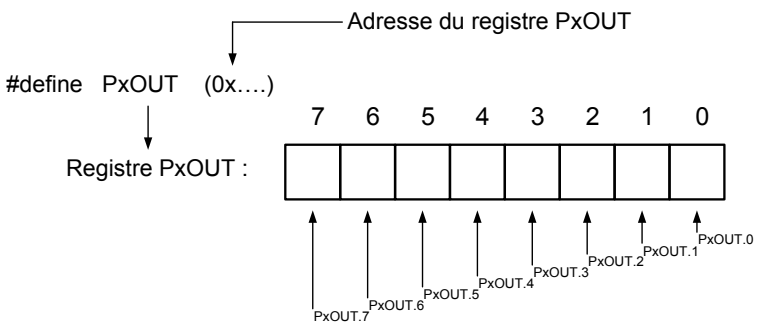

FIGURE 2.3 – Tampon de sortie

Chapitre 3

Numération et arithmétique des ordinateurs

La représentation des nombres, ainsi que l'arithmétique, est très critique pour une utilisation optimale par un processeur. Historiquement on a cherché des méthodes qui minimisent le nombre d'opérations à effectuer pour réduire la taille du matériel. Bien qu'aujourd'hui ce ne soit plus si critique, c'est toujours important pour maximiser la vitesse et minimiser la consommation des processeurs.

3.1 Codage binaire et hexadécimal

Il est utile de rappeler le codage des nombres entiers positifs en base 2 et 16 avant de progresser vers les nombres négatifs et les nombres fractionnaires :

Entier	Binaire	Hexadécimal	Entier	Binaire	Hexadécimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

TABLE 3.1 – Codage binaire et hexadécimal des entiers positifs sur 4 bits

3.2 Représentation des nombres

Les circuits logiques, composants des processeurs, ne fonctionnent qu'en binaire. Il faut donc convertir tous les nombres usuels, pour pouvoir les utiliser. Il est aisé d'attribuer un code binaire à un nombre donné. Par exemple 000 pour le zéro et 001 pour le 1. Mais ceci devient un problème lorsqu'on ajoute le signe moins et la virgule qui ne sont pas des chiffres.

3.2.1 Entiers

On représente les entiers positifs simplement à l'aide d'une conversion décimale-binaire directe (Table 3.1).

Pour les nombres entiers négatifs nous devons faire un choix, prenons par exemple le codage avec 4 bits. Nous avons donc $2^4 = 16$ codes différents disponibles et nous devons aussi coder le zéro ce qui donne $16 - 1$ nombres disponibles. Comme c'est impaire nous aurons soit un nombre de plus du coté positif ou un nombre de plus du coté négatif. Le choix fait a été celui d'avoir un nombre négatif de plus (Table 3.2). Ce qui donne 2^{n-1} du côté positif pour accommoder le zéro et 2^n du côté négatif.

Complément à deux Il nous reste à déterminer quels codes attribuer aux nombres négatifs. En effet on peut choisir de placer les nombres négatifs à la suite des nombres positifs **ou** avant le zéro en faisant le calcul $0 - 1$ en binaire ou encore en décalant tous les nombres. Comparons les trois possibilités et notons bien les différences :

Entier	Cas 1	Cas 2	Cas 3
0	0000	0000	1000
1	0001	0001	1001
7	0111	0111	1111
-1	1000	1111	0111
-2	1001	1110	0110
-8	1111	1000	0000

TABLE 3.2 – Trois possibilités pour coder les nombres entiers négatifs

On constate que le cas 1 offre une discontinuité entre -1 et 0 ce qui rend cette notation peu intéressante d'un point de vue arithmétique. Ce n'est donc pas utilisé en pratique.

Les cas 2 et 3, par contre, sont symétriques par rapport à zéro ce qui permet de faire l'opération $0 - 1$ sans erreur. Tous les nombres sont continus ce qui est souhaitable.

Le cas 3 à la particularité d'avoir le zéro décimal à un nombre différent du zéro binaire mais ce n'est pas un problème en soit. Ce cas est souvent utilisé pour coder l'exposant des nombres en virgule flottante et s'appelle **décalage par excès**. L'excès étant la valeur de décalage de l'échelle qui correspond au zéro décimal, e.g. $1000 = 8$ (excès).

Dans le cas 2 on peut remarquer aussi que le nombre de gauche vaut "0" pour un nombre positif et "1" pour un nombre négatif, c'est pour cela qu'on l'appelle aussi bit de signe. Noter bien que si on avait choisi de coder les nombres de -7 à 8, à la place de -8 à 7, le bit de signe ne serait pas possible. Ceci est du au fait que le zéro n'a pas de signe. En pratique et en grande majorité, on préfère ce dernier et on le nomme **complément à deux**.

Le nom **complément à deux** est une contraction de complément modulo 2. Un nombre binaire en complément à deux se calcule donc par le complément à 2^n du nombre non signé :

$$A_{comp2} = 2^n - |A| \tag{3.1}$$

On part donc du chiffre le plus élevé et on décrémente. Exemple pour A = -1 sur 4 bits, $A_{comp2} = 2^4 - 1 = 15 = 1111$.

Le grand mérite de la notation en complément à deux est le fait qu'on peut additionner sans se préoccuper du signe alors que si on avait choisi une autre notation il aurait fallu utiliser une condition sur le signe pour en tenir compte. Pour vérifier prenons deux nombres m et r sur n bits et effectuons tous les cas de signe :

$$\begin{aligned} +m \equiv m \quad & +r \equiv r \\ -m \equiv 2^n - m \quad & -r \equiv 2^n - r \\ (+m) + (+r) &= m + r \\ (-m) + (+r) &= 2^n - m + r = r - m \\ (+m) + (-r) &= m + 2^n - r = m - r \\ (-m) + (-r) &= 2^n - m + 2^n - r = -m - r \end{aligned} \tag{3.2}$$

Les 2^n n'affectent pas les résultats par arithmétique modulo 2^n. On constate donc que tous les résultats sont corrects quelque soit le signe.

Négation d'un nombre positif De la Table 3.2, on peut constater qu'il suffit d'inverser les bits du nombre positif puis d'y ajouter "1" pour obtenir la négation :

$$-A = \overline{A} - 1 \tag{3.3}$$

C'est une manière plus rapide de calculer le complément modulo 2. Une démonstration algébrique est donnée en annexe.

Extension d'un nombre entier Dans le cas ou on désire passer d'un format de n bits à m bits avec m supérieur à n, on doit remplir les cases vides m-n. Il est trivial que les cases vides d'un nombre positif doivent être des zéros, exemple, 1 sur 4 bits : 0001 et sur 8 bits : 00000001. L'extension des nombres positifs consiste donc à ajouter autant de zéros que nécessaire, mais quand est-il des nombres négatifs en complément à 2 ? Prenons un exemple -1 qui est 1111 sur 4 bits et si on ajoute des zéros cela donne 00001111 = 8 en complément à deux, ça ne marche pas. Pour qu'un nombre négatif reste négatif, il faut que son bit de signe soit "1", on devrait donc mettre 10001111 = -112 ce qui ne joue toujours pas. On peut trouver la réponse en faisant l'opération 0 - 1 sur 8 bits et on trouve 11111111 = -1. On constate donc que pour les nombres négatifs, il faut ajouter des "1" dans les case vides. Une démonstration algébrique vous est proposée en annexe.

3.2.2 Virgule flottante

Pour les nombre réels, on doit coder la virgule et sa position en plus du signe. Ceci complique la représentation car il faut un deuxième nombre, qui est un entier, pour représenter la position de la virgule. On choisit de préférence d'utiliser ce nombre pour représenter un exposant plutôt qu'une position de virgule mais ça reviens au même.

$$X dans \mathbb{R} = signe \cdot m \cdot 2^{exposant} \tag{3.4}$$

Exemple de codage avec 8 bits :

signe	m3	m2	m1	m0	e2	e1	e0

Noter bien que l'exposant peut être positif ou négatif d'où la nécessité de choisir un format pour l'exposant comme le complément à deux ou le décalage par excès (le décalage consiste simplement à soustraire 2^{n-1} pour retrouver une moitié négative).

3.2.3 Virgule fixe

On peut simplifier la représentation des nombre réels si la virgule est fixe, c-à-d si l'exposant ne change pas. Ceci est très utilisé dans les processeurs qui ne possèdent pas d'unité de calcul en virgule flottante.

0	0	0	1 .	0	0	0

Dans l'exemple ci-dessus nous avons 4 bits pour le nombre entier et 4 bits pour le nombre fractionnaire. La virgule n'est pas codée en tant que tel et le nombre complet utilise 8 bits. Seul deux constantes sont nécessaires pour décrire le format en virgule fixe : la longueur de la partie entière et la longueur de la partie fractionnaire :

```
#define FIXEDPT_IBITS   4
#define FIXEDPT_FBITS   4
```

3.3 Opérateurs

Les opérations arithmétiques sont différentes pour chaque format de nombre mais elles peuvent toutes se ramener à un additionneur 1 bit qui se réalise facilement à l'aide d'un porte logique XOR et AND.

A	B	A + B	Retenue
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

TABLE 3.3 – Table de vérité de l'additionneur 1 bit

Le circuit complet d'un additionneur 1 bit doit encore sommer une retenue éventuelle en entrée donc en réalité il additionne 3 bits. Exactement de la même manière que lorsqu'on calcule en colonnes à la main.

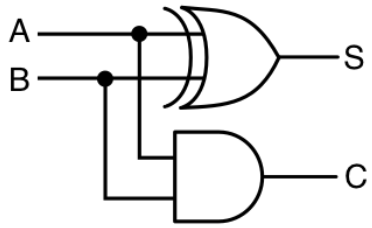

FIGURE 3.1 – Réalisation de l'additionneur 1 bit (sans entrée de retenue)

3.3.1 Addition

Additionner deux nombres entiers positifs est évident mais attention aux débordements ! Par exemple : 4 + 4 = 8, sur 4 bits (n=4) en complément à deux, nous n'avons pas de code pour le 8. Pour comprendre ce qu'il se passe, effectuons l'addition en binaire :

```
  0100  4
+ 0100  4
-------
  1000 -8
```

Selon ce calcul, nous avons donc un résultat de -8 à la place de +8. Il faut donc un mécanisme pour nous avertir quand un débordement se produit (voir chapitre 8).

Nous avons un problème similaire lors de l'addition de deux nombres entiers négatifs, par exemple : -4 + (-5) = -9. Si nous faisons le calcul en représentation en complément à deux sur 4 bits :

```
  1100  -4
+ 1011  -5
-------
  0111  +7
```

On peut constater que le débordement conduit à retourner au nombre le plus élevé de l'autre côté de la gamme. En binaire, le calcul s'effectue, débordement ou pas, et le processeur continue comme si de rien n'était. On appelle ceci, l'arithmétique modulo 2^n. L'opération modulo, par exemple en langage C, retourne le reste de la division entière d'un nombre. Cette opération s'effectue automatiquement à l'aide d'une addition en arithmétique modulo, par exemple en représentation 4 bits non-signée : (8 + 9) modulo 16 = 1. Pour bien comprendre effectuons l'addition en binaire :

```
  1000  8
+ 1001  9
-------
  0001  1
```

Le résultat de l'addition est bien celui du reste de la division entière par 16.

L'addition en virgule fixe reste la même que l'addition pour les entiers mais l'addition en virgule flottante est très différente car il faut ajuster l'exposant.

3.3.2 Soustraction

La soustraction peut se décomposer en un changement de signe et une addition ce qui permet de se ramener aux cas ci-dessus.

$$A - B = A + (-B) \tag{3.5}$$

De plus un seul composant matériel, l'additionneur, est nécessaire pour les deux opérations. La négation ne nécessite qu'une inversion et l'ajout de "1" qui s'effectue aussi à l'aide du même additionneur.

3.3.3 Multiplication

On peut décomposer une multiplication en sommes de produits qui se ramène d'un point de vue matériel à des additions et des décalages. Prenons un exemple 3 x 5 sur 4 bits (n=4) non signé :

```
      0011  3
x     0101  5
-------------
  00000011
  00000000
  00001100
  00000000
-------------
  00001111 15
```

On constate donc qu'il suffit de créer quatre produits intermédiaires qui sont juste des décalages du nombre 3 lorsque le nombre multipliant est non nul. Ensuite on fait la somme des quatre nombres avec un additionneur. Noter bien que si on décale de 4 crans vers la gauche, le nombre intermédiaire à une taille de 8 bits. Les résultats sont donc sur $2n$ bits.

Le cas des nombres négatifs est différent au niveau de l'extension des nombres intermédiaires. En effet, dans notre multiplication ci-dessus nous avons introduit des zéros pour étendre un nombre. Prenons l'exemple suivant (-3) x 2 sur 4 bits signés :

```
      1101 -3
x     0010  2
-------------
  00000000
  11111010
  00000000
  00000000
-------------
  11111010 -6
```

Dans ce cas ci, nous devons étendre le nombre intermédiaire négatif avec des "1" pour obtenir la bonne solution. Le résultat reste -6 quelque soit le nombre de "1" devant le nombre (1010 = -6, 11111010 = -6). Le changement des bits de l'extension nécessite une condition sur le signe qui rend le calcul d'un nombre signé plus long que celui d'un nombre non-signé. Il faut donc une autre technique pour un processeur qui est indépendante du signe.

La multiplication de nombres en virgule fixe est légèrement différente car il faut décaler le résultat de multiplication de la taille de la partie fractionnaire vers la droite :

```
#define FIXEDPT_IBITS   8
#define FIXEDPT_FBITS   8

typedef int16_t fixedpt;

/* Multiplies two fixedpt numbers, returns the result. */
fixedpt fixedpt_mul(fixedpt A, fixedpt B)
{
    return (((fixedptd)A * (fixedptd)B) >> FIXEDPT_FBITS);
}
```

Le but de ce décalage est de replacer la virgule au bon endroit dans le résultat car le nombre de bits de la partie fractionnaire vaut $2n$ et donc il faut le décaler de n à droite.

Réduction de la complexité du calcul de la multiplication En faisant le calcul selon la méthode en colonnes trois problèmes se posent :

1. Le nombre de produits intermédiaires est de n ce qui nécessite n emplacements de mémoire sous forme de registres rapides pour effectuer le calcul à la vitesse maximum.
2. La taille des résultats intermédiaires vaut $2n$ donc il faut des emplacements mémoire doubles.
3. Si une des opérandes est négative alors il faut calculer sans le signe puis effectuer une correction

Le premier problème peut se résoudre en accumulant le résultat intermédiaire au fur et à mesure. Le deuxième en ne mémorisant que les n bits de poids fort. Le dernier problème en utilisant l'**algorithme de Booth** (voir annexe A).

3.3.4 Division

La division entière ou Euclidienne est très simple à implémenter à l'aide de la soustraction successive du diviseur :

```
int divide(N, D) {
    Q = 0;
    while  N >= D {
        N = N - D;
        Q++;
    }
    return Q;           //ou N pour le reste
}
```

La division entière peut donc s'effectuer avec un additionneur, une négation et un test. Le temps d'exécution est par contre relativement long si N»D.

Pour le cas de la virgule fixe, il faut décaler le multiplicande vers la gauche avant division :

```
#define FIXEDPT_IBITS   8
#define FIXEDPT_FBITS   8

typedef int16_t fixedpt;

/* Divides two fixedpt numbers, returns the result. */
static inline fixedpt
fixedpt_div(fixedpt A, fixedpt B)
{
    return (((fixedptd)A << FIXEDPT_FBITS) / (fixedptd)B);
}
```

Comme pour la multiplication, il existe des algorithmes plus efficaces pour faire la division entière comme celui appelé SRT.

3.4 Exercices

Ex 1 Implémenter l'algorithme de Booth en C puis en assembleur.

Ex 2 Écrire en C une fonction de division de deux nombres entiers en complément à 2 qui retourne le résultat en utilisant seulement des additions ou des soustractions. Attention aux signes et à la division par zéro.

Ex 3 Implémenter l'algorithme SRT en C.

Chapitre 4

Interruptions

Les interruptions sont une méthode pour gérer l'exécution de tâches simultanées. Elles se ramènent à la notion de parallélisme dans les processeurs d'ordinateurs. La problématique provient du fait qu'un processeur exécute une tâche à la fois et qu'il est fréquent que plusieurs événements apparaissent simultanément comme par exemple lorsqu'on appuie sur une touche et que le processeur effectue un calcul.

4.1 Interaction entre le processeur et les périphériques

Le processeur exécute des programmes et en même temps il interagit avec les périphériques. Il existe deux possibilités pour effectuer toutes les tâches, le sondage et les interruptions.

4.1.1 Sondage

Le sondage consiste à lire l'état d'un périphériques de manière périodique pour savoir s'il a quelque chose à dire à l'unité centrale. S'il y a plusieurs périphériques, par exemple un clavier et une souris, on les sonde l'un après l'autre. Cette méthode est simple et marche bien pour un nombre réduit de périphériques qui ont des requêtes fréquentes comme une interface de communication. Si ils ont des requêtes peu fréquentes alors on gaspille du temps à leur redemander si quelque chose a changé.

4.1.2 Interruptions

La méthode par interruptions est initiée par le périphérique qui demande "audience" au processeur. Ce dernier peut répondre à son "sujet" quand il lui plaît. Ce procédé fonctionne bien avec des périphériques qui ont des requêtes peu fréquentes telles que la souris ou le clavier.

4.2 Mécanisme de l'interruption

Pour que ça fonctionne, il faut définir un protocole de traitement d'une interruption. Le protocole le plus simple est le suivant :

1. On enregistre la requête sous la forme d'un drapeau ou **flag** qui est levé
2. Lorsque le processeur est libre, il exécute un programme pour traiter l'interruption qu'on appelle : **routine de service d'interruption ou Interrupt Service Routine ou encore ISR**
3. Après l'exécution de l'ISR, le flag est baissé et le processeur retourne à ce qu'il faisait avant.

4.3 Vecteurs d'interruption

Chaque périphérique possède un flag unique et il y a en général une ISR par type de périphérique. Le processeur doit donc associer chaque flag à une ISR. C'est le vecteur d'interruption qui est simplement un pointeur sur l'adresse de l'ISR en mémoire. Ce pointeur est inscrit dans une table qui se situe à un endroit bien précis de la mémoire. Le processeur va donc toujours chercher le vecteur d'interruption dans cette table par un offset de l'adresse de début qui est fixe. Un exemple de table des vecteurs d'interruption est donné ci-dessous :

Dans cet exemple la table des vecteurs est placée à l'adresse FFFF, c'est à dire en haut d'un espace mémoire de 16 bits (65536 adresses). Les deux premières colonnes sont des constantes et la troisième est remplie lors de la compilation.

Adresse Vecteur	Nom Vecteur	Adresse ISR
FFFE	GPIO_VECTOR	43FF
FFFC	TIMER_VECTOR	27F0
FFFA	SPI_VECTOR	3FA4
FFF8	USB_VECTOR	40EF

TABLE 4.1 – Exemple de table de vecteurs d'interruption

4.4 Priorités des interruptions et masques

Lorsque plusieurs interruptions sont en attente, le processeur peut en exécuter certaines en priorité. Le fabriquant du processeur définit une liste de priorités par type d'interruption. Il est possible de modifier la priorité dans le programme utilisateur en utilisant des masques. Le masque est simplement une désactivation de l'interruption momentanée. Certaines interruptions critiques, telles que le reset, ne peuvent pas être masquées. On les nomme "non-maskable interrupts" ou NMI.

Un exemple de masquage est donné ci-dessous où l'on doit impérativement effectuer deux associations sans être interrompu entre deux :

```
void contrôle(void) {
    int temp0, temp1;
    for( ; ; ) {
        __disable_interrupt;            //disable all interrupts
        temp0=temp[0];
        temp1=temp[1];
        __enable_interrupt;             //enable all interrupts
        if (temp0 !=temp1) alarm_on()
    }
}
```

Les affectations temp0 et temp1 seront donc bien effectuées l'une après l'autre donc dans un laps de temps minimum.

4.5 Déclaration d'une interruption en C

L'interruption est similaire à une fonction à part qu'on ne peut pas passer de paramètres ni en retourner. On interagit donc à l'aide de variables globales. L'autre différence est la présence d'un vecteur d'interruption pour référencer notre interruption dans la table des vecteurs. Ce lien peut être effectué à l'aide de la directive # pragma qui permet de placer du code à une adresse mémoire spécifiée pour certains compilateurs.

```
// Timer Interrupt Service Routine
#pragma vector = TIMER_VECTOR
__interrupt void Timer_ISR(void) {

  P1OUT ^= BIT0;                    // P1.0 = toggle (LED)

}
```

L'ISR ci-dessus sera activée par un timer et sa seule fonction est de commuter une LED. Noter bien que la syntaxe d'une interruption dépend du compilateur utilisé.

4.6 Interruptions externes

Les interruptions externes proviennent des périphériques, on les appelle aussi interruptions matérielles à l'opposé des interruptions logicielles qui proviennent du programme lui même.

4.6.1 Interruption depuis un GPIO

Pour une interruption depuis un GPIO, il nous faut un vecteur d'interruption approprié, PORT1_VECTOR par exemple :

```
// Port 1 interrupt service routine
#pragma vector = PORT1_VECTOR
__interrupt void Port_1(void) {

  P1OUT ^= BIT0;                            // P1.0 = toggle
  P1IFG &= ~BIT7;                           // P1.7 IFG cleared

}
```

On constate que le flag est remis à zéro avant de sortir de l'interruption.

4.6.2 Interruption depuis un timer

Pour une interruption depuis un timer, il nous faut un vecteur d'interruption approprié, TIMERA_VECTOR par exemple :

```
// Timer Interrupt Service Routine
#pragma vector = TIMERA_VECTOR
__interrupt void TimerA_ISR(void) {

  P1OUT ^= BIT0;                            // P1.0 = toggle (LED)

}
```

4.6.3 Interruption depuis une interface série

Pour une interruption depuis une interface série, il nous faut un vecteur d'interruption approprié, USCI_A0_VECTOR par exemple dans le cas du MSP430 :

```
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void) {

  switch(__even_in_range(UCA0IV,4)) {
    case  0 :    break;
    case  2 :    while (!(UCA0IFG & UCTXIFG));
                 UCA0TXBUF = UCA0RXBUF;
                 break;
    case  4 :    P2OUT ^= 0x01;            // toggle P2.0 avec exclusive-OR
                 break;
    default :    break;
  }
}
```

Cette interruption est plus compliquée à cause du fait que plusieurs causes peuvent provoquer l'interruption et de ce fait doivent être sélectionnées avec un switch-case (voir Chapitre 10).

4.7 Latence

La latence est le temps total nécessaire pour traiter une requête depuis la levée du flag jusqu'au retour au programme principal. Cela tient compte de plusieurs phénomènes :

- Attente du feu vert du contrôleur d'interruption
- Mise en pause du programme principal (vidage du pipeline)
- Saut à l'adresse de l'ISR (Sauvegarde du contexte)
- Exécution de l'ISR
- Retour à l'adresse du programme principal (Restauration du contexte)

Comme on peut le constater, l'exécution d'une interruption peut être assez longue dépendamment des circonstances. Il faut donc faire attention à ne pas surcharger le processeur avec des interruptions.

Chapitre 5

Timers

Avec le port d'entrée/sortie, le timer est le périphérique le plus indispensable du microcontrôleur. En effet, un CPU est très inefficace dès qu'il s'agit de compter le temps, puisqu'il ne peut rien faire d'autre, sous peine de "perdre du temps".

Un timer est donc indispensable dès qu'il s'agit de développer des applications dans lesquelles des contraintes temporelles doivent être respectées, comme :

- génération de délais ou de temps d'attente
- génération de signaux à caractéristiques temporelles définies
- prises de rendez-vous

Le coeur du timer est un *compteur synchrone*. En général, des fonctionnalités y sont ajoutées, qui permettent d'enrichir les possibilités du timer.

5.1 Principe

Le plus souvent, le timer est construit autour d'un *incrémenteur*. C'est un circuit séquentiel synchrone, construit avec un registre de N bits, et un incrémenteur combinatoire (fig. 5.1). La structure détaillée de ce type de circuit séquentiel est étudiée au cours "Systèmes Logiques".

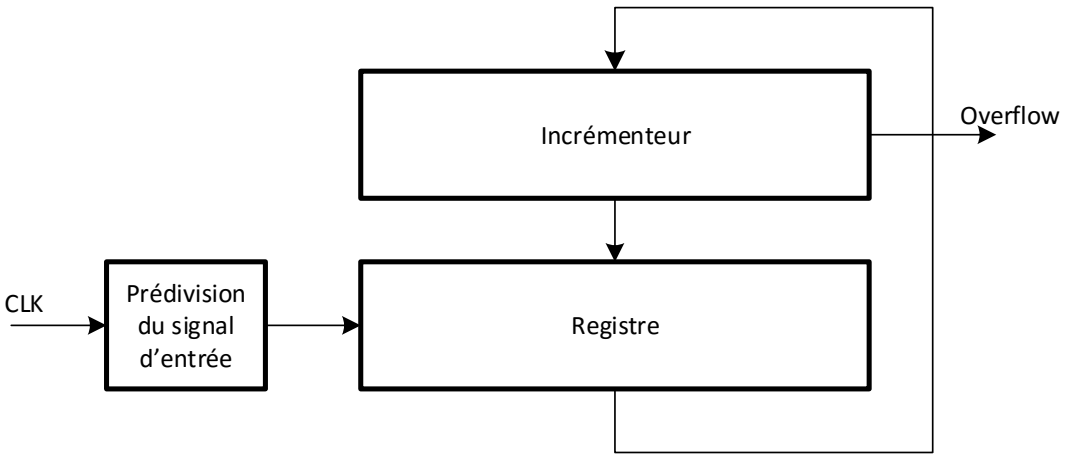

FIGURE 5.1 – Schéma de base d'un timer

L'incrémentation se fait au rythme d'un signal souvent appelé CLK (fig. 5.1), qui est soit :

- périodique, auquel cas le circuit mesure (compte) le temps, et on parle de *timer* ;
- apériodique quelconque, auquel cas le circuit compte simplement les flancs de ce signal tant qu'il est actif et on parle plutôt de *compteur*.

Souvent, un prédiviseur de fréquence permet de réduire la fréquence du signal CLK pour ralentir le rythme du comptage.

La figure 5.2 montre l'évolution de la valeur du registre en fonction du temps, pour un *timer* de 4 bits. La pente de la courbe dépend du rythme auquel le registre est incrémenté, donc de la fréquence du signal CLK. Lorsque le registre atteint sa valeur maximale, égale à $2^N - 1$, il repasse à 0. Durant ce passage à 0, un signal (qui est la retenue sortante du circuit incrémenteur) passe à '1' et émet éventuellement une requête d'interruption.

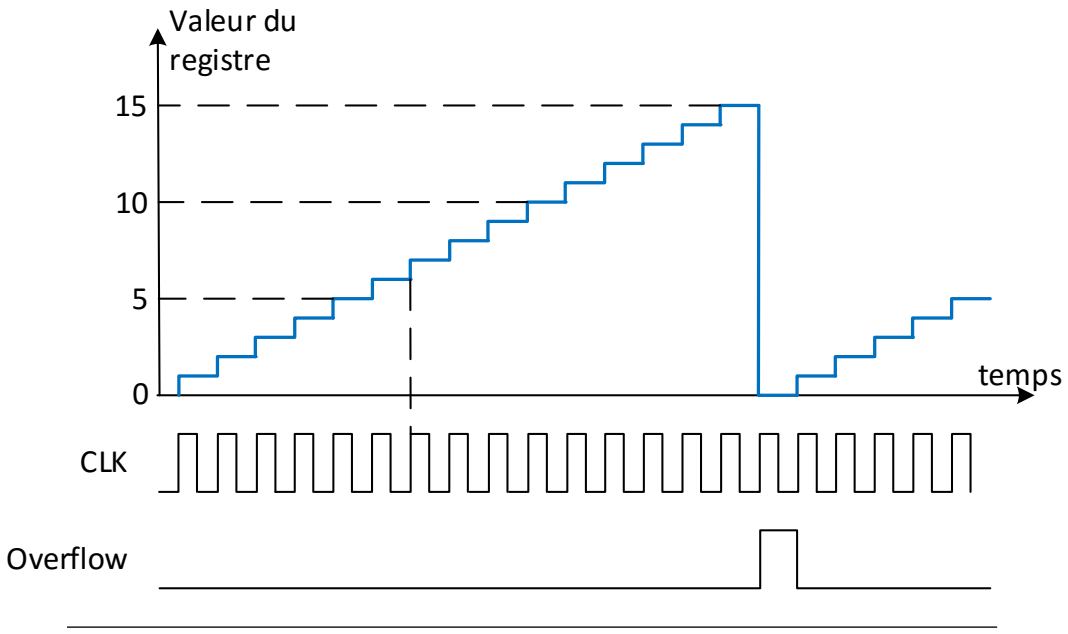

FIGURE 5.2 – Comportement temporel du timer (N=4)

Dans le cas d'un compteur, le signal CLK est apériodique. La figure 5.3 montre l'évolution de la valeur du registre en fonction du temps, pour un *compteur* de 4 bits. De même, la retenue sortante du circuit incrémenteur émet éventuellement une requête d'interruption lors du passage par 0 de la valeur du registre.

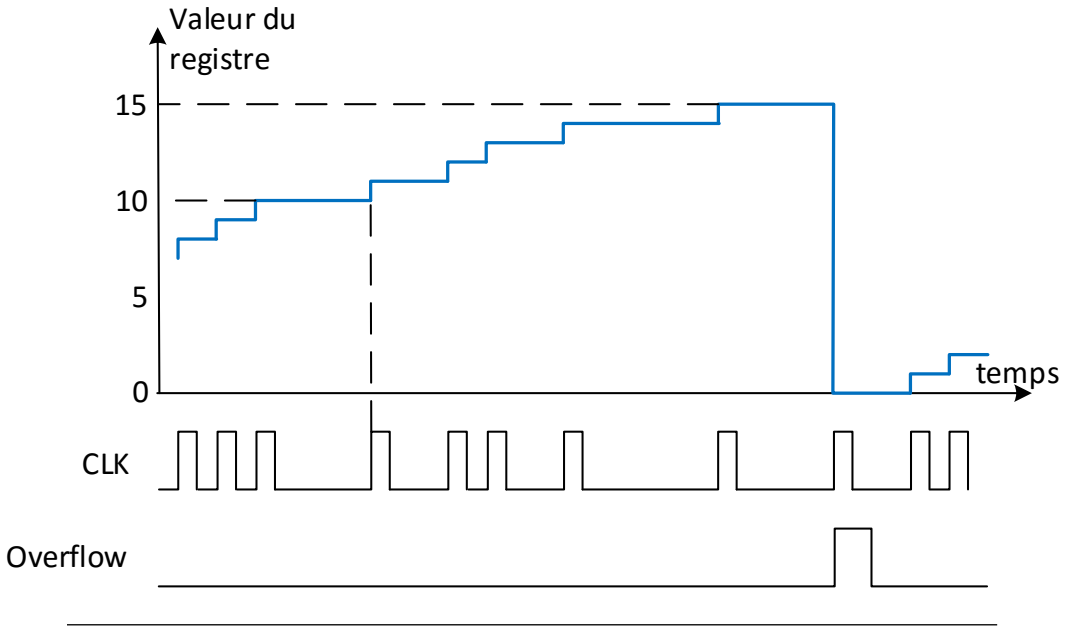

FIGURE 5.3 – Comportement temporel d'un compteur (N=4)

Comme nous le verrons dans la suite, certains timers utilisent un décrémenteur au lieu d'un incrémenteur, voire les deux. Finalement, on peut rencontrer différents types de timers dans un même microcontrôleur, chacun étant adapté à un besoin spécifique.

5.2 Cas du MSP430

Les microcontrôleurs de la famille MSP430 embarquent jusqu'à 5 types de timers différents :

- Real Time Clock, ou *Horloge Temps Réel* (RTC) pour la création d'horloges
- WatchDog Timer ou *Chien de Garde* (WDT) pour la prévention de plantées d'origines diverses
- 3 types de timers à usage général

5.2.1 Real Time Clock

Après initialisation, le RTC maintient à jour les informations horaires les plus courantes :

- année
- numéro du mois dans l'année
- jour du mois
- jour de la semaine
- heure
- minute
- seconde

Ce type de timer nécessite une horloge dont la fréquence est 1Hz, dérivée d'une horloge standard à 32768Hz. Cette dernière fréquence fut introduite aux débuts de la montre à quartz car il est aisé de réaliser des quartz à cette fréquence. On obtient l'horloge à 1Hz par une simple division de 32768 par 2^{15}.

5.2.2 WatchDog Timer

La "plantée" d'un programme est la plupart du temps due à un défaut dans le programme ou à un concours de circonstances qui fait que le programme est bloqué dans un boucle sans fin. Le cas le plus courant est l'attente d'un évènement qui n'arrive pas et qui n'arrivera jamais. Le microcontrôleur ne réagit plus. Le seul moyen est de le *resetter*. Un timer *chien de garde* resette donc le CPU, donc le programme, si celui-ci ne l'a pas remis à 0 avant. Lors du développement, on désactive généralement le chien de garde. Dans le MSP430, ceci est effectué au moyen de l'instruction :

```
WDTCTL = WDTPW +WDTHOLD;
```

En opération, cette instruction est mise en commentaire. Le chien de garde est alors actif, et il est nécessaire d'ajouter des instructions dans les boucles d'attente et à des endroits bien choisis du programme pour le remettre à 0 avant qu'il ne force le programme à redémarrer.

5.3 Timer à usage général : le timer A

Dans le MSP430, le timer à usage général est un périphérique complexe auquel le CPU peut soutraiter des fonctionnalités sophistiquées. En plus des fonctionnalités usuelles, on trouve :

- capture de l'état du registre de comptage lors d'un évènement
- requêtes d'interruption dès que le registre de comptage *est égal à* une constante donnée (comparaison)
- génération de signaux PWM sur une patte externe, sans aucune aide du CPU autre que l'initialisation

Toutes ces fonctionnalités sont contrôlées par le logiciel, en positionnant des champs logiques dans des registres de contrôle. Un champ logique est un groupe de variables logiques dans un registre.

Dans une version de MSP430 donnée, plusieurs Timers A peuvent coexister.

La figure 5.4 est un schéma très simplifié du timer A, illustrant ses 3 principaux sous-blocs. A la partie supérieure, on retrouve le bloc de comptage et son registre appelé TAxR (x est le numéro du timer ; x=0 ou 1 s'il y a deux timers de type A dans le microcontrôleur). TAIFG est le signal d'overflow du compteur. En dessous, le bloc dit de "capture/comparaison" permet de capturer l'état du registre TAxR lors d'un évènement ou de comparer la valeur de TAxR avec une constante contenue dans un registre appelé TAxCCRy. Le bloc de "capture/comparaison" contient un sous-bloc, noté "Output module", qui est chargé de générer un signal PWM à partir du contenu du registre TAxR et de la constante contenue dans le registre TAxCCRy. Comme on le voit sur la figure 5.4, Chaque sous-bloc offre de la fonctionnalité utile.

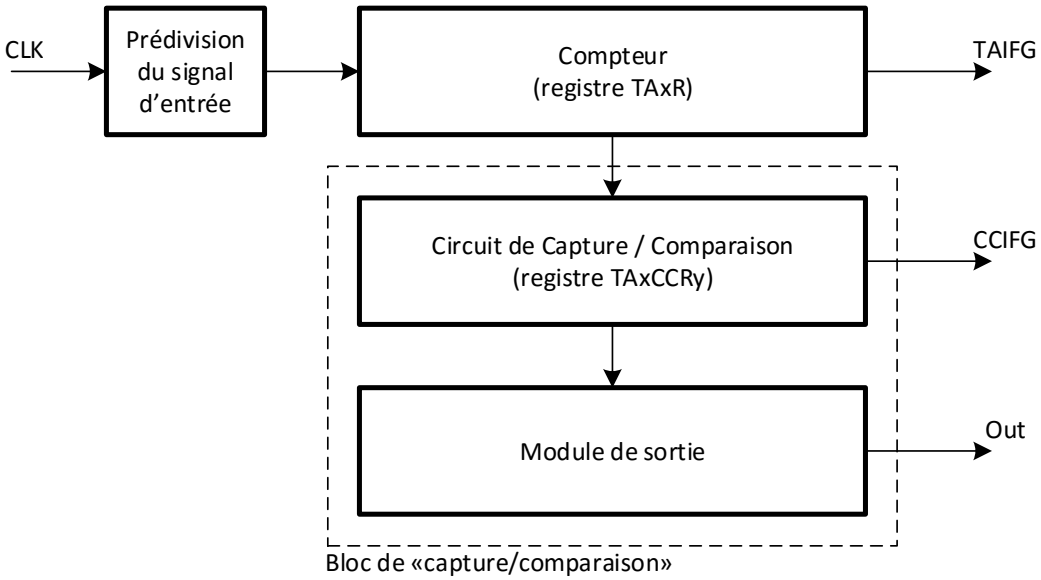

FIGURE 5.4 – Schéma simplifié du timer A

5.3.1 Bloc de comptage

La figure 5.5 illustre en détail le bloc de comptage. On y voit les différents éléments et leurs champs de contrôle, tous inclus dans un registre de contrôle appelé TAxCTL, que nous verrons plus loin.

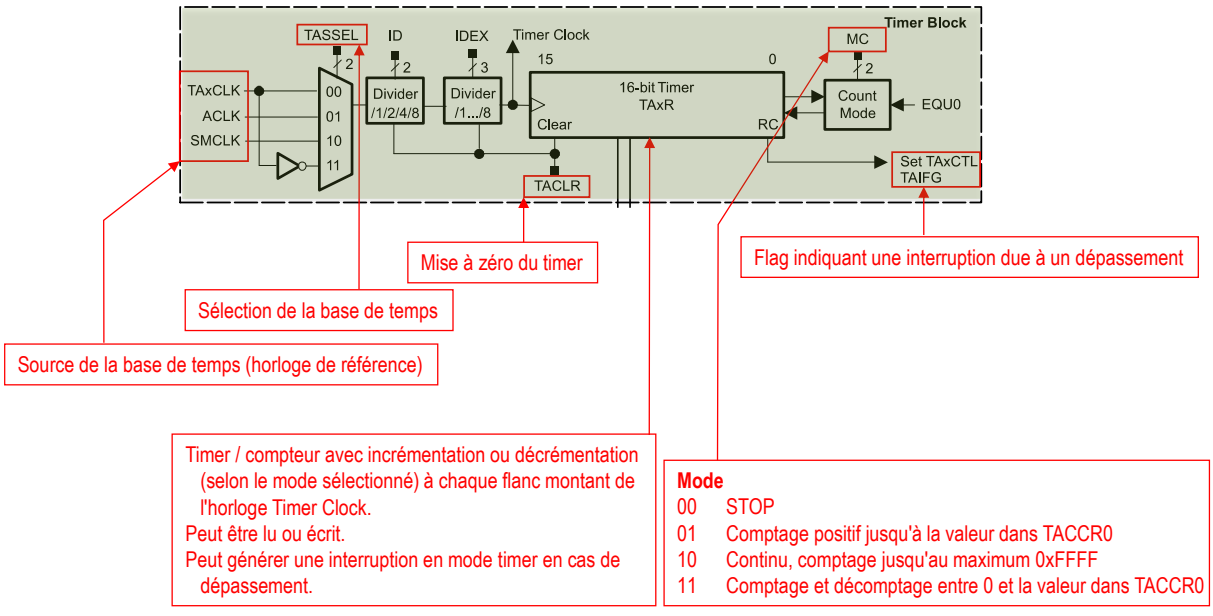

FIGURE 5.5 – Schéma de détail du bloc de comptage

Le registre de comptage TAxR a 3 modes de comptage possibles. Le mode est déterminé par la valeur du champ MCx dans le registre de contrôle TACTLx :

- Mode 1 (MCx = 01) : Comptage jusqu'à la valeur contenue dans le registre TAxCCR0, localisé dans un sous-bloc de capture/comparaison, puis retour à 0 ;
- Mode 2 (MCx = 10) : Comptage jusqu'à 0xFFFF, puis retour à 0 ;
- Mode 3 (MCx = 11) : Comptage jusqu'à la valeur contenue dans le registre TAxCCR0, puis décomptage jusqu'à 0.

Les figures 5.6, 5.7 et 5.8 montrent les chronogrammes associés à chaque mode.

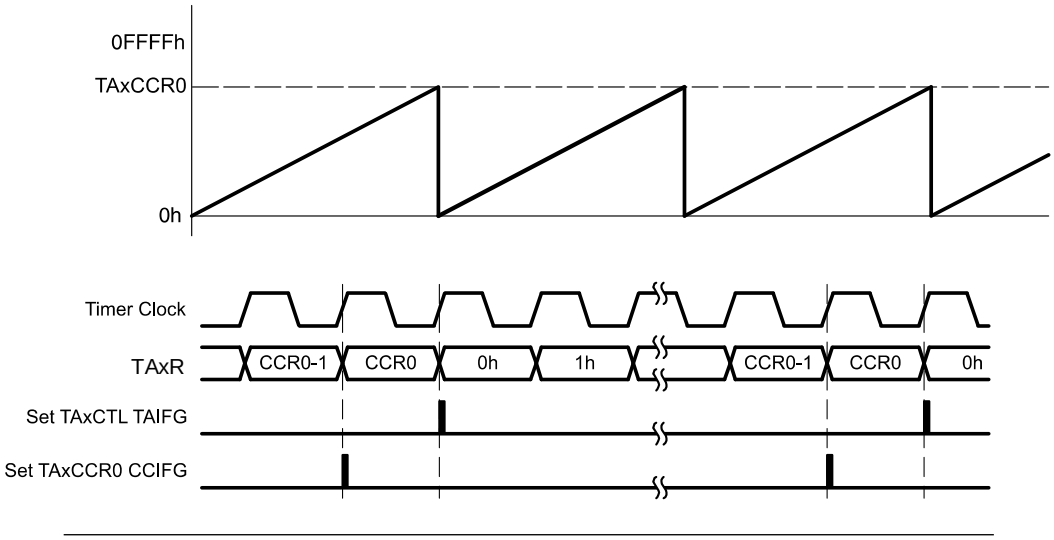

FIGURE 5.6 – Comptage en mode 1 (MCx = 01)

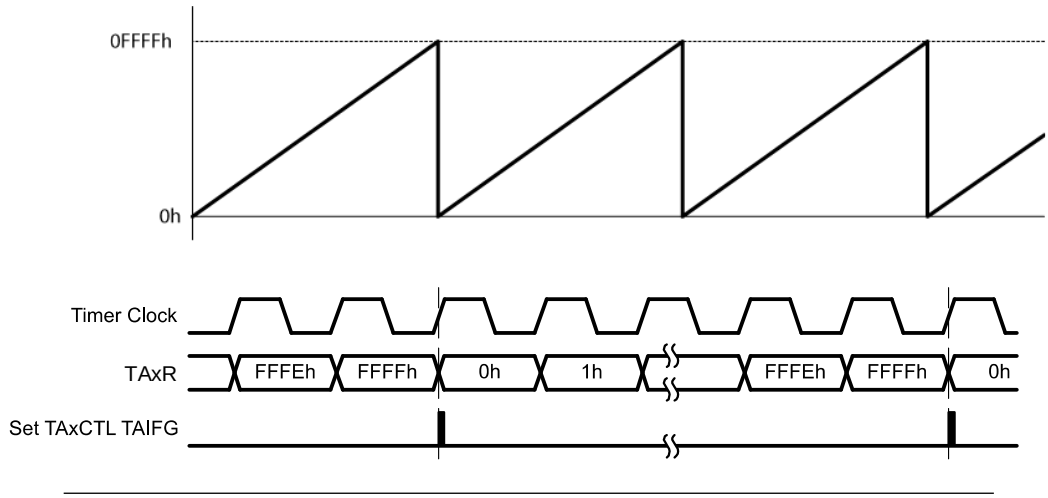

FIGURE 5.7 – Comptage en mode 2 (MCx = 10)

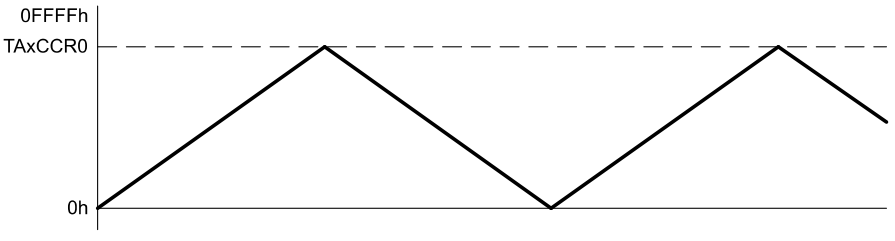

FIGURE 5.8 – Comptage en mode 3 (MCx = 11)

Dans le cas du mode 3, la figure 5.9 précise le comportement des signaux TAIFG et du signal de sortie CCIFG du bloc de capture/comparaison nř0 (celui qui contient le registre TAxCCR0).

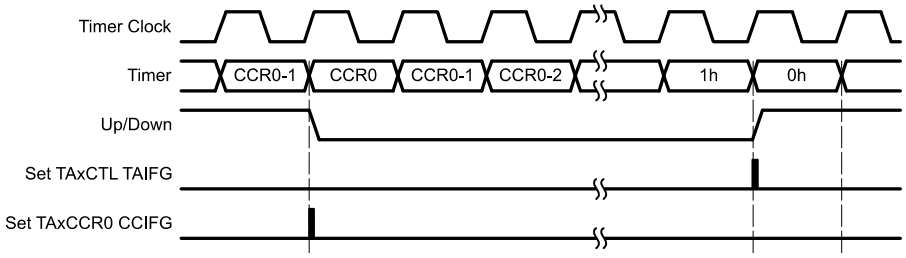

FIGURE 5.9 – Comptage en mode 3 : détail des signaux CCIFG et TAIFG)

Grâce à cette variété des modes de comptage, le timer A va être capable de générer un grand nombre de signaux différents.

5.3.2 Contrôle du bloc de comptage

TAxR : Timer A Register (nř x) ou registre de comptage du timer nř x. Sa valeur évolue entre 0 et 0xFFFF ou le contenu de TAxCCR0.

TAxCTL : Timer A Control (nř x) ou registre de contrôle du comptage du timer nř x. Il permet de spécifier le comportement du bloc de comptage. Il est composé de 5 champs, pour :

- sélectionner l'horloge de référence (champ TASSEL)
- sélectionner le facteur de prédivision de l'horloge de référence (champ ID)
- sélectionner le mode de comptage (champ MC)
- la remise à 0 (reset) du registre de comptage TAxR
- l'autorisation des interruptions (TAIE) issues du registre de comptage. A l'évidence, ces interruptions ne peuvent être générées qu'au moment particulier où le registre de comptage TAxR est à 0, puisqu'on ne connait pas la valeur maximale que peut prendre le registre de comptage TAxR.

Un dernier champ (TAIFG) contient le flag d'état de l'interruption issue du registre de comptage TAxR. Le détail du registre de contrôle TAxCTL est donné à la figure 5.10 et dans le tableau 5.1.

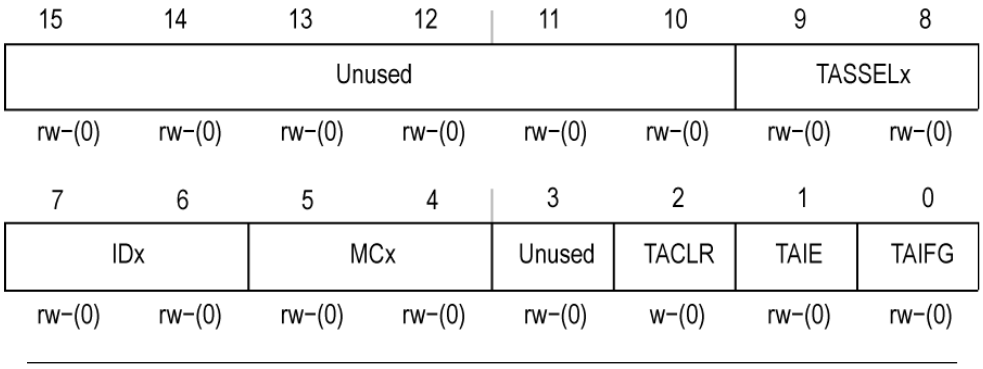

FIGURE 5.10 – Registre TAxCTL

Exemple de configuration

Les instructions ci-dessous configurent le timer A0 en mode comptage de 0 (inclus) jusqu'à 8191 (inclus), soit 8192 cycles, avec l'horloge SMCLK prédivisée par 8.

```
#define TASSEL_2     (2*0x100u)
#define ID_3         (3*0x40u)
#define MC_1         (1*0x10u)

TA0CCR0 = 8191;
TA0CTL  = TASSEL_2 | MC_1 | ID_3;
```

Champ	Valeur	Description
TASSEL		Sélection de l'horloge d'incrémentation
	00	TAxCLK
	01	ACLK
	10	SMCLK
	11	INCLK
ID		Prédivision de l'horloge d'incrémentation
	00	/1
	01	/2
	10	/4
	11	/8
MC		Mode de comptage
	00	Stop. Le timer est arrêté
	01	Mode Up. Le timer compte jusqu'à TAxCCR0
	10	Mode continu. Le timer compte jusqu'à 0xFFFF
	11	Mode Up/Down. Le timer compte jusqu'à TAxCCR0 puis décompte jusqu'à 0
TACLR		Met TAxR à 0. TACLR revient automatiquement à 0
TAIE		Autorisation des requêtes d'interruptions de TAIFG
	0	Interruptions non autorisées
	1	Interruptions autorisées
TAIFG		Indicateur d'interruption
	0	Aucune interruption n'est en attente
	1	Une interruption est en attente de traitement

TABLE 5.1 – Description des champs du registre TAxCTL

5.3.3 Blocs de capture/comparaison

Pour enrichir encore les possibilités offertes par le timer A, celui-ci dispose de plusieurs blocs de "capture/comparaison". Selon le type de microcontrôleur, le timer A dispose de 3, 5 ou 7 de ces blocs. La figure 5.11 illustre plus en détail la structure du timer A, en mettant en évidence uniquement les registres et les signaux de sortie du timer. Comme dit précédemment, il peut y avoir 3, 5 ou 7 blocs de "capture/comparaison". Sur la figure, n vaut donc 2, 4 ou 6 ; x est le numéro du timer.

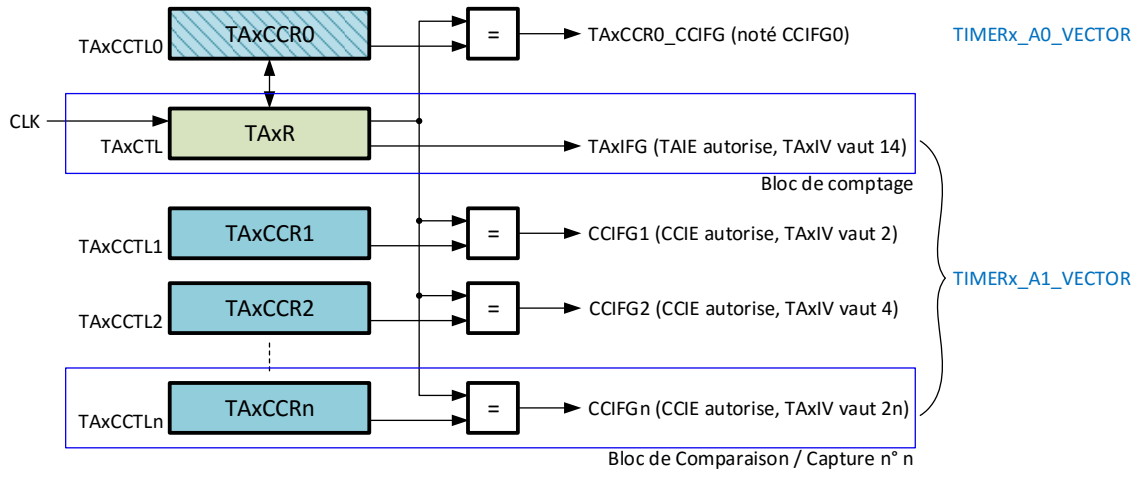

FIGURE 5.11 – Organisation structurelle du timer A

Le timer est ainsi organisé en tranches, chacune capable de générer ses signaux propres. Dans la figure 5.11, chaque rectangle représente un ńregistre de donnéeż, c'est à dire un registre qui contient une information représentant une grandeur. A gauche de chaque ńregistre de donnéeż est noté le nom de son registre de contrôle. Par exemple, à gauche du registre TAxR on retrouve le nom de son registre de contrôle : TAxCTL (TAx ConTroL). Tous ces registres sont sur 16 bits. Grâce à leur module de sortie, les circuits de capture/comparaison peuvent aussi générer automatiquement des signaux particuliers sur une patte du microcontrôleur. Ces fonctionnalités n'apparaissent pas sur la figure 5.11.

5.3.4 Contrôle du bloc de capture/comparaison nř0

Ce bloc fonctionnel est en étroite interaction avec le registre de comptage TAxR, puisqu'il peut (selon la valeur du champ MC) définir la borne supérieure du comptage.

TAxCCR0 : registre de capture/comparaison nř 0 pour le Timer A (nř x). Comme TAxR, TAxCCR0 est un ń registre de donnée ż ; leurs valeurs peuvent être comparées (mode comparaison) ou le contenu de TAxR peut être copié dans TAxCCR0 (mode capture). Lors de ces deux événements (égalité des deux registres ou transfert de TAxR dans TAxCCR0), une requête d'interruption est émise ; c'est le signal appelé TAxCCR0_CCIFG, que nous nommerons CCIFG0. Ce circuit de capture/compare, centré autour du registre TAxCCR0, est contrôlé par le registre de contrôle TAxCCTL0.

TAxCCTL0 : registre de contrôle du circuit de capture/compare nř 0 pour le Timer A (nř x). Il permet de spécifier comment TAxCCR0 se comporte. Il est composé de 6 champs, pour :

- sélectionner la fonction Capture ou Comparaison (CAP)
- sélectionner le mode de capture (sur flanc montant, descendant, etc...) (CM)
- sélectionner le signal de capture (CCIS)
- synchroniser ou non du signal de capture avec l'horloge de comptage (SCS)
- sélectionner le mode de sortie (OUTMOD)
- autoriser ou non les interruptions émises par le bloc de capture comparaison nř 0 (CCIE)

D'autres champs contiennent différents signaux tels que le flag de l'interruption issue du bloc (CCIFG0). La figure 5.12 illustre en détail un bloc de capture/comparaison (nř y). On y voit les différents sous blocs et leurs champs de contrôle, tous éléments du registre de contrôle TAxCCTLy.

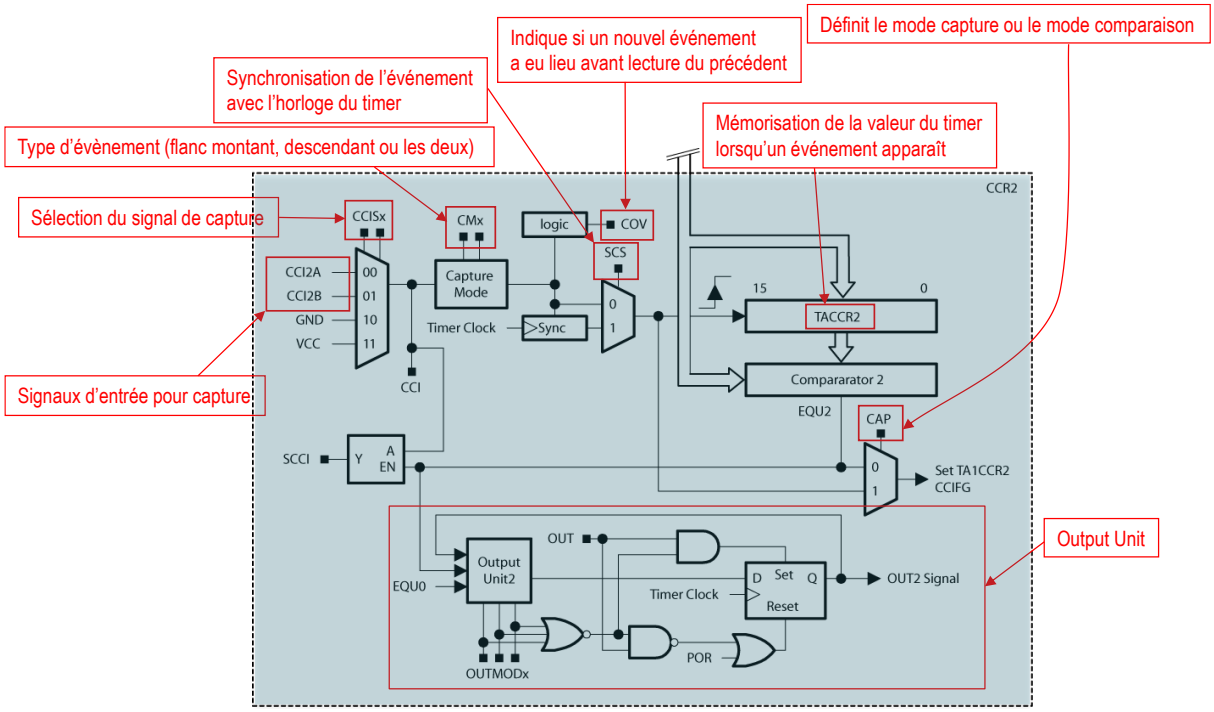

FIGURE 5.12 – Schéma de détail du bloc de capture/comparaison

Le détail du registre de contrôle TAxCCTLy, pour le bloc de capture/comparaison nřy, est donné à la figure 5.13 et dans le tableau 5.2.

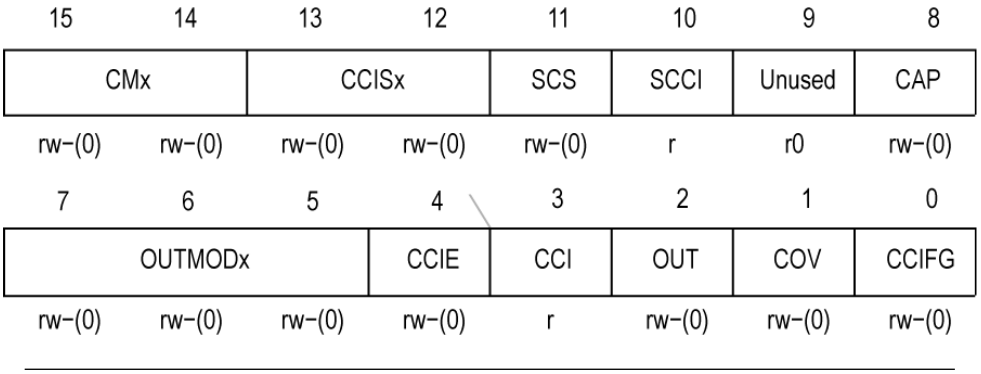

FIGURE 5.13 – Registre TAxCCTLy

Champ	Valeur	Description
CM		Mode de capture
	00	Pas de capture
	01	Capture sur le flanc montant du signal de capture
	10	Capture sur le flanc descendant
	11	Capture sur les deux flancs
CCIS		Sélection du signal de capture
	00	CCIxA
	01	CCIxB
	10	GND
	11	VCC
SCS		Synchronisation du signal de capture
	0	Pas de synchronisation
	1	Synchronisation du signal de capture avec le prochain flanc de CLK
SCCI		Image du signal de capture après synchronisation
CAP		Sélection entre capture et comparaison
	0	Mode comparaison
	1	Mode capture
OUTMOD		Génération du signal de sortie OUT (module de sortie)
		voir chapitre 5.3.8
CCIE		Autorisation des requêtes d'interruptions de CCIFG
	0	Interruptions non autorisées
	1	Interruptions autorisées

TABLE 5.2 – Description des champs du registre TAxCCTL

Exemple de configuration

Les instructions ci-dessous reprennent la configuration vue au chapitre 5.3.2, en autorisant de plus le bloc de capture/comparaison du timer A0 à émettre des requêtes d'interruptions par son signal CCIFG0.

```
#define TASSEL_2     (2*0x100u)
#define ID_3         (3*0x40u)
#define MC_1         (1*0x10u)
#define CCIE         (1*0x10u)

TA0CCR0 = 8191;
TA0CTL  = TASSEL_2 | MC_1 | ID_3;
TA0CCTL0  = CCIE;
```

5.3.5 Contrôle des blocs de capture/comparaison nř1,2,3...

Ces circuits sont des copies conformes du circuit de capture/comparaison nř 0. La seule différence est que leur registre central (TAxCCRy) n'a pas d'influence sur la borne supérieure de TAxR. Ils interagissent toutefois avec TAxR en mode capture. Pour le circuit nř y, le registre de donnée est nommé TAxCCRy, le registre de contrôle est TAxCCTLy. Bien entendu, dans le registre de contrôle TAxCCTLy, les champs portent les même noms (CAP, CM, CCIS, SCS, etc...).

5.3.6 Différence entre Capture et Comparaison

La figure 5.14 illustre le comportement du registre TAxR dans le mode 1 (continuous).

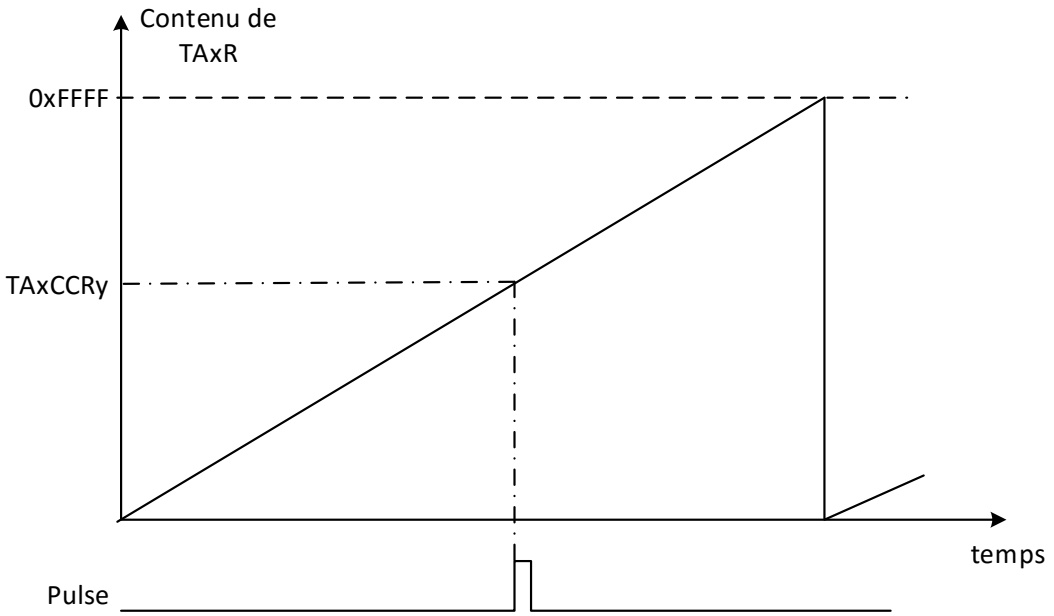

FIGURE 5.14 – Comparaison : TAxR est l'entrée / Capture : Pulse est l'entrée

Une fois le processus de comptage lancé, TAxR évolue en dent de scie, comme illustré. Le circuit de capture/comparaison utilise ce signal pour son opération.

En mode ń comparaison ż, le contenu du registre TAxCCRy est une entrée. Dès que TAxR atteint TAxCCRy, le signal noté ń Pulse ż est généré, qui peut déclencher l'exécution d'une routine de service d'interruption. En mode ń comparaison ż, le signal appelé ń Pulse ż dans la figure 5.14 est en fait CCIFG.

En mode ń capture ż, le contenu du registre TAxCCRy est une sortie. Un signal ń Pulse ż doit être fourni, qui déclenche la copie de TAxR dans TAxCCRy au moment où la pulse apparaît. En fait, c'est le flanc actif de la pulse qui déclenche le processus de capture, qui peut donc se produire sur le flanc montant, descendant ou les deux, du signal ń pulse ż. Typiquement, le signal ń pulse ż est une entrée du microcontrôleur. Au moment où la capture est exécutée, une requête d'interruption peut être émise.

5.3.7 Interruptions du timer A

Un timer A contenant n bloc de capture/comparaison peut générer $n + 1$ requêtes d'interruptions. Toutefois, seuls deux vecteurs d'interruption sont définis. Revenant à la figure 5.11, on voit que le signal CCIFG0 issu du bloc de capture/comparaison nř0 est associé à lui tout seul à un des vecteurs (TIMERx_A0_VECTOR). Tous les autres signaux d'interruption (TAxIFG, CCIFG1, CCIFG2, ...) sont associés à l'autre vecteur (TIMERx_A1_VECTOR).

Si une interruption provient d'un des signaux TAIFG, CCIFG1 ou CCIFG2, il est donc nécessaire de tester lequel de ces signaux est effectivement responsable de la requête. L'information y relative est disponible dans registre particulier TAIV, appelé abusivement "vecteur d'interruption". Il est ainsi possible de déterminer rapidement la source de l'interruption, en testant la valeur de TAIV au moyen d'une instruction *SWITCH...CASE*.

La description du registre TAIV est donnée à la table 5.3 :

Champ	Valeur	Source de l'interruption
TAIV	0x00	pas d'interruption en attente
	0x02	CCIFG1
	0x04	CCIFG2
	0x06	CCIFG3
	0x08	CCIFG4
	0x0A	CCIFG5
	0x0C	CCIFG6
	0x0E	TAIFG

TABLE 5.3 – Description du registre TAIV

Exemple de programme avec interruption

Le code ci-dessous configure le timer A en mode continu (jusqu'à 0xFFFF). Le signal TAIFG s'active lors du passage de TAxR à 0, et émet une requête d'interruption. Dans la routine d'interruption, la patte nř1 du port P5 change d'état.

```
#include <msp430.h>
void main(void)
{
   P5DIR |= 0x02;
   TACTL = TASSEL_2 + MC_2 + TAIE;
   __enable_interrupts();

   while(1);
}

// Timer_A Interrupt Vector (TAIV) handler
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A (void)
{
   switch(TAIV)
   {
      case  2 : break;
      case  4 : break;
      case 14 : P5OUT ^= 0x02;
                break;
   }
}
```

5.3.8 Module de sortie

Dans chaque bloc de capture/comparaison, le module de sortie est capable de générer plusieurs types de signaux sur sa sortie OUT. Il est possible de connecter ce signal OUT vers une patte du microcontrôleur, en configurant le port correspondant au moyen du registre PxSEL. Il est nécessaire de consulter la fiche technique du microcontrôleur utilisé pour savoir quelles pattes peuvent être connectées à quels modules de sortie de timer.

La génération du signal OUT est basée sur deux signaux EQU0 et EQUy internes aux circuits de capture/comparaison nř 0 et y :

- EQU0 s'active quand TAxR = TAxCCR0
- EQUy s'active quand TAxR = TAxCCRy

Elle dépend aussi du mode de comptage dans lequel se trouve le bloc de comptage.

Les figures 5.15, 5.16 et 5.17 montrent l'évoluetion du signal de sortie OUT en fonction du mode de comptage, pour chaque valeur du champ OUTMOD (registre TAxCCTLy).

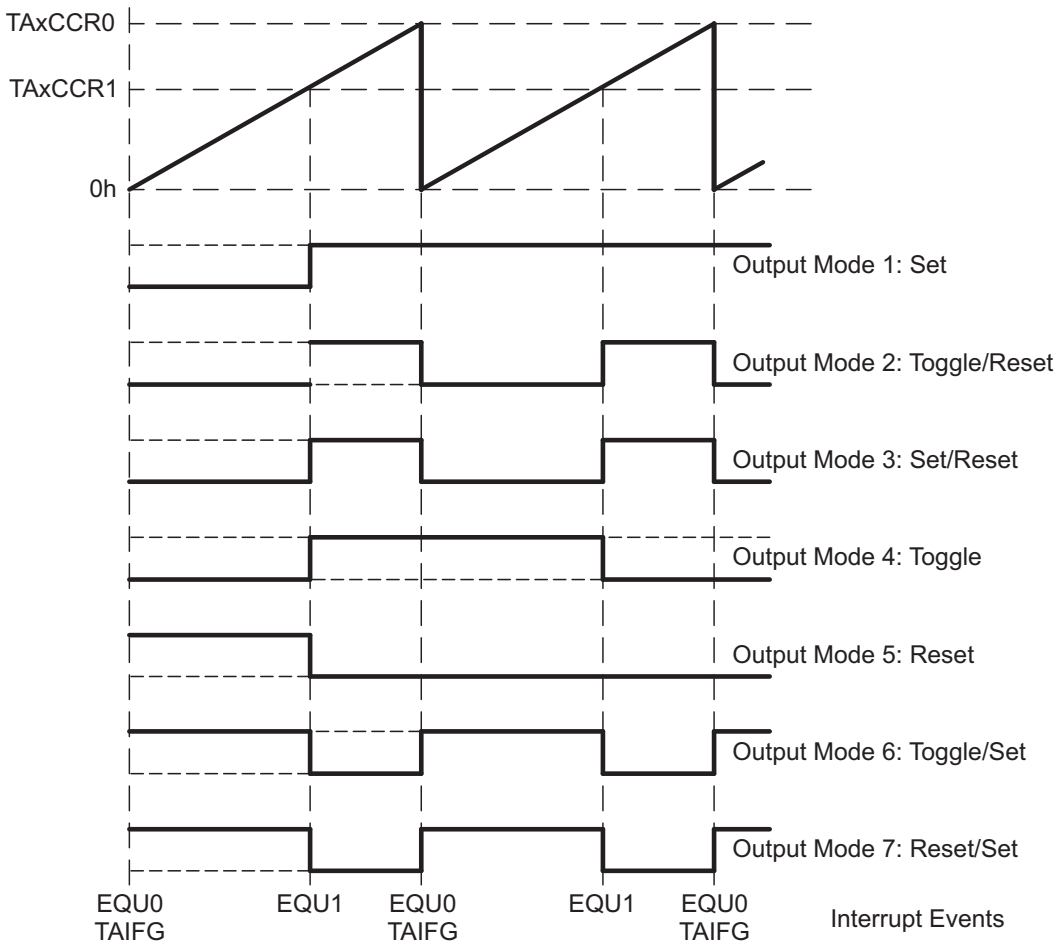

FIGURE 5.15 – Exemple de sortie - TimerA en mode Up (MC = 01)

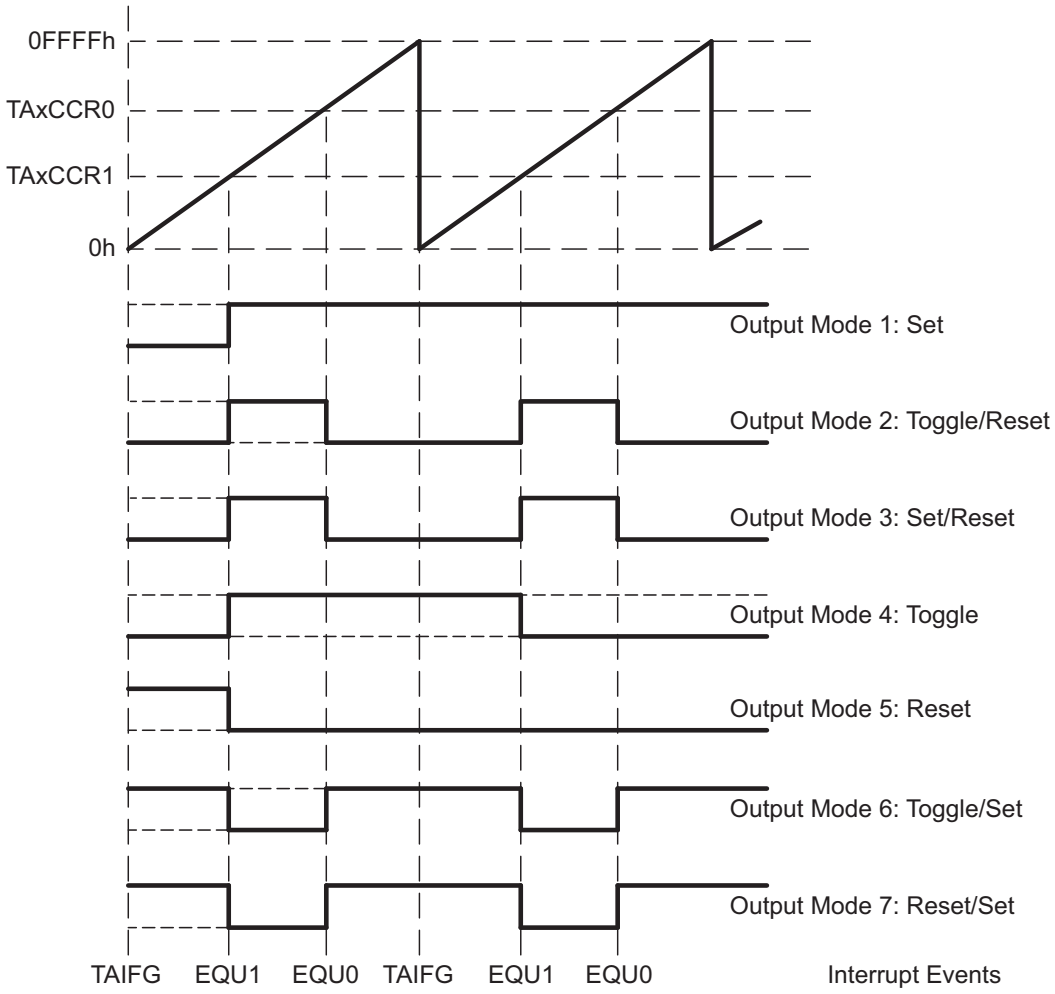

FIGURE 5.16 – Exemple de sortie - TimerA en mode Continu (MC = 10)

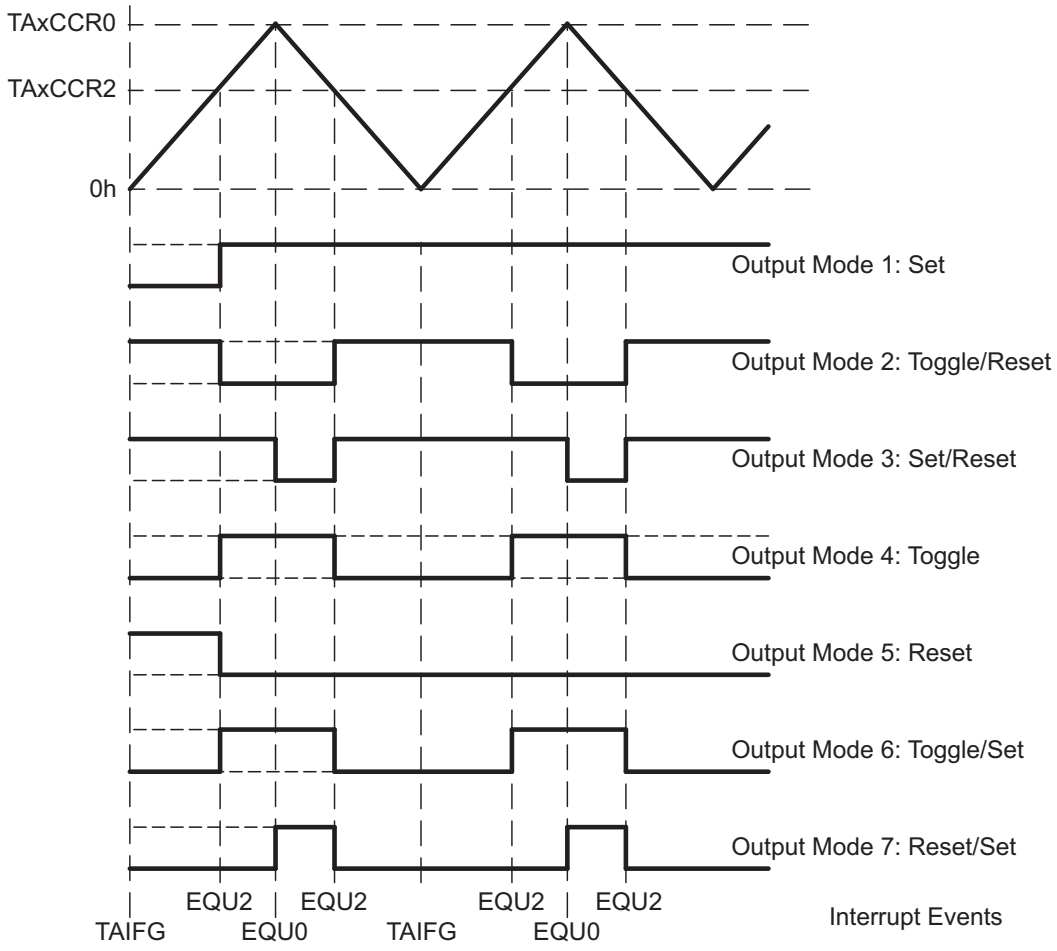

FIGURE 5.17 – Exemple de sortie - TimerA en mode Up/Down (MC = 11)

Chapitre 6

Système d'horloges

Comme tout système numérique, le microcontrôleur et ses périphériques ont besoin d'une horloge. Ce signal contrôle toutes les bascules composant les différentes machines séquentielles qui composent le système numérique. La structure détaillée de ce type de circuit séquentiel est étudiée au cours "Systèmes Logiques".

Le signal d'horloge est toujours généré par un oscillateur, dont les performances définissent les qualités :

- fréquence ;
- précision ;
- stabilité.

6.1 Oscillateurs

On distingue 4 types d'oscillateur. Ceux-ci se différencient par le mécanisme donnant naissance à l'oscillation et par les composants externes mis en oeuvre :

- oscillateurs RC (consulter le cours de "Electronique Analogique") ;
- oscillateurs à résonateur céramique ou à quartz ;
- circuits oscillateurs spécifiques, éventuellement calibrés en usine et à compensation des effets de température ;
- circuits calés sur des signaux externes tels que GPS, dont la précision est obtenue par des horloges atomiques.

Dans les microcontrôleurs, on rencontre les deux premiers types. Très souvent, un multiplicateur de fréquence permet de générer une horloge à plus haute fréquence que ce que les oscillateurs produisent.

6.1.1 Oscillateurs RC

L'oscillateur RC le plus simple est probablement celui illustré à la figure 6.1. Il est basé sur un inverseur logique à hystérèse ou *Trigger de Schmitt*, une résistance et une capacité.

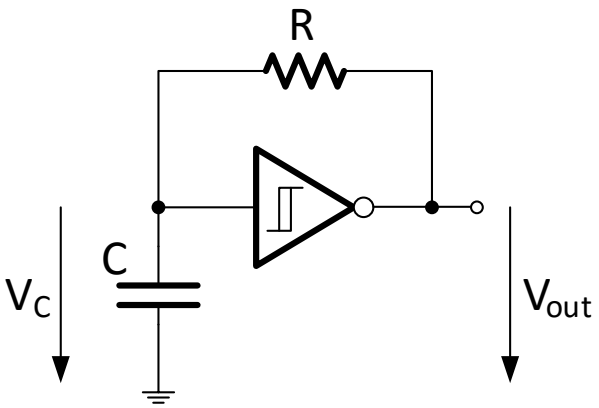

FIGURE 6.1 – Oscillateur RC à Trigger de Schmitt

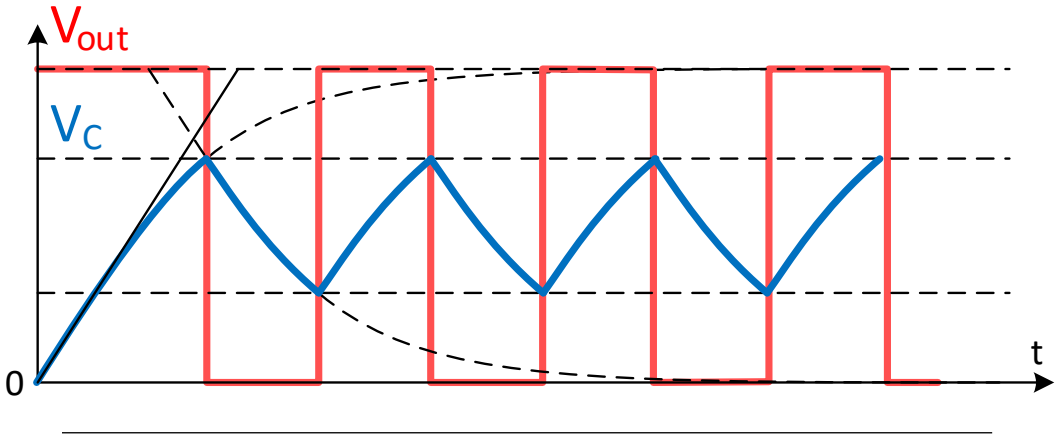

FIGURE 6.2 – Signal de l'oscillateur RC

La figure 6.2 illustre les signaux à l'entrée et à la sortie de l'inverseur à hystérèse.

Du fait de la faible précision de la valeur des composants, tant la résistance, la capacité que les seuils d'hystérèse, la fréquence de l'oscillation est peu précise et peu stable en température. De plus, elle dépend de la tension d'alimentation. Par contre, tous les composants peuvent être intégrés. C'est pourquoi on rencontre ce type d'oscillateur dans la plupart des microcontrôleurs modernes, pour toutes les applications qui ne nécessitent pas une fréquence précise.

6.1.2 Oscillateurs à quartz

Si la fréquence doit être précise, l'oscillateur à quartz est le circuit le plus couramment utilisé. Son fonctionnement est basé sur l'utilisation d'un cristal piézoélectrique, dans lequel l'énergie est stockée soit sous forme mécanique soit sous forme électrique. L'oscillation consiste en un échange alternatif entre l'énergie mécanique et l'énergie électrique. Le circuit oscillateur entretient l'oscillation, en compensant les pertes d'énergie qui sont très faibles. Le schéma typique d'un oscillateur à quartz est donné à la figure 6.3.

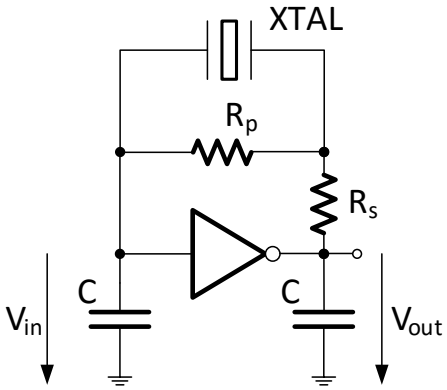

FIGURE 6.3 – Schéma de l'oscillateur à quartz

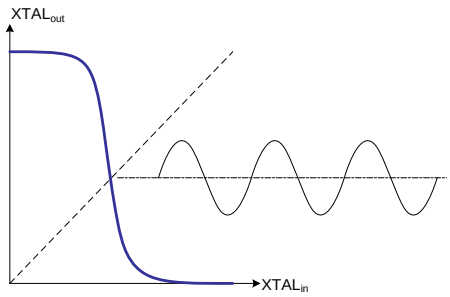

FIGURE 6.4 – Polarisation de l'inverseur

Dans ce circuit, la résistance *Rp* a une grande valeur et ne sert qu'à polariser l'inverseur logique à son point de repos donnant le gain maximal. Ainsi connecté, l'inverseur maintien le résonateur en oscillation (figure 6.4). Le signal de sortie de l'inverseur est ensuite transformé en un signal carré qui constitue l'horloge de base pour le reste du circuit.

Une variante du résonateur à quartz est le résonateur céramique. Son fonctionnement est similaire à celui du quartz. Il se justifie pour son coût moindre, au prix de performances moindres.

6.1.3 Critères de performance des oscillateurs

Un oscillateur est caractérisé par :

- sa fréquence ;
- la précision de la fréquence ;

- la stabilité instantanée de la fréquence, qui est liée au facteur de qualité ;
- la stabilité de la fréquence dans le temps.

Fréquence

Dans un oscillateur RC, la fréquence est peu précise car elle dépend de composants dont les valeurs sont peu précises. Dans un oscillateur à résonateur, la fréquence est essentiellement déterminée par le résonateur. Sa fréquence de résonance a une erreur relative, spécifiée en *ppm*, ou *part par million*. Un cristal de qualité moyenne a une erreur de fréquence de 100 ppm, soit 0,001%.

Facteur de qualité

Le facteur de qualité du signal est défini par $Q = \frac{f_c}{BW}$. Si Q est grand, la bande passante est petite par rapport à la fréquence centrale de l'oscillation et donc la stabilité en fréquence est grande.

Stabilité

La stabilité en fréquence d'un résonateur est aussi donnée en ppm. La fréquence suit les dérives des composants. On distingue trois types de dérives :

- le vieillissement des composants, de l'ordre de 5 à 50 ppm ;
- l'effet des variations de température, de l'ordre de 20 à 200 ppm ;
- l'effet des variations d'humidité, en général inférieure à 10 ppm.

Ces trois effets sont liés à la nature mécanique du résonateur. Le vieillissement peut être associé au relâchement de tensions mécaniques internes au cours du temps. La température modifie les dimensions physiques du résonateur à cause de la dilatation. L'humidité diffuse dans le matériau lui-même.

6.2 Multiplicateur de fréquence

Un multiplicateur de fréquence contient deux fonctions de base :

- génération "libre" d'un signal à une fréquence proche de la fréquence visée ;
- asservissement de la fréquence obtenue à une fréquence de référence

6.2.1 Principe de fonctionnement

Boucle ouverte

Dans la version la plus simple du multiplicateur de fréquence, le facteur de multiplication est un nombre entier. Le signal de sortie est généré "librement" par un oscillateur contrôlé en tension (VCO, ou *Voltage Controlled Oscillator*). Ce contrôle est nécessaire pour permettre l'asservissement. Le signal généré ayant une fréquence multiple de la fréquence de référence, il est divisé par un facteur N pour pouvoir être comparé avec cette dernière (figure 6.5).

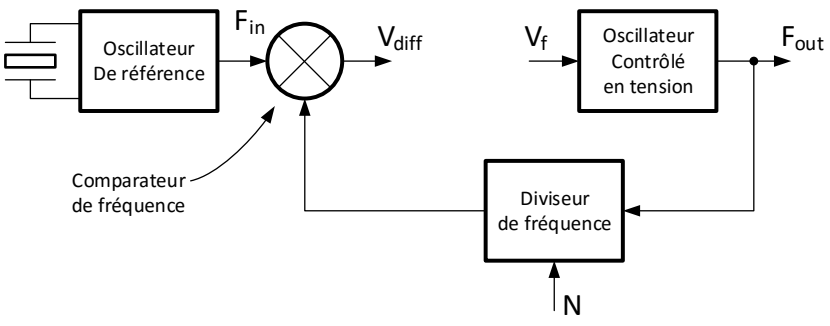

FIGURE 6.5 – Multiplicateur de fréquence - Boucle ouverte

Boucle fermée

Le comparateur de fréquence produit une tension proportionnelle à la différence des fréquences. Si la fréquence de référence est supérieure (resp. inférieure) à la fréquence de sortie divisée par N, alors la tension appliquée au VCO doit être réduite (resp. augmentée). Pour que cette tension soit stable, un filtre passe-bas est inséré entre la sortie du comparateur de fréquence et l'entrée du VCO (figure 6.6).
La fréquence du signal de sortie est : $F_{out} = N.F_{in}$

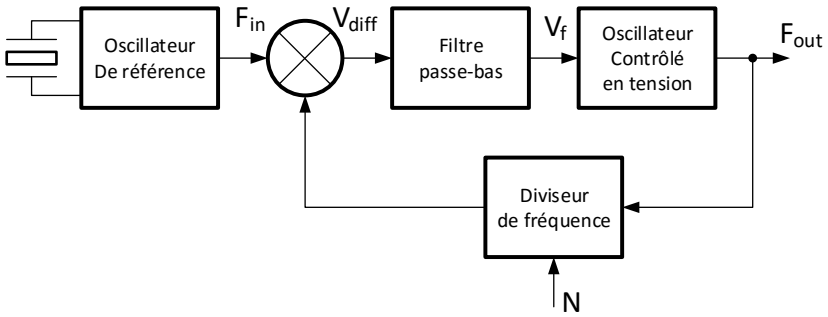

FIGURE 6.6 – Multiplicateur de fréquence - Boucle fermée

Version embarquée

Pour permettre un facteur de multiplication non-entier, il suffit d'ajouter un second diviseur de fréquence avant le comparateur de fréquence (figure 6.7).

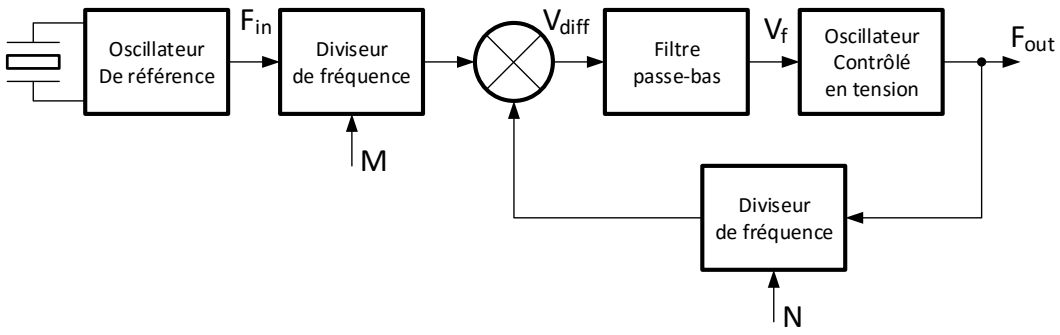

FIGURE 6.7 – Multiplicateur de fréquence à facteur fractionnaire

La boucle de régulation tend à obtenir : $\frac{F_{in}}{M} = \frac{F_{out}}{N}$

La fréquence du signal de sortie est donc : $F_{out} = \frac{N}{M}.F_{in}$

En pratique, le comparateur de fréquence et le filtre passe-bas sont réalisés au moyen d'un intégrateur numérique, qui consiste en un compteur/décompteur. Durant un intervalle de temps défini, celui-ci s'incrémente sur les impulsions du signal de référence et se décrémente sur les impulsions du signal de sortie. A la fin de l'intervalle de temps, on sait si le VCO doit augmenter ou réduire la fréquence du signal de sortie. Une conséquence importante est que la fréquence moyenne du signal de sortie est stable, mais la fréquence instantanée varie en dents de scie.

6.3 Cas du MSP430

La figure 6.8 est un schéma synoptique du circuit d'horloge, appelé UCS pour *Unified Clock System*. Ceci est valable pour les microcontrôleurs de la famille MSP430F5xxx.
Cinq sources de fréquence sont disponibles :

- VLOCLK : oscillateur interne basse fréquence à très basse consommation, dont la fréquence est d'environ 10 kHz
- REFOCLK : oscillateur interne calibré à environ 32768 Hz ;
- XT1CLK : oscillateur à large gamme de fréquence, pouvant être connecté à un résonateur à quartz ou céramique 32768 Hz, ou une source externe de 4 MHz à 32 MHz ;
- XT2CLK : oscillateur haute fréquence (en option), pouvant être connecté à un résonateur à quartz ou céramique ou une source externe ;
- DCOCLK : horloge obtenue par multiplication de l'une des 4 sources précédentes.

A partir de ces 5 sources, qui peuvent être activées ou désactivées chacune individuellement, le circuit UCS produit 3 signaux d'horloge pour le CPU et les différents périphériques du microcontrôleur :

- ACLK (Auxiliary Clock) est plutôt réservé aux périphériques à basse consommation ou se contentant d'une horloge à basse fréquence, comme les timers. Toutefois, ACLK peut être connecté à n'importe laquelle des 5 sources de fréquence.
- SMCLK (Subsystem Master Clock) est disponible pour les périphériques ;
- MCLK (Master Clock) est l'horloge du CPU ;

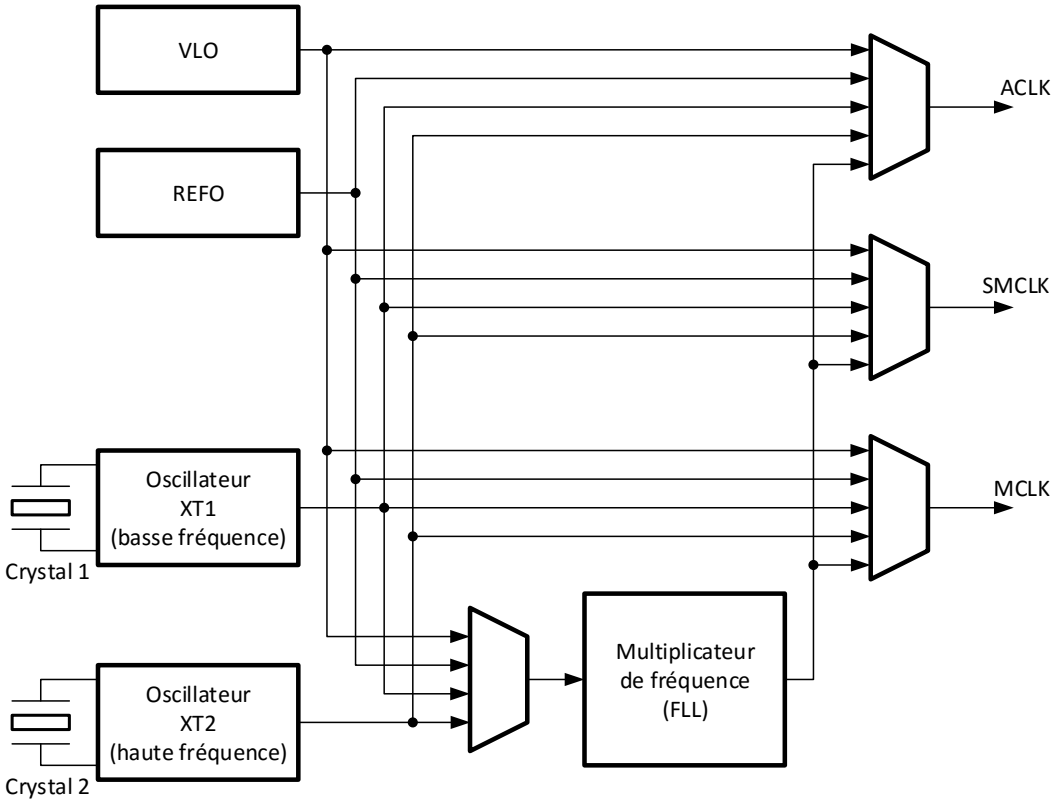

FIGURE 6.8 – Schéma synoptique du circuit d'horloge du MSP430

6.3.1 Frequency Locked Loop

Le multiplicateur de fréquence est nommé *Frequency Locked Loop*. Son schéma est donné à la figure 6.9. On retrouve les éléments vus au chapitre **??** :

- 10-bit Frequency integrator : le comparateur de fréquence

- DCO : le VCO
- Prescaler et Divider/(N+1) : le diviseur de F_{out}
- Divider (/1/2/4/8/12/16) : le diviseur de F_{ref}

La fréquence du signal de sortie DCOCLK est donnée par :

$$F_{DCOCLK} = F_{FLLREFCLK} \cdot \frac{(FLLN+1).(FLLD)}{FLLREFDIV}$$

L'élément *DC Generator* génère un courant de polarisation pour le VCO. Il permet aussi de sélectionner la gamme de fréquences que peut générer le VCO. Ces fréquences sont comprises dans un intervalle de quelques dizaines de MHz, mais le VCO ne peut pas toutes les générer simultanément. La gamme de fréquence est donc déterminée par le *DC Generator*. Les différentes gammes dépendent du modèle de microcontrôleur.

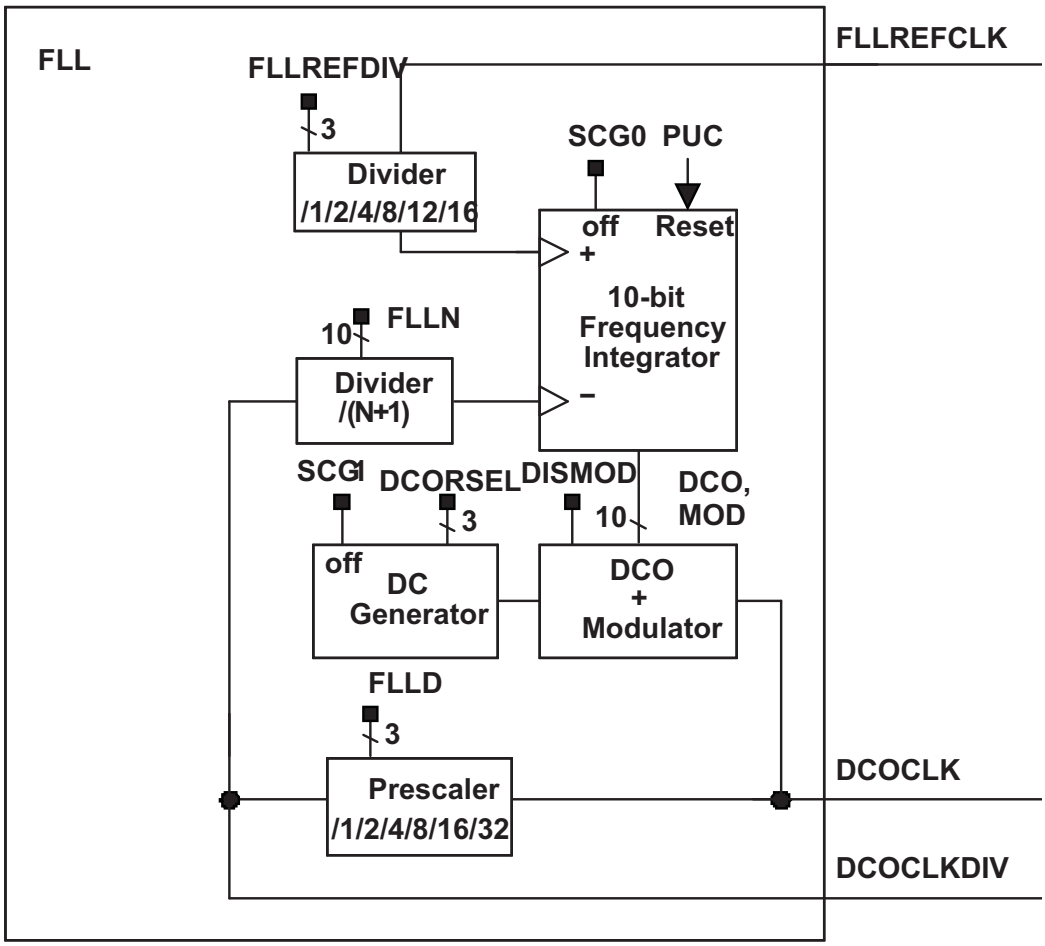

FIGURE 6.9 – Schéma du multiplicateur de fréquence

A titre d'exemple, la figure 6.10 donne les gammes de fréquence que le MSP430F5529 peut générer. Si le paramètre DCORSEL (DCO Range Select) est mal configuré, la fréquence visée sera trop excentrée dans la gamme sélectionnée, voire hors de la gamme, et la boucle de régulation de la FLL ne pourra faire son travail correctement.

Exemple

On souhaite générer une fréquence de 3,2768 MHz à partir d'une horloge de 32768 Hz. On peut choisir DCORSEL = 2,3 ou 4. La valeur 2 est la meilleure car la gamme correspondante est la mieux centrée autour de 3,2768 MHz. Le facteur de multiplication vaut 100. Il peut être obtenu avec :

- $FLLREFDIV = 1$;
- $FLLD = 2$;
- $FLLN = 49$.

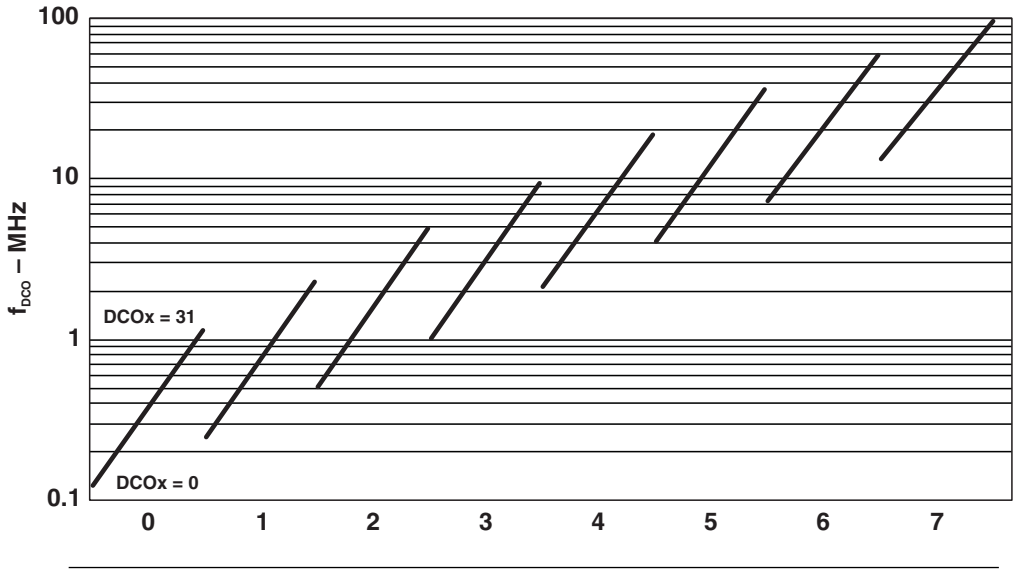

FIGURE 6.10 – Gammes de fréquence du MSP430F5529

6.3.2 Configuration du circuit UCS

Le schéma détaillé du circuit UCS est donné à la figure 6.11. En plus des 5 sources de fréquence principales décrites précédemment, une 6^{eme} source est visible : il s'agit de la sortie du *prescaler*, elle est notée DCOCLKDIV. De plus, 2 autres signaux d'horloge sont visibles, en plus de ACLK, MCLK et SMCLK. Ce sont ACLK/n, qui est dérivée de ACLK par une division de fréquence et disponible sur une patte externe du microcontrôleur, et MODCLK qui sert pour certains périphériques spéciaux dont le convertisseur AD. Finalement, les horloges ACLK, MCLK et SMCLK peuvent subir une postdivision de fréquence par 1, 2, 4, 8, 12, 16, 32.

La configuration du circuit UCS demande donc de spécifier :

- la connection des 3 horloges (ACL, MCLK, SMCLK) avec les 6 sources de fréquence, y compris DCOCLKDIV. Ceci se fait par des champs appelés SELx dans les divers registres de configuration.
- le facteur de prédivision des horloges ACLK, MCLK et SMCLK. Ceci se fait par des champs appelés DIVx dans les divers registres de configuration.
- la configuration de la FLL si elle est utilisée. Ceci comprend :
  - Détermination des facteurs FLLREFDIV, FLLN et FLLD, de façon à approcher au mieux la fréquence visée, à partir de la fréquence de référence ;
  - Détermination du facteur DCORSEL.

6.3.3 Registres de contrôle du circuit UCS

Ils sont nommés UCSCTLx et sont au nombre de 10. En principe, on ne touche pas à UCSCTL0, qui contient des grandeurs internes au comparateur de fréquence. Les plus couramment utilisés sont :

- UCSCTL1 : contient DCORSEL (figure 6.12 et table 6.1) ;
- UCSCTL2 et UCSCTL3 : configuration du multiplieur de fréquence FLL (figures 6.13 et 6.14 et tables 6.2 et 6.3) ;
- UCSCTL4 : connections de ACLK, MCLK et SMCLK (figure 6.15 et table 6.4) ;
- UCSCTL5 : facteurs de prédivision de ACLK, MCLK et SMCLK (figure 6.16 et table 6.5) ;

Les autres registres permettent de faire des réglages fins sur les oscillateurs XT1 et XT2.

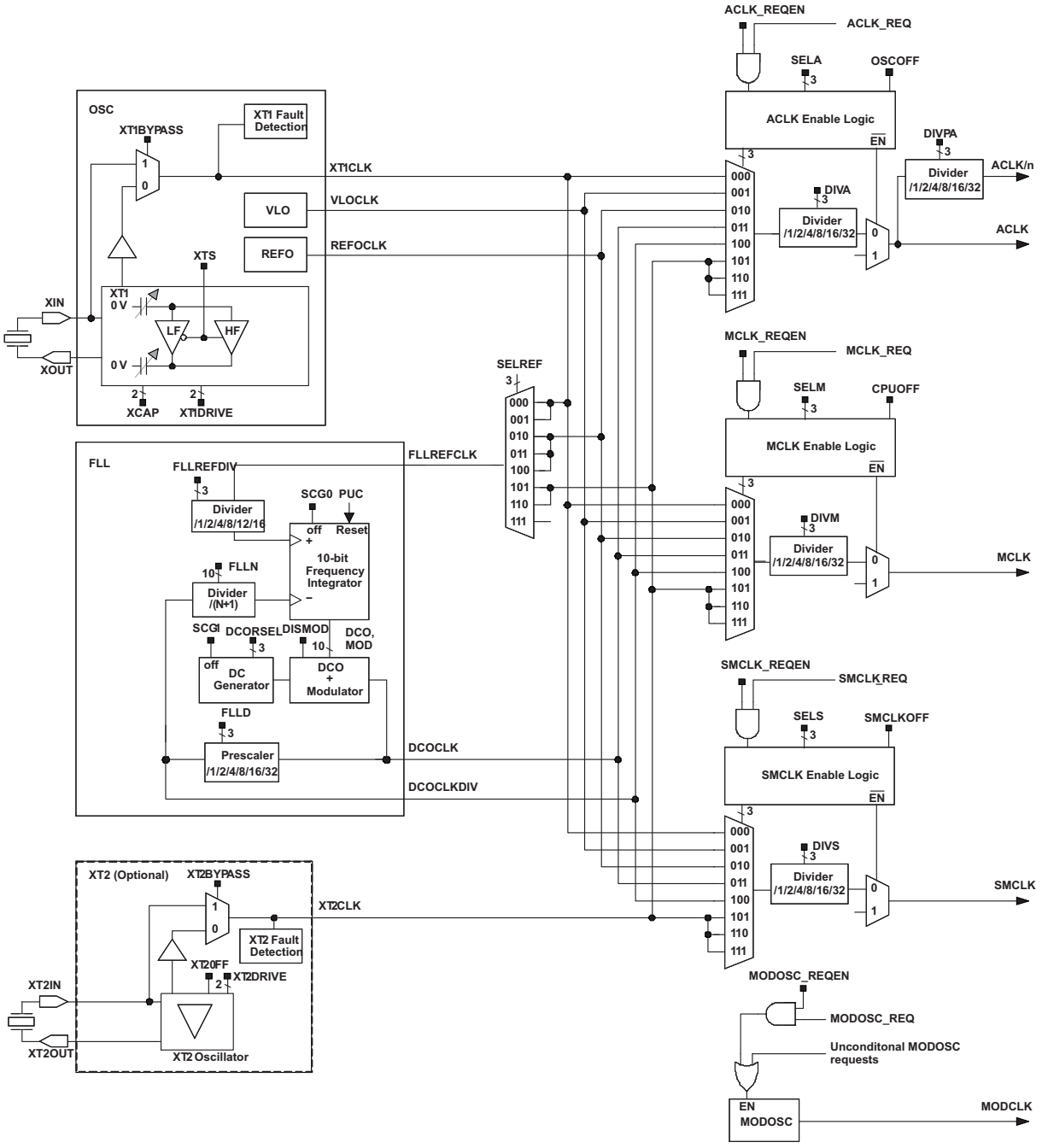

FIGURE 6.11 – Schéma complet du circuit UCS

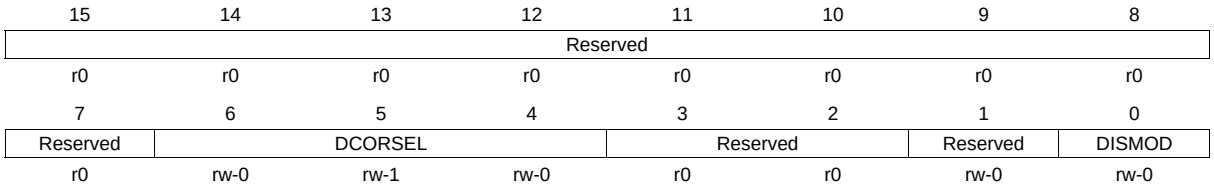

FIGURE 6.12 – UCSCTL1

Champ	Valeur	Description
DCORSEL		Sélection de la gamme (voir figure 6.10)
DISMOD	-	Fonction liée au comportement interne du comparateur de fréquence

TABLE 6.1 – UCSCTL1

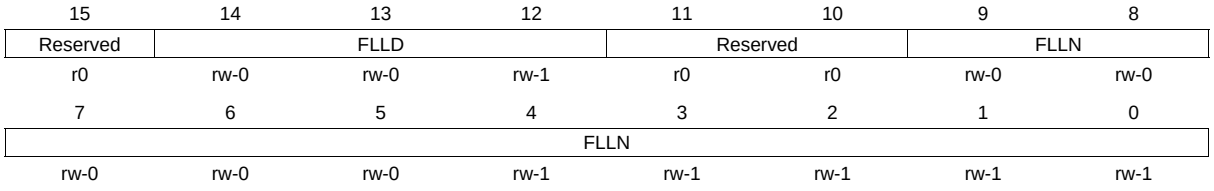

FIGURE 6.13 – UCSCTL2

Champ	Valeur	Description
FLLD		Facteur de multiplication ($F_{DCOCLK} = F_{FLLREFCLK} . \frac{(FLLN+1).(FLLD)}{FLLREFDIV}$)
	000	FLLD = 1
	001	FLLD = 2
	010	FLLD = 4
	011	FLLD = 8
	100	FLLD = 16
	101	FLLD = 32
	110	réservé
	111	réservé
FLLN		Facteur de multiplication (voir chapitre Frequency Locked Loop

TABLE 6.2 – UCSCTL2

15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0

7	6	5	4	3	2	1	0
Reserved	SELREF			Reserved	FLLREFDIV		
r0	rw-0	rw-0	rw-0	r0	rw-0	rw-0	rw-0

FIGURE 6.14 – UCSCTL3

Champ	Valeur	Description
SELREF		Sélection de la source de référence
	000	XT1CLK
	001	réservé
	010	REFOCLK
	011	réservé
	100	réservé
	101	XT2CLK
	110	réservé
	111	réservé
FLLREFDIV		Prédivision de la référence ($F_{DCOCLK} = F_{FLLREFCLK} . \frac{(FLLN+1).(FLLD)}{FLLREFDIV}$)
	000	FLLREFDIV = 1
	001	FLLREFDIV = 2
	010	FLLREFDIV = 4
	011	FLLREFDIV = 8
	100	FLLREFDIV = 12
	101	FLLREFDIV = 16
	110	réservé
	111	réservé

TABLE 6.3 – UCSCTL3

15	14	13	12	11	10	9	8
Reserved					SELA		
r0	r0	r0	r0	r0	rw-0	rw-0	rw-0
7	6	5	4	3	2	1	0
Reserved	SELS			Reserved	SELM		
r0	rw-1	rw-0	rw-0	r0	rw-1	rw-0	rw-0

FIGURE 6.15 – UCSCTL4

Champ	Valeur	Description
SELA		Connection de ACLK
	000	XT1CLK
	001	VLOCLK
	010	REFOCLK
	011	DCOCLK
	100	DCOCLKDIV
	101	XT2CLK
	110	réservé
	111	réservé
SELS		Connection de SMCLK
	000	XT1CLK
	001	VLOCLK
	010	REFOCLK
	011	DCOCLK
	100	DCOCLKDIV
	101	XT2CLK
	110	réservé
	111	réservé
SELM		Connection de MCLK
	000	XT1CLK
	001	VLOCLK
	010	REFOCLK
	011	DCOCLK
	100	DCOCLKDIV
	101	XT2CLK
	110	réservé
	111	réservé

TABLE 6.4 – UCSCTL4

15	14	13	12	11	10	9	8
Reserved	DIVPA			Reserved	DIVA		
r0	rw-0	rw-0	rw-0	r0	rw-0	rw-0	rw-0
7	6	5	4	3	2	1	0
Reserved	DIVS			Reserved	DIVM		
r0	rw-0	rw-0	rw-0	r0	rw-0	rw-0	rw-0

FIGURE 6.16 – UCSCTL5

Champ	Valeur	Description	Champ	Valeur	Description
DIVPA		génération de ACLK/n	DIVS		Postdivision de SMCLK
	000	n = 1		000	f_{SMCLK}
	001	n = 2		001	$f_{SMCLK}/2$
	010	n = 4		010	$f_{SMCLK}/4$
	011	n = 8		011	$f_{SMCLK}/8$
	100	n = 16		100	$f_{SMCLK}/16$
	101	n = 32		101	$f_{SMCLK}/32$
	110	réservé		110	réservé
	111	réservé		111	réservé
DIVA		Postdivision de ACLK	DIVM		Postdivision de MCLK
	000	f_{ACLK}		000	f_{MCLK}
	001	$f_{ACLK}/2$		001	$f_{MCLK}/2$
	010	$f_{ACLK}/4$		010	$f_{MCLK}/4$
	011	$f_{ACLK}/8$		011	$f_{MCLK}/8$
	100	$f_{ACLK}/16$		100	$f_{MCLK}/16$
	101	$f_{ACLK}/32$		101	$f_{MCLK}/32$
	110	réservé		110	réservé
	111	réservé		111	réservé

TABLE 6.5 – UCSCTL5

6.3.4 Exemple de configuration

Partant d'un quartz 32768 Hz connecté sur l'oscillateur XT1, le code suivant génère :

- ACLK à 327680 Hz
- MCLK à 10,48576 MHz

Le signal ACLK est utilisé par le timer A0 pour générer des interruptions toutes les 100 ms.

```
#include <msp430.h>
int main(void) {

    // configuration du systeme d'horloge UCS
    UCSCTL1 &= ~(BIT6 | BIT5 | BIT4); // reset DCORSEL
    UCSCTL1 |= DCORSEL_5;             // range du DCO

    UCSCTL2 = 9 | FLLD__32;           // multiplication par 10 et par 32

    // XT1CLK entrée de FLL, prédivision par 1
    UCSCTL3 = SELREF__XT1CLK + FLLREFDIV__1;

    // sélection des sources pour ACLK (FXT1*10) et MCLK(FXT1*320)
    UCSCTL4 = SELA__DCOCLKDIV + SELM__DCOCLK;

    UCSCTL5 = DIVA__1;                // division de ACLK par 1

    UCSCTL6 &= ~XT1OFF;               // activation de XT1 pour ACLK

// configuration du timer A0
// génère une interruption sur CCIFG0 tous les 32768 tops
    TA0CCTL0 = CCIE;
    TA0CCR0 = 32767;
    TA0CTL = TASSEL_1 + MC_1 + TACLR;
    __enable_interrupt();
    while(1);
}

// Timer_A CCIFG0 handler
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
    todo();
}
```

Chapitre 7

Assembleur

On appelle *langage machine* le langage natif du CPU, c'est à dire le langage qui peut être directement *interprété* par le processeur. Les instructions sont des mots binaires organisés en champs, qui commandent chacun des actions spécifiques telles que :

- le type d'opération à effectuer
- les opérandes
- le mode d'adressage

Le langage machine est donc intrinsèquement lié au CPU auquel il s'applique,

Le langage d'assemblage (ou langage assembleur ou simplement assembleur par abus de langage, abrégé ASM) est une version lisible du langage machine. Il consiste à représenter les codes binaires des instructions par des symboles appelés mnémoniques (du grec mnêmonikos, relatif à la mémoire), c'est-à-dire faciles à retenir.

Par exemple, l'unité de contrôle d'un processeur particulier reconnaît l'instruction en langage machine suivante :

```
01000000 00110101 00000000 01010101 b (en binaire)
4035 0055 h (en hexadécimal)
```

On voit bien qu'il est impossible de comprendre directement ce que peut faire cette instruction. En langage assembleur, cette instruction est traduite par un équivalent plus facile à comprendre pour le programmeur :

```
MOV.W #85,R5
```

qui signifie "mettre la valeur décimale 85" (0x55 en hexadécimal) dans le registre R5.

7.1 Jeu d'instructions

Le jeu d'instructions est l'ensemble de toutes les instructions que le CPU est capable d'exécuter. On peut en général les regrouper en plusieurs catégories :

- instructions de transfert de données ; elles consistent à copier le contenu d'un registre ou d'une case mémoire vers un autre registre ou une autre cas mémoire.
- opérations arithmétiques ;
- opérations logiques ;
- sauts et rupture de séquence contrôlés ou non par un test, qui permettent (par exemple) de faire des appels de sous-programme.

En plus de cette classification, les instructions peuvent être à un ou deux opérandes. Dans les machines simples, jusqu'à 16-bits, les instructions à deux opérandes sont en général de la forme :

```
OP SRC DST
```

Le résultat de l'opération écrase l'un des opérandes.

Le jeu d'instruction d'un CPU peut être plus ou moins complet selon la complexité du CPU. Par exemple, celui de la famille MCS-51 (famille de microcontrôleurs Intel 8051 et 8052) comprend environ 110 instructions, alors que le MSP430 a "seulement" 27 instructions, tout en offrant des performances comparables. Une raison est liée au *mode d'adressage*, qui définit la façon dont le CPU accède aux données qu'il s'apprête à traiter, que ce soit pour un simple transfert de données ou une opération arithmétique sophistiquée.

7.2 Modes d'adressage

Selon que l'opérande d'une instruction est une constante, une variable, un pointeur ou autre, il est évident que cet opérande n'est pas identifié de la même manière. Reprenant l'instruction prise comme exemple dans l'introduction :

```
MOV.W #85,R5
```

la valeur "85", qui était codée par "0055" pourrait représenter :

- le nombre (la constante) 0x55 (ou 85 en décimal) ;
- la case mémoire numéro 0x55 ;
- si c'est la case mémoire numéro 0x55, est-ce son contenu qui est l'opérande ou autre chose ?
- etc...

Même si les modes d'adressage varient grandement d'un CPU à un autre, on peut en identifier quelques uns qui leurs sont communs. Ce sont :

- l'adressage immédiat ;
- l'adressage direct ;
- l'adressage indirect ;
- l'adressage relatif ;
- l'adressage indexé.

D'autres modes d'adressage sont imaginables selon le type de machine.

7.2.1 Adressage immédiat

En adressage *immédiat*, la donnée est identifiée explicitement. Dans l'exemple déjà vu

```
MOV.W #85,R5
```

le symbole # spécifie que la valeur est à prendre telle quelle. L'adressage immédiat est donc associé à une constante.

7.2.2 Adressage direct

En adressage *direct*, parfois aussi appelé *absolu*, l'identifiant est l'adresse de la case contenant la donnée. L'exemple devient

```
MOV.W 85,R5
```

C'est l'adressage "par défaut", le plus courant, spécifié par le symbole & ou par l'absence de symbole. L'adressage direct est donc associé à une variable. De fait, la case mémoire ou le registre identifié est une variable.

7.2.3 Adressage indirect

En adressage *indirect*, l'identifiant est l'adresse d'une case mémoire, qui contient l'adresse de la donnée visée . L'exemple devient

```
MOV.W @85,R5
```

Cet adressage est souvent spécifié par le symbole @. L'adressage indirect permet donc d'implémenter un pointeur.

7.2.4 Adressage relatif

Le terme "adressage" est ici abusif dans la mesure où l'adressage relatif est utilisé pour faire des sauts et branchements. L'adresse est ici celle d'une instruction. L'adressage relatif spécifie l'adresse de la prochaine instruction sous forme d'un saut par rapport à la position de l'instruction en cours d'exécution. En pratique, il n'y a pas à se soucier du calcul du déplacement pour faire le saut au bon endroit ; c'est le logiciel de développement qui se charge de calculer la bonne valeur. Un exemple d'instruction avec adressage relatif est :

```
JMP Label
```

`Label` est une *étiquette* permettant de repérer la prochaine instruction, par exemple :

```
Label   MOV.W @85,R5
```

L'exécution de l'instruction `JMP Label` fait que la prochaine instruction est `MOV.W @85,R5` quel que soit l'endroit du programme où se trouve cette dernière.

7.2.5 Adressage indexé

L'adressage *indexé* permet de gérer des tableaux de façon efficace. Il est le plus souvent associé à de l'adressage direct ou indirect. L'index permet d'accéder à plusieurs opérandes successifs à partir d'une position de départ.

7.2.6 Adressage par registre

Comme nous le verrons dans le chapitre suivant, le coeur du CPU contient des registres à usage général et à accès rapide, permettant de minimiser le temps d'exécution de certaines instructions. Ces registres sont identifiés explicitement, par un nom symbolique (par exemple A, ACC, R0, R1, R2, etc...). Parfois appelé adressage *implicite* ou *inhérent*, le mode d'adressage par registre utilise donc explicitement le nom du registre impliqué. Par exemple :

```
MOV R1,0x40
```

est une instruction du jeu d'instruction MCS-51 (microcontrôleurs de la famille 8051) et qui consiste à copier le contenu de la case d'adresse 0x40 (adressage direct pour la source de la donnée) dans le registre appelé R1. Bien que R1 pourrait être identifié par son adresse, il l'est ici par son nom explicite. L'intérêt est de pouvoir représenter le code de l'instruction sur moins d'octets que si l'adresse de R1 devait être fournie.

L'exemple suivant (toujours du jeu d'instruction MCS-51) utilise de l'adressage indirect par registre et de l'adressage avec un registre particulier appelé *accumulateur* :

```
MOV A,@R2
```

7.2.7 Orthogonalité du jeu d'instruction

Le jeu d'instruction d'un CPU est dit orthogonal lorsque tous les modes d'adressage disponibles sont utilisables avec toutes les instructions, pour autant que le mode d'adressage considéré a du sens avec l'instruction considérée.

7.3 Jeu d'instruction du MSP430

Les chapitres suivants donnent les modes d'adressage disponibles, et des exemples d'instructions du MSP430 pour chaque mode d'adressage. La liste exhaustive peut être consultée dans le manuel d'utilisateur de la famille MSP430 (document SLAU208, pages 185 à 340).

7.3.1 Modes d'adressage

L'espace mémoire complet est adressable sans exception. Toutefois, les modes d'adressage disponibles sont différents pour l'opérande source et l'opérande destination :

- sept modes d'adressage pour les opérandes de source
- quatre modes d'adressage pour les opérandes de destination

Ces modes d'adressage sont applicables aussi bien à des instructions de transfert de données(MOV) qu'à des opérations arithmétiques (ADD, SUB,...). La table 7.1 donne les différents modes d'adressage. Les champs *Ad* (destination) et *As* (source) définissent le mode pour chaque opérande et font partie du code de l'instruction.

As	Ad	Mode	Syntaxe	Description
00	0	Registre	Rn	Les opérandes sont les contenus des registres
01	1	Indexé	x(Rn)	[Rn+x] pointe l'opérande. Les instructions utilisant ce mode d'adressage contiennent au moins deux mots. Le second contient x
01	1	Symbolique	Adresse	[PC+x] pointe l'opérande. Il s'agit d'un mode indexé x(PC) Les instructions utilisant ce mode d'adressage contiennent au moins deux mots. Le second contient x
01	1	Direct (Absolu)	&Adresse	Le ou les deux mots de 16 bits suivant l'instruction contiennent les valeurs des adresses. Le mode utilisé est le mode indexé x(SR)
10	-	Indirect	@Rn	[Rn] pointe l'opérande
11	-	Indirect avec autoincrément	@Rn+	[Rn] pointe l'opérande puis Rn+1 -> Rn Il y a donc post-incrémentation
11	-	Immédiat	*#N*	Les instructions utilisant ce mode d'adressage contiennent au moins deux mots. Le second contient N Il s'agit d'un mode avec auto-incrémentation PC+1 -> PC puis [PC] pointe N

TABLE 7.1 – Modes d'adressage du MSP430

Les modes *indexé*, *symbolique* et *direct* sont considérés comme trois cas particuliers du mode *indexé*. Le mode *direct* considère l'adresse comme un index à ajouter à l'adresse 0x0000. Le mode *symbolique* considère l'adresse comme un index à ajouter au compteur de programme PC.

7.3.2 Format des instructions

Les instructions sont en principe codées sur un mot de 16 bits, auquel s'ajoutent éventuellement 1 ou 2 mots selon le mode d'adressage utilisé. Par exemple, l'adressage immédiat nécessite de joindre le ou les opérandes au code de l'instruction.

Pour le codage de l'instruction elle-même, il existe trois formats selon le nombre d'opérandes :

- format I pour les instructions à 2 opérandes. Par exemple, une addition exécute une opération du type
  $SRC + DST -> DST$ (source + destination va dans destination)
- format II pour les instructions à 1 opérandes ;
- format III, spécifique pour les instructions de test et branchement.

les figures 7.1, 7.2 et 7.3 illustrent comment sont structurées les instructions selon leur format.

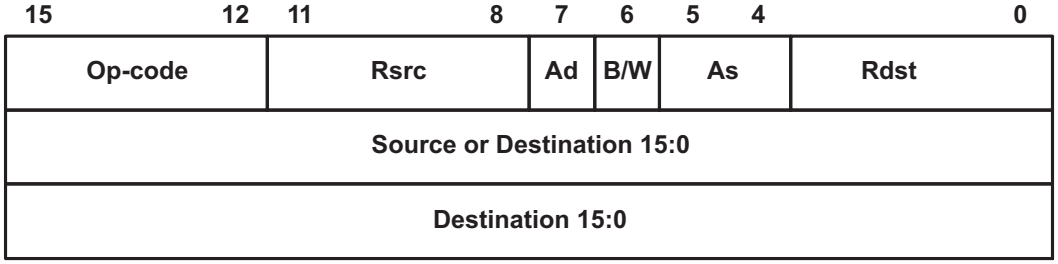

FIGURE 7.1 – Format I des instructions à deux opérandes

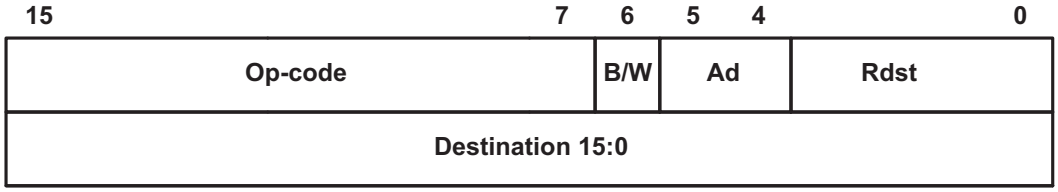

FIGURE 7.2 – Format II des instructions à un opérande

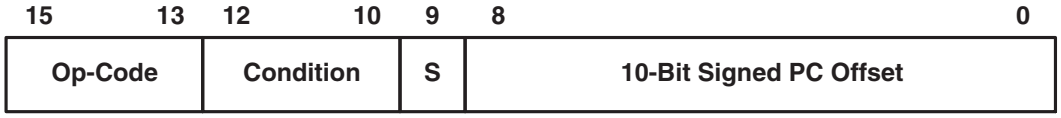

FIGURE 7.3 – Format III pour instructions de test et branchement

7.3.3 Exemple 1 : adressage par registre

Le CPU MSP430 contient 16 registres notés R0 à R15. Chacun d'entre eux peut être utilisé avec ce mode d'adressage. L'instruction

```
MOV.W R10,R8
```

permet de copier le contenu du registre R10 dans le registre R8. Le suffixe *.W* spécifie que les opérandes sont des mots (de 16 bits). L'instruction est codée sur un mot de 16 bits et utilise le format I, comme indiqué dans la figure 7.1. Du fait que l'adressage est *par registre*, cette instruction n'a pas besoin d'autres informations que l'identification des registres R10 et R8. Son code est :

```
Instruction : 0100 1010 0000 1000 ou 0x4A08
```

Op-code = 4, Rsrc = 10, Ad=0, B/W=0, As=0, Rdst=8

7.3.4 Exemple 2 : adressage indexé par registre

L'adressage indexé du MSP430 est en fait un adressage indirect indexé. En effet, le contenu du registre considéré est additionné à l'index pour obtenir l'adresse de l'opérande. L'instruction

```
MOV.B R10,5(R8)
```

permet de copier le contenu du registre R10 dans la case mémoire dont l'adresse est égale à 5 plus le contenu du registre R8. Le suffixe *.B* spécifie que les opérandes sont des octets. L'instruction utilise le format I, et est codée sur deux mots de 16 bits ; un pour l'instruction elle-même, l'autre est égal à 5. Son code est :

```
Premier mot : 0101 1010 1100 1000   ou 0x4AC8
Second mot  : 0000 0000 0000 0101   ou 0x0005
```

Op-code = 4, Rsrc = 10, Ad=1, B/W=1, As=0, Rdst=8

7.3.5 Exemple 3 : adressage direct

L'instruction

```
ADD.W   R10,&3310
```

permet d'additionner le contenu du registre R10 à celui de la case mémoire d'adresse 3310 et de placer le résultat dans cette case mémoire. L'instruction utilise le format I, et est codée sur deux mots de 16 bits ; un pour l'instruction elle-même, l'autre est égal à 0x0CEE (3310 en décimal). Son code est :

```
Premier mot : 0101 1010 1000 0010   ou 0x5A82
Second mot  : 0000 1100 1110 1110   ou 0x0CEE
```

Op-code = 5, Rsrc = 10, Ad=1, B/W=0, As=0, Rdst=2 Comme indiqué dans la table 7.1, cet adressage est considéré comme de l'adressage indexé. Le registre R2 est ici utilisé comme générateur de l'adresse de départ 0x0000.

7.3.6 Exemple 4 : adressage indirect

L'instruction

```
ADD.W   @R10,R8
```

permet d'additionner le contenu de la case mémoire dont l'adresse est dans le registre R10 au contenu de R8 et de placer le résultat dans R8. L'instruction utilise le format I, et est codée sur un mot de 16 bits. Son code est :

```
Instruction : 0101 1010 0010 1000 ou 0x5A28
```

Op-code = 5, Rsrc = 10, Ad=0, B/W=0, As=2, Rdst=8

7.3.7 Exemple 5 : adressage indirect avec autoincrément

L'instruction

```
ADD.W   @R10+,R8
```

permet d'additionner le contenu de la case mémoire dont l'adresse est dans le registre R10 au contenu de R8 et de placer le résultat dans R8. A la fin de cette instruction, le contenu de R10 est incrémenté de 2 unités pour pointer sur le prochain mot. Il serait incrémenté de 1 si le suffixe *.B* avait été utilisé. L'instruction utilise le format I, et est codée sur un mot de 16 bits. Son code est :

```
Instruction : 0101 1010 0011 1000 ou 0x5A38
```

Op-code = 5, Rsrc = 10, Ad=1, B/W=0, As=3, Rdst=8

7.3.8 Exemple 6 : adressage immédiat

L'instruction

```
ADD.W   #3253,&3310
```

permet d'additionner la valeur 3253 (décimal) au contenu de la case mémoire d'adresse 3310 et de placer le résultat dans cette case mémoire. L'instruction utilise le format I, et est codée sur trois mots de 16 bits ; un pour l'instruction elle-même, le second pour la valeur 3253, le troisième est égal à 0x0CEE (3310 en décimal). Son code est :

```
Premier mot    : 0101 1010 1000 0010 ou 0x50B2
Second mot     : 0000 1100 1011 0101 ou 0x0CB5 (3253 en décimal)
Troisième mot  : 0000 1100 1110 1110 ou 0x0CEE
```

Op-code = 5, Rsrc = 00, Ad=1, B/W=0, As=0, Rdst=2

7.3.9 Exemple 7 : adressage symbolique

L'instruction

```
CALL    #routine
```

permet "d'envoyer" le CPU exécuter l'instruction située à l'adresse `routine`. Cette instruction sera de fait considérée comme la première du groupe qui suit, qui constitue donc un *sous-programme*. La fin de ce sous-programme est délimitée par une instruction appelée `RET`. L'instruction donnée en exemple utilise le format II, et est codée sur deux mots de 16 bits ; un pour l'instruction elle-même, le second pour l'adresse `routine`. Si `routine = 0x31A2`, le code de l'instruction est :

```
Premier mot    : 0001 0010 1011 0000 ou 0x50B2
Second mot     : 0011 0001 1010 0010 ou 0x31A2
```

Op-code = 0x25, B/W=0, Ad=3, Rdst=0

7.3.10 Exemple 8 : adressage symbolique

L'instruction

```
JMP    label
```

permet de faire un saut dans le programme, dans un sens ou dans l'autre, d'au maximum 1024 adresses, sans retour prévu. L'instruction donnée en exemple utilise le format III, et est codée sur un mot de 16 bits. Si `label = 0x31A8`, et si l'instruction est localisée à l'adresse 0x314E, alors le code de l'instruction est :

```
Instruction : 0011 1100 0010 1101  ou 0x3C2D
```

Le champ "PC Offset" (voir figure 7.3) est calculé de sorte que
$0x0314E + 2 * PC\_Offset = 0x31A8$

7.4 Chaine de compilation

Le développement d'un programme pour microcontrôleur consiste à générer du code machine à partir d'un programme écrit en C ou en C++, en passant par une vue en langage *assembleur*. En général, le processus de génération du code machine doit permettre de réutiliser du code déjà écrit, comme une librairie de fonctions que l'on aura déjà testées, validées et documentées. Une question se pose alors, qui est le calcul des adresses de début de ces fonctions. Ce calcul est réalisé lors d'une opération appelée *édition de lien*.

la figure 7.4 illustre le processus de développement de code.

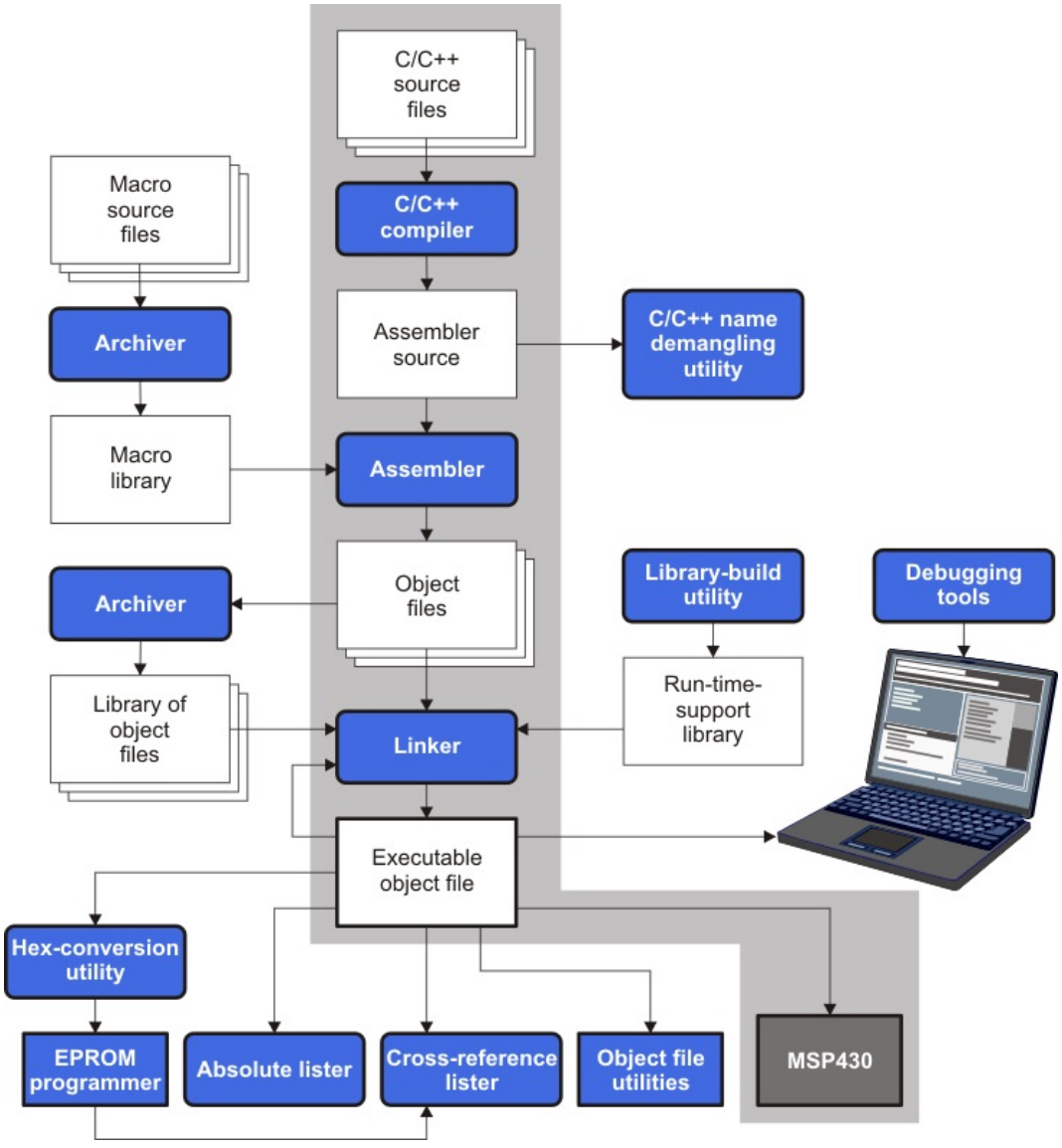

FIGURE 7.4 – Processus de développement de code

La partie grisée de la figure montre le chemin le plus courant pour le développement. Les autres parties sont optionnelles.

Le point d'entrée du processus (haut de la figure) est le code source, écrit en langage C ou C++.

La première étape est la compilation, qui traduit le code source en langage assembleur. Dans l'outil de développement, il est possible de voir le code assembleur correspondant au code source. A ce stade, la localisation du code assembleur dans la mémoire est provisoire, puisqu'il faut encore tenir compte des fonctions déjà existantes.

Dans la seconde étape, l'outil d'assemblage (appelé *l'assembleur* transforme donc les fichiers en langage assembleur en fichiers *objet* relocalisables. Cette étape implique de connaître en particulier les liens entre les instructions de type `JMP` et `CALL` et les instructions vers lequel elles pointent.

La troisième étape est l'édition de lien, qui consiste à combiner les fichiers relocalisables en un fichier exécutable (en langage machine) unique, dans lequel les adresses exactes de toutes les instructions sont fixées.

Une fois le fichier contenant le code exécutable obtenu, il peut être transféré dans la mémoire du microcontrôleur via un port spécifique appelé JTAG. En général, le transfert s'effectue par une liaison USB. L'interface entre le port USB et le port JTAG offre des fonctionnalités de debug, telles que :

- possibilité de placer des points d'arrêt lors de l'exécution du code ;
- possibilité d'exécuter le programme pas à pas ;
- etc...

Pour cette raison, cette interface est souvent appelée *sonde programmation et debug* ou *sonde de debug* ou encore *sonde JTAG*.

Chapitre 8

Unité centrale

L'unité centrale ou CPU (Central Processing Unit) et le coeur du microcontrôleur. Elle effectue des opérations élémentaires sur des données ou sur des adresses. A l'intérieur du cpu on trouve une unité arithmétique et logique qu'on nomme ALU (Aritmetic and Logic Unit). c'est elle qui effectue les opérations de base tel que addition, multiplication, décalages et fonctions logiques. Pour qu'elle fonctionne optimalement, il faut l'alimenter avec des données de manière continue. Par la suite, on va détailler ce processus.

8.1 Principes

Un processeur se décompose en deux unités fondamentales : i) une unité de contrôle et ii) une unité de traitement (fig. 8.1). L'unité de contrôle décode les instructions et pilote le chemin de données qui exécute les opérations. Le pilote du chemin de données s'appelle le séquenceur, il communique avec le chemin de données à l'aide de signaux de commande et reçois des informations sur son état.

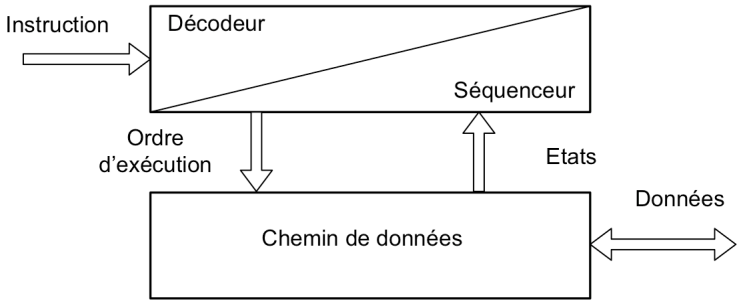

FIGURE 8.1 – Principe de fonctionnement de l'unité centrale

Le chemin de donnée et l'unité de contrôle sont reliées par deux bus, le bus de données et le bus d'adresses (fig. 8.2). Le premier défini la largeur des mots traitables par le processeur (8, 16, 32 bits), le deuxième défini la taille mémoire adressable, par exemple : 16 bits = 65536 adresses.

Le séquenceur est aussi responsable d'amener et de stocker des données dans la mémoire. On constate que le processeur à besoin de deux types d'information, les instructions et les données. Il y a donc deux possibilités, l'une ou l'on stock les deux informations dans une seule mémoire et l'autre ou on utilise deux mémoires séparées. Ce choix à provoqué une distinction importante en architecture des processeurs qu'on a nommé Von Neumann et Harvard.

8.1.1 Architectures de Von Neumann et Harvard

Cette différence de stockage des données et des instructions peut amener des avantages et des inconvénients. Si les données et les instructions sont séparées, il faut deux mémoires et deux bus d'accès. Par contre, si il y a deux mémoires, le chargement se fait en parallèle. On peut donc considérer que l'architecture de Von Neumann est plus simple mais moins performante que celle de Harvard.

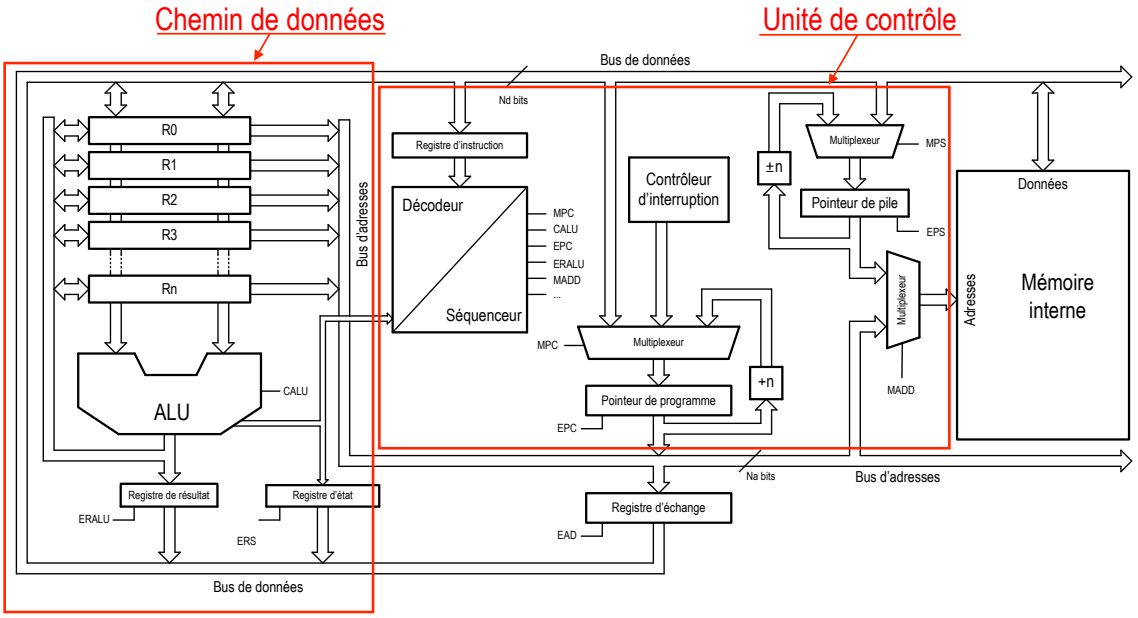

FIGURE 8.2 – Exemple d'unité centrale

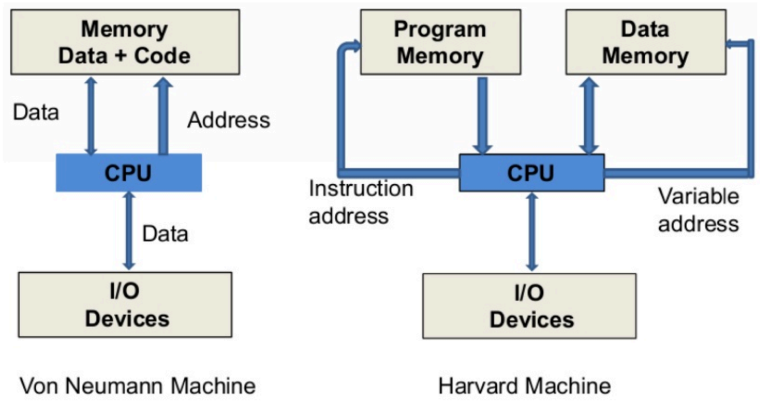

FIGURE 8.3 – Les architectures de Von Neumann et Harvard

8.1.2 Pile

La gestion d'une pile mémoire est essentielle au fonctionnement d'un programme. Elle s'occupe de stocker temporairement l'information pour gérer :

- Le changement de contexte comme lors d'une interruption ou d'un appel de fonction
- Le stockage des paramètres de fonction
- Le stockage des variables locales
- Certaines variables lors de branchements multiples (switch-case)

La pile peut être de type matérielle ou logicielle, dans le premier cas elle aura un nombre limité de registres et dans le deuxième elle sera juste une zone de la mémoire RAM.

8.2 Registres à usages spécifiques

Parmi tous les registres de la machine, il y en a trois qui sont essentiels au bon fonctionnement de la machine et qui sont accessibles au programmeur.

8.2.1 Registre d'état

C'est le registre contenant l'état de la machine, appelé SR (Status Register), qui peut contenir les informations suivantes :

- Statut de la dernière opération arithmétique sur 4 bits (CVNZ)
- Interruptions activées ou non (1 bit)
- Mode low power 2-3 bits
- Configuration des horloges 2-3 bits

La longueur du registre est normalisée par rapport à la largeur du bus de données.

Les bits de statut arithmétique sont utilisés pour des tests ou pour détecter des dépassements de capacité :
C : Carry, c'est l'indication qu'une retenue a été générée
V : oVerflow, c'est l'indication d'un dépassement lors du calcul avec des nombres signés
N : Negative, c'est l'indication d'un résultat négatif
Z : Zero, c'est l'indication d'un résultat nul

8.2.2 Pointeur de programme

Registre contenant l'adresse mémoire de l'instruction en cours appelé PC (Program Counter). La longueur de ce registre est celui de la largeur du bus d'adresses. Ce registre est incrémenté, après chaque instruction, d'un entier qui correspond à la longueur de l'instruction précédente.

8.2.3 Pointeur de pile

Registre contenant l'adresse mémoire du dessus de la pile appelé SP (Stack Pointer). La longueur de ce registre est celui de la largeur du bus d'adresses. Ce registre est incrémenté, après chaque empilement/désempilement, d'un entier qui correspond à la longueur de l'information stockée dans la pile.

8.3 Cycle des instructions

Pour analyser en détail le fonctionnement du processeur soumis à une instruction, on va diviser le processus en cinq parties :

1. Recherche de l'instruction (IF : Instruction Fetch)
2. Décodage de l'instruction (ID : Instruction Decode)
3. Recherche des opérandes (OF : Operand Fetch)
4. Exécution de l'instruction (EX : EXecute)
5. Écriture du résultat (WR : Write Register ou WB : Write Back)

8.3.1 Recherche de l'instruction

Dans cette phase, on doit amener l'instruction de la mémoire vers un registre d'instruction dans le séquenceur. La première chose à faire c'est de placer sur le bus d'adresses la valeur du pointeur de programme (Le pointeur de programme contient l'adresse de la ligne du programme). Cette adresse est alors dirigée sur la mémoire qui retourne la valeur qu'elle contient sur le bus de données (fig. 8.4).

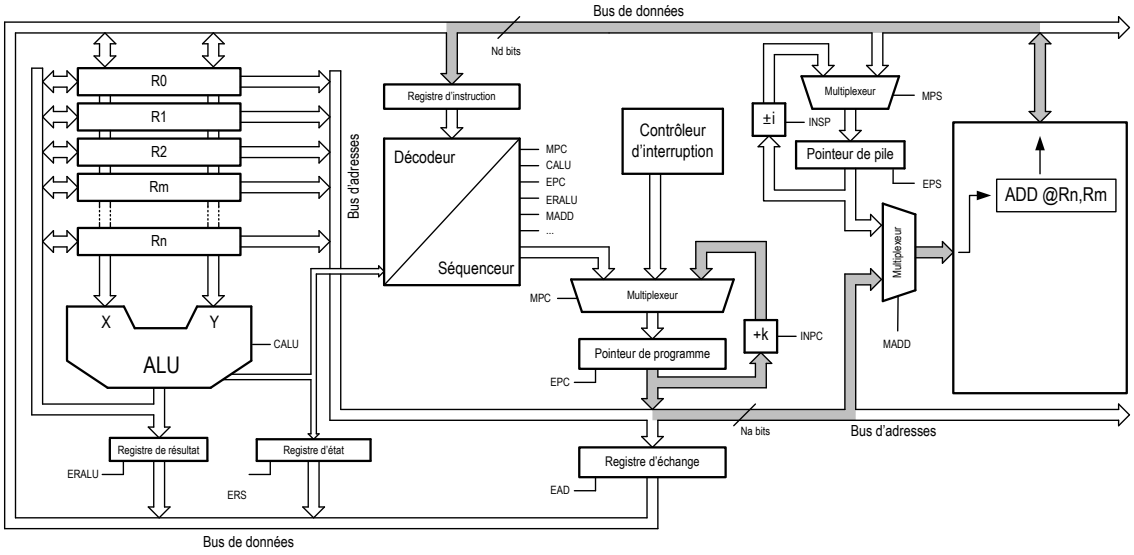

FIGURE 8.4 – Recherche de l'instruction

8.3.2 Décodage de l'instruction

Dés que l'instruction est arrivée dans le séquenceur, il peut la décoder en lisant d'abord l'opcode qui identifie l'instruction à effectuer par exemple add (fig. 8.6). Deuxièmement, il lit la méthode d'adressage utilisée pour trouver l'endroit de la source et de la destination.

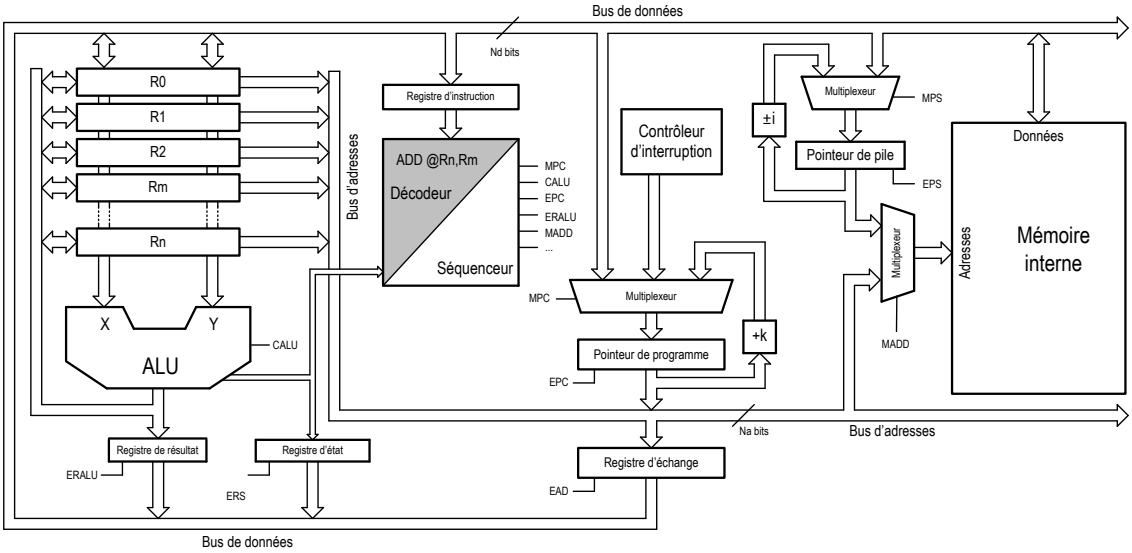

FIGURE 8.5 – Décodage de l'instruction

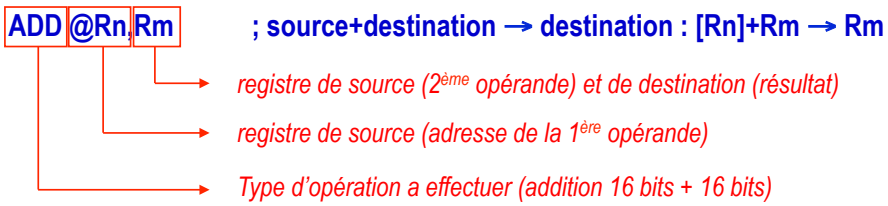

FIGURE 8.6 – Exemple d'instruction assembleur

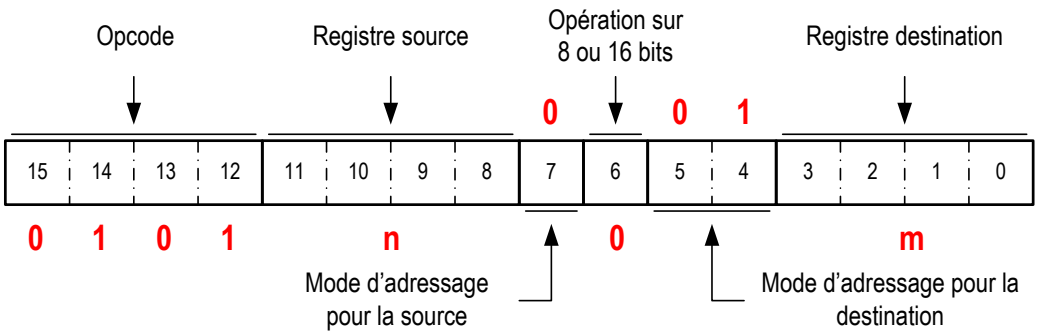

FIGURE 8.7 – Exemple de codage d'une instruction

8.3.3 Recherche des opérandes

Après décodage, le séquenceur entre en oeuvre et va chercher les opérandes suivant le mode d'adressage utilisé. Si l'adressage est direct, les registres internes au chemin de données sont activés et présentent leurs valeurs aux entrées de l'ALU. Si l'adressage est indirect, le séquenceur place la valeur contenue dans un registre sur le bus d'adresse. Ensuite la valeur en mémoire située à cette adresse est placée sur le bus de donnée pour atteindre finalement l'entrée de l'ALU.

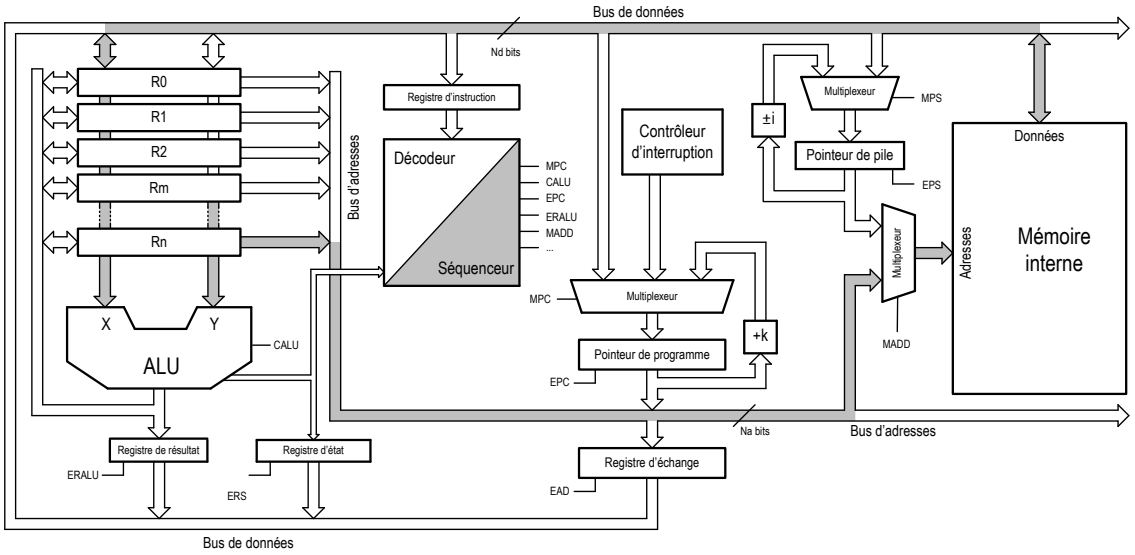

FIGURE 8.8 – Recherche des opérandes

8.3.4 Exécution de l'instruction

A partir du moment ou les données sont présentent aux deux entrées de l'ALU, le séquenceur commande l'exécution de l'opération. Suivant la complexité de l'opération, l'exécution se décomposera en plusieurs sous opérations.

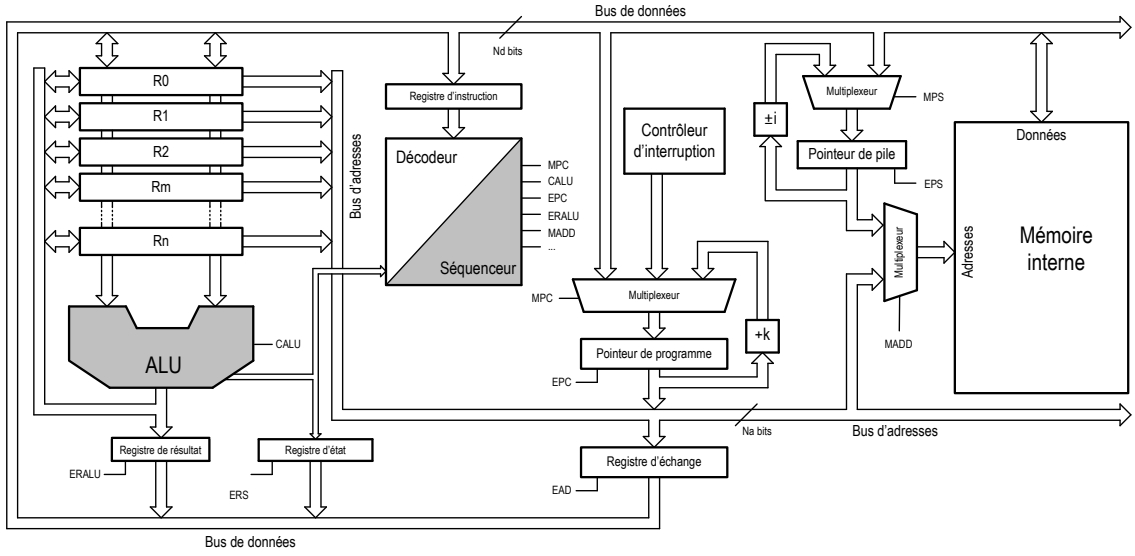

FIGURE 8.9 – Exécution de l'instruction

8.3.5 Écriture du résultat

Après le calcul de l'ALU, le résultat est placé dans le registre de destination. Le registre d'état est aussi mis à jour en même temps. Ce registre nous donne les informations suivantes :

- Indication d'une retenue : C (pour Carry)
- Indication d'un dépassement : V (pour oVerflow)
- Indication d'un résultat négatif : N (pour Negative)
- Indication d'un résultat nul : Z (pour Zero)

Grâce à ces informations, nous pouvons programmer en utilisant des conditions. Par exemple, pour tester lorsque qu'une boucle for arrive à zéro.

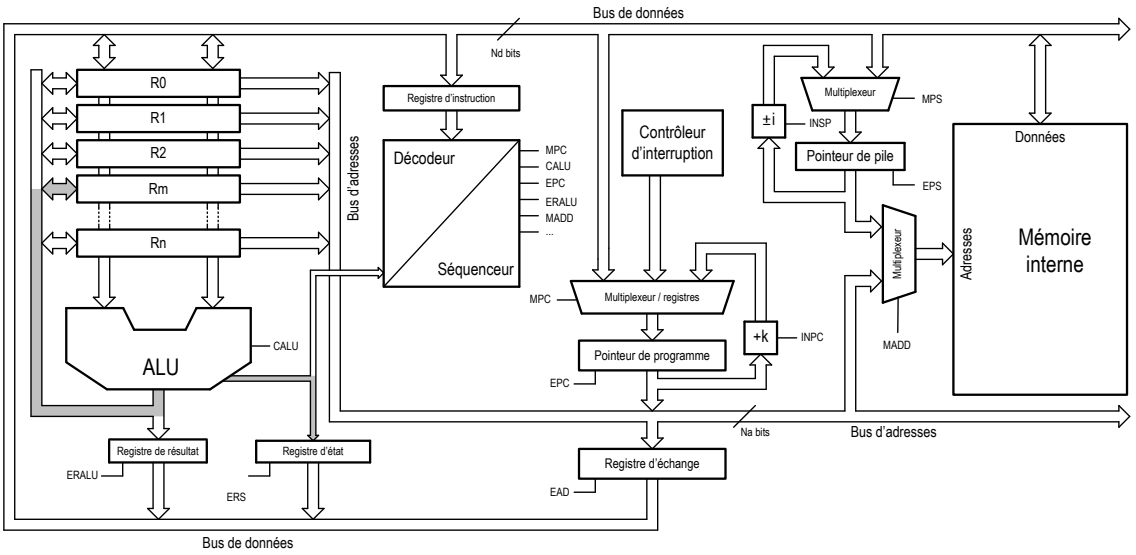

FIGURE 8.10 – Écriture du résultat

8.4 Séquenceur des instructions

Le séquenceur est un composants clé qui à pour but de gérer le bon fonctionnement du processeur. Il prend en compte les temps d'exécutions de chaque sous composants. On peut le réaliser de deux manières différentes : i) cablé (fig. 8.11), c-à-d par des fonctions logique séquentielles ou ii) microprogrammé (fig. 8.12), qui signifie qu'il peut être reconfiguré.

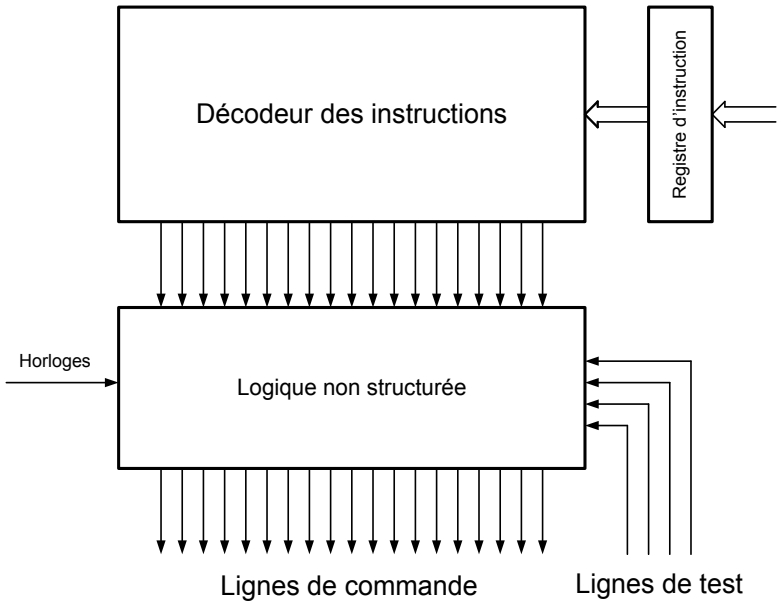

FIGURE 8.11 – Séquenceur câblé

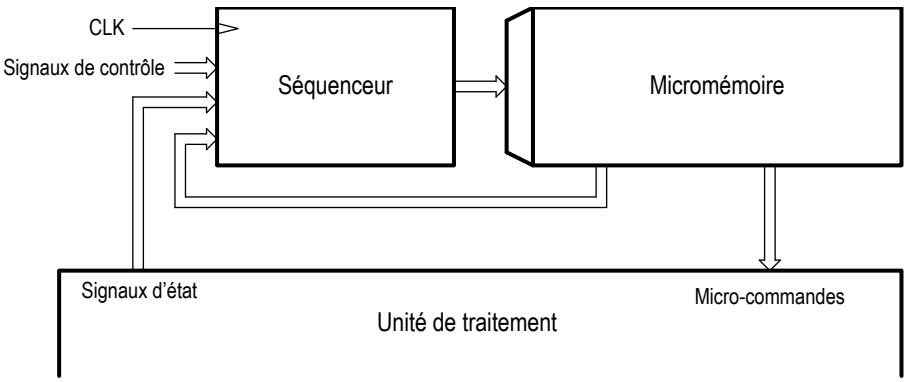

FIGURE 8.12 – Séquenceur microprogrammé

8.5 Pipeline du cycle des instructions

Les cinq étapes citées auparavant doivent être exécutées à la suite ce qui donne un cycle minimum de cinq coup d'horloges pour exécuter une instruction (fig. 8.14). On désire donc commencer une nouvelle instruction dés que possible c-à-d dés qu'un étape est terminée ((fig. 8.16)).

Pour se faire, on intercale des registres entre les différentes phases du processus ((fig. 8.15)). Ceci permet d'isoler chaque phase et ainsi de permettre la réutilisation des phases à chaque cycle d'horloge du processeur.

Certaines opérations sont plus longues que d'autres ce qui fait que l'horloge doit être ajustée pour l'étape la plus lente. Pour éviter de devoir ralentir l'horloge du cpu on peut ajouter des niveaux de pipeline

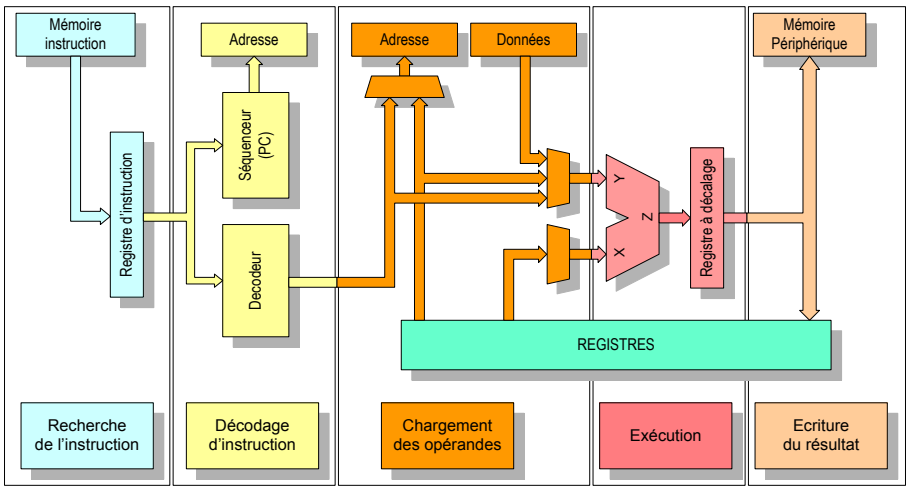

FIGURE 8.13 – Flux des instructions sans pipeline

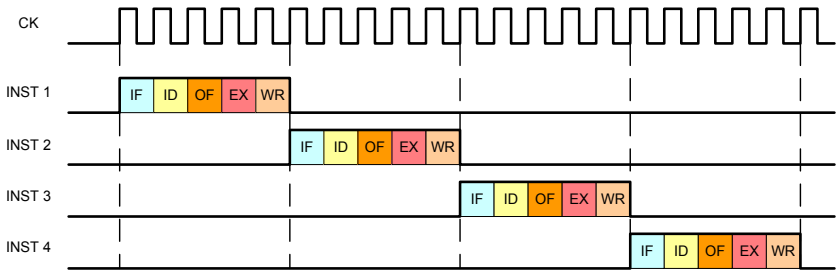

FIGURE 8.14 – Flux des instructions sans pipeline

pour accélérer le processeur. Par exemple un multiplieur prendra plus de temps qu'un additionneur, il est donc judicieux de réaliser le multiplieur avec plusieurs étages de pipeline.

8.6 Processeur superscalaire

Un processeur capable d'exécuter plusieurs instructions en parallèle est dit superscalaire. Ce n'est pas un ensemble de processeurs mais un seul CPU avec plusieurs ALU qui peut aussi avoir plusieurs décodeurs/séquenceurs. On utilise aussi la terminologie anglo-saxone "dual-issue" pour un processeur qui peut exécuter deux instructions en parallèle et "multiple-issue" pour un processeur qui peut en exécuter plusieurs. Un processeur qui peut exécuter une seule instruction par cycle d'horloge est donc appelé "single-issue", c'est le cas de la plupart des micro-contrôleurs.

Un processeur qui possède plusieurs ALU mais pas plusieurs décodeurs d'instructions est dit avec pipeline superscalaire. L'ARM cortex M7, par exemple, possède cinq pipelines : deux ALU entières, un pour les opération mémoire, un pour faire des multiplications/accumulations et un pour le calcul en virgule flottante. Cela permet d'équilibrer les opérations qui ont des temps d'exécutions différents tel que le calcul en virgule flottante par rapport au calcul entier. Par exemple, une instruction flottante prend 4 étages de pipeline alors qu'une instruction entière en prend 2. Donc si on place une instruction flottante suivie par une instruction entière, on a bien deux instructions qui s'exécutent en parallèle. L'instruction entière se terminera un cycle avant l'instruction flottante.

La présence de processeurs de plus en plus puissants dans les micro-contrôleurs est inévitable. Ceci est dû à plusieurs facteurs :

1. La densité d'intégration des technologies CMOS => plus de transistors disponibles
2. La réduction de la consommation dynamique liée aux circuits plus petits

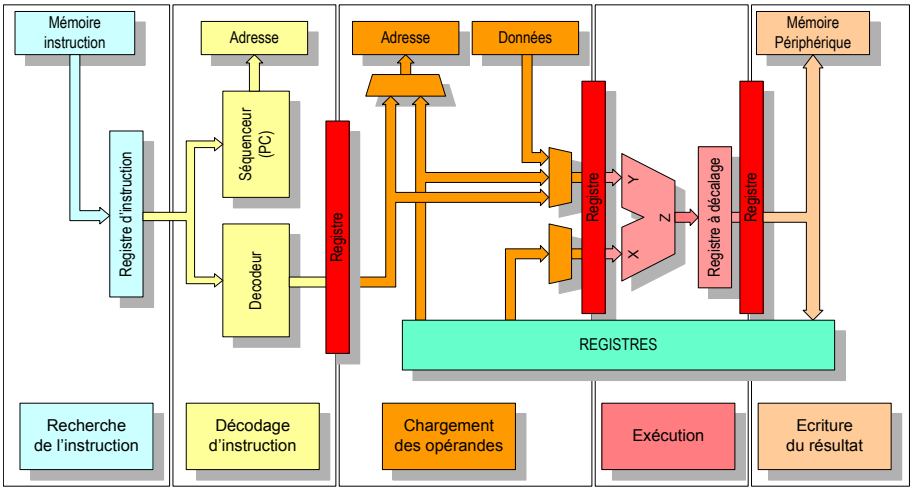

FIGURE 8.15 – Flux des instructions avec pipeline

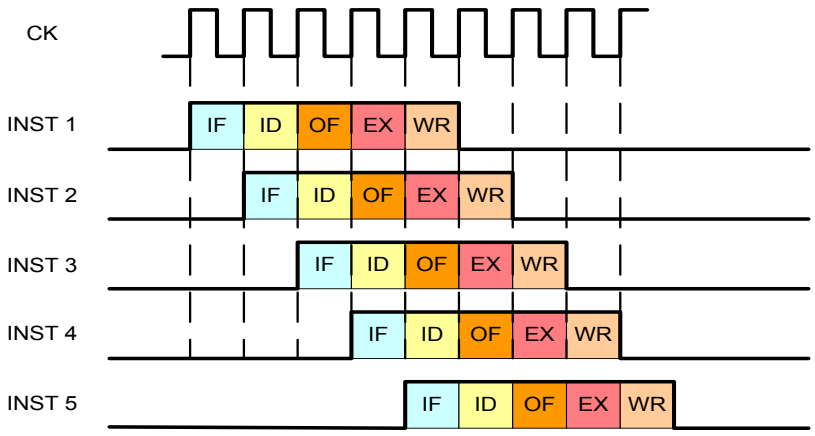

FIGURE 8.16 – Flux des instructions avec pipeline

3. La vitesse et complexité des interfaces de communications, par exemple l'USB
4. La complexité des périphériques disponibles, par exemple les afficheurs

On peut donc s'attendre à trouver des microcontroleurs qui s'approchent des fonctionnalités d'un ordinateur de base.

Chapitre 9

Programmation en C

Pour atteindre le matériel depuis un langage de plus haut niveau, il faut un certain nombre d'instructions et de règles pour pouvoir rester dans un ensemble cohérent.

9.1 Programmation système et programmation applicative

Les règles de programmation sont différentes si on écrit du code relatif à du matériel par rapport à des applications purement logiciel. Le premier exemple, déjà traité au chapitre 4, est les interruptions. Ce sont des fonctions système que l'on trouve seulement lorsqu'on programme à bas niveau.

Programmation Système	Programmation Applicative
Interruptions	Pas d'interruptions
Variables globales	Éviter les variables globales
Langages de bas niveau	Langages de haut niveau
Structuration faible	Structuration forte

TABLE 9.1 – Comparaison des styles de programmation

9.2 Directives et variables spécifiques à la programmation système

9.2.1 Variables volatiles

Dans le cas ou une zone de stockage, tel qu'un registre, peut être modifié par autre chose que le programme principal on doit lui attribuer le modificateur volatile. Cela permet au compilateur de savoir que cette variable peut changer même si le code n'affecte jamais cette variable. Le compilateur ne va donc pas essayer d'optimiser une variable volatile (Exemple ci-dessous).

9.2.2 Directive pragma

La directive #pragma est une commande de précompilation définie par le standard ISO/ANSI C. Elle permet de garder une certaine portabilité du logiciel en contrôlant de manière particulière les extensions spécifiques à un fabricant. Cette directive permet, par exemple, de spécifier une adresse absolue qui désigne la position mémoire pour la déclaration suivante. La variable doit être déclarée soit ___no_init ou const.

```
#pragma location=0x22E
__no_init volatile char PORT1;  //PORT1 est un registre du périphérique
                                //du même nom placé à l'adresse 0x22E
```

La directive #pragma peut prendre d'autres fonctions selon le compilateur utilisé.

9.2.3 Fonctions intrinsèques

Les fonctions intrinsèques se reconnaissent par un double souligné devant (___). Les fonctions intrinsèques permettent un accès direct aux instructions de bas niveau sans programmer en assembleur. Si le compilateur ne trouve pas les fonctions intrinsèques par défaut alors il faut inclure : intrinsics.h. Ces fonctions ne sont pas standard, elle dépendent du compilateur utilisé.

Exemple de fonctions intrinsèques :

```
__no_init                    //indique de ne pas initialiser une variable
__disable_interrupt          //désactive toutes les interruptions
__enable_interrupt           //active toutes les interruptions
__low_power_mode_n           //enclenche le mode low power n
__low_power_mode_off_on_exit //désactive le mode low power en sortie d'
    interruption
__asm()                      //permet l'insertion de code assembleur
```

9.3 Librairies d'abstraction du matériel : HAL

Pour rendre l'écriture de programmes plus propre on peut ajouter de la hiérarchie. Une façon de faire est de placer toutes les fonctions relatives aux périphériques dans des librairies d'abstraction de matériel ou HAL (Hardware Abstraction Library).

Une librairie HAL doit permettre de masquer tout ce qui est relatif à un périphérique donné. La structure typique d'un HAL est la suivante :

- Constantes spécifiques
- Variables spécifiques
- Une fonction d'initialisation du périphérique
- Une fonction de mise en veille ou désactivation
- Des fonctions spécifiques comme lecture ou écriture

Exemples de structure HAL pour un accéléromètre (CMA3000) :

```
#define CTRL                    0x02
#define MODE_400                0x04        // Measurement mode 400 Hz ODR

int8_t Cma3000_xAccel;
int8_t Cma3000_yAccel;
int8_t Cma3000_zAccel;

void Cma3000_init(void)
{
    ...
}

void Cma3000_disable(void)
{
    ...
}

void Cma3000_readAccel(void)
{
    ...
}

...
```

Les interruptions relatives à un périphérique doivent être aussi traitées dans le HAL.

9.4 Pilotes

Un pilote ou driver permet la gestion d'un périphérique dans le cadre d'un système d'exploitation. C'est un HAL avec en plus les interfaces standards propre à un système d'exploitation. Il existe donc des

standards pour écrire des pilotes.

9.5 Contrôle de l'exécution

Dans le cadre d'une programmation simple (mono-tâche), le processeur effectue les instructions linéairement à sa vitesse maximale. On a donc des instants actifs, avec des instructions à exécuter, et des instants inactifs où il n'y a rien à faire. Dans le cas le plus simple, la partie inactive est une attente active ou le processeur exécute une boucle infinie genre : while(1). Ceci revient à faire le même test qui est toujours vrai et donc à gaspiller de l'énergie. Il est de ce fait plus intéressant de mettre le processeur en pause. Ceci implique qu'il faut gérer le réveil au moment requis par le système.

Le cycle actif/pause peut être géré de deux manières :

1. Par interruption, une requête externe par exemple
2. Par un timer, réveil à intervalle régulier.

Exemple de code pour MSP430 dans le cas (1) :

```
void main(void) {
    while(1) {
        __bis_SR_register(LPM3_bits + GIE); //Mode basse consommation LPM3
                                            //avec interruptions activées
        ...                                 //Code de l'application,
                                            //mode low power!
    }
}

#pragma vector=PORT2_VECTOR
__interrupt void Port2_ISR(void) {

    ...                                     //Code de l'interruption,
                                            //mode actif par défaut
    __low_power_mode_off_on_exit
}
```

Pour bien comprendre le code ci-dessus, il faut savoir qu'une interruption est toujours exécutée en mode actif (pas low power) et que dès la sortie, le processeur retourne dans le mode d'avant l'interruption. Il faut donc utiliser la fonction intrinsèque ___low_power_mode_off_on_exit pour retourner dans le programme principal en mode actif et exécuter le code de l'application. Le mécanisme qui régit cette opération est la sauvegarde du contexte qui sauve sur la pile le registre de statut SR lors de l'interruption puis le dépile lors de la sortie. Rappel : les modes basse consommation sont désignés dans le SR.

Exemple de code pour MSP430 dans le cas (2) :

```
void main(void) {
    while(1) {
        __bis_SR_register(LPM3_bits + GIE); //Mode basse consommation LPM3
                                            //avec interruptions activées
        ...                                 //Code de l'application,
                                            //mode low power!
    }
}

#pragma vector=TIMER_A0_VECTOR
__interrupt void Timer_ISR(void) {

    ...                                     //Code de l'interruption,
                                            //mode actif par défaut
    __low_power_mode_off_on_exit
}
```

Chapitre 10

Interfaces séries

Les interfaces parallèles ne sont quasiment plus utilisées aujourd'hui à cause du nombre de broches nécessaires et des performances électriques demandées. L'interface série n'utilise que 2 ou 3 broches et bénéficie de meilleur performances. Elles demandent, par contre, plus de matériel et de logiciel pour effectuer la sérialisation et dé-sérialisation.

10.1 Notion de pile protocolaire

Il est utile d'inscrire les interfaces séries au sein des notions de la téléinformatique comme celle de la pile protocolaire. Cette pile permet de découper toutes les opérations nécessaires en couches. Chaque couche à une fonction bien précise et est développée indépendamment des autres.

10.1.1 Modèle OSI (Open Systems Interconnection)

Le modèle OSI est un modèle théorique d'une pile de communication avec sept couches (fig. 10.1).

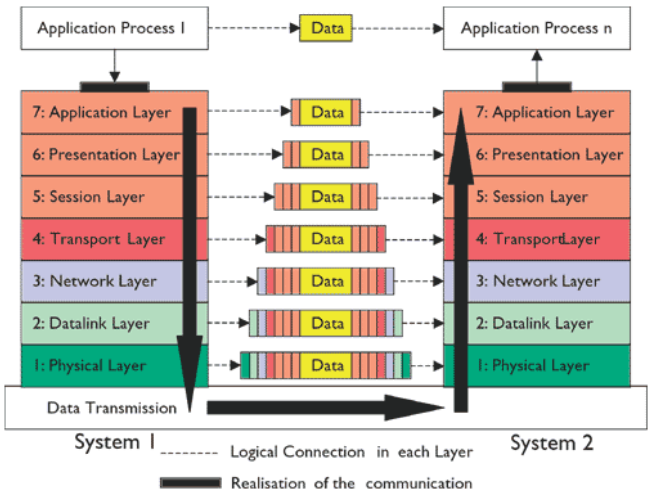

FIGURE 10.1 – Modèle OSI

Les couches qui sont nécessaires à l'établissement d'une communication sont les couches 1 à 4. Les autres sont des couches liées aux services et aux applications.

1. Couche physique : Traite les aspects électriques de la ligne série
2. Couche paquet : Encapsule les données avec des bits de contrôle e.g. pour la détection d'erreurs
3. Couche réseau : Indique la destination (s'il y a plus que deux noeuds sur le réseau)
4. Couche transport : Découpe les données en paquets et gère la retransmission en cas d'erreurs

10.1.2 Pile protocolaire IP

Le protocole IP (Internet Protocol) est simplifié par rapport au modèle OSI (fig. 10.2). Il ne contient que quatre couches ce qui parait plus simple mais en pratique n'est pas idéal car il groupe plusieurs fonctions dans une seule couche.

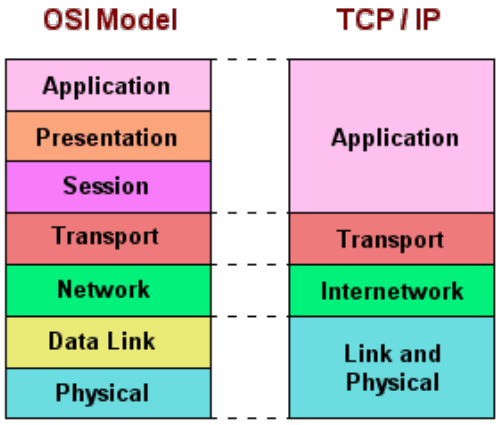

FIGURE 10.2 – Comparaison des piles protocolaires

10.2 UART (Universal Asynchronous Receiver Transmitter)

Les UART sont des sérialiseurs/désérialiseurs simples. Un UART permet d'implémenter la couche 2 du modèle OSI. L'UART est asynchrone ce qui veut dire qu'il ne propage pas de signal d'horloge de l'émetteur au récepteur. Il est néanmoins aisé d'ajouter une ligne d'horloge pour rendre l'UART synchrone.

Au niveau matériel, l'UART est composé de 4 registres dont deux à décalage. Le générateur d'horloge donne la fréquence de transmission ou vitesse de transmission. Du point de vue logiciel on accède à l'UART simplement en écrivant dans le registre d'émission et en lisant le registre de réception.

10.2.1 RS-232C

Historiquement, les interfaces séries étaient très simples d'un point de vue logiciel mais par contre plus compliquées d'un point de vue matériel. C'est le cas de l'interface RS-232C développée en 1969.

La couche physique du RS-232C à néanmoins évolué depuis 1969, par contre la couche 2 est toujours la même. On peut réaliser cette couche 2 avec un UART.

10.3 Interface SPI

Dix ans après la RS-232C, Motorola invente l'interface SPI en 1979. C'est un changement radical, avec une communication synchrone entre les émetteurs et les récepteurs. Ce changement permet d'aller beaucoup plus vite dans la transmission.

L'interface SPI utilise deux registres à décalage bouclés. Ceci permet de relire l'information transmise est ainsi vérifier que ce qui a été transmis est bien correcte. Pour écrire ou lire, on utilise le même principe que pour l'UART, c-à-d écrire dans le registre d'émission ou lire le registre de réception.

Il est possible de connecter plusieurs périphériques sur une interface SPI à l'aide de signaux sortant d'un GPIO et allant sur une entrée "chip-select" CS. Il faut juste sélectionner le bon CS depuis le code.

10.4 Interface I2C

Trois ans après la SPI, Philips invente l'interface I2C en 1982. Cette interface amène une innovation majeur qui est la réalisation d'une couche 3 logicielle permettant d'adresser 256 noeuds. On peut donc relier tous les périphériques d'une carte à l'aide d'une interface I2C. Ils sont connectés sur deux lignes communes SDA et SCL :

Les lignes, SDA et SCL, sont tirées vers l'alimentation positive (Vdd) à l'aide de deux résistance pull-up. Elles doivent être dimensionnées par rapport à la vitesse de transmission choisie. Pour sélectionner un périphérique en particulier, il faut connaitre son adresse I2C et la placer dans le registre d'adresse de l'interface.

10.5 Interface USB

L'interface série la plus récente que l'on trouve dans les microcontrôleurs est l'USB (Universial Serial Bus) développée en 1996. Cette interface à comme avantage majeur sur les autres l'utilisation d'une signalisation basse tension différentielle LVDS (Low Voltage Differential Signal). Par contre, cette interface est purement point à point c-à-d qu'elle ne relie qu'un noeud à un autre. De ce fait, elle ne remplace pas les précédentes.

L'interface USB est synchrone mais ne possède pas de ligne de clock. L'horloge est retrouvée à la réception grâce à une séquence d'entrainement de 8 bits. Une fois l'horloge synchronisée un packet ID (PID) est envoyé pour donner le type de message envoyé. Si le PID indique des données alors elles sont transmises. Finalement le paquet est terminé par une séquence "End of Packet" de trois bits.

D'un point de vue logiciel, l'interface USB s'accède grâce à un HAL ou par un pilote si il y a un système d'exploitation.

10.6 Comparaisons et choix d'une interface

Interface	Noeuds	Vitesse	Async/Sync	Duplex	Utilisation
RS-232C	1	115kHz	A	Full	Terminal
SPI	1 à 5	100MHz	S	Full	Données
I2C	256	1MHz	S	Half	Configuration
USB 2.0	1	480MHz	S	Half	External

TABLE 10.1 – Comparaisons entre les différentes interfaces séries

10.7 Cas du MSP430

Le MSP430 dispose de toutes les interfaces discutées précédemment. Il s'agit donc de déterminer les registres de configuration propres à chaque interface puis de les paramétrer correctement.

Exemple de configuration d'une interface série asynchrone, style RS-232C :

```
UCA0CTL0 |= UCPEN | UCPAR;  //parity enabled, even, one stop(default), 8-
    bit (default), async (default)
UCA0CTL1 |= UCSSEL_2;       //SMCLK
UCA0BR0 = 0x09;             //115200 bauds selon table du fabriquant
UCA0BR1 = 0x00;
UCA0MCTL = 0x02;            //115200 bauds selon table du fabriquant

UCA0IE |= UCRXIE | UCTXIE;  //Interrupt enable

P2DIR |= 0x01;              //P2.0 vers sortie

#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
  switch(__even_in_range(UCA0IV,4))
  {
    case  0 :    break;
    case  2 :    while (!(UCA0IFG & UCTXIFG));
                 UCA0TXBUF = UCA0RXBUF;
                 break;
```

```
        case  4 :    P2OUT ^= 0x01;      // toggle P2.0 avec exclusive-OR
                     break;
        default :    break;
      }
    }
```

L'interaction avec les registres de réception ou d'émission est effectuée par interruption dans ce cas ci. Le vecteur USCI_A0_VECTOR contient plusieurs sources d'interruptions possibles c'est pourquoi il faut les sélectionner à l'aide d'un switch case. Le "case 2" correspond à une réception et le "case 4" à une émission. Lors d'une réception on test si l'émission n'est pas active puis on copie le message reçu sur la sortie. C'est un relai qui allume une LED lorsque le message a été relayé.

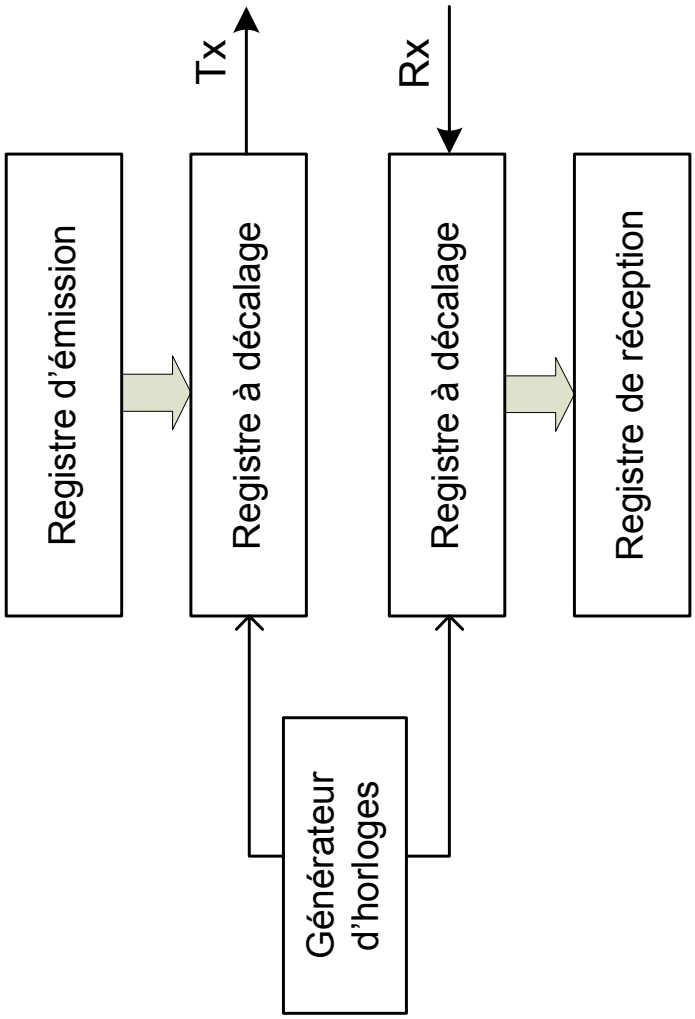

FIGURE 10.3 – Principe de fonctionnement d'une UART

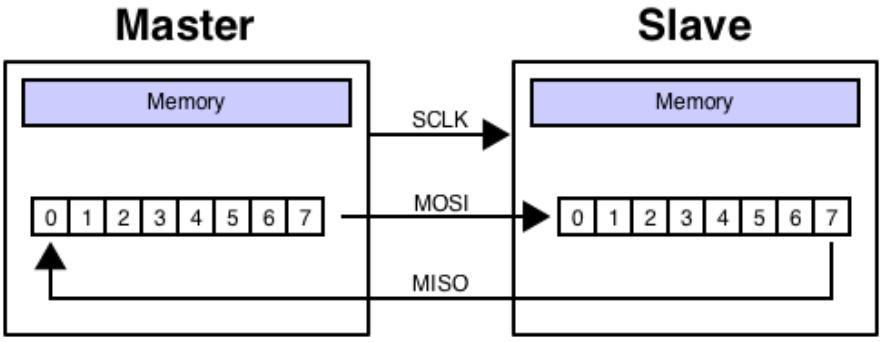

FIGURE 10.4 – Principe de fonctionnement d'une interface SPI

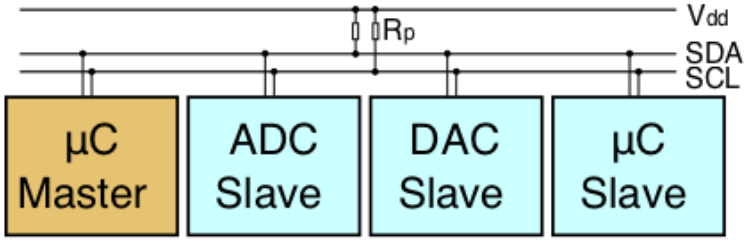

FIGURE 10.5 – Système avec un bus I2C

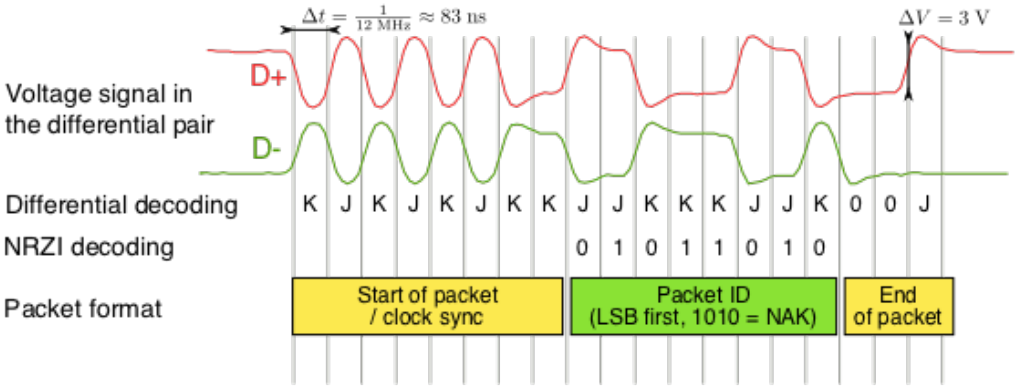

FIGURE 10.6 – Exemple de signal USB

Chapitre 11

AD et DA

Les Convertisseurs Analogique-Numérique (ADC pour *Analog to Digital Converter*) et Numérique-Analogique (DAC pour *Digital to Analog Converter*) sont des éléments importants de toute chaîne d'acquisition de donnée.
En effet, les opérations complexes de traitement de signal sont aujourd'hui exécutées par des circuits numériques, dont les microcontrôleurs font partie. Pour être traités par un microcontrôleur, les signaux analogiques doivent donc être convertis en nombres.
La figure 11.1 illustre une chaîne typique d'acquisition de donnée.

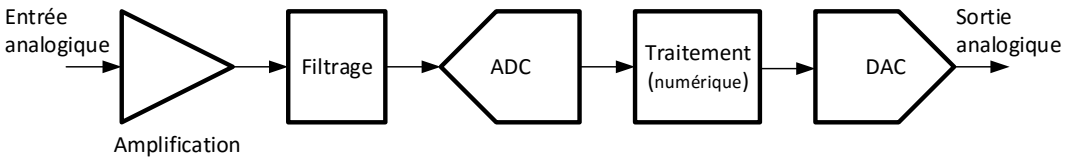

FIGURE 11.1 – Chaîne d'acquisition et traitement de signal

11.1 Convertisseur AD

L'ADC est un circuit dont la fonction est de convertir une grandeur physique V_{in} - en général une tension, parfois un courant - en un nombre N représenté sur plusieurs bits, "proportionnel" à cette grandeur physique. Le terme proportionnel est abusif dans la mesure où la fonction de transfert $N = f(V_{in})$ est une fonction en escalier, comme illustré à la figure 11.2.

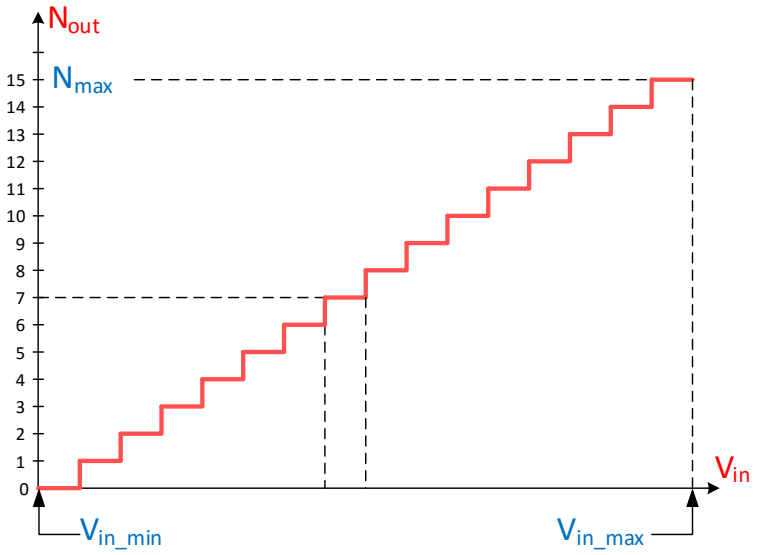

FIGURE 11.2 – Fonction de transfert d'un ADC idéal 4-bits

Les caractéristiques importantes de cette fonction de transfert sont :

- l'intervalle des tensions d'entrée $[V_{in\_min}, V_{in\_max}]$
- la résolution, égale à $log_2(N_{max})$
- les imperfections qui l'entachent, et qui sont :
  - offset
  - erreur de gain
  - "non linéarité" de la fonction

Il existe plusieurs méthodes pour convertir une grandeur physique en un nombre. Elles se différencient principalement par la résolution atteignable et la rapidité du processus de conversion, et aussi par leur complexité.
Dans les microcontrôleurs, le convertisseur AD le plus souvent rencontré est le convertisseur à approximations successives, dont l'avantage est d'offrir un bon compromis entre résolution et rapidité, pour une consommation de courant très faible.

11.2 Convertisseur à approximations successives

Ce type de convertisseur utilise un processus de dichotomie pour traduire progressivement une tension analogique en un nombre. Le temps de conversion est fonction du nombre de bits souhaités.
Le principe est illustré à la figure 11.3.

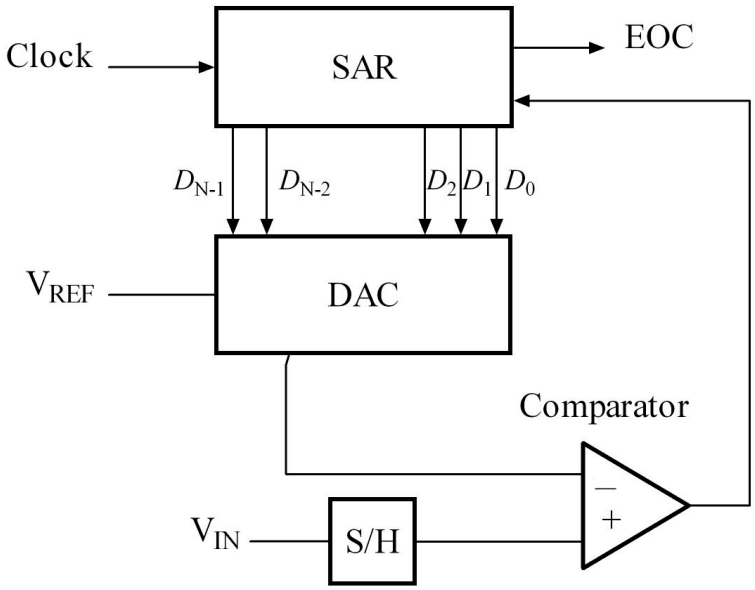

FIGURE 11.3 – Principe du convertisseur à approximations successives

Un séquenceur (généralement nommé SAR pour *Successive Approximation Register*), couplé à un convertisseur Digital-Analogique (DAC) génère une tension analogique, qui est comparée au signal à convertir. Le résultat de cette comparaison est alors introduit dans le SAR, qui va le prendre en compte, pour la suite du processus de dichotomie, jusqu'à complétion. La figure 11.4 illustre la chronologie du processus de conversion, dans le cas d'un ADC 4-bits.
Dans cet exemple, le DAC est capable de générer 16 tensions notées $x.FS$ (FS signifie *Full Scale*), avec $0 \leqslant x \leqslant 15$. Les valeurs minimale et maximale de ces tensions sont appelées *tensions de référence*.
Le test du MSB détermine si V_{in} est supérieur ou inférieur à $1/2.FS$. Dans l'exemple, dès lors que $V_{in} \leqslant 1/2.FS$, le MSB vaut 1. Pour tester le bit de poids inférieur (MSB-1), le séquenceur commande le DAC pour qu'il génère $3/4.FS$. Cette fois, $V_{in} \geqslant 3/4.FS$, donc (MSB-1) = 0 ; ceci signifie qu'il ne fallait pas ajouter $1/4.FS$ à $1/2.FS$ mais $1/8.FS$.
Le processus continue ainsi jusqu'à l'obtention du LSB, au rythme d'un bit par étape.

Le convertisseur réalise donc sa conversion en positionnant en premier le bit de poids fort (MSB) et en descendant progressivement jusqu'au LSB. Ces convertisseurs ont des résolutions d'une douzaine de bits

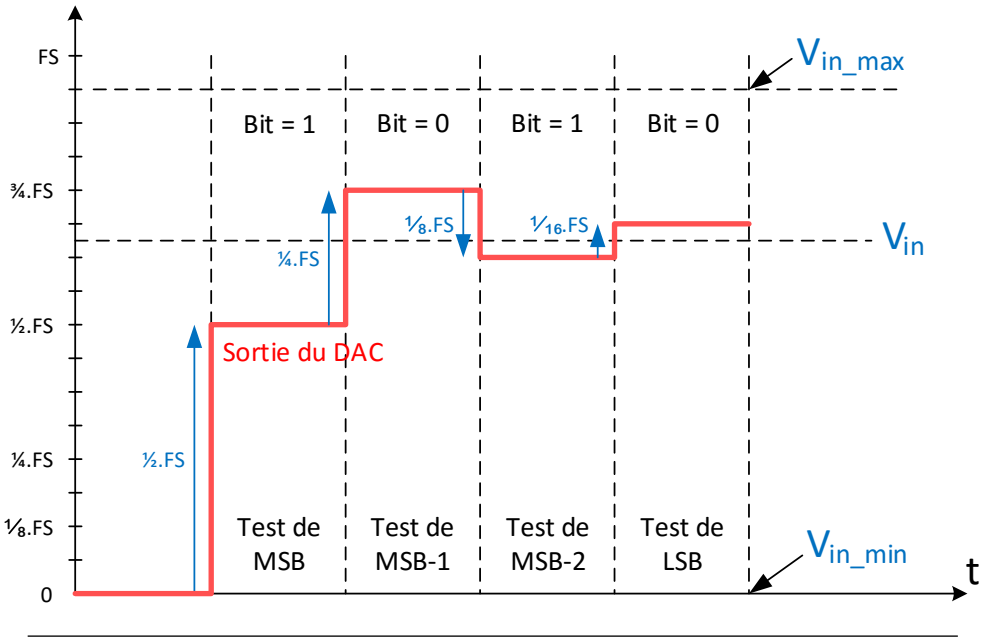

FIGURE 11.4 – Etapes de conversion d'un ADC 4-bits à approximations successives

environ, mais peuvent atteindre 16 bits au moyen de technologies spéciales.

Il est important de noter que la tension d'entrée doit être mémorisée pendant toute la durée du processus de conversion. Ceci est effectué au moyen d'un circuit auxiliaire appelé "Echantillonneur-Bloqueur" (*Sample and Hold*). Le terme "Echantillonneur" fait référence à l'instant auquel la tension est saisie ; "Bloqueur" fait référence à sa mémorisation.

Le convertisseur DAC peut être construit au moyen de :

- un diviseur de tension résistif couplé à des interrupteurs
- un diviseur de courant de type R-2R
- un diviseur de tension à capacités pondérées

Le détail de ces circuits sort du cadre de ce cours.

11.3 Cas du MSP430

Les microcontrôleurs MSP430 peuvent contenir plusieurs types de convertisseurs AD :

- AD 10-bits à approximations successives
- AD 12-bits à approximations successives
- AD 16-bits Sigma-Delta

11.3.1 Cas du MSP430F5529

Ce microcontrôleur contient un convertisseur AD 12 bits à approximations successives avec les caractéristiques suivantes.

- Plus de 200 kSPS : 200'000 échantillons par seconde
- Echantillonneur-Bloqueur avec période programmable, contrôlée par logiciel ou par timers
- Lancement des conversions par logiciel ou par timers
- Référence de tension interne sélectionnable par logiciel (1.5 V, 2.0 V, ou 2.5 V)
- Référence interne ou externe
- Jusqu'à 12 canaux d'entrée configurables individuellement
- Canaux d'entrée pour capteur de température interne, AVCC, et références externes
- Registres de stockage de 16 résultats de conversion

11.3.2 Schéma de l'ADC 12-bits

La figure 11.5 représente le schéma complet du périphérique ADC12, l'ADC 12-bits présent dans le MSP430F5529.

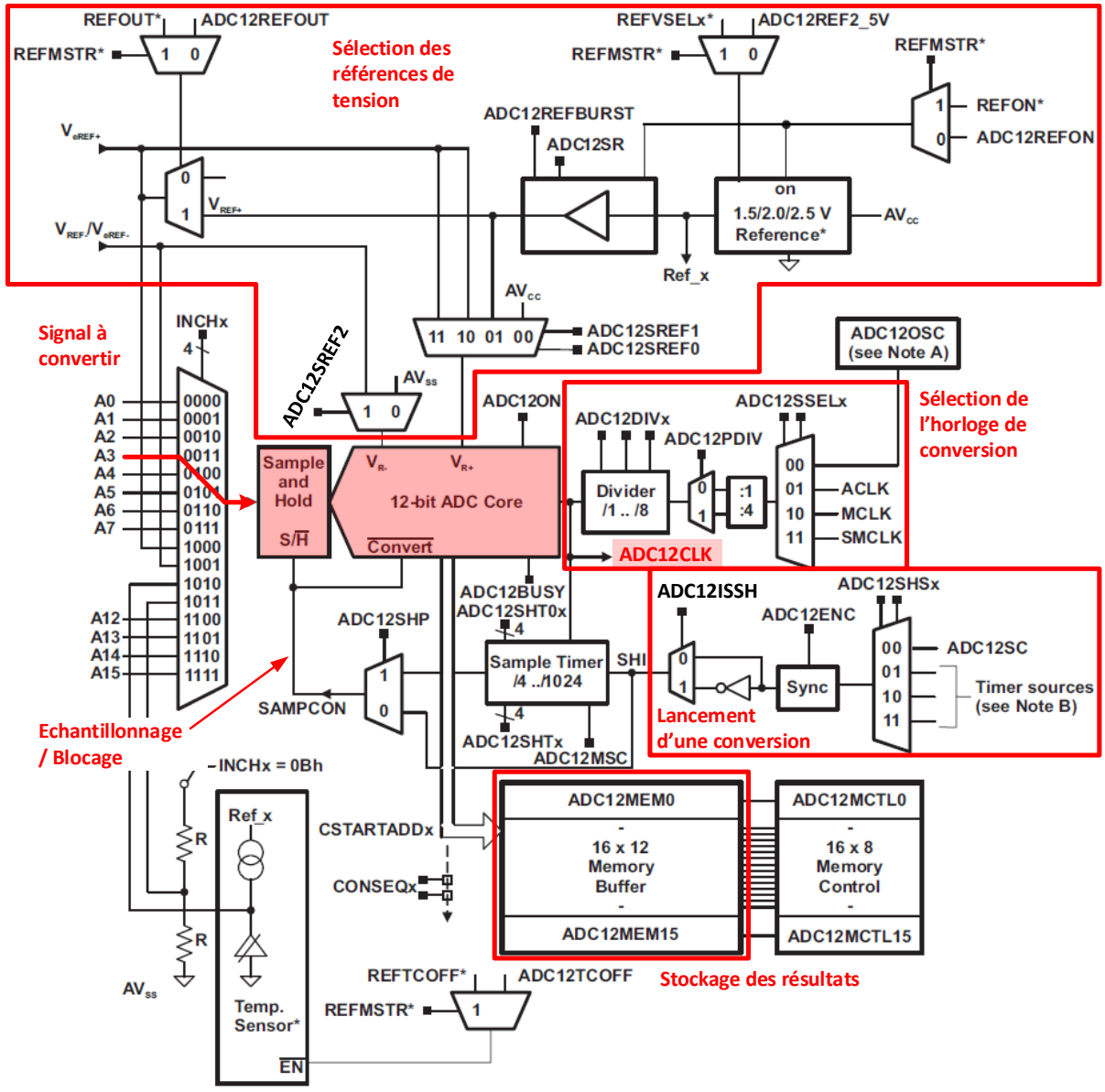

FIGURE 11.5 – Schéma du convertisseur ADC12 dans le MSP430F5529

On y trouve :

- le coeur de l'ADC, avec l'échantillonneur-bloqueur, au centre
- le bloc de sélection des références de tension
- le bloc de sélection de l'horloge de conversion, qui définit le rythme auquel les bits du résultat sont obtenus
- le bloc de lancement d'une conversion. Il est possible de lancer automatiquement des conversions
- une mémoire pouvant contenir les résultats de 16 conversions
- à gauche, le circuit de sélection du signal à convertir

Comme on l'a déjà vu avec d'autres périphériques, les réglages du convertisseur ADC12 se font par le logiciel, en positionnant les champs logiques apparaissant sur le schéma, qui sont accessibles dans des registres de contrôle.

11.3.3 Echantillonnage et conversion

Le processus de conversion commence par l'échantillonnage du signal d'entrée, puis continue avec la conversion par approximations successives. Le processus est lancé sur le flanc montant du signal SHI, comme indiqué à la figure 11.6.

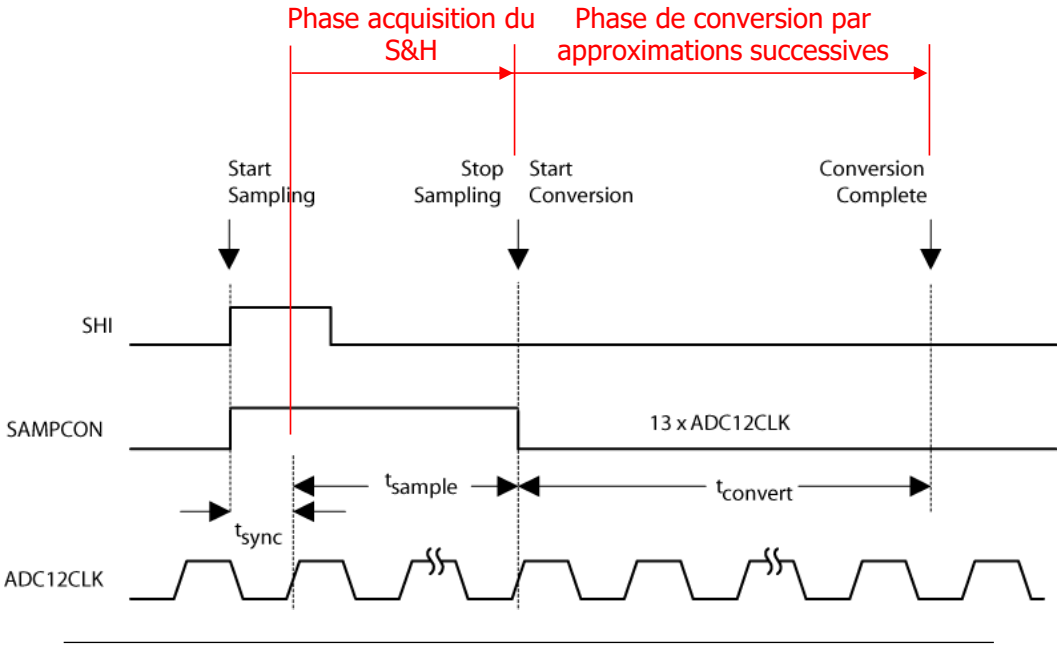

FIGURE 11.6 – Chronogramme du processus d'échantillonnage et conversion (ADC12SHP = 1)

Le signal SAMPCON (*Sampling Control*) joue un rôle particulier car son flanc montant détermine le début de la saisie du signal d'entrée, qui dure tant que $SAMPCON = 1$. La conversion à proprement parler démarre au flanc descendant de SAMPCON et dure 13 périodes du signal ADC12CLK pour une conversion sur 12 bits. La durée du signal SAMPCON doit être réglable pour pouvoir ajuster le temps d'échantillonnage à la résistance interne de la source d'où provient V_{in}. En effet, l'entrée de l'échantillonneur-bloqueur (le *Sample and Hold*) est une capacité ; sa charge est soumise à une constante de temps qui dépend de la résistance de la source.

11.3.4 Sélection de l'horloge de conversion

L'horloge de conversion ADC12CLK rythme le processus de conversion par approximations successives. Elle détermine donc le temps alloué au calcul de chaque bit, du MSB au LSB.

Lorsque l'utilisateur souhaite se simplifier la vie (ADC12SHP = 1), elle détermine aussi la durée de l'échantillonnage, via un timer interne à l'ADC[1]. Ce timer, noté *Sample Timer* sur la figure 11.5 peut générer une durée d'échantillonnage comprise entre 4 et 1024 cycles du signal ADC12CLK.

Les sources d'horloge usuelles (ACLK, SMCLK et MCLK) peuvent être utilisées. Par défaut, un autre signal est disponible, noté ADC12OSC, et dont la fréquence est d'environ 5 MHz.

11.3.5 Lancement d'une conversion

Plusieurs méthodes sont envisageables :

- lancement par logiciel, en activant le bit ADC12SC (*Start Conversion*)
- lancement automatique par un timer

1. La durée d'échantillonnage peut aussi être contrôlée par le programme. Dans ce cas (ADC12SHP = 0) le signal SHI contrôle directement la durée d'échantillonnage et la conversion. SHI est alors généré par le programme ou par un des timers à usage général

11.3.6 Sélection des références de tension

Les références de tension déterminent l'intervalle dans lequel la tension à convertir doit se trouver. La borne inférieure peut être égale à GND, mais pas nécessairement. Par contre, la borne supérieure est toujours inférieure à la tension d'alimentation VCC. La raison est que les circuits internes au convertisseur ont besoin d'une marge entre V_{REF_Sup} et VCC pour pouvoir fonctionner correctement. Le résultat N_{ADC} de la conversion est donné par la formule suivante :

$$N_{ADC} = 4095.\frac{V_{in}-V_{Ref-}}{V_{Ref+}-V_{Ref-}}$$

11.3.7 Stockage des résultats

Lorsqu'une conversion est terminée, l'ADC peut émettre une requête d'interruption. Toutefois, dans de nombreuses applications, il est nécessaire de convertir plusieurs tensions différentes. Les raisons peuvent être :

- les signaux générés par plusieurs capteurs, et connectées sur l'ADC via ses différentes entrées (A0 à A15) doivent être convertis en nombres ;
- plusieurs valeurs successives d'une même tension doivent être converties, pour pouvoir calculer une moyenne ou y appliquer une fonction de filtrage numérique ;
- on ne souhaite pas surcharger le CPU avec des interruptions trop fréquentes ; etc...

Pour résoudre ce genre de situation, l'ADC dispose d'une mémoire pouvant contenir jusqu'à 16 résultats de conversion. Il est possible de définir des séquences de conversion, et de faire en sorte qu'une interruption ne soit émise qu'à la fin de la séquence.
Chacun des 16 registres de stockage, notés ADC12MEMx est associé à un registre de contrôle noté ADC12MCTLx (x = 0..15).

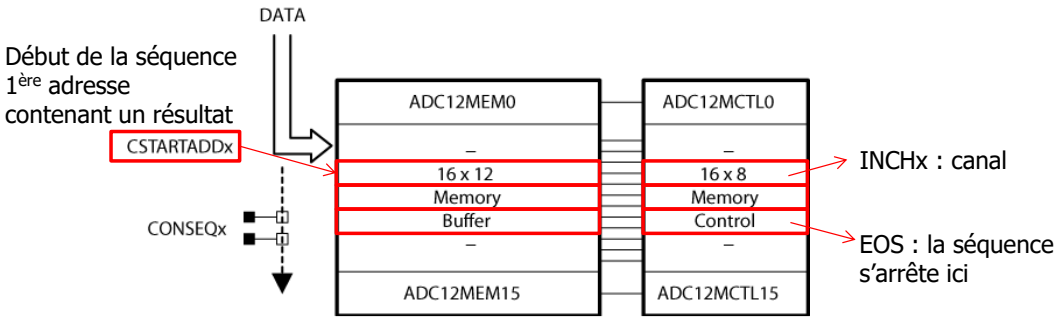

FIGURE 11.7 – Organisation des registres de stockage et leur contrôle

Un champ noté ADC12CSTARTADDx contient l'adresse du registre dans lequel se trouvera le résultat de la conversion. SI plusieurs conversions doivent être enchaînées, ce champ indique dans quel registre se trouve le premier résultat, les autres se trouvant dans les registres ADC12MEMx suivants. Pour contrôler les conversions, chaque registre ADC12MCTLx contient 3 champs :

- INCHx : le numéro de l'entrée à convertir ;
- SREFx : le jeu de tensions de référence à utiliser ;
- EOS : le registre ADC12MCTLy contenant ce bit à '1' marque la fin de la séquence.

Ainsi, chaque conversion d'une séquence dispose de ses paramètres propres. Il est bien entendu possible de convertir le signal d'une seule entrée, auquel cas il s'agit d'une séquence d'une seule conversion.

Lors de l'écriture d'un résultat de conversion dans un registre, une requête d'interruption est émise. Dans le cas d'une séquence de conversions, il suffit de n'autoriser que celle correspondant à la fin de la séquence.

11.3.8 Registres de contrôle du convertisseur ADC12

Ils sont nommés ADC12CTLx et sont au nombre de 3. Dans les tableaux suivants, on donne uniquement la signification des champs de contrôle les plus importants.

15	14	13	12	11	10	9	8
ADC12SHT1x				ADC12SHT0x			
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ADC12MSC	ADC12REF2_5V	ADC12REFON	ADC12ON	ADC12OVIE	ADC12TOVIE	ADC12ENC	ADC12SC
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

FIGURE 11.8 – ADC12CTL0

Champ	Valeur	Description
ADC12SHT1		Durée d'échantillonnage pour les conversions contrôlées par ADC12MCTL8 à ADC12MCTL15
ADC12SHT0		Durée d'échantillonnage pour les conversions contrôlées par ADC12MCTL0 à ADC12MCTL7
	0000b	4 cycles de ADC12CLK
	0001b	8 cycles de ADC12CLK
	0010b	16 cycles de ADC12CLK
	0011b	32 cycles de ADC12CLK
	0100b	64 cycles de ADC12CLK
	0101b	96 cycles de ADC12CLK
	0110b	128 cycles de ADC12CLK
	0111b	192 cycles de ADC12CLK
	1000b	256 cycles de ADC12CLK
	1001b	384 cycles de ADC12CLK
	1010b	512 cycles de ADC12CLK
	1011b	768 cycles de ADC12CLK
	11xxb	1024 cycles de ADC12CLK
ADC12MSC	-	Permet d'enchaîner des conversions multiples
ADC12ON	-	Mise en marche du convertisseur
	0	Convertisseur arrêté
	1	Convertisseur en marche
ADC12ENC	-	Lancement du contrôleur de conversions
	0	Contrôleur arrêté
	1	Contrôleur prêt
ADC12SC	-	Lancement d'une conversion
	1	Start

TABLE 11.1 – ADC12CTL0

15	14	13	12	11	10	9	8
ADC12CSTARTADDx				ADC12SHSx		ADC12SHP	ADC12ISSH
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ADC12DIVx			ADC12SSELx		ADC12CONSEQx		ADC12BUSY
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)

FIGURE 11.9 – ADC12CTL1

Champ	Valeur	Description
ADC12CSTARTADD	-	Adresse du registre de contrôle de la conversion
		ou de la première conversion, dans le cas d'une séquence
ADC12SHS		Sélection du signal de conversion
	00b	bit ADC12SC (lancement par logiciel)
	01b	timer (consulter la fiche technique du MCU)
	10b	timer (consulter la fiche technique du MCU)
	11b	timer (consulter la fiche technique du MCU)
ADC12SHP		Contrôle de la durée d'échantillonnage
	0	SAMPCON égale le signal de conversion
	1	Le niveau haut de SAMPCON est fixé par le timer de conversion
		voir le champ ADC12SHT0 ou ADC12SHT1 dans ADC12CTL0
ADC12ISSH	-	Inversion du signal de conversion (11.5)
	0	Pas d'inversion
	1	Inversion
ADC12DIV	-	Prédivision de l'horloge de conversion ADC12CLK
	000b	Division par 1
	001b	Division par 2
	010b	Division par 3
	...	...
	110b	Division par 7
	111b	Division par 8
ADC12SSEL	-	Sélection de l'horloge de conversion ADC12CLK
	00b	ADC12OSC
	01b	ACLK
	10b	MCLK
	11b	SMCLK
ADC12CONSEQ	-	Mode de conversion
	00b	Conversion unique sur un canal unique
	01b	Séquence de conversions
	10b	Canal unique en répétition
	11b	Séquence en répétition
ADC12BUSY	-	Indicateur de conversion en cours
	0	Pas de conversion en cours
	1	Une conversion en cours

TABLE 11.2 – ADC12CTL1

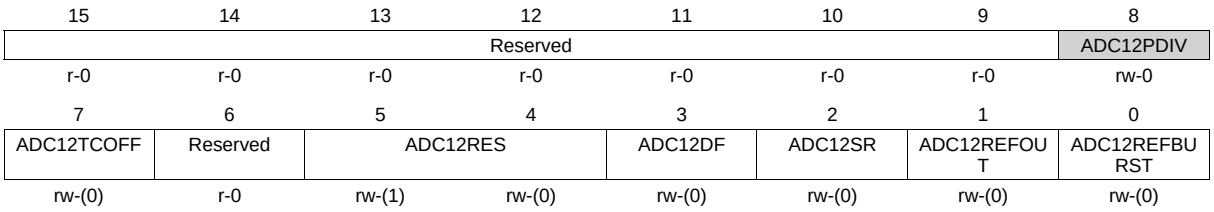

FIGURE 11.10 – ADC12CTL2

Champ	Valeur	Description
ADC12PDIV	-	Prédivision de l'horloge de conversion ADC12CLK
		en plus de la prédivision contrôlée par ADC12DIV
	0	Prédivision par 1
	1	Prédivision par 4
ADC12RES		Spécifie le nombre de bits du résultat de conversion
	00b	8 bits (9 cycles de ADC12CLK)
	01b	10 bits (11 cycles de ADC12CLK)
	10b	12 bits (13 cycles de ADC12CLK)
	11b	Réservé
ADC12DF		Format du résultat de conversion sur 16-bits
	0	Binaire non signé. -VREF = 0000h, +VREF = 0FFFh
		Le résultat est justifié à droite (vers les poids faibles)
	1	Binaire signé. -VREF = 8000h, +VREF = 7FF0h
		Le résultat est justifié à gauche (vers les poids forts)

TABLE 11.3 – ADC12CTL2

11.3.9 Exemples de configuration

Exemple 1

L'entrée A0 est convertie sur ordre logiciel. La référence est AVcc. ADC12SC est remis à 0 automatiquement à la fin de la conversion. Le résultat est dans ADC12MEM0. Dans la boucle principale, le CPU attend en LPM0 jusqu'à l'interruption de ADC12. Au retour de l'interruption, le CPU ne retourne pas en LPM0 pour pouvoir lancer une nouvelle conversion. Si A0 > 0.5*AVcc, P1.0 est mis à '1', sinon '0'.

```
//******************************************************************
// MSP430F552x Demo - MSP430F55xx_adc_01
// Bhargavi Nisarga, Texas Instruments Inc.
//******************************************************************
#include <msp430.h>

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stoppe le WDT
  ADC12CTL0 = ADC12SHT02 + ADC12ON;         // Temps d'échant., ADC12 on
  ADC12CTL1 = ADC12SHP;                     // Timer d'échant. (SHP = 1)
  ADC12IE = 0x01;                           // Interruption autorisée
  ADC12CTL0 |= ADC12ENC;                    // Armement de l'ADC
  P6SEL |= 0x01;                            // P6.0 mis en connection avec A0
  P1DIR |= 0x01;                            // P1.0 en sortie

  while (1)
  {
    ADC12CTL0 |= ADC12SC;                   // Lancer échant./conversion
    __bis_SR_register(LPM0_bits + GIE);     // LPM0
  }
}

#pragma vector = ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
  switch(__even_in_range(ADC12IV,34))
  {
  case  0 : break;                          // Vecteur  0 :  Pas d'interruption
  case  2 : break;                          // Vecteur  2 :  ADC overflow
  case  4 : break;                          // Vecteur  4 :  ADC timing
      overflow
  case  6 :                                 // Vecteur  6 :  ADC12IFG0
    if (ADC12MEM0 >= 0x7ff)                 // ADC12MEM = A0 > 0.5*AVcc?
      P1OUT |= BIT0;                        // P1.0 = 1
    else
      P1OUT &= ~BIT0;                       // P1.0 = 0
    __bic_SR_register_on_exit(LPM0_bits);// le CPU ne retournera pas en
       LPM0
  case  8 : break;                          // Vecteur  8 :  ADC12IFG1
  case 10 : break;                          // Vecteur  10 : ADC12IFG2
  case 12 : break;                          // Vecteur  12 : ADC12IFG3
  case 14 : break;                          // Vecteur  14 : ADC12IFG4
  ...
  case 16 à 28 omis pour des raisons de mise en page
  ...
  case 30 : break;                          // Vecteur  30 : ADC12IFG12
  case 32 : break;                          // Vecteur  32 : ADC12IFG13
  case 34 : break;                          // Vecteur  34 : ADC12IFG14
  default : break;
  }
}
```

Exemple 2

Les entrées A0 à A3 sont converties en séquence. Les références sont AVcc. Les résultats sont dans ADC12MEM0-3. Dans la boucle principale, le CPU attend en LPM4 jusqu'à l'interruption.

```
//******************************************************************
// MSP430F552x Demo - MSP430F55xx_adc_09
// Bhargavi Nisarga, Texas Instruments Inc.
//******************************************************************

#include <msp430.h>

volatile unsigned int results[4];          // tableau pour stocker les
    résultats

int main(void)
{
  WDTCTL = WDTPW+WDTHOLD;                  // Stoppe le WDT
  P6SEL = 0x0F;                            // P6.0-3 connectés avec A0-A3
  ADC12CTL0 = ADC12ON+ADC12MSC+ADC12SHT0_2;// Temps d'échant., ADC12 on
  ADC12CTL1 = ADC12SHP + ADC12CONSEQ_1;    // Timer d'échant., séq. simple
  ADC12MCTL0 = ADC12INCH_0;                // ref+ = AVcc, channel = A0
  ADC12MCTL1 = ADC12INCH_1;                // idem channel A1
  ADC12MCTL2 = ADC12INCH_2;                // idem channel A2
  ADC12MCTL3 = ADC12INCH_3+ADC12EOS;       // idem channel A3, fin séquence
  ADC12IE = 0x08;                          // Autorisation ADC12IFG.3
  ADC12CTL0 |= ADC12ENC;                   // Armement de l'ADC

  while(1)
  {
    ADC12CTL0 |= ADC12SC;                  // Lancer échant./conversion
    __bis_SR_register(LPM4_bits + GIE);    // LPM4, autorisation interrupts
  }
}

#pragma vector=ADC12_VECTOR
__interrupt void ADC12ISR (void)
{
    switch(__even_in_range(ADC12IV,34))
  {
  case  0 : break;                         // Vecteur  0 :  No interrupt
  case  2 : break;                         // Vecteur  2 :  ADC overflow
  case  4 : break;                         // Vecteur  4 :  ADC timing
      overflow
  case  6 : break;                         // Vecteur  6 :  ADC12IFG0
  case  8 : break;                         // Vecteur  8 :  ADC12IFG1
  case 10 : break;                         // Vecteur  10 :  ADC12IFG2
  case 12 :                                // Vecteur  12 :  ADC12IFG3
    results[0] = ADC12MEM0;                // sauver résultat, IFG remis à 0
    results[1] = ADC12MEM1;                // sauver résultat, IFG remis à 0
    results[2] = ADC12MEM2;                // sauver résultat, IFG remis à 0
    results[3] = ADC12MEM3;                // sauver résultat, IFG remis à 0
    __bic_SR_register_on_exit(LPM4_bits);// le CPU ne retournera pas en
        LPM4
  case 14 : break;                         // Vecteur  14 :  ADC12IFG4
  // case 16 à 32 omis pour des raisons de mise en page
  case 34 : break;                          // Vector 34 :  ADC12IFG14
  default : break;
  }
}
```

Annexe A

Démonstrations algébriques

De manière générale, on peut écrire un nombre entier non-signé sous la forme suivante :

$$\begin{aligned} A_{10} &= \sum_{i=0}^{n-1} a_i 2^i = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_1 2^1 + a_0 2^0 \\ avec &: a_i \in \{0,1\} \\ et &: A_{10} \in [0...2^n - 1] \end{aligned} \tag{A.1}$$

Un nombre entier signé, en complément à deux, peut être écrit sous cette forme :

$$A_{10} = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i = -a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_1 2^1 + a_0 2^0 \tag{A.2}$$

A.1 Changement de signe

$$\begin{aligned} -A_{10} = \; & -\Bigg(\underbrace{-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i}_{A_{10}}\Bigg) \; = \; \underbrace{-2^{n-1} + \underbrace{\sum_{i=0}^{n-2} 2^i + 1}_{2^{n-1}}}_{0} + a_{n-1}2^{n-1} - \sum_{i=0}^{n-2} a_i 2^i \\ = \; & -(1 - a_{n-1})2^{n-1} + \sum_{i=0}^{n-2}(1 - a_i)2^i + 1 \; = \; -\bar{a}_{n-1}2^{n-1} + \sum_{i=0}^{n-2} \bar{a}_i 2^i + 1 \\ = \; & \bar{A} + 1 \end{aligned} \tag{A.3}$$

A.2 Extension d'un nombre signé

On cherche à trouver les bits b_i qui permettent d'étendre un nombre négatif ($a_{n-1} = 1$) en complément à deux de n bits à m bits :

$$a_{n-1}a_{n-2}...a_0 \Rightarrow b_{m-1}b_{m-2}...b_{n-1}a_{n-2}...a_0$$

Exemple :

$$m = 8, n = 4 : a_3a_2a_1a_0 = b_7b_6b_5b_4b_3a_2a_1a_0$$

Avec a_3 le bit de signe original et b_7 le bit de signe du nombre étendu. a_3 se déplace donc en première position b_7 pour conserver le bit de signe. Comme ils valent 1 on les simplifies et la notation peut s'écrire de manière générale :

$$\begin{aligned} -2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \; &= \; -2^{m-1} + \sum_{i=n-1}^{m-2} b_i 2^i + \sum_{i=0}^{n-2} a_i 2^i \quad m > n \\ -2^{n-1} \; &= \; -2^{m-1} + \sum_{i=n-1}^{m-2} b_i 2^i \\ 2^{m-1} \; &= \; 2^{n-1} + \sum_{i=n-1}^{m-2} b_i 2^i \end{aligned} \tag{A.4}$$

Cette égalité est toujours vraie si est seulement si les b_i sont égaux à 1 et quelque soit m, n (m > n). Exemples :

$$m = 4, n = 2 : \quad 2^{4-1} = 2^{2-1} + b_1 2^1 + b_2 2^2 \Leftrightarrow 8 = 2 + 1 \cdot 2 + 1 \cdot 4$$

$$m = 8, n = 4 : \quad 2^{8-1} = 2^{4-1} + b_3 2^3 + b_4 2^4 + b_5 2^5 + b_6 2^6 \Leftrightarrow 128 = 8 + 1 \cdot 8 + 1 \cdot 16 + 1 \cdot 32 + 1 \cdot 64$$

A.3 Multiplication de nombres en complément à deux

La multiplication en colonnes classique fonctionne en complément à 2 mais elle nécessite une correction à la fin du calcul en fonction du signe des opérandes pour retrouver le résultat correct. Soit m et r, deux entier sur n bits :

$$\begin{aligned} +m \equiv m \quad & + r \equiv r \\ -m \equiv 2^n - m \quad & - r \equiv 2^n - r \\ (+m) \times (+r) &= mr \\ (-m) \times (+r) &= 2^n r - mr \\ (+m) \times (-r) &= 2^n m - mr \\ (-m) \times (-r) &= 2^n 2^n - 2^n m - 2^n r + rm \end{aligned} \tag{A.5}$$

Le facteur $2^n 2^n$ disparait par arithmétique modulo. Il reste néanmoins une correction à effectuer qui est différente pour chaque cas. Cette différence de temps de calcul en fonction du signe est gênante mais peut être évitée grâce à l'algorithme de Booth (1950) [1]. L'algorithme consiste en la transformation suivante :

$$\begin{aligned} A \cdot B = \; & B\Bigg(\underbrace{-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i}_{A_{10}} \Bigg) = B\Bigg(-a_{n-1}2^{n-1} + 2\sum_{i=0}^{n-2} a_i 2^i - \sum_{i=0}^{n-2} a_i 2^i \Bigg) \\ = \; & B\Bigg(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^{i+1} - \sum_{i=0}^{n-2} a_i 2^i \Bigg) = B\Bigg(\sum_{i=1}^{n-1} a_{i-1}2^i - \sum_{i=0}^{n-1} a_i 2^i \Bigg) \\ = \; & B\Bigg(\sum_{i=1}^{n-1} (a_{i-1} - a_i)2^i - a_0 2^0 \Bigg) \end{aligned} \tag{A.6}$$

Et comme le terme a_{-1} peut être considéré nul car en dehors du nombre alors :

$$A \cdot B = \sum_{i=0}^{n-1} B(a_{i-1} - a_i)2^i \tag{A.7}$$

On peut montrer numériquement que le résultat de la multiplication est correct indépendamment du signe de A et de B [1].

L'algorithme de Booth peut être étendu pour inclure des séquences de "1" grâce à la même propriété utilisée ci-dessus qui est que tout "1" logique peut s'écrire $a_i 2^i = a_i 2^{i+1} - a_i 2^i$ et donc toute séquence de "1" peut s'écrire $a_i 2^{i+k} - a_i 2^i$ (avec k la longueur de la séquence de 1). Ceci permet de réduire le nombre de multiplications à effectuer. Prenons un exemple pour bien comprendre :

$$A \cdot B = A \cdot \text{"00111110"} = A \cdot (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = A \cdot 62$$

$$A \cdot B = A \cdot \text{"00111110"} = A \cdot (2^6 - 2^1) = A \cdot (64 - 2)$$

On passe donc de 5 à 2 termes à calculer.

La réalisation de cet algorithme est simplifiée si on parcours le multiplicateur de gauche à droite en faisant une addition du multiplicande lorsqu'on rencontre "01" (début de séquence) et une soustraction du multiplicande lorsqu'on rencontre "10" (fin de séquence) avec les décalages appropriés. On peut aussi le faire dans l'autre sens avec début à "10" et fin à "01". Cet algorithme peut être encore étendu en utilisant le concept du codage redondant. Par exemple pour le cas ci-dessus on considère deux bits à chaque itération. On peut aussi en considérer trois ou même quatre à chaque itération ce qui donne plus de flexibilité pour coder avantageusement le multiplicateur.

Bibliographie

[1] Andrew D. Booth, *A Signed Binary Multiplication Technique*, Journal of Mechanical and Applied Mathematics, 1951