Le micro contrôleur 68HC11 et son assembleur : notions de base

S. Cauët et O. Bachelier

mars 2003

1 *GENERALITES SUR LES MICRO CONTRÔLEURS*

1.1 Pourquoi utiliser un micro contrôleur ?

Un micro contrôleur (μC) est un circuit intégré qui, en lui-même, n'a pas de fonction spécifique ou particulière dédiée. On ne peut pas pour autant dire qu'il ne sait rien faire car il dispose d'un ensemble de fonctions prédéfinies appelé jeu d'instructions. Ce dernier permet au μC d'effectuer un certain nombre d'actions (ET logique, OU logique, additions d'entiers...) principalement sur sa mémoire. En effet, le μC manipule des données qui sont stockées dans des zones de mémoire.

Pour mettre en oeuvre un μC, il est donc nécessaire d'écrire un programme et de l'y implanter afin que le μC puisse exécuter les instructions composant le programme. Un programme est donc une séquence d'instructions exécutées l'une après l'autre. Pour écrire un programme correct, il convient de savoir quel est le langage compris par le μC (nous y reviendrons plus tard).

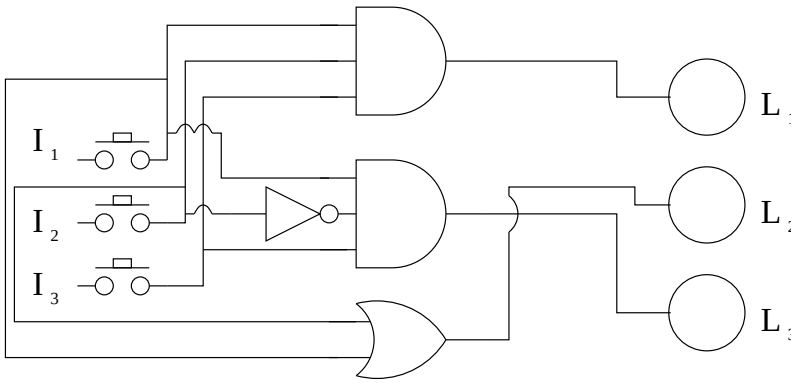
Pour comprendre l'utilité d'un μC, on peut considérer un petit exemple. Soient trois interrupteurs I_1, I_2, I_3 et trois lampes L_1, L_3 et L_3.

Problème 1 : K_i allume L_i, $\forall i \in \{1;2;3\}$.

Solution 1 : un simple petit montage électrique suffit.

Problème 2 : les trois interrupteurs ensemble allument L_1, Si seuls I_1 et I_3 sont actionnés, L_3 s'allume. Si I_1 ou I_2 sont actionnés, L_2 doit s'allumer.

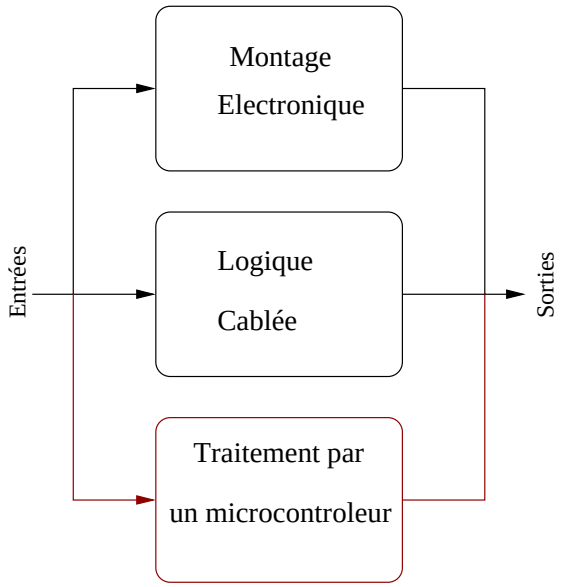
Solution 2 : Il suffit de mettre en place une logique câblée avec des portes comme celle de la figure ci-dessous.



Problème 3 : si les trois interrupteurs sont actionnés, L_1 s'allume pendant cinq secondes puis se met à clignoter. Si seuls I_1 et I_3 (mais pas I_2) sont actionnés, L_1 s'allume et L_2 clignote pendant 10s. Si I_1 et I_2 sont actionnés, L_1 et L_3 clignotent en phase alors que L_2 clignote en opposition.

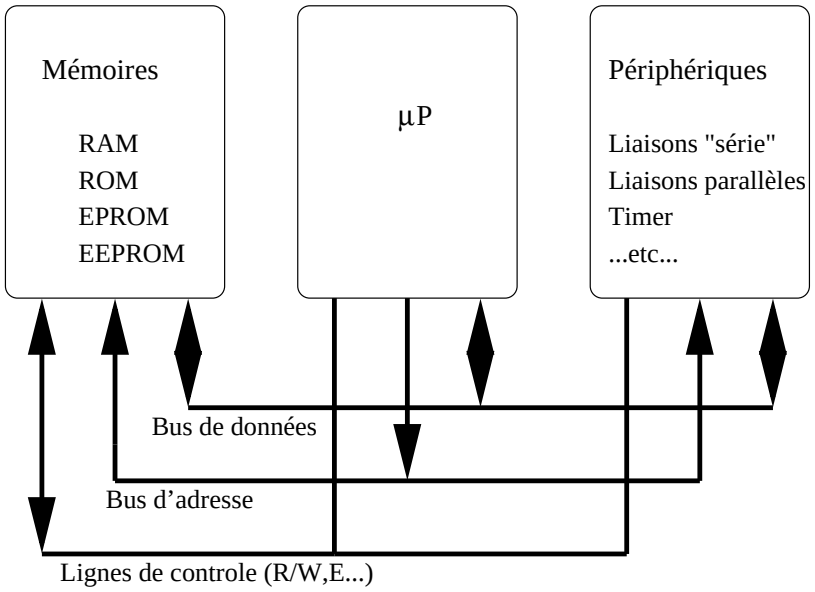
Solution 3 : pour un tel problème, il est déraisonnable d'envisager un montage électrique ou encore une logique câblée. En revanche, il est tout à fait judicieux de programmer un micro contrôleur à cet effet.

Sur la figure 2, on voit quelles sont les possibilités pour construire une fonction Entrées/Sorties. Le traitement par μC correspond à des fonctions sophistiquées.



1.2 Architecture d'un microcontrôleur

L'architecture d'un μC est résumée par la figure ci-après.



On constate que dans un μC, un microprocesseur (μP) fonctionne en interaction avec des zones de mémoire et un certain nombre de périphériques par l'intermédiaire de lignes de commande ou de bus.

Tout ce fonctionnement est cadencé par une **horloge** et chaque action effectuée à l'intérieur du μC requiert un certain nombre de cycles d'horloge.

Toutes ces entités sont regroupées en un seule circuit intégré (une puce). Le rôle de quelques unes de ces entités est détaillé ci-dessus.

1.2.1 Le rôle du μP

Sa seule fonction est de prendre des données provenant de périphériques ou de la mémoire, de leur faire subir un traitement, puis de les renvoyer sur des périphériques ou dans la mémoire. Il se compose plus précisément des éléments suivants :

1. une **A.L.U. ou Unité Arithmétique et Logique :** c'est un organe qui permet de faire des opérations arithmétiques et logiques élémentaires sur les données manipulées par le μP (addition, soustraction,multpilications, divisions, Et logique, OU logique, ...).
2. **des registres de travail :** ce sont des zones de mémoire spécifiques dont le μP dispose pour pouvoir exécuter les diverses instructions (ex : opérations élémentaires de l'A.L.U.)
3. **la logique de contrôle :** c'est un organe du μP qui lui permet de gérer les données circulant sur les lignes de contrôle.

4. **la logique d'adresse :** elle permet au μP de gérer les informations à émettre sur le bus d'adresse.
5. **le registre d'instruction décodeur :** c'est lui qui doit interpréter les différentes instructions du programme et renseigner ainsi l'unité de commande.
6. **l'unité de commande :** elle contrôle l'ensemble du fonctionnement du μP et active au bon moment les différents éléments.

1.2.2 Le rôle des bus et des lignes

Le bus d'adresse est un ensemble de lignes électriques (16 pour le 68HC11) permettant de faire circuler sur chacune des lignes une information booléenne (ou binaire) c'est-à-dire de type 0 ou 1 *(On parle de bit (abréviation de "Binary Digit"). Un octet se définit par une information de 8 bits et un "mot" par une information de 16 bits).*

Sur ce bus, peuvent donc circuler 2^{16} informations différentes qui sont en fait les différentes adresses. Une adresse est un nombre correspondant à un octet de la mémoire où :

- soit se trouve une donnée que le μP doit récupérer ;
- soit ce dernier doit envoyer une donnée.

Le **bus de données** est un ensemble de lignes électriques (8 pour le 68HC11) sur lequel circulent les informations "utiles" que le μP manipule (des nombres, des codes de caractères,...) autres que les adresses.

Les lignes de contrôle servent entre autres à spécifier si les données sont lues (R) à partir de la mémoire ou si elles sont écrites en mémoire (W).

1.2.3 Les différentes mémoires

Il existe dans un μC, des mémoires de natures différentes :

- La RAM (Read Access Memory) : c'est une zone de mémoire dans laquelle le μP peut lire ou écrire à tout instant de l'exécution d'un programme. On y trouve les données utiles à l'exécution de ce programme telles que les variables ;
- La ROM (Read Only Memory) : c'est une zone de mémoire dans laquelle le μP ne peut que lire. On peut y trouver des informations prédéfinies sur le μC qui ne peuvent ni ne doivent être modifiées (telles que le programme dans la plupart des applications industrielles) ;

- L'EPROM (Electrical Programmable Read Only Memory) : c'est une ROM dans laquelle on peut écrire en dehors de l'exécution par un procédé bien particulier et qui ne pourra plus être modifiée. C'est dans ce bloc-mémoire que l'on écrit le programme à exécuter. Si on utilise une EPROM, cela signifie que le μC est entièrement et définitivement dédié à l'application correspondant au programme (il n'y en a pas dans le 68HC11 E2 qui sera utilisé en TP).

- l'EEPROM (Erasable Electrical Programmable Only Memory) : elle est comparable à l'EPROM mais on peut la reprogrammer c'est-à-dire renouveler les instructions qui s'y trouvent, ceci toujours en dehors de l'exécution. En pratique, le programme sera "chargé" dans l'EEPROM par un logiciel. C'est le type de ROM présent sur le 68HC811E2.

1.2.4 Les périphériques

Ce sont des organes du μC qui permettent d'étendre les fonctions de base.Ils sont capables d'échanger des informations binaires avec le μP.

Parmi eux, on trouve entre autres

- le **Timer** qui utilise l'horloge afin de permettre au μP de gérer des processus qui font intervenir le temps avec précision ;

- les **périphériques de liaison** (série ou parallèle) qui permettent au μC d'échanger des données avec l'environnement extérieur (captuers, actionneurs) ;

- dans le cadre plus général de cet échange avec l'extérieur, les **CAN** (Convertisseurs Analogiques Numériques) qui permettent de convertir une tension analogique provenant d'un signal extérieur en une donnée numérique exploitable par le μP et les **CNA** (Convertisseurs Numériques Analogiques) qui émettent vers l'extérieur, sous forme d'une tension analogique, une donnée numérique fournie par le μP.

- la **logique d'interruption** ; une **interruption** est un événement engendré par l'extérieur, qui survient inopinément et qui doit être immédiatement pris en compte par le μP en pleine exécution d'un programme. La logique d'interruption est un organe du μC chargé d'assister le μP dans ce cas de figure.

1.3 Les applications du μC

De taille réduite, les μC sont présents partout ou du moins dans la quasi-totalité des appareils électriques un peu sophistiqués c'est-à-dire réalisant plusieurs fonctions différentes (en particulier des fonctions d'affichage). Les secteurs industriels

qui utilisent les μC sont nombreux : électroménager, Hi Fi, automobile, cartes à puces, appareils médicaux, domotique, automates en tous genres, ...et bien sûr n'importe quel ordinateur personnel comporte un μP.

Les principaux fabricants de ce type de composants sont Intel (le Pentium), Motorola (le μP des ordinateurs Apple, le **68HC11**), Thomson, NEC, Toshiba, Texas Instruments (calculatrices), PIC de Microchips...

1.4 Savez-vous parler le langage du μP ?

1.4.1 Le langage machine

La représentation binaire

Il n'est pas facile de communiquer directement avec un μP. Soyons honnête, nous ne parlons pas la même langue. le μP parle le **binaire** (la base 2) ou plus précisément, il ne comprend et n'interprète que des informations binaires, c'est-à-dire composées de 0 et de 1.Ces deux valeurs correspondent en fait aux deux niveaux de tension que l'on peut rencontrer dans un μC, soit 0V (le "0") soit un niveau supérieur de tension (entre 2 et 5V) dépendant de la tension d'alimentation (le "1").

Ainsi, chaque chiffre du code (donc forcément un 0 ou un 1) est appelé **bit** *(pour binary digit)*.

Une information codée sur 4 bits (exemples : 0011 ou encore 1111) est appelé **quartet**. On peut obtenir $2^4 = 16$ codages binaires différents qui peuvent être interprétées de diverses façons. Par exemple, on peut coder les nombres entiers de 0 à 15 sur un seul quartet :

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
0	1	2	3	4	5	6	7	8	9

1010	1011	1100	1101	1110	1111
10	11	12	13	14	15

Une information codée sur 8 bits (exemples : 00110101 ou encore 10101111) est appelé **octet** *(byte en anglais)*. On peut obtenir $2^8 = 256$ codages binaires différents qui peuvent être interprétées de diverses façons. Par exemple, on peut coder les nombres entiers de 0 à 255 sur un seul octet.

Une information codée sur 2 octets (16 bits) est appelé un **mot** *(word)*.Exemple : 1101010100001111.

La représentation hexadécimale

Comme il est parfois fastidieux d'écrire les informations (le plus souvent des nombres) en binaire, on lui préfère une écriture en **base hexadécimale** (base 16). Ainsi, un quartet peut être représenté comme son interprétation en nombre de la base 16. Ceci revient à noter les 16 quartets différents ainsi :

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
0	1	2	3	4	5	6	7	8	9

1010	1011	1100	1101	1110	1111
A	*B*	*C*	*D*	*E*	*F*

Le langage machine est celui que comprend réellement le μC. Tout programme écrit en langage machine est directement interprété par le μP donc le temps d'exécution est rapide. Pour être compris par un μP, tout langage doit être traduit en langage machine.

Remarque : Lorsque, dans un programme en Assembleur, une valeur est précédée du caractère '$', ceci signifie qu'elle est donnée en hexadécimal. Si le programme est en langage C, le '$' est rempalcé par '0x'.

1.4.2 Le Langage mnémonique

Il s'agit d'un langage un peu plus accessible pour l'être humain. Il est très utilisé car les programmes écrits en ce langage nécessitent peu de mémoire lors de leur stockage. Comme tout langage, il est traduit en langage machine par un programme appelé "Assembleur". Pour cette raison, par abus de langage, on appelle le langage mnémonique **Assembleur**.

L'assembleur reste rapide d'exécution. Il est extrêmement rare de programmer en langage machine mais si le temps d'exécution et la taille du programme sont des facteurs prépondérants dans le choix du programmeur, l'assembleur est un choix judicieux.

Chaque fabricant voire chaque μC a son propre Assembleur. Nous étudierons celui du 68HC11 lors des TD et TP.

1.4.3 Les langages structurés

L'assembleur reste néanmoins difficile d'utilisation lorsque les fonctions réalisées sont sophistiquées. On préfère alors programmer dans des langages disposant de structures toutes faites (**les tests, les boucles**) et de fonctions prédéfinies généralement situées dans des **bibliothèques**.

On utilise un **compilateur** pour traduire le langage structuré en langage machine mais ce dernier peut générer beaucoup de code machine. En conséquence, un langage structuré nécessite plus de mémoire et induit des temps d'exécution plus longs. Malgré l'existence de compilateurs optimisant leur traduction,

le langage mnémonique reste le meilleur moyen de créer des programmes courts et rapides lorsque la difficulté du problème posé le permet.

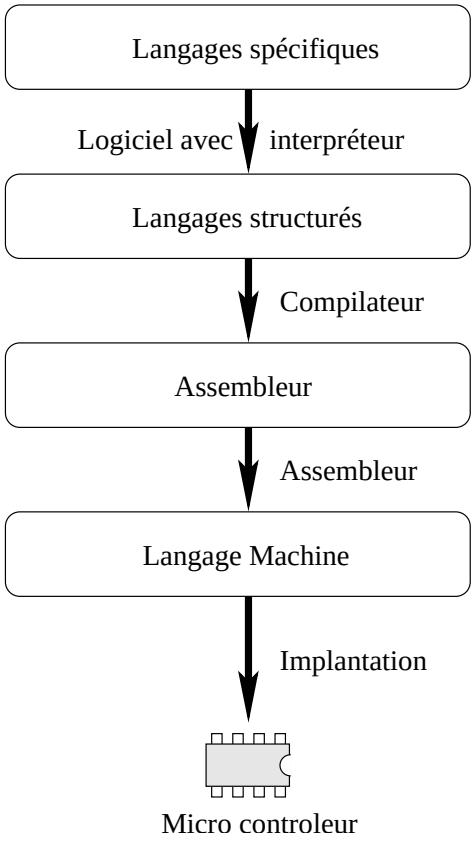
Parmi les langages structurés connus, on peut citer le Basic, Fortran, le C, le Pascal, l'ADA, les langages orientés objet (C++,ADA orienté objet, JAVA...), etc... Nous étudierons le langage C en TD et TP.

1.4.4 Les langages spécifiques

Il s'agit là de langages que l'on n'utilise que dans des cadres très particuliers. Il en existe de nombreuses sortes. Certains s'utilisent avec des logiciels (langage MATLAB, langage LabView, HTML, macros EXCEL, ...). Ces langages subissent différents niveaux d'interprétation, différentes conversions mais finissent toujours sous la forme d'un code binaire.

1.4.5 En résumé

Les paragraphes précédents sont résumés par la figure ci-après.



2 LE μC 68HC11

2.1 Structure du 68HC11

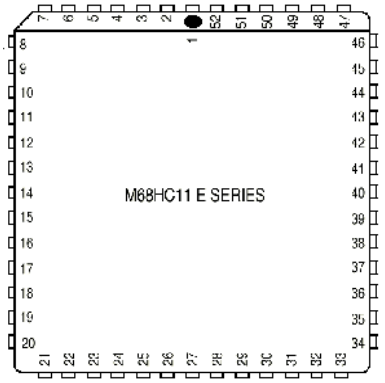
Comme déjà vu plus haut, le μC 68HC11 intègre en un même circuit différents éléments que l'on retrouve dans tous les μC à savoir :

- le μP (que l'on appelle parfois CPU) ;
- les mémoires ;
- les interfaces Entrée/Sorties (dont la logique d'interruption).

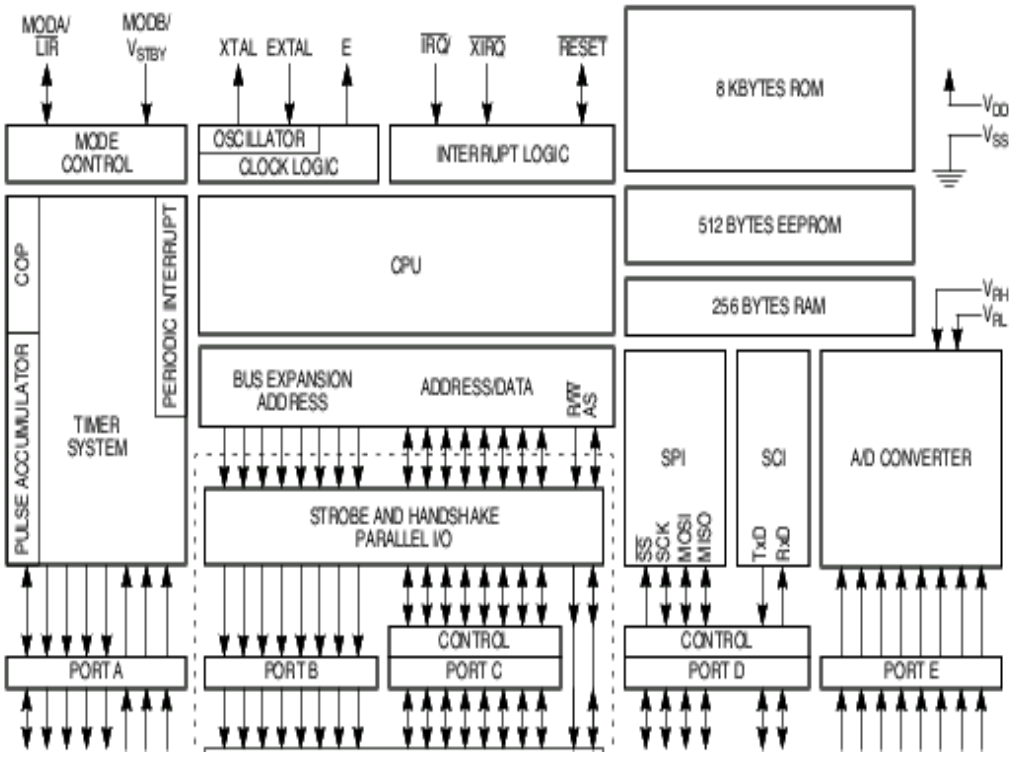
Mais il comprend aussi un certain nombre de ressources que l'on ne retrouve pas toujours sur d'autres μC, à savoir :

- des timers ;
- des CAN, CNA ;
- des liaisons séries synchrone et asynchrone ;
- des extensions mémoire.

D'aspect général, le 68HC11 se présente comme un petit carré de silicium biseauté sur un angle et comportant 52 broches pouvant être connectées à des composants extérieurs (figure ci-dessous).



Mais le schéma général du μC est donné par la figure 5.



2.2 La mémoire du 68HC11E2

Dans l'étude des problèmes que nous rencontrerons **en pratique**, nous serons amener à distinguer trois zones de mémoire selon leurs adresses :

- \$0000→\$0055 : la RAM ou seront stockées les données utilisées et crées par le programme.
- \$1000→\$103F : les registres de fonctions dont l'utilisation doit être appréhendée au fur et à mesure que leur utilisation est nécessaire (voir tableau registres de fonctions).
- \$F800→\$FFFF : l'EPROM dans lequel le programme sera téléchargé.

2.3 Les entrées/sorties du 68HC11

Ce paragraphe est dédié à l'étude des échanges d'information entre le μC et l'environnement extérieur au moyen de ports d'entrée/sortie.

2.3.1 Les ports d'entrées et les ports de sorties

A l'exception du port D qui comprend seulement 6 **lignes d'entrée/sortie**, les 4 autres ports en comportent 8. Ces lignes autorisent :

- soit le transfert d'une information de type "0" ou "1" provenant de l'extérieur vers le μC : on parle alors de lignes d'entrée.

- soit le transfert d'une information du même type du μC vers l'extérieur : on parle alors de lignes de sortie.

Certaines lignes peuvent être configurées en entrée ou en sortie.

Ces informations, codées sur 8bits, peuvent circuler des ports vers le μP ou du μP vers les ports par le **bus de données**.Les 5 ports sont appelés PORTA, PORTB, PORTC, PORTD et PORTE.

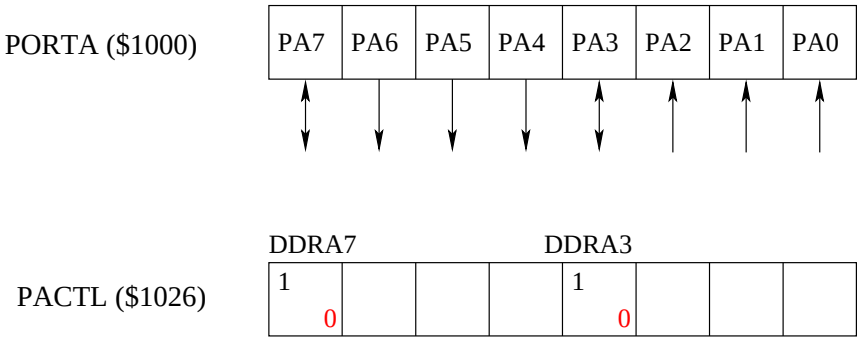
2.3.2 Configuration des entrées/sorties

Comme nous venons de le voir, certaines lignes sont toujours des ligne de d'entrée, d'autres, toujours des lignes de sortie. Enfin, certaines peuvent être d'entrée ou de sortie. On le voit bien sur le schéma de l'architecture globale du 68HC11.

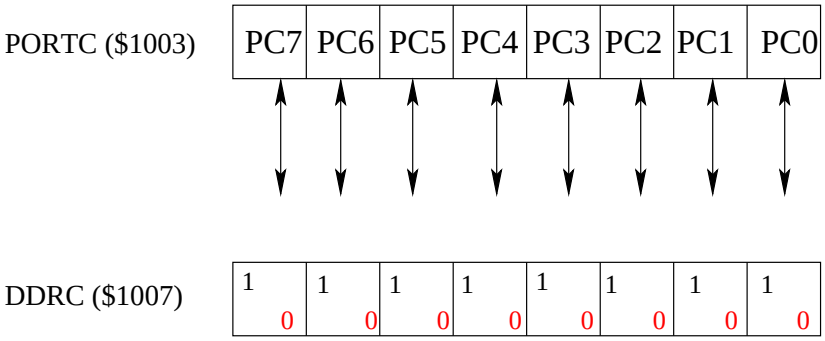
- Le PORTB (adresse \$1004) ne se configure pas. Toutes ses lignes sont de sortie.

- Le PORTE (adresse \$100A) ne se configure pas. Toutes ses lignes sont d'entrée.

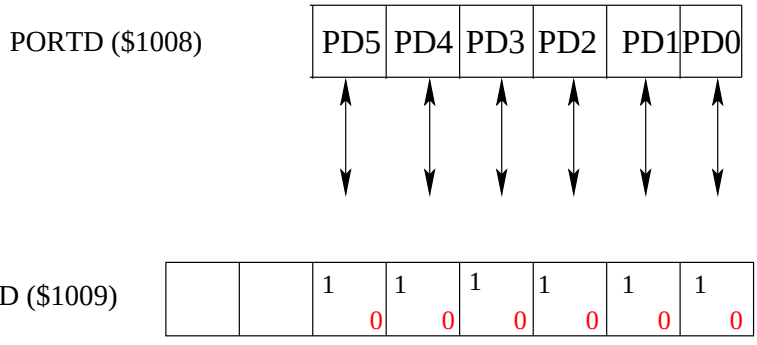
- Le PORTA (adresse \$1000) possède 3 lignes d'entrée, 3 lignes de sortie et deux lignes d'entrée/sortie qu'il reste à configurer si l'on veut les utiliser. Ainsi, si l'on décompose le PORTA en 8 bits de PA0 à PA7, chaque bit correspond à une ligne. Les lignes PA7 et PA3 peuvent être choisies en entrée ou en sortie. Ce choix est effectué en imposant les bits DDRA7 et DDRA3 du registre de fonction PACTL (adresse \$1026). Voir figure ci-après. Un bit à 1 configure la ligne en sortie alors que ce même bit à 0 configure la ligne en entrée.



– Le PORTC (adresse \$1003) est entièrement configurable en entrée/sortie ; La configuration de ses lignes se fait par une écriture dans le registre DDRC (adresse \$1007). Voir figure ci-dessous.



– Le PORTD (adresse \$1008) comporte 6 lignes configurables par l'écriture dans le registre DDRD (adresse \$1009). Voir figure ci-après.



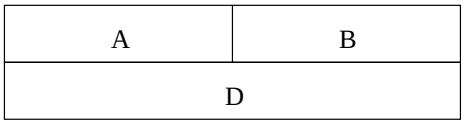
2.4 Les registres internes

Est présenté, dans ce paragraphe, l'intérêt de plusieurs registres très utiles dans le fonctionnement du 68HC11.

2.4.1 Les registres de 8 bits

Les accumulateurs : Il existe deux registres appelés **accumulateurs** correspondant à un octet. Ils sont notés A et B et servent à faire toutes sortes de manipulations sur des données de 8 bits. On peut y stocker temporairement une valeur, faire une opération sur cette valeur à l'intérieur de l'accumulateur ou encore utiliser cet accumulateur pour ranger la donnée en mémoire.

La concaténation des accumulateurs A et B constituent ce qui est appelé le registre D (voir figure ci-dessous).



Le registre CCR : il s'agit d'un registre qui est susceptible d'être modifié à chaque fois qu'une instruction est exécutée. Il se présente comme suit :

CCR	S	X	H	I	N	Z	V	C

On peut s'attarder sur quelques bits de CCR qui seront peut-être utilisés dans le cadre de cet enseignement :

- **C (Carry)** : c'est le bit de retenue. Il est mis à 1 lorsque le résultat d'une instruction génère une retenue (ce peut être une opération arithmétique, une rotation ou encore une complémentation).
- **V (Overflow)** : c'est le bit de débordement. Il est mis à 1 lorsqu'une opération arithmétique génère un débordement de l'accumulateur utilisé.
- **Z (Zero)** : il est mis à 1 lorsque le résultat de l'instruction est nul.
- **N (Negatif)** : il est mis à 1 lorsque le résultat de l'instruction est négatif.
- **I (Interrupt)** : C'est le bit des interruptions. Mis à 1, il inhibe les interruptions c'est-à-dire la prise en compte immédiate d'événements extérieurs survenant pendant l'exécution d'un programme.

Il est inutile pour l'instant d'aborder la signification des bits restants.

2.4.2 Les registres de 16 bits

Il en existe cinq qui sont ici énumérés :

- **le registre D** : c'est la concaténation de A et B déjà mentionnée.
- **le registre X** : a l'instar de A et B, c'est un registre **à tout faire** mais il peut manipuler une donnée de deux octets. Par ailleurs, il est très utile en tant qu'**index** lorsque le μP exécute une instruction utilisant un adressage indexé (nous y reviendrons).
- **Le registre Y** : il joue le même rôle que X si ce n'est que toute instruction utilisant Y plutôt que X se code sur un octet supplémentaire donc son exécution est alors plus lente.
- **le registre SP (Stack Pointer)** : c'est que l'on appelle le **pointeur de pile**. Il indique en permanence la prochaine adresse libre de la **pile**. La pile est une zone de mémoire que l'on peut utiliser pour ranger momentanément des données avant de s'en servir à nouveau selon un principe **LIFO** (Last In First Out : dernier entré, premier sorti). On peut donc empiler puis dépiler des données ce qui signifie que la pile est de taille variable. Le registre SP indique en quelque sorte la hauteur de la pile.Il existe des instructions pour empiler puis dépiler des données.
- **Le registre PC (Program Counter)** : ce registre appelé **compteur ordinal** contient l'adresse de la prochaine instruction du programme qui sera exécutée. Il permet donc au μP de gérer la séquence des instructions du programme, en particulier lorsque des boucles interviennent.

3 Le Jeu d'instructions : "ASSEMBLEUR 68HC11"

Cette partie a pour but de donner quelques bases pour l'initiation à la programmation d'un μC 68HC11 en langage mnémonique. Comme il a déjà été mentionné, chaque constructeur voire chaque μC utilise son propre Assembleur aussi l'on parle d'**Assembleur 68HC11**. C'est celui qui est présenté brièvement dans ce qui suit.

3.1 Le programme

3.1.1 Un programme, quesaco ?

Il s'agit d'une suite d'instructions qui sont exécutées en séquence, c'est-à-dire l'une après l'autre (toutefois, il peut arriver que l'exécution reprenne à une instruction antérieure et ce plusieurs fois : c'est ce qu'on appelle une **boucle**).

Il convient donc de fournir au μP une suite d'instructions compréhensible par lui, donc dans un langage adéquat.

3.1.2 Programme Source/Programme objet

Lorsque le programmeur édite un fichier pour y écrire une suite d'instructions dans le **langage mnémonique (l'assembleur**), il crée ce que l'on appelle un p**rogramme source**. Comme nous l'avons vu à propos des différents langages, ce programme, s'il doit être compris par un lecteur averti, ne l'est pas par le μP. Il n'est donc pas **exécutable**.

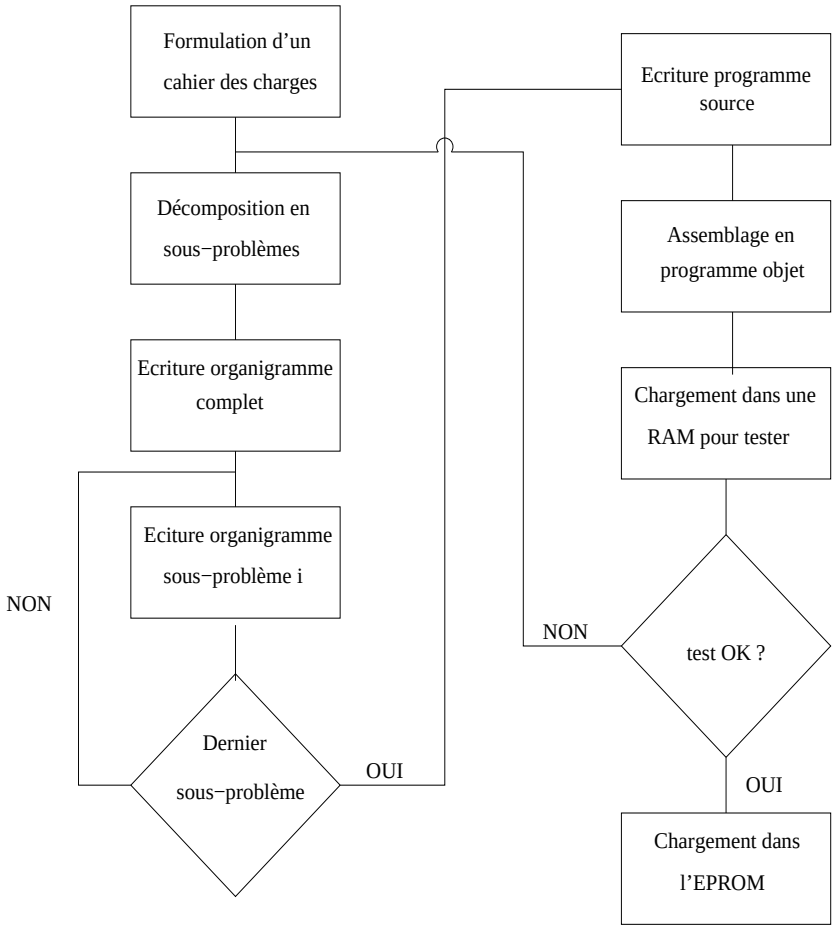
Le μP ne comprend que le **langage machine**. Il est donc indispensable que le programme source soit traduit en langage machine. C'est le rôle de l'assembleur qui convertit le programme source en un **programme objet**. Ce dernier est interprétable par le μP et donc exécutable. Ainsi le programmeur écrit-il un programme source en assembleur avant de le faire "assembler" en un programme objet qui peut être implanté dans l'EEPROM puis exécuté.

C'est de l'écriture d'un programme source en assembleur dont il est question dans cette partie.

3.1.3 Comment écrire un programme ?

Il importe de prendre de bonnes habitudes en programmation, entre autres de faire précéder l'écriture d'un programme par la conception d'un organigramme. C'est une sorte de schéma qui résume la structure du programme. Ne pas faire d'organigramme, c'est risquer de sérieuses désillusions lors de la phase de tests du programme.

La démarche de conception d'un programme est résumée par l'organigramme ci-après.



Les rectangles correspondent aux actions à effectuer et les losanges aux différents tests et boucles présents dans un programme. En pratique, on peut retrouver des instructions très précises à l'intérieur de ces rectangles et losanges.

Il est très important, confronté à un problème, de l'analyser consciencieusement et de le décomposer en divers sous-problèmes plus faciles à résoudre.

3.2 Le logiciel

Par "logiciel", on entend généralement tout simplement programme. L'acception première est un peu différente : il s'agit en réalité de l'ensemble des instructions qui peuvent être utilisées par un μC et qu'on appelle aussi jeu d'instructions. Lorsque ce dernier est exprimé en langage mnémonique, on parle aussi d'"Assembleur". Nous abordons ici quelques aspects de cet assembleur.

Remarque : le mot "Assembleur" désigne donc trois concepts sensiblement différents :

1. *le langage mnémonique ;*
2. *l'ensemble des instructions qui composent ce langage (la nuance est faible) ;*
3. *l'outil qui convertit un programme source en un programme objet.*

3.2.1 Structure d'une instruction

Une instruction est composée de deux parties distinctes et consécutives :

- un opérateur (obligatoire) ;
- un ou plusieurs opérandes (facultatif).

Certaines instructions ne nécessitent pas d'opérandes. Ce sont des opérations sur registre. La nature des opérateurs et opérandes conduit à classer les instructions du logiciel selon différents modes d'adressage.

3.2.2 Les équivalences et les étiquettes

Lorsque dans l'écriture d'un programme, une valeur (qui peut d'ailleurs être une adresse) revient de manière récurrente ou bien a une signification toute particulière que l'on veut faire apparaître dans le programme, on peut dès le début de ce dernier écrire une ligne telle que :

REG_BASE = $1000

Cette instruction signifie à l'assembleur que, dans le suite du programme, il devra comprendre $1000 à chaque fois qu'il rencontrera REG_BASE.

Par ailleurs, l'on est parfois amené à identifier une ligne particulière à laquelle le compteur ordinal est susceptible de revenir lors de l'exécution du programme (au cours de l'exécution d'une boucle notamment). L'on peut alors "baptiser" cette ligne afin de rendre lisible le programme source.

Exemple : **RETOUR LDAA #$1000** $\Rightarrow$ Lorsque l'on envisage un retour à l'instruction LDAA #$1000 (dont la compréhension n'est pas nécessaire ici), on peut spécifier au μP de repartir à la ligne RETOUR.

3.2.3 Les modes d'adressage

Adressage inhérent

C'est un mode d'adressage qui se passe d'opérande. Il correspond à des opérations sur registre.

Exemple : **CLRA** ⇒ efface le contenu de l'accumulateur A.

Adressage immédiat

L'opérande manipulé se trouve directement après l'opérateur et n'est pas spécifié par l'intermédiaire d'une adresse mais directement par sa valeur comme l'indique le symbole #.

Exemple : **LDAA #$12** ⇒ L'accumulateur est chargé avec une donnée correspondant à 12 en hexadécimal.

Adressage étendu

Cette fois-ci, la donnée n'est pas spécifiée directement mais au travers de son adresse en mémoire (le symbole # est absent) ou bien la donnée manipulée (déjà présente dans l'accumulateur) va agir sur le contenu d'une adresse.

Exemples :

1/ **LDAA $0035** ⇒ L'accumulateur A reçoit le contenu de l'adresse 35 (en hexadécimal comme l'indique le symbole $).

2/ **STAA $0035** ⇒ Le contenu de A est chargé à l'adresse $35.

3/ **ADDA $0035** ⇒ Le contenu de $35 est ajouté à celui de A et le résultat est mis dans A.

Adressage direct

Analogue à l'adressage étendu ; la seule différence est que l'adresse mentionnée est spécifiée seulement sur un octet, le poids fort étant alors considéré comme nul.

Exemple : **LDAA $35** ⇒ L'accumulateur A reçoit le contenu de l'adresse $35.

Adressage indexé

Ce mode se rapproche quelque peu de l'adressage étendu mais l'adresse est "partiellement contenue" dans un registre (**X** ou **Y**). Tout dépend donc du contenu de cet index.

Exemple : **LDX #$1000** ⇒ le registre **X** reçoit la valeur hexadécimale 1000 (adressage immédiat). **LDAA $05,X** L'accumulateur est chargé avec une valeur située à l'adresse égale à la somme du déplacement indiqué (soit $5) et du contenu de X (soit $1000). Ainsi, l'instruction précédente se résume à **A← ($1005)** (Les parenthèses signifient "contenu de").

Adressage relatif

C'est un mode d'adressage que l'on retrouve lors des tests ou des boucles qui nécessitent une rupture (avec un éventuel retour en arrière) dans la séquence d'instruction. Pour faciliter l'écriture du programme dans un tel cas, on ne spécifie pas explicitement l'adresse à laquelle le **PC** doit se rendre mais une étiquette.

Exemple :

```
        LDAA    003A
Retour  DECA
        BNE     Retour
```

Dans ce petit extrait de programme, la première ligne charge **A** avec le contenu de l'adresse $3A, puis on décrémente le contenu de **A** à la deuxième ligne et la troisième ligne signifie que si ce contenu ne devient pas égal à zéro, alors le μP doit revenir à la ligne indiquée par **Retour**.

3.2.4 Instructions sur bits

Il existe quelques instructions spéciales qui permettent d'utiliser l'information présente non par sur un octet entier mais sur un ou plusieurs bits de cet octet. De même, d'autres instructions, permettent de modifier partiellement un octet. Ces instructions utilisent des masques. En voici quelques unes.

L'instruction BSET :

Exemple : **BSET 4F,53**

Dans cette instruction, le premier opérande ($4F) est une adresse et le second (03), un masque. L'instruction agit sur une donnée d'un octet située en mémoire à l'adresse $4F. Le masque (ici 3) correspond à la valeur 01010011. Ceci signifie que sur les 8bits de l'adresse $4F, tous ceux correspondant à un **1** dans le masque, doivent être imposés à **1**.

Ainsi si la valeur en $4F était E4, elle deviendra F7.

L'instruction BCLR :

Elle fonctionne de la même manière que BSET mais en imposant à **0** les bits spécifiés par un **1** dans le masque. Ainsi, si l'adresse 4F contient initialement la valeur FF, **BCLR 4F,53** va modifier cette valeur en AC.

L'instruction BRSET :

Il s'agit de tester un ou plusieurs bits d'une adresse pour savoir si un branchement est réalisé ou pas.

Exemple : **BRSET 4F,53,branchement**

Dans cette expression, on teste si les bits correspondant au masque sont à **1**. Si c'est le cas, le compteur ordinal PC se rend à l'adresse "branchement". Cette dernière peut être indiquée soit explicitement, soit par une étiquette de branchement.

Ainsi, si l'adresse 4F contient FF, le branchement a lieu alors que si elle contient F0, il n'a pas lieu.

L'instruction BRCLR :

Elle fonctionne de manière analogue à l'instruction BRSET mais le branchement a lieu si les bits indiqués par le masque sont à **0**.

Remarque : L'adresse est codée sur un octet. C'est un adressage direct. L'adressage étendu ne peut être utilisé ici. En revanche, on peut utiliser l'adressage indexé (exemples : ***BCLR 01,X,53*** *ou* ***BRCLR 5,X,53,F830****).*