Langage évolué ou assembleur ?

Préambule : 2
Un ordinateur c'est quoi ? 2
Vision statique : contrôle, ALU, registres et indicateurs 2
Vision dynamique : des cycles qui se répètent à l'infini 3
Machines à accumulateur : le point de passage obligé 4
Du langage source à la machine 6
Du programme source au code binaire exécutable : traduire et relier 6
Opérations et types : des algorithmes différents 8
Adresses et valeurs 10
Classes de mémorisation : portée d'un nom et durée de vie 10
Objets statiques et symboles : nommer les adresses fixes 11
Modes d'adressage 13
Pointeurs et tableaux ou structures : adressages indirects 13
Variables dynamiques : registres et pile, des variables sans adresse ou dont l'adresse n'est pas fixe 14
Les variables « volatiles » 15
Fonctions et sous programmes 16
Sous programmes : adresses d'entrée et adresse de retour 16
Fonctions ou procédures : arguments et valeur retournée 16
Le rôle de la pile : contexte d'une fonction 18
Annexe : les instructions du cœur HC12 19

Langage évolué ou assembleur ?

Préambule :

Le texte qui suit tente d'éclairer le lien entre langage (logiciel) et machine (matériel). Il comporte au passage l'introduction d'éléments de vocabulaire, essentiels à la compréhension de toute documentation technique.
Les mots nouveaux dont la définition est donnée dans le texte sont soulignés en gras lors de leur première apparition, ils devront évidemment faire partie du bagage de connaissances à l'issue de la lecture de ce document.
Quand des exemples précis sont fournis à titre d'illustration des explications, ils gagneront à être étudiés soigneusement, tous les exemples concernent le microcontrôleur Motorola 68HC12, machine utilisée en travaux pratiques et en maquettes. Nous avons délibérément choisi de ne pas asséner une liste exhaustive des instructions de cette machine, mais il est impossible de programmer correctement un microcontrôleur sans connaître un minimum l'architecture de la machine et le langage assembleur associé.

Un ordinateur c'est quoi ?

La première vision que l'on peut avoir d'un ordinateur est celle d'une gare de triage. Des données (stockées en mémoire, issues de périphériques ou allant vers ces éléments) circulent d'un point à un autre du système, subissant éventuellement au passage des opérations arithmétiques ou logiques. Le cœur matériel du système est l'**unité centrale** ; son rôle est de coordonner toutes les opérations, d'en assurer le bon déroulement temporel.
La contrainte temporelle est essentielle :

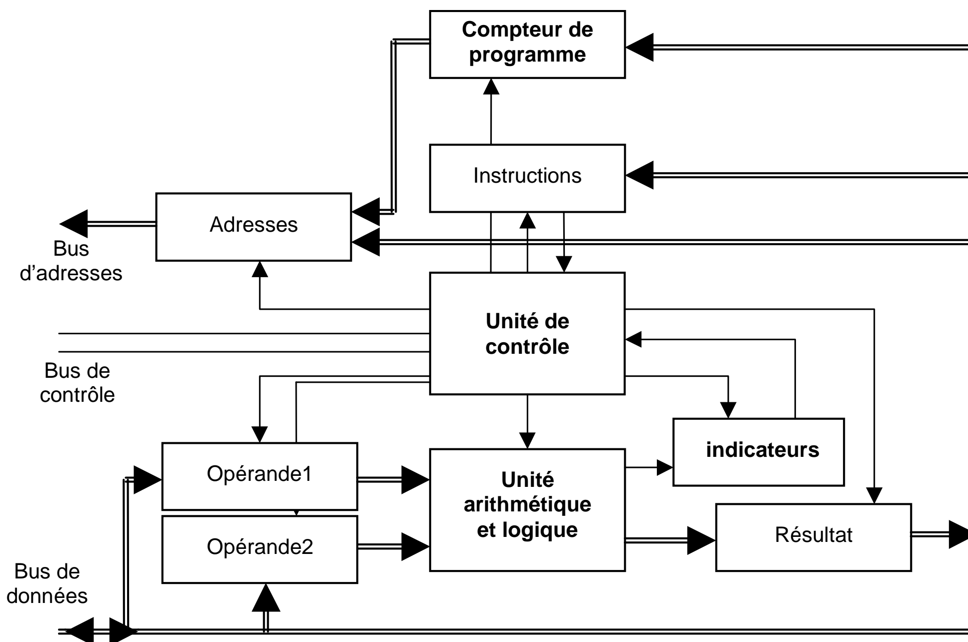
- Dans la plupart des machines il n'y a qu'un seul chemin de données, une sorte de voie unique. Deux données ne peuvent donc pas circuler en même temps sur cette voie unique (le **bus de données**).
- Une simple opération comme res = a + b nécessitent un ordonnancement rigoureux des opérations, il faut chercher les opérandes de quelque part, faire une addition, ranger le résultat quelque part. L'unité centrale doit assurer que ces différentes étapes d'une opération se déroulent dans le bon ordre.

La conduite du bon déroulement des opérations est assurée par l'**unité de contrôle**, partie essentielle de l'unité centrale.
C'est généralement au cœur de l'unité centrale qu'ont lieu les opérations élémentaires [1], additions soustractions, opérations logiques etc. La partie de l'unité centrale consacrée aux opérations est l'**unité arithmétique et logique** (*ALU, pour Arithmetic and Logical Unit*).
Stocker des données en mémoire, les rechercher, les transférer vers un périphérique ou les lire en provenance d'un périphérique suppose que chaque objet dans une machine ait une adresse unique. Cette adresse est généralement un nombre entier, la valeur de l'adresse de chaque élément de la machine (cases mémoires, registres de périphériques etc.) est fixée une fois pour toute lors de la configuration du système. L'unité centrale joue un rôle privilégié dans la gestion des adresses : dans la majorité des cas c'est elle seule qui fixe à chaque instant (au moyen du **bus d'adresses**) les adresses de provenance ou de destination des données qui circulent sur le bus de données.
L'unité centrale échange avec le monde extérieur des informations complémentaires : sens du transfert (écriture ou lecture), signaux de synchronisation etc. Tous ces signaux sont regroupés, par abus de langage, dans ce que l'on appelle le **bus de contrôle**.

Vision statique : contrôle, ALU, registres et indicateurs

Le schéma synoptique d'une unité centrale imaginaire pourrait donc ressembler à quelque chose comme celui de la figure ci-dessous :

[1] Sauf dans les machines complexes qui disposent de co-processeurs dédiés au calcul.



L'accès au monde extérieur (mémoires, périphériques) se fait via les bus de données, d'adresses et de contrôle. Evidemment un calcul nécessite la mémorisation à l'intérieur de l'unité centrale d'informations multiples (par exemple les deux opérandes d'une addition, ou la valeur d'une variable et son adresse) qu'un chemin de données unique ne peut assurer. Sur le schéma on a donc fait figurer, en plus des blocs déjà évoqués, des **registres** (mémoires internes à l'unité centrale) qui contiennent des données temporaires comme les opérandes d'une opération, le résultat d'un calcul. Un élément particulier est le **compteur de programme** (ou **compteur ordinal**), registre qui contient en permanence l'adresse en mémoire de l'instruction à exécuter, ce registre est mis à jour automatiquement par l'unité de contrôle, l'utilisateur n'en modifie explicitement la valeur que très exceptionnellement.
Comme son nom l'indique, le registre d'instruction contient, sous forme binaire, les informations nécessaires à l'unité de contrôle pour exécuter l'instruction en cours. L'utilisateur n'a aucun accès direct à ce registre qui est géré entièrement par le matériel.
Une opération réalisée par l'unité arithmétique et logique fournit deux choses :

- un résultat, bien sûr, 3 – 5 produit – 2,
- et un indicateur, – 2 est un nombre négatif, il peut être intéressant dans la suite du programme de disposer de cette information (`if (res< 0)` ...).

Ces informations complémentaires au résultat sont mémorisées dans un registre particulier : les **indicateurs** (***flags***), qui sont des valeurs binaires (1 ou 0, pour vrai ou faux). Les indicateurs les plus courants sont :

- **N** pour indiquer un résultat négatif,
- **Z** pour indiquer un résultat nul,
- **C** (*carry*) indique qu'une opération arithmétique a généré un report (ou une retenue),
- **V** (*overflow*) indique qu'une opération arithmétique a provoqué un débordement (30000 * 2 => – 5536).

Ayant à notre disposition un modèle de machine, voyons comment elle pourrait réaliser une opération.

Vision dynamique : des cycles qui se répètent à l'infini

Soit à réaliser l'opération res = a + b, où res, a et b sont des nombres entiers.
La première transformation que va subir notre programme est d'être traduit en langage machine, suite d'instructions élémentaires qui prennent en compte l'unicité du chemin de données. Décrit dans le langage courant ces instructions pourraient être :

Transférer a dans opérande1 (a => opérande1)
Transférer b dans opérande2 (b => opérande2)
Calculer opérande1 + opérande2 (opérande1 + opérande2 => résultat)
Transférer résultat dans res (résultat => res)

Une observation rapide du programme nous montre que notre architecture de machine dilapide les ressources du système : l'absence de connexion directe entre l'unité arithmétique et logique et le chemin de

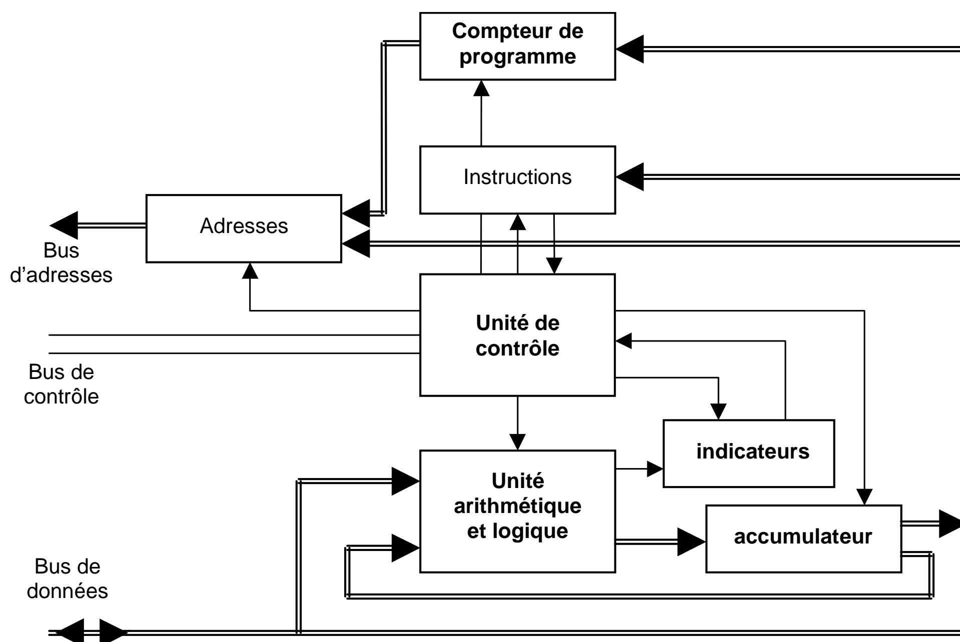
données nous oblige à passer systématiquement par des registres, et à introduire une opération qui n'utilise pas le chemin de données. Nous reviendrons sur ces aspects.
Chaque instruction du langage machine cache en réalité plusieurs **cycles** élémentaires générés par l'unité de contrôle, par exemple « transférer a dans opérande1 » peut être décomposée en :

Transférer l'instruction dans le registre d'instructions (*PC++ => Instruction)[2] (***Fetch cycle***)
Placer l'adresse de a dans le registre d'adresse (*PC++ => Adresses)
Charger opérande1 (bus_données => opérande1)

Etc.
Pour résumer : notre instruction en langage C (res = a + b) a généré quatre instructions « machine », chaque instruction machine a elle même généré plusieurs cycles d'accès au monde extérieur et de calcul. On devine ici que les choix d'architecture et de séquencement sont intimement liés, c'est le métier des concepteurs d'unités centrales d'optimiser ces choix en fonction des applications visées. Les solutions ne sont évidemment pas uniques.
La plupart des machines traduiront chaque instruction en au moins deux cycles : cycle fetch et cycle d'exécution. Chacun de ces cycles pouvant générer une séquence de sous cycles d'accès à la mémoire.

Machines à accumulateur : le point de passage obligé

Une méthode classique d'optimisation des ressources consiste à choisir une architecture à **accumulateur**. Dans sa version élémentaire une machine à accumulateur se présente comme l'indique la figure ci-dessous :



L'unité arithmétique et logique reçoit ses opérandes du chemin de données et d'un registre spécial, l'accumulateur. Comme son nom l'indique l'accumulateur recueille le résultat de toutes les opérations. Le microcontrôleur 68HC12 est une machine à accumulateur, profitons en pour découvrir nos premières lignes de langage machine, dans une écriture lisible : le **langage assembleur**[3].

```
short a, b, res ;
...
res = a+b ;

  0000 fc0000       LDD   a
  0003 f30000       ADDD  b
  0006 7c0000       STD   res
```

[2] On utilise ici une notation inspirée du C, l'étoile indique le rôle de pointeur joué par le compteur de programme, ce pointeur est systématiquement incrémenté après chaque utilisation.
[3] A plusieurs reprises on trouvera des exemples de programmes. Ceux-ci sont à lire avec une **extrême attention**, ils sont le support d'apprentissage et de compréhension de bien des choses.

Les premières lignes sont extraites d'un programme en C qui nous sert de support, les trois dernières lignes indiquent la traduction en langage machine, dans un format :

```
Adresse_en_mémoire  code_binaire     langage_assembleur
```

LDD veut dire « *load double* », charge (sous entendu l'accumulateur), double pour indiquer qu'il s'agit d'un objet sur deux octets (short). La suite se comprend aisément : ADDD pour addition double, STD (*store*) pour range en mémoire un objet sur deux octets.
L'intérêt d'une machine à accumulateur est évident dès que l'on introduit des opérations en chaîne :

```
short a, b, c, res ;
...
res = a+b-c ;

  0000 fc0000       LDD   a
  0003 f30000       ADDD  b
  0006 b30000       SUBD  c
  0009 7c0000       STD   res
```

L'accumulateur contient le résultat intermédiaire des opérations en cours.
En réalité le microcontrôleur 68HC12 est l'héritier de machines 8 bits (6809), c'est à dire de machines qui effectuent les calculs élémentaires sur un octet. L'accumulateur est en fait constitué de deux registres d'un octet : A et B. Dit autrement, on peut considérer que ce microcontrôleur comporte deux accumulateurs, A et B, que l'on peut regrouper, pour certaines opérations, en un accumulateur double de 16 bits : D ≡ (A:B).
L'ensemble des opérations d'additions et de soustractions disponibles est résumé dans le tableau ci-dessous :

Table 5-4. Addition and Subtraction Instructions

Mnemonic	Function	Operation
Addition Instructions		
ABA	Add B to A	$(A) + (B) \Rightarrow A$
ABX	Add B to X	$(B) + (X) \Rightarrow X$
ABY	Add B to Y	$(B) + (Y) \Rightarrow Y$
ADCA	Add with carry to A	$(A) + (M) + C \Rightarrow A$
ADCB	Add with carry to B	$(B) + (M) + C \Rightarrow B$
ADDA	Add without carry to A	$(A) + (M) \Rightarrow A$
ADDB	Add without carry to B	$(B) + (M) \Rightarrow B$
ADDD	Add to D	$(A{:}B) + (M : M + 1) \Rightarrow A : B$
Subtraction Instructions		
SBA	Subtract B from A	$(A) - (B) \Rightarrow A$
SBCA	Subtract with borrow from A	$(A) - (M) - C \Rightarrow A$
SBCB	Subtract with borrow from B	$(B) - (M) - C \Rightarrow B$
SUBA	Subtract memory from A	$(A) - (M) \Rightarrow A$
SUBB	Subtract memory from B	$(B) - (M) \Rightarrow B$
SUBD	Subtract memory from D (A:B)	$(D) - (M : M + 1) \Rightarrow D$

Deux registres supplémentaires, X et Y apparaissent dans ce tableau, nous les retrouverons plus tard, M symbolise une case mémoire sur un octet, un mot de 16 bits est donc fort logiquement noté (M:M+1), les adresses en mémoire faisant référence à un octet..
En examinant (le faire… avec attention !) le tableau précédent on peut remarquer que ce microcontrôleur ne dispose pas d'addition sur 16 bits avec retenue. Comment faire de l'arithmétique sur des entiers longs (32 bits) avec une telle architecture ?
Le programme ci-dessous fournit une réponse possible :

```
long la, lb, lc, lres ;
...
lres = la+lb ;
```

```
000c fc0000        LDD   la:2
000f f30000        ADDD  lb:2
0012 7c0000        STD   lres:2
0015 fc0000        LDD   la
0018 f90000        ADCB  lb:1
001b b90000        ADCA  lb
001e 7c0000        STD   lres
```

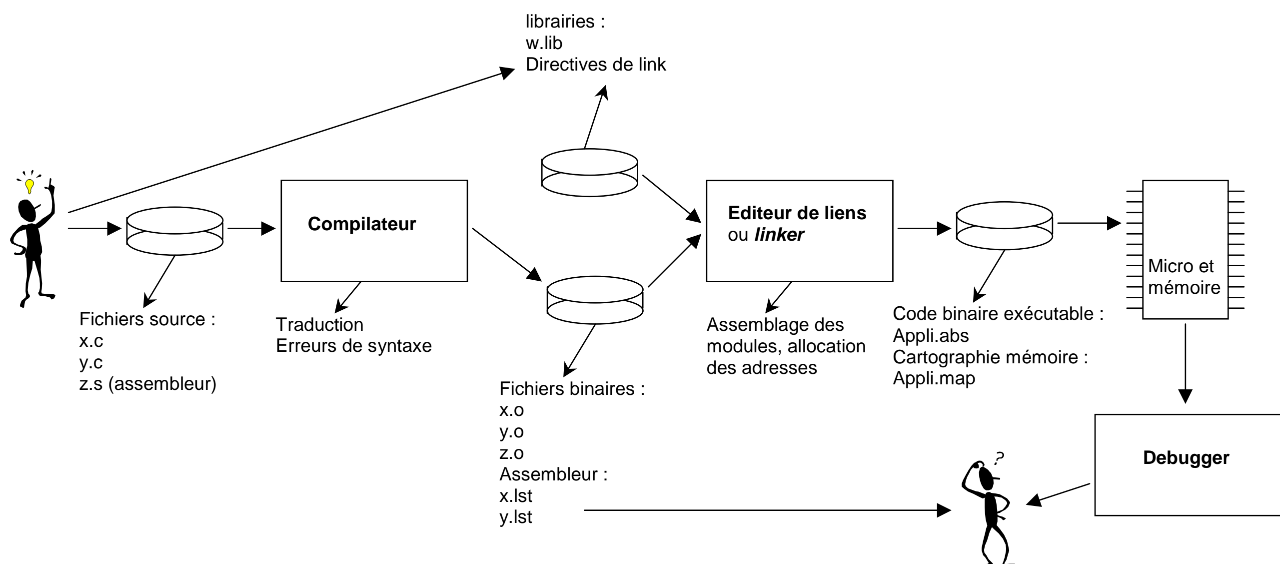
Les 16 bits de poids faible (adresses la+2 et lb+2, notées en assembleur la:2 et lb:2) sont additionnés sur 16 bits, les 16 bits de poids fort sont additionnés octet par octet, avec prise en compte de la retenue éventuelle.
A l'occasion de cette rapide découverte de l'architecture d'une machine, nous avons introduit la notion de langage assembleur, vision humainement lisible des instructions binaires exécutables par une unité centrale. Le paragraphe qui suit précise ce passage d'un programme source en langage évolué (le C en l'occurrence) à la machine.

Du langage source à la machine

Quelle que soit la chaîne de développement utilisée, le passage du langage source au langage machine se fait en (au moins) deux étapes fondamentales. La **compilation** et l'**édition de liens**. La première étape est une étape de traduction, la seconde alloue des adresses aux données et au programme.
Au niveau de la machine la notion de type, essentielle dans les langages évolués, n'existe qu'à l'état embryonnaire, le matériel réalise des opérations logiques élémentaires sur des données qui sont dépourvues de toute signification. La même opération en langage évolué, portant sur des types différents, sera donc traduite par des algorithmes différents au niveau de la machine. Au delà des types simples, les langages évolués permettent de manipuler des objets structurés (tableaux, structures du C). La manipulation de tels objets est rendue possible, au niveau du processeur, par des opérations de calcul d'adresses, on parle de **modes d'adressages**.

Du programme source au code binaire exécutable : traduire et relier

Les deux étapes fondamentales qui font passer d'un programme source à des instructions binaires stockées dans une mémoire sont résumées par le diagramme ci-dessous. On veillera à toujours avoir conscience de « où on se situe » dans la chaîne quand on met au point un programme.

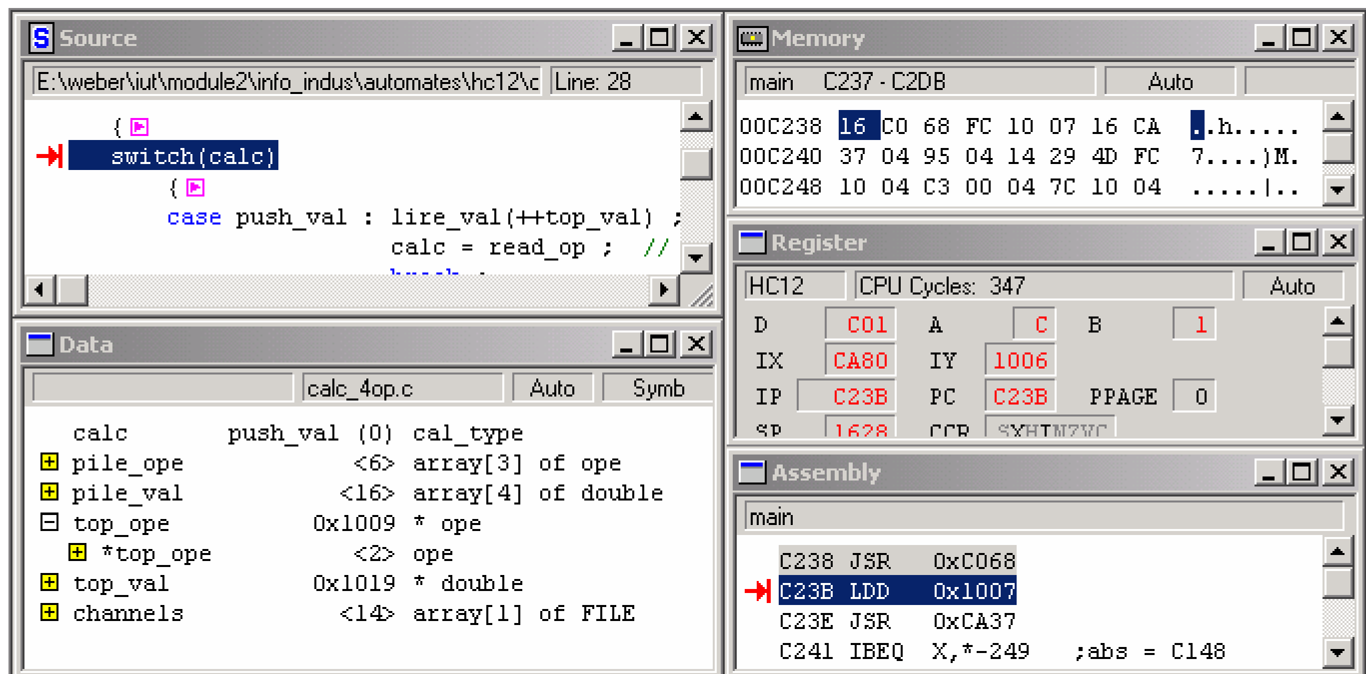


Le rôle du compilateur est double : traduire, bien sûr, mais avant de traduire il faut s'assurer que le programme est syntaxiquement correct. L'un des principes qui guide les inventeurs de langages est que les erreurs graves en programmation sont des erreurs qui se manifestent à l'exécution d'un programme. Une syntaxe rigoureuse, concernant les types de données notamment, tend à rejeter au niveau grammatical des erreurs qui auraient des conséquences catastrophiques à l'exécution. Détectées à la compilation ces erreurs peuvent être corrigées avant que le programme ne soit exécuté.
Nous n'avons pas distingué dans le synoptique précédent les modules écrits en langage évolué des modules écrits directement en assembleur. De tels modules existent toujours (dans une proportion de l'ordre de 5%)

même dans les grandes applications. La plupart des compilateurs offrent le moyen d'insérer des parties en assembleur dans un code écrit principalement en langage évolué.
L'éditeur de liens est le « maître des adresses ». L'utilisateur fournit une description matérielle de la machine cible à laquelle son programme est destiné, au moyen d'un fichier de directives, l'éditeur de liens tente alors de placer dans la mémoire les instructions du programme et certaines données (voir plus loin), y rajoute des modules de bibliothèque (fonctions diverses) et recalcule en conséquence l'ensemble des adresses des objets du programme.
Un exemple de fichier de directive d'éditeur de lien est donné ci-dessous :

```
NAMES
END

SECTIONS
    RAM = READ_WRITE 0x1000 TO 0x3FFF;
    /* unbanked FLASH ROM */
    FLASH_PAGE4000 = READ_ONLY 0x04000 TO 0x07FFF;
    FLASH_PAGEC000 = READ_ONLY 0x0C000 TO 0x0FEFF;
    EEPROM = READ_WRITE 0x0800 TO 0x0FFF;
END

PLACEMENT
    _PRESTART, STARTUP,
    ROM_VAR, STRINGS,
    NON_BANKED, DEFAULT_ROM,
    COPY                    INTO  FLASH_PAGE4000, FLASH_PAGEC000;
    DEFAULT_RAM             INTO  RAM;
END

STACKSIZE 0x100

VECTOR ADDRESS 0xFFFE  _Startup
```

Non portables, ces fichiers obéissent à une syntaxe qui dépend des outils de développement. Ils sont rarement édités manuellement, l'utilisateur passe par un gestionnaire de projet qui prend en charge la description matérielle du système cible.
L'éditeur de liens fournit, outre le fichier binaire qui sera chargé en mémoire, un compte rendu (fichier *.map sur les outils Metrowerks, par exemple) qui donne une description complète de l'application, avec les correspondances entre les noms (de variables, de fonctions) et leurs adresses en mémoire. Là encore les outils de debug symbolique évitent le plus souvent à l'utilisateur d'avoir à explorer ces fichiers en détail : un debugger évolué permet à l'utilisateur d'accéder aux objets de son programme à partir de leurs noms dans le code source, sans avoir à connaître les valeurs numériques des adresses.



Un ensemble de fenêtres couplées permet de visualiser simultanément le programme source, les valeurs des variables, le code machine correspondant et l'état de la machine (registres). L'utilisateur peut placer des points d'arrêt au niveau du code source ou de l'assembleur équivalent, lancer des exécutions en pas à pas etc.

Opérations et types : des algorithmes différents

Nous avons déjà rencontré une variation dans le code généré en fonction du type des variables à propos de l'addition. Donnons un autre exemple : la détermination de la valeur maximum d'un nombre. Au niveau de la machine l'opération de comparaison (CPD pour une comparaison sur 16 bits) fait une soustraction, sans ranger le résultat dans l'accumulateur, et positionne les indicateurs en fonction du résultat de cette soustraction. La recherche du maximum consiste donc à tester la valeur des indicateurs. L'exécution du programme se poursuit à une adresse qui dépend du résultat du test, au moyen d'un saut (la nouvelle valeur du compteur de programme est calculée).
Le programme (C et assembleur équivalent mélangés) initial :

```
unsigned short a_us, b_us, max_us ;
...
if(a_us > b_us) max_us = a_us ;
  0030 fc0000        LDD    a_us
  0033 bc0000        CPD    b_us
  0036 2302          BLS    *+4 ;abs = 003a
  0038 2003          BRA    *+5 ;abs = 003d
else max_us = b_us ;
  003a fc0000        LDD    b_us
  003d 7c0000        STD    max_us
```

L'instruction CPD retranche le contenu d'un mot mémoire au contenu de l'accumulateur. L'instruction BLS *+4 incrémente le compteur de programme de 4 si une retenue (indicateur C) a été générée ou si la soustraction fournit un résultat nul (bit Z). Ce **branchement conditionnel** a donc lieu si a_us – b_us <= 0 (3 – 4 génère une retenue : il faut retrancher 4 de 13, en décimal), le maximum est alors b_us, d'où l'instruction de la ligne 003a. Si le test fournit un résultat faux, un **branchement inconditionnel** (BRA) fait sauter la ligne 003A. Dans tous les cas le résultat est rangé en mémoire à partir de l'accumulateur.
Une modification semble-t-il mineure du programme conduit à l'exemple ci-dessous :

```
short a_ss, b_ss, max_ss ;

if(a_ss > b_ss) max_ss = a_ss ;
  0020 fc0000        LDD    a_ss
```

```
  0023 bc0000       CPD   b_ss
  0026 2f02         BLE   *+4 ;abs = 002a
  0028 2003         BRA   *+5 ;abs = 002d
else max_ss = b_ss ;
  002a fc0000       LDD   b_ss
  002d 7c0000       STD   max_ss
```

Le programme est le même à une exception près, soulignée en gras, l'instruction de branchement conditionnel est maintenant BLE, qui teste l'expression logique $Z + (N \oplus V)$, le branchement a lieu si le résultat est nul ou si les indicateurs Z et V sont différents. La seule différence entre les deux programmes sources réside dans les types (signés ou non) des variables.
A titre d'exercice on écrira en assembleur le programme qui retourne la valeur maximum de deux pointeurs (sur le HC12 les pointeurs par défaut sont sur 16 bits).
Nous avons au passage découvert deux nouvelles catégories d'instructions : les comparaisons et les branchements calculés par un déplacement par rapport à la position courante du compteur de programme. Compte tenu de leur importance, nous donnons ci-dessous un tableau qui résume tous les branchements dits « courts » (déplacement signé sur un octet). Les mêmes types de branchements signés, mais avec un déplacement calculé sur 16 bits, sont obtenus en rajoutant un 'L' devant le code de l'instruction (LBRA au lieu de BRA, par exemple).
De toute façon le programmeur n'a pas à se soucier de ce type de détails, le compilateur détermine de lui même le branchement qui convient, en fonction de la taille réelle du saut déterminée au moment de la compilation.
On notera que la technique qui consiste à calculer le branchement par un déplacement à partir de la valeur courante du compteur de programme conduit à générer du code qui ne dépend pas des adresses physiques où est enregistré le code binaire exécutable (*PIC*, pour *position independant code*). Cela facilite évidemment le travail de l'éditeur de lien, cela permet également de fournir des modules logiciels sous forme de code binaire, en mémoire ROM, par exemple.

Table 5-17. Short Branch Instructions

Mnemonic	Function	Relation	Equation or Operation
	Unary Branches		
BRA	Branch always		$1 = 1$
BRN	Branch never		$1 = 0$
	Simple Branches		
BCC	Branch if carry clear		$C = 0$
BCS	Branch if carry set		$C = 1$
BEQ	Branch if equal		$Z = 1$
BMI	Branch if minus		$N = 1$
BNE	Branch if not equal		$Z = 0$
BPL	Branch if plus		$N = 0$
BVC	Branch if overflow clear		$V = 0$
BVS	Branch if overflow set		$V = 1$
	Unsigned Branches		
		Relation	
BHI	Branch if higher	$R > M$	$C + Z = 0$
BHS	Branch if higher or same	$R \geq M$	$C = 0$
BLO	Branch if lower	$R < M$	$C = 1$
BLS	Branch if lower or same	$R \leq M$	$C + Z = 1$
	Signed Branches		
BGE	Branch if greater than or equal	$R \geq M$	$N \oplus V = 0$
BGT	Branch if greater than	$R > M$	$Z + (N \oplus V) = 0$
BLE	Branch if less than or equal	$R \leq M$	$Z + (N \oplus V) = 1$
BLT	Branch if less than	$R < M$	$N \oplus V = 1$

Adresses et valeurs

Jusqu'ici nous avons considéré que les variables de nos programmes avaient une adresse connue, repérée par leur nom. Nous avons rencontré en passant un premier exemple de calcul d'adresse : les branchements. Dans ce dernier cas l'adresse calculée concerne la mémoire programme, qui contient des instructions.
Avant d'approfondir un peu ce sujet (relativement vaste, il faut bien le dire), il n'est sans doute pas inutile de revenir sur un concept de base des langages évolués, le C notamment : les classes de mémorisation.

Classes de mémorisation : portée d'un nom et durée de vie

Un chapitre fondamental de tout bon livre de programmation porte le titre « Qu'est-ce qu'un nom ? »[4]. De quoi s'agit-il ?
Tout objet (variable ou fonction dans le cas du langage C) porte un nom et possède un certain nombre d'attributs que nous allons préciser. Le nom est un moyen symbolique de repérer l'objet, de l'utiliser. Concrètement derrière le nom d'une variable ou d'une fonction se cache l'adresse en mémoire de l'objet. Ce nom va permettre, par exemple, d'affecter une valeur à une variable, donc de ranger cette valeur en mémoire, sans avoir besoin de connaître explicitement l'adresse de la case mémoire qui a été déterminée par le processus de compilation et d'édition de liens. Suivant un mécanisme similaire, le nom d'une fonction permet d'appeler un sous programme, c'est à dire du code exécutable logé quelque part dans la mémoire, sans connaître explicitement l'adresse de ce code.
Le premier attribut d'une variable (ou d'une fonction dans le cas du C) est son type. Nous ne reviendrons pas ici sur la notion de type, rappelons simplement que le type d'un objet détermine deux propriétés :

- Un codage en mémoire ; par exemple, un type `int` génère une case mémoire dont le contenu binaire sera interprété comme un nombre entier codé en complément à 2.

---

[4] Voir, par exemple, B.W. Kernighan & D.M. Ritchie, Le langage C, paragraphes A4 et A11.

- Des opérations légales. Ce point est souvent oublié par les programmeurs débutants ; par exemple, la signification de l'opérateur de division (/) n'est absolument pas la même suivant qu'il s'applique à des types entiers ou à des types flottants[5].

Le second attribut, ou groupe d'attributs d'une variable est sa **classe de mémorisation**. La classe de mémorisation d'un objet précise :

- La portée du nom de l'objet dans le programme. Dit autrement, la classe de mémorisation d'un objet permet de répondre à la question « Où cet objet est-il connu ? ». En simplifiant un peu on peut distinguer les noms qui ont une **portée globale** et les noms qui ont une **portée locale**. Les noms globaux sont accessibles partout dans un programme, les noms locaux sont déclarés dans un bloc (par exemple dans le corps d'une fonction en C) et ne sont connus qu'à l'intérieur du bloc où ils sont déclarés[6].
- La durée de vie de l'objet. On distingue en C les objets **statiques**, qui existent tant que le programme est chargé en mémoire et les objets **dynamiques** qui sont créés lors de l'activation d'un bloc (appel de fonction, par exemple) et détruits lors de la sortie du bloc. Les variables automatiques du langage C sont des objets dynamiques, ils sont créés lors de l'appel d'une fonction, leur place en mémoire est libérée lors du retour de la fonction.

En C :

- Les fonctions sont des objets statiques globaux.
- Les variables automatiques, déclarées sans attribut particulier à l'intérieur d'un bloc délimité par des accolades, sont des objets dynamiques locaux.

```
int ma_fonc(...)
{
   int var_automat ;
...
}
```

- Les variables déclarées à l'extérieur de tout bloc délimité par des accolades sont des objets statiques globaux (variables externes).

```
int var_externe ;
Int ma_fonc(...)
{
...
}
```

- Les variables déclarées dans un bloc délimité par des accolades, avec l'attribut static sont des objets statiques locaux.

```
int ma_fonc(...)
{
   static int var_statloc ;
...
}
```

- Les objets dynamiques globaux sont des chimères qui n'existent donc pas.

Les objets statiques ont une adresse fixe en mémoire, nous verrons qu'ils génèrent un symbole au niveau de l'assembleur ; s'ils sont de plus globaux le symbole est transmis à l'éditeur de liens. Les objets dynamiques font appel à un mécanisme d'allocation dynamique de mémoire géré par le matériel au moyen d'une pile ; leur adresse peut changer entre plusieurs appels de la fonction à laquelle ils appartiennent, ils nécessitent l'introduction de modes d'adressage particuliers (adressage indirect) voisins de ceux qui sont générés par les pointeurs, nous les aborderons après avoir introduit les notions de modes d'adressage.

Pour terminer ces précisions générales, rappelons qu'en cas de conflit de nom entre un nom global et un nom local c'est le nom local qui l'emporte[7].

Objets statiques et symboles : nommer les adresses fixes

[5] Le lecteur est instamment convié à préciser ces différences.

[6] Le langage C introduit une nuance supplémentaire liée au fichier source, il est possible de créer un nom dont la portée est limitée au fichier source dans lequel il est déclaré.

[7] Dans les systèmes Unix historiques la table des processus était repérée par une variable globale nommée u. D'où un avertissement pour les programmeurs système : « *never, never, never use the letter U as a variable name…* » !

Les variables statiques et les fonctions ont des adresses fixes en mémoire. A partir de leur nom le compilateur génère un **symbole d'assemblage** qui représente cette adresse :

```
int var_glob ;
void somme_glob(void) ;
int arg_left, arg_right ;
void main(void)
{
arg_left = 3000 ;
arg_right = 5000 ;
somme_glob() ;
var_glob *= arg_left ;
}

void somme_glob(void)
{
var_glob = arg_left + arg_right ;
}
```

Devient en assembleur :

```
  11:      arg_left = 3000 ;
 0000 cc0bb8       LDD   #3000
 0003 7c0000       STD   arg_left
  12:      arg_right = 5000 ;
 0006 ce1388       LDX   #5000
 0009 7e0000       STX   arg_right
  13:      somme_glob() ;
 000c 160000       JSR   somme_glob
  14:      var_glob *= arg_left ;
 000f fc0000       LDD   var_glob
 0012 fd0000       LDY   arg_left
 0015 13           EMUL
 0016 7c0000       STD   var_glob
  15:  }
 0019 3d           RTS
  16:
  17:  void somme_glob(void)
  18:  {
  19:      var_glob = arg_left + arg_right ;
 0000 fc0000       LDD   arg_left
 0003 f30000       ADDD  arg_right
 0006 7c0000       STD   var_glob
  20:  }
 0009 3d           RTS
```

Le listing ci-dessus est généré par le compilateur C utilisé (MetroWerks). Il fournit le code assembleur entrelacé des instructions C d'origine. On notera en passant l'appel d'une fonction : **JSR** (*Jump Sub Routine*) et le retour de fonction : **RTS** (*ReTurn Subroutine*). On découvre également que le microcontrôleur HC12 contient d'autres registres que A et B : X et Y.
Dans le compilateur utilisé on constate que les noms des objets génèrent directement des symboles d'assemblage.
L'éditeur de lien fixe les adresses réelles des différents objets et nous rend un compte rendu du résultat (fichier *.map avec la chaîne MetroWerks) :

```
*************************************************************************
OBJECT-ALLOCATION SECTION
Name         Module       Addr   hSize   dSize     Ref    Section     RLIB
-------------------------------------------------------------------------
MODULE:      main.c.o --
- PROCEDURES:
main                      C056      1A      26       0   .text
somme_glob                C070       A      10       1   .text
- VARIABLES:
var_glob                  1000       2       2       3   .common
```

```
arg_left                1002     2     2     3   .common
arg_right               1004     2     2     2   .common
```

Ce compte rendu nous indique que la fonction main est chargée à l'adresse 0XC056 et que le code binaire correspondant occupe 26 octets (0X1A en hexa). Le code exécutable est chargé dans une **section d'édition de lien** qui porte traditionnellement le nom de section texte. Cette notion de section correspond à une organisation de la mémoire en zones ayant des rôles spécifiques : données globales, données locales, code exécutable etc. Les données globales de notre programme appartiennent (voir ci-dessus) à une section dite commune (*common*), pour partagée entre toutes les parties du programme.
Pour résumer ce paragraphe : un **nom** devient en assembleur un **symbole** qui représente une **adresse** en mémoire.

Modes d'adressage

Un objet en mémoire peut être référencé de plusieurs façons, on parle de **modes d'adressage**. Dans le programme précédent on en distingue deux :
L'adressage **immédiat** correspond à une valeur directement écrite dans le programme source. Il sert généralement à initialiser une variable :

```
cc0bb8        LDD   #3000
```

ou

```
ce1388        LDX   #5000
```

Les valeurs 3000 (0bb8 en hexa) ou 5000 (1388 en hexa) apparaissent dans le champ de l'instruction qui les utilisent, ce sont donc des données dont les valeurs sont consignées dans la mémoire programme, mélangées au code exécutable.
L'adressage **direct** correspond à la référence d'une donnée ou d'un module de programme par son adresse en mémoire[8] :

```
7c1002        STD   arg_left
…
7e1004        STX   arg_right

16C070        JSR   somme_glob
```

Le code du programme contient ici les adresses (par opposition aux valeurs) des données ou de la fonction appelée. On notera la correspondance entre les adresses qui apparaissent dans le code binaire exécutable et celles consignées dans le compte rendu de l'éditeur de liens.

Pointeurs et tableaux ou structures : adressages indirects

L'adressage direct et l'adressage immédiat correspondent à des adresses ou des valeurs immuables, dans beaucoup de cas il est intéressant de pouvoir faire du calcul d'adresse. Cela suppose que les adresses elles mêmes soient traitées comme des variables. En C ces idées conduisent aux notions de pointeurs et de tableaux ou de structures, la traduction en assembleur utilise la famille des adressages indexés pour effectuer des opérations sur les adresses.
Retourner la valeur cachée derrière une adresse :

```
char val_periph ;
...
void lire(void)
{
char *periph = (char *)0X100 ;
val_periph = *periph ;
}
```

Est traduit par :

```
0000 ce0100      LDX   #256
0003 e600        LDAB  0,X
0005 7b0000      STAB  val_periph
0008 3d          RTS
```

---

[8] Les codes qui suivent sont observés dans la mémoire physique, après que l'éditeur de lien ait calculé les adresses réelles, ils diffèrent de ceux qui ont été donnés précédemment qui étaient issus de la traduction en assembleur du programme en C, avant que les adresses aient été calculées par l'éditeur de lien. **Cherchez les différences**.

Sur cet exemple on peut remarquer que la variable automatique `periph`, tout à fait inutile, a disparu du code généré par le compilateur.
On voit également apparaître le rôle du registre X, sa fonction première est de faciliter le calcul d'une adresse, on parle de **registre d'index**. Le bout de programme qui suit suppose que les déclarations convenables aient été faites :

```
tabb[indice] = taba[indice] ;
  0000 f60000       LDAB  indice
  0003 b715         SEX   B,X
  0005 e6e20000     LDAB  taba,X
  0009 6be20000     STAB  tabb,X
```

L'instruction SEX effectue une copie du registre 8 bits source (ici B) dans un registre 16 bits de destination (ici X) en effectuant une extension de signe.
En compliquant un peu le programme on fait apparaître une autre arithmétique d'adresses (adressage indexé avec base et déplacement) :

```
tabb[indice] = taba[indice + 5] ;
  0000 f60000       LDAB  indice
  0003 b715         SEX   B,X
  0005 e6e20000     LDAB  taba:5,X
  0009 6be20000     STAB  tabb,X
```

En utilisant deux tableaux, avec une gestion d'indices non triviale, on voit l'intérêt de disposer de plusieurs registres d'index :

```
for(indice = 0 ; indice < 5 ; indice++)
  0000 790000       CLR   indice
tabb[2*indice + 3] = taba[indice + 5] ;
  0003 f60000       LDAB  indice
  0006 b714         SEX   B,D
  0008 b745         TFR   D,X // Transfer Register
  000a 59           ASLD
  000b b746         TFR   D,Y
  000d e6e20000     LDAB  taba:5,X
  0011 6bea0000     STAB  tabb:3,Y
  0015 720000       INC   indice
  0018 f60000       LDAB  indice
  001b c105         CMPB  #5
  001d 2de4         BLT   *-26 ;abs = 0003
```

Dans les programmes précédents la plupart des variables utilisées étaient globales, de façon à rendre lisible facilement le lien entre le langage C et l'assembleur. ce n'est évidemment pas une bonne pratique de programmation que de travailler systématiquement avec des variables globales.

Variables dynamiques : registres et pile, des variables sans adresse ou dont l'adresse n'est pas fixe

L'exemple le plus classique de variable automatique est une variable de boucle. Reprenons un exemple simple de manipulation de tableau : le calcul de la somme de ses éléments :

```
{
int i ;
var_glob = 0 ;
for(i = 0; i < 5; i++) var_glob = var_glob + taba[i] ;
  0000 ce0000       LDX   #0
  0003 c7           CLRB
  0004 ebe20000     ADDB  taba,X
  0008 08           INX
  0009 8e0005       CPX   #5
  000c 2df6         BLT   *-8 ;abs = 0004
  000e 7b0000       STAB  var_glob
...
}
```

L’indice du tableau (la variable i) n’apparaît pas sous son nom dans le code assembleur généré, elle est stockée directement dans le registre d’index X.
Evidemment on se pose la question de ce que va faire un compilateur si on lui demande de manipuler un plus grand nombre de variables automatiques. Faisons le produit scalaire de deux tableaux (les tableaux sont encore ici des variables globales) :

```
char prod_scal(void)
{
  0000 3b          PSHD
char prod, i ;
for(i = 0, prod = 0; i < 5; i++)prod = prod + taba[i] * tabb[i] ;
  0001 6981        CLR   1,SP
  0003 6980        CLR   0,SP
  0005 e681        LDAB  1,SP
  0007 b715        SEX   B,X
  0009 e6e20000    LDAB  taba,X
  000d a6e20000    LDAA  tabb,X
  0011 12          MUL
  0012 eb80        ADDB  0,SP
  0014 6b80        STAB  0,SP
  0016 6281        INC   1,SP
  0018 e681        LDAB  1,SP
  001a c105        CMPB  #5
  001c 2de7        BLT   *-23 ;abs = 0005
return prod ;
  001e e680        LDAB  0,SP
}
  0020 30          PULX
  0021 3d          RTS
```

Les variables automatiques i et prod ne génèrent aucun symbole d’assemblage, elles ont des adresses qui ne sont pas connues à la compilation du programme. En suivant pas à pas le code assembleur, on identifie aisément que ces deux variables sont stockées quelque part dans une zone mémoire adressée par le registre **SP**, le **pointeur de pile**. Ce registre, présent sur l’immense majorité des machines, est initialisé au lancement (boot) du premier programme qui s’exécute après la mise sous tension, et est ensuite géré automatiquement. Il gère la zone mémoire vers laquelle il pointe en une pile « dernier entré - premier sorti » : comme une pile d’assiettes. On range toujours une assiette au sommet de la pile et cette assiette est la première qui sera reprise. Les deux instructions de base de la gestion d’une pile sont donc « ranger », ici PUSH, et sortir, ici PULL.
En réalité le programme précédent ne range pas explicitement, il fait un rangement initial (PSHD) pour créer une place (deux octets) dans la pile, et libère la place à la fin de l’algorithme (PULX).
La variable prod de notre exemple est logée en sommet de pile (pointée par SP), le compteur de boucle i est logé juste en dessous, à l’adresse SP+1[9].
Il est clair que nos deux variables ont des adresses qui dépendent dynamiquement de l’ensemble du programme qui s’exécute sur la machine.

Les variables « volatiles »

Dans la gestion des périphériques les optimisations effectuées par un compilateur peuvent parfois réserver de mauvaises surprises.
Prenons par exemple un programme qui écrit sur un port parallèle et qui lit ce port, quelque chose du genre :

```
char attend(void)
{
char *periph = (char *)0X378 ;
*periph = 1 ;
while(!(*periph & 4));
return *periph ;
}
```

[9] Pour simplifier la compréhension, la plupart des machines ont des piles qui sont numérotées « à l’envers » : le sommet de pile est en adresse basse, le fond de pile en adresse haute. C’est évidemment un détail sans intérêt sauf pour l’initialisation du pointeur de pile.

Un optimiseur un peu imprudent peut considérer que le résultat du test de boucle est connu, et donc omettre de tester la valeur réelle de la case mémoire située à l'adresse 0X378. Pour empêcher le compilateur de supprimer des accès à une case mémoire, il suffit de déclarer cette case comme **volatile** :

```
char attend(void)
{
volatile char *periph = (char *)0X378 ;
*periph = 1 ;
while(!(*periph & 4));
return *periph ;
}
```

Cet adjectif indique au compilateur que quelles que soient les options d'optimisation tous les accès à la case mémoire pointée doivent être effectués.

Fonctions et sous programmes

Le découpage d'une application en un ensemble de petits modules relativement indépendant est à la base de la « programmation structurée ». Cest modules sont caractérisés par trois choses :

- Du code exécutable, donc un espace mémoire qui contient ce code,
- Des variables locales appartenant au module, et invisibles des autres modules,
- Un mécanisme de communication des données entre modules appelant et appelé, arguments d'appel et valeur(s) retournée(s).

Sous programmes : adresses d'entrée et adresse de retour

L'appel d'un sous programme représente un branchement, chargement du pointeur de programme à l'adresse de la première instruction du sous programme. Avant d'effectuer le branchement le programme appelant doit mémoriser l'adresse de la première instruction qui suit l'appel du sous programme, de façon à pouvoir reprendre l'exécution à la fin du sous programme.
Le mécanisme le plus classique de réalisation matérielle de cette sauvegarde de l'adresse de retour consiste à utiliser une pile :

- Le processeur sauvegarde dans la pile la valeur suivante du PC (programme appelant) lors de l'appel à un sous programme (instructions JSR, BSR ou CALL pour le HC12),
- Quand le sous programme appelé termine son exécution il est sensé laisser la pile dans l'état où il l'avait trouvée, l'instruction de retour (RTS ou RTC[10] pour le HC12) consiste donc simplement à retrouver l'adresse du programme appelant au sommet de la pile … si le programmeur ne s'est pas amusé à « bricoler » la pile dans le corps du sous programme ; ce genre de bricolage se termine généralement mal !

Les appels de sous programmes peuvent être construits avec différents modes d'adressage : absolus ou relatifs à la valeur courante du PC, indexés etc. D'où des variantes dans la longueur de l'instruction et son mnémonique en langage assembleur. Les compilateurs permettent généralement de choisir une politique globale cohérente au moyen de paramètres de compilation (par exemple : génération de code indépendant de la position en mémoire, où tous les appels seront relatifs à la valeur courante du PC, *position independant code*).

Fonctions ou procédures : arguments et valeur retournée

La notion de fonction rajoute à celle de sous programme un mécanisme d'échange d'informations entre programme appelant et programme appelé. Les compilateurs gèrent cette politique de façon transparente pour l'utilisateur. De façon générale il n'est donc pas indispensable de connaître en détail cette politique, sauf dans deux cas :

- Le programmeur souhaite écrire des modules dans différents langages, le cas le plus courant est évidemment l'écriture de fonctions en assembleur, ces fonctions devant être appelées par un programme écrit en C. Dans cette dernière situation le meilleur conseil que l'on puisse donner est d'écrire une fonction vide, qui décrit complètement l'interface avec l'extérieur (arguments d'appel et valeur

[10] Le couple CALL RTC correspond à une gestion de la mémoire sur des adresses qui dépassent la capacité du PC (16 bits). On introduit alors une mémoire paginée en sections de 64 kbytes, l'adresse de retour contient alors deux informations : le numéro de page et la valeur du PC.

retournée), faire traduire cette fonction vide en assembleur par le compilateur et partir de ce code assembleur généré. Cette méthode fonctionne avec n'importe quelle chaîne de développement.
La plupart des outils modernes de développement offrent des méthodes plus conviviales, sous forme de code assembleur mis à l'intérieur d'une fonction C, mais les détails d'écriture varient d'un compilateur à un autre (pragmas, instruction asm() qui se présente comme une pseudo fonction, etc.).
L'interfaçage entre langages évolués différents (le cas classique est celui des *subroutine* Fortran appelées d'un programme C). Une analyse détaillée des politiques des compilateurs est alors nécessaire. Là encore on trouve des outils qui permettent de panacher les langages (exemple de C et Pascal chez Borland, par exemple).

- Le deuxième cas où les détails de gestion de la pile sont incontournables concerne l'écriture de parties d'un système d'exploitation, mais c'est une autre histoire…

La méthode la plus classique de gérer le code de retour d'une fonction C est d'utiliser des registres.
Le lecteur attentif analysera avec profit le dernier exemple détaillé ci-dessus (le calcul d'un produit scalaire), il retracera les détails d'occupation de la pile et cherchera où est la valeur retournée.
Pour illustrer ce qui précède, modifions le calcul du produit scalaire de deux tableaux de caractères pour qu'il ait une allure raisonnable de fonction C :

```
int prod_scal_fonc(char *v1, char *v2, int taille)
{
int prod, i ;
for(i = 0, prod = 0; i < taille; i++)prod = prod + v1[i] * v2[i] ;
return prod ;
}
```

Ce programme de trois lignes en C illustre assez bien ce que le programmeur gagne à utiliser un langage évolué, surtout si on remarque que le programme écrit peut être compilé pour n'importe quelle cible.
Pour la famille HC12 et le compilateur MetroWerks, avec les options de compilation par défaut, on obtient :

```
int prod_scal_fonc(char *v1, char *v2, int taille)
{
  0000 6caa         STD   6,-SP
int prod, i ;
for(i = 0, prod = 0; i < taille; i++)prod = prod + v1[i] * v2[i] ;
  0002 c7           CLRB
  0003 87           CLRA
  0004 6c82         STD   2,SP
  0006 6c84         STD   4,SP
  0008 201e         BRA   *+32 ;abs = 0028
  000a ec8a         LDD   10,SP
  000c e382         ADDD  2,SP
  000e b745         TFR   D,X
  0010 e600         LDAB  0,X
  0012 b716         SEX   B,Y
  0014 ec88         LDD   8,SP
  0016 e382         ADDD  2,SP
  0018 b745         TFR   D,X
  001a e600         LDAB  0,X
  001c b714         SEX   B,D
  001e 13           EMUL
  001f e384         ADDD  4,SP
  0021 6c84         STD   4,SP
  0023 ee82         LDX   2,SP
  0025 08           INX
  0026 6e82         STX   2,SP
  0028 ec82         LDD   2,SP
  002a ac80         CPD   0,SP
  002c 2ddc         BLT   *-34 ;abs = 000a
return prod ;
  002e ec84         LDD   4,SP
}
  0030 1b86         LEAS  6,SP
  0032 3d           RTS
```

L’appel de la fonction précédente se fait par une instruction C qui génère le passage des arguments :

```
pro_scal = prod_scal_fonc(taba, tabb, taille) ;
  0003 cc0000       LDD   #taba
  0006 3b           PSHD
  0007 ce0000       LDX   #tabb
  000a 34           PSHX
  000b fc0000       LDD   taille
  000e 160000       JSR   prod_scal_fonc
  0011 1b84         LEAS  4,SP
  0013 7c0000       STD   pro_scal
```

Il est temps de faire une pose. Le lecteur est invité à **prendre une feuille de papier**, à dessiner la pile, en admettant, par exemple, que le pointeur de pile vaut 0X2000 avant l’exécution de la première instruction du bout de code ci-dessus, et de représenter tous les mouvements de pile entre le début de l’appel et l’affectation du résultat dans la variable pro_scal.
A titre d’indication : l’instruction « `STD 6,-SP` » remonte la pile de 6 octets en tout et place au sommet de cette pile les deux octets des accumulateurs A et B, de sorte qu’il reste quatre octets libres dans la pile. Retrouver l’ensemble de l’occupation de la pile faite par ce programme. Comment ont été passés les paramètres ? Où sont rangées les variables locales de la fonction ?

Le rôle de la pile : contexte d’une fonction

Si l’on a effectivement suivi l’exemple précédent en détail on est arrivé au résultat suivant :
Dans une fonction la pile est constituée de trois parties :

- Au **sommet** de la pile les **variables locales**,
- Sous cette zone la valeur de retour du PC dans le programme appelant (**adresse de retour**),
- **Sous** cette adresse de retour les **arguments passés** à la fonction dans la pile.

L’ensemble de ces zones porte le nom de **contexte d’exécution d’une fonction**.

Annexe : les instructions du cœur HC12

(Documentation Freescale : HC12 Users Guide)