Méthode de programmation

Je tiens à remercier chaleureusement Monsieur **Philippe Meyne**, enseignant à l'IUT GEII d'Evry pour sa participation importante à l'élaboration de ce cours.

Constat :

La programmation en assembleur est complexe donc dangereuse.

- Ecriture de toutes les structures
- Accès direct à la machine (risque d'écriture en zone mémoire par exemple)
- etc.

=> L'approche doit être méthodique.

L'écriture d'un logiciel ne se fait pas pour le plaisir d'écrire un logiciel mais elle s'effectue dans un cadre précis (cahier des charges).

I.Cycle de développement d'un système

L'expérience montre qu'un système, pour être opérationnel doit être conçu en sept étapes :

- Spécification
- Conception générale
- Conception détaillée
- Réalisation
- Tests unitaires
- Tests d'intégration
- Validation

Ces étapes sont reliées entre elles de la façon suivante :

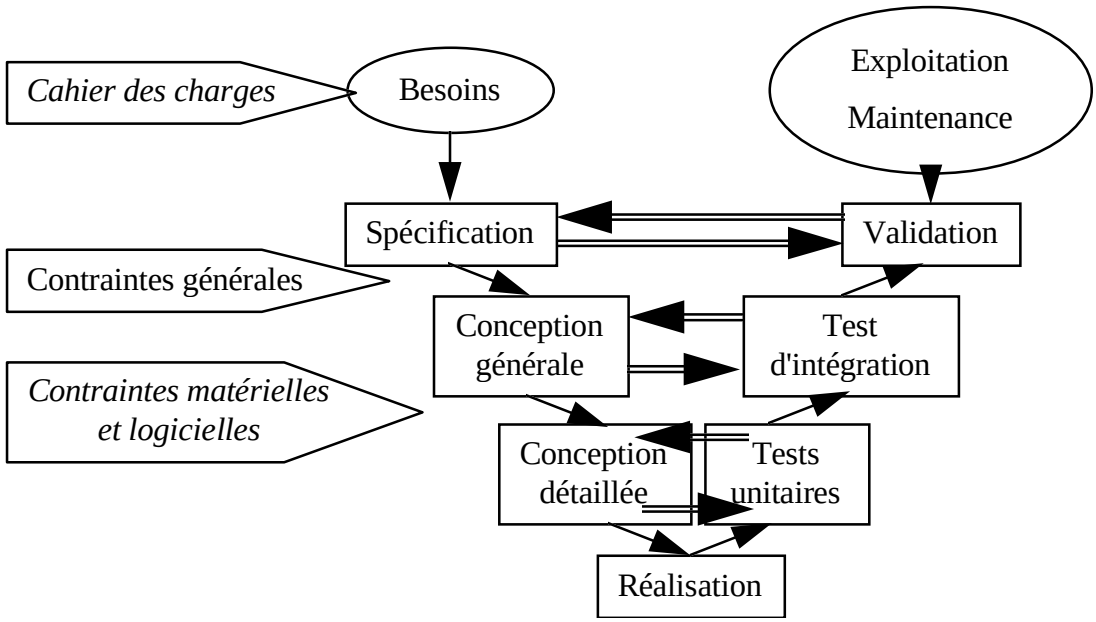


Figure I.1 : *Cycle en V de conception d'une application.*

Besoins : Cahier des charges + besoins non exprimés.

Ex : Voiture rouge…

On ne dit pas qu'il faut des roues, des rétroviseurs…

=> Sources de difficultés possibles.

Spécification du système : Le système est une boîte noire dont il faut connaître les entrées, les sorties et les relations entre elles.

Conception générale : Définition des grandes fonctions constituant l'architecture du système.

Ex :

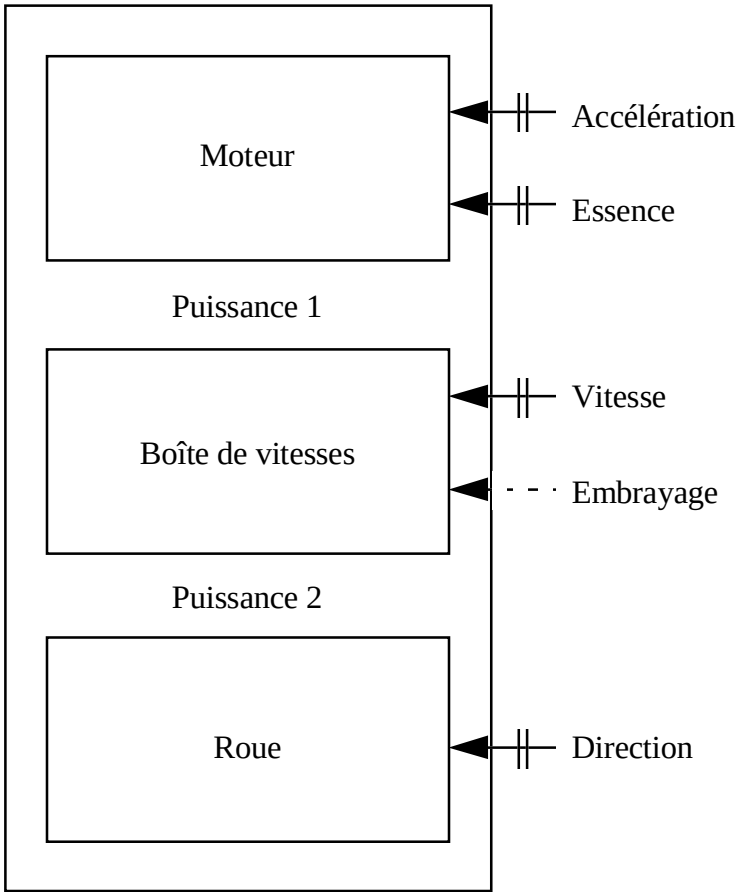


Figure II.2 : *Exemple de conception générale de voiture.*

Remarque : Apparaissent deux types de données : celles dont la présence est continue (essence) et celles dont la présence est fugitive (embrayage).

Evénement : présence furtive de l'information

Données : présence constante de l'information, donc mémoire.

Conception détaillée : Description de toutes les fonctions de base.

- Schéma pour les fonctions matérielles
- Pseudo-code pour les fonctions logicielles

Réalisation : Câblage et programmation du dispositif.

Phases de tests et de validation :

- Test OK => pas de problème.
- Test non OK => Comment retrouver le problème ?

=> Conception documentée

Chaque étape donne lieu à un document décrivant précisément les opérations effectuées, suffisamment précisément pour permettre des corrections ou d'évaluer le produit.

II.Modèle d'un objet informatique

Quel formalisme utiliser pour décrire un objet (entité logicielle ou matérielle) informatique ?

Exemple : En physique, on utilise le formalisme mathématique.

Informatique : Objet à trois faces

Structure : Comment opère-t-il dans l'univers ?
Informations : Données en entrées et sorties.
Comportement : Ce qu'il réalise.

Exemple : printf en C - printf("chaîne %d\t%f", i, x);

Modèle structurel :

Figure II.1 : *Modèle structurel de printf.*

Modèle informationnel : Dictionnaire des variables.

Nom	Type	Allocation mémoire. Classe	Commentaires
format	chaîne de caractères	registre… auto…	
i	entier	registre… auto…	
x	réel	registre… auto…	
écran	adresse	registre, adresse	destination de l'action

Modèle comportemental : Comment fonctionne printf ?

- Printf est une fonction standard de C, disponible dans la bibliothèque stdio.h, permettant l'affichage à l'écran de données formatées.
- Pseudo-code, grafcet, organigramme…

III.Modèle comportemental : le pseudo-code

Définition : Code en langage naturel qui permet de décrire le fonctionnement d'un objet informatique.
Code général : Il peut être traduit en C en assembleur, en Pascal… Chacune de ses structures possède un équivalent dans les langages classiques.

III.1.Structures de base

Affectation : <variable> ← <valeur>
<variable1> ← <Variable2>
Ex : i ← 1 La variable i est initialisée à la valeur 1.
i ← j Le contenu de j est transféré dans la variable i.

Test : si <condition> alors traitement 1
sinon traitement 2
finsi
si <condition> alors traitement 1
finsi

Boucle : tant que <condition>
faire traitement
fintq
répéter
traitement
jusqu'à condition.

III.2.Types de données

Scalaire : Une seule adresse.

Vecteur : Pointeur sur une adresse, associé à un index.

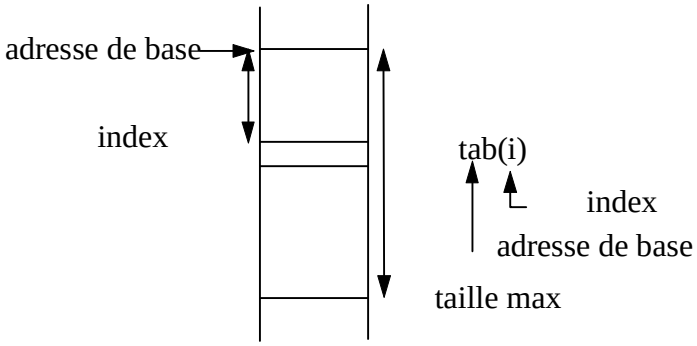


Figure III.1 : *Tableau à 1 dimension.*

Tableau à n dimensions : Combinaison de vecteurs.

Pile : Sauvegarde du contexte, passage de paramètres…

IV.Traduction des structures en assembleur : exemples

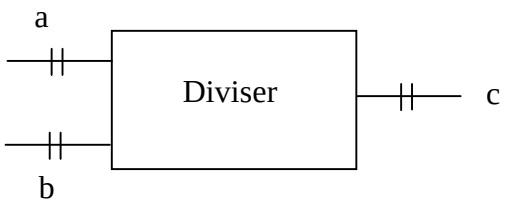
IV.1.Division entières de deux nombres

IV.1.1.Principe

$\frac{a}{b} = c$ a=b.c, a et b connus, c inconnu.

On fait croître c jusqu'à ce que le produit bc dépasse a.

IV.1.2.Modèle fonctionnel



IV.1.3.Pseudo-code

```
debut   c ← 0
        tant que (bc<a)
              faire  | c ← c+1
        fin tq
        si (bc>a) alors     | c ← c-1
```

fin

Remarque : Sortie d'une boucle *tant que* avec une condition fausse.

IV.1.4.Dictionnaire des variables

Nom	Type	Allocation mémoire. Classe	Commentaires
a	entier.b	d0	
b	entier.b	d1	
c	entier.b	d2	
bc	entier.w	d3	Résultat

IV.1.5.Codage en assembleur

```
debut   clr.b   d2          * c ← 0
        move.b  d2, d3
        mulu.b  d1, d3      * d3 ← b.c
tq      cmp.b   d0, d3      * tq (bc<a)
        bcc     fintq
        add.b   #$1, d2     * c ← c+1
        move.b  d2, d3
        mulu.b  d1, d3      * d3 ← b.c
        bra     tq
fintq   cmp.b   b3, d0      * si bc=>a
        bcs     oui         * aller à oui si bc<a
retour  rts                 * retour du sous-programme
oui     sub.b   #$1, d2
        bra     retour
```

Remarques :

- Expression de la condition de boucle tant que

  On sort de la boucle *tant que* si bc ≥ a soit C=0 (pas de retenue lors de la soustraction)

  => bcc : vrai si C=0

- Dans la boucle *tant que* on teste avant le traitement. Or, dans le pseudo-code proposé, le test utilise le résultat d'un calcul. Le codage serait plus simple avec une boucle *répéter jusqu'à*. On passe alors au moins une fois dans la boucle.

```
debut   move.b  #$0, d2
rep     add.b   #$1, d2     * répéter c ← c+1
        move.b  d2, d3
        mulu.b  d1, d3      * d3 ← b.c
        cmp.b   d0, d3      * jusqu'à bc ≥ a
        bcs     rep         * faire le saut si bc<a soit c=1 (d3-d0).
```

IV.1.6.A retenir

Structure de boucle tant que :

```
        tq      cmp.<format>    op1, op2
        bxx             fintq
        …
        bra             tq
fintq   …
```

Structure de boucle répéter jusqu'à :

```
rep     …
        cmp.<format>    op1, op2
        bxx             rep
        …
```

Structure de si <cond> alors <trait> :

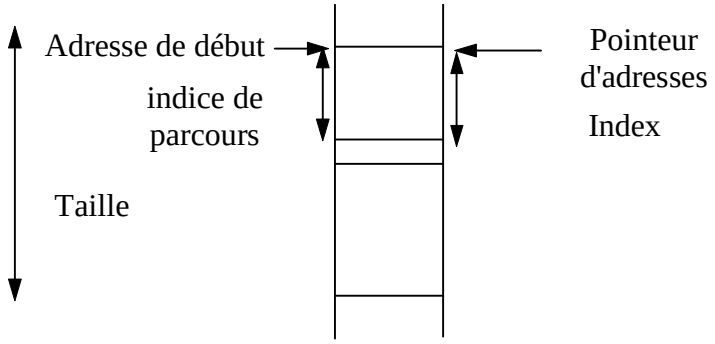
```
        cmp.<format>    op1, op2
        bxx             oui
        …
finsi   …
        bra             suite
oui     …
        bra             finsi
suite   …
```

IV.2.Test d'une zone mémoire

IV.2.1.Principe

On écrit dans une zone mémoire un caractère et on le relit. Si le caractère lu est identique à celui écrit, on continue ; sinon, on incrémente un compteur pour connaître le nombre de cellules défectueuses.

Problème : comment accéder à une zone mémoire ?



=> Tableau.

IV.2.2.Modèle fonctionnel

IV.2.3.Pseudo-code

```
debut     compteur ← 0
          index ← 0
          tant que (index < fin) faire
                tab(index) ← $55
                si (tab(index) ≠ $55 alors
                      compteur ← compteur + 1
                finsi
                index ← index + 1
          fintq
```

fin

IV.2.4.Dictionnaire des variables

Nom	Type	Allocation mémoire. Classe	Commentaires
compteur	entier.l	d0	
index	entier.l	d1	
tab	tableau d'entier.b	a0	pointeur

IV.2.5.Codage en assembleur

```
debut   lea     adresse, a0     * a0 pointe sur le tableau
        clr.l   d0              * compteur ← 0
        clr.l   d1              * index ← 0
tq      cmp.l   #taille, d1     * tant que index < fin
        beq     fintq
        move.b  #$55, (a0,d1.l) * tab(index) ← $55
        move.b  (a0, d1.l), d2  * relecture de la valeur
        cmp.b   #$55, d2        * si tab(index) ≠ $55
        bne     oui
finsi   add.l   #$1, d1
        bra     tq
fintq   rts
oui     add.l   #$1, d0
        bra     finsi
```

Remarques : - lea (Load Effective Adress) : permet d'initialiser un pointeur d'adresses.
- Accès au tableau : move.b (a0,d1), d2
d1 = d1.w par défaut

IV.2.6.A retenir

Initialisation d'un pointeur d'adresses

Gestion d'un tableau

V.Stratégie de conception

Modulaire descendante.

Principe : Découpage d'un problème complexe en un ensemble de fonctions de base.

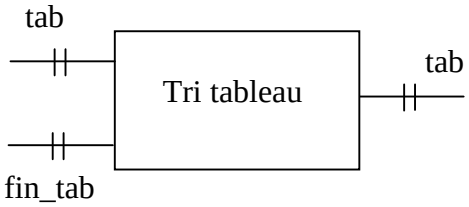
Intérêt : Débogage. On débogue chacune des fonctions puis les relations entre elles.

Exemple : Tri d'un tableau. Classer les éléments par ordre décroissant.

- Principe : ① On recherche le plus grand élément.
  ② On remplace le premier élément par le plus grand.
  ③ On recommence sur les éléments du tableau sauf le premier.
- Deux fonctions de base :
  ⊙ Rechercher le plus grand élément.
  ⊙ Remplacer un élément par un autre.
- Trois étapes :
  ❶ Fonction principale.
  ❷ Recherche du plus grand élément.

❸ Remplacement d'un élément par un autre.

❶ Modèle structurel :



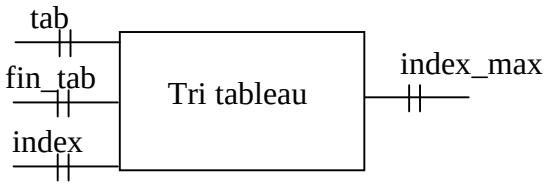
Modèle informationnel (dictionnaire des variables) :

Nom	Type	Allocation mémoire. Classe	Commentaires
tab	tableau de nombres	Registre d'adresse	Réserver de la place en mémoire
index	entier	Registre de données	Index courant
fin_tab	entier	Constante	Index de fin du tableau

Modèle comportemental (pseudo-code) :

```
début
    index ← 0
    tant que ( index <= fin_tab )
    faire
        rechercher_max
        Remplacer
        index ← index+1
    fintq
fin
```

❷ Modèle structurel :



Modèle informationnel (dictionnaire des variables) :

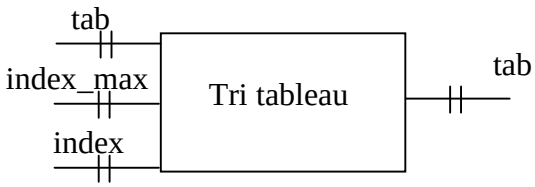
Nom	Type	Allocation mémoire. Classe	Commentaires
index_max	entier	Registre de données	
max	nombre	Registre de données	variable interne
indexi	entier	Registre de données	variable interne

Modèle comportemental (pseudo-code) :

```
début
    max ← tab(index)
```

```
indexi ← index
tant que ( indexi <= fin_tab )
faire
        si ( tab(indexi) > max )
        alors
                index_max ← indexi
                max ← tab( indexi )
        finsi
        indexi ← indexi+1
fintq
fin
```

❸ Modèle structurel :



Modèle informationnel (dictionnaire des variables) :

Nom	Type	Allocation mémoire. Classe	Commentaires
tampon	nombre	Registre de données	variable interne

Modèle comportemental (pseudo-code) :

```
début
        tampon ← tab(index)
        tab(index) ← tab(index_max)
        tab(index_max) ← tampon
fin
```

➢Code assembleur : Voir feuille.

VI.Application à la programmation en assembleur

VI.1.Gestion des données

Relation entre le type de donnée et leur mise en oeuvre en assembleur.

Scalaire : Adresse avec les différents types d'adressage.

Vecteur : Tableau à une seule dimension => adressage indirect indexé.

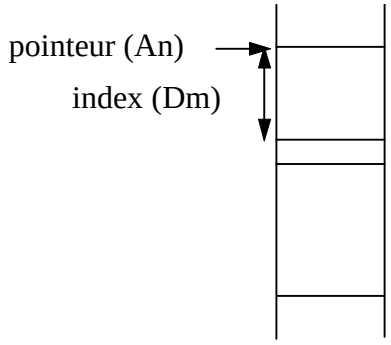


Figure V.1 : *Tableau à 1 dimension en assembleur.*

Exemple : move.b #32,(An,Dn)
Initialisation du pointeur d'adresse : lea <adr>,An
Cas particulier : Une chaîne de caractères est un tableau de caractères terminé par le caractère $00 (pour le C).

Matrice : Tableau à deux dimensions => adressage indirect indexé avec déplacement.

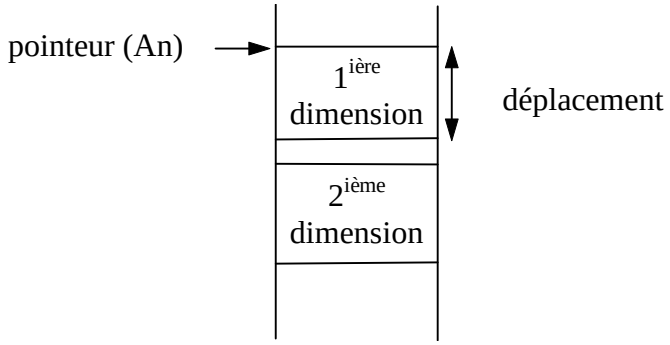


Figure V.2 : *Tableau à 2 dimensions en assembleur.*

Exemple : move.b #$5C,(An,Dm) accès à la première dimension.
move.b #$5C,$10(An,Dm) accès à la seconde dimension.
Initialisation du pointeur d'adresse : lea <adr>,An

Pile : FIFO (First In First Out) : parfois appelée file.
LIFO (Last In First Out)

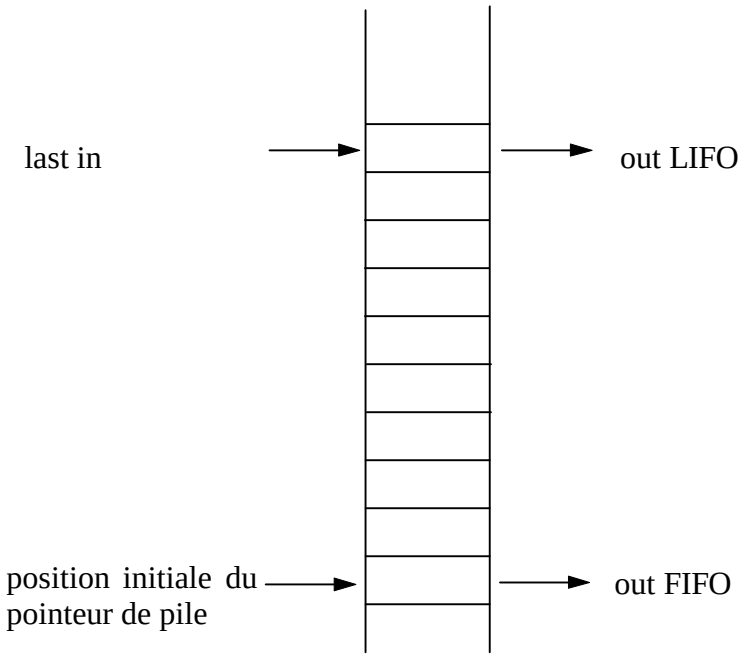


Figure V.3 : *Piles FIFO et LIFO.*

Mise en oeuvre : adressages indirects post-incrémenté et pré-décrémenté.
Empilement de 10 données :

```
        lea     $FF9100,A0      pointe sur une table
        lea     $FF9200,A1      pointeur de pile
        clr.b   D0              compteur
Pile    cmp.b       #10,D0
        beq     Fin
        move.b  (A0,D0),-(A1)
        add.b   #1,D0
        bra     Pile
Fin     …
```

Dépilement de type FIFO :

```
        lea     $FF9200,A1          pointeur de pile
        clr.b   D0                  init du compteur
FIFO    cmp.b   #10,D0
        beq     Fin_FIFO
        move.b  -(A1),(A0,D0)       dépilement
        add.b   #1,D0
        bra     FIFO
Fin_FIFO
        …
```

Dépilement de type LIFO :

```
        clr.b   D0                  init du compteur
LIFO    cmp.b   #10,D0
        beq     Fin_LIFO
        move.b  (A1)+,(A0,D0)       dépilement
        add.b   #1,D0
        bra     LIFO
Fin_LIFO
        …
```

Application : Sauvegarde du contexte lors d'un appel à sous programme.

```
SProg   movem.l D0-D7/A0-A6,-(A7)   empilement en pile
                                    système des contenus de tous les registres internes.
        movem.l (A7)+,D0-D7/A0-A7   dépilement
        rts
```

VI.2.Structures de base

Si…alors :

Si (condition) **alors**

faire

traitement

finsi

```
        cmp.<format>    op1,op2
        b condition vraie   étiquette

        traitement
étiquette
```

Si…alors…, sinon :

Si (condition) **alors**

faire

traitement1

Sinon

Faire

traitement2

finsi

```
        cmp.<format>    op1,op2
        b condition vraie   cas2
cas1
        traitement1

cas2
        traitement2
```

Tant que :

Tantque (condition)

faire

traitement

fintq

```
Tq   cmp.<format>      op1,op2          bra          Tq
     b condition fausse fintq            Fintq

     traitement
```

Répéter jusqu'à :

Répéter	Rep		
traitement		*traitement*	
		cmp.<format>	op1,op2
jusqu'à (condition)		b condition fausse	Rep

Remarque : L'utilisation de l'instruction cmp n'est pas obligatoire. En effet, certaines instructions positionnent les bits du registre code condition.

exemple :

i ← 10		move.b	
répéter		#10,D0	
i ← i-1			
jusqu'à (i=0)	Rep	sub.b	
		#1,D0	
		bne	Rep

Si le résultat de la soustraction est nul, le bit Z est positionné par l'instruction sub.