Conception d'un générateur de programmes de test de microprocesseurs

Annie Liothin

CONSERVATOIRE NATIONAL DES ARTS ET METIERS.

CENTRE AGREE
DE GRENOBLE (C.U.E.F.A)

MEMOIRE

présenté en vue d'obtenir
le diplôme d'ingénieur
du Conservatoire National des Arts et Métiers

en

Informatique

par

Annie LIOTHIN

CONCEPTION D'UN GENERATEUR DE PROGRAMMES DE TEST DE MICROPROCESSEURS

SOUTENU LE : 17 Février 1981

JURY

Président : L. BOLLIET

Membres : R. BOUTTAZ
F. GRILLOT
P. NAMIAN
C. ROBACH

CONSERVATOIRE NATIONAL DES ARTS ET METIERS.

CENTRE AGREE
DE GRENOBLE (C.U.E.F.A)

MEMOIRE

présenté en vue d'obtenir
le diplôme d'ingénieur
du Conservatoire National des Arts et Métiers

en

Informatique

par

Annie LIOTHIN

CONCEPTION D'UN GENERATEUR DE PROGRAMMES DE TEST DE MICROPROCESSEURS

SOUTENU LE : 17 Février 1981

JURY

Président : L. BOLLIET

Membres : R. BOUTTAZ
F. GRILLOT
P. NAMIAN
C. ROBACH

Le travail présenté dans ce mémoire a été réalisé dans le cadre de la Formation Permanente conjointement à l'Atelier de Microinformatique de Grenoble et dans l'équipe "Conception et Sécurité des systèmes" dirigée par Madame Gabrielle SAUCIER.

Je remercie trés chaleureusement les membres du Jury :

Monsieur le Professeur NAMIAN, Directeur de la Chaire de Mathématiques et d'Informatique du Conservatoire National des Arts et Métiers, qui m'a fait l'honneur d'accepter mon dossier de Thèse,

Monsieur le Professeur Louis BOLLIET, Directeur du Groupement d'Intérêt Scientifique Mini et Micro Informatique de Grenoble, de m'avoir accueillie au sein de l'Atelier de Microinformatique de Grenoble et de me faire l'honneur de présider le Jury.

Mademoiselle Chantal ROBACH, Chargé de Recherche au CNRS, qui m'a témoigné sa confiance en dirigeant mes travaux et qui m'a aidé avec efficacité et compétence,

Monsieur Raymond BOUTTAZ, Directeur Technique de l'Atelier de Microinformatique de Grenoble, qui m a toujours fourni l'aide matérielle dont j'ai eu besoin,

Monsieur François GRILLOT, Chef du service Test Automatique et Conception assistée sur Ordinateur à la Société Electronique Marcel Dassault, qui a bien voulu examiner ce travail et siéger dans ce Jury.

Je tiens aussi à remercier Monsieur Jean-Paul EYNARD, Ingénieur au CNRS, membre de l'Atelier de Microinformatique et Monsieur Raoul VELAZCO, Chercheur dans l'équipe "Conception et Sécurité des Systèmes" de l'IMAG pour leur aide précieuse lors de la réalisation de ce travail.

Que toutes les personnes de l'Atelier de Microinformatique qui m'ont apporté leur aide ou qui témoigné leur amitié trouvent ici l'expression de ma sincère gratitude.

Le Système de Traitement de Texte REDAK m'a été d'une grande utilité pour la frappe de ce mémoire.

TABLE DES MATIERES

INTRODUCTION

Les études sur le test fonctionnel d'un microprocesseur et de ses périphériques se situent dans le cadre d'un Projet Pilote de l'IRIA : "Sureté de Fonctionnement des systèmes : SURF".

Nos efforts ont porté sur la réalisation d'une bibliothèque de programmes de test pour microprocesseurs standards.

Le problème du test des microprocesseurs se pose à tout concepteur quelle que soit la gamme d'application envisagée (systèmes grand public ou à haute sureté de fonctionnement).

Ces programmes de test sont conçus de manière à être essentiellement utilisés en inspection d'entrée par le constructeur de systèmes informatiques à base de microprocesseurs.

Le test des microprocesseurs, et plus généralement des (V)LSI, se heurte aux problèmes suivants:

- on ne connait pas de schémas logiques équivalents du circuit : ce schéma pour être représentatif, doit être

déduit de la structure du masque lui-même qui est du domaine du secret industriel.

- d'autre part la connaissance des schémas logiques équivalents est d'une utilisation délicate. En effet, les hypothèses de pannes classiques (collages couts-circuits,...) qui ont permis des progrès considérables dans le domaine du test, sont remises en cause par les nouvelles technologies utilisées.

Ceci explique que la méthode de test proposée est de type fonctionnel, basée sur la notion d'erreur ou de mauvais fonctionnement du microprocesseur.

Ainsi, en première partie. on présentera cette méthode de test fonctionnel applicable à tout microprocesseur connu uniquement par son manuel d'utilisation et permettant une automatisation aisée.

La deuxième partie concerne l'aspect logiciel. Sachant que l'on veut produire à la demande et dans un délai très court, les programmes de test pour un microprocesseur donné. le travail exposé dans ce mémoire a consisté à créer un outil

-de description des microprocesseurs : le langage ROBIN.

-et de génération des programmes de test : le logiciel ROBIN.

En troisième partie. on donnera un exemple de mise en oeuvre pour la génération du programme de test pour le microprocesseur 6800 de Motorola.

En quatrième partie, on présentera notre participation à la conception d'un testeur qui permettra, entre autres, de valider les programmes de test.

CHAPITRE I. METHODOLOGIE DE TEST [1] , [2] , [3]

1. DESCRIPTION DES MICROPROCESSEURS

La généralisation d'une méthode de test à tout microprocesseur passe par la définition d'une description normalisée des microprocesseurs.
Cette description doit être un outil de représentation :

- par lequel tout microprocesseur puisse être décrit,

- sur lequel une méthode de test fonctionnelle puisse être appliquée.

Cette description, déduite des opérations symboliques du microprocesseur, est définie à partir de deux types de renseignements :

- des renseignements fonctionnels qui décrivent les fonctions du microprocesseur : le jeu d'instructions.

- des renseignements matériels qui sont les informations relatives à l'architecture du microprocesseur (format des données, type et nombre d'unités fonctionnelles ..) : les diagrammes logiques.

Ces renseignements sont disponibles à tout un chacun, en l'occurence le manuel utilisateur du microprocesseur.

1.1. Définitions préalables

La description des microprocesseurs fait appel à des notions de théorie des graphes [4] et nécessite les définitions préalables suivantes :

a) Arborescence à puits ou p-arborescence

. Dans un graphe orienté G = (X,U) on appelle puits un sommet $p \in X$ tel qu'il existe pour tout $x \in X$ un chemin de x à p dans G.

. L'ensemble des prédécesseurs d'un sommet $x \in X$ est l'ensemble $\Gamma^{-}(x)$ des sommets y tels que $(y,x) \in U$. On appelle source un sommet s tel que $\Gamma^{-}(s) = \emptyset$.

. On appelle arborescence à puits ou p-arborescence un arbre qui possède un puits.

. On appelle forêt de p-arborescences un graphe dont chaque composante connexe est une p-arborescence.

b) p-arborescence bipartie

Une p-arborescence est bipartie si l'ensemble X de ses sommets est partitionné en deux classes X1 et X2 telles que tout $u \in U$ a une de ses extrémités dans X1 et l'autre extrémité dans X2.

On ne considère dans la suite que des p-arborescences bipartites.

1.2. Graphe d'exécution abstraite

Tout microprocesseur est décrit par un ensemble de graphes caractérisant tous les fonctionnements possibles du microprocesseur ainsi que son architecture : pour cela on associe à chaque instruction du microprocesseur un graphe d'exécution abstraite représentant les fonctions et le matériel activés par cette instruction.

a) Définitions

Un graphe d'exécution abstraite est une p-arborescence bipartie ou une forêt de p-arborescences telles que :

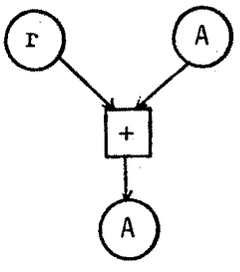
- aux noeuds du premier type sont associés les éléments de mémorisation utilisés par l'instruction,
- aux noeuds de deuxième type sont associées les micro-opérations (ou fonctions) activées par l'instruction,
- il y a un arc entre une micro-opération et un élément de mémorisation EMi si la micro-opération envoie le résultat du traitement dans EMi,
- il y a un arc entre un élément de mémorisation EMj et une micro-opération si la micro-opération traite la valeur contenue dans l'élément de mémorisation.
- un graphe d'exécution abstraite a, en outre, la propriété suivante : les feuilles du graphe (sources et puits) sont toutes des noeuds du premier type (éléments de mémorisation).

Un graphe d'exécution abstraite est dit simple s'il comporte une seule composante connexe sinon il est dit multiple.

Par convention de représentation graphique, les noeuds de premier type sont représentés par des cercles et les noeuds de deuxième type par des carrés.

b) Exemple : soit l'instruction ADD r qui réalise l'addition du contenu de l'accumulateur et du registre r

et stocke le résultat dans l'accumulateur. Le graphe d'exécution abstraite associé est:

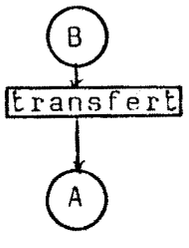


1.3. Exemples

Les concepts précédents sont illustrés par des instructions du microprocesseur ZILOG Z80.

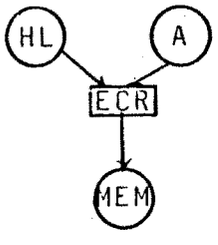
- Graphes d'exécution abstraite simples

load A,B



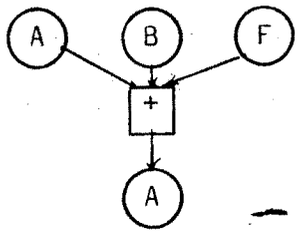
Charger le contenu du registre B dans A.

load (HL),A



Charger en mémoire d'adresse HL le contenu de A.

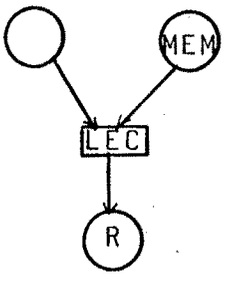
ADDC A,B



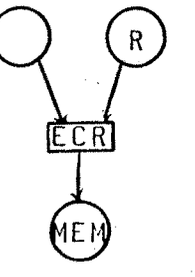
Addition entre A et B, avec retenue d'entrée.

Convention de représentation : pour une micro-opération de lecture ou écriture mémoire, on convient que la branche de gauche (incidente au noeud Lecture ou écriture) provient de la variable "adresse du mot lu ou écrit".

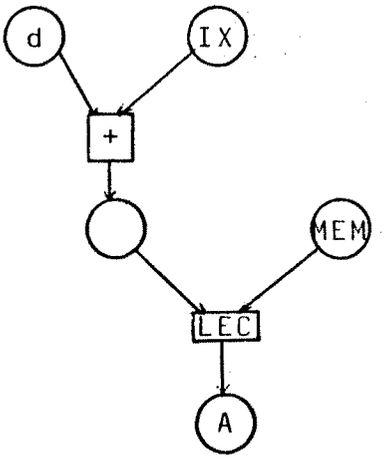
adresse du
mot lu



adresse du
mot écrit



LOAD A,(IX+d)



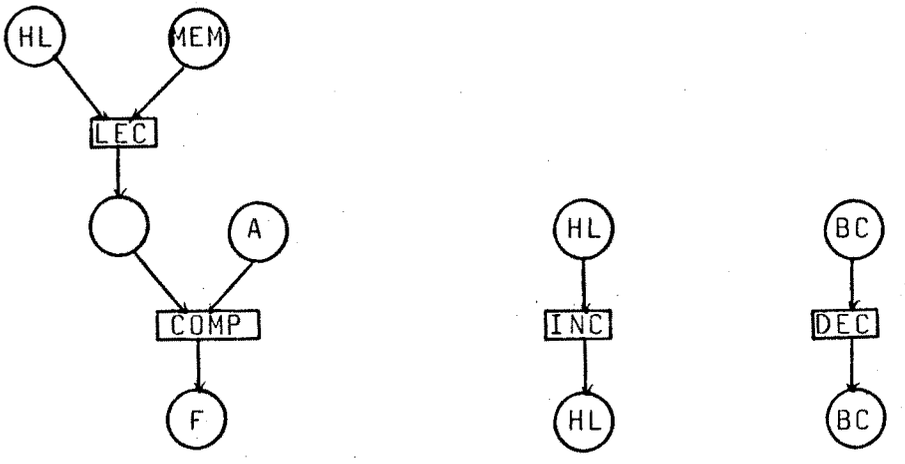
Un noeud de mémorisation intermédiaire peut ne pas être spécifié (non connu par l'utilisateur): il est alors représenté par un noeud vide (non étiqueté).

- Graphe d'exécution abstraite multiple

Instruction CPI :

Comparaison entre le contenu du mot mémoire dont l'adresse est contenue dans la paire de registres HL, et l'accumulateur A; le résultat est chargé dans le registre d'indicateurs F.

La comparaison est répétée tant que (A) ≠ (mem) et BC ≠ 0.



2. ANALYSE DES INSTRUCTIONS

Chaque instruction étant représentée par son graphe d'exécution abstraite, on recherche des propriétés significatives de ces graphes face au test. On recherche des critères :

- de complexité, mesurée par la quantité de matériel activé par l'instruction,
- d'accessibilité, mesurée par la facilité d'accès de l'information de test pour cette instruction.

On utilisera deux niveaux d'analyse mettant en jeu des renseignements de plus en plus précis :

- sur les graphes non renseignés,
- puis sur les graphes renseignés.

2 1. Analyse structurelle des graphes d'exécution abstraite

Il s'agit de l'analyse de la "complexité" des graphes ne portant aucun renseignement sur la signification des noeuds. Il s'agit de considérer les graphes "nus".
On peut essentiellement définir :

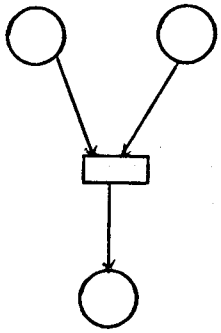
- une relation de dominance structurelle,
- des paramètres d'affinement structurels.

a) Relation de dominance structurelle

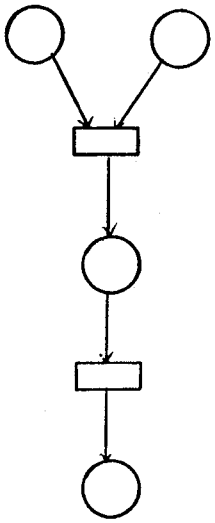
- Définition : le graphe d'une instruction I1 est stucturellement dominé par le graphe d'une instruction I2

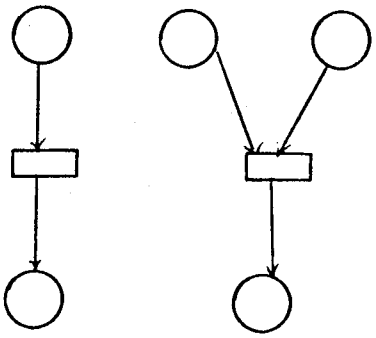
si le graphe de I1 peut être immergé dans le graphe de I2

- Exemple :

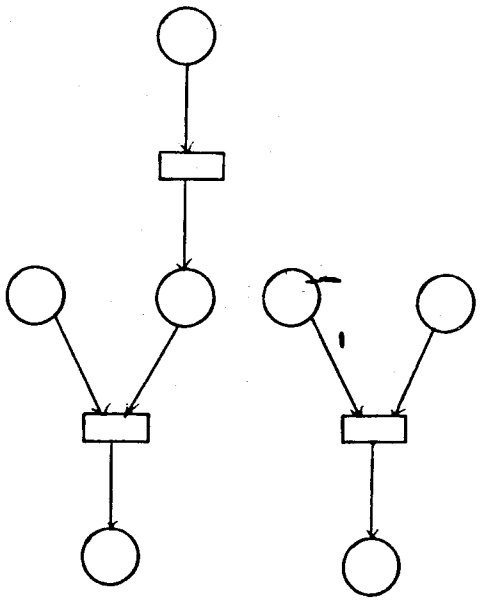


est dominé par





est dominé par



b) Paramètres d'affinement

Ce sont des paramètres classiques de la théorie des graphes on peut associer au graphe d'exécution d'une instruction :

- la profondeur ou nombre de couches de l'arborescence la plus profonde,
- le degré de multiplicité qui est le nombre de composantes connexes (graphe multiple).

2.2. Analyse fonctionnelle des graphes

Cette analyse se fait par rapport aux graphes renseignés et permet d'étudier l'accessibilité des instructions. Ainsi, on définit :

- une relation de dominance fonctionnelle,
- des paramètres fonctionnels attachés aux noeuds c'est à dire aux éléments de mémorisation et aux micro-opérations.

a) Relation de dominance fonctionnelle

Définitions :

- la commandabilité d'un élément de mémorisation traduit la facilité d'amener une information de test vers cet élément;
- l'observabilité d'un élément de mémorisation traduit

la facilité d'observer un résultat de test issu de cet élément.

Les éléments de mémorisation peuvent être classés selon leur commandabilité et leur observabilité : on leur associe une valeur de commandabilité t et une valeur d'observabilté t'.

Un élément de mémorisation est dit de commandabilté i (i>1) si l'information amenée à cet élément depuis l'extérieur doit passer par au moins un élément de commandablilité i-1.

Un élément de mémorisation est dit d'observabilité j(j>1) si l'information issue de cet élément doit transiter par au moins un élément d'observabilité j-1 avant d'être observé à l'extérieur.

Le monde extérieur est dit de commandabilité 1 et d'observabilité 1 : il est générateur de l'information de test et observateur du résultat de test.

Exemples :

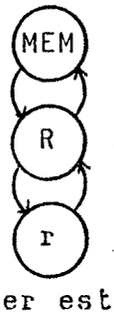
- La mémoire de travail et les périphériques, qui sont le monde extérieur, sont directement commandables et observables.

t=t'=1

- Les registres internes, pouvant communiquer directement avec l'extérieur, sont de commandabilité 2 et d'observabilité 2.

t=t'=2

- Les registres internes ne pouvant communiquer avec l'extérieur que par l'intermédiaire d'un registre du type précédent sont de commandabilité 3 et d'observabilité 3.



t=t'=3

- Un cas particulier est celui des valeurs immédiates, qui au moment de l'exécution de l'instruction sont des valeurs internes au microprocesseur et sont définies comme étant de commandabilité 0.

- L'accessibilité d'une instruction est définie par la commandabilité de ses sources et l'observabilité de ses puits. Elle est représentée par le couple (t,t') où

. t est la plus grande des valeurs de commandabilité des différentes sources,

. t' est la plus grande des valeurs d'observabilité des puits.

L'instruction est dite (t,t')-accessible

Relation de dominance fonctionnelle

Une instruction (t1,t'1)-accessible est fonctionnellement dominée par une instruction (t2,t'2)-accessible si et seulement si (t1,t'1)<(t2,t'2), sachant que

(a,b)<(c,d) <=> (a≤c et b≤d) et (a,b) ≠ (c,d).

b) Paramètres fonctionnels

Des paramètres d'affinement sont associés aux feuilles du graphe et aux micro-opérations.

- Pour les feuilles : le format de l'élément de mémorisation associé à la feuille (8 bits, 16bits,...) par rapport au format élémentaire qui est défini comme la taille de la plus petite information adressable en mémoire.
- pour les micro-opérations : on peut essentiellement distinguer :

. les micro-opérations effectuant un traitement ou manipulation de données,

. les micro-opérations de transfert/lecture/écriture,

et attacher un poids distinct à ces opérations.

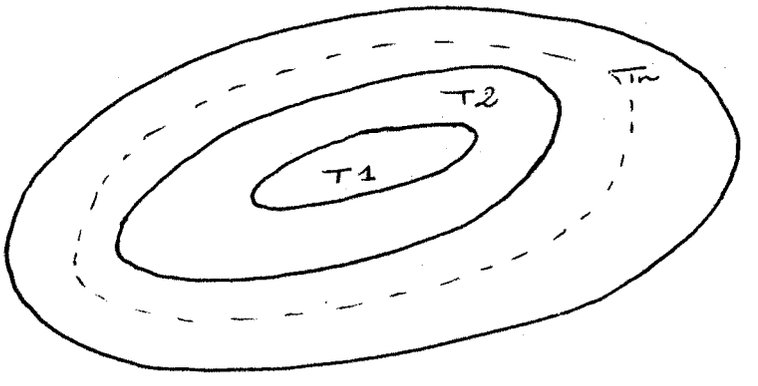
3. ORDONNANCEMENT DES INSTRUCTIONS POUR LE TEST

On recherche un ordre partiel ou total donnant un ordonnancement de test des instructions. Cet ordre est déduit des relations de dominance et des paramètres structurels et fonctionnels.

On étudie l'approche "boule de neige" ou "start-small" [5].

Cette approche consiste à tester au préalable une petite partie de matériel, en général la plus petite quantité permettant de dérouler les tests ultérieurs (hardcore) Pour chaque test supplémentaire, on considère une partie du matériel ne dépendant que des parties précédement testées (accès et observation de l'information de test).

Lorsqu'un test détecte une erreur. on en déduit que l'élément défectueux appartient au matériel rajouté pour ce test.



L'utilisation de ce type d'approche nécessite donc un ordonnancement des instructions face au test. Les avantages d'une telle approche sont les suivants :

- pour une instruction donnée on ne teste, parmi le

matériel et les fonctions activées par cette instruction, que la partie non encore testée par des instructions classées et testées précédement (test progressif et adaptatif),

- facilité de génération et d'observation de l'information de test pour le matériel testé,
- information sur la cause de l'erreur détectée et localisation à un ensemble d'instructions près,
- prise en compte possible des pannes multiples,
- programmes modulaires permettant une adaptation possible à tout microprocesseur et des facilités de modifications.

a) Partition structurelle

Etant donné un ensemble d'instructions, la relation de dominance structurelle induit une partition de ces instructions en classes CO...Ci,.. telle que : une instruction appartient à Ci si et seulement si elle n'est dominée par aucune autre instruction $\in \{CO \cup \ldots \cup Ci-1\}$.

b) Affinement structurel

Etant donné un ensemble d'instructions, les paramètres structurels

- degré de multiplicité.
- profondeur,

des graphes d'exécution abstraite associés aux instructions, induisent une partition de ces instructions

en sous-classes Ck,p telles que : une instruction appartient à Ck,p si et seulement si elle est de degré de multiplicité k et de profondeur p.

c) Partition fonctionnelle

Etant donné un ensemble d'instructions, la relation de dominance fonctionnelle induit une partition de ces instructions en blocs B0,..,Bi,.. tels que : une instruction appartient à Bj si et seulement si elle n'est fonctionnellement dominée par aucune autre instruction $\in \{B0 \cup .. \cup Bj-1\}$.

d) Affinement fonctionnel

Etant donné un ensemble d'instructions, les paramètres fonctionnels

- taille des éléments de mémorisation
- type de micro-opération.

des graphes d'exécution abstraite associés aux instructions, induisent une partition de ces instructions en sous-blocs Be,t où

- e représente le format maximum des sources et des puits.
- t indique le type de micro-opération : t=1 pour une micro-opération de traitement, t=0 sinon.

Ces blocs sont tels que : une instruction appartient à Be,t si le format maximum de ses sources ou puits est égal à e et la micro-opération réalisée est de type t.

e) Ordonnancement des instructions pour le test

Dans une approche de type boule de neige , on définit l'algorithme d'ordonnancement des instructions pour le test comme suit :

1) Partition structurelle sur l'ensemble des instructions du microproceseur donnant l'ensemble ordonné $\{C0, .,Cn\}$. Les instructions de la classe Ci sont testées avant les instructions de la classe Ci+1.

2) Pour chaque classe Ci ainsi définie, affinement structurel: $Ci=\{Ck,p\}$. Les sous-classes sont ordonnées de telle sorte que : Ck,p est testée avant Ck',p' si et seulement si (k < k') ou (k = k' et p < p').

3) A l'intérieur de chaque sous-classe ainsi définie, partition fonctionnelle donnant l'ensemble ordonné $\{B0,....Bm\}$. Les instructions d'un bloc Bj sont testées avant les instructions du bloc Bj+1.

4) Pour chaque bloc Bj ainsi défini, affinement fonctionnel $Bj=\{Be,t\}$. Les sous-blocs sont ordonnés de telle sorte que Be,t est testé avant Be',t' si et seulement si (e < e') ou (e = e' et t < t').

4. LES ALGORITHMES DE TEST

L' objectif du test est de détecter les erreurs ou mauvais fonctionnements possibles, en raisonnant sur les graphes d'exécution abstraite.

Le test d'une instruction comporte deux phases de vérification :

- vérification d'identité du graphe d'exécution abstraite associé à l'instruction,
- vérification des noeuds du graphe.

4.1. Définitions préalables

a) Représentation d'une instruction

Une classe ou un groupe d'instructions est entièrement défini par son graphe d'exécution abstraite et peut être représenté comme suit :

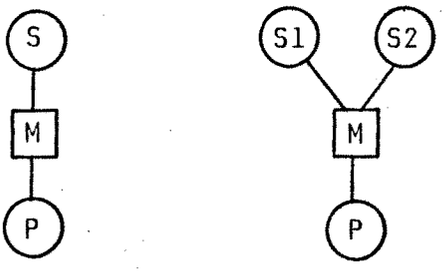
B = ({sources} , {micro-opérations} , {puits})

L'ensemble {sources} est l'ensemble des éléments de mémorisation où l'on doit générer l'information de test.

L'ensemble {puits} est l'ensemble des éléments de mémorisation où l'on doit observer un résultat de test.

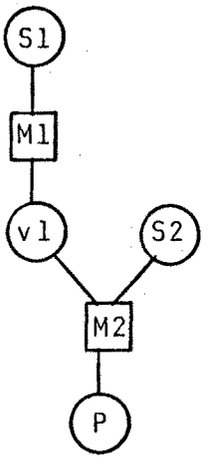
Plus précisément on donnera à une classe une représentation infixée.

Exemples :



(S,M,P) ({S1,S2} ,M P) (S1,M1,P1) (S2,M2,P2)

graphes simples forêt de p-arborescences



graphe composé

(S1,M1,v1);({v1,S2} ,M2,P2)

b) Domaine de variation Dv

Pour chaque classe, on précisera le domaine de variation des sources, des puits et des micro-opérations. Les éléments de mémorisation sont classés en groupes homogènes selon leur format et leur mesure d'accessibilté.

Pour le Z80, on définit 5 groupes :

H (hardcore) = {mémoire extérieur valeurs immédiates},

R = {A,F,B,C,D,E,H L} c'est à dire l'ensemble des registres pour lesquels (t,t') = (2,2) et de format 8

bits,

V = {SP,IX,IY} les registres pour lesquels (t.t') = (2,2) et de format 16 bits.

T = {A',F'.B',C',D',E',H',L'} l'ensemble des registres de 8 bits pour lesquels (t,t') = (3,3),

R2 = {BC,DE,HL} les paires de registres.

Pour une instruction donnée, le domaine de variation-sources est le domaine auquel appartient la source et le domaine de variation-puits le domaine auquel appartient le puits.

c) Domaine de conformité

Pour un groupe d'instructions (défini par la mesure d'accessibilité et le format des éléments de mémorisation) on définit le domaine de conformité comme l'union des domaines de variation-sources des différentes instructions de la classe.

Remarque : On considère que le domaine de variation de l'ensemble hardcore est vide.

Exemple :

- LD r,n	Dv(S) = ∅
	Dv(P) = R
- LD HL,(nn)	Dv(S) = ∅, Dv(P) = R2
LD rr',(nn)	Dv(S) = ∅, Dv(P) = R2
LD IX,(nn)	Dv(S) = ∅, Dv(P) = V
LD IY,(nn)	Dv(S) = ∅, Dv(P) = V

Pour l'ensemble des instructions, Dc = $\emptyset$ et Do = $\{R2 \cup V\}$

c'est à dire Do = $\{BC, DE, HL, IX, IY, SP\}$

4.2. Vérification de l'identité du graphe

Il s'agit de vérifier que le graphe réellement exécuté est bien le graphe décrit : on vérifie ici la conformité de l'activation.

Pour cela on considère trois types d'erreurs:

a) Erreur sur le choix des sources

- la source requise n'est pas sélectionnée,
- une autre source que celle requise est simultanément sélectionnée,
- une autre source est sélectionnée à la place de celle requise.

b) Erreur sur le choix du puits

- le puits requis n'est pas sélectionné,
- un autre puits que celui requis est simultanément sélectionné,
- un autre puits est sélectionné à la place du puits requis.

c) Erreur sur le choix de la micro-opération

- la fonction requise n'est pas exécutée,
- une autre fonction est activée simultanément,
- une autre fonction est activée à la place.

Le test de conformité du graphe d'exécution abstraite revient à vérifier le séquenceur du microprocesseur, tout au moins en partie. On vérifie que les commandes d'activation des éléments de mémorisation et des unités fonctionnelles sont générées correctement.

Algorithme de test de conformité

Soit un ensemble d'instructions dont les domaines de conformité et d'observation sont respectivement Dc et Do.

Soit O1,O2,..,Op les micro-opérations activées par les instructions de cet ensemble.

Pour une instruction $I = \{S , Oi , P\}$. l'algorithme de test d'identité est le suivant :

- initialiser S = X, Dc -S = Y, Dv(P) = Y,
  tels que $Oi(X) \neq Oj(X)\ \forall j \neq i$
  et $Oi(X) \neq Y$;
- exécuter l'instruction;
- observer Do.

4.3 Vérification des noeuds du graphe d'exécution abstraite

Sachant que les noeuds du graphe sont correctement activés, il reste à vérifier le bon fonctionnement de ces noeuds.

a) Test des éléments de mémorisation

On peut vérifier :

- soit un ensemble de registres considéré comme un plan mémoire,
- soit des registres indépendants.

Le meilleur test est celui qui considère des registres comme un plan mémoire mais il n'est pas toujours applicable : registres de différents niveaux d'accessibilité, formats différents, registres spéciaux.
On peut noter qu'un test global sur l'ensemble des registres pourra être généralement appliqué pour des microprocesseurs où les registres sont banalisés.

Les hypothèses de pannes retenues sont celles généralement admises lors du test des mémoires ; la méthode de test utilisée sera l'une des méthodes de test classiques de mémoires : "Galloping", "Marching".... [6] .

b) Test des micro-opérations

Les unités de traitement d'un microprocesseur (UAL, circuits de décalages, compteurs,...) se vérifient par le bon déroulement de leurs actions, c'est à dire le bon déroulement de l'ensemble des micro-opérations. Ceci est d'autant plus intéressant que certaines instructions impliquent l'activation de plusieurs unités de traitement.

Le test des micro-opérations revient à déterminer un ensemble de valeurs pour les opérandes d'entrée suffisant pour assurer le bon fonctionnement de ces opérations. La méthode de test est fondée sur une analyse algébrique des opérations [3].

4.4 Implémentation du test

Pour une instruction donnée trois ensembles de vecteurs de test sont ainsi définis :

- un ensemble Vg permettant d'assurer la conformité du graphe d'exécution abstraite associé à l'instruction,

- un ensemble Vm qui est l'ensemble des vecteurs de test vérifiant tous les éléments de mémorisation utilisés par l'instruction,

- un ensemble Vo qui est l'ensemble des vecteurs de test vérifiant le bon fonctionnemnt des

micro-opérations activées par l'instruction.

La procédure de test d'une instruction peut être représentée comme suit :

1) Mise à jour

- Lister les éléments de mémorisation et les micro-opérations de cette instruction qui ont été testées au cours d'un pas de test antérieur.
- Retirer des ensembles de test Vm et Vo les vecteurs de test qui sont relatifs à ces éléments de mémorisation et à ces micro-opérations. On obtient les ensembles Vm' et Vo'.

On définit l'ensemble d'initialisation comme :

$$Vg \cup Vm' \cup Vo'$$

2) Procédure d'initialisation : INIT

Cette procédure a pour but de charger un vecteur de test $Tk = \{Vg \cup Vm' \cup Vo'\}$ dans un ensemble d'éléments de mémorisation donné :

- soit initialisation du domaine de conformité Dc dans la phase de test d'identité du graphe,
- soit initialisation des seules sources du graphe.

Remarque : On entend par vecteur de test l'ensemble des valeurs chargées dans les éléments de mémorisation.

On pourra considérer deux types d'initialisation :

- une procédure d'initialisation "hardcore" qui permettra de charger, en début de test, une donnée dans un élément de mémorisation en utilisant le minimum de matériel Ce sera généralement, pour les registres d'accessibilté (2,2), un chargement par valeur immédiate.
- une procédure d'initialisation "minimale" permettant de charger un élément de mémorisation en un minimum de temps et de place mémoire. Ce sera un chargement direct à partir de la mémoire extérieure lorsque ce transfert aura été testé.

3) Procédure d'exécution : EXEC

L'instruction est alors exécutée.

4) Procédure d'observation : OBSV

Cette procédure permet de sortir vers une mémoire extérieure les résultats du test pour une comparaison ultérieure avec des résultats justes pré-établis.
Le choix du vecteur de test Tk nous conduit à observer

- soit les éléments du domaine d'observation Do dans la phase du test de l'identité du graphe,
- soit seulement les puits du graphes,
- soit, éventuellement, l'état du microprocesseur c'est à dire l'ensemble de tous les registres.

5) Itération

On itère les procédures 2,3 et 4 pour tous les k tels que:

$$Tk = \{Vg \cup Vm' \cup Vo'\}$$

CHAPITRE II. REALISATION D'UN GENERATEUR DE PROGRAMMES DE TES

1. INTRODUCTION

Le but de ce travail est de fournir une bibliothèque de programmes de test pour microprocesseurs.

Pour répondre à cet objectif, il faut donc définir un logiciel de test. qui soit capable de fournir des programmes de test pour un microprocesseur quelconque, à la demande et dans un délai très court.

Ce système de production de programmes de test pour microprocesseurs, à partir de leur description symbolique, s'appuie sur les deux remarques suivantes:

- l'expérience acquise dans ce domaine nous laisse prévoir un programme de test comportant plusieurs milliers d'instructions pour un microprocesseur donné . Ces instructions sont en langage d'assemblage car aucun langage de haut niveau existant ne peut convenir.

Plusieurs mois de programmation sont nécessaires pour un microprocesseur (une mise au point de 500 instructions par homme/mois est en général retenue). D'autre part ce travail sera à recommencer pour chaque nouveau microprocesseur.

A titre d'exemple, un programme de test fonctionnel, pour le microprocesseur Z80 de ZILOG a été écrit à l'Atelier de Micro-informatique de Grenoble.

Ce programme de test comportait environ 5000 instructions et a demandé 4 à 5 mois pour être opérationnel. De plus, l'écriture de ce programme de test en assembleur a été facilitée par l'utilisation d'un macro-assembleur.

- la méthode de test implique un aspect itératif dans l'écriture et l' exécution des instructions sous test. Chaque instruction testée doit se répéter plusieurs fois:

  - itération sur un choix d'initialisation particulier à l'instruction,
  - itération sur un ensemble d'opérandes de test,
  - itération sur la destination des résultats obtenus.

Exemple: l'instruction "LD A,r" du ZILOG Z80 avec r∈(B,C,D,E,H,L)

Cette instruction devra s'exécuter pour chaque registre r:

LD A,B
LD A,C
.....
LD A,L

De plus, un même ensemble d'opérandes peut s'appliquer à plusieurs instructions, et sera défini en fonction de la phase de test.

La conclusion qui s'impose est qu'il faut créer un outil de description des microprocesseurs et de génération des

programmes de test. Cet outil doit être universel: à partir de la description d'un microprocesseur par ses graphes d'exécution abstraite (développée dans le chapitre précédent) et des algorithmes de test pour chacun de ses graphes, cet outil fournira des instructions binaires exécutables pour ce microprocesseur.

Ces instructions constitueront donc le programme de test spécifique à ce microprocesseur.

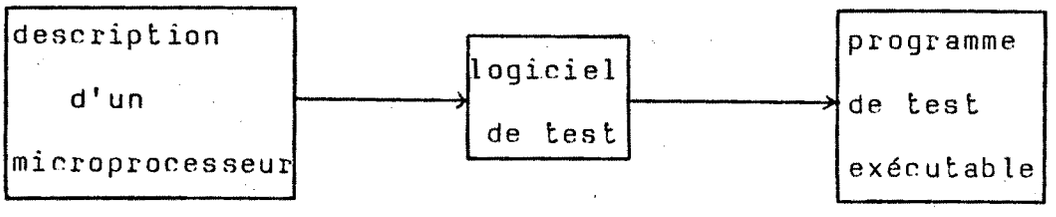


Figure 1. Outil de production des programmes de test

2. STRATEGIE DE REALISATION

Deux étapes de réalisation ont été prévues, elles s'appuieront sur les possibilités offertes par des outils logiciels développés à l'Atelier de Micro-informatique de Grenoble. Ces outils sont un générateur d'analyseurs (GASEL) et un générateur d'assembleurs (GAGE).

Première étape :

Elle consiste en un outil dit "minimal" de génération "semi-automatique" de programmes de test.
Le logiciel "minimal" est appelé ROBIN.
Il permet d'obtenir des programmes de test en assembleur pour certains microprocesseurs "cobayes" représentatifs et d'étudier leur comportement en environnement réel sur des outils de développement disponibles à l'Atelier de Micro-informatique (Tektronix,Exorciser).

Le logiciel ROBIN "minimal" est un outil non entièrement automatique, mais sa réalisation nous a permis de tirer divers enseignements pour la deuxième étape : le logiciel ROBIN "final".
Cette première étape est opérationnelle.

Deuxième étape

La deuxième étape consistera en une étude plus approfondie sur les possibilités du test. Elle devra nous conduire à un outil de génération de programmes de test "quasi-automatique" : le logiciel ROBIN "final".

Le logiciel ROBIN "final" devra, en plus du logiciel ROBIN "minimal". introduire l'aspect automatique du test

- dans la classification des instructions et leur ordonnancement pour le test et dans la détermination des opérandes pour chaque instruction de la classe,

- dans le choix de l'algorithme de test pour chaque classe.

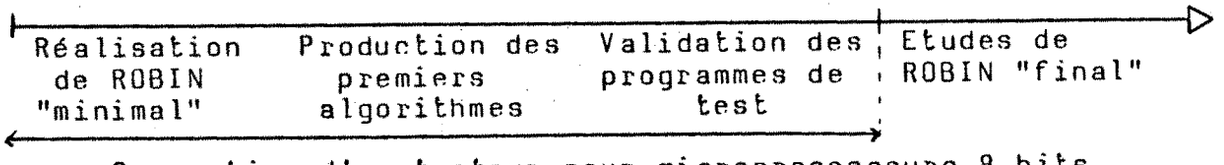
Suite à la première et parallèlement à la deuxième étape, on conduit les études sur la conception d'un testeur de microprocesseurs 8 bits qui permettra de valider les programmes de test produits par le système ROBIN en environnement réel.

Ainsi, à l'aide du logiciel ROBIN, l'écriture d'un programme de test pour un microprocesseur n'exigera qu'une semaine à 15 jours de travail au lieu de 4 à 5 mois.

Mais, dans la première étape, la procédure n'est pas entièrement automatique. La classification des instructions ainsi que la détermination des opérandes de test ne sont pas automatisées, il en résulte donc un travail préliminaire de préparation.

C'est pourquoi, nous développerons un outil logiciel ROBIN "final" qui sera plus souple et plus complet.

Ces étapes conduisent à la définition et à la réalisation de deux niveaux d'un langage spécifique au test fonctionnel : le langage ROBIN, ce langage facilitant l'écriture des algorithmes de test.



Conception d'un testeur pour microprocesseurs 8 bits

3. LES OUTILS LOGICIELS EXISTANTS [7], [8], [9]

Ces outils ont été développés à l'Atelier de Micro-informatique de Grenoble.

3.1. Le système de production d'assembleurs : GAGE

Le système GAGE est un logiciel de développement (en fortran) qui, à partir d'une description symbolique d'un assembleur d'un microprocesseur particulier, génère l'assembleur croisé de ce microprocesseur.
Cette description symbolique est faite dans un langage appelé : le langage GAGE.

a) Le logiciel GAGE

Le logiciel GAGE est constitué de trois parties distinctes (cf. figure 2.1.) :

- l'analyseur du langage GAGE,
- une librairie de programmes,
- des actions sémantiques.

L'analyseur syntaxique du langage GAGE est produit à partir d'un générateur d'analyseurs appelé : GASEL (cf. §3.2.).

La librairie de programmes est constituée des programmes communs de service (édition de listings, gestion des identificateurs,...).

Les actions sémantiques constituent le générateur (GAGE lui-même) qui traduit sous forme de programmes et de tables les descriptions des assembleurs qui lui sont fournies.

b) Les assembleurs

Les assembleurs sont également en Fortran et comportent trois parties (cf. figure 2.2.) :

- une analyse syntaxique commune à tous les assembleurs,
- des parties spécifiques à un assembleur particulier,
- une librairie de programmes communs.

L'analyseur syntaxique est aussi généré par le logiciel GASEL.

Les parties spécifiques à un assembleur sont produites par le logiciel GAGE.

Ainsi, ce système GAGE nous permet de disposer de tous les assembleurs dont nous aurons besoin.

3.2. Le Générateur d'Analyseurs Syntaxiques et Lexicographiques: GASEL

Le générateur d'analyseurs syntaxiques et lexicographiques (GASEL) a été créé à l'origine pour faciliter la réalisation du programme GAGE (cf. figure 2.1.), et la réalisation des assembleurs (cf. figure 2.2).

GASEL est un logiciel (en Fortran) qui, à partir de la description symbolique d'un automate reconnaissant un langage L donné, génère des tables d'analyse syntaxique et lexicale propres à ce langage.

La description de cet automate se fait dans un langage spécifique : le langage GASEL, tenant compte des trois niveaux classiques des techniques de compilation:

- le codage,
- la lexicographie,
- l'analyse syntaxique et sémantique.

Les tables d'analyse sont sous la forme de tables Fortran qui sont interprétées par un programme spécifique : l'Analyseur.
La description de la lexicographie mène à la génération d'une table qui est l'automate lexical.
De même, la description de l'analyse syntaxique mène à la génération des tables propres à ce langage L.
La partie sémantique est traitée par des actions que l'utilisateur déclare dans la description de l'automate syntaxique . L'écriture de ces actions sémantiques, ainsi répertoriées, est à la charge de l'utilisateur sous la forme de un ou plusieurs programmes en Fortran.

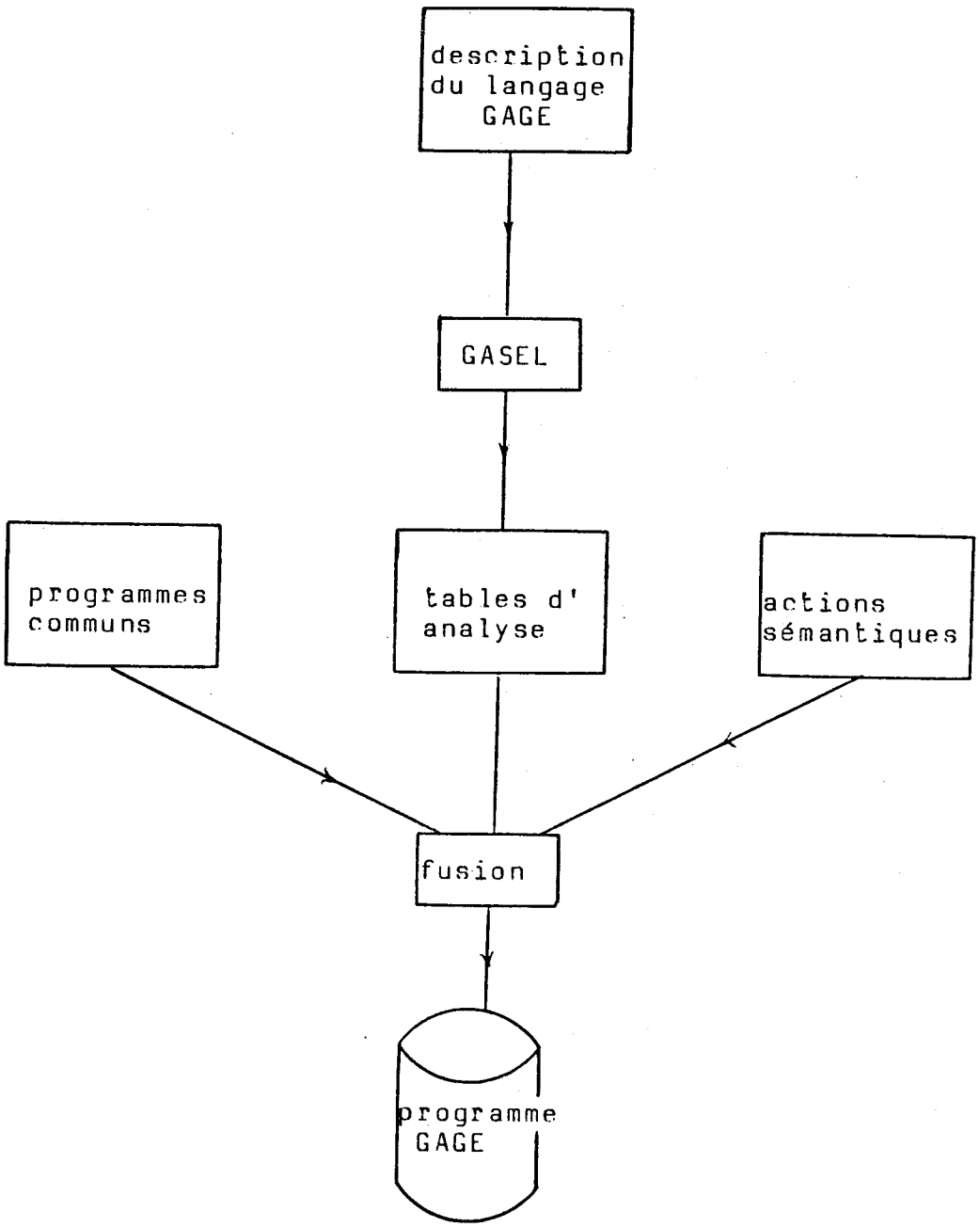


Figure 2.1. Réalisation de GAGE

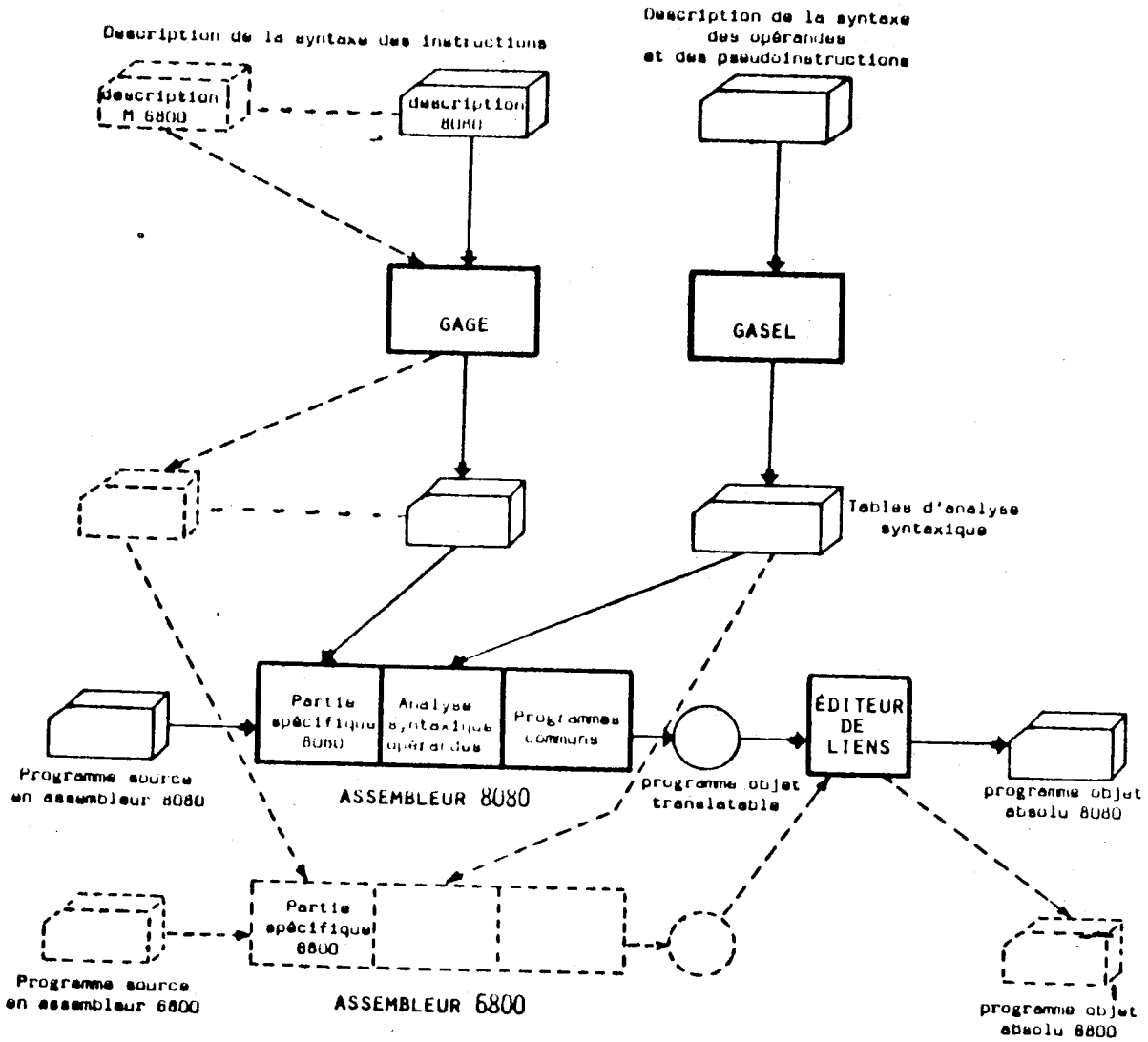


Figure 2.2. Génération et utilisation des assembleurs

4. LE GENERATEUR DE PROGRAMMES DE TEST

4.1. Algorithmes de test fonctionnel

Le test fonctionnel d'une instruction, comme nous l'avons vu en première partie, est généralement composé de trois phases :

- une phase d'initialisation des sources, c'est à dire la génération de l'information de test vers les sources du graphe,
- une phase d'exécution de l'instruction sous test,
- une phase d'observation des résultats, c'est à dire l'émission des résultats de l'instruction testée à partir du puits du graphe vers l'extérieur.

Exemple: l'instruction "LD A,B" du microprocesseur Z80 de ZILOG.

"LD A,B"

Le test de cette instruction se déroulera de la manière suivante:

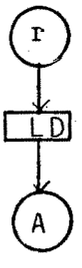
- initialisation des registres, en particulier le registre A avec une valeur différente du registre B,

- exécution de l'instruction "LD A,B",

- observation des registres, particulièrement le registre B.

D'autre part, le test d'une instruction implique son exécution sur des ensembles d'opérandes, pour une source donnée , ce qui conduit à une répétivité de cette séquence de test. De plus, chaque instruction peut être exécutée avec des sources différentes.

Exemple: test de l'instruction "LD A,r" du ZILOG Z80
L'itération se fait sur r $\in$ Reg et Reg=(A,B,C.D,E,H,L) l'ensembles des registres,
et sur t $\in$ T et T = (t1,t2,..,tn) l'ensemble des vecteurs de test.



"LD A,r"

Ainsi, le test de cette instruction se déroulera suivant l'algorithme ci-dessous:

Pour tout r $\in$ Reg

faire

pour tout t $\in$ T

faire

- INIT: initialiser tous les autres registres de Reg à 0,

initialiser r avec t,

- EXEC exécution de "LD A,r",
- OBSV: observation des résultats,
  mémoriser soit r soit Reg,

fin_faire

fin_faire

En observant l'ensemble des registres du microprocesseur à chaque exécution d'une instruction, ceci nous permet de vérifier l'identité du graphe d'exécution abstraite pour son test de conformité.

C'est donc sur cet aspect itératif qu'est basé le logiciel ROBIN.

4.2. Objectifs du logiciel ROBIN

Les objectifs du logiciel ROBIN est d'être un outil,qui à partir:

- d'une description symbolique d'un microprocesseur,
- d'une classification de ses instructions suivant l'ordonnancement défini en première partie,
- et d'une définition de ses algorithmes de test,

nous fournit le programme de test de ce microprocesseur.

La génération de ce logiciel de test passe par la définition d'un langage qu'on appellera le langage ROBIN.

4.2.1. Le langage ROBIN

Ce langage a été défini pour faciliter l'écriture des algorithmes de test. Il permet donc

- la spécification des ensembles sur lesquels porteront les itérations,
- L'écriture des instructions assembleurs paramétrées qu'on appellera : instructions d'itérations,
- L'écriture des instructions assembleurs,
- un appel automatique de séquences d'instructions prédéfinies, qu'on appellera : macro-instructions.

4.2.2 Le programme ROBIN

L'automate reconnaissant le langage ROBIN est capable d'analyser des instructions assembleurs, des macro-instructions et surtout les séquences particulières d'itérations.

Cet automate est décrit en GASEL (cf. annexe A2), et le logiciel GASEL nous fournit les tables d'analyse lexicale et syntaxique propres au langage ROBIN.
Les actions sémantiques (cf. annexe A6) assurent l'expansion séquentielle des instructions d'itérations.
Ainsi, les actions sémantiques, les tables d'analyse générées par GASEL, et les programmes d'interprétation de ces tables constituent le logiciel ROBIN.

Remarque: les actions sémantiques sont écrites en fortran

car elles opèrent sur des tables fournies par GASEL, ces tables étant générées en fortran.

La structure du logiciel ROBIN se décompose en :

- une partie principale où se trouve un analyseur lexicographique et syntaxique muni :
  - d'une table d'analyse lexicale (LEXI),
  - d'une table d'analyse syntaxique (AUTO)
  - d'une table de mots-clés réservés par le langage GASEL : (CLES),
  - d'une table des types internes des caractères (TYPCAR) (cf. Annexe A3).

- un ensemble de sous-programmes où se trouvent les actions sémantiques ainsi que tous les sous-programmes de gestion des tables d'identificateurs (cf. annexe A4), et des sous-programmes de gestion des tables propres à ROBIN (cf. Annexe A5).

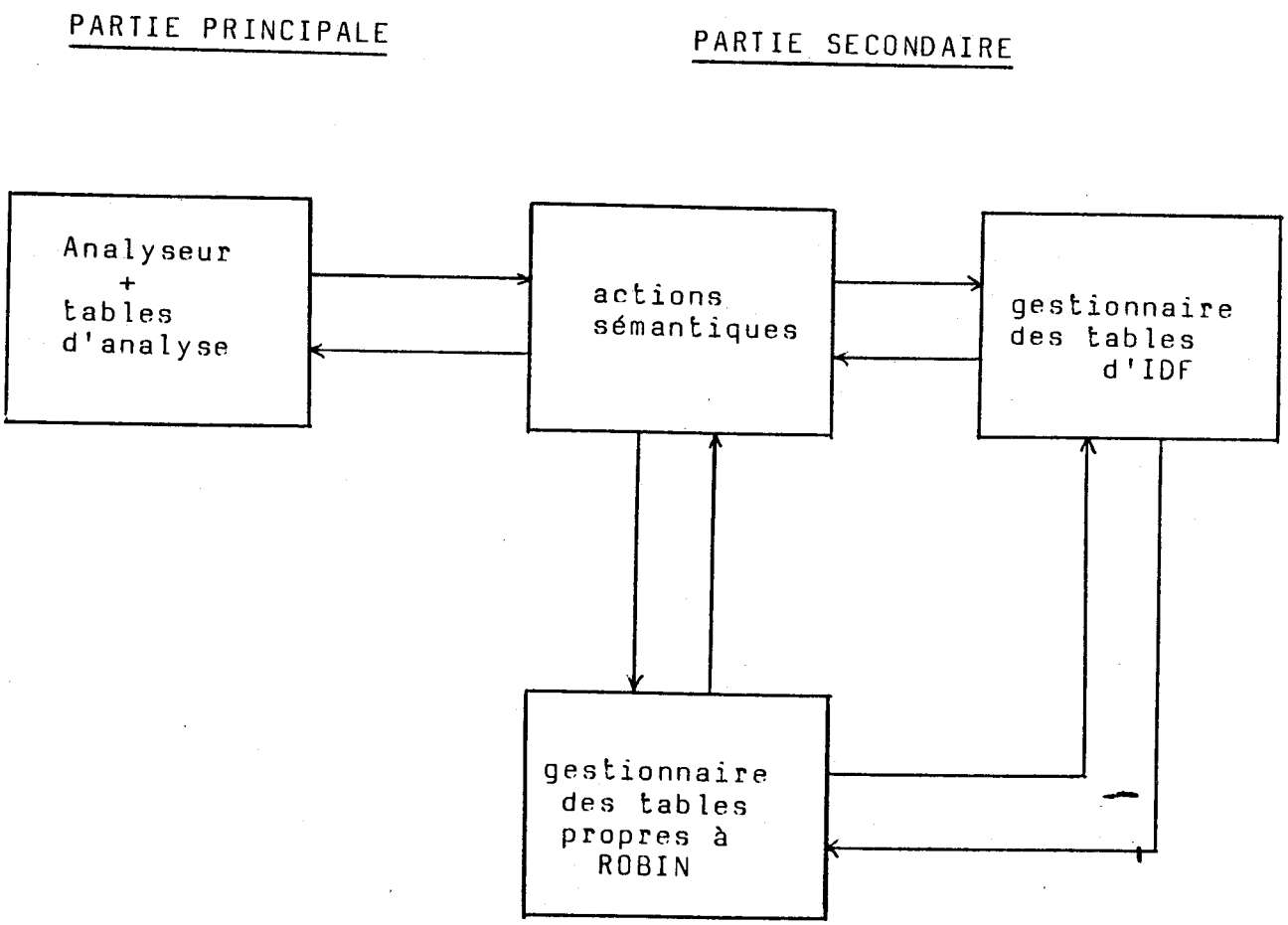


Figure 3. Configuration interne de ROBIN.

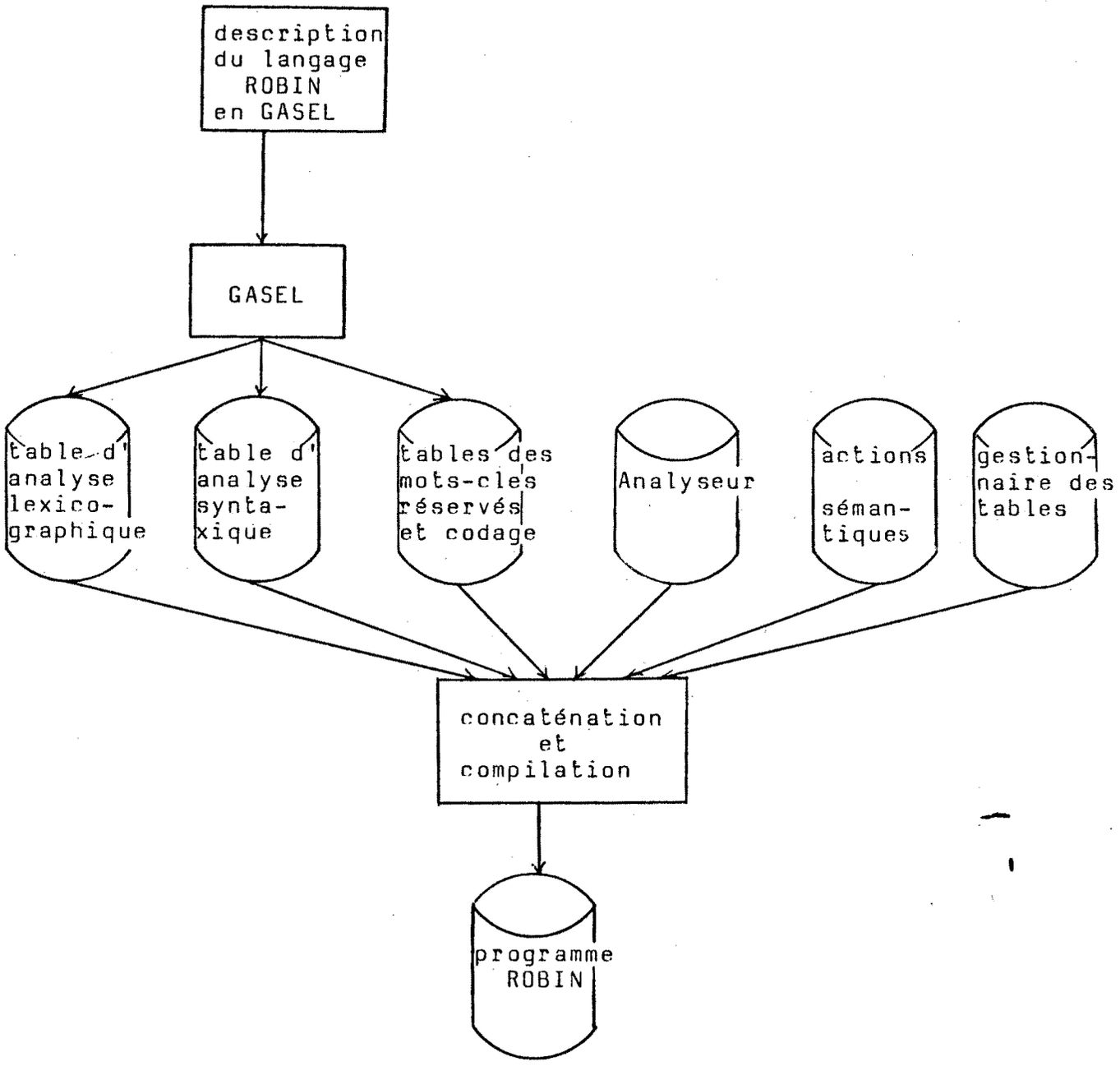


Figure 4. Réalisation fonctionnelle de ROBIN.

4.2.3. Utilisation du programme ROBIN

L'exécution du programme ROBIN avec comme donnée : l'algorithme de test pour un microprocesseur particulier (écrit en ROBIN), fournit le programme de test en langage d'assemblage pour ce microprocesseur (cf. figures 5 et 6).

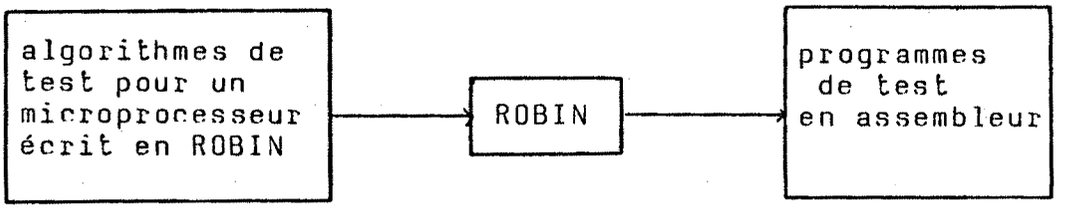


Figure 5. Utilisation du programme ROBIN.

Ainsi pour chaque microprocesseur, le programme de test en ROBIN sera à écrire.

Cette manière de procéder, c'est à dire la génération des programmes de test en langage d'assemblage et non en binaire directement exécutable, peut paraitre lourde, mais elle se justifie par les remarques suivantes :

- c'était la méthode la plus rapide pour l'obtention des premiers programmes de test qui nous permettent de définir plus précisement le testeur, et de mieux préparer le deuxième niveau de ROBIN,

- on a une totale liberté d'implantation des programmes de test, en outre, nous pouvons les assembler sur n'importe quel outil de développement possédant les assembleurs de ces microprocesseurs,
- le système GAGE permet de disposer de tous les assembleurs possibles, ce qui nous dispense dans un premier temps de générer du binaire exécutable.

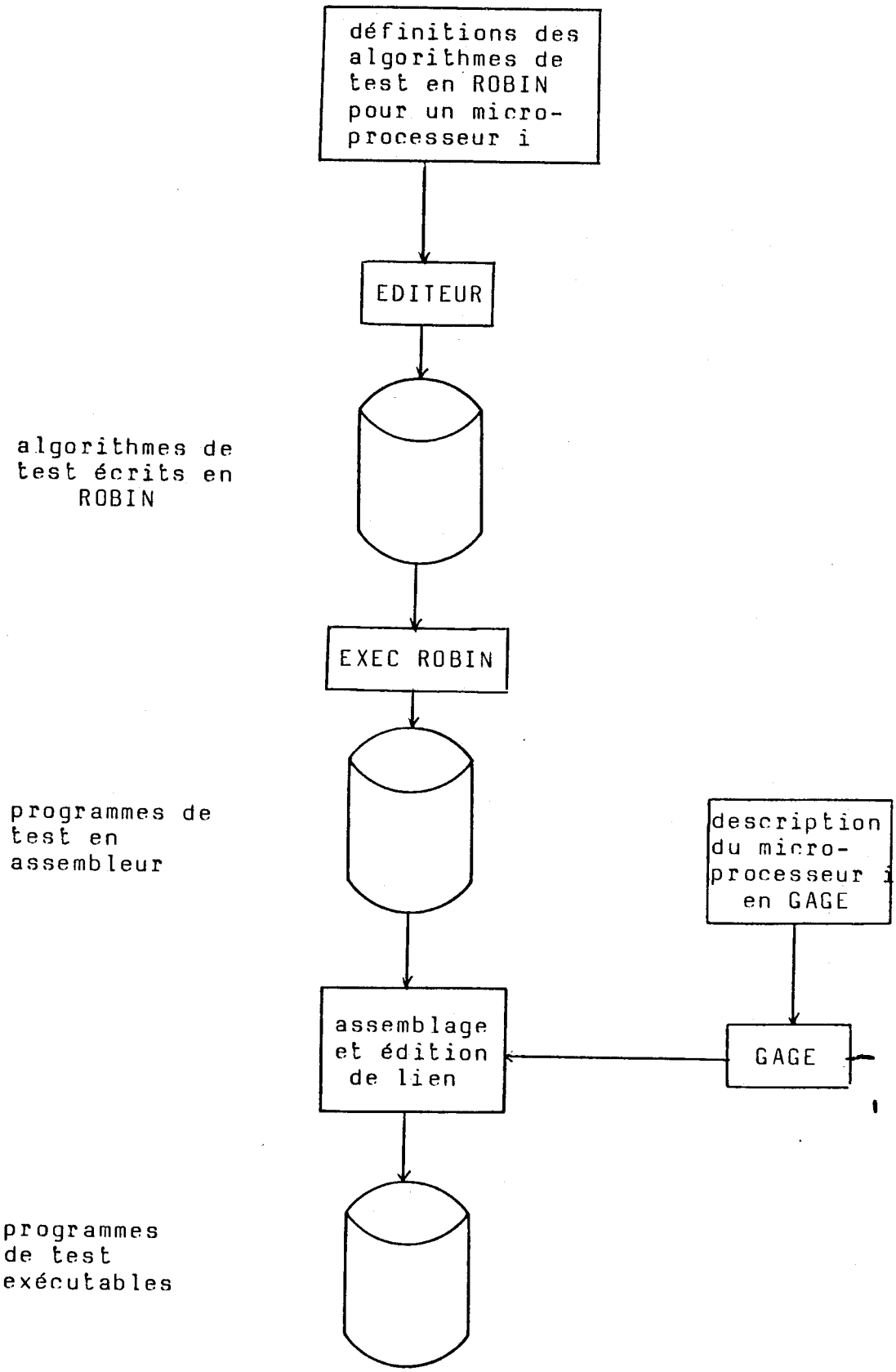


Figure 6. Réalisation des programmes de test pour un microprocesseur particulier

4.3. Description du langage ROBIN

Comme nous l'avons vu précédement, ce langage a été créé pour faciliter l'entrée des algorithmes de test.

Un programme écrit en ROBIN comporte :

- un titre,
- deux modules principaux : DECLARE et ALGORITHM,
- un module secondaire : PROCEDURE.

L'automate reconnaissant ce langage exige un titre en première ligne suivant la syntaxe:

titre "test . ..... "

et reconnait les modules définis ci-dessus

4.3.1. Le module DECLARE

Ce module contient toutes les déclarations d'ensembles sur lesquels porteront les itérations. Ces déclarations servent à définir :

- des ensembles de registres,
- des ensembles de vecteurs de test,
- des ensembles d'adresse mémoire,
- des ensembles de code-opérations qui auront le même algorithme de test.

Ainsi les éléments de ces ensembles seront soit des identificateurs, soit des valeurs hexadécimales,

décimales ou binaires. Toute autre reconnaissance déclanchera automatiquement une erreur.
Une déclaration aura la syntaxe suivante:

nom_d'ensemble : (élément1.élément2,.....élémentn)

Exemple : l'ensemble des registres du Z80 de ZILOG pourra s'écrire de la manière suivante

Reg : (A,B,C,D,E,H,L)

4 3.2. Le module PROCEDURE

Ce module a été défini pour mettre en évidence les sous-programmes d'initialisation et d'observation.

Par souci de simplicité pour l'automate du langage ROBIN c'est dans ce module que seront définies les macro-instructions si leur emploi est nécessaire pour la programmation des algorithmes de test.

Ces macro-instructions ont une procédure analogue à la macro-copy d'un macro-assembleur.

Une macro-instruction sera à définir suivant la syntaxe

```
FILE nom_de_macro
}corps de la macro
ENDF
```

et devra être appelée dans le module ALGORITHM suivant la syntaxe:

```
COPY nom_de_macro
```

Avantages :

Le programme ROBIN génère l'expansion des macro-instructions. Pour l'assemblage des programmes de test, un assembleur simple suffira.

Ce module est optionnel, il peut ne pas être utilisé si l'algorithme de test ne prévoit pas de macro-instructions.

4.4.3. Le module ALGORITHM

Ce module contient l'algorithme de test qui sera composé d'instructions assembleurs et surtout des séquences particulières d'itérations appelées aussi "boucle_iter" Une "boucle_iter" débute par le mot-clé "iter" et se termine par le mot-clé "f_iter".

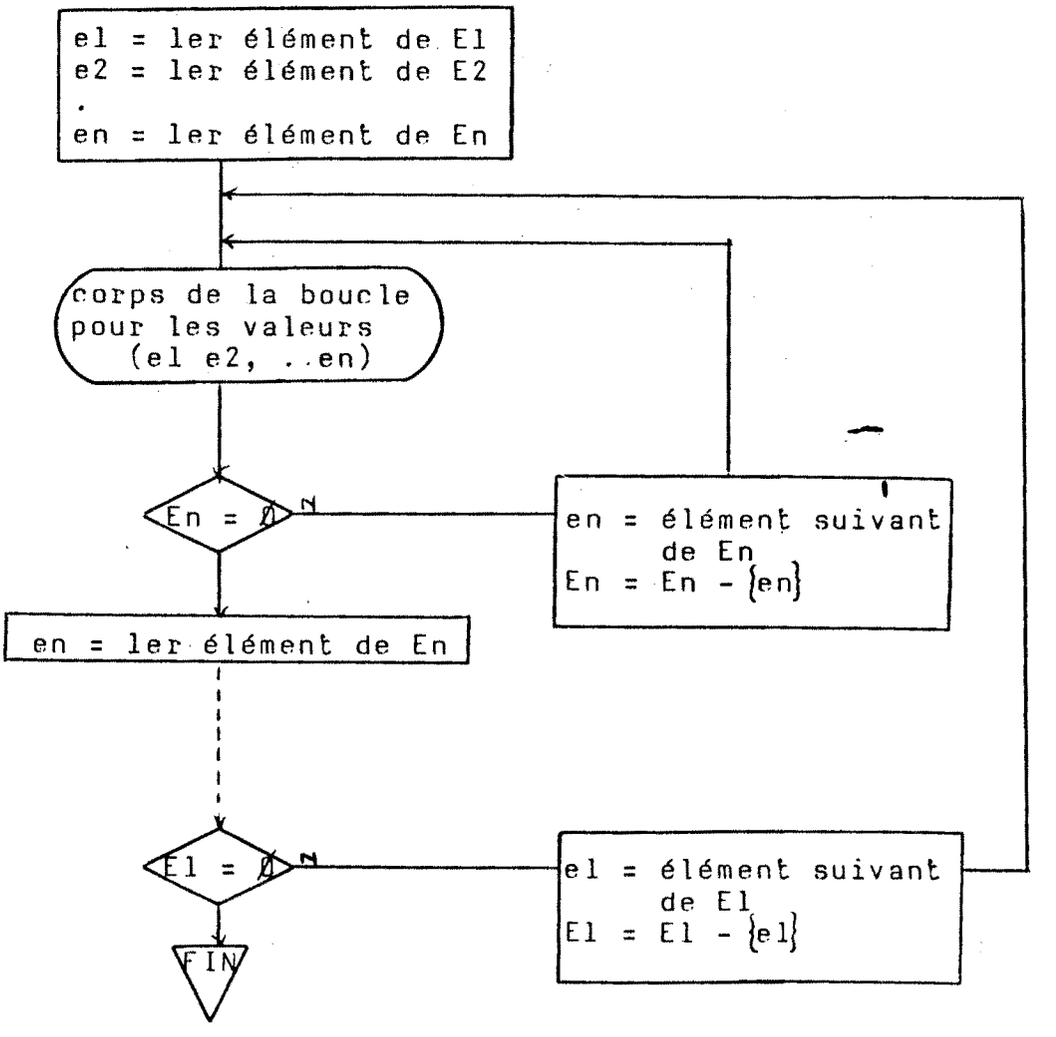
a) Forme générale de la "boucle_iter"

La forme générale d'une "boucle iter" est la suivante:

```
iter: E1 E2 E3 . . En
      {
        Corps de la boucle
      }
f_iter
```

où E1, E2.....En sont des ensembles ordonnés sur lesquels porteront l'itération.

L'organigramme équivalent à une "boucle iter" sera le suivant:



b) Ensembles E1, E2,..,En

Le nombre d'ensembles a été limité à 5 car ce nombre s'avérait suffisant pour l'écriture des algorithmes de test.
On a défini la syntaxe d'écriture de ces ensembles de la manière suivante

mot-clé(nom_d'ensemble ou suite d'éléments)

Les mots-clés réservés à cette syntaxe sont :

- "source" représentant la 1ère source du graphe,
- "si" représentant la ième source du graphe,
- "ti" représentant le ième vecteur de test ,
- "puits" représentant les puits du graphe,
- "codop" représentant le code-opération de l'instruction.

La syntaxe d'écriture de ces ensembles permet de préciser si le mot-clé correspondant à l'itération porte :

- sur un ensemble déja déclaré dans le module DECLARE. Dans ce cas, le nom de l'ensemble sera précisé dans la "boucle iter" correspondante :

mot-clé(E1)

- sur un ensemble non déclaré dans le module

DECLARE. Dans ce cas, les éléments seront précisés directement dans la "boucle_iter" :

mot-clé(e1,e2,. .,en)

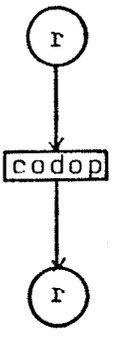
Cette possibilité offre un gain de place considérable en ne gardant en mémoire que les ensembles les plus souvent référencés (cf. annexe A5 a).

Exemple : soit les instructions "INC r" et "DEC r du Z80,

avec r ∈ Reg et Reg=(A B C.D.E,H.L)

et le vecteur de test correspondant t ∈ T

et T = (t1 t2,t3, . ,tn).



L'écriture en ROBIN des ensembles d'itération sera :

iter: codop(INC,DEC) source(Reg) t1(T)

corps de l itération

f_iter

Dans ce cas. le "codop" est l'itération la plus extérieure. jusqu'à "s2" qui est l'itération la plus intérieure. On a, ici, trois niveaux d'imbrication dans les ensembles d'itérations.

c) Corps de l'itération

Le corps de l'itération concerne directement le test fonctionnel d'une instruction et comporte donc trois phases :

- INIT : initialisation des source,
- EXEC : exécution de l'instruction,
- OBSV : observation des résultats.

Ainsi, le corps de l'instruction d'itération est composé d'instructions assembleurs. Certaines de ces instructions sont modifiées par une partie itérative.

Cette partie itérative ou variable est précisée par un mot-clé précédé du symbole "<" et suivi du symbole ">" :

<mot_clé>

La partie variable peut donc être soit un code-opération, soit une source soit un puits. soit un vecteur de test.

On donne un exemple complet d une "boucle iter" :

```
iter: codop(INC,DEC) source(Reg) t1(T)
      init: call initl;
            ld <source> <t1>;
      exec: <codop> <source>;
      obsv: copy observe;
f iter
```

Les premiers pas de cette itération nous créera la séquence suivante:

```
call initl
ld A,t1
INC A
..
...   instructions générées par la macro-copy
..
call initl
ld A,t2
INC A

jusqu'à

call initl
ld L,tn
INC L
..
...
```

Ainsi toutes les instructions comprises dans une

"boucle_iter" sont toutes générées à chaque pas d'itération, les parties variables prenant leur valeur courante à chaque itération.
Afin de faciliter la programmation du programme ROBIN, chaque instruction d'itération ou assembleur se trouvant dans une "boucle_iter" devra se terminer par un ";" sinon elle ne sera pas prise en compte.
En effet, il était nécessaire de définir une fin d'instruction assembleur parce que l'analyse de l'instruction ITER ne s'arrête pas sur une fin de ligne et continue son analyse en séquence.

Pour le cas particulier d'une "boucle_iter", toutes les instructions faisant partie de cette boucle doivent être sauvegardées en mémoire (cf. tableau TINSTR en annexe A5.c.).
Elles constituent la séquence de base pour la génération des itérations.
Par contre, toute instruction assembleur se trouvant à l'extérieur d'une "boucle_iter" devra être écrite tout à fait normalement, c'est à dire sans l'ajout d'un ";".

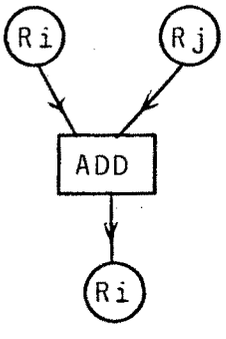
d) Autres possibilités d'écriture

d.1.) Notion de parallélisme

Ces imbrications ne pouvaient pas s'appliquer dans tous les cas. Certaines instructions à tester exigeaient un certain parallélisme au lieu d'une imbrication dans les ensembles d'itérations.

Exemple: l'instruction d'addition de deux registres.

- Soit Vi : le vecteur de test de l'addition pour Ri,
- Soit Vj : le vecteur de test de l'addition pour Rj,
- avec Vi = (vil vi2,...,vin) et
- avec Vj = (vjl,vj2,...,vjn).

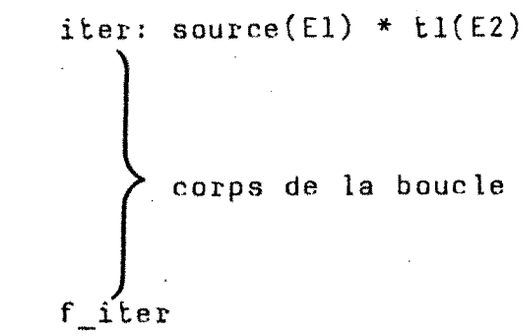


"ADD Ri,Rj"

Dans ce cas, il faut qu'à chaque vecteur de test vik s'appliquant à Ri ne corresponde qu'un seul vecteur de test vjk s'appliquant à Rj pour tester l'instruction "ADD Ri,Rj".

Pour résoudre ce problème, on a défini la syntaxe

suivante :



Ainsi, une étoile placée entre deux ensembles signifie que ces ensembles itèreront en parallèle. Dans l'exemple ci-dessus, le premier élément de E1 sera associé au premier élément de E2, et etc.., jusqu'au dernier élément de E1 qui sera associé au dernier élément de E2. Ces ensembles d'itérations doivent bien sûr posséder le même nombre d'éléments.

Tous les ensembles d'une "boucle_iter" peuvent itérer en parallèle

codop(E1) * source(E2) * s2(E3) * t1(E4) * puits(E5)

d 2.) Isolation d'instructions

Cette possibilité d'écriture a été rajoutée au langage ROBIN pour faire face à certains problèmes bien particuliers. Cette écriture nous permet d'isoler certaines instructions.

```
iter: source(E1)
    init: ld A,<source>;
        iter: codop (E2) t1(E3)
            exec: <codop> B <t1>;
            obsv: copy observe;
f_iter
```

Dans cet exemple, nous voyons que l'instruction "ld a <source>" ne sera générée qu'à chaque changement de "source" et non à chaque changement de "codop" et de "t1".

4.3.4. Particularités

Toute chaine de caractères qui commence par le symbole "&" et qui se termine par ce symbole "&" est considéré comme commentaire par l'analyseur syntaxique.

D'autre part, tout programme écrit en ROBIN devra se terminer par le mot-clé : FIN,sinon une fin anormale sera détectée.

Toute erreur détectée à l'exécution du programme ROBIN peut aisément se corriger grace aux listes d'erreurs jointes en annexes A7.

4 3.5. Exemple d'un programme complet écrit en ROBIN

```
titre "test du 6800"

declare:
&
          declarations d'ensembles de registres
&
regab:(a,b)
regxs:(x,s)
&
          valeurs immediates
&
valim1:(1h,2h,4h,8h,10h,20h,40h,80h)
valim2:(1,2,4,8,10h,20h,40h,80h,100h,200h,400h,800h,1000h,2000h,4000h,8000h)
procedure:
&
&
          file  obs1
          staa adrsto
          stab adrsto-1
          tpa
          staa adrsto-2
          stx adrsto-3
          sts adrsto-5
adrsto set adrsto-7
          endf
algorithm:
&
*****************************************************************************
      classe1     s-classe1
*****************************************************************************
&
       section init
adrsto set 0fffh
          lds #2h
init1     clra
          clrb
          ldx #0
          tap
          rts
;
;         ldaa #n      ldab #n
;
iter: puits(regab) source(valim1)
   init:          jsr init1;
   exec:          lda(puits)  #(source);
   obsv:          copy obs1;
fiter
;
;     ldx #m   lds #m
;
iter:  puits(regxs) source(valim2)
   init:          jsr init1;
   exec:          ld(puits) #(source);
   obsv:          copy obs1;
                  lds #2;
fiter
fin
```

4.4. Lexicographie et cartes syntaxiques du langage ROBIN

Le langage ROBIN a été décrit dans la grammaire de Backus en annexe A1.

a) Lexicographie

Si nous ne déclarons aucune lexicographie celle de GASEL est prise par défaut. Mais pour le langage ROBIN il manquait seulement la reconnaissance d'une valeur hexadécimale de la forme 0ffh, Gasel ne reconnaissant comme valeur hexadécimale que les valeurs de la forme #ff.

C'est pourquoi il a fallu redéclarer une lexicographie pour ROBIN et écrire le traitement sémantique correspondant à cette nouvelle unité lexicale.

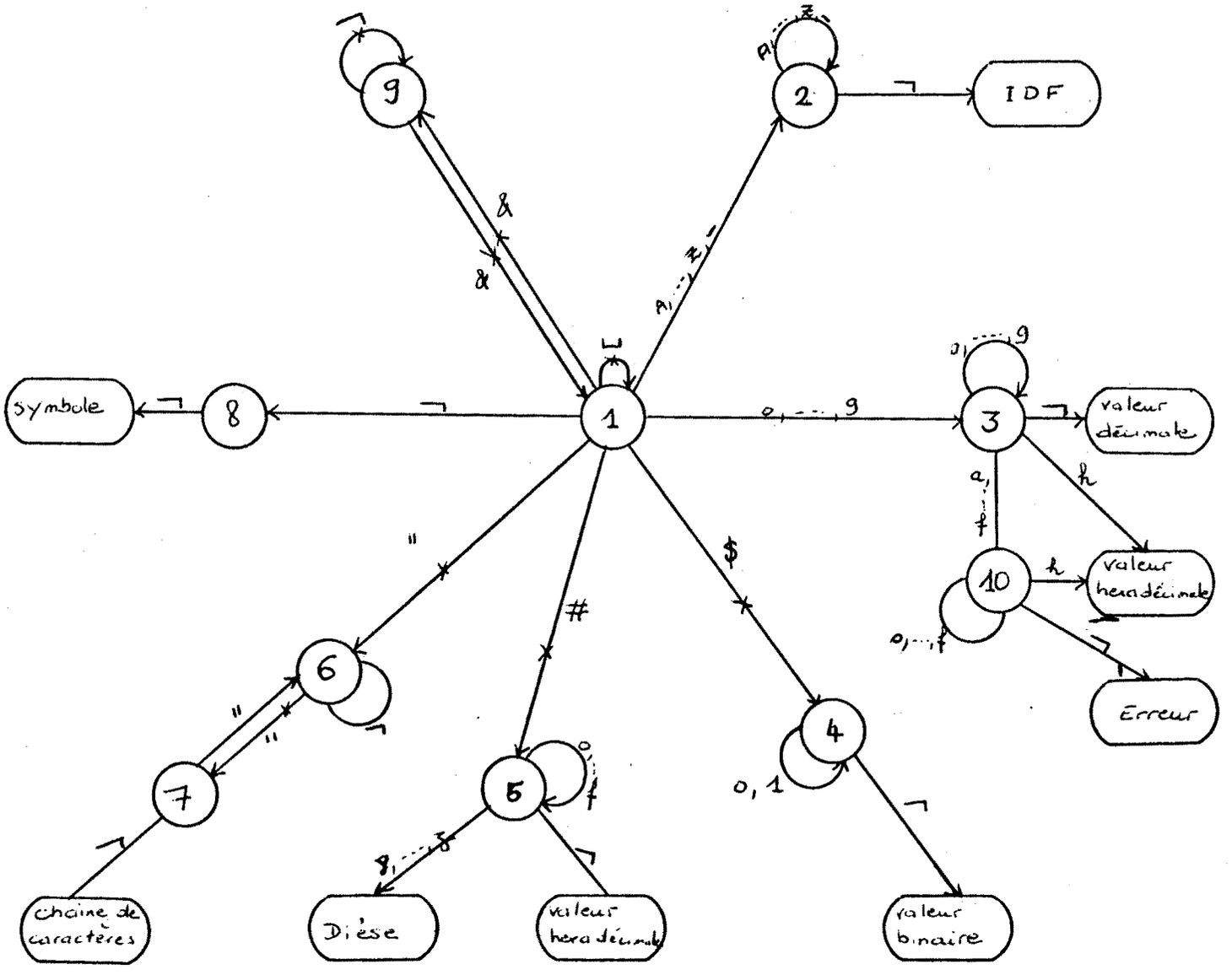
Les unités lexicales reconnues par ROBIN sont :

- valeur décimale (127),
- valeur binaire ($001),
- valeur hexadécimale (#ff, 0ffh),
- chaine de caractères ("chaine"),
- symboles spéciaux qui sont ( ) < > ? + * = % ' " , . / ; :
- identificateur (toto,t345).

Schéma de l'automate lexical

⟶ transition avec conservation du caractère lu.

⟶× transition sans conservation du caractère lu.

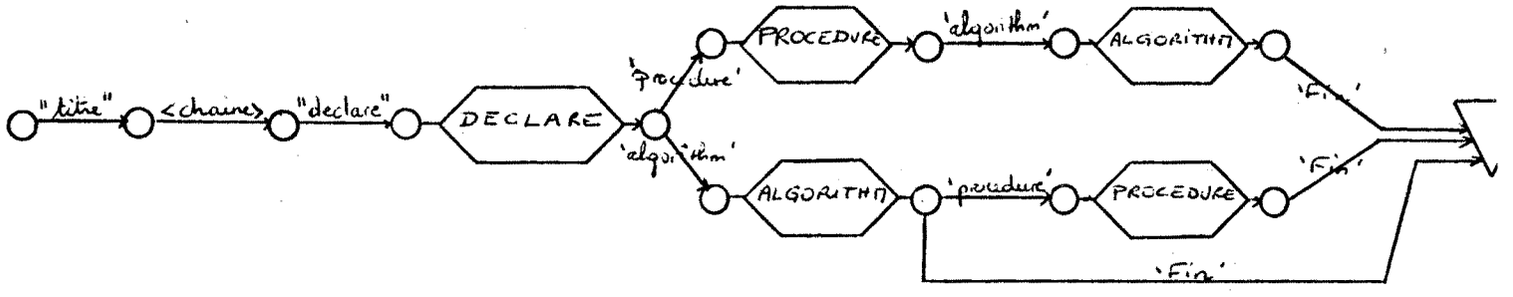


b) Cartes syntaxiques du langage ROBIN

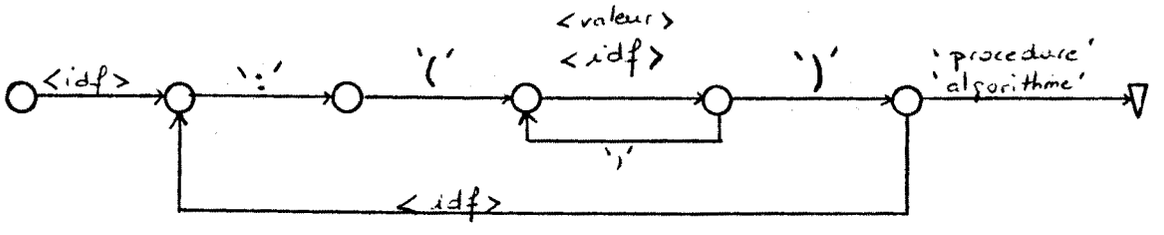
Les flèches en trait plein correspondent aux transitions possibles après chaque état (cercles). Les flèches en pointillés indiquent une réanalyse sur l'unité courante lue.

Le symbole ¬ représente tout autre mot-clé ou toute autre unité syntaxique rencontrée.

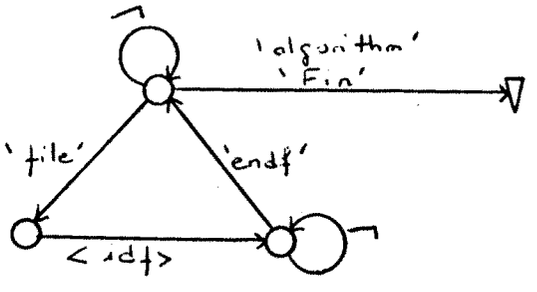
PROGRAMME



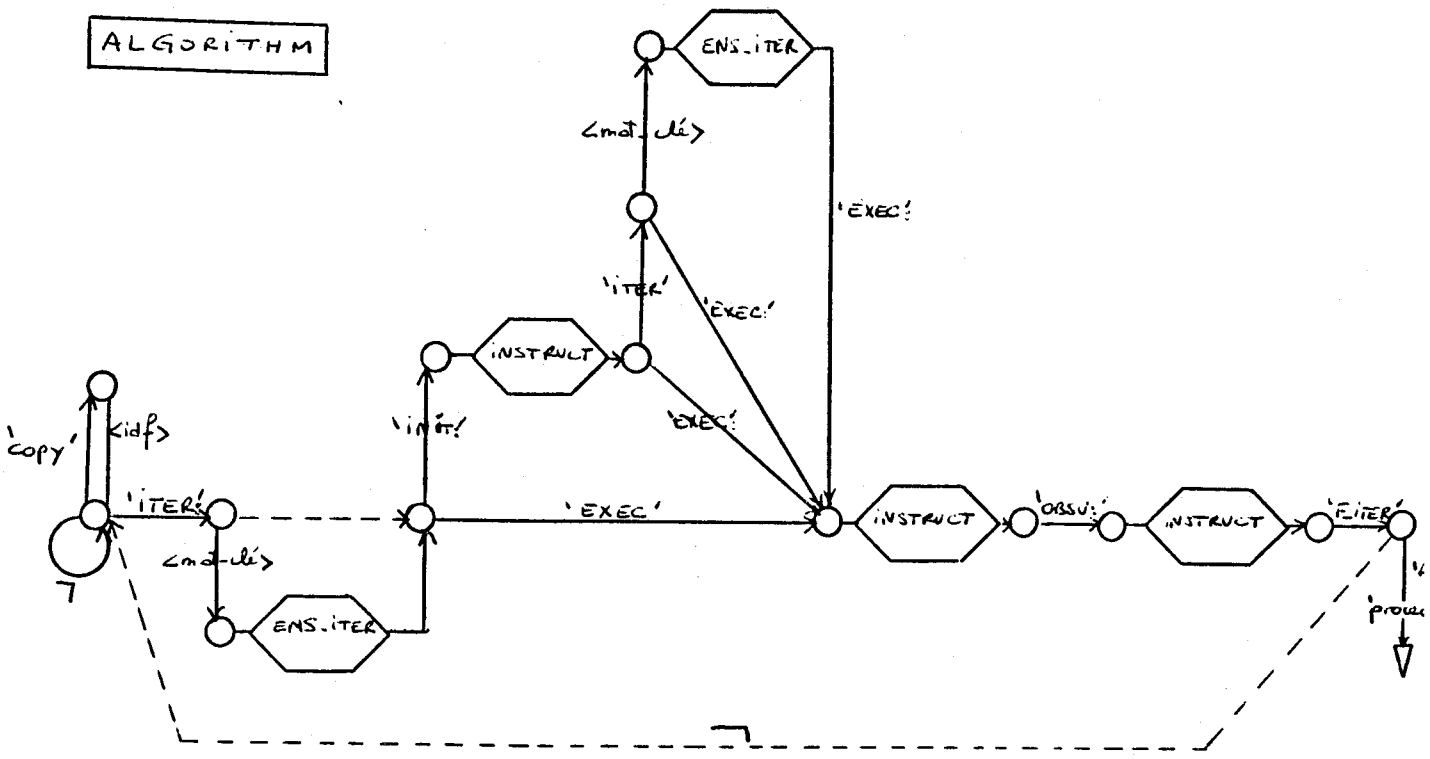
DECLARE



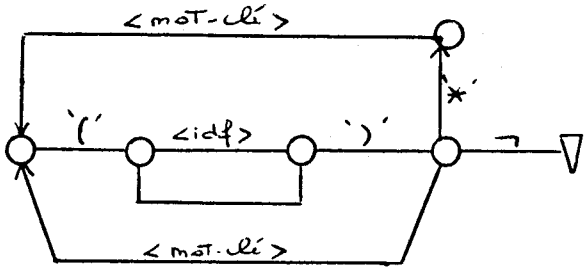
PROCEDURE



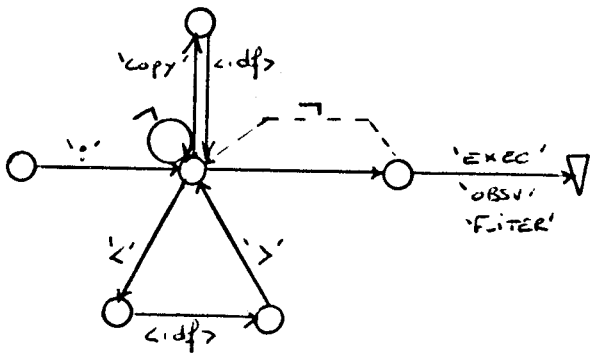
ALGORITHM



ENS - ITER



INSTRUCT



CHAPITRE III. EXEMPLE D'APPLICATION DU SYSTEME ROBIN

Un exemple d'application du système ROBIN est montré pour le microprocesseur 6800 de Motorola.

La production du programme de test se déroule en deux étapes :

- une étape de préparation,
- l'écriture du programme de test en ROBIN et son exécution.

1. Etape de préparation.

Cette étape doit aboutir au classement des instructions suivant la méthode proposée en première partie.

1.1. Architecture du microprocesseur

Une première étude est celle de l'architecture du microprocesseur d'après son schéma fonctionnel qui nous renseigne sur son organisation interne et externe.

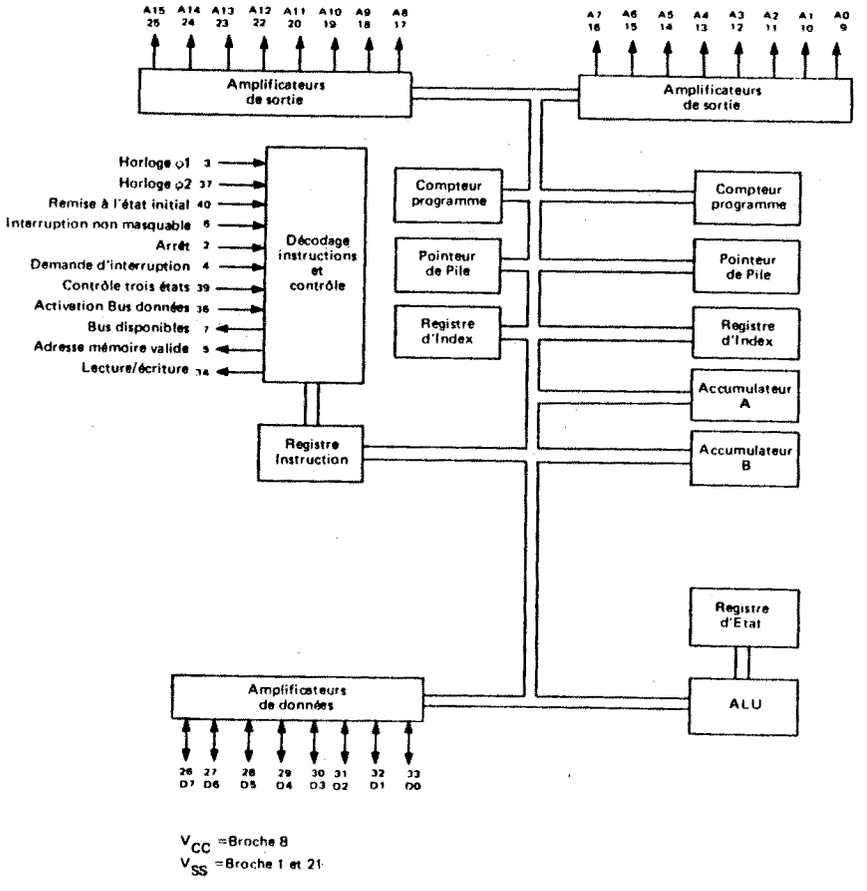


Figure 7. Schéma fonctionnel.

Les caractéristiques principales du microprocesseur 6800 sont :

- une longueur du mot de 8 bits,
- un bus de données 8 bits bidirectionnel,
- un bus d'adresse de 16 bits capable d'adresser 64K,
- des lignes de commandes,
- 6 registres internes représentés sur la figure 7,
- 7 modes d'adressages : direct, relatif, immédiat, indexé, étendu, implicite, et accumulateur.

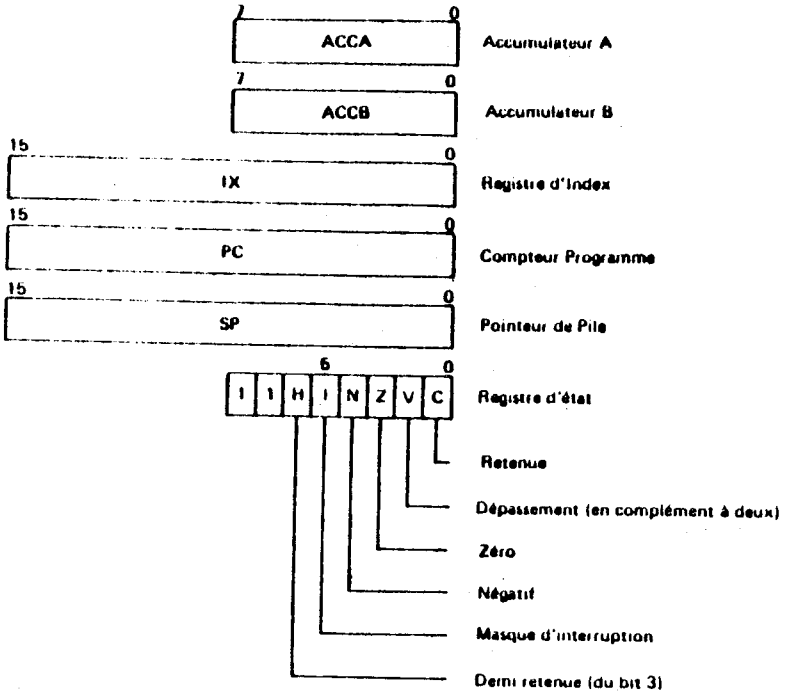


Figure 8. Registres programmables.

1.2. Jeu d'instruction

Le microprocesseur 6800 a un jeu de 72 types d'instructions, mais compte tenu de ses 7 modes d'adressage, il possède 197 instructions. L'étude du jeu d'instruction nous permet de déterminer les mesures d'accessibité de la mémoire et de ses registres :

- mémoire et extérieur : t=t'=1,
- registres A,B,SP,IX : t=t'=2,
- le registre d'état : t=t'=3 car le registre d'état est chargé et observé à travers l'accumulateur A.

Le jeu d'instruction est le suivant :

TABLE 3 – ACCUMULATOR AND MEMORY INSTRUCTIONS

OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED =	DIRECT OP	DIRECT ~	DIRECT =	INDEX OP	INDEX ~	INDEX =	EXTND OP	EXTND ~	EXTND =	IMPLIED OP	IMPLIED ~	IMPLIED =	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	5 H	4 I	3 N	2 Z	1 V	0 C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	•	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	•	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	•	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	•	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	•	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A · M → A	•	•	↕	↕	R	•
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B · M → B	•	•	↕	↕	R	•
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A · M	•	•	↕	↕	R	•
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B · M	•	•	↕	↕	R	•
Clear	CLR							6F	7	2	7F	6	3				00 → M	•	•	R	S	R	R
	CLRA													4F	2	1	00 → A	•	•	R	S	R	R
	CLRB													5F	2	1	00 → B	•	•	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A - M	•	•	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B - M	•	•	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A - B	•	•	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				M̄ → M	•	•	↕	↕	R	S
	COMA													43	2	1	Ā → A	•	•	↕	↕	R	S
	COMB													53	2	1	B̄ → B	•	•	↕	↕	R	S
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 - M → M	•	•	↕	↕	①	②
	NEGA													40	2	1	00 - A → A	•	•	↕	↕	①	②
	NEGB													50	2	1	00 - B → B	•	•	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add of BCD Characters into BCD Format	•	•	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M - 1 → M	•	•	↕	↕	4	•
	DECA													4A	2	1	A - 1 → A	•	•	↕	↕	4	•
	DECB													5A	2	1	B - 1 → B	•	•	↕	↕	4	•
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A⊕M → A	•	•	↕	↕	R	•
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B⊕M → B	•	•	↕	↕	R	•
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	•	•	↕	↕	⑤	•
	INCA													4C	2	1	A + 1 → A	•	•	↕	↕	⑤	•
	INCB													5C	2	1	B + 1 → B	•	•	↕	↕	⑤	•
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	•	•	↕	↕	R	•
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	•	•	↕	↕	R	•
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	•	•	↕	↕	R	•
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	•	•	↕	↕	R	•
Push Data	PSHA													36	4	1	A → M_{SP}, SP - 1 → SP	•	•	•	•	•	•
	PSHB													37	4	1	B → M_{SP}, SP - 1 → SP	•	•	•	•	•	•
Pull Data	PULA													32	4	1	SP + 1 → SP, M_{SP} → A	•	•	•	•	•	•
	PULB													33	4	1	SP + 1 → SP, M_{SP} → B	•	•	•	•	•	•
Rotate Left	ROL							69	7	2	79	6	3				M	•	•	↕	↕	⑤	↕
	ROLA													49	2	1	A — C b7 b0	•	•	↕	↕	⑤	↕
	ROLB													59	2	1	B — C b7 b0	•	•	↕	↕	⑤	↕
Rotate Right	ROR							66	7	2	76	6	3				M	•	•	↕	↕	⑤	↕
	RORA													46	2	1	A — C b7 b0	•	•	↕	↕	⑤	↕
	RORB													56	2	1	B — C b7 b0	•	•	↕	↕	⑤	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M	•	•	↕	↕	⑤	↕
	ASLA													48	2	1	A — 0 C b7 b0	•	•	↕	↕	⑤	↕
	ASLB													58	2	1	B — 0 C b7 b0	•	•	↕	↕	⑤	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M	•	•	↕	↕	⑤	↕
	ASRA													47	2	1	A — b7 b0 C	•	•	↕	↕	⑤	↕
	ASRB													57	2	1	B — b7 b0 C	•	•	↕	↕	⑤	↕
Shift Right, Logic	LSR							64	7	2	74	6	3				M	•	•	R	↕	⑤	↕
	LSRA													44	2	1	A — 0 b7 b0 C	•	•	R	↕	⑤	↕
	LSRB													54	2	1	B — 0 b7 b0 C	•	•	R	↕	⑤	↕
Store Acmltr.	STAA				97	4	2	A7	6	2	B7	5	3				A → M	•	•	↕	↕	R	•
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	•	•	↕	↕	R	•
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A - M → A	•	•	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B - M → B	•	•	↕	↕	↕	↕
Subtract Acmltrs.	SBA													10	2	1	A - B → A	•	•	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A - M - C → A	•	•	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B - M - C → B	•	•	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	•	•	↕	↕	R	•
	TBA													17	2	1	B → A	•	•	↕	↕	R	•
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M - 00	•	•	↕	↕	R	R
	TSTA													4D	2	1	A - 00	•	•	↕	↕	R	R
	TSTB													5D	2	1	B - 00	•	•	↕	↕	R	R
																		H	I	N	Z	V	C

LEGEND:

- OP Operation Code (Hexadecimal);
- ~ Number of MPU Cycles;
- = Number of Program Bytes;
- + Arithmetic Plus;
- − Arithmetic Minus;
- · Boolean AND;
- M_{SP} Contents of memory location pointed to be Stack Pointer
- + Boolean Inclusive OR;
- ⊕ Boolean Exclusive OR;
- M̄ Complement of M;
- → Transfer Into;
- 0 Bit = Zero;
- 00 Byte = Zero.

Note – Accumulator addressing mode instructions are included in the column for IMPLIED addressing

CONDITION CODE SYMBOLS:

- H Half-carry from bit 3;
- I Interrupt mask
- N Negative (sign bit)
- Z Zero (byte)
- V Overflow, 2's complement
- C Carry from bit 7
- R Reset Always
- S Set Always
- ↕ Test and set if true, cleared otherwise
- • Not Affected

TABLE 4 – INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

POINTER OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	IMPLIED OP	IMPLIED ~	IMPLIED #	BOOLEAN/ARITHMETIC OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$X_H - M, X_L - (M+1)$	•	•	⑦	↕	⑦	•
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	•	•	•	↕	•	•
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	•	•	•	↕	•	•
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M+1) \rightarrow X_L$	•	•	⑨	↕	R	•
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M+1) \rightarrow SP_L$	•	•	⑨	↕	R	•
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M+1)$	•	•	⑨	↕	R	•
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M+1)$	•	•	⑨	↕	R	•
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•	•

TABLE 5 – JUMP AND BRANCH INSTRUCTIONS

OPERATIONS	MNEMONIC	RELATIVE OP	RELATIVE ~	RELATIVE #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	IMPLIED OP	IMPLIED ~	IMPLIED #	BRANCH TEST	5 H	4 I	3 N	2 Z	1 V	0 C
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•
Branch If ≥ Zero	BGE	2C	4	2										N ⊕ V = 0	•	•	•	•	•	•
Branch If > Zero	BGT	2E	4	2										Z + (N ⊕ V) = 0	•	•	•	•	•	•
Branch If Higher	BHI	22	4	2										C + Z = 0	•	•	•	•	•	•
Branch If ≤ Zero	BLE	2F	4	2										Z + (N ⊕ V) = 1	•	•	•	•	•	•
Branch If Lower Or Same	BLS	23	4	2										C + Z = 1	•	•	•	•	•	•
Branch If < Zero	BLT	2D	4	2										N ⊕ V = 1	•	•	•	•	•	•
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	•	•	•	•	•	•
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•
No Operation	NOP										02	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI										3B	10	1		⑩					
Return From Subroutine	RTS										39	5	1		•	•	•	•	•	•
Software Interrupt	SWI										3F	12	1	See Special Operations	•	•	•	•	•	•
Wait for Interrupt	WAI										3E	9	1		•	⑪	•	•	•	•

TABLE 6 – CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

OPERATIONS	MNEMONIC	IMPLIED OP	IMPLIED ~	IMPLIED #	BOOLEAN OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Clear Carry	CLC	0C	2	1	$0 \rightarrow C$	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	$0 \rightarrow I$	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	$0 \rightarrow V$	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	$1 \rightarrow C$	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	$1 \rightarrow I$	•	S	•	•	•	•
Set Overflow	SEV	0B	2	1	$1 \rightarrow V$	•	•	•	•	S	•
Acmltr A → CCR	TAP	06	2	1	$A \rightarrow CCR$	⑫					
CCR → Acmltr A	TPA	07	2	1	$CCR \rightarrow A$	•	•	•	•	•	•

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

1 (Bit V) Test: Result = 10000000?
2 (Bit C) Test: Result = 00000000?
3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
4 (Bit V) Test: Operand = 10000000 prior to execution?
5 (Bit V) Test: Operand = 01111111 prior to execution?
6 (Bit V) Test: Set equal to result of N⊕C after shift has occurred.
7 (Bit N) Test: Sign bit of most significant (MS) byte = 1?
8 (Bit V) Test: 2's complement overflow from subtraction of MS bytes?
9 (Bit N) Test: Result less than zero? (Bit 15 = 1)
10 (All) Load Condition Code Register from Stack. (See Special Operations)
11 (Bit I) Set when interrupt occurs. If previously set, a Non Maskable Interrupt is required to exit the wait state.
12 (All) Set according to the contents of Accumulator A.

1.3. Classement des instructions

a) Construction des graphes d'exécution abstraite

Pour chaque instruction, on définit tous les graphes d'exécution abstraite associés à ses divers modes d'adressage.

Toutes les instructions, dont le graphe d'exécution abstraite est identique, sont regroupées sous ce même graphe.

Ainsi, en procédant de cette manière pour toutes les instructions du microprocesseur 6800, on obtient 17 types de graphes. chacun de ces graphes représentant une seule instruction ou un groupe d'instructions.

Ces graphes sont donnés en Annexe 8.

b) Ordonnancement des instructions

D'après la méthode d'ordonnancement déjà définie en première partie on ordonne donc l'ensemble des graphes suivant les relations de dominance structurelle et fonctionnelle et les paramètres d'affinement définis!

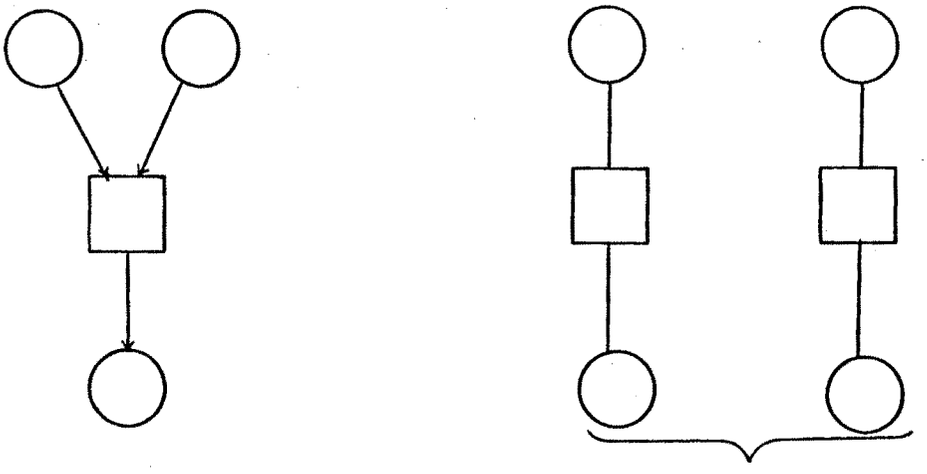
- Partition structurelle.

La relation de dominance structurelle nous donne une partition structurelle de ces graphes en classes. Ainsi pour le microprocesseur 6800 on obtient 7 classes.

La relation de dominance structurelle n'étant pas une relation d'ordre total, on trouve dans une même classe

des graphes dits "non comparables".

Exemple de graphes non comparables :



Ensuite, les classes contenant plusieurs graphes sont ordonnées suivant un affinement structurel en sous-classes.

Les critères d'affinement sont :

- le degré de multiplicité,
- la profondeur des graphes.

- Partition fonctionnelle

Les instructions appartenant à une même sous-classe, sont partitionnées en blocs suivant leur mésure d'accessibilité (t,t') d'après la relation de dominance fonctionnelle.

Les critères d'affinement sont :

- format maximum des éléments de mémorisation,
- type de micro-opération.

Ainsi, en fin de classement, l'ensemble des instructions

du microprocesseur 6800 est ordonné en 7 classes 17 sous-classes, 27 blocs et 40 sous-blocs (cf Annexe 8).

En conclusion à l'étape de préparation, on remarque que :

- d'après le classement structurel, on teste d'abord les instructions qui activent un minimum de matériel (classe 1), puis une partie de matériel rajoutée etc. jusqu'au maximum de matériel mis en jeu (dernière classe),
- d'après le classement fonctionnel, à l'intérieur d'une sous-classe, on teste d'abord les instructions dont les opérandes sont les plus accessibles, etc., jusqu'aux moins accessibles.

2. Utilisation du logiciel ROBIN

D'après le classement des instructions ainsi obtenu, il reste à écrire le programme de test du microprocesseur 6800 et le soumettre en entrée au logiciel ROBIN qui fournira son programme de test en assembleur.

2.1. Ecriture du programme de test en ROBIN

Le programme de test en ROBIN passe par :

- la définition des procédures d'initialisation et d'observation qui seront utilisées pour le test de chaque instruction,
- l'écriture de l'algorithme de test pour chaque instruction.

a) Procédures standards d'initialisation et d'observation : le module PROCEDURE

La procédure d'initialisation standard sera réalisée par un sous-programme de chargement à 0 de tous les registres

Exemple :

```
initl clra
      clrb
      ldx #0
      tap
      rts
```

La procédure d'observation standard sera réalisée par une macro instruction qui sauvegardera l'état du microprocesseur (c'est à dire tous les registres) dans une pile.

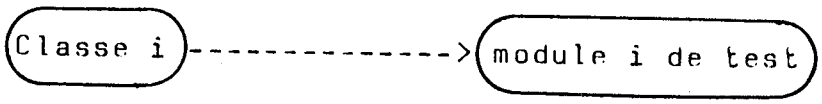
Exemple :

```
        file obsl
        staa adrsto
        stab adrsto-1
        tap
        staa adrsto-2
        stx adrsto-3
        sts adrsto-5
adrsto  set adrsto-7
        endf
```

Ainsi, on décide que toutes les instructions nécessaires à l'initialisation et à l'observation seront testées dans une classe 0 appelée aussi "hardcore".
Les instructions, faisant partie du hardcore, sont donc : JSR m, RTS, CLRA, CLRB, LDX #0, TAP, STAA m, STAB m, STX m, STS m.

b) Ecriture des instructions d'itération : le module ALGORITHM

On écrit le programme de test classe par classe, chaque classe constituant un fichier de données pour le programme ROBIN.



Pour chaque classe, on écrit les instructions d'itérations correspondant soit à une instruction soit à un groupe d'instructions à tester dans l'ordre des sous-classes blocs et sous-blocs.

D'autre part, toutes les instructions appartenant à un sous-bloc ne font pas obligatoirement toutes partie de la même instruction d'itération. En effet, ces instructions peuvent s'appliquer à des vecteurs de test différents ou, simplement, n'ont pas la même syntaxe d'écriture en assembleur.

Exemples :

premier exemple : test des instructions LDAA #n et LDAB #n appartenant à la classe 1.

```
iter: puits(regab) source(valim1)
      init: jsr init1;
      exec: lda<puits> #<source>;
      obsv: copy obs1;
f_iter
```

où regab est défini comme l'ensemble des registres (a,b)

et valim1 comme l'union des vecteurs de test

- des éléments de mémorisation,
- de la micro-opération de transfert
- et de l'identité du graphe.

En annexe 8, on trouve le listing complet correspondant au test de la classe 1.

deuxième exemple : Pour les instructions LDAA n, LDAB n, LDAA m, LDAB m appartenant à la classe 2, l'instruction d'itération est :

```
iter: puits(regab) source(adr1,adr2)
      init: ldaa #Offh;
            staa <source>;
            jsr init1;
      exec: lda<puits> <source>;
      obsv: copy obs1;
f_iter
```

où adr1 est une adresse sur 8 bits,
et adr2 une adresse sur 16 bits.

Les éléments de mémorisation et la micro-opération de transfert ayant déjà été testés dans l'instruction d'itération précédente on ne vérifie ici que l'identité du graphe.

Ici, la valeur Offh a été choisie comme opérande de test pour cette instruction d'itération. Cette valeur est stockée en mémoire à l'aide d'instructions appartenant au hardcore ou préalablement testées, après quoi tous les registres sont initialisés à zéro.

Après l'exécution de l'instruction testée, seul le registre "puits" doit contenir la valeur Offh et tous les autres registres doivent rester à zéro. Dans tout autre cas, une erreur sera détectée .

troisième exemple : instruction de saut JMP n,X où PC <-- PC + déplacement n + contenu du registre X.

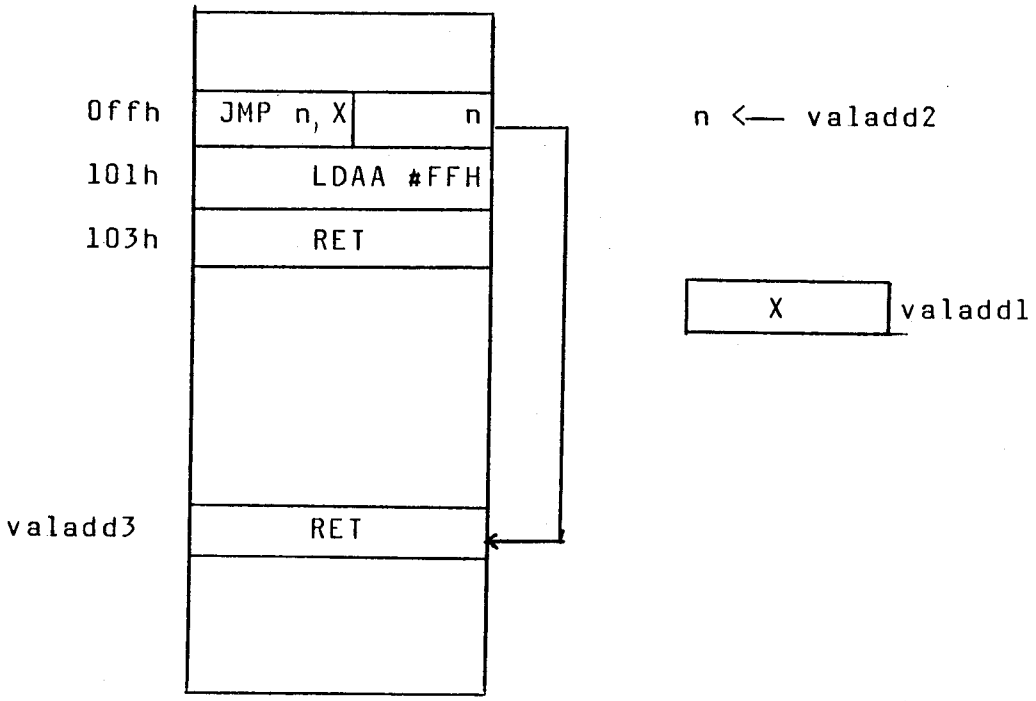
Pour tester cette instruction nous devons appliquer les vecteurs de test de l'addition:

- ensemble VALADD1 s'appliquant à X,
- ensemble VALADD2 s'appliquant à n,
- ensemble VALADD3 étant VALADD1 + VALADD2

On écrira les instructions suivantes :

```
        ldaa #6eh          (codop de "jmp n x" chargé
        staa 0ffh           à l'adresse 0ffh)
        ldx #86ffh         (codop de "ldaa #ff" à
        stx 101h            l'adresse 101h)
        ldx #3900h         (codop de "ret" à
        stx 103h            l'adresse 103h)

   iter: source(valadd1) * s2(valadd2) * s3(valadd3)
     init: ldaa #<s2>;     (déplacement n à
           staa 100h;       l'adresse 100h)
           jsr init1;      (mise à 0 des registres)
           ldx #3900h;     (codop de "RET" à
           stx <s3>;        l'adresse n+(x))
           stx #<source>;
     exec: jsr 0ffh;
     obsv: copy obs1;
   f_iter
```



On teste ainsi l'instruction "jmp n,x" par un saut à un sous-programme implanté à l'adresse 0ffh ce sous-programme étant généré automatiquement par le programme de test.

L'observation nous permet de vérifier le bon déroulement du saut. Si l'instruction passait en séquence, le registre A contiendrait 0ffh au lieu de 0.

c) Le module DECLARE

Les déclarations d'ensembles se font au fur et à mesure des besoins des instructions d'itérations.

Dans les exemples précédents, on définira dans le module DECLARE :

```
regab:(a,b)
valiml:(opl,op2,op3,..,opn)
```

2.2. Résultats obtenus

On s'aperçoit qu'à la fin de la classe 1, on aura testé :

- tous les registres,
- certains opérateurs de calcul (incrémentation. décalage, .),
- les transferts de registres à registres,

Les classes suivantes complèteront le test pour :

- les autres opérations UAL,
- les différents modes d'adressage,
- .....

suivant l'ordre défini pour le test.

L'importance du logiciel ROBIN se résume dans les tableaux ci-dessous :

Pour le microprocesseur 6800 :

CLASSES	nombre d'instructions d'itération	nombre d'instructions assembleurs générées
1	12	2243
2	17	2363
3	3	88
4	10	910
5	5	330
6	5	413
7	2	96
TOTAL	54	6443

Pour le microprocesseur Z80 de ZILOG :

CLASSES	nombre d'instructions d'itération	nombre d'instructions assembleurs générées
1	12	6875
2	25	13337
3	4	387
4	6	435
5	8	539
6	6	155
TOTAL	61	21728

On remarque que le maximum d'instructions est généré dans les deux premières classes pour chacun des microprocesseurs. Ceci est dû au fait que dans ces classes sont testés les éléments de mémorisation, les micro-opérations et la conformité du graphe.
Les autres classes ne vérifient que la conformité du graphe.

Le nombre d'instructions assembleurs générées pour le Z80 est beaucoup plus important que pour le 6800 parce que le Z80 possède :

- 2 bancs de 11 registres chacun au lieu de 6 registres pour le 6800,
- 150 types d'instructions au lieu de 72.

En conclusion à cet exemple d'application, on remarque que

l'étape de préparation pour aboutir au classement des instructions reste longue et fastidieuse. D'autant plus, qu'on a seulement réalisé des programmes de test pour des microprocesseurs 8 bits. Une étude a été menée pour le microprocesseur 68000 de Motorola, l'étape de préparation aboutit à 80 graphes partitionnés en 26 classes.

C'est pourquoi, il est nécessaire d'automatiser :

- la classification des instructions,
- et la détermination des opérandes de test pour chaque instruction.

CHAPITRE IV. PARTICIPATION A LA CONCEPTION D'UN TESTEUR

1 Introduction

Nous nous proposons d'étudier la conception et la réalisation d'un testeur pour microprocesseurs 8 bits. Ce testeur servira à valider la méthode de test, développée en première partie, et à soulever les divers problèmes et contraintes concernant la réalisation d'un testeur "universel" (dont la gamme d'application s'étendra aux microprocesseurs 16 bits et aux circuits annexes).

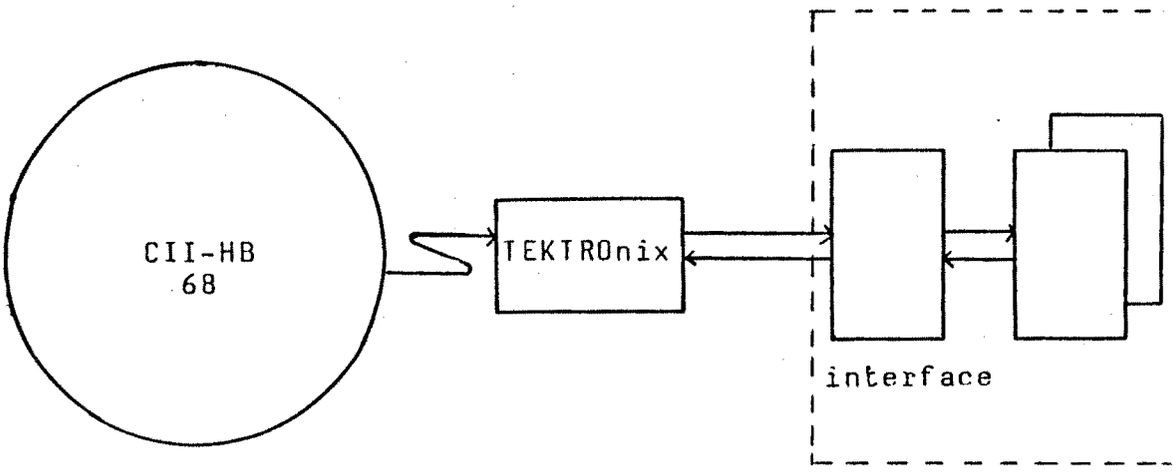
2 Objectifs de ce testeur

Ce système de test est un organe de :

- dialogue avec un opérateur,
- stockage des programmes de test et des résultats attendus,
- gestion du test.

Pour développer ce système de test le plus rapidement possible on utilisera comme organe de stockage et de dialogue : l'outil de développement (TEKTROnix) disponible à l'Atelier de Microinformatique.

De plus, cet outil possède un logiciel de communication qui permettra le transfert aisé des programmes de test. ceux-ci étant générés sur le CII-HB 68 du CICG (Centre Interuniversitaire de Calcul de Grenoble).



génération des programmes de test

stockage des programmes de test

carte microprocesseur sous test et mémoire

Il nous reste à définir le testeur qui se décomposera en :

- une carte interface (ou carte maitre),
- une carte microprocesseur sous test (ou carte esclave),
- deux cartes mémoires accessibles par la carte maitre et par la carte esclave.

a) La carte microprocesseur sous test

Cette carte esclave contiendra :

- le microprocesseur à tester,
- des circuits de mise à niveau des signaux,
- des circuits d'adaptation aux bus de fond de panier,
- une horloge.

C'est une carte individualisée, spécifique à chaque type de microprocesseurs.

b) Les cartes mémoires

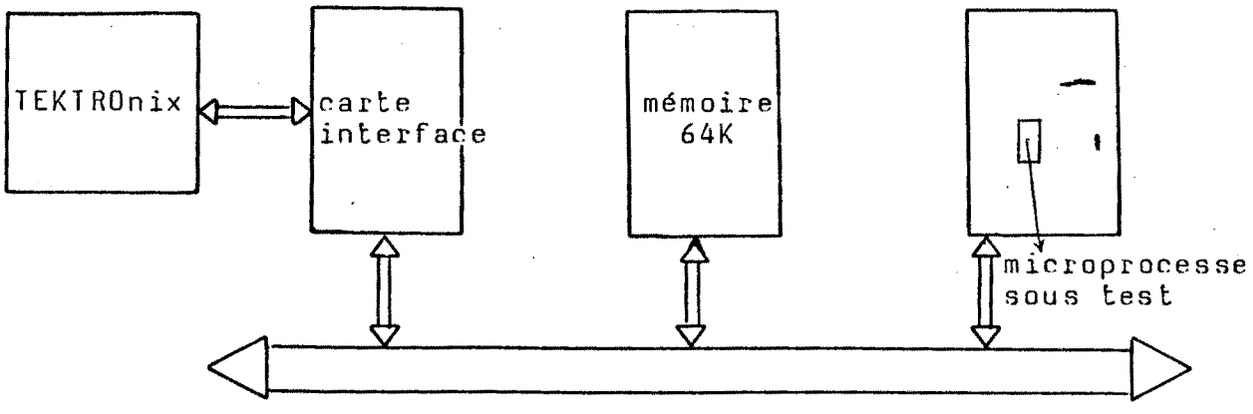
Elles contiendront au total 64K ce qui correspond à l'espace maximum d'adressage des microprocesseurs 8 bits existants (leur bus d'adresse étant de 16 fils).

c) La carte interface

Cette carte assure la communication avec le TEKTROnix pour :

- le transfert des modules de test du TEKTROnix vers la mémoire du testeur,
- le transfert des résultats produits par le microprocesseur sous test vers la mémoire de masse du TEKTROnix.

Elle assure aussi la communication avec la carte sous test pour la gestion des signaux nécessaires au lancement et fin d'exécution d'un module de test.



ARCHITECTURE DU TESTEUR

3. Déroulement d'un programme de test

Le déroulement d'un programme de test se fera module par module (c'est à dire classe par classe).

Le test de chaque module comprend quatre phases :

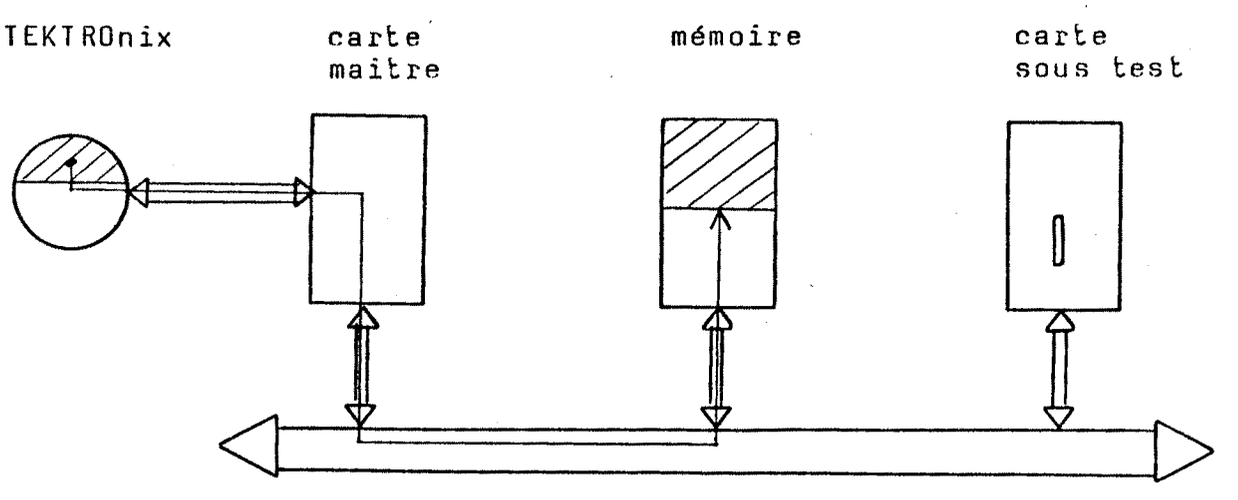
a) - Chargement du module de test en mémoire du testeur,

b) - Lancement de l'exécution du module de test : le microprocesseur sous test exécute ce module de test et, en fin d'exécution, envoit une interruption sur le microprocesseur maitre,

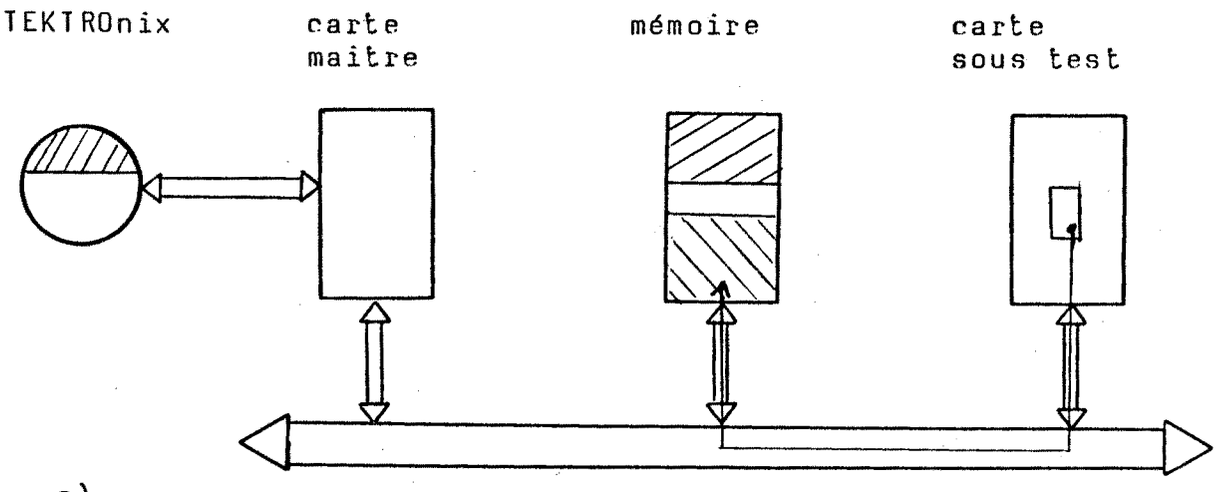
c) - Sur interruption, le microprocesseur maitre recupère les résultats de test obtenus et les charge dans la mémoire de masse du TEKTROnix,

d) - La comparaison des résultats obtenus avec des résultats justes pré-établis se fera sur le TEKTROnix.

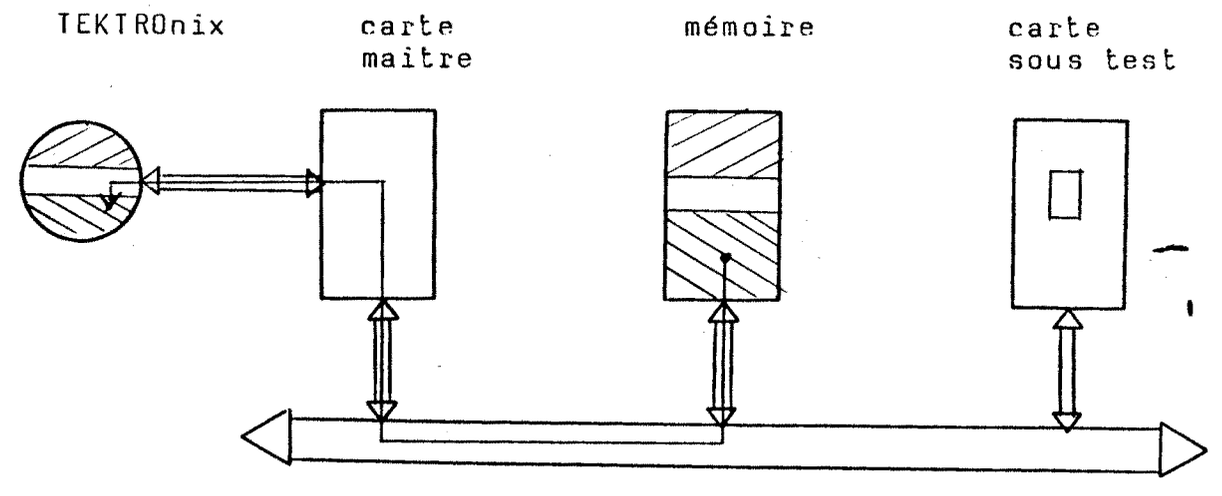
a)



b)

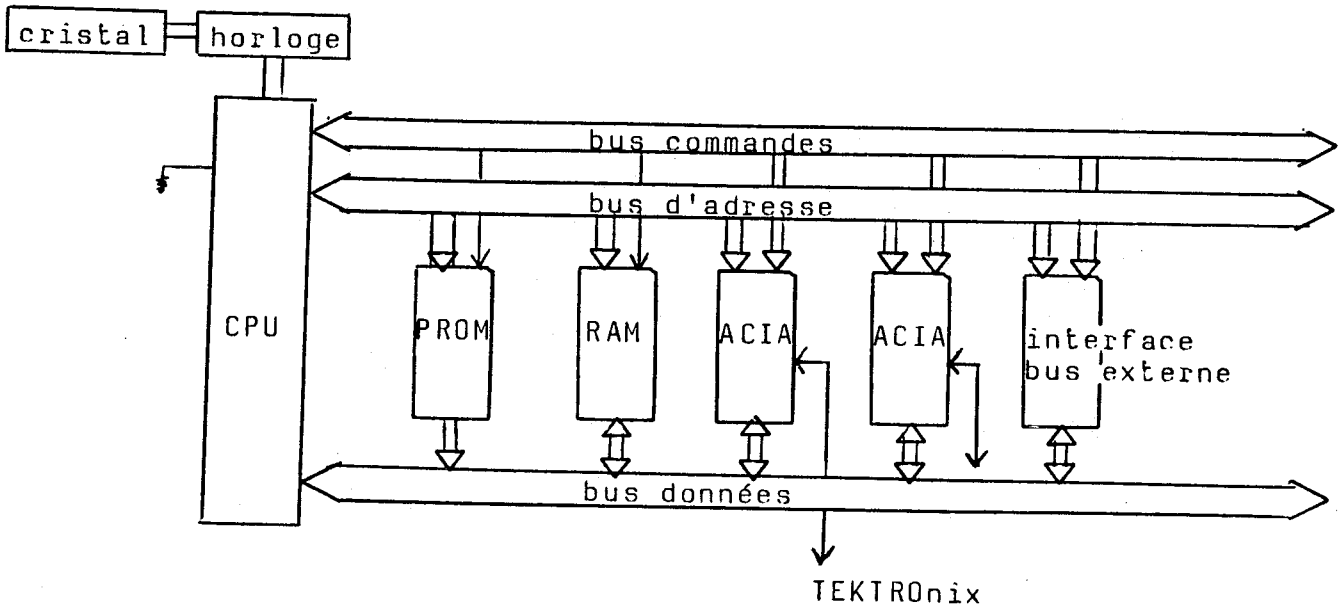

TEKTROnix
carte maitre
mémoire
carte sous test

c)


TEKTROnix
carte maitre
mémoire
carte sous test

4. Choix d'une architecture pour la carte maitre

La carte maitre sera une carte à microprocesseur dont l'architecture est donnée ci-dessous :



La mémoire est composée :

- d'une PROM de 4K octets qui contiendra le moniteur,
- d'une RAM statique de 2K octets.

Les circuits d'interface série (ACIA) permettent la communication avec le TEKTROnix et avec, éventuellement une console.

L'interface avec le bus externe assure la communication entre :

- la carte maitre et les cartes mémoires,
- la carte maitre et la carte esclave.

5. Problèmes d'adressage

D'aprés la méthode de test,le microprocesseur sous test doit pouvoir adresser 64K (cartes mémoires).
D'autre part, la carte maitre nécessite 6K pour le moniteur et son espace de travail, le microprocesseur maitre adresse aussi des ports d'entrées-sorties. Il doit aussi pouvoir adresser les 64K externes.
Ces 6K de mémoires (moniteur,..) ne peuvent être implantés dans la mémoire externe, compte tenu des contraintes imposées par la méthode de test. Les programmes et les résultats de test ont besoin de toute la mémoire externe.

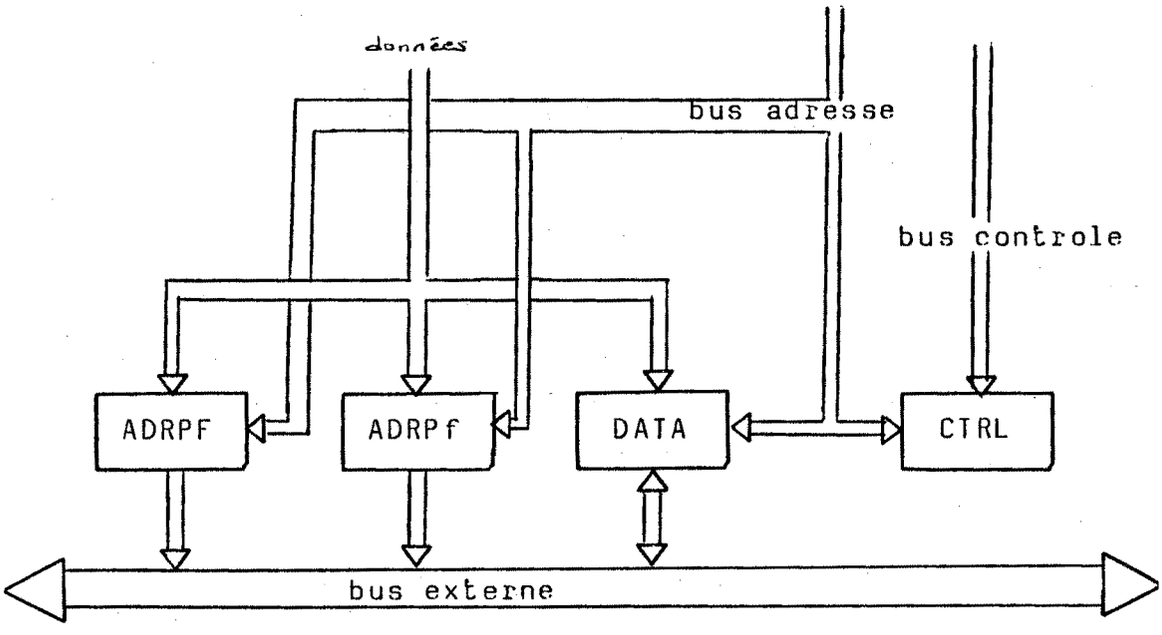
On a résolu ce problème d'adressage de la manière suivante:

- on adresse la mémoire externe comme un dispositif externe de stockage,
- l'interface aux bus externes sert à adresser la mémoire externe et comprend :
  - un latch pour le poids fort de l'adresse (ADRPF),
  - un latch pour le poids faible de l'adresse (ADRPf),
  - un buffer pour les données (DATA),
  - un buffer de controle (CTRL).

Un accès à la mémoire externe se fera par :

- un chargement de l'adresse sur les latchs "adresse",
- une validation des signaux de controle et des

données (en lecture ou écriture) par une instruction de lecture ou écriture dans le buffer DATA.



Principe d'accès à la mémoire externe

LD A,PF	Chargement dans ADRPF
OUT (ADRPF),A	du poids fort de l'adresse.
LD A,Pf	Chargement dans ADRPf
OUT (ADRPf),A	du poids faible de l'adress
LD A,data	Chargement de la donnée
LD (DATA),A	dans DATA.

CONCLUSION

Nous avons présenté dans ce mémoire une méthode de test fonctionnel, applicable à tout microprocesseur, afin d'établir une bibliothèque de programmes de test.

Une première version "semi-automatique" de génération des programmes de test (le logiciel ROBIN) est opérationnelle. Elle permet de fournir un logiciel de test pour un microprocesseur quelconque à la demande et dans un délai trés court.

Cette version devrait être améliorée pour être rendue plus automatique : la phase de classification des instructions, pour l'instant manuelle, sera automatisée ainsi que la détermination des opérandes de test.

La conception d'un testeur 8 bits a permis la validation des programmes de test pour les microprocesseurs 8 bits. L'enseignement tiré de cette étude permettra la réalisation d'un système de test universel pour microprocesseurs.

L'intérêt de ce travail réside essentiellement dans son application immédiate et dans sa facilité d'utilisation par des non spécialistes des problèmes de test.

ANNEXE 1. LANGAGE ROBIN DECRIT PAR LA GRAMMAIRE DE BACKUS

<langage ROBIN> := <titre> <définition de déclarations> <suite>
FIN

<suite> := <définition de procédure> <définition d'algorithme>|
<définition d'algorithme> <définition de procédure>|
<définition d'algorithme>

a) <titre>

<titre> := TITRE "chaine de caractères"

b) <définition de déclarations>

<définition de déclarations> := DECLARE: <déclarations>
<déclarations> := <description de déclaration>|
<description de déclaration> <déclarations>
<description de déclaration> := <nom d'ensemble> : (<ensemble d'éléments>)
<ensemble d'éléments> := <élément>|
<élément> <ensemble d'éléments>
<élément> := identificateur|
valeur
<nom d'ensemble> := identificateur

c) <définition de procédure>

<définition de procédure> := PROCEDURE <procédure>

<procédure> := <instruction>|
<instruction> <procédure>

<instruction> := <instruction assembleur>|
<macro-instruction>

<macro-instruction> := FILE <nom de macro> <instructions assembleurs> ENDF

<nom de macro> := identificateur

<instructions assembleurs> := <instruction assembleur>|
<instruction assembleur>
<instructions assembleurs>

<instruction assembleur> := <partie d'instruction>|
<partie d'instruction> <instruction assembleur>

<partie d'instruction> := identificateur|
symbole|
valeur

d) <définition d'algorithme>

<définition d'algorithme> := ALGORITHM: <instructions>

<instructions> := <description d'instruction>|
<description d'instruction> <instructions>

<description d'instruction> := <instruction assembleur>|
<appel de macro>|
<itération>

<itération> := ITER: <ensembles d'itération> <instructions d'itération> F-ITER

<ensembles d'itération> := <ensemble d'itération>|
<ensemble d'itération> <ensembles d'itération>

<ensemble d'itération> := <mot-cle> (<ensemble d'éléments>)|
<mot-cle> (<ensemble d'éléments>)*|
ø

<mot-clé> := source|
codop|
puits|
s2|
s3

<appel de macro) := COPY <nom de macro>

<instructions d'itération> := <initialisation> <exécution> <observation>

<initialisation> := INIT: <instructions itératives> <2ème iter>|
ø

<2ème iter> := ITER: <ensembles d'itération>|
ø

<exécution> := EXEC: <instructions itératives>

<observation> := OBSV <instructions itératives>

<instructions itératives> := <instruction itérative> , ;
<instructions itératives>

<instruction itérative> := <instruction assembleur>|
<appel de macro>|
<instruction particulière>

<instruction particulière> := <instruction assembleur>
< <mot-cle> > <instruction assembleur>|

< <mot-cle> > <instruction assembleur>

<instruction assembleur> < <mot-cle> >

ANNEXE 2. DESCRIPTION DE ROBIN EN GASEL

```
titre "description de Robin le 31.6.80"
lexique
     type lettre+a  = a b c d e f
     type lettre+b  = g i j k l m n o p q r s t u v w x y z +
     type lettre+h  = h
     type zero+un   = 0 1
     type chiffres  = 2 3 4 5 6 7 8 9
     type dollar    = $
     type dieze     = #
     type apost     = apost
     type perluet   = perluet
     type symbols   = . < ( + ! * ) ; - / , % > ? : @ ' =
     type space     = blanc finligne
     type nondef    = nondef

actions
   val+dec,val+bin,val+hex,chain+c,symbol,ident,diese,err+lexi,     & 01-08 &
   valeur+h
automate
   1 :    space              *1 ;
          lettre+a,lettre+b, lettre+h   2 ;
          zero+un,chiffres    3 ;
          dollar             *4 ;
          dieze              *5 ;
          apost              *6 ;
          perluet            *9 ;
          nondef        out  err+lexi;
          autre               8 ;
   2 :    lettre+a,lettre+b,lettre+h,zero+un,chiffres       2 ;
          autre          out  ident          & identificateur  & ;
   3 :    zero+un,chiffres    3 ;
          lettre+h       out valeur+h;
          lettre+a            10 ;
          autre          out  val+dec         & valeur decimale & ;
   4 :    zero+un             4 ;
          autre          out  val+bin         & valeur binaire  & ;
   5 :    zero+un,chiffres,lettre+a     5 ;
          lettre+b,lettre+h       out diese;
          autre          out  val+hex         & valeur hexadecimale & ;
   6 :    apost              *7 ;
          autre               6 ;
   7 :    apost               6 ;
          autre          out  chain+c         & chaine de caracteres & ;
   8 :    autre          out  symbol         & symboles speciaux & ;
   9 :    perluet            *1 ;
          autre              *9 ;
   10 :   zero+un,chiffres,lettre+a     10 ;
          lettre+h      out valeur+h ;
          autre         out err+lexi ;
syntaxe
actions rg+titre, rg+ens, rg+elem,rg+val,assemb1,assemb0,assemb2,deb+algo,
```

```
        iter, fin+iter, mot+cle, cher+ens,
        deb+instr, fin+instr, rg+point, cher+mcl, maj+point, iter1,
        rg+macro, fin+macro, copy1, copy2, etoile
 classes
   <valeur>
   <chaine>
   <symbole>
   <idf>
   <motcle> : (source=1,codop=2,puits=3,s2=4,s3=5)
&-----------------------------------&
automate principal:
&-----------------------------------&
          'titre'                          go p2;
          erreur 500;
          autre                           *go recup;
p2:       <chaine>        =rg+titre=       go p3;
          erreur 500;
          autre                           *go recup;
p3:       'declare'                        call declare,p4;
          erreur 501;
                      &pas de declarations&
          autre                           *go recup;
p4:       'procedure'                      call procedure,p5;
          'algorithm'                      call algorithm,p6;
          erreur 502 fatale;
                      &ni proc. ni algo&
p5:       'algorithm'                      call algorithm,p7;
          erreur 503 fatale;
                       &manque algo&
p6:       'procedure'                      call procedure,p7;
          autre                           *go p7;
p7:       'fin'                            stop;
          erreur 505 fatale;
recup:    'declare'                       *go p3;
          'procedure'                     *go p4;
          'algorithm'                     *go p4;
          'fin'                           *go p7;
          autre                            go recup;
&-----------------------------------&
      automate  declare :
&-----------------------------------&
          ':'                              go d1;
          autre                           *go d1;
d1:       <idf>           =rg+ens=         go d2;
          erreur 507;
                      &manque nom d'ensemble&
          autre                           *go recupdec;
d2:       ':'                              go d3;
          autre                           *go d3;
d3:       '('                              go d4;
          erreur 508;
```

```
                                &manque ( apres nom d'ens&
          autre                                  *go recupdec;
d4:       <idf>                  =rg+elem=        go d5;
          <valeur>               =rg+val=         go d5;
          erreur 510;
                                &aucun element dans l'ens&
          autre                                  *go recupdec;
d5:       ','                                     go d6;
          ')'                                     go d7;
          erreur 511;
                                &declarations d'ensemble incorrecte&
          autre                                  *go recupdec;
d6:       <idf>                  =rg+elem=        go d5;
          <valeur>               =rg+val=         go d5;
          erreur 512;
          autre                                  *go recupdec;
d7:       'procedure'                            *ret;
          'algorithm'                            *ret;
          <idf>                  =rg+ens=         go d2;
          autre                                   go recupdec;
recupdec: 'procedure'                           *ret;
          'algorithm'                           *ret;
          autre                                  go recupdec;
&-----------------------------------&
      automate procedure :
&-----------------------------------&
          ':'                    =assemb2=        go pr1;
          erreur 515;
                                &manque : apres procedure&
          autre                                  *go pr1;
pr1:      'algorithm'            =assemb0=       *ret;
          'fin'                  =assemb0=       *ret;
          'file'                                  go pr2;
          autre                                   go pr1;
pr2:      <idf>                  =rg+macro=       go pr3;
          erreur 516;
          autre                                 *go pr1;
pr3:      'endf'                 =fin+macro=      go pr1;
          autre                                   go pr3;
&-----------------------------------&
     automate algorithm :
&-----------------------------------&
          ':'                    =deb+algo=       go s0;
          erreur 520 fatale;
                                &manque : apres algorithme&
s0:       'iter'                                  go s1;
          autre                  =assemb1=       *go aligne;
s1:       ':'                    =iter=           go s9;
          erreur 528;
                                &manque : apres iter&
          autre                                 *go s2;
```

```
s9:       <motcle>                          *call ensiter,s2;
          autre                             *go s2;
s2:       'init'                             call instruct,s3;
          'exec'                             call instruct,s4;
          erreur 536;
          autre                             *go recupalgo;
s3:       'exec'                             call instruct,s4;
          'iter'          =iter1=     go s6;
          erreur 529;

                         &manque exec ou iter dans un iter&
          autre                             *go recupalgo;
s4:       'obsv'                             call instruct,s5;
          erreur 530;

                         &pas d'observation dans un iter&
          autre                             *go recupalgo;
s5:       'f+iter'        =fin+iter=         go s0;
          erreur 531;

                         &manque f+iter en fin d'iteration&
          autre                             *go recupalgo;
s6:       ':'                                go s7;
          autre                             *go s7;
s7:       'exec'                             call instruct,s4;
          <motcle>                          *call ensiter,s8;
          erreur 538;
          autre                             *go recupalgo;
s8:       'exec'                             call instruct,s4;
          erreur 539;
          autre                             *go recupalgo;
sligne:   'iter'          =assemb0=          go s1;
          'fin'           =assemb0=         *ret;
          'procedure'     =assemb0=         *ret;
          'copy'                             go sligne1;
          autre                              go sligne;  &ligne assembleu
sligne1:  <idf>           =copy2=            go sligne;
          erreur 516;
          autre                              go recupalgo;
recupalgo: 'f+iter'                          go s0;
          'iter'                             go s1;
          'fin'                             *ret;
          'procedure'                       *ret;
          autre                              go recupalgo;
&-----------------------------------------&
     automate ensiter :
&-----------------------------------------&
          <motcle>        =mot+cle=          go s10;
          erreur 532;
                         &pas de motcle&
          autre                             *go recupens;
s10:      '('                                go s11;
          erreur 533;
                         &manque '(' &
```

```
            autre                                   *go recupens;
s11:        <idf>           =cher+ens=               go s12;
            <valeur>        =rg+val=                 go s12;
            erreur 510;
            autre                                   *go recupens;
s12:        ','                                      go s14;
            ')'                                      go s13;
            erreur 511;
            autre                                   *go recupens;
s13:        'init'                                  *ret;
            'exec'                                  *ret;
            '*'             =etoile=                 go ensiter;
            autre                                   *go ensiter;
s14:        <idf>           =rg+elem=                go s12;
            <valeur>        =rg+val=                 go s12;
            erreur 512;
            autre                                   *go recupens;
recupens:   'init'                                  *ret;
            'exec'                                  *ret;
            autre                                    go recupens;
&-----------------------------------&
       automate instruct :
&-----------------------------------&
            ':'             =deb+instr=              go i1;
            erreur 540;
            autre                                   *go i1;
i1:         ';'             =fin+instr=              go i1;
            '<'             =rg+point=               go i2;
            'copy'                                   go i5;
            autre                                   *go i4;     &ligne assembleur&
i2:         <motcle>         =cher+mcl=              go i3;
            erreur 541;
            autre                                   *go i4;
i3:         '>'             =maj+point=              go i1;
            erreur 542;
            autre                                   *go i4;
i4:         'f+iter'                                *ret;
            'iter'                                  *ret;
            'exec'                                  *ret;
            'obsv'                                  *ret;
            'fin'                                   *ret;
            autre                                    go i1;
i5:         <idf>           =copy!=                  go i1;
            erreur 516;
            autre                                   *go i4;
            end
```

ANNEXE 3. CODAGE INTERNE DES CARACTERES

a) Tableau CODIN

Le tableau CODIN contient les codes des caractères.

CARA	CODE	CARA	CODE	CARA	CODE	CARA	CODE	CARA	CODE
A	10	N	23	0	0	<	39	>	52
B	11	O	24	1	1	(	40	?	53
C	12	P	25	2	2	+	41	:	54
D	13	Q	26	3	3	&	42	#	55
E	14	R	27	4	4	!	43	@	56
F	15	S	28	5	5	$	44	'	57
G	16	T	29	6	6	*	45	=	58
H	17	U	30	7	7	)	46	"	59
I	18	V	31	8	8	;	47	autres	60
J	19	W	32	9	9	-	48		
K	20	X	33	_	36	/	49		
L	21	Y	34		37	.	50		
M	22	Z	35	.	38	%	51		

b) Tableau TYPCAR

Ce tableau contient les types internes des caractères

CARA	A->F	H	G->Z._	0,1	2->9	$	=	"	&	¬	blanc	illégal
TYPCAR	1	2	3	4	5	6	7	8	9	10	11	12

ANNEXE 4. STRUCTURE DES TABLES

Certaines tables du logiciel ROBIN ont une structure de table d'identificateur, structure utilisée dans le logiciel GASEL.
Ces tables, appelées tables d'IDF utilisent un codage particulier des identificateurs pour économiser de la place en mémoire.

a) Codage des IDF

Ce codage consiste à placer 3 caractères sur les entiers Fortran des machines de 16 bits.
Les caractères autorisés sont :

- les 26 lettres de l'alphabet et le caractère souligné '_',
- et les 10 chiffres.

Un IDF doit toujours commencer par une lettre Si I, J et K représentent les codes de 3 caractères la formule de codage est la suivante :

Soit II = I+1, JJ = J+1, KK = K+1 (en l'absence de caractères. JJ et KK valent 0 ce sont les mots qui contiennent seulement 1 ou 2 caractères).

On procède comme suit :

si II>19 mot = ((II-39)*38 +JJ)*38+KK

sinon mot = -(((19-II)*38 +38-JJ-1)*38 +38-KK-1

Cette codification a aussi l'avantage de conserver l'ordre

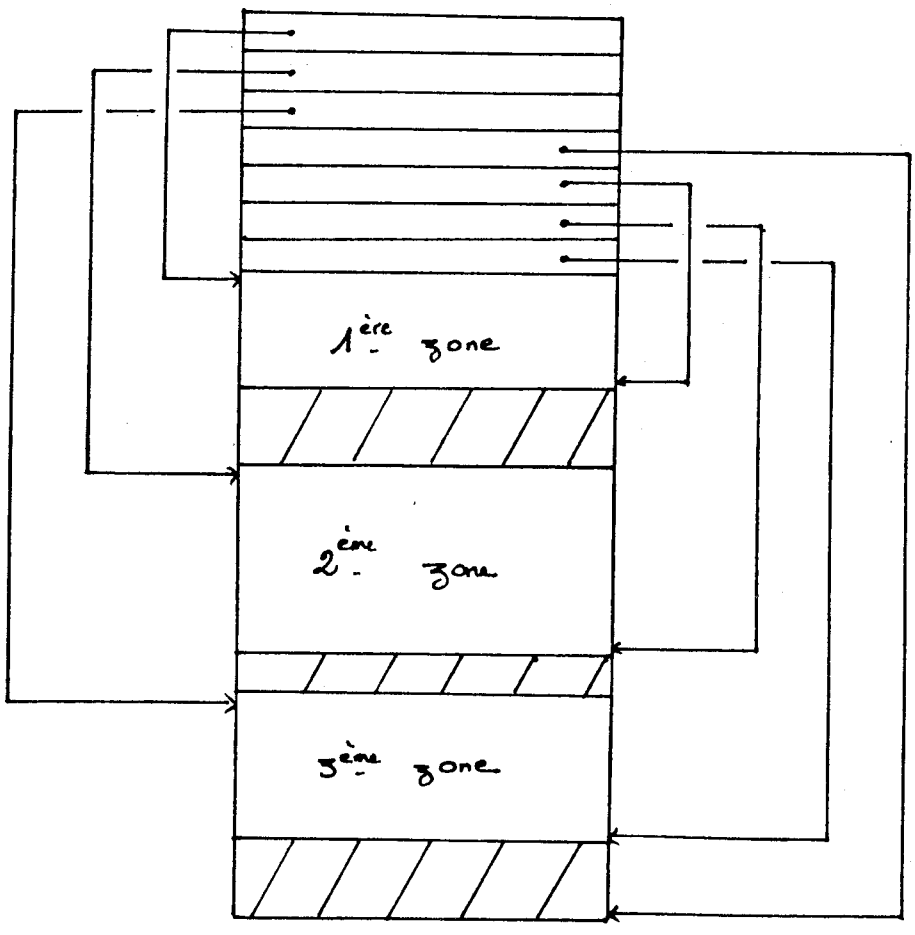
alphabétique.

b) Structure des tables d'IDF

Une table d'IDF est divisée en trois zones :

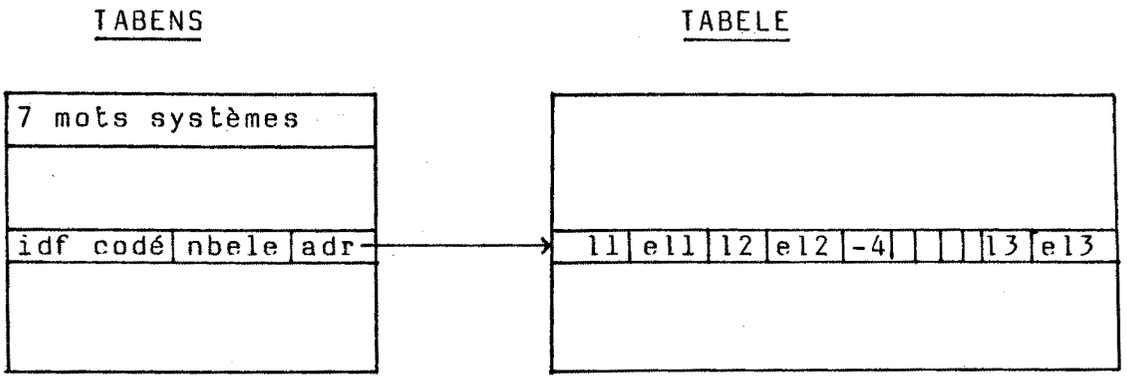
- 1ère zone : IDF dont la longueur est comprise entre 1 et 3 mots,
- 2ème zone : IDF dont la longueur est comprise entre 4 et 6 mots,
- 3ème zone : IDF dont la longueur est comprise entre 7 et 9 mots.

En tête de chaque table, on trouve une zone qui contient des informations de service. Ce sont des pointeurs de début et de fin de zone ainsi que la taille maximum de la table.



ANNEXE 5. TABLES PROPRES AU LOGICIEL ROBIN

a) Rangement des ensembles d'itérations



nom des ensembles éléments des ensembles

Le tableau TABENS contient les noms des ensembles d'itérations

- ce tableau a une structure de table d'IDF,
- les noms d'ensembles sont définis dans le module DECLARE,
- les informations internes sont :
  - . idf codé : le nom d'ensemble codé.
  - . nbele : nombre d'éléments que contient cet ensemble,
  - . adr : adresse du premier élément de l'ensemble dans le tableau TABELE.

Le tableau TABELE contient les éléments des ensembles d'itération

- les éléments des ensembles d'itération sont soient

des identificateurs (<9 car.), soient des valeurs décimales, hexadécimales ou binaires.

- si l'élément est un identificateur, le premier mot contiendra sa longueur (11) et à la suite l'identificateur (e11) codé sur 1, 2 ou 3 mots.
- si l'élément est une valeur, le premier mot contiendra le nombre de valeurs successives rencontrées en négatif, et à la suite les valeurs lues.

b) Utilisation des ensembles d'itération

	TCLASS		TINFO				NBENT
1	source	3 0	nbele	adr	iele	adrc	i
2	codop	1 1					
3	puits	2 2					
4	s2	0 0					
5	s3	0 0					

Le tableau TCLASS est un tableau de 10 mots qui donne des informations sur les 5 niveaux d'itération pour une instruction d'itération donnée :

- la première partie du tableau précise quelle est la boucle la plus extérieure (dans cet exemple : le puits), etc... jusqu'à la boucle la plus inférieure (ici : le codop),
- la deuxième partie du tableau précise si les éléments doivent itérer normalement (0) ou

parallèlement (≠0)

Le tableau TINFO est un tableau de 20 mots qui contient des informations relatives aux ensembles en cours de traitement :

- nbele : nombre d'éléments total à traiter,
- adr : adresse dans TABELE du premier élément de l'ensemble,
- iele : le traitement s'effectue sur le ième élément.
- adrc : adresse dans TABELE du ième élément.

Le tableau NBENT est un tableau de 5 mots.

Si en cours de traitement adrc pointe sur un nombre de valeurs (un nombre négatif dans TABELE) alors nbent(i) précisera sur quelle valeur le traitement s'effectue.

c) Séquences de base d'une instruction d'itération

Les instructions de base d'une instruction d'itération sont rangées dans un tableau TINSTR.

Exemple :

```
iter : source (E1) puits (E2)
       init: LD <source>,#FFH;
       exec: ADD <puits>,<source>;
       obsv  CALL OBS1:
f-iter
```

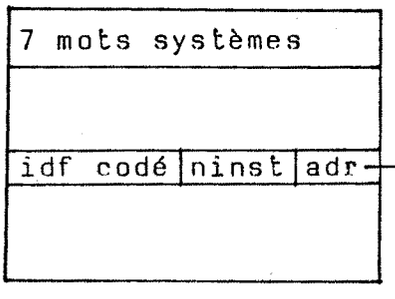
TINSTR

```
LD <1>,#FFH;
ADD <2>,<1>;
CALL OBS1;
```

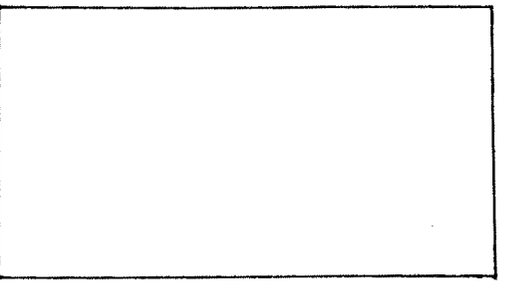
Les parties variables sont stockées sous la forme <chiffre>. La taille de ce tableau étant de 2400 mots, cela permet de mémoriser 30 lignes assembleurs.

d) Macro-instructions

TABMAC



TMACRO



Le tableau TABMAC contient les noms des macro-instructions et a une structure de table d'IDF. A un nom de macro (idf codé) est associé le nombre d'instructions de cette macro (ninst) ainsi que son adresse (adr) de la première instruction de la macro dans TMACRO.

Le tableau TMACRO contient les instructions assembleurs associées au nom de la macro-instruction.

ANNEXE 6. ACTIONS SEMANTIQUES

rg-titre

- rangement du titre dans le tableau NEWTIT
- initialisation des pointeurs de TABENS et TABMAC

fin-rg-titre

rg-ens

- nbele = 0
- rangement du nom d'ensemble dans TABENS suivi de nbele et adr (adresse dans TABELE où seront rangés les éléments propres à cet ensemble.

fin-rg-ens

rg-elem

- nbele = nbele +1
- rangement dans TABELE de l'identificateur lu précédé de sa longueur
- si "module DECLARE" alors M.A.J. de nbele dans TABENS
  sinon M.A.J. de nbele dans TINFO

fin-rg-elem

rg-valeur

- nbele = nbele + 1
- rangement dans TABELE de la valeur lue précédée du nombre de valeurs lues successives
- si "module DECLARE" alors M.A.J. de nbele dans TABENS

| sinon M A.J. de nbele dans TINFO

fin-rg-valeur

deb-algo

- sauvegarde de l'adresse du dernier élément rangé dans TABELE

fin-deb-algo

iter

- initialisation des variables, TCLASS et TINFO à 0
- initialisation à blanc du tableau TINSTR

fin-iter

mot-cle

- nbele = 0
- M.A.J. de TCLASS c.a.d rangement dans TCLASS de la valeur associée au mot-clé rencontré

fin-mot-cle

cher-ens

- si l'IDF rencontré appartient à TABENS
  alors M.A.J. de TINFO à partir des informations de TABENS
  sinon M.A.J. de TINFO (c'est un élément de l'ensemble et non un nom d'ensemble)
  alera rg-elem

fin-cher-ens

deb-instr

- calcul du pointeur (iinstr) du tableau TINSTR où sera rangée l'instruction d'itération

fin-deb-instr

rg-point

- recopie de la chaine de caractères lue dans TINSTR
- M.A.J. du pointeur iinstr

fin-rg-point

cher-mcl

- recopie de la valeur associée au mot-clé dans TINSTR

fin-cher-mcl

maj-point

M.A.J. du pointeur dans l'instruction lue

fin-maj-point

fin-instr

si non macro-copy alors recopie de la chaine lue dans TINSTR

fin-fin-instr

fin-iter

- vérification de conformité du tableau TCLASS
- pour tout mot-clé rencontré dans une instruction d'itération

faire

iele = 0

tant que iele < nbele

faire

- M.A.J. de "adrc" du ième élément dans TINFO
- remplacement dans TINSTR de la valeur associée au mot-clé par l'élément correspondant à

l'adresse adrc de TABELE

- écriture sur fichier de sortie des instructions assembleurs générées

- iele = iele + 1

finfaire

finfaire

fin-fin-iter

rg-macro

- rangement du nom de la macro dans TABMAC

fin-rg-macro

fin-macro

- de rg-macro à fin-macro : rangement dans TMACRO des lignes lues

fin-fin-macro

copy1

- recopie de la macro dans TINSTR

fin-copy1

copy2

recopie directe de la macro sur fichier de sortie

fin-copy2

étoile

- M.A.J. dans TCLASS du degré de parallélisme

fin-étoile

ANNEXE 7. LISTE DES ERREURS DE GASEL

1 Fin de fichier prématurée

2 Pile des automates vide

3 Pile des automates pleine

4 Table des IDF ou table des clés trop petite

5 Trop d'erreurs

10 Tableau inéditable (dimension trop grande)

21 Chaine d'entrée trop longue, on la tronque

22 Pile des automates non vide à l'arrêt

23 Pile de codage des entiers pleine

24 Trop de digits pour le format demandé

30 Unité lexicale incorrecte

31 Nom de tableau trop long (on le tronque)

32 Double définition IDF

33 IDF trop long (on le tronque)

34 Tableau mal initialisé

35 Double définition d'IDF

40 Erreur sur une constante

101 CLASSES : Symbole "<" manquant

102 CLASSES : IDF classe manquant

103 CLASSES : Symbole ">" manquant

104 CLASSES : IDF clé manquant

105 CLASSES : symbole "(" manquant

106 CLASSES : symbole "=", ",", ")" manquant

107 CLASSES : symbole " " manquant

108 CLASSES : valeur clé manquante

111 Syntaxe : erreur dans la partie syntaxique

113 double définition d'IDF (VERFID)

114 ACTIONS SYNTAXIQUES : manque nom d'actions

117 spécif 'AUTOMATE' manquante (erreur système)

118 nom d'automate syntaxique manquant

119 Cas non testé (autres cas dans ERREUR ou AUTRE)

120 Paragraphe non terminé par ERREUR ou AUTRE

122 condition entre quotes de SYMBOLE ou IDF

123 clé interdite en condition

124 AUTOMATE SYNTAXIQUE : symbole "'" manquant

125 IDF action ou classe non défini

126 AUTOMATE SYNTAXIQUE : nom d'action manquant

127 AUTOMATE SYNTAXIQUE : symbole "=" manquant

128 AUTO SYNTA : instruc 'GO', 'CALL', 'RET', 'STOP'

129 AUTO SYNTA : étiquette manquante

130 'CALL' : symbole "," manquant

131 AUTO SYNTA : num erreur manquant ou incorrect

132 AUTO SYNTA : erreur de syntaxe dans ERREUR

133 instruction 'GO' manquante

134 AUTO SYNTA : symbole ";" manquant

135 Table de l'automate à générer trop petite

136 TITRE manquant ou mal défini

137 TITRE tronqué à 40 caractères

138 IDF 'LEXIQUE', 'CODAGE' ou 'SYNTAXE'

140 deux analyses syntaxiques (une de trop)

141 deux déclarations de classes (une de trop)

142 deux déclarations d'actions pour l'auto synta

143 les étiquettes suivantes ne sont pas définies

150 deux codages (un de trop)

151 deux analyses lexicales (une de trop)

152 caractère déjà entré

153 Idf non représenté par une lettre unique

154 nombre représenté par un chiffre décimal unique

155 états non consécutifs

156 table de travail pleine

158 Etats de l'automate lexical incorrects

161 Nom de type absent

162 manque "=" après un nom de type

163 absence du numéro d'état

164 AUTO LEXICAL : symbole "." manquant

165 AUTO LEXICAL : abs. de AUTRE ou classe de transition

166 absence de numéro d'état

167 symbole ";" manquant aprés une transition

168 unité lexicale IDF, SYMB-SPE ou VALEUR

169 idem erreur 168

170 CODAGE : symbole "=" manquant

171 manque une valeur de codage

172 code interne d'un symbole spécial incorrect (>61)

180 AUTO LEXICAL : manque nom d'action lexicale en sortie

182 LEXIQUE : erreur dans la partie lexicale

183 ACTIONS LEXICALES : manque noms d'actions

LISTES DES ERREURS DE ROBIN

500 manque le titre

501 manque le module DECLARE

502 ni PROCEDURE, ni ALGORITHM

503 manque ALGORITHM

505 manque FIN

507 manque nom d'ensemble

509 manque "(" après un nom d'ensemble

510 aucun élément dans l'ensemble

511 manque "," ou ")"

512 manque IDF ou valeur après ","

515 manque ":" après PROCEDURE

516 manque IDF après FILE ou COPY

520 manque ":" après ALGORITHM

528 manque ":" après ITER

529 ni EXEC ni ITER

530 manque OBSV

531 manque F-ITER

532 manque mot-clé

533 manque "(" après mot-clé

536 manque INIT ou EXEC

538 manque EXEC ou un mot-clé

539 manque EXEC

540 manque ":" au début d'une instruction

541 manque un mot-clé après "<"

542 manque ">" après un mot-clé

600 tableau TABELE trop petit

601 mot-clé déjà rencontré

602 iclass > 5

603 tableau TINFO trop petit

604 pas d'itération à traiter

605 erreur de valeur dans TCLASS

606 manque ">" après "<"

607 tableau TSORTI trop petit

608 instruction mal positionnée

609 dépassement du tableau TINSTR

610 instruction d'itération vide

611 nom de macro non connu

612 nombre d'éléments différents pour des ensembles parallèles

613 tableau TCLASS non conforme

ANNEXE 8. GRAPHES ET PROGRAMME DE TEST DU MICROPROCESSEUR 6800

MESURE D'ACCESSIBILITE

. mémoire et extérieur		t = t' = 1
. registres	A, B SP, PC, IX	t = t' = 2
. flag F		t = t' = 3

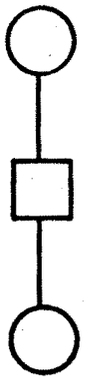
remarque : le registre de code condition est chargé et observé à partir de A

. remarque : SP pointe sur la première zone vide de la pile

ex. : PUSH A := A → Mem (SP)
SP ← SP - 1

CLASSE 1

S-classe : a

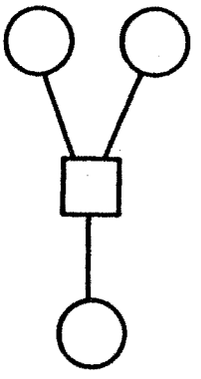


a

(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
(0,2)		CLRA CLRB LDAA # n LDAB # n			LDX # m LDS # m JMP # m	
(2,2)		TAB TBA				
	*	DAA				
	*	INCA		*	INS	
	*	INCB		*	INX	
	*	DECA		*	DES	
	*	DECB		*	DEX	
	*	NEGA				
	*	NEGB				
	*	ASLA LSRA NOP				
	*	ASLB LSRB				
	*	ASRA				
	*	ASRB				
	*	COMA				
	*	COMB				
2,3)		TAP TPA				
		TSTA TSTB				
3,3)	*	CLC				
	*	CLI				
	*	CLV				
	*	SEC				
	*	SEI				
	*	SEV				

CLASSE 2

S-classe : a)

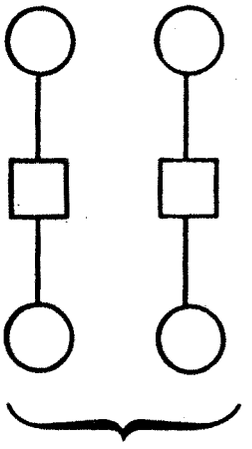


a

(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
(0,1)	*	CLR m				
(1,2)		LDAA n LDAB n LDAA m LDAB m			LDX n LDX m LDS n LDS m JMP n,X	
2,1)		STAA n STAB n STAA m STAB m			STX n STS n STX m STS m	
2,2)	* *	ADDA # n ADDB # n SUBA # n SUBB # n ANDA # n ANDB # n ORAA # n ORAB # n EORA # n EORB # n ABA # SBA # BITA # n BITB # n CMPA # n CMPB # n CBA ROLA ROLB RORA RORB		* *	BRA n CPX # m	

CLASSE 2

S-classe : b

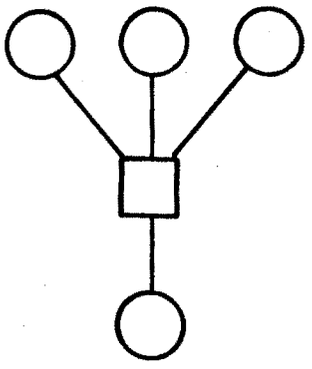


b

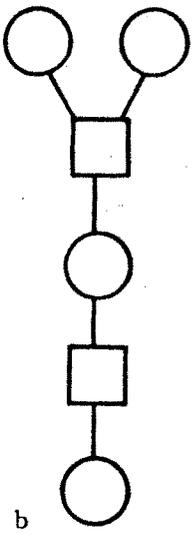
(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
(2,2)				*	TXS	
				*	TSX	

CLASSE 3

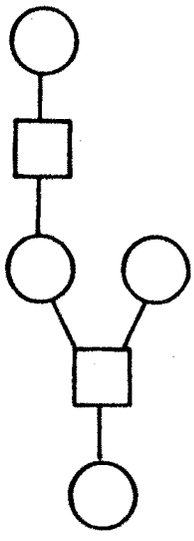
S-classe : a, b, c, d



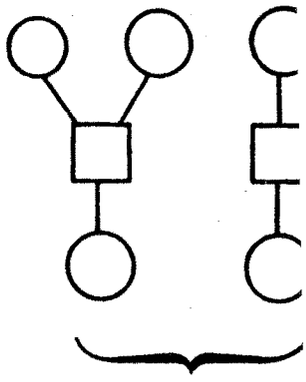
a



b



c

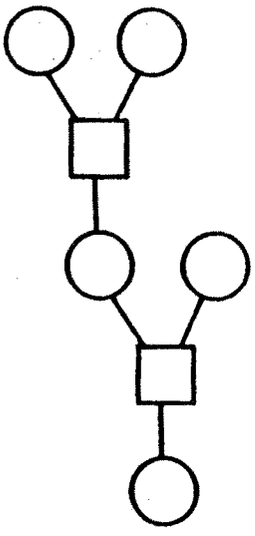


d

(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
) (3,2)	* * * *	ADCA # n ADCB # n SBCA # n SBCB # n				
) (2,3)	*	TST m				
) (2,2)		RTS RTI PULA PULB				
) (2,2)		PSHA PSHB				

CLASSE 4

S-classe : a)

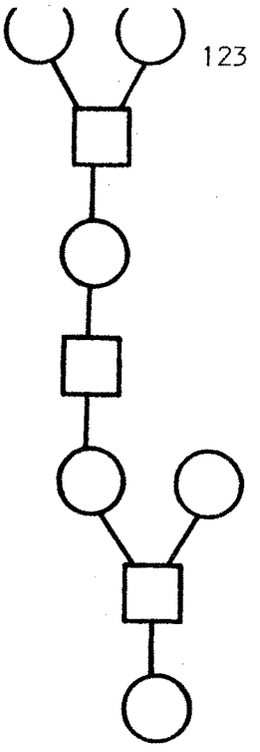


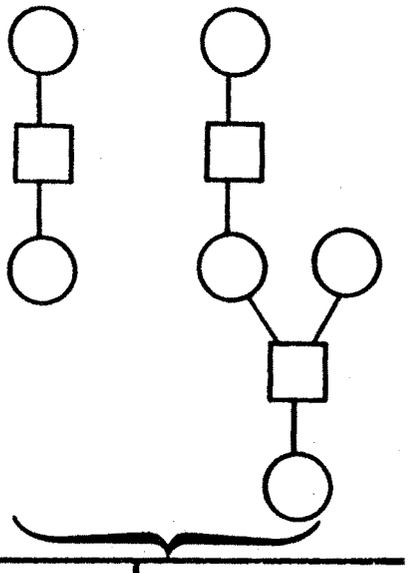
a

(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
(2,1)	*	CLR n,X STAA n,X STAB n,X			STX n,X STS n,X	
(2,2)		LDAA n,X LDAB n,X			LDX n,X LDS n,X	
	*	ADDA n			BCC n	
	*	ADDB n			BCS n	
	*	SUBA n			BEQ n	
	*	SUBB n			BGE n	
	*	ANDA n			BGT n	
	*	ORAA n			BHI n	
	*	ORAB n			BLE n	
	*	EORA n			BLS n	
	*	EORB n			BMI n	
	*	ANDB n			BNE n	
					BVC n	
					BVS n	
					BPL n	
					BLT n	
(2,3)	*	BITA n				
	*	BITB n				
	*	BITA m				
	*	BITB m				
	*	CMPA n		*	CPX n	
	*	CMPB n		*	CPX m	
	*	CMPA m				
	*	CMPB m				

CLASSE 4

S-classe : b, c

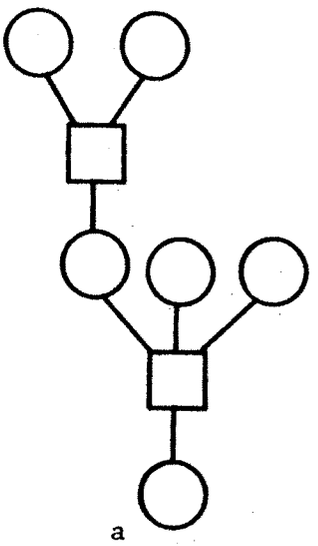




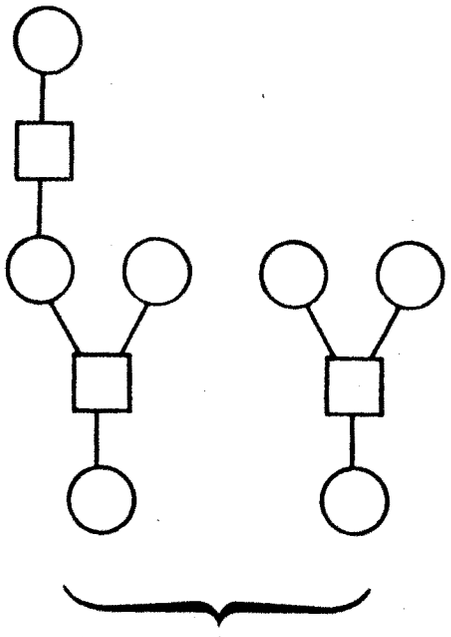
(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
b) (1,1)	* * * * * * *	INC m DEC m NEG m ASL m ASR m LSR m COM m				
c) (2,2)					JSR m BSR m	

CLASSE 5

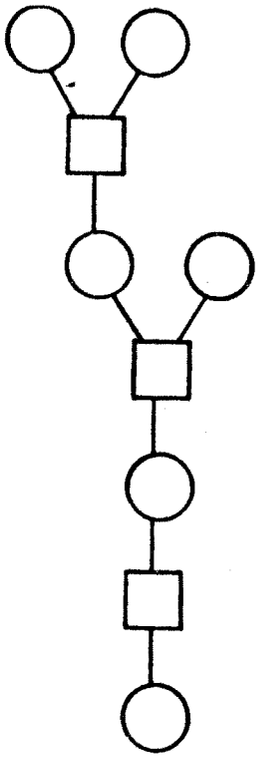
S-classe : a, b, c



a



b

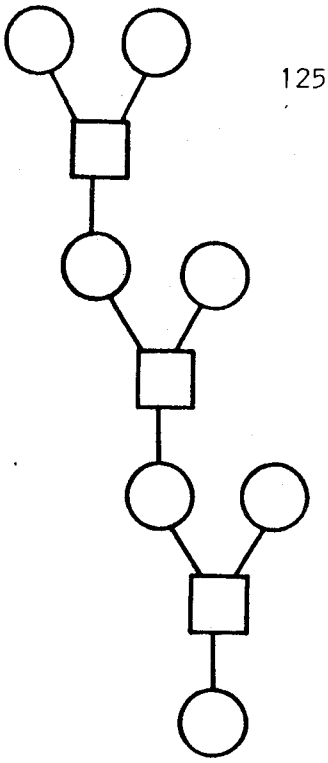


c

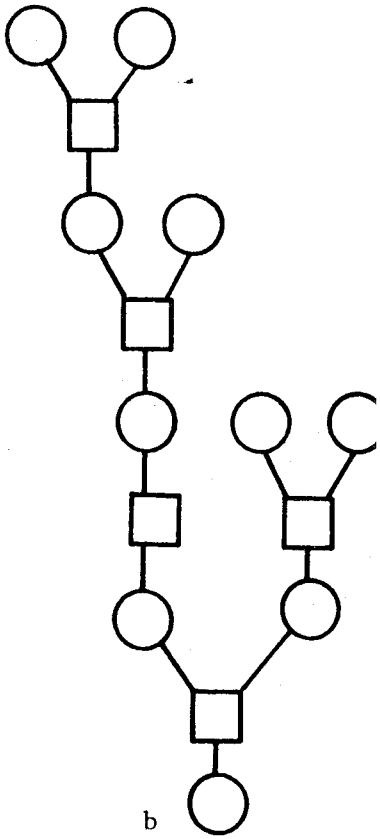
(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
) (3,2)	*	ADCA n				
	*	ADCB n				
	*	SBCA n				
	*	SBCB n				
	*	ADCA m				
	*	ADCB m				
	*	SBCA m				
	*	SBCB m				
) (2,2)					JSR n,X	
) (2,3)	*	TST n,X				

CLASSE 6

S-classe : a,b



a

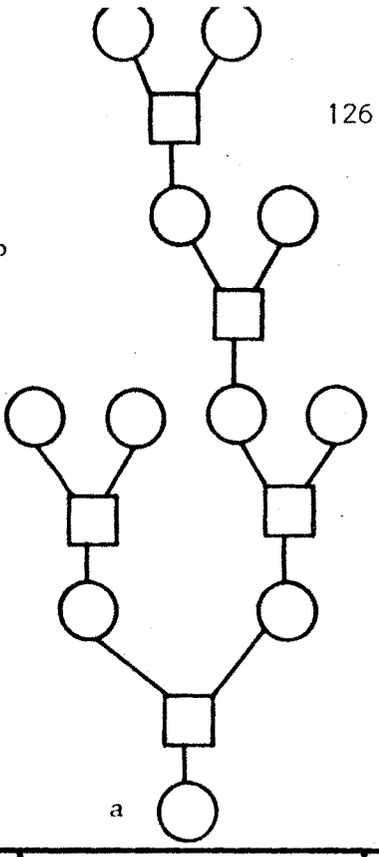


b

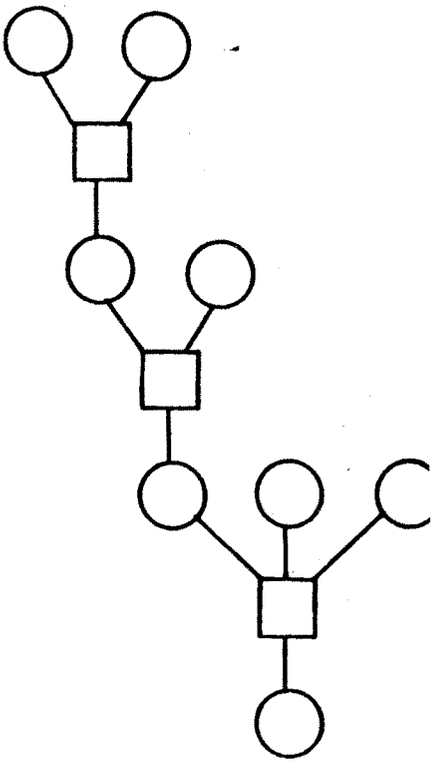
(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
(2,2)	*	ADDA n,X				
	*	ADDB n,X				
	*	SBCA n,X				
	*	SBCB n,X				
	*	ANDA n,X				
	*	ANDB n,X				
	*	ORAA n,X				
	*	ORAB n,X				
	*	EORA n,X				
	*	EORB n,X				
(2,3)	*	BITA n,X				
	*	BITB n,X				
	*	CMPA n,X				
	*	CMPB n,X		*	CPX n,X	—
(3,1)	*	ROR m				
	*	ROL m				
)						
(2,1)	*	INC n,X				
	*	DEC n,X				
	*	NEG n,X				
	*	ASL n,X				
	*	ASR n,X				
	*	LSR n,X				
	*	COM n,X				

CLASSE 7

S-classe : a, b



a



b

(T,T')	OP	FORMAT 8 BITS	COMMENTAIRES	OP	FORMAT 16 BITS	COMMENTAIRES
) (3,1)	*	ROR n,X				
	*	ROL n,X				
) (3,2)	*	ADCA n,X				
	*	ADCB n,X				
	*	SBCA n,X				
	*	SBCB n,X				

```
titre "test du 6800"

declare:
&
          declarations d'ensembles de registre
&
regab:(a,b)
regxs:(x,s)
&
         valeurs immediates
&
valim1:(1h,2h,4h,8h,10h,20h,40h,80h)
valim2:(1,2,4,8,10h,20h,40h,80h,100h,200h,400h,800h,1000h,2000h,4000h,8000h)
valim3:(0,55h,0aah,0ffh)
valim4:(0,0ffh)
valdaa:(0,0ch,0c0h,0cch)
flagdaa:(0c0h,0c1h,0e0h,0e1h)
valinc:(7fh,0,0fffh,80h,3fh,40h,1fh,20h,0fh,10h,7,8,3,4,1,2)
valinc16:(valinc16)
procedure:
        section init
adrsto set 0fffffh
init1     clra
          clrb
          ldx  #0
          tap
          rts
&
&
          file  obs1
          staa adrsto
          stab adrsto-1
          tpa
          staa adrsto-2
          stx adrsto-3
          sts adrsto-5
adrsto set adrsto-7
          endf
algorithm:
&
*****************************************************************************

      classe1     s-classe1

*****************************************************************************
&
;
;   ldaa #n   ldab #n
;
;
```

```
adrsto set 0fffh
         lds #2h

iter: puits(regab) source(valim1)
   init:          jsr init1;
   exec:          lda<puits>  #<source>;
   obsv:          copy obs1;
f+iter
;
;
;
;     ldx #m   lds #m
;
;
;
iter:  puits(regxs) source(valim2)
   init:          jsr init1;
   exec:          ld<puits> #<source>;
   obsv:          copy obs1;
                  lds #2;
f+iter
;
;
;
;       jmp #m
;
;
;
          jsr init1
          jmp #a1
          ldaa  #0ffh
a1        ldab #0ffh
          copy obs1
; ---------------------------------------------------------------------
;
;
;    tab    tba
;
;
;
iter: codop(tab,tba) * source(regab)  s2(valim1)
   init:          jsr init1;
                  lda<source>  #<s2>;
   exec:          <codop>;
   obsv:          copy obs1;
f+iter
;
;
;
;     inca  incb  deca   decb   nega  negb
;
iter: codop(inc,dec,neg) source(regab) s2(valinc)
   init:          jsr init1;
                  lda<source>  #<s2>;
   exec:          <codop><source>;
   obsv:          copy obs1;
f+iter
;
```

```
;
;    asla  aslb  asra  asrb  lsra  lsrb
;
iter: codop(asl,asr,lsr) source(regab) s2(valim3)
   init:          jsr init1;
                  lda<source>  #<s2>;
   exec:          <codop><source>;
   obsv:          copy obs1;
f+iter
;
;
;     coma  comb
;
iter: source(regab) s2(valim4)
   init:          jsr init1;
                  lda<source> #<s2>;
   exec:          com<source>;
   obsv:          copy obs1;
f+iter
;
;
;   daa
;
iter: source(valdaa) * s2(flagdaa)
   init:          jsr init1;
                  ldaa  #<s2>;
                  tap;
                  ldaa  #<source>;
   exec:          daa;
   obsv:          copy obs1;
f+iter
;
;
;     ins  inx  des  dex
;
iter: codop(in,de) source(regxs) s2(valinc16)
   init:          jsr init1;
                  ld<source>  #<s2>;
   exec:          <codop><source>;
   obsv:          copy obs1;
                  lds #2;
f+iter
;
;
;    nop
;
          jsr init1
          nop
          copy obs1
;
;
```

```
;    testa  testb
;
iter: source(regab) s2(0,0ffh,7fh)
   init:          jsr init1;
                  lda<source>  #<s2>;
   exec:          tst<source>;
   obsv:          copy obs1;
ftiter
; ---------------------------------------------------------------------
;
;    clc  cli  clv  sec  sei  sev
;
iter: codop(clc,cli,clv,sec,sei,sev) source(0,0ffh)
   init:          jsr init1;
                  ldaa  #<source>;
                  tap;
                  clra;
   exec:          <codop>;
   obsv:          copy obs1;
ftiter
fin
```

BIBLIOGRAPHIE

1 ROBACH C., "Test et testabilité de systèmes informatiques", thèse Docteur ès Sciences, Université de Grenoble, Juin 1979.

2 ROBACH C , SAUCIER G. et VELAZCO R. "Le test fonctionnel paramétrable de microprocesseurs", Revue R.A.I.R.O. Automatique/Systems Analysis and Control" vol. 14, n. 3, Octobre 1980. PP. 293-308

3 EYNARD J.P., LIOTHIN A., ROBACH C., SAUCIER G. et VELAZCO R., "Conception et réalisation d'une bibliothèque de programmes de test", Rapport intermédiaire de contrat IRIA, Projet Pilote SURF, Juin 1980.

4 BERGE C., "Théorie des graphes et ses applications" Collection Universitaire de Mathématiques, Dunod, Paris, 1963.

5 DENT J.J., "Diagnostic Engineering Requirements", Proc. A.F.I.P.S., S.J.C.C., 1968, pp. 503-507.

6 Tutorial on LSI testing (Warren 6 FEE), présenté à COMPCON Spring 77, San Francisco (USA), Février 1977, IEEE Catalog n. EHO 122-2.

7 BECKERS Y., FONTANILLE P., "Le Générateur d'Analyseurs G.A.S.E.L. Rapport interne, Atelier de Microinformatique, Grenoble, Novembre 1978.

8 BECKERS Y., "le système de production d'assembleurs G A.G.E" Rapport interne, Atelier de Microinformatique, Grenoble, Octobre 1979.

9 ADAM J.M., "Participation à la réalisation d'un générateur d'assembleurs", Rapport de Stage, Atelier de Microinformatique, Grenoble, 1978.

10 VAUQUOIS B., Cours polycopié : calculabité des langages - Logique et Programmation - C3, Maitrise d'Informatique, Grenoble, 1969.

11 GRIFFITHS M., Langage Algorithmique et Compilateurs - C4 Maitrise d'Informatique, Grenoble, Octobre 1969.

12 LILEN H., "Du microprocesseur au micro-ordinateur", Edition Radio, Paris, 1977.

13 AUMIAUX , "L'emploi des microprocesseurs", Edition Masson, 1977.

14 ZAKS R. et LE BEUX P., "Les microprocesseurs", SYBEX, 1978.