Sur la description et la vérification de processeurs microprogrammables

Alberto Ruiz de Olano y Ruiz de Larrea

N° d'ordre : 1999

Année 1977

THESE

présentée

DEVANT L'UNIVERSITE PAUL SABATIER DE TOULOUSE (SCIENCES)

en vue de l'obtention

du TITRE de DOCTEUR de SPECIALITE E.E.A.

Option Automatique

par

Alberto RUIZ DE OLANO Y RUIZ DE LARREA

Ingénieur EUIT de Mondragon, Espagne
Maître ès Sciences

SUR LA DESCRIPTION ET LA VERIFICATION DE PROCESSEURS MICROPROGRAMMABLES

Soutenue le 21 juin 1977, devant la commission d'examen :

MM.	G. GRATELOUP	Président
	G. ARREGUI	
	A. COSTES	Examinateurs
	M. COURVOISIER	
	M. DIAZ	
	M. GALINIER	
	J.F. MEYER	

A Alicia-Lara,

A Andrée,

A nos parents et amis.

" JAKINTZAK AZKATUKO ZAITU "

" C'est à celui qui domine sur les esprits par la force de la vérité, non à ceux qui font des esclaves par la force de la violence, c'est à celui qui connaît l'univers, non à ceux qui le défigurent, que nous devons nos respects "

F. Marie AROUET (1694-1778)

A V A N T - P R O P O S

Ce mémoire représente la synthèse des travaux de recherche que nous avons effectués au sein de l'équipe "Représentation , Analyse et Sécurité des Systèmes de Commande" du Laboratoire d'Automatique et d'Analyse des Systèmes du C. N. R. S..

Nous voudrions que Monsieur le Professeur G. GRATELOUP, Directeur du Laboratoire d'Automatique et d'Analyse des Systèmes, qui a bien voulu accepter de présider notre jury de thèse, trouve dans ces lignes l'expression de notre sincère reconnaissance.

Nous remercions également :

Monsieur G. ARREGUI, Professeur à l'E.U.I.T. de Mondragon (GUIPUZCOA).

Monsieur A. COSTES, Maître de Conférences à l'I. N. P. de TOULOUSE.

Monsieur M. COURVOISIER, Maître Assistant à l'U. P. S. de TOULOUSE.

Monsieur M. DIAZ, chargé de Recherche au C. N. R. S.

Monsieur M. GALINIER, Maître de Conférences à l'U. P. S. de TOULOUSE.

Monsieur le Professeur J.F. MEYER de l'Université du MICHIGAN - ANN ARBOR, U. S. A.

qui nous font l'honneur de participer à notre jury de thèse.

Que Monsieur M. DIAZ, Responsable de l'équipe R. A. S. S. C. au L. A. A. S. qui a su nous orienter au cours de nos travaux dans un cadre d'amitié et confiance ainsi que Monsieur M. COURVOISIER, dont les remarques ont enrichi la présentation de ce mémoire, reçoivent ici le témoignage de notre gratitude.

Nous exprimons nos meilleurs sentiments d'amitié et reconnaissance à tous les membres de l'équipe R A S S C, en particulier Messieurs P. AZEMA et J. RENALIER, ainsi qu'à tout ceux qui directement ou indirectement, ont contribué à la réalisation de ce mémoire, notamment Messieurs F. CEREJA et J. M. PONS.

Enfin, nous voulons remercier le Centre Régional des Oeuvres Universitaires et spécialement, Monsieur le Professeur H. MASCART qui nous ont donné la possibilité, grâce à leur soutien matériel, de nous consacrer entièrement à ces travaux de recherche.

. . .

S O M M A I R E

I N T R O D U C T I O N

L'évolution actuelle des circuits intégrés, en particulier celle des mémoires et des microprocesseurs, conduit les concepteurs à développer des systèmes de commande microprogrammés.

La complexité qui peut être atteinte lors de la conception de tels systèmes rend nécessaire l'utilisation de procédures formelles permettant la vérification des microprogrammes réalisés.

Une des approches possibles consiste à effectuer une double description de la commande, l'une décrivant les primitives de la spécification fonctionnelle, l'autre permettant la caractérisation de la réalisation microprogrammée ; ceci implique en premier lieu d'effectuer ces deux descriptions de façon précise, par exemple sous forme de programmes. Dans ce cas les deux programmes résultant permettent de vérifier la microprogrammation réalisée en montrant que ces deux programmes, d'une part le programme décrivant les spécifications, d'autre part le programme définissant le niveau microprogrammé, conduisent à des résultats identiques.

Une telle approche, qui paraît réaliste et prometteuse, est celle que nous avons adoptée dans notre étude. En conséquence, le travail présenté dans ce mémoire traite essentiellement des deux problèmes les plus importants qui s'y rattachent, c'est-à-dire : d'une part, le choix d'une méthode de description formelle des spécifications et d'autre part l'étude de procédures permettant de vérifier la validité de la microprogrammation.

En fait, ces deux problèmes sont en corrélation étroite, car l'étude de la vérification nécessite que l'objet à valider soit décrit d'une manière précise et adéquate pour cette étude.

En général la description des processeurs est faite en associant au langage courant une série d'illustrations graphiques. Ce genre de description ne se prête pas facilement à une étude de vérification, principalement à cause des ambiguïtés du langage utilisé.

Une manière formelle de décrire la structure d'un système

est celle qui repose sur le langage VDL (13). En effet, ce langage peut être utilisé afin de décrire formellement la sémantique des langages de programmation et en particulier permet de spécifier de façon formelle un processeur, à la fois au niveau structurel et au niveau des actions effectuées entre les registres. Un tel système de définition peut être organisé à la manière d'une "Machine abstraite" (14). Ce concept de machine abstraite, d'abord utilisé pour spécifier la sémantique des langages de programmation, sert aussi pour la description des processeurs. La machine abstraite correspondante comporte un ensemble d'instructions qui définissent en quelque sorte la structure du processeur.

Une extension de ce système de définition de processeurs consiste à utiliser certains opérateurs du langage APL (10) (19) en tant qu'opérateurs de base puisqu'ils sont très bien adaptés au traitement de vecteurs et tableaux binaires.

Dans le domaine plus précis des processeurs microprogrammés la description peut être effectuée à deux niveaux différents, ces deux niveaux correspondant à la manière dont est menée la conception d'un processeur. Dans une première étape de la conception, la spécification du fonctionnement est élaborée au plus haut niveau en tenant compte de l'utilisation pour laquelle il est destiné ; nous appelons ce niveau, le niveau "spécification". A partir de ce niveau de description on peut prendre les décisions nécessaires en ce qui concerne la manière dont les différentes actions doivent être implémentées et, en particulier, dans le cas des processeurs microprogrammables la façon dont le microprogramme est conçu. Ce deuxième niveau constitue le niveau que nous appelons le niveau "réalisation microprogrammée".

Ces deux niveaux étant définis et décrits, le problème qui se pose alors est de vérifier si le deuxième niveau (la réalisation microprogrammée) réalise effectivement ce qu'on attend de lui, c'est-à-dire ce qui est spécifié par le premier niveau (spécification) considéré correct. Pour effectuer cette vérification, une procédure à deux étapes peut être utilisée. La première étape consiste à représenter les descriptions de ces deux niveaux sous la forme de deux programmes abstraits ; la deuxième consiste à

vérifier si ces deux programmes satisfont les conditions de la théorie de la simulation algébrique d'un programme par un autre (17), ce qui permettra en particulier de vérifier la validité de la microprogrammation réalisée par rapport à la description du niveau spécification.

Les descriptions qui utilisent les deux langages VDL/APL des deux niveaux d'un processeur microprogrammé semblent intéressantes dans le cadre de processeurs qui présentent un certain parallèlisme d'exécution dans leur fonctionnement et si, de plus, on prétend réaliser un système de vérification automatique. Néanmoins, pour une procédure "manuelle" destinée en particulier aux processeurs qui présentent des exécutions verticales ou séquentielles nous avons préféré réaliser les descriptions sous forme d'organigrammes en utilisant les opérateurs APL pour spécifier les opérations de base. Cette procédure de vérification "manuelle" est illustrée par un exemple d'application sur un processeur microprogrammé simple,connu sous le nom de S - Machine (7). De plus nous présenterons dans ce mémoire, l'application de la procédure de vérification à un processeur microprogrammable basé sur un microprocesseur "cascadable" dont nous avons préalablement réalisé une description sous forme d'organigramme de chacun des deux niveaux (21).

Ainsi, le premier chapitre du mémoire présente les deux approches de description de processeurs dont nous avons déjà fait mention dans cette introduction : la première, basée sur l'utilisation du langage VDL/APL et la seconde qui repose sur la notion d'organigramme, toutes deux précisées par l'utilisation d'opérateurs APL. Ces deux possibilités de description sont illustrées par un exemple d'utilisation : la S - Machine.

Lors du deuxième chapitre, nous exposons la théorie de la simulation algébrique entre programmes et détaillerons son application dans le cas de la S - Machine. La présentation sera menée à partir de la description sous forme d'organigramme de celle-ci.

Le troisième chapitre présente la structure d'un microcalculateur réel microprogrammable (de la famille AMD 2900), sa description en deux niveaux sous forme d'organigrammes et l'application de la procédure de vérification à cette double description du microcalculateur.

Afin de ne pas alourdir le texte, une annexe présente la description en langage VDL/APL des deux niveaux de la S- Machine, le contenu de sa mémoire de microprogramme et les programmes de simulation également en langage APL, de chacun des deux niveaux de description.

CHAPITRE I

DESCRIPTION FORMELLE DES PROCESSEURS

I-1. INTRODUCTION

Nous allons exposer, tout d'abord, le langage de définition de processeurs proposé par un groupe de recherche du Laboratoire IBM de Vienne (13). Il s'agit du langage de définition connu sous le nom de VDL (Vienna Definition Language).

Le langage et le système de définition que nous présentons ici sont un résumé de ceux conçus par le groupe de Vienne mais avec quelques différences mineures dues à LEE (14), introduites au niveau de la définition des instructions du système.

Cette présentation du langage VDL permet la compréhension de la définition VDL (réalisée par BIRMAN (1)) d'un processeur microprogrammé : la S - Machine (7).

La complexité de la mise en oeuvre du langage VDL, nous a amenés proposer ensuite une description simple et formelle de la S - Machine, description basée sur l'utilisation d'organigrammes structurés. Ceci nous permettra alors d'obtenir en premier lieu et facilement un programme de simulation de la S - Machine.

I-2. LANGAGE DE DEFINITION VDL

I-2-1. Sémantique du langage de définition VDL

Un système utilisé pour décrire les opérations qui traitent un ensemble de données pour obtenir un résultat désiré, dans un procédé de transformation, doit être basé sur un ensemble d'opérations fondamentales applicables à l'ensemble des données.

On peut montrer que les caractéristiques primitives nécessaires et suffisantes pour décrire les opérations d'un processeur qui exécute certaines transformations sous la direction et le contrôle d'un programme sont les suivantes :

a) Un ensemble de données (données du programme et renseignements sur le matériel) ;

b) Un ensemble fermé d'opérations sur l'ensemble de données ;
c) Un ensemble d'opérations pour décider de l'ordre d'exécution des opérations sur les données.

Les instructions des langages de programmation peuvent être classées dans le groupe b) ou bien dans le groupe c).

Dans un langage ou système de définition ces classes d'éléments primitifs sont aussi présentes et, de plus, sur elles, on construit une série d'extensions convenables.

I-2-1-1.L'ensemble des données

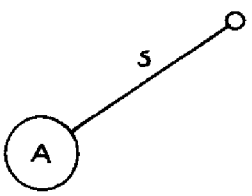
Dans un système de définition VDL, l'ensemble des données est représenté comme un ensemble de structures en arbre.

Un arbre est représenté par un ensemble de couples où chaque couple décrit une seule branche de l'arbre. Les deux éléments de ce couple sont : un sélecteur et un objet; le sélecteur étant un pointeur de l'objet et porteur du nom de la branche qui mène à l'objet.

Dans le langage des graphes orientés on dirait que l'arbre représentant l'ensemble des données est un graphe orienté sans circuit, où l'ensemble des successeurs de chaque noeud est disjoint des ensembles de successeurs des autres noeuds.

Un objet dit "nommé" est décrit par un couple <S:A> où S représente le sélecteur de l'objet A.

L'élément de l'ensemble des données le plus simple est l'arbre montré dans la figure suivante :

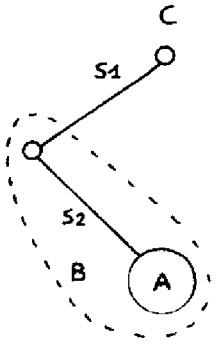


Objet simple
{ <S:A> }

Pour affecter une nomenclature aux différents arbres, on utilisera le symbole "=". Par exemple l'arbre $\{<S:A>\}$, s'il est appelé B sera écrit :

$$B = \{<S:A>\}$$

La représentation et la notation d'un objet composé se fait selon la manière montrée par l'exemple suivant :

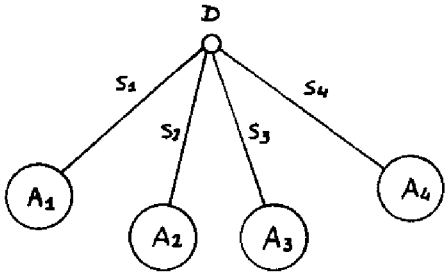


$C = \{<S_1 : B>\}$

où $B = \{<S_2 : A>\}$

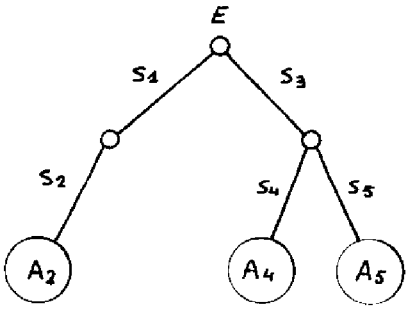
$C = \{<S_1 : \{<S_2 : A>\}>\}$

Un autre type d'objet composé est celui constitué de plusieurs objets simples émanant d'un même noeud :



$$D = \{<S_1 : A_1>, <S_2 : A_2>, <S_3 : A_3>, <S_4 : A_4>\}$$

A partir de ces objets il est possible de représenter un objet à "multiniveaux" comme celui de la figure suivante :



$$E = \{<S_1 : \{<S_2 : A_2>\}>, <S_3 : \{<S_4 : A_4>, <S_5 : A_5>\}>\}$$

L'ensemble de couples qui décrivent la structure d'un objet est dit "ensemble caractéristique" de l'objet.

Les objets qui représentent les "valeurs" dans un objet structuré sont nommés "objets élémentaires" (ils ne peuvent pas être subdivisés). En conséquence, ces objets élémentaires constituent les "feuilles" (ou noeuds terminaux) des arbres décrivant les données.

I-2-1-2. Opérations sur les données

Fonctions de sélection :

Une opération fondamentale dans ce système de définition est l'organisation de l'ensemble des données. On peut voir que l'opération consistant à se déplacer dans l'arbre nécessite seulement la connaissance du nom du sélecteur (S). Dans le cas d'un objet simple comme : B = $\{<S:A>\}$ une fois que l'on a l'objet B on peut sélectionner sa composante A, en appliquant le sélecteur (S), ce qui est représenté par S(B). L'application d'un seul sélecteur à un objet, donne comme résultat une "composante immédiate" de cet objet. Ainsi par exemple la composante immédiate de l'objet C = $\{<S_1 : \{<S_2 : A>\}>\}$, est l'objet dont l'ensemble caractéristique est $\{<S_2 : A>\}$.

Si un objet est composé de plusieurs branches, le sélecteur appliqué détermine le choix des composants immédiats à obtenir.

Par exemple, l'application du sélecteur S4 à D = $\{<S1 : A1>, <S2 : A2>, <S3 : A3>, <S4 : A4>\}$ fournit la composante A4.

Ainsi, afin de ne pas avoir d'ambiguïté, il est nécessaire que l'application d'un sélecteur à un objet ne produise qu'un seul résultat, toujours défini. Ceci amène à l'établissement de la règle suivante :

a) Les sélecteurs des composants immédiats des objets doivent être différents ;
b) Le résultat de l'application à un objet d'un sélecteur qui n'apparaît pas dans cet objet est l'objet nul, représenté par Ω .

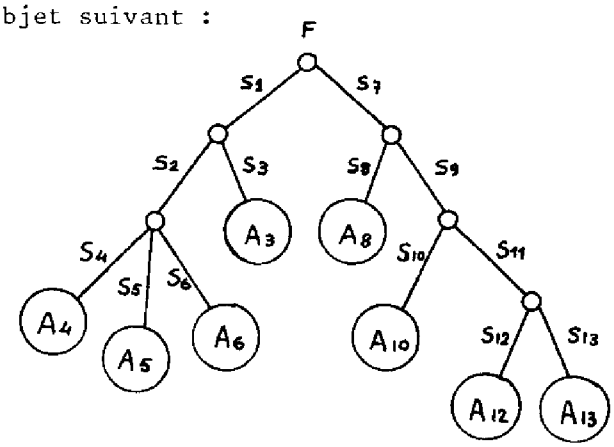
On déduit de ceci que l'application à un objet élémentaire d'un sélecteur quelconque entraîne l'obtention de l'objet nul Ω . Par extension, on peut dire que S(Ω) = Ω .

L'application successive de sélecteurs à un objet amène à une représentation de la forme :

S1 (S2 (S3 (S4 (A))))

ou bien plus simplement : S°1 S°2 S°3 S4 (A)

Un exemple d'application successive de sélecteurs peut être donné à partir de l'objet suivant :



où l'application S°12 S°11 S°9 S7 (F) donne comme résultat A12.

L'opérateur de mutation M :

Les opérations de déplacement à travers l'arbre ne sont pas suffisantes pour réaliser des opérations de modification. Une nouvelle opération est donc introduite dans le but d'effectuer des changements dans l'arbre des données. Cette opération de "mutation" est présentée comme :

$$\mathcal{M}\ (A\ ;\ <\ S\ :\ B\ >\ )$$

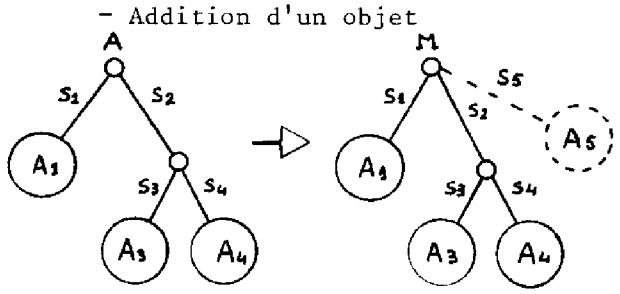
A est l'argument principal, représentant l'ensemble caractéristique de l'objet à muter et < S : B > est un objet "nommé" à ajouter à la copie de A.

Selon que S est un sélecteur existant ou non dans l'objet A à modifier, la mutation sera nommée modification (faite en remplaçant l'objet sélectionné par S dans A, par l'objet B) ou une addition (B est ajouté à l'objet A avec le sélecteur S).

Un cas particulier du remplacement est l'effacement où B est l'élément nul Ω.

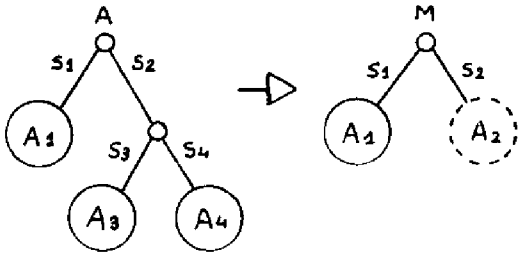
- Exemples des trois cas de mutation :

- Addition d'un objet



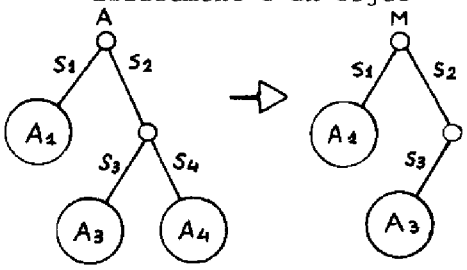
- $A=\{<S1:A1>,<S2:\{<S3:A3>,<S4:A4>\}>\}$
- $M=\ \mathcal{M}\ (A, <S5 : A5>)$

- Remplacement d'un objet



$M = \mu(A, \langle S2 : A2 \rangle)$

- Effacement d'un objet



$M = M(A, \langle S°4\ S2 : \Omega \rangle)$

Il faut noter que S2 ne disparaît pas du fait de son douple emploi pour A3 et A4.

Remarque : A partir de l'opérateur de mutation il est possible de définir aisément l'opérateur de construction μO par :

$$\mu O\ (\langle S : B \rangle) \underset{Df}{=} \mu\ (\Omega, \langle S : B \rangle)$$

On obtient ainsi l'ensemble des modifications possibles sur les données.

I-2-1-3. Opérations de contrôle

Relations entre objets :

Le but des opérations de contrôle est le choix d'un ensemble particulier d'opérations à effectuer, choix basé sur l'état actuel des objets dans le système.

Les fonctions qui permettent de telles décisions sont les fonctions de relation sur le domaine des valeurs représentant les deux possibilités vrai ou faux V,F , c'est-à-dire : $=, \neq, \geqslant, >, <, \leqslant, \in, \notin$.

Si l'on excepte le cas de l'ensemble des nombres naturels dans lequel tous ces opérateurs sont définis, alors seulement les opérateurs $=, \neq, \in$ et $\notin$ sont définis et applicables sur le domaine universel des objets.

Ainsi, dans le but de créer une structure qui permette la sélection d'un certain ensemble de données on est amené, dans le cas général, à l'utilisation d'un ensemble de prédicats.

Un prédicat indique l'égalité entre deux objets ou l'appartenance d'un objet à une classe d'objets spécifiée. La nomenclature de ces prédicats comporte la particule is - suivie de l'objet ou classe d'objets représenté entre parenthèses. Par exemple :

is -'a' ('a') = Vrai is - 'a' ('b') = Faux

La notation 'a' et 'b' indique que ces objets sont des objets élémentaires pour les différencier de a et b qui peuvent être des objets simples ou composés. L'inégalité est représentée comme :

« ¬ is - » ou bien « non is - »

Par exemple : ¬ is -'a' ('a') = Faux

Dans un système de définition il est souvent nécessaire de disposer d'un moyen pour classifier les objets. Une manière de décrire l'ensemble d'objets qui satisfont un prédicat est le suivant, illustré par un exemple :

is - digit = is -'o'v is -'1' v is -'2'v... vis -'9'

Ce prédicat appliqué à un digit donnera la valeur "VRAI".

Pour faciliter l'écriture, on écrit cette représentation sous la forme suivante :

is - digit = o v I v2... v8 v9

Le concept de relation d'équivalence entre deux objets peut être défini comme la relation qui requiert que les structures des deux objets satisfassent un prédicat commun (à la différence de la relation d'égalité qui demande l'égalité des sélecteurs composés des deux objets).

On notera l'équivalence avec le symbole « ≡ »

Ainsi A≡B si et seulement si : is - pred (A) = is - pred (B) = Vrai

Expressions conditionnelles :

Les prédicats nous apportent une aide pour le test des relations entre les objets en composant leurs structures ou en testant un objet par rapport à un modèle prédéterminé. Les expressions conditionnelles (4)

fournissent un moyen pour faire dépendre le choix d'une séquence d'actions, du résultat d'une comparaison.

Dans le langage de définition, cet outil est représenté par des expressions de la forme : (p1 → e1, p2 → e2, ... pn → en) où pi est un prédicat et ei une expression qui définit l'action à entreprendre dans le cas où le prédicat pi est vrai.

La valeur de l'expression conditionnelle est celle de ej tel que : pj soit vrai et les pi ($\forall i<j$), faux.

Avec cette nomenclature, on peut construire des définitions de certaines fonctions comme celles données ici à titre d'exemple :

Fonction logique "OU"

$$(P \vee Q) = (P=1 \rightarrow 1, T \rightarrow 1)$$

T étant un prédicat toujours vrai

Fonction logique "ET"

$$(P \wedge Q) = (\neg P=1 \rightarrow 0, T \rightarrow Q)$$

Fonction "NON"

$$(\neg P) = (P=1 \rightarrow 0, T \rightarrow 1)$$

Dans un système VDL de définition de processeurs, ces expressions conditionnelles sont principalement utilisées pour effectuer le choix de séquences d'actions ou de groupes d'instructions.

Jusqu'ici nous avons présenté quelques éléments de base qui font partie de la sémantique du langage VDL. Dans la section suivante, nous présentons la structure d'un système de définition de processeurs basée sur ce langage.

I-2-2. Structure du système de définition

Pour décrire l'exécution d'un processus, que ce soit un programme dans un certain langage, un algorithme ou une machine, on se base sur le concept des états successifs et de la fonction de transition d'état, ce qui

est aussi la base de la théorie des automates ou machines séquentielles.

Dans la section I-2-1. on a défini les aspects sémantiques liés aux fonctions ou objets du langage VDL ; pour créer maintenant un système de définition on est amené à définir une machine abstraite dans laquelle ces fonctions sont les opérations et où les objets sont des structures, le plus souvent contenus dans une mémoire. Une telle machine contient dans sa mémoire ou Etat les données à manipuler et les définitions des instructions exprimées avec les opérations de base de la machine.

Dans le domaine des machines actuelles on peut considérer les instructions comme des procédures et l'utilisation des instructions comme l'"appel" de ces procédures.

L'état de la machine est défini comme un objet dont les composantes immédiates sont : l'ensemble des données, les instructions à exécuter (le programme) et les définitions de ces instructions.

Il n'existe pas d'état unique de la machine, mais un ensemble d'états et chacun d'entre eux est développé à partir de l'état précédent grâce à la fonction de Transition d'état (FTE). Cette FTE est décrite comme l'exécution d'une seule instruction ou même comme l'exécution de chaque cycle d'une instruction ; en conséquence les changements d'un état sont définis à la fin de chaque étape.

Pour pouvoir garantir la validité d'un programme ou algorithme, l'exécution doit conduire dans un état terminal reconnaissable. Il est donc nécessaire de définir un état initial et un état final reconnaissables (ces deux états sont définis explicitement dans la représentation de l'état de la Machine comme un objet complet).

En plus de l'état de la machine qui inclut les données à manipuler, les instructions à exécuter et les définitions de ces instructions, la machine doit comprendre le moyen d'interpréter chaque instruction, ce qui

équivaut à évaluer la fonction de Transition d'état. Cette partie de la machine est analogue à l'unité de commande d'un processeur en ce qui concerne l'interprétation et l'exécution des instructions, et elle peut être exprimée en termes des primitives du système de définition et, en conséquence, n'est pas séparée du système.

I-2-2-1. L'état de la machine

Pour faciliter la description, l'état de la machine est défini comme étant un objet composé qui contient les trois composantes immédiates suivantes :

- Représentation des données (D)
- Définition des instructions (DI)
- Objet de commande (C)

Pour permettre aux instructions qui affectent le cheminement de l'exécution de le faire sans avoir été définies complètement, avant l'exécution d'un programme par exemple, deux propriétés fondamentales des instructions sont spécifiées :

- Les définitions des instructions peuvent contenir des expressions conditionnelles (conditions de Mc Carthy) qui détermineront la partie d'une définition qui doit être interprétée.

- Les instructions peuvent être définies de façon à être interprétables de deux manières :

a) Macros : où la définition consiste en un ensemble d'instructions qui doivent remplacer celle qui est en train d'être interprétée dans la partie de commande ;

b) Instructions d'assignation : Ce sont des instructions qui affectent la partie "Données" de l'état et qui fournissent des valeurs aux listes d'arguments d'autres instructions qui doivent être exécutées dans des

états ultérieurs.

La partie de commande :

Cette partie de l'état de la machine abstraite est un objet en forme d'arbre contenant les objets composés représentant les instructions candidates à l'exécution.

L'arbre de commande contient initialement une seule instruction qui doit être prédéfinie comme faisant partie de l'état initial. A partir de l'exécution de celle-ci, d'autres instructions vont s'incorporer à l'arbre de contrôle en remplaçant celles qui sont exécutées. La définition sous forme d'objets composés de ces instructions est prise dans la partie "Définition des instructions" de l'état de la machine, les arguments de ces instructions étant actualisés, selon les opérations réalisées au cours de l'exécution des instructions précédentes, avant d'être incorporées à l'arbre de commande.

Lorsque l'arbre de commande n'a plus d'instructions à exécuter on dit que la machine a terminé son exécution.

La partie "Définition des Instructions" :

La définition d'une instruction comporte une déclaration qui définit le nom de l'instruction, un ensemble de paramètres et une expression conditionnelle qui définit la partie de l'objet qui est effective.

Nous avons déjà dit que les instructions sont de deux types : d'assignation et macros.

Les instructions d'assignation effectuent essentiellement des opérations de mutation sur l'arbre de données et aussi rendent possible le passage de certaines valeurs calculées aux listes de paramètres d'autres instructions.

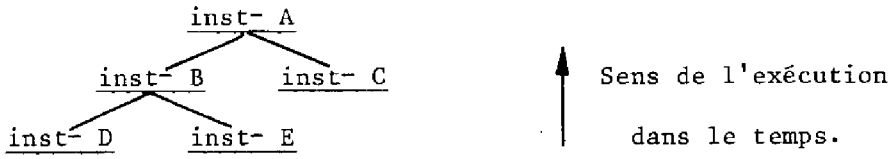
Les instructions dites "macros" regroupent d'autres instructions ; ceci peut être illustré de la façon suivante :

- Soient trois instructions A B et C qui doivent être interprétées dans cet ordre : A, B, C ; ceci est représenté par :

Macro = inst- C ;
inst- B ;
inst- A

Le caractère (;) indique la fin d'un niveau.

- Soit une autre instruction du type macro qui contient le groupe suivant où l'ordre d'exécution est indiqué dans la figure suivante :



qui montre l'existence de parallélisme entre l'exécution des instructions D et E ainsi qu'entre B et C.

Dans la description VDL ceci est représenté par :

Macro = inst- A ;
inst- C,
inst- B ;
inst- D,
inst- E

Etapes d'un interpréteur du système de définition :

Un interpréteur qui serait capable de manipuler l'arbre représentant l'état du système comprend les étapes suivantes :

a) Sélection dans la partie de commande de l'état de l'instruction à exécuter (feuille). S'il y a plus d'une candidate, le choix de l'une d'entre elles est arbitraire. La sélection s'effectue par l'intermédiaire d'un sélecteur composé.

b) Ayant déterminé le sélecteur composé de l'instruction à exécuter, le nom de celle-ci est utilisé pour retrouver sa définition (comme objet composé) dans la partie de l'état dite "Définition des Instructions".

c) Evaluation des expressions conditionnelles de cette instruction pour choisir un groupe et l'on fait une copie de ce groupe.

- Si ce groupe est un groupe "d'assignation" :

d) Evaluation des expressions dans la copie du groupe faite en c).

e) Passage de la valeur à retourner au groupe d'arguments d'une autre instruction.

f) Réalisation des opérations de mutation sur l'arbre de données d'après les calculs effectués.

g) Effacement de l'instruction en cours d'exécution, de l'arbre de commande.

- S'il s'agit d'un groupe "macro" :

d) Augmentation ou actualisation, dans la copie du groupe, des sélecteurs composés de chaque instruction avec le sélecteur composé utilisé pour la sélection de l'instruction en cours d'exécution.

e) Evaluation des arguments de toutes les instructions du groupe macro avec les arguments actuels de l'instruction en cours d'exécution.

f) Remplacement de l'instruction en exécution dans l'arbre de commande et la copie du groupe créé dans les étapes précédantes.

Un interpréteur qui réalise les opérations énumérées ci-dessus, est une entreprise difficile à implémenter dans la pratique. Cependant sa structure est utile pour la description des processeurs, c'est-à-dire que le processeur est décrit en langage VDL de manière telle que la description puisse être interprétée par un interpréteur comme celui qui a été décrit ici.

Un exemple simple d'une telle description par la S- Machine dû à Birman est présenté en I-3-2.

L'intérêt principal d'une telle description et interprétation réside dans la possibilité de décrire aisément des opérations "parallèles" ainsi que dans le degré de formalisme atteint, tout en partant d'un ensemble d'axiomes assez simples.

Il apparaît qu'une présentation aussi rapide du langage VDL ne peut être suffisante pour en présenter toutes ses caractéristiques.

Pour avoir une connaissance plus complète du langage et système de définition VDL, le lecteur se reportera à l'ouvrage COMPUTER SEMANTICS de J. LEE (14).

Jusqu'ici, nous avons présenté la forme interne de la machine de définition qui consiste explicitement en l'Etat de la machine et implicitement en la fonction de Transition d'état. Cette machine ou système de définition constitue un support très complet pour développer des définitions formelles de processeurs, d'algorithmes ou de langages de programmation.

Dans le domaine de la description des processeurs, un tel système de définition pourrait être utile pour la conception de nouvelles structures logiques avec les liaisons conceptuelles suivantes :

- Les instructions du système de définition représentent la circuiterie logique (chemin des données) ;

- Les instructions du type "assignation" représentent les transferts entre registres ;

- Les instructions du type "macro" représentent l'exécution des microinstructions ;

- Les arguments des instructions représentent les contenus des registres ;

- Les paramètres des définitions représentent les registres.

Le côté négatif est la difficulté d'implémentation d'un interpréteur pour le système de définition qui permet de valider de manière automatique le modèle conçu.

Dans la section suivante, nous allons présenter, afin d'illustrer nos propos, la S- Machine d'une manière informelle, d'abord, et par la suite en langage VDL.

I-3. LA S-MACHINE

I-3-1. Présentation de la S- MACHINE

La S-Machine est un calculateur didactique conçu par C.W. GEAR (7) dans le but d'être utilisé comme support pour un exemple de fonctionnement d'un processeur microprogrammé.

Cette machine comporte, en particulier, une pile constituée par un nombre variable de mots de 32 bits où seul le mot du début de pile, qui se trouve au sommet, est accessible et son adresse est repérée par un pointeur. Si l'on ajoute un nouveau mot dans la pile, celui-ci occupe le mot précédent, et le pointeur se voit incrémenté d'une unité ; si l'on supprime un mot de la pile (celui qui est en sommet) le nouveau sommet est occupé par le mot suivant et le pointeur est décrémenté d'une unité. Cette pile se trouve, dans cette structure, située à l'intérieur de la mémoire principale dont elle constitue une partie de capacité variable.

Les données de base de la S- Machine sont des mots de 32 bits. La mémoire principale contient 2^{24} mots.

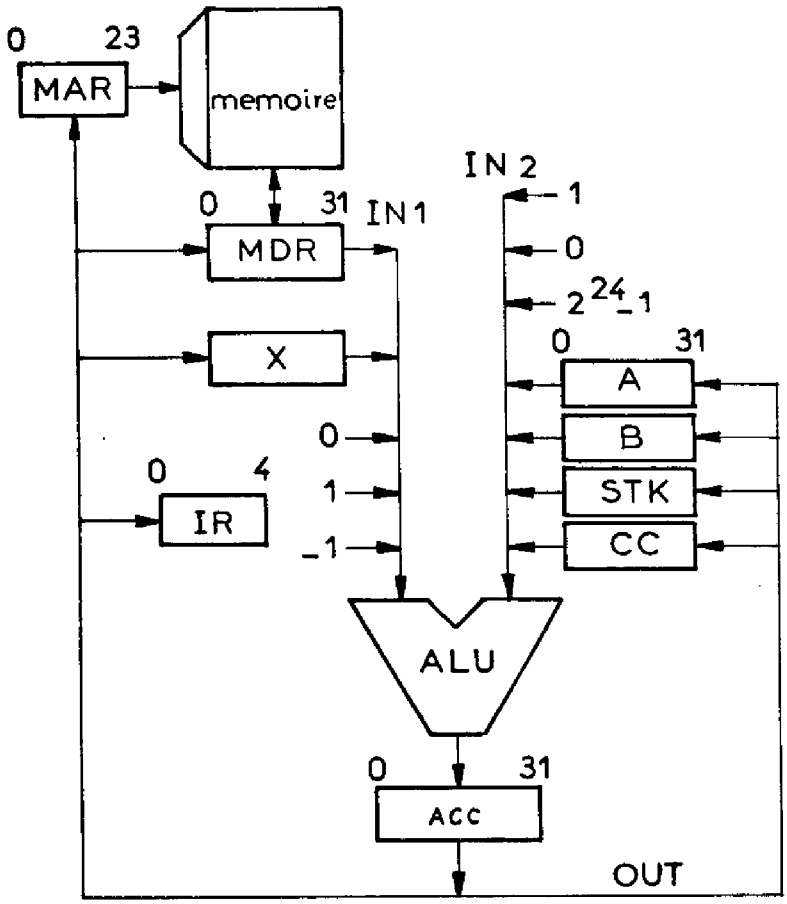


Figure I-3

La figure I-3. montre le chemin des données de la S- Machine qui comporte les registres suivants :

MAR : Registre d'adresse Mémoire ; 24 bits

MDR : Registre de données Mémoire ; 32 bits

X : Registre d'index ; 32 bits

A, B: Registres de travail ; 32 bits chacun

STK : Pointeur de pile ; 32 bits

ACC : Registre qui retient le résultat de l'UAL ; 32 bits

CC : Compteur ordinal d'instruction ; 32 bits

IR : Registre de mémorisation du code opération ; 5 bits

Les instructions de la S- Machine sont codées sur un seul mot de 32 bits selon le format représenté sur la figure I-4.

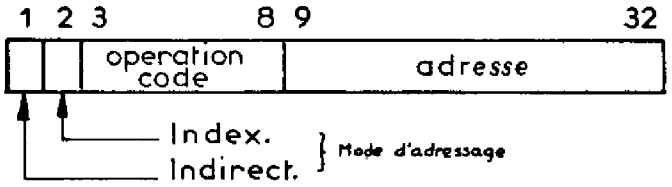


Figure I-4

Certaines instructions effectuent des changements du contenu de la pile dont le bas est le dernier mot de la mémoire principale, c'est-à-dire celui d'adresse $2^{24}-1$.

L'enregistrement d'un nouveau mot dans la pile est fait en le chargant dans un mot mémoire d'adresse indiquée par le contenu du registre STK décrémenté au préalable d'une unité. L'opération inverse, qui consiste à supprimer un mot de la pile est réalisée virtuellement en incrémentant d'une unité le contenu du registre STK.

Les instructions de la S- Machine sont de deux types :

- Celles qui manipulent la pile et qui n'utilisent pas la partie adresse du mot instruction. Elles sont appelées "instructions à zéro adresse"

- Celles qui utilisent la partie adresse du mot instruction et qui sont appelées "instructions à une adresse".

Dans ce type d'instruction, le 1er et 2ème bit du mot permettent de spécifier le mode d'adressage, direct ou indirect, indexé ou non (d'après la valeur 1 ou 0 des bits correspondants).

Une description sommaire et informelle des actions de ces instructions est donnée dans le tableau suivant.

INSTRUCTIONS A UNE ADRESSE

CODE OPE	MNEMONIQUE	STK	ACTION
0	LOAD	- 1	Chargement de la pile avec le mot mémoire d'adresse donnée.
1	LDI	- 1	Chargement immédiat de la pile avec la partie adresse.
2	STORE	+ 1	Enregistrement du mot en sommet de la pile dans un mot mémoire d'adresse donnée.
3	TRA	0	Branchement inconditionnel à adresse donnée.
4	TPL	0	Branchement à adresse si sommet de pile > 0.
5	TMI	0	Branchement à adresse si sommet de pile < 0.
6	TZE	0	Branchement à adresse si sommet de pile $= 0$.
7	TNZ	0	Branchement à adresse si sommet de pile $\neq 0$.
8	ENTER	- 1	Branchement à sous-programme en plaçant la valeur actuelle de CC dans la pile.
9	LDX	0	Chargement du registre X avec le mot mémoire M (adresse).
10	LDXI	0	Chargement immédiat de X avec la partie adresse de l'instruction.
11	LOOP	0	Incrémentation d'une unité de X et branchement à adresse si $X \neq 0$.
12	LS	0	Décalage à gauche de N positions, N étant la valeur adresse.
13	RS	0	Décalage à droite de N positions, N étant la valeur adresse.

La colonne correspondant à STK représente les changements effectués sur la pile, + 1 représente la suppression d'un niveau, - 1 l'incrémentation et 0 indique que la pile reste inchangée.

INSTRUCTIONS A ZERO ADRESSE

CODE OPE	MNEMONIQUE	STK	ACTION
32	ADD	+ 1	Addition des deux premiers mots de la pile (résultat au 2ème).
33	SUB	+ 1	Soustraction des deux premiers mots de la pile (résultat au 2ème).
34	AND	+ 1	ET logique entre les deux premiers mots de la pile (résultat au 2ème).
35	OR	+ 1	OU logique entre les deux premiers mots de la pile (résultat au 2ème).
36	EOR	+ 1	OU exclusif entre les deux premiers mots de la pile (résultat au 2ème).
37	NOT	0	Complémentation du mot en sommet de la pile.
38	XTS	- 1	Chargement du contenu de X en sommet de la pile.
39	STX	+ 1	Chargement dans X du mot en sommet de la pile.
40	ADX	+ 1	Addition du contenu du mot en sommet de pile au contenu du registre X.
41	SBX	+ 1	Soustraction du contenu du mot en sommet de la pile au contenu de X.
42	RET	+ 1	Branchement à l'adresse indiquée par le mot en sommet de la pile.
43	POP	+ 1	Suppression du premier mot de la pile.
44	STOP	0	Arrêt de l'exécution, SW ← 0

La manière d'opérer de la S- Machine peut être suivie sur la figure I-3., où l'on voit que le bus IN-1 fournit l'opérande de la partie gauche de l'UAL et IN-2 celui de la partie droite. Ces deux bus sont d'une largeur de 32 bits.

Dans l'UAL, on peut réaliser sur ces deux opérandes, une des huit opérations suivantes : +, -, $\wedge$, V, $\oplus$, $\sim$, LS et RS.

Le résultat est envoyé par l'intermédiaire du bus OUT sur un des huit registres montrés dans la figure.

Les opérations sur la mémoire (READ/WRITE) sont commandées par la

micromachine. Dans le cas READ, un mot adressé par le contenu du registre MAR est placé dans le registre MDR. La commande WRITE stocke le contenu du registre MDR à l'adresse mémoire indiquée par le contenu de MAR.

Le séquencement des opérations est déterminé par l'unité de commande de la machine, qui est microprogrammée. Le microprogramme est placé dans une mémoire différente de la mémoire principale appelée CS (de : Control Store) dans la figure I-5. qui montre schématiquement l'unité de commande.

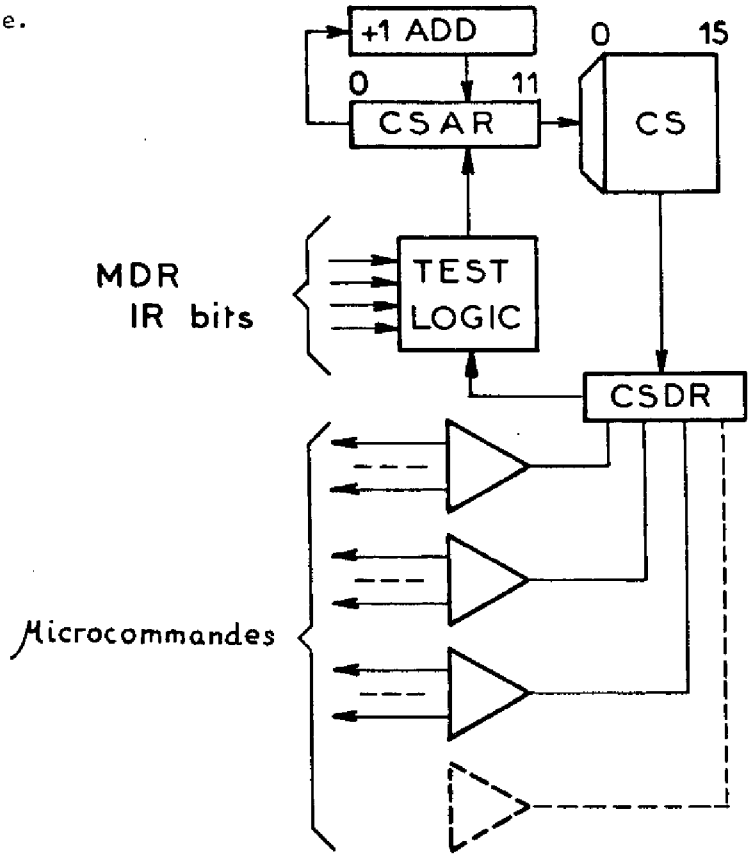


Figure I-5

Le fonctionnement du séquenceur (ou micromachine), représenté dans la figure I-5., consiste, d'abord à lire dans CS un mot et à l'enregistrer dans CSDR (Control Store Data Register). Ce mot représentant une microinstruction du microprogramme est, ensuite, décodé d'après les différents champs et les sorties des décodeurs sont connectées aux portes du chemin des données.

Il existe deux types de microinstruction dont les formats sont représentés par la figure I-6.

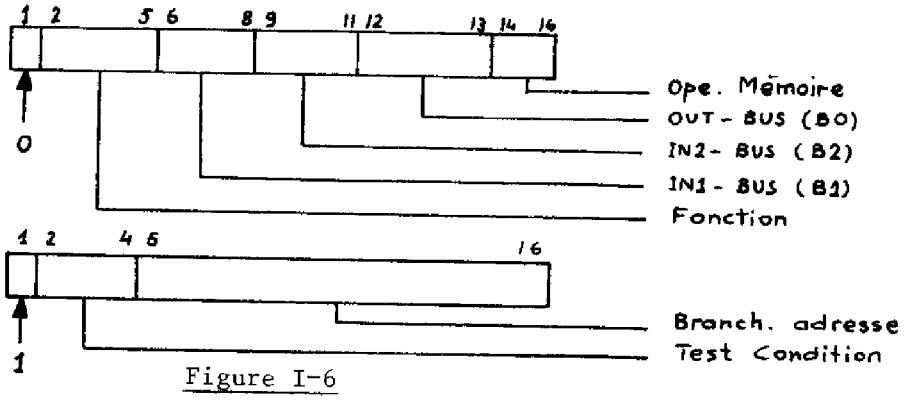


Figure I-6

Ces deux types de microinstruction sont différenciés par le premier bit du mot de 16 bits. Si ce premier bit est un "zéro" il s'agit d'une microinstruction qui comporte les codes correspondant aux commandes pour les portes du chemin de données, pour déterminer le type d'opération de l'UAL et pour définir les opérations sur la mémoire principale. Si par contre le premier bit est un "un", la microinstruction détermine le test à réaliser sur les contenus de MDR ou IR et comporte une adresse pour effectuer des branchements à l'intérieur du microprogramme, selon les résultats des tests.

Un exemple de microinstruction du premier type est la suivante, représentée en mode mnémonique :

1 ADDM , ZERO , CC , MAR , R

L'interprétation de cette microinstruction par le séquenceur est :

- Chargement dans le bus IN-1 de la valeur "zéro" (vecteur de 32 zéros) et dans le bus IN-2 chargement du contenu de CC. L'UAL reçoit l'ordre d'effectuer une addition entre les deux opérandes qui se trouvent sur IN-1 et IN-2. Le résultat de l'opération est envoyé sur le registre MAR et on commande une

lecture dans la mémoire principale. En conséquence, à la fin du cycle, on obtient dans MDR le mot qui se trouvait à l'adresse indiquée par le contenu du registre CC. Le chiffre 1 devant la microinstruction indique l'adresse de la mémoire du microprogramme où elle est placée.

Un exemple de microinstruction du type "test et branchement" est le suivant :

7 T1 , 9

(indique l'adresse de cette microinstruction dans CS)

dont l'interprétation faite par le séquenceur est :

- Si le deuxième bit du registre MDR est un zéro, la microinstruction qui sera exécutée à la suite de celle-ci est celle qui se trouve à l'adresse 9 du microprogramme, sinon on continuera en séquence (c'est-à-dire que l'on traitera, au cycle suivant, la microinstruction qui se trouve à l'adresse 8).

Le tableau suivant montre les interprétations des codes mnémoniques des différentes microinstructions :

MICROINSTRUCTIONS DE TEST ET BRANCHEMENT

MNEMONIQUE	ACTION
TO	Branchement si le premier bit de MDR est à la valeur zéro.
T1	Branchement si le second bit de MDR est à la valeur zéro.
T2	Branchement si le troisième bit de MDR est à la valeur zéro.
TMDR	Branchement si tous les bits de MDR sont à la valeur zéro.
TI	Branchement à une adresse calculée à partir de IR.
TRM	Branchement inconditionnel.

MICROINSTRUCTIONS DE COMMANDE SUR LE CHEMIN DES DONNEES

MNEMONIQUE	ACTION
ADDM	(IN-1) + (IN-2)
SUBM	(IN-1) - (IN-2)
ANDM	(IN-1) $\wedge$ (IN-2)
ORM	(IN-1) $\vee$ (IN-2)
EORM	(IN-1) $\oplus$ (IN-2)
NOTM	$\sim$(IN-1 + IN-2)
LSM	Décalage d'un bit à gauche de (IN-1 + IN-2)
RSM	Décalage d'un bit à droite de (IN-1 + IN-2)

Le microprogramme, écrit sous forme mnémonique, de la S-MACHINE est donné en annexe.

Nous ne montrerons pas dans ce mémoire les différents codes binaires correspondant aux champs des microinstructions puisque ces renseignements ne sont d'aucune utilité pour améliorer la description de l'ensemble.

Durant ces dernières pages nous avons décrit la S-Machine d'une manière sommaire et imprécise qui n'est pas suffisante pour une bonne compréhension de la machine. Il est clair que pour aboutir à une description complète du fonctionnement et de la structure de la S-Machine, il faudrait encore remplir plusieurs pages et même encore, on trouverait des ambiguïtés inhérentes au langage ce qui rendrait plus difficile une description correcte.

D'autre part une description faite de cette manière se prête difficilement à une étude de vérification de l'implémentation du microprogramme dans le système. Nous allons montrer par la suite deux manières de décrire la S- Machine qui permettent, grâce à la précision et au degré de formalisme qu'elles présentent, de réaliser une étude de vérification sur les descriptions des différents niveaux de la machine.

I-3-2. Définition VDL de la S-Machine

La définition VDL de la S- Machine que nous présentons dans cette section et en annexe, a été réalisée par BIRMAN (1) en se basant sur le système de définition présenté dans le 1er chapitre.

Cette définition a été étendue par l'utilisation de plusieurs opérateurs du langage APL (10). L'utilisation de ces opérateurs APL pour spécifier l'action de certaines fonctions est une aide très intéressante pour la définition de la Machine ; ceci est dû au fait que le langage APL a la particularité de pouvoir traiter des opérandes de dimension variable, c'est-à-dire, des scalaires, vecteurs, matrices ou tableaux, avec une grande souplesse.

La description de la S- Machine avec ce langage VDL/APL a été faite pour deux niveaux différents. Le premier, que nous appelerons niveau "spécification" est constitué par les éléments de la S- Machine qui sont "accessibles" à l'utilisateur du système et en constituent les primitives de base. Ces éléments sont les suivants :

- Mémoire Principale
- STK, Registre pointeur de pile
- X, Registre d'index
- CC, Registre compteur ordinal
- SW, Interrupteur de marche/arrêt

Dans la description de ce niveau, les actions de chaque instruction sont définies en fonction de ces éléments, sans tenir compte de la structure réelle du système.

Le deuxième niveau, que nous appelons niveau "réalisation microprogrammée", contient la plupart des éléments du système réel nécessaires pour mettre à terme l'exécution des instructions selon le microprogramme,

à savoir :

- Mémoire principale
- Mémoire de microprogramme
- STK
- X
- CC
- SW
- MDR (Registre de données, mémoire principale)
- MAR (Registre d'adresse, mémoire principale)
- A, B } (Registres de travail)
- IR (Registre pour le code opération de l'instruction
- CSAR (Registre d'adresse, mémoire de microprogramme)
- CSDR (Registre de données, mémoire de microprogramme)

Ces définitions VDL/APL de la S- Machine ont été énoncées sous la forme de machines abstraites. Ceci signifie que l'on réduit la structure réelle à une autre où l'on fait abstraction de plusieurs facteurs inhérents au fonctionnement réel de la machine. Ainsi il est plus facile d'étudier la machine d'une manière globale (surtout les aspects logiciels du système) sans avoir à se préoccuper des problèmes technologiques qui doivent être traités à part.

Les machines abstraites définies pour décrire les deux niveaux comportent un "état" et un objet VDL avec les composantes suivantes :

1) Partie de commande
2) Macrobibliothèque
3) Données

La partie de commande est représentée par un arbre et est constituée à partir d'instructions VDL. Les feuilles de cet arbre sont candidates à l'exécution pour le cycle suivant.

Les instructions que l'on rencontre sont de deux sortes :

- Macroinstructions, dont le traitement consiste à remplacer la "feuille" où elles se trouvent par un sous-arbre d'instructions.

- Instructions élémentaires, dont l'exécution produit des assignations pour certains paramètres et retourne les nouvelles valeurs à l'arbre de commande.

Le niveau "spécification" est décrit par la machine abstraite S et le niveau "réalisation microprogrammée" par la machine abstraite MS.

La définition de S qui est donnée en ANNEXE est constituée de trois parties :

a) Prédicat is - S

b) Etat initial

c) Macrobibliothèque

Le prédicat is - S identifie les éléments de la S- Machine qui sont connus au niveau "spécification" c'est-à-dire : Mémoire principale, registres STK, X, CC et bit sw.

L'état initial spécifie la valeur de STK et de CC ainsi que l'arbre de commande "S -Control (S)" (les autres éléments ont des valeurs quelconques).

La macrobibliothèque contient les définitions des instructions VDL. La macrodéfinition la plus importante dans S est "exec-pgm" :

exec-pgm =

is-run (S) → exec-pgm

exec-inst (i)

i : fetch-inst

else → Ω

Cette macro s'appelle elle-même (récursivité) jusqu'à ce que le prédicat is-run prenne la valeur zéro (prédicat non vérifié), ce qui arrive quand SW est mis à zéro. Chaque itération de cette macroinstruction exécute une instruction machine. Le cycle de fonctionnement comporte deux parties

qui sont les suivantes :

- Recherche d'instruction, réalisée par la macro "fetch-inst"
- Exécution d'instruction, réalisée par la macro "exec-inst"

Dans la description de la machine abstraite μS correspondant au niveau "réalisation microprogrammée" les points essentiels sont les mêmes que ceux développés ci-dessus pour la description de S, mais il faut tenir compte que le niveau μS traite l'exécution du microprogramme. Celui-ci est aussi donné en ANNEXE.

Nous présentons ici le déroulement de l'interprétation d'une instruction de la S- Machine. Cette interprétation est celle correspondant à l'instruction LOAD dans la partie S (Définition du niveau "spécification") (voir en ANNEXE).

A l'état initial l'arbre de commande de l'Etat de la machine est constitué par la macro : exec-pgm, c'est-à-dire, S- Control (S) : exec-pgm

L'expression conditionnelle de l'instruction exec-pgm est la suivante :

(is-run (S) → exec-pgm , T → Ω)

exec-inst (i) ← (Prédicat toujours vrai)

i : fetch-inst

Si le prédicat is-run (S) est vérifié (ceci signifie que SW (S) = 1) le groupe des trois instructions : exec-pgm

exec-inst (i)

i : fetch-inst

se substitue à celle qui était présente à l'état initial (exec-pgm). D'après la représentation du groupe, l'instruction fetch-inst doit être interprétée en premier lieu. La définition de cette instruction est cherchée dans la partie définition des instructions ou bibliothèque (section 2 dans Macro Library).

On y trouve la structure suivante :

fetch-inst =

PASS : build-inst (a)

a : fetch-word (CC)

En conséquence dans l'arbre de commande l'instruction fetch-inst en cours d'exécution est substituée par le groupe montré ci-dessus, et la nouvelle "feuille" à exécuter est a : fetch-word (CC).

Dans la section 3 de la bibliothèque on trouve que

fetch-word (t) =

PASS : mem [m ;]

m : 2 ⊥ t [8 + 24]

Il s'agit donc, d'une instruction d'assignation où t représente le contenu binaire du registre compter ordinal CC. L'exécution de cette instruction affectera au paramètre (a) un vecteur binaire qui a été cherché à l'adresse (m) de la mémoire.

La définition de l'instruction suivante est trouvée dans la section 4 de la bibliothèque, elle est de la forme suivante :

build-inst (t) =

PASS : MO (< S-id : id >, < S - ix : ix >,

< S-op : op >, < S - ad : ad >)

id : t [0]

ix : t [1]

op : t [2 + 26]

ad : t [8 + 24]

Dans ce groupe d'assignation t représente le vecteur binaire (a) qui a été calculé dans la partie fetch-word. L'opération réalisée est la construction d'un objet composé dont les objets élémentaires sont:(id)

correspondant au bit d'indirection (pour l'adressage), (ix) pour l'indexation, (op) pour le code opération et (ad) pour l'adresse de l'instruction de la S- Machine représentée par le vecteur (a). La ligne PASS indique que cet objet composé ainsi construit est "passé" à d'autres niveaux ; ici cet objet est passé avec la notation (i) à l'instruction exec-inst (i) qui est interprétée une fois que le niveau correspondant à i : fetch-inst a été exécuté.

La définition d'exec-inst (i) est la suivante (section 5 de bibliothèque) :

exec-inst (i) =

is- load (i) → ...

is- ldi (i) → ...

is- store (i) → ...

⋮

is- stop (i) → ...

Cette instruction contient une expression conditionnelle où un seul des prédicats est vérifié pour chaque instruction de la S- Machine, I représentée maintenant par l'objet composé (i)

La forme de ces prédicats est la suivante :

is- load (i) =

PASS : 0 = 2⊥S - op (i)

is- ldi (i) =

PASS : 1 = 2⊥S - op (i)

•
•
•

is- stop (i) =

PASS : 44 = 2⊥S - op (i)

Ce qui montre que le prédicat vérifié est celui qui correspond à l'instruction de la S- Machine indiquée par le décodage de (op). Dans notre exemple le prédicat qui sera vérifié est : is- load (i); en conséquence, le groupe choisi est le suivant :

load - stk (a)

a : fetch-word (b)

b : calc - addr (i)

adv - ctr

Selon cette représentation les instructions load-stk (a) et adv-ctr peuvent être interprétées indifféremment l'une après l'autre.

La première instruction à exécuter dans ce groupe est b : calc-addr (i). Sa définition est la suivante (section 13) :

calc-addr (i) = calc-id (i,a)

a : calc-ix (i)

La définition de calc-ix (i) (section 14) est :

calc-ix (i) =

(1 = S- ix (i)) → PASS : (32↑2) ⊤ (2*32)|b + c

b : 2 ⊥ x

c : 2 ⊥ S- ad (i)

else → PASS : 8↑0, S- ad (i)

On voit dans cette définition que (a) recevra le résultat de l'addition du contenu du registre d'index X à la partie adresse de l'instruction si le bit d'indexation est à la valeur "1". Sinon, (a) recevra la valeur de la partie adresse de l'instruction (représentée par l'objet composé (i)).

La définition de calc- id (i,a) est la suivante (section 15)

calc - id (i, a) =

(1 = S - id (i)) → PASS : 8⍱0, b[8 + ⍵24]

b : fetch-word (a)

else → PASS : a

Cette représentation montre que si le prédicat 1 = S - id (i) est vérifié, l'argument (a) recevra un vecteur binaire trouvé à l'adresse (a) de la mémoire et dont les huit premiers bits sont à zéro. Si ce prédicat n'est pas vérifié le paramètre a reste inchangé.

Ceci étant fait, l'arbre de commande doît exécuter l'instruction a : fetch-word (b) où (b) a reçu la valeur de (a) calculée dans l'instruction calc-add. L'instruction fetch-word a déjà été montrée précédemment et son action est donc de passer sur le paramètre (a) de load-stk (a) un vecteur binaire qui se trouve à l'adresse (b) de la mémoire.

La définition de load-stk (a) (section 9) est :

load - stk (a) = store - word (a, stk)

push - stk

où pus - stk =

stk : (32⍱2) ⍑ (2 ⍟ 32)⍳¯1 + 2 ⊥ stk

store - word (a,t) =

mem [2 ⊥ t [8 + ⍵24] ;] : a

En conséquence l'action de load-stk (a) est d'enregistrer dans la mémoire, à l'adresse donnée par la valeur décimale de STK décrémenté d'une unité, la valeur (a) calculée par l'instruction a : fetch-word (b). Enfin, la définition de l'instruction adv-ctr = cc : (32⍱2)⍑(2⍟32)⍳1 + 2⊥cc indique que le compteur ordinal est incrémenté d'une unité.

Ce déroulement de l'interprétation d'une instruction de la S- Machine dans le système de définition VDL montre la grande quantité d'opérations nécessaires pour sa réalisation. Un interpréteur symbolique qui effectue cette tache d'une manière automatique pourrait justifier un tel système de définition ; cependant nous estimons que pour une procédure "manuelle", ce système VDL de définition est mal adapté à une procédure de vérification, ce qui est le but de notre étude.

Pour l'étude de la vérification, il faut dérouler l'interprétation de chaque instruction dans chacun des deux niveaux de description de la machine (niveau "spécification" et niveau réalisation microprogrammée) en effectuant une comparaison de ces interprétations. La tâche est donc extrêmement longue.

Cette difficulté nous a amené, à partir de cette première phase de nos travaux à envisager une autre manière de décrire la S- Machine mieux adaptée à une procédure de vérification manuelle. Nous la présentons dans le paragraphe suivant.

I-4. FORME D'ORGANIGRAMME POUR LA DESCRIPTION D'UN PROCESSEUR:S-MACHINE

Une autre manière de définir un processeur avec un degré de formalisme suffisant pour éviter les ambiguïtés et les imprécisions du langage ordinaire est la description sous forme d'organigramme. Pour la description de processeurs à exécution "verticale" (par opposition à horizontale ou parallèle), la forme d'organigramme présente une grande simplicité et une bonne précision si l'on utilise les opérateurs d'un langage précis et puissant pour exprimer les opérations à effectuer sur le contenu des registres binaires. Nous avons déjà vu que dans le cadre de la définition VDL, le langage APL apportait une aide importante en spécifiant d'une manière concise et précise les fonctions du système. Dans les organigrammes, nous utilisons ces opérateurs APL pour définir les

différentes opérations sur les registres.

La figure I-7. montre l'organigramme général qui décrit le niveau "spécification" de la S- Machine. L'organigramme de la figure I-7 est une représentation sous forme d'organigramme de la description VDL de S présentée en ANNEXE.

Dans la figure I-7, m représente un registre de 32 bits qui reçoit le mot qui se trouve dans la mémoire principale, à l'adresse donnée par CC.

ID est une variable qui reçoit la valeur du premier bit de m (adressage indirect ou direct).

IX est une variable qui reçoit la valeur du second bit de m (adressage indexé ou non indexé).

OP est une variable qui reçoit la valeur des bits 3, 4, 5, 6, 7, 8 de m, bits correspondant au code opération de l'instruction en cours d'exécution.

AD est une variable qui reçoit les 24 derniers bits de m, c'est-à-dire la partie adresse de l'instruction. Selon que la valeur décimale de OP soit comprise entre 0 et 13 ou entre 32 et 41 il s'agit d'une instruction à une adresse ou zéro adresse, respectivement.

Les figures I-8. et I-9. montrent dans ces deux cas, les descriptions correspondant à l'exécution de chaque instruction de la S- Machine.

En général, les noms de registres notés par une lettre minuscule indiquent l'utilisation de registres non spécifiés lors de la conception fonctionnelle du système.

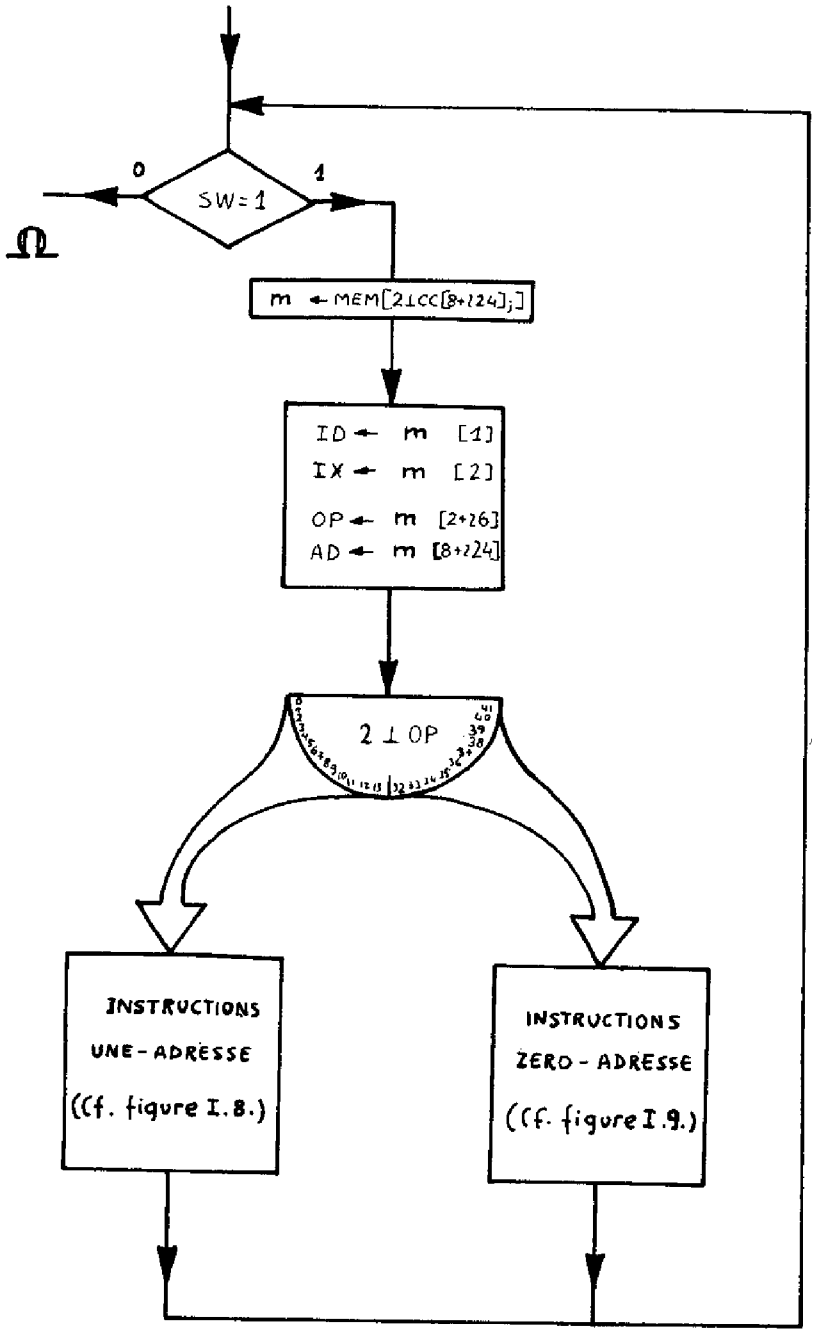

0
SW=1
1
Ω
m ← MEM[2⊥CC[8+?24];]
ID ← m [1]
IX ← m [2]
OP ← m [2+?6]
AD ← m [8+?24]
2 ⊥ OP
INSTRUCTIONS
UNE-ADRESSE
(Cf. figure I.8.)
INSTRUCTIONS
ZERO-ADRESSE
(Cf. figure I.9.)

Figure I-7

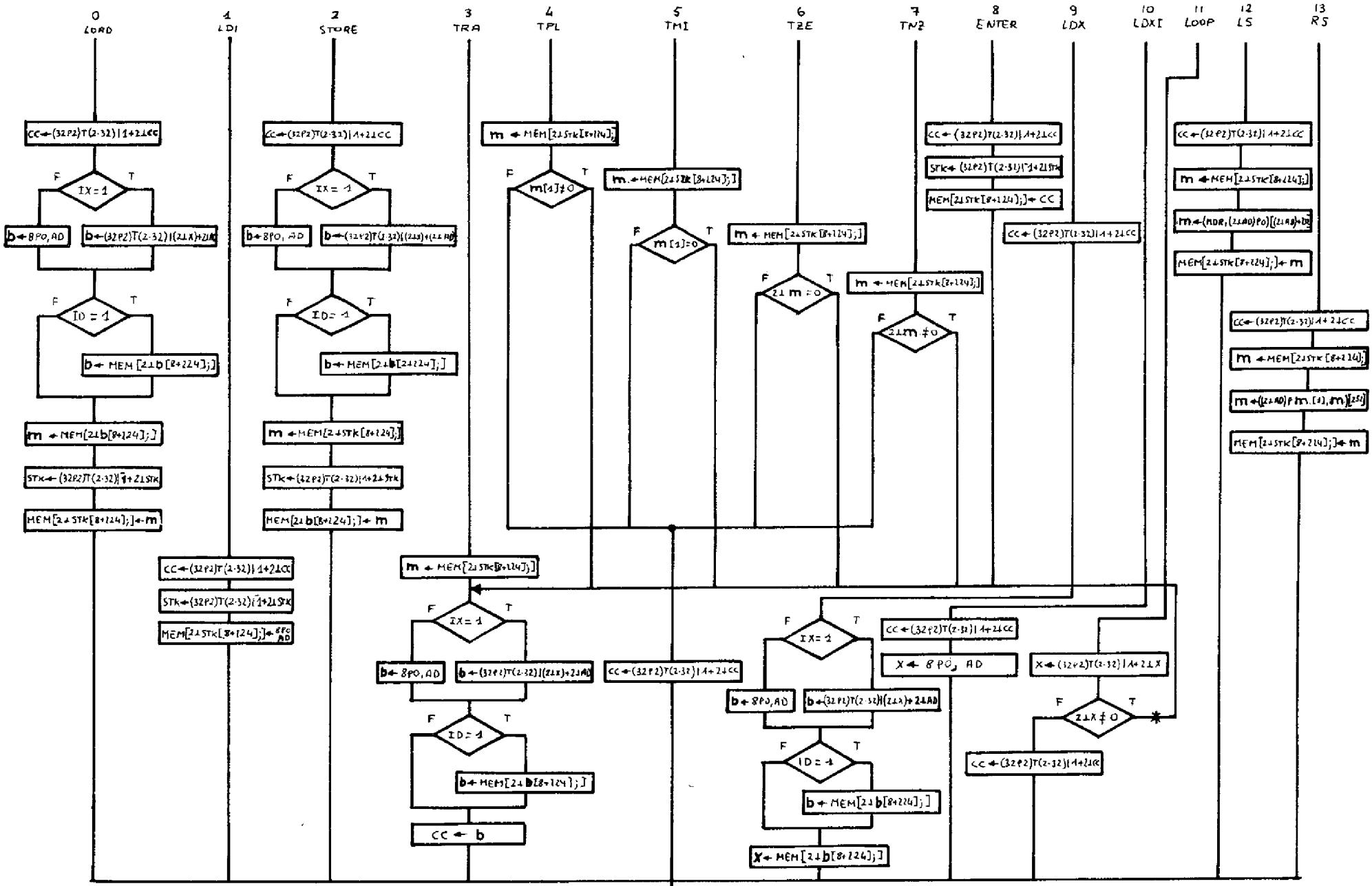

0 LOAD
1 LDI
2 STORE
3 TRA
4 TPL
5 TMI
6 TZE
7 TNZ
8 ENTER
9 LDX
10 LDXI
11 LOOP
12 LS
13 RS
IX = 1
ID = 1
m[1] ≠ 0
m[1] = 0
2⊥m = 0
2⊥m ≠ 0
2⊥X ≠ 0

Figure I - 8

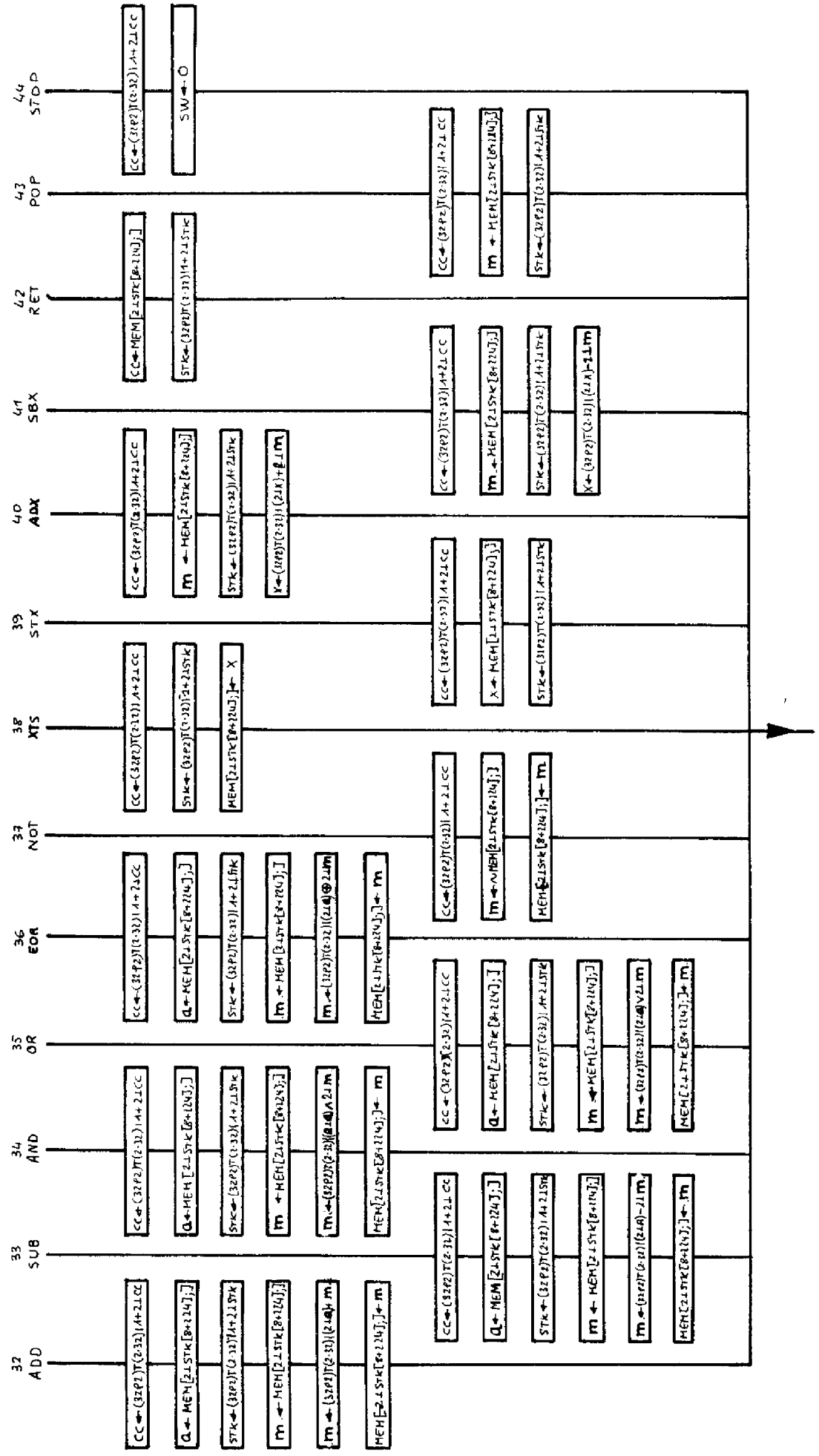

32 ADD
33 SUB
34 AND
35 OR
36 EOR
37 NOT
38 XTS
39 STX
40 ADX
41 SBX
42 RET
43 POP
44 STOP

Figure I-9

La figure I-10. représente l'organigramme général qui décrit le fonctionnement de la partie du système qui traite chacune des microinstructions, c'est-à-dire l'unité de commande ou micromachine. Cette description sous forme d'organigramme correspond à cette donnée en ANNEXE en langage VDL pour MS.

Cet organigramme montre comment pour chaque cycle machine une microinstruction est chargée dans le registre CSDR.

Si le premier bit de ce registre est un "un", il s'agit d'une microinstruction du type test et branchement à l'intérieur du microprogramme. Elle sera traitée comme l'on montre dans la figure I-12. dans laquelle, TE reçoit les bits 2, 3, 4 de CSDR, définissant le type de test qui doit être effectué sur les bits de MDR ou de IR ; AD reçoit les 12 derniers bits de la microinstruction qui précisent une adresse de la mémoire de microprogramme.

Si le premier bit de CSDR est un zéro, la microinstruction est du type commande d'opérations sur le chemin des données. Son traitement est représenté sur la figure I-11. où : FU reçoit les bits 2, 3, 4 et 5 de la microinstruction en cours d'exécution dont la valeur définit le type d'opération à exécuter par l'ALU ; B1 et B2 reçoivent les bits qui déterminent le choix des registres dont la valeur sera transférée dans les bus IN-1 et IN-2, BO détermine le registre dans lequel est stocké le résultat ; ME reçoit les deux derniers bits de la microinstruction qui définissent si l'opération à effectuer sur la mémoire principale est une lecture, une écriture ou qu'aucune opération n'est exécutée.

Chaque passage par la boucle représentée dans la figure I-10. implique l'exécution d'une microinstruction du microprogramme.

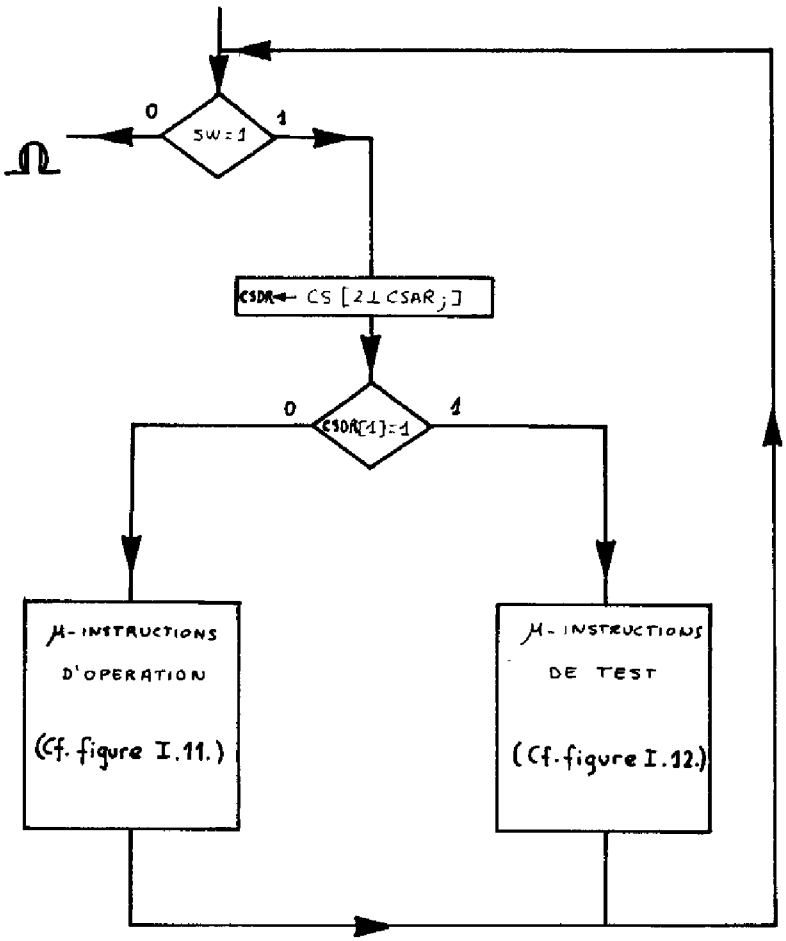

0
SW=1
1
CSDR ← CS [2⊥CSAR;]
0
CSDR[1]=1
1
μ- INSTRUCTIONS D'OPERATION
(Cf. figure I.11.)
μ- INSTRUCTIONS DE TEST
(Cf. figure I.12.)

Figure I-10

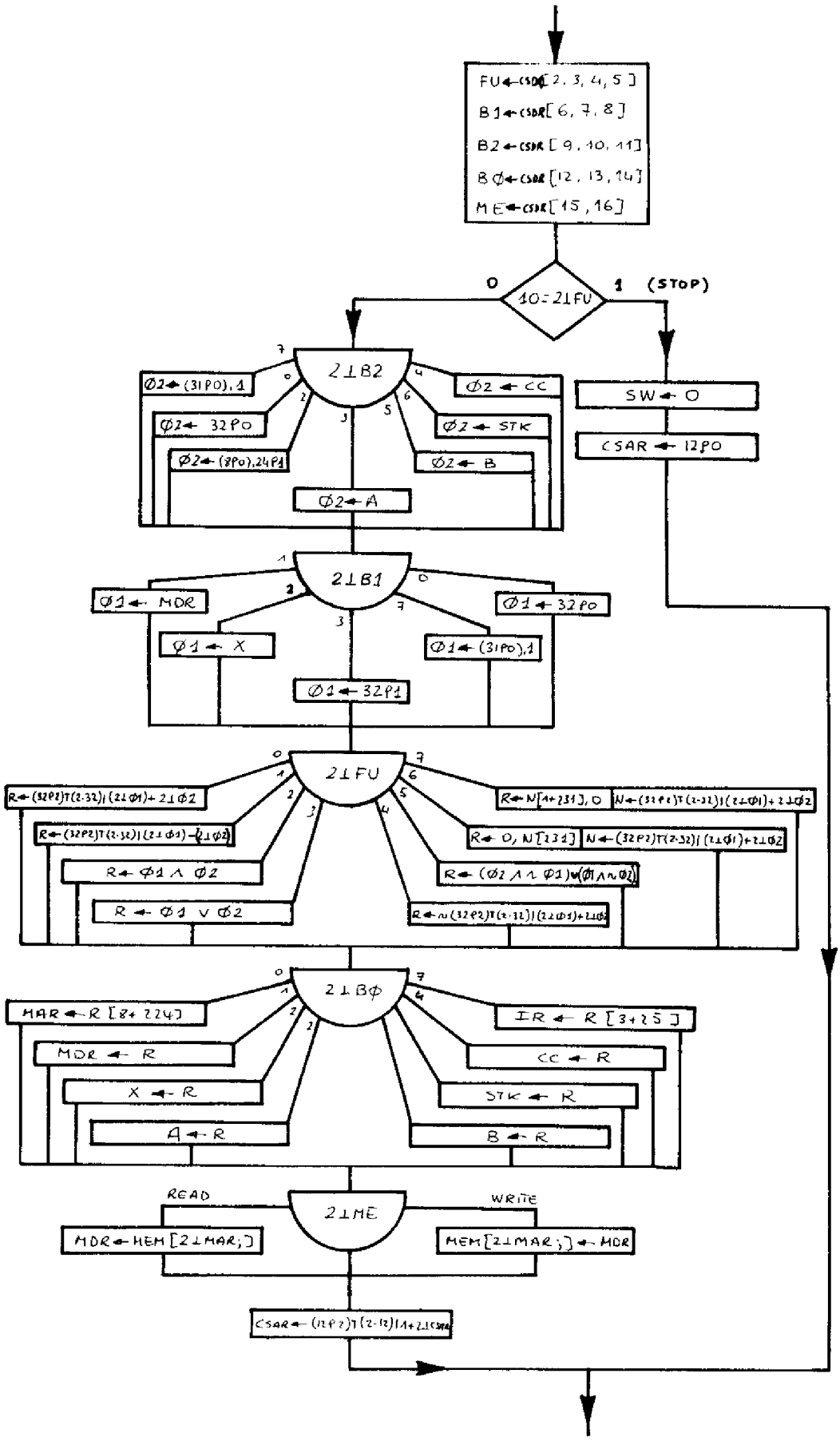

FU ← CSDR [2, 3, 4, 5]
B1 ← CSDR [6, 7, 8]
B2 ← CSDR [9, 10, 11]
BØ ← CSDR [12, 13, 14]
ME ← CSDR [15, 16]
0
10 = 2⊥FU
1 (STOP)
2⊥B2
Ø2 ← (31PO), 1
Ø2 ← 32PO
Ø2 ← (8PO), 24P1
Ø2 ← CC
Ø2 ← STK
Ø2 ← B
Ø2 ← A
SW ← 0
CSAR ← 12PO
2⊥B1
Ø1 ← MDR
Ø1 ← X
Ø1 ← 32PO
Ø1 ← (31PO), 1
Ø1 ← 32P1
2⊥FU
R ← (32P2)T(2-32)| (2⊥Ø1) + 2⊥Ø2
R ← (32P2)T(2-32)| (2⊥Ø1) − 2⊥Ø2
R ← Ø1 ∧ Ø2
R ← Ø1 ∨ Ø2
R ← N[1+231], 0
N ← (31P1)T(2-32)| (2⊥Ø1) + 2⊥Ø2
R ← 0, N[231]
N ← (32P2)T(2-32)| (2⊥Ø1) + 2⊥Ø2
R ← (Ø2 ∧ Ø1) ∨ (Ø1 ∧ ∼Ø2)
R ← ∼(32P2)T(2-32)| (2⊥Ø1) + 2⊥Ø2
2⊥BØ
MAR ← R [8+22 4]
MDR ← R
X ← R
A ← R
IR ← R [3+25]
CC ← R
STK ← R
B ← R
READ
WRITE
2⊥ME
MDR ← MEM [2⊥MAR;]
MEM [2⊥MAR;] ← MDR
CSAR ← (12P2)T(2-12)| 14+2⊥CSAR

Figure I-11

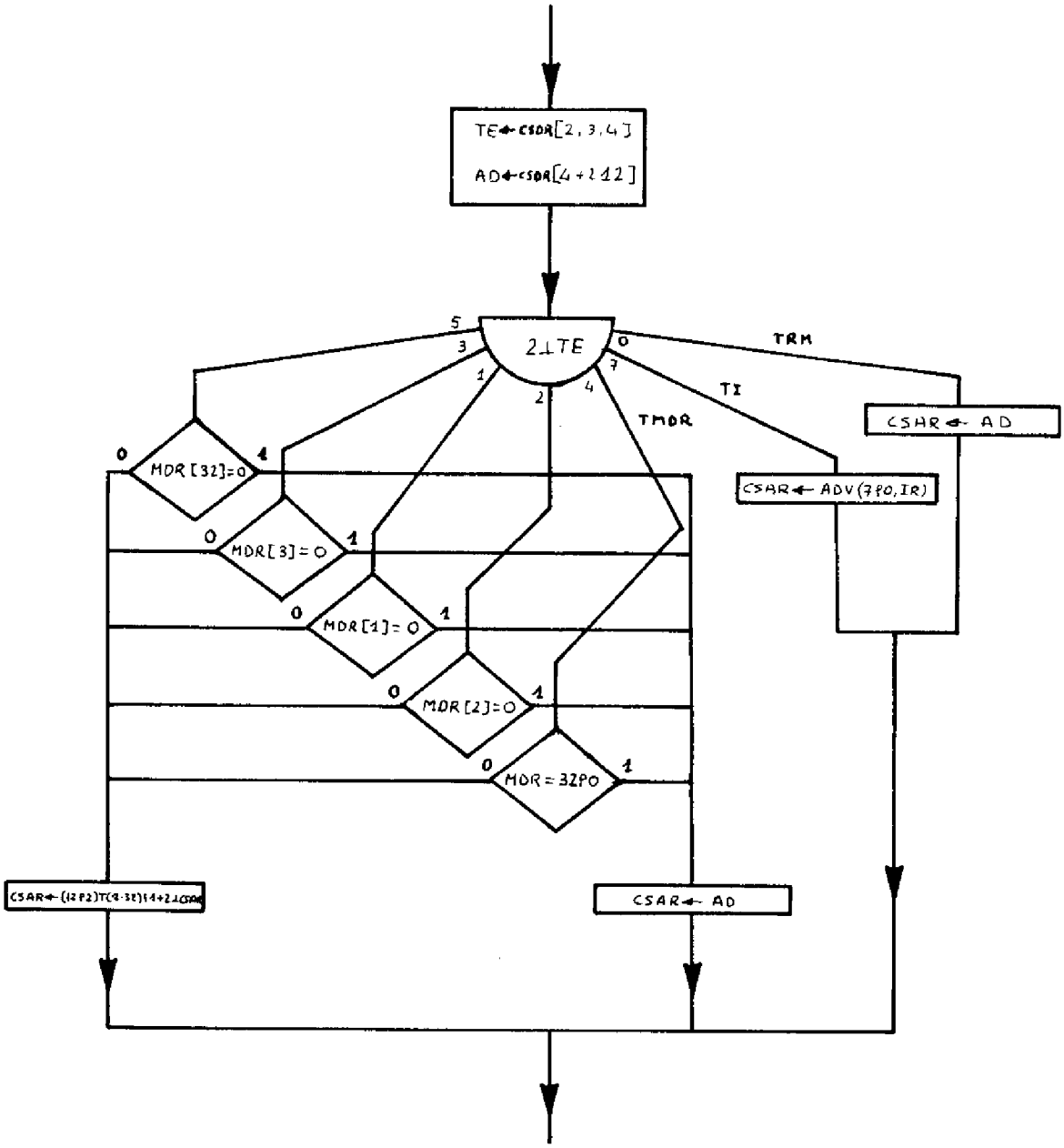

TE←CSDR[2,3,4]
AD←CSDR[4+2,12]
2⊥TE
5
3
1
2
4
7
0
TRM
TI
TMDR
CSAR←AD
MDR[32]=0
MDR[3]=0
MDR[1]=0
MDR[2]=0
MDR=32P0
CSAR←ADV(7P0,IR)
CSAR←AD

Figure I-10

Pour compléter la description du niveau "réalisation microprogrammée", nous avons traduit le contenu de la mémoire de microprogramme sous forme d'organigramme. Ceci est présenté dans la figure I-13. qui montre l'organigramme général correspondant au microprogramme.

Sur cette figure, on distingue une partie commune à l'exécution de toutes les instructions de la S- Machine ; cette partie débute par le chargement du registre MAR avec le contenu du registre compteur ordinal CC ; ceci étant effectué, une opération de lecture est commandée à la mémoire principale. Ensuite, on réalise l'incrémentation du contenu du compteur ordinal. Le résultat de la lecture sur la mémoire principale étant enregistré dans le registre MDR, on commande le chargement dans le registre B de la partie adresse de l'instruction et le chargement du registre IR avec la partie de cette instruction qui comporte le code opération décalée d'un bit à gauche (pour des raisons d'adresse).

Une fois cette partie de recherche et construction d'instruction effectuée, un test sur le premier bit du code opération de l'instruction provoque un branchement dans le microprogramme vers un groupe de microinstructions qui correspondra à l'exécution des instructions à "une adresse" ou à "zéro adresse" selon que la valeur de ce bit est un "un" ou un "zéro".

Le groupe des instructions à une adresse comporte une partie du microprogramme commune,appelée calcul d'adresse. Cette partie n'est pas développée sur l'organigramme de la figure I-13. et nous la présentons ici :

Le test TI représente une opération de branchement à l'intérieur du microprogramme selon le contenu du registre comportant le code opération IR.

Les figures I-14. et I-15. montrent le séquencement d'actions effectuées par les microinstructions pour chacune des différentes instructions de la S- Machine.

Dans ces trois derniers organigrammes, nous avons utilisé une notation informelle pour simplifier l'écriture ; cependant il faut souligner que pour effectuer plus tard la vérification on tient compte de l'écriture formelle correspondante. Ainsi par exemple, la notation formelle équivalente à l'expression : MAR ← CC ; READ est la suivante :

{ MAR ← CC [8 + ? 24]
{ MDR ← MEM [2 ⊥ MAR ;]

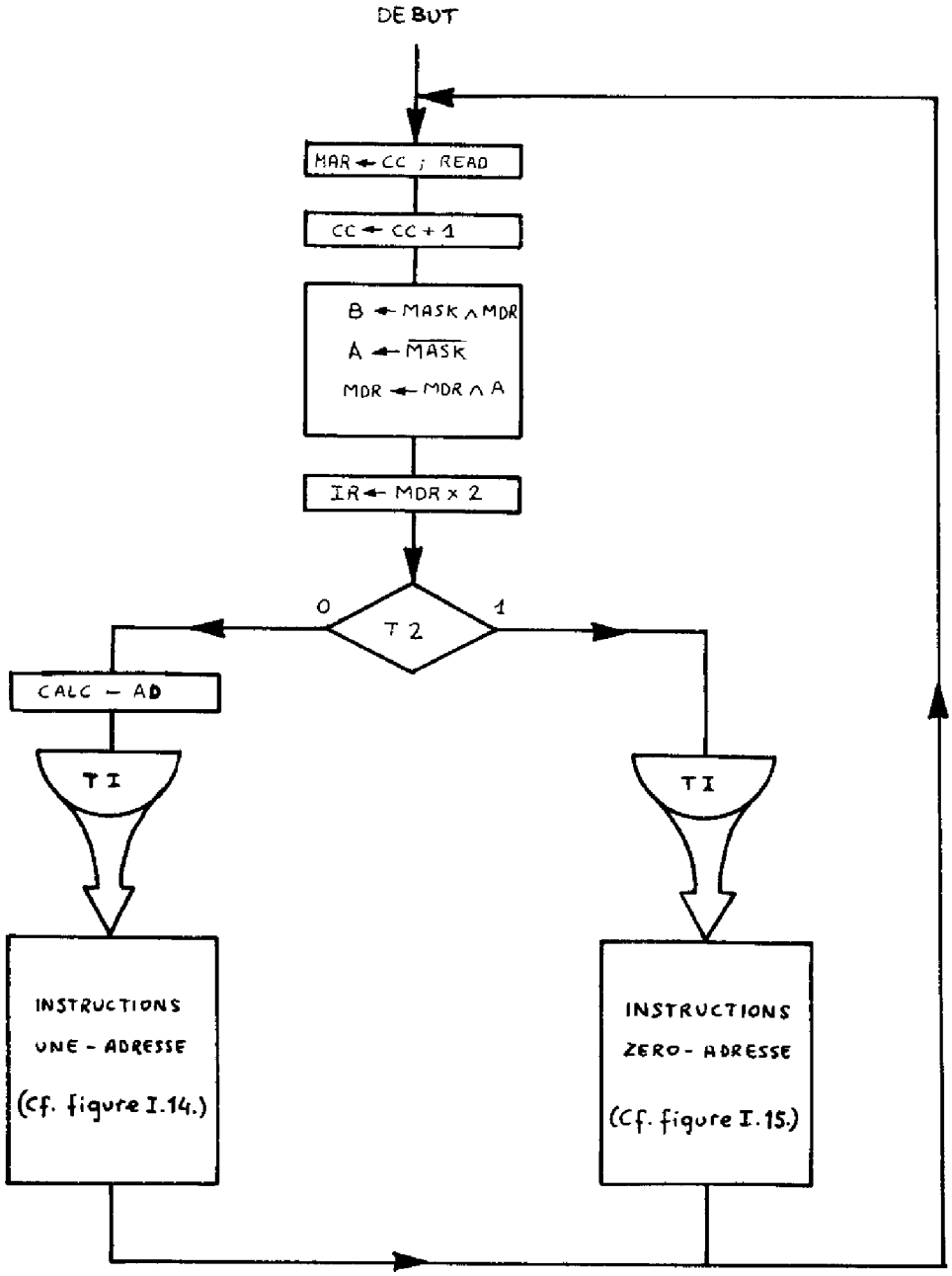

DEBUT
MAR ← CC ; READ
CC ← CC + 1
B ← MASK ∧ MDR
A ← $\overline{MASK}$
MDR ← MDR ∧ A
IR ← MDR x 2
T 2
0
1
CALC – AD
TI
TI
INSTRUCTIONS
UNE - ADRESSE
(Cf. figure I.14.)
INSTRUCTIONS
ZERO - ADRESSE
(Cf. figure I.15.)

Figure I-13

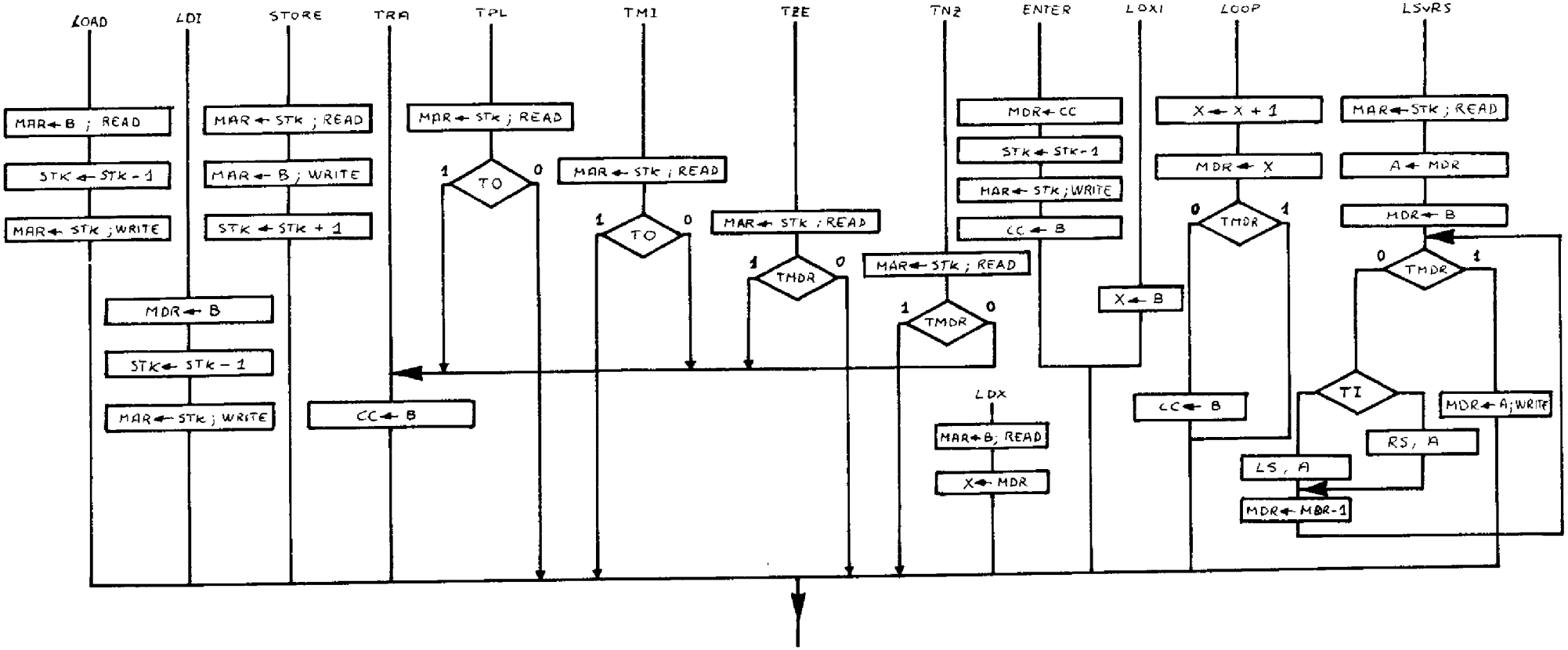

LOAD
MAR ← B ; READ
STK ← STK - 1
MAR ← STK ; WRITE
LDI
MDR ← B
STK ← STK - 1
MAR ← STK ; WRITE
STORE
MAR ← STK ; READ
MAR ← B ; WRITE
STK ← STK + 1
TRA
CC ← B
TPL
MAR ← STK ; READ
1
TO
0
TM1
MAR ← STK ; READ
1
TO
0
TZE
MAR ← STK ; READ
1
TMDR
0
TNZ
MAR ← STK ; READ
1
TMDR
0
ENTER
MDR ← CC
STK ← STK - 1
MAR ← STK ; WRITE
CC ← B
LDX
MAR ← B ; READ
X ← MDR
LDXI
X ← B
LOOP
X ← X + 1
MDR ← X
0
TMDR
1
CC ← B
LSVRS
MAR ← STK ; READ
A ← MDR
MDR ← B
0
TMDR
1
TI
LS , A
MDR ← MBR-1
RS , A
MDR ← A ; WRITE

Figure I-14

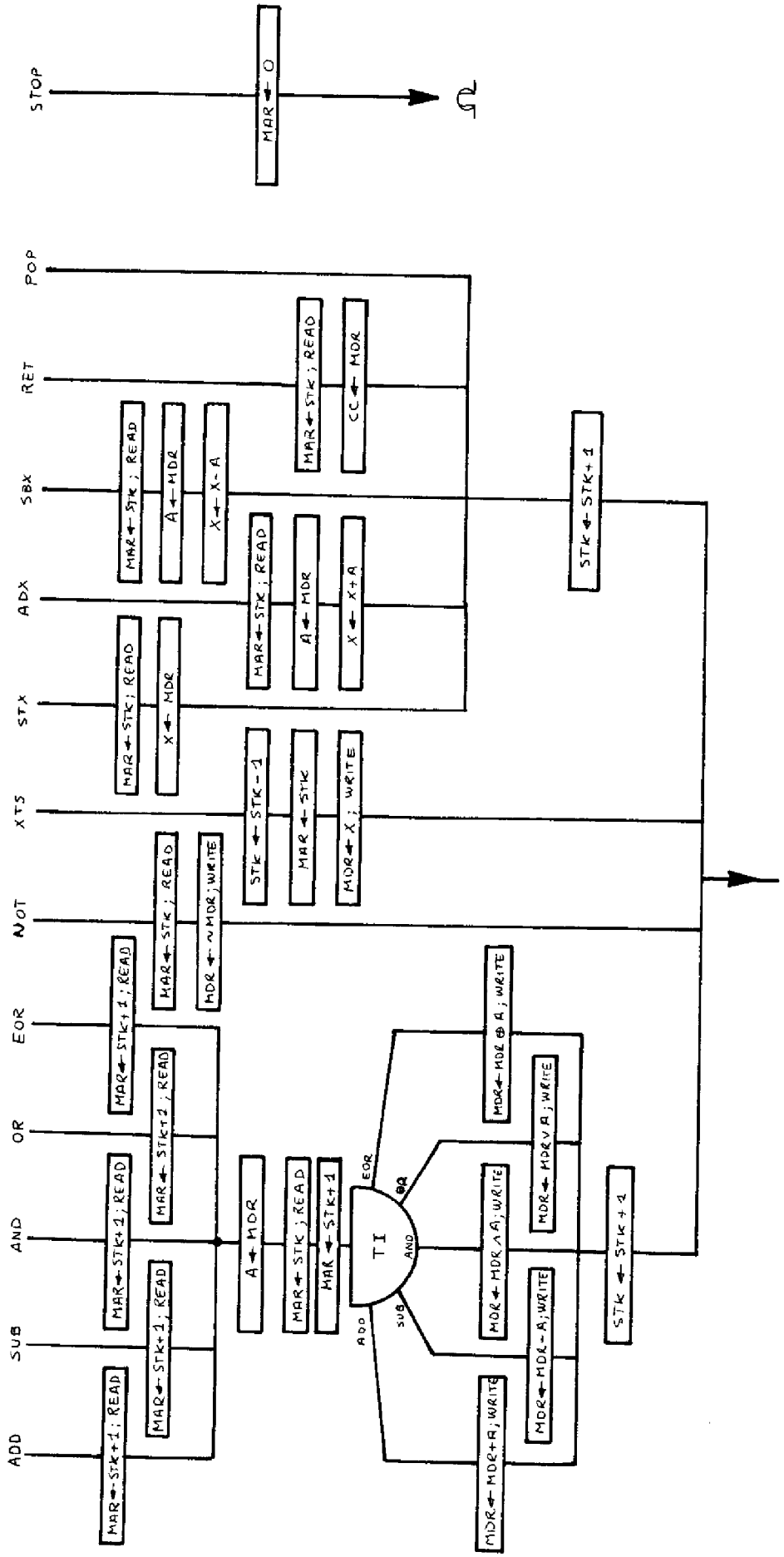

STOP
MAR ← 0
POP
RET
SBX
ADX
STX
XTS
NOT
EOR
OR
AND
SUB
ADD
MAR ← STK ; READ
CC ← MDR
A ← MDR
X ← X - A
X ← X + A
STK ← STK + 1
STK ← STK - 1
MAR ← STK
MDR ← X ; WRITE
MDR ← ~ MDR ; WRITE
MAR ← STK + 1 ; READ
TI
EQ
MDR ← MDR ⊕ A ; WRITE
MDR ← MDR ∨ A ; WRITE
MDR ← MDR ∧ A ; WRITE
MDR ← MDR - A ; WRITE
MDR ← MDR + A ; WRITE

Figure I-15

A partir de ces organigrammes, nous avons mené une étude de vérification qui consiste à mettre en évidence (d'une manière algébrique) que le niveau "réalisation microprogrammée" exécute les différentes instructions de la manière définie par le niveau "spécification" qui a un rôle que l'on pourrait associer à celui d'un "cahier de charges". Cette étude est présentée dans le chapitre II.

Nous avons voulu montrer, aussi, que la description en forme d'organigramme, facilite la réalisation d'une simulation de la structure de chaque niveau ; ceci est le sujet du paragraphe suivant.

I-5. UTILISATION DU LANGAGE APL POUR LA SIMULATION DE LA STRUCTURE DE LA S- MACHINE

Nous avons envisagé l'implémentation d'une simulation de la S- Machine, dans le cadre de la vérification de la concordance entre les deux niveaux décrits précédemment.

Etant donné que nous avions, au départ, une même structure définie à deux niveaux différents et sous forme d'organigramme, la tâche de mise en place d'une simulation de chacun de ces niveaux s'avérait assez simple.

Dans un premier essai , nous avons réalisé cette simulation en associant les organigrammes à des réseaux de Petri, puisque nous avions à notre disposition une série de programmes d'implantation de réseaux de Petri en langage APL réalisés par J. Renalier et P. Azéma (18) qui nous ont permis de mettre en place la simulation très rapidement ; cependant cette simulation à base de réseaux de Petri a dû être écartée car le temps de calcul était trop important pour des programmes avec plusieurs

instructions, ce qui entraînait des coûts de calcul prohibitifs. D'autre part cet outil offert par les réseaux de Petri aurait été plus intéressant si dans la structure de la S- Machine, on avait des parallélismes dans l'exécution, ce qui n'est pas le cas.

L'étape suivante a donc été de réaliser la simulation directement à partir des organigrammes et aussi en langage APL. Nous avons déjà expliqué les raisons pour lesquelles nous avons choisi ce langage pour la représentation des actions sur le contenu des registres binaires. Nous pouvons ajouter maintenant à ces raisons certaines autres, telles que la facilité de traitement au niveau du bit et l'aspect conversationnel de ce langage.

Les figures I-16. et I-17. montrent les organigrammes-blocs de la simulation du niveau "spécification" (ABSM) et de celle du niveau "réalisation microprogrammée" (SEXM). L'écriture APL de ces deux simulations ainsi que le microprogramme sont présentés en ANNEXE.

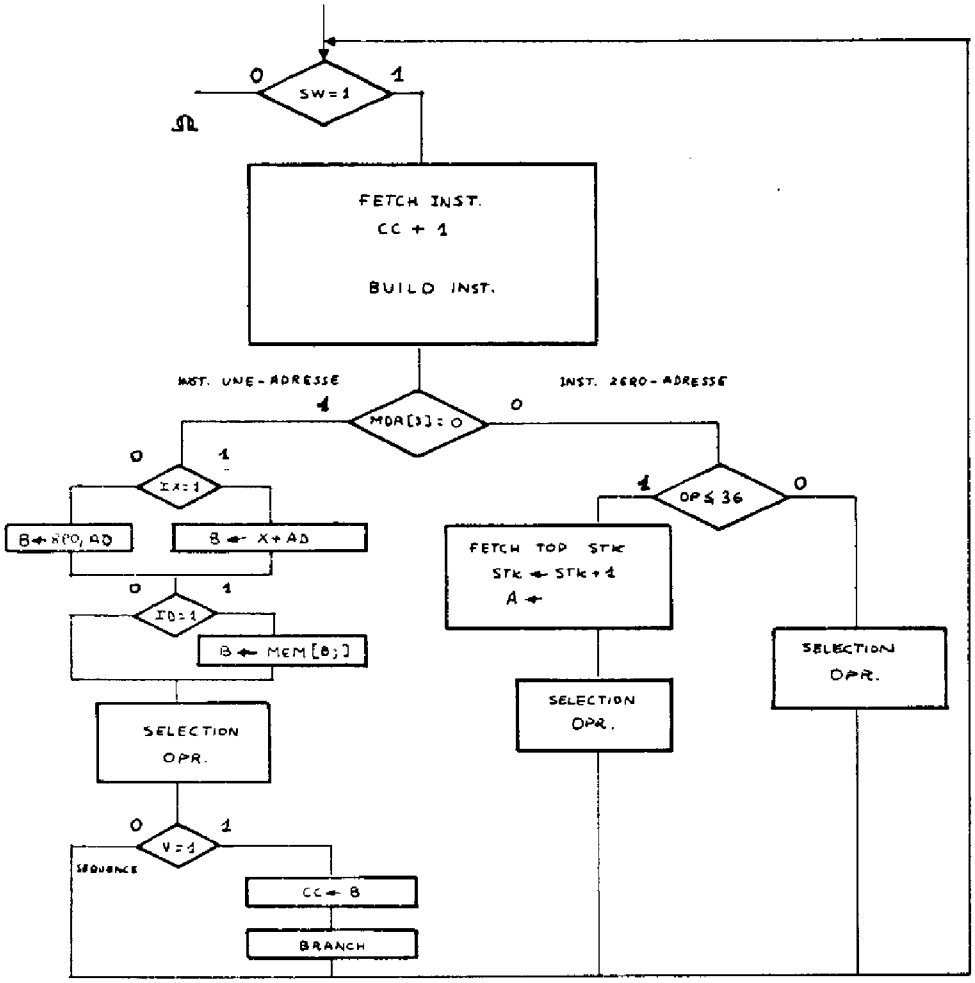


Figure I-16

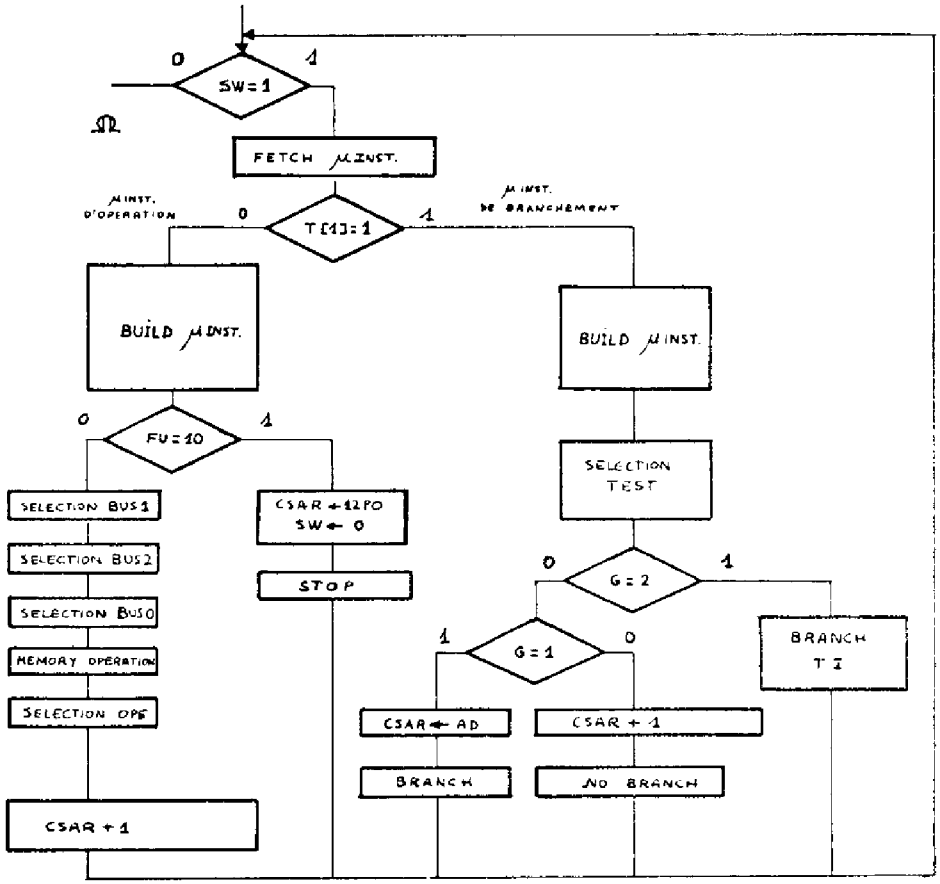


Figure I-17

Cette simulation, par son principe, constitue une première vérification simple, pratique mais néanmoins peu efficace.

Nous voulons maintenant, prouver de façon plus précise l'exactitude de la réalisation microprogrammée.

Pour ceci, deux solutions sont possibles : l'une utilise le langage VDL et l'autre la représentation par organigramme.

Nous avons déjà souligné le travail important et délicat, lié à la réalisation d'un interpréteur VDL. Le travail correspondant ne peut se justifier que si l'interpréteur est ensuite utilisé un grand nombre de fois. Comme ceci n'était pas évident à ce moment de l'étude, nous avons recherché une forme plus accessible et pratique, ce qui nous a conduit à utiliser les organigrammes. Leur principal inconvénient réside dans le fait qu'ils doivent être écrits à nouveau, peut-être complétement, si la machine à analyser est modifiée. Néanmoins, cet inconvénient est considérablement réduit par la simplicité de leur élaboration. De plus, nous allons maintenant montrer qu'ils peuvent être utilisés de façon très performante pour effectuer une vérification formelle de la réalisation microprogrammée.

CHAPITRE II

VERIFICATION DE LA MICROPROGRAMMATION

II-1-1 INTRODUCTION

Le problème de la vérification de programmes a été, presque exclusivement, centré sur la méthode formalisée par Floyd (6). Cette méthode présente un programme avec deux prédicats sur les variables traitées par celui-ci ; ces prédicats sont les assertions initiale et finale. Si le programme débute dans un certain état initial pour lequel l'assertion initiale est vraie et s'arrête dans un état final, l'assertion finale doit être vraie pour que le programme soit correct. De plus, dans cette méthode, il existe des assertions inductives additionnelles qui sont affectées à des points intermédiaires du programme, ainsi qu'une procédure pour établir la vérification de la correction partielle du programme en prouvant un ensemble de conditions de vérification.

Dans l'optique de ce chapitre, ceci serait un cas particulier du problème qui nous concerne, celui de la vérification de la simulation entre deux programmes.

Le concept de la correcte simulation algébrique d'un programme par un autre est emprunté à Milner (17). D'une manière simple, on peut dire qu'une simulation entre deux programmes est assimilable à une correspondance entre certains points de l'exécution des deux programmes. La preuve de la validité de cette simulation est faite par rapport à une relation de simulation.

Il est évident qu'une procédure de vérification ayant la possibilité d'être automatisée en grande partie présenterait le plus haut intérêt (cet aspect a été abordé par Birman et Joyner(2)). Cependant, considérant que l'envergure d'une telle automatisation (qui entraîne l'utilisation d'outils tels que des démonstrateurs de théorèmes et/ou des interpréteurs symboliques) dépasserait le cadre de ce mémoire, nous présentons dans ce chapitre une approche "manuelle"

qui aide à la vérification en apportant une certaine systématisation de la procédure et un degré de formalisme plus élevé qu'une autre méthode "intuitive".

Une application de la théorie de la simulation entre programmes est celle de la vérification du bon fonctionnement d'un microprogramme par rapport à la conception initiale du système dans lequel il est implémenté.

Dans le premier chapitre, nous avons donné une description formelle de la S- Machine ainsi qu'une simulation des deux niveaux de la description qui permet, en particulier, de détecter des erreurs "grossières" au niveau de la réalisation microprogrammée. Dans ce deuxième chapitre, nous réalisons une étude de vérification formelle pour valider définitivement la structure de la Machine.

Nous présentons, d'abord, une base théorique pour la simulation algébrique entre programmes avec une première application à la S- Machine décrite au premier chapitre ; elle sera ensuite suivie de l'étude du traitement spécial que l'on doit appliquer aux parties d'un organigramme dont on veut vérifier que la conception est correcte, quand celui-ci comporte des boucles.

II-1-2. Théorie de la simulation

Le concept de simulation que nous prenons comme base de cette étude est dû à Milner (17) et Birman (1). Avant de présenter les conditions qui doivent être remplies pour qu'une relation entre deux programmes puisse être considérée comme une simulation, il faut introduire un certain nombre de définitions.

II-1-2-1. Notation

D et D' étant deux ensembles, R est une relation entre eux si :

$R \subset D \times D'$

Un élément de R est représenté par la notation

$$(d,d') \in R$$

L'inverse de R est notée R^{-1} et est définie comme :

$$R^{-1} = \{(d',d) | (d,d') \in R\}$$

La relation R est une fonction partielle si pour chaque $d \in D$ il existe au moins un $d' \in D'$ tel que

$$(d,d') \in R$$

et on peut alors écrire $R : D \rightarrow D'$ ou bien $R(d) = d'$ pour $(d,d') \in R$.

Si R est une fonction et si pour un $d \in D$ il existe un $d' \in D'$ tel que $R(d) = d'$ on dit que R est définie pour d et ceci est représenté par $< Rd >$.

Soit $F : D \rightarrow D$ une fonction interne en D et soit $d \in D$; on définit $F^i d$ comme :

$F^\circ d = d$

si $< F^{i-1} d > \rightarrow F^i d = F(F^{i-1} d)$

sinon $\rightarrow$ indéfini

II-1-2-2. Conditions pour la relation de simulation

Un programme abstrait est représenté comme un triplet P = (D, Do, F), où D est un ensemble appelé "domaine" de P ; $Do \subset D$ est un ensemble de valeurs initiales et $F : D \rightarrow D$ est une fonction partielle.

L'exécution du programme P est représentée par une séquence (do, d1,...) dans laquelle :

$$d_o \in D_o \; ; \; d_{i+1} = F d_i \; , \; i = 1,2 \ldots$$

Dans un programme en forme d'organigramme on peut considérer le domaine D comme : $D \subset M \times E$, où M est l'ensemble des points de marquage représentant des différentes étapes dans le déroulement du programme, et E étant l'ensemble de valeurs possibles du vecteur d'état du programme.

La figure II-1. montre un exemple pour illustrer ceci.

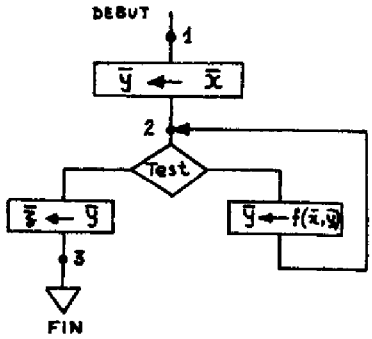
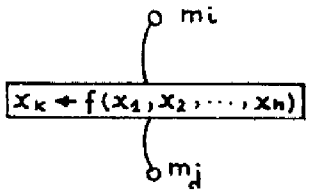


Figure II-1

Dans cet exemple très simplifié, E contient toutes les valeurs possibles du vecteur $\bar{y}$ pour ce programme et M = { 1, 2, 3 } .

Pour aider à l'interprétation de ce qui serait une exécution symbolique dans un organigramme nous allons considérer une fonction "avance" qui prend comme argument $s \in MXE$ et qui fournit comme résultat une valeur qui se trouve aussi dans MXE après exécution d'une instruction. L'exemple suivant montre l'action de cette fonction. (Dans cet exemple M représente l'ensemble des noeuds).



$$(x_1, x_2, \cdots, x_n) \in E$$

$$m_i, m_j \in M$$

$$\underline{\text{avance}} \begin{pmatrix} m_i \\ x_1, x_2, \cdots, x_n \end{pmatrix} = \begin{pmatrix} m_j \\ x_1, x_2, \cdots, x_{k-1}, f(x_1, \cdots, x_n), \cdots x_n \end{pmatrix}$$

Considérons aussi une autre fonction $\underline{\text{avance}}^{\circ}$ qui fournit la première valeur dans D trouvée par application répétitive d'avance :

$$\underline{\text{avance}}^{\circ}(S) = S \text{ , si } S \in D$$
$$= \underline{\text{avance}}^{\circ}(\underline{\text{avance}}(S)) \text{ , si } S \notin D$$

A partir de ceci on peut définir la fonction suivante :

$$F(d) = \text{avance}^{\circ}(\text{avance}(d)) \text{ , si } d \in D \text{ et si } \langle \text{avance}^{\circ}(\text{avance}(d)) \rangle$$

indéfinie , autrement

Cette fonction étant définie nous pouvons maintenant présenter les conditions qui doivent être remplies par la relation de simulation.

Soient P = (D,Do,F) et P' = (D',D'o,F') deux programmes. Une relation $R \subset D \times D'$ est une SIMULATION de P par P' si :

C1.- $\forall (d,d') \in R$, $< F(d) >$ si et seulement si $< F'(d') >$ et si les deux sont définies, alors $(F(d), F'(d')) \in R$

C2.- $\forall d_o \in D_o$, $\exists\, d'_o \in D'_o$ tel que $(d_o, d'_o) \in R$

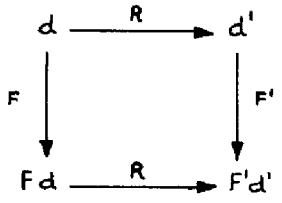
C3.- $\forall d'_o \in D'_o$, $\exists\, d_o \in D_o$ tel que $(d_o, d'_o) \in R$

C4.- $\forall d \in D$, $\forall d' \in D'$ tels que $(d,d') \in R$ il doit résulter que $\forall d_1 \in D$ si $(d_1, d') \in R$ alors $d_1 = d$

D'un point de vue mathématique la condition C1 exprime que R doit être un homomorphisme de P dans P. (Cette condition est appelée condition de simulation faible). Les trois autres conditions déterminent que R doit être un isomorphisme de P dans P'.

Dire que R est une simulation de P par P', signifie d'une certaine manière, que P' est capable d'exécuter tout ce que P réalise et ceci en fournissant des résultats égaux.

La condition C1, qui exprime que R doit être un homomorphisme entre les structures algébriques (D,F) et (D',F'), signifie aussi que si R est telle que la condition est satisfaite, le diagramme de la figure suivante semicommute.



Il existe une équivalence entre le contenu de cette condition C1 et l'expression suivante due à Milner.

[1] $\forall (d,d') \in R$, $(F(d), F'(d')) \in R \iff RF' \subset FR$

Cette expression est intéressante pour l'étude de la situation du niveau "spécification" d'un processeur par le niveau "implémentation", puisque dans une telle étude il est convenable de définir la fonction "exécution" F comme une composition de la forme :

$$F = F1, F2, \ldots, Fn$$

et, comme l'on verra plus loin, on peut ainsi procéder à une étude des états de la machine à des différents points intermédiaires de l'exécution des instructions.

Une telle situation est représentée dans le diagramme suivant pour deux programmes abstraits P = (D, Do, F) et P' = (D', Do', F').

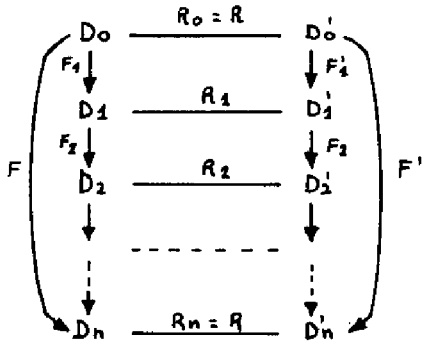


Figure II-2

Comme exemple on pourrait prendre n = 2 en assignant à F1 et F1' l'exécution de la tâche "recherche d'instruction" de S et $\not$MS respectivement et à F2 et F2' la réalisation de la tâche "exécution d'instruction".

Le Théorème suivant dû à Leeman (15) définit un moyen d'établir l'existence d'une relation de simulation entre deux programmes qui se présenteraient comme il est montré dans la figure précédente .

THEOREME 1 - Soit $R \subset D \times D'$ une relation

s'il existe des domaines : $D \doteq Do, D1, \ldots, Dn = D$

$D' = Do', D'1, \ldots, Dn' = D'$

et des fonctions : $Fk : Dk-1 \rightarrow Dk$

$F'k : D'k-1 \rightarrow D'k$

telles que : $F = F1\ F2 \ldots Fn$

$F' = F'1\ F'2 \ldots Fn'$

et des relations : $R = Ro, R1, R2, \ldots, Rn = R$

telles que le diagramme de la figure II-2. semicommute alors R est une simulation de P par P'.

- Démonstration -

Dire que le diagramme de la figure II-2. semicommute signifie que : $R_{k-1} F'_k \subset F_k R_k \quad , \quad k = 1,2,\ldots,n$

Par conséquent :

$$R F' = R_0 F'_1 F'_2 \cdots F_n \subset F_1 R_1 F'_2 \cdots F_n \subset F_1 F_2 R_2 F'_3 \cdots F_n \subset$$

$$\subset \cdots \subset F_1 F_2 \cdots F_n R_n = F R$$

Ce qui démontre par [1] que R est une simulation de P par P'.

Une propriété de la relation de simulation entre programmes qui s'avère très intéressante pour simplifier le problème de la vérification est celle de la transitivité démontrée dans le théorème suivant qui est un théorème de la théorie des homomorphismes. Dans ce théorème nous utilisons le symbole $\gtrsim$ pour représenter l'expression "simulé par".

THEOREME 2 - La relation $\gtrsim$ entre programmes est réflexive et transitive.

- Démonstration -

Utilisant comme repère les 4 conditions définies plus haut il est facile de voir que :

$\forall P \exists R$ telle que $P \gtrsim P$ (par rapport à R)

P étant : $P = (D, D_0, F)$ et $R = \{(d,d) \mid d \in D\}$

La relation de simulation R serait la relation d'identité dans le domaine D.

En ce qui concerne la transitivité, soient

$P = (D, D_0, F)$, $P' = (D', D'_0, F')$ et $P'' = (D'', D''_0, F'')$

trois programmes et soient R et R' telles que :

$P \wr P'$ par rapport à R

$P' \wr P''$ par rapport à R'

soit $Q = R R' = \{(d,d'') \mid d \in D, d'' \in D'', (\exists d' \mid (d,d') \in R, (d',d'') \in R')\}$

En regardant maintenant les conditions qui doivent être satisfaites par une relation pour être une simulation, on vérifie que : $P \wr P''$ par rapport à Q.

II-1-3. Application aux processeurs microprogrammés

Illustration d'une approche sur la S- Machine

Le thème de la simulation entre programmes touche à plusieurs aspects de la science des processeurs. Ainsi on peut distinguer deux lignes importantes ; d'une part la modification de programmes (par exemple, optimisation d'un programme compilateur) où le problème se pose de savoir si le programme amélioré est fonctionnellement équivalent au programme précédent. D'autre part nous avons la vérification de la correcte émulation du fonctionnement spécifié d'un processeur, par une réalisation, par exemple, de type microprogrammée.

Notre travail prend en considération cet aspect pour lequel une première étude a été menée par Birman (1) en 1973. A la différence de BIRMAN, qui a réalisé son étude sur les descriptions dans un langage VDL/APL des deux niveaux de la S- Machine, nous avons préféré baser la nôtre sur les descriptions en forme d'organigramme de ces deux niveaux ce qui s'est avéré plus simple, dans le cadre d'une procédure de vérification "manuelle".

Comme dans les descriptions présentées au chapitre 1, nous utiliserons les expressions APL tout au long de la procédure de vérification.

Comme introduction de cet aspect de la simulation entre programmes, orienté vers l'étude de la vérification de l'implémentation correcte du microprogramme d'un processeur, nous présentons ci-dessous un exemple d'application de la procédure de vérification.

Soient deux programmes dont les organigrammes sont montrés dans la figure II-3. Ces deux programmes sont censés traiter les mêmes informations et aboutir aux mêmes résultats mais, comme il est montré dans la figure II-3., de manières différentes.

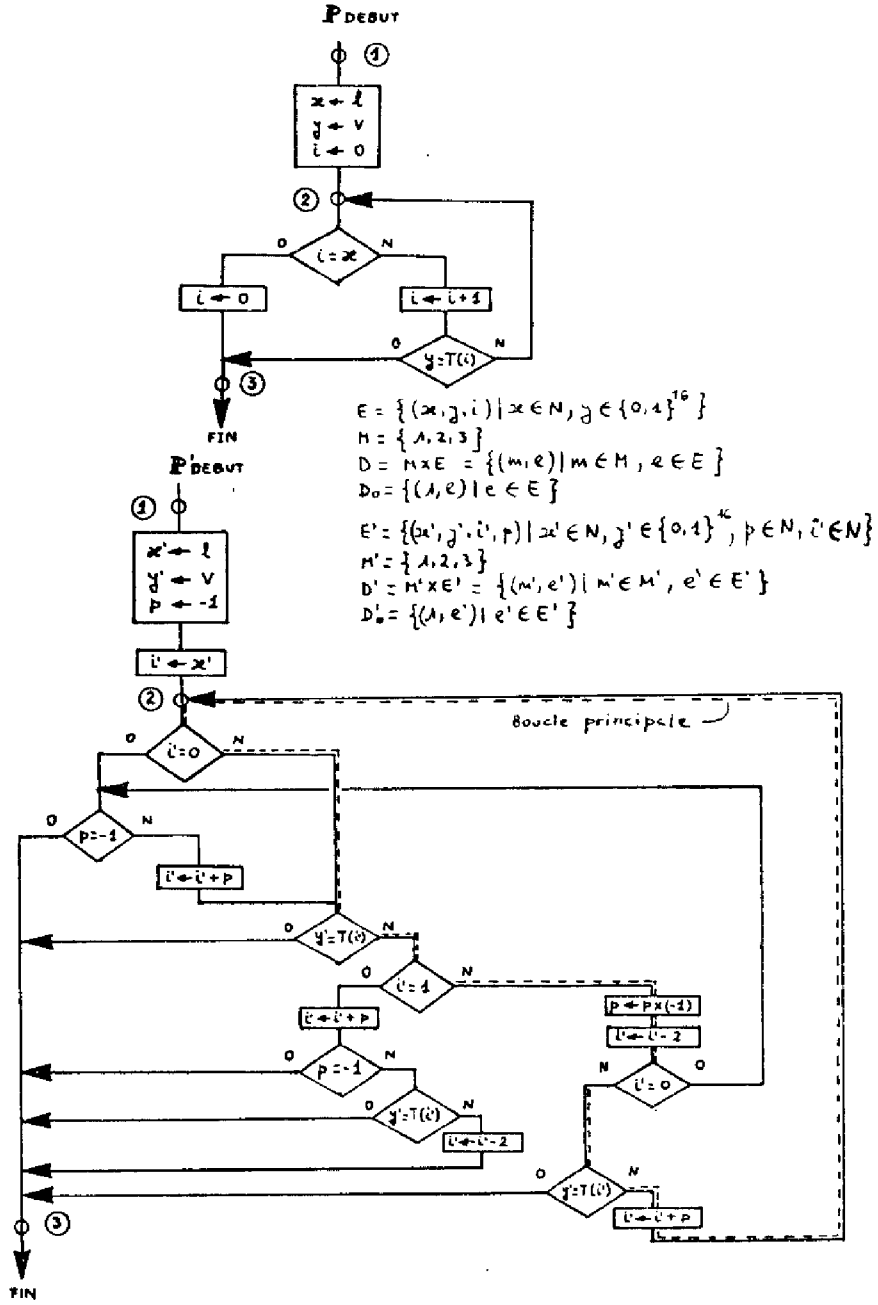


Figure II-3

Le problème consiste, donc, à vérifier que le programme P' remplit les conditions nécessaires pour que l'on puisse affirmer qu'il simule P.

Le programme P réalise la scrutation d'un tableau T qui contient "I" éléments (vecteurs binaires de 16 bits par exemple) différents et compare une valeur donnée, V, à chaque élément du tableau. Ceci donne comme résultat le numéro de la ligne du tableau T où se trouve l'élément égal à V.

Dans le cas où $V \notin T$ le résultat est zéro.

Il est évident que si P' a exactement le même rôle que P il est loin d'être optimal puisque le procédé suivi par P' est plus compliqué que celui de P. La différence est dans l'ordre de la scrutation, ce qui est montré par la figure II-4.

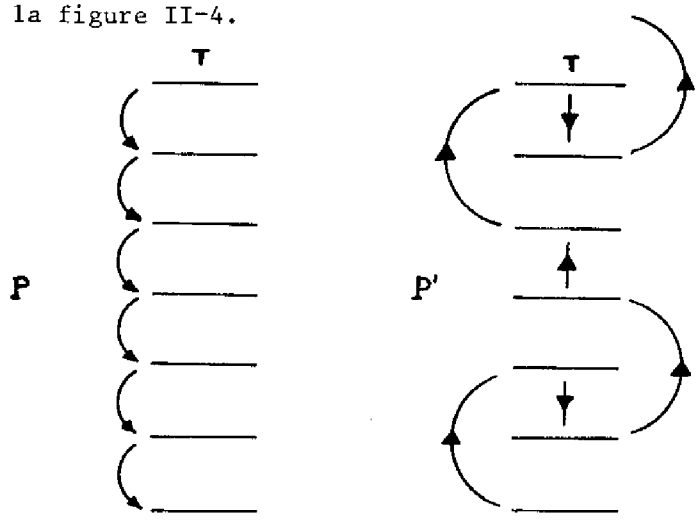


Figure II-4

Pour cet exemple, qui ne prétend être qu'une illustration de la méthode de vérification nous ne montrerons que les lignes générales de la procédure sans nous attarder à une vérification exhaustive.

Nous soulignons aussi que l'existence d'une boucle dans chacun des programmes P et P' demande, dans le cadre de la vérification de la simu-

lation, une attention spéciale (ceci sera exposé d'une manière plus détaillée sur la S-Machine).

La vérification de la simulation correcte étant faite par rapport à une certaine relation R, nous allons établir cette relation entre ces deux programmes P et P'. R est composée de l'union de trois relations partielles établies en trois points de marquage des organigrammes : au début du programme, en entrée de la boucle et en sortie du programme. Le critère de choix des éléments qui composent ces relations est donné par la connaissance du rôle du programme P.

$R = R1 \cup R2 \cup R3$

$R1 = \{(d,d') \mid m = m' = 1,\ x = x',\ y = y'\}$

$R2 = \{(d,d') \mid m = m' = 2,\ x = x',\ y = y'\}$

$R3 = \{(d,d') \mid m = m' = 3,\ x = x',\ y = y',\ i = i'\}$

La preuve de la correcte simulation est faite en regardant si R remplit les conditions établies en II-1-2-2.

- Etude concernant C1 -

a) Soit (d,d') R1, m = m' = 1. L'exécution symbolique de F(d) et de F'(d') est représentée par les programmes qui suivent, délimités par DEBUT et FIN.

P DEBUT (m = 1)	P' DEBUT (m' = 1)
x ← r1 : 1	x' ← S1 : 1
y ← r2 : V	y' ← S2 : V
i ← r3 : 0	p ← S3 : -1
P FIN (m = 2)	i' ← S4 : S1
	P' FIN (m = 2)

Pour vérifier que $(F(d), F'(d')) \in R2$ on montre que quand le point de marquage 2 est atteint, les relations suivantes sont satisfaites :

1) $x = x'$
2) $y = y'$

Les paramètres ri et si sont des paramètres intermédiaires utilisés dans une interprétation symbolique pour faciliter la comparaison des valeurs

symboliques au moment de la vérification.

b) Soit $(d,d') \in R2$ tel que $x = x' = 0$

P DEBUT (m = 2) P' DEBUT (m' = 2)

i ← r1 : 0

P FIN (m = 3) P' FIN (m' = 3)

Pour vérifier que $(F(d), F'(d')) \in R3$

1) x = x'
2) y = y'
3) i = i' = 0

c) Soit $(d,d') \in R2$ tel que $x = x' \neq 0$ et que $V = T[n]$

P DEBUT (m = 2)

n fois {
i ← r1 : 0 + 1
T non
i ← r2 : r1 + 1
T non
:
i ← rn-1 : rn-2 + 1
T non
i ← rn : rn-1 + 1
T oui
}

P FIN (m = 3)

P' DEBUT (m' = 2)

$Sup\left[\frac{\ell-n}{2}\right]$ fois {
T1 non
T2 non
T3 non
p ← S1 : - p
i' ← S2 : i' - 2
T4 non
i' ← S3 : S2 + S1
T1 non
T2 non
T3 non
p ← S4 : - p
i' ← S5 : S3 - 2
T4 non
i' ← S6 : S5 + S4
:
}

P' FIN (m' = 3)

On vérifie que :

1) x = x'
2) y = y'
3) i = i' = n

d) Soit $(d,d') \in R2$ tel que $x = x' \neq 0$ et $V \notin T$

P DEBUT (m = 2)
.....
i ← r1 : 0

P FIN (m = 3)

Pour P' il existe trois manières différentes de finir, selon la valeur de ℓ et on peut voir dans l'organigramme de la figure que pour les trois cas en arrivant à m' = 3 on a bien i' = 0.

Les conditions C2, C3 et C4 sont remplies puisque

$R1 = \{ (d,d') \mid x = x', y = y' \}$ (R composée par des relations d'égalité)

et parce que les domaines des valeurs initiales Do et Do' sont identiques ; par conséquent pour n'importe quel $do \in Do$ $\exists d'o$ tel que $(do, do') \in R1$

et de même : $\forall do' \in Do'$ $\exists do$ tel que $(do, d\flat) \in R1$

Aussi la relation réciproque (ou inverse) R^{-1} est à valeur unique (R est constitué par des relations d'égalité binaires).

Nous avons montré d'une manière générale la procédure de vérification qui sera utilisée par la suite sur la structure de la S- Machine pour étudier la réalisation correcte du microprogramme.

- Etude sur la S- Machine -

La S- Machine, présentée au chapitre I en langage de définition et en mode d'organigramme, est le système sur lequel nous allons montrer l'application de la procédure de vérification.

Cette machine hypothétique ne présente pas des possibilités d'entrée /sortie et sa microprogrammation est verticale ; d'autre part il existe une limitation à un seul niveau d'adressage indirect. Ces limitations font de la S- Machine une structure inintéressante pour des réalisations pratiques ; cependant elle garde l'allure des systèmes réels et en conséquence sa simplicité la rend intéressante pour des études théoriques comme celle que nous avons entreprise.

Dans les descriptions des deux niveaux de la S- Machine sont présents les éléments suivants, (qui peuvent être considérés comme des "primitives" de la structure décrite).

Niveau "spécification" S :

- Mémoire principale
- STK, Registre "pointeur de pile"
- X, Registre d'index
- CC, Registre compteur ordinal
- SW, Interrupteur de marche/arrêt

Ces éléments sont ceux qui sont accessibles par l'utilisateur (ou programmeur) du système.

Niveau "réalisation microprogrammée" MS :

- Mémoire principale (MEM)
- Mémoire de microprogramme (CS)
- STK
- CC
- SW
- MDR (Registre de données mémoire)
- MAR (Registre d'adresse, mémoire)
- X
- A, B } (Registres de travail)
- IR (Registre d'instruction)
- CSAR (Registre d'adresse, mémoire de microprogramme)
- CSDR (Registre de données, mémoire de microprogramme)

En regardant les organigrammes qui décrivent les structures générales des deux niveaux (figures I-7 et I-13. au premier chapitre), on trouve une première différence en ce qui concerne la présence du test de la valeur de SW. Il se trouve en début de la boucle dans la description du niveau "spécification" mais il n'apparaît pas dans celle du niveau "réalisation microprogrammée".

On pourrait penser, par conséquent, que l'implémentation microprogrammée n'a pas de dispositif d'arrêt ; cependant une meilleure analyse montre que ce test est présent d'une manière explicite dans la description de la partie "séquenceur" représentée dans la figure I-10. La description du séquenceur montre que quand SW devient zéro (après une instruction STOP) au début du cycle d'exécution de la microinstruction suivante, le test de SW, qui dans ce niveau est fait pour chaque microinstruction, produit l'arrêt de l'exécution. La différence, donc, est que le test de SW est fait une fois pour chaque instruction dans le niveau "S", tandis que dans le niveau "μS", il est fait pour chaque microinstruction.

Une autre différence qui apparaît au premier abord est la manière dont l'incrémentation du compteur ordinal est faite dans chacun des deux niveaux. Dans le niveau "S", l'incrémentation de CC est faite pendant l'exécution de chaque instruction, exception faite des instructions de branchement (au cas où le branchement est réalisé). Dans "μS", l'incrémentation de CC est réalisée juste après la recherche d'instruction et par conséquent pour toutes les instructions.

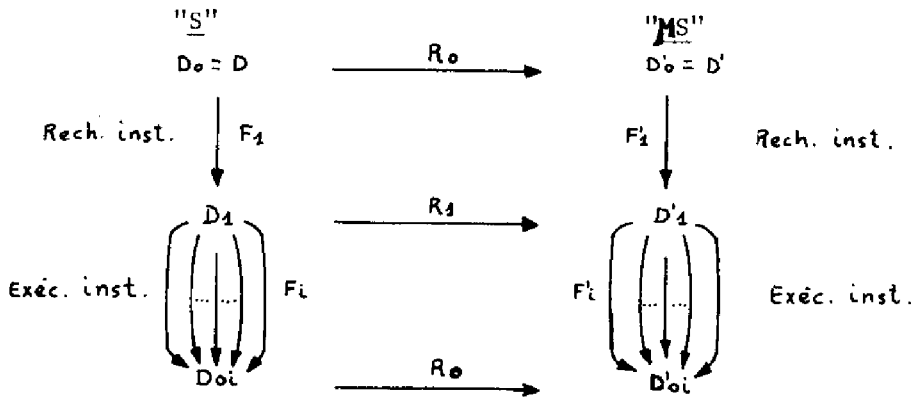
Etant donné que l'égalité de la valeur de CC dans les deux niveaux doit se vérifier en fin d'instruction et avant la recherche de l'instruction suivante, le moment où l'on fait l'incrémentation de CC, pendant le déroulement de chaque instruction, est indifférent.

D'après ce qui a été montré au paragraphe II-1-2-2., pour vérifier que la condition C1 (condition de simulation "faible") est satisfaite par une relation de simulation entre les deux niveaux "S" et "μS" de la S-Machine, nous pouvons diviser les organigrammes de ces deux niveaux en deux parties :

1) Recherche et construction de l'instruction.

2) Exécution de l'instruction.

Ceci est représenté dans la figure suivante :



Les fonctions F1 et F1' sont communes à toutes les instructions dans chaque niveau correspondant et par conséquent il est intéressant de remarquer que si Ro, R1, Do et Do' sont tels que la condition C1 est satisfaite pour une instruction quelconque, on peut affirmer qu'elle sera aussi satisfaite pour les autres instructions, quelle que soit la séquence. De cette manière, une fois vérifiée cette première partie de recherche d'instruction (F1, et F'1), l'étude de vérification se réduit à la partie "exécution" de chaque instruction.

La représentation du déroulement des actions des programmes en forme d'organigramme qui décrivent les deux niveaux de la S- Machine, sera présentée par la suite à la manière d'une interprétation symbolique. Cette interprétation symbolique est caractérisée par des variables qui reçoivent comme valeurs initiales des constantes symboliques, c'est-à-dire de valeur non spécifiée. Ainsi, par exemple, si au cours de l'exécution d'un programme on trouve des assignations de constantes symboliques, la valeur assignée est une expression comportant des opérateurs et des constantes symboliques au lieu d'une valeur du domaine correspondant.

Par exemple si B est une variable qui possède une valeur symbolique $B avant l'exécution de l'assignation B←B + 1, après celle-ci B aura la

valeur $B + 1.

L'exécution symbolique a l'avantage de pouvoir être automatisée. Pour nous, l'intérêt principal est dans le fait qu'elle fournit une manière simple et formelle de suivre le déroulement de l'exécution, sans nécessité d'affecter des valeurs concrètes.

Soit M : $\{0,1,\ldots,n\}$ l'ensemble des points de marquage des programmes et soient r et s des états de "S" et "MS" respectivement. Les domaines des états possibles des programmes "S" et "MS" sont définis par les expressions suivantes :

$$D = \{ M, r \mid SW(r) = 1 \quad V \quad SW(r) = 0 \}$$

$$D' = \{ M', s \mid SW(s) = 1 \; V \; SW(s) = 0 \}$$

Le symbole V représente le concept de "ou exclusif". Ces deux expressions qui montrent les deux possibilités de l'état de la machine : marche (SW = 1) et arrêt (SW = 0), englobent,en ce qui concerne les états pour lesquels SW = 1, les sous-domaines suivants :

- Pour le point de marquage 0 correspondant aux états initiaux de chaque instruction :

$$D_0 = \{0, r \mid CC(r) \geq (32P2)T1 \; ; \; STK(r) \leq (32P2)T2*24 \}$$

$$D'_0 = \{0, s \mid CC(s) \geq (32P2)T1 \; ; \; STK(s) \leq (32P2)T2*24 \; ; \; CSAR(s) = (32P2)T1 \}$$

La seule valeur spécifiée d'une manière précise est celle du registre d'adresse de la mémoire du microprogramme, puisque dans la réalisation microprogrammée la partie "recherche d'instruction" débute par la première microinstruction du microprogramme.

- Pour le point de marquage 1 correspondant aux états possibles de la machine après la partie "recherche et construction d'instruction" et

juste avant la partie "exécution" de celle-ci :

$$D_1 = \{1, r \mid cc(r) \geq (32\,P2)T1 \; ; \; STK(r) \leq (32\,P2)T2*24 \; ;$$
$$POP(r) = 6 \; ; \; AD(r) < (24\,P2)T2*24 \}$$

$$D'_1 = \{1, \lambda \mid cc(\lambda) > (32\,P2)T1 \; ; \; STK(\lambda) \leq (32\,P2)T2*24 \; ;$$
$$PIR(\lambda) = S \; ; \; B(\lambda) < (24\,P2)T2*24 \}$$

La Relation de simulation entre les niveaux "S" et "MS" est obtenue par l'union de Ro, relation de simulation qui doit être vérifiée au point de marquage "zéro" et R1, relation de simulation qui doit être vérifiée au point de marquage "un".

Ainsi, $R = Ro \cup R1$

$$Ro = \{(m, m') \mid m = m' = 0 \; ; \; (r, \lambda) \mid r \in D \; ; \; \lambda \in D' \; ; \; MEM(r) = MEM(\lambda) \; ;$$
$$CC(r) = cc(\lambda) \; ; \; STK(r) = STK(\lambda) \; ;$$
$$X(r) = X(\lambda) \; ; \; SW(r) = SW(\lambda) \}$$

$$R1 = \{(m, m') \mid m = m' = 1 \; ; \; (r, \lambda) \mid r \in D \; ; \lambda \in D' \; ; \; MEM(r) = MEM(\lambda) \; ;$$
$$STK(r) = STK(\lambda) \; ; \; X(r) = X(\lambda) \; ;$$
$$AD(r) = B(\lambda) \; ; \; OP(r) \; \alpha \; IR(\lambda) \}$$

Le symbole α exprime une relation entre OPE et IR qui n'est pas la relation d'égalité à cause du mode d'adressage particulier de la partie du microprogramme correspondant à chaque instruction.

Cette relation est telle que :

$$Si \quad 2 \perp OP < 14 \quad , \quad OP = (2\,PIR[1], IR)[26]$$

$$Si \quad 2 \perp OP > 31 \quad , \quad OP = (1, 0, IR)[26]$$

et vérifie que les programmes "S" et "MS" choisissent la partie "exécution" correspondant à la même instruction (après la partie "recherche d'instruction").

L'application de F1 et F'1 (correspondant à la "recherche et cons-

truction d'instruction") donne en représentation symbolique :

```
        S                                    MS
    (m=0)                                (m'=0)
M←P1:MEM[2⊥CC[8+ι24];]              MAR←Q1:CC
   P2:M[1]                          MDR←Q2:MEM[MAR;]
   P3:M[2]                           CC←Q3:(32ρ2)⊤(2*32)|1+2⊥Q1
OP ←P4:M[2+ι6]                        B←Q4:(8ρ0,24ρ1)∧Q2
AD←P5:M[8+ι24]                        A←Q5:~(8ρ0,24ρ1)
    (m=1)                           MDR←Q6:Q5∧Q2
                              (*)    IR←Q7:(Q6,0)[1+ι5]
                                    (m'=1)
```

(comme dans un exemple antérieur pi et qi sont des paramètres intermédiaires)

Sur cette représentation symbolique on vérifie que partant des états r et s de "S" et "MS" tels que Ro est satisfaite, en arrivant au point de marquage 1 la relation R1 est satisfaite aussi. Cependant en ce qui concerne le chargement du registre IR (repéré par (*) dans l'exécution symbolique de MS), nous sommes en désaccord avec la définition donnée par Birman. En effet, pour que celle-ci soit correcte, il faudrait supposer que, dans la figure I-3. qui montre le schéma du chemin des données de la S- Machine, le registre IR est connecté aux fils 4, 5, 6, 7 et 8 du BUS-OUT. Si, par contre, on avait défini la connexion de IR aux fils 5, 6, 7, 8, 9, le décalage réalisé sur MDR en (*) ne serait pas nécessaire.

On vérifie donc que :

$$\forall (r,s) \in R_0 \quad , \quad (F_1(r), F'_1(s)) \in R_1$$

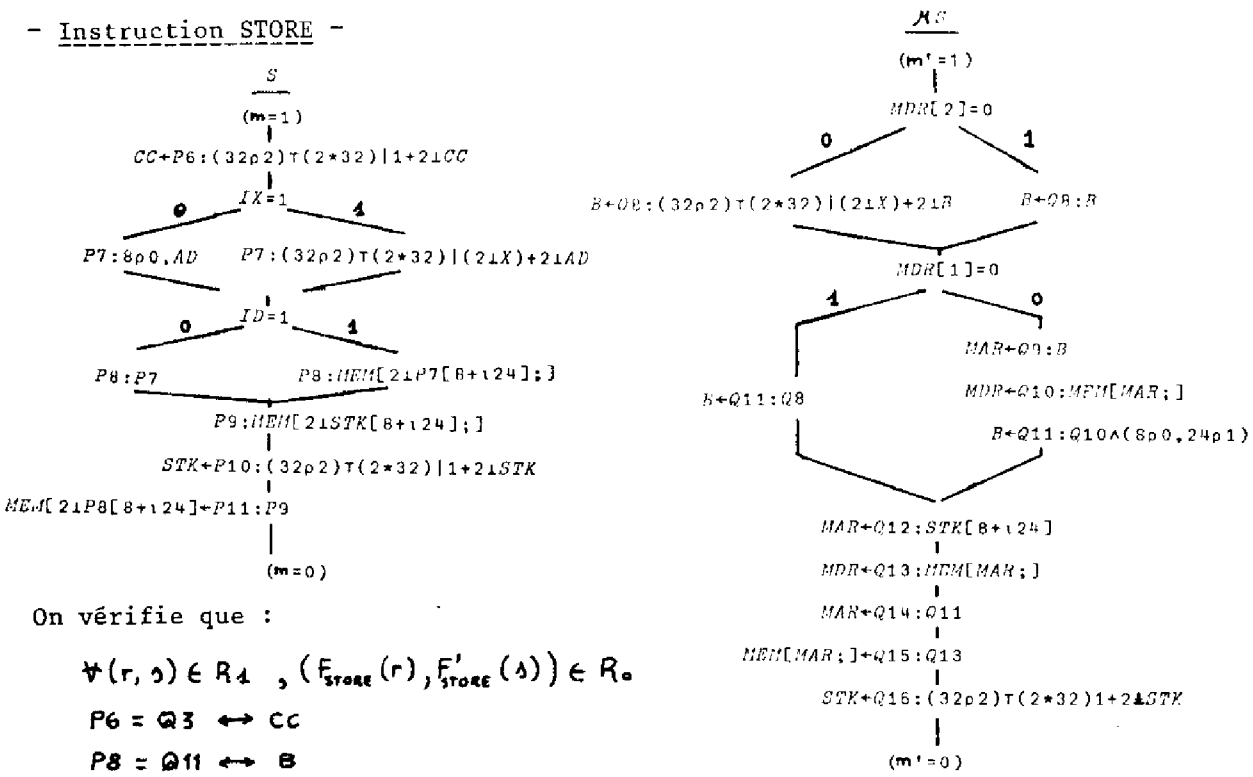
ce qui satisfait la condition C1.

La suite de la procédure consiste à vérifier que la condition C1 est satisfaite après l'application des fonctions Fi et F'i à des états machine des domaines D1 et D'1.

Cette vérification ne présente pas de problèmes particuliers pour

la plupart des instructions et par conséquent nous n'allons montrer ici que la vérification d'une d'entre elles considérée comme typique. Il s'agit de l'instruction STORE.

- Instruction STORE -



On vérifie que :

$\forall (r, s) \in R_1 \ , (F_{STORE}(r), F'_{STORE}(s)) \in R_0$

P6 = Q3 ↔ CC

P8 = Q11 ↔ B

P9 = Q13 ↔ MEM[STK;]

Avec ce qui a été montré pour F1 et F'1 on peut conclure que la réalisation microprogrammée de l'instruction STORE est une simulation de la spécification de cette instruction.

Les conditions C2, C3 et C4 sont satisfaites, et ceci ne nécessite pas de démonstration puisque comme on l'a déjà vu dans un exemple précédent la relation de simulation R est composée de relations binaires d'égalité.

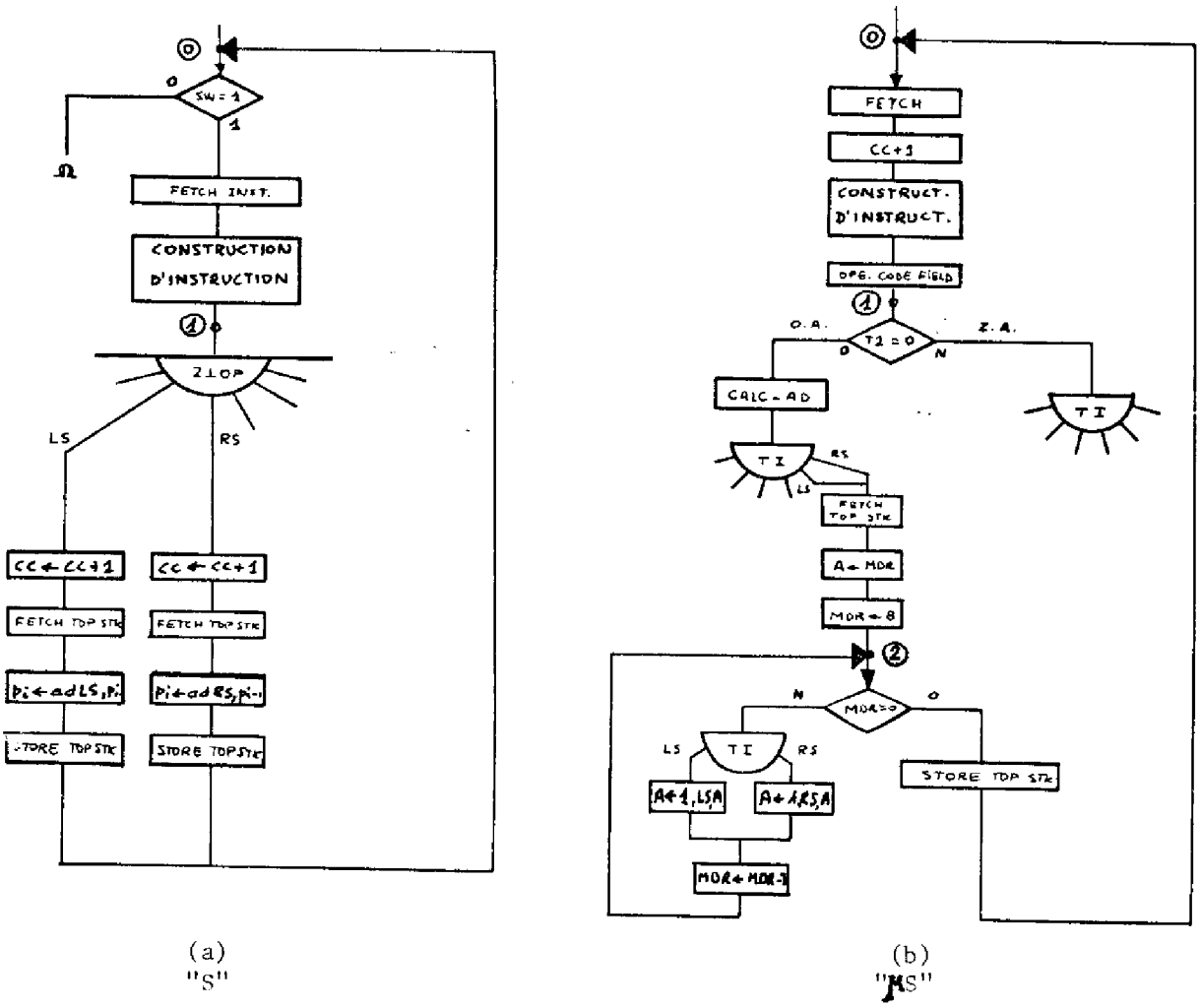
- Instructions LS et RS -

L'étude de ces deux instructions requiert une attention spéciale puisque leur réalisation microprogrammée comporte une boucle qui exécute un décalage d'un bit par passage et il y aura, donc, un nombre de passages dans

la boucle identique au nombre spécifié dans la partie adresse de l'instruction.

Pour faciliter l'étude nous proposons un rapprochement des organigrammes correspondant à ces deux instructions en tenant compte de la propriété de transitivité de la relation de simulation. Ce rapprochement consiste à modifier l'organigramme décrivant l'instruction LS ou RS dans le niveau "spécification" jusqu'à ce que sa structure soit la plus semblable possible à celle du niveau "réalisation microprogrammée" et ceci de telle manière que la nouvelle structure simule celle du niveau "spécification" (20).

Les organigrammes, correspondant à ces deux instructions, pour chacun des deux niveaux sont montrés par la figure II-5.



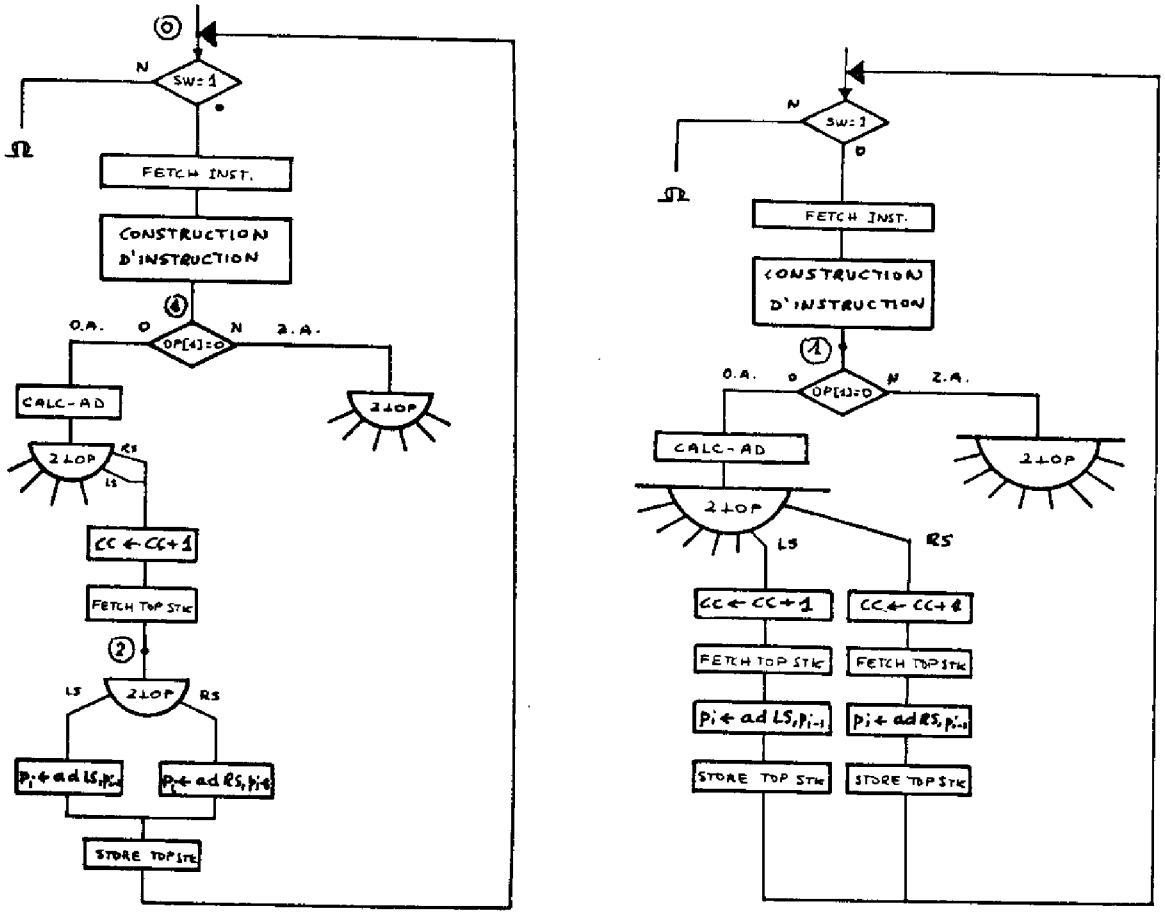
(a) "S"

(b) "MS"

Figure II-5

Les figures suivantes II-6 (a) et II-6 (b) montrent les variations successives de "S" : S' et S".

Pour vérifier que "S'" simule "S_{LSVRS}", c'est-à-dire la partie de S qui concerne LS et RS, il suffit de dérouler les programmes en interprétation symbolique et de vérifier que les relations Ro et Rl sont validées.



" S" "
(b)

" S' "
(a)

Figure II-6

De la même manière il est facile de vérifier que " S" " simule " S' " par rapport aux mêmes relations Ro et Rl ; et donc, si les expressions S & S' et

S' & S" sont vérifiées, le problème est de montrer que S" & μS_{LSVRS}.

Ces deux organigrammes S" et μS_{LSVRS} ne diffèrent essentiellement que par la manière de réaliser le décalage. Dans S" il est effectué directement et pour le nombre de bits spécifié. Dans μS où la description des opérations doit tenir compte des possibilités du matériel, ce décalage est fait en plusieurs passages par l'Unité Arithmétique et Logique qui ne peut faire que le décalage d'un seul bit à chaque cycle.

L'interprétation symbolique des programmes abstraits des figures II-5 (b) et II-6 (b), correspondant à μS_{LSVRS} et S" respectivement est :

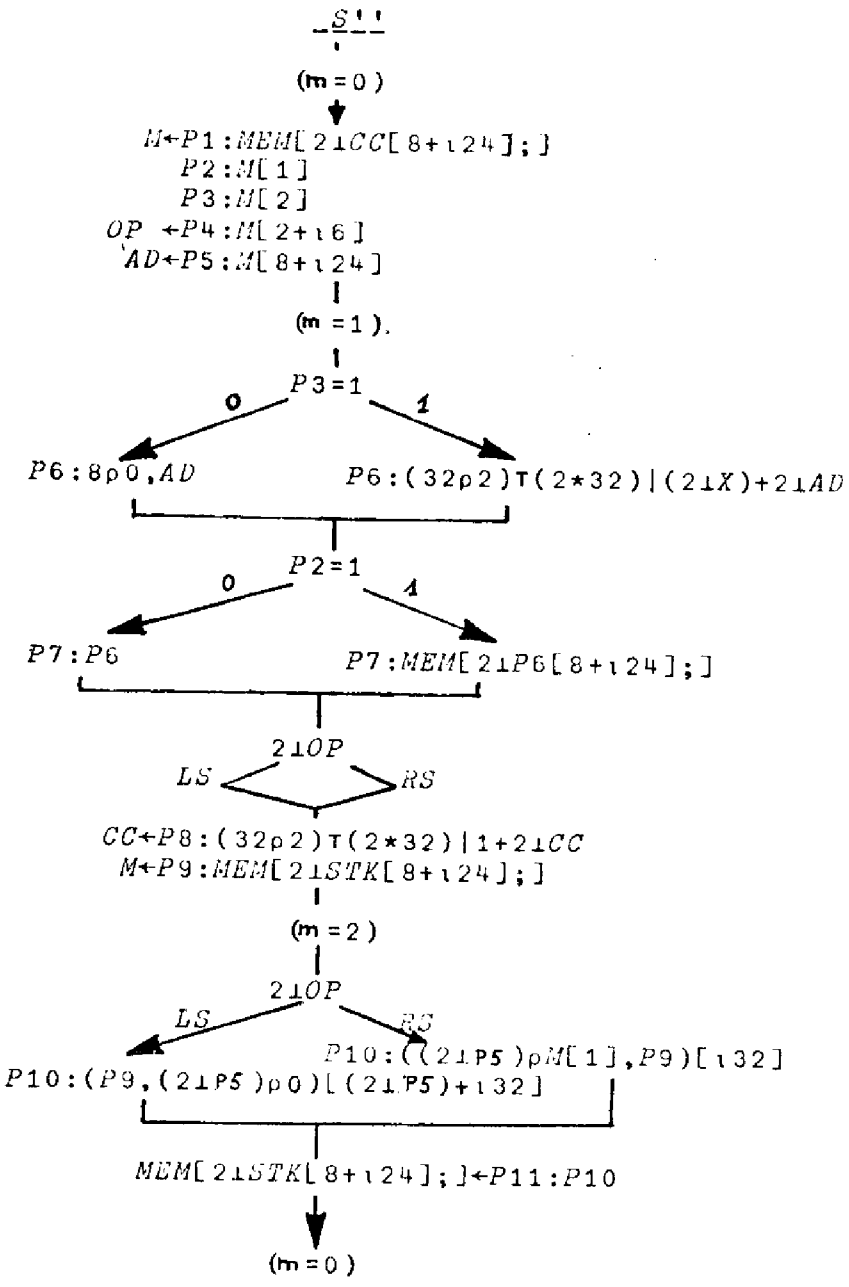


Figure II-7

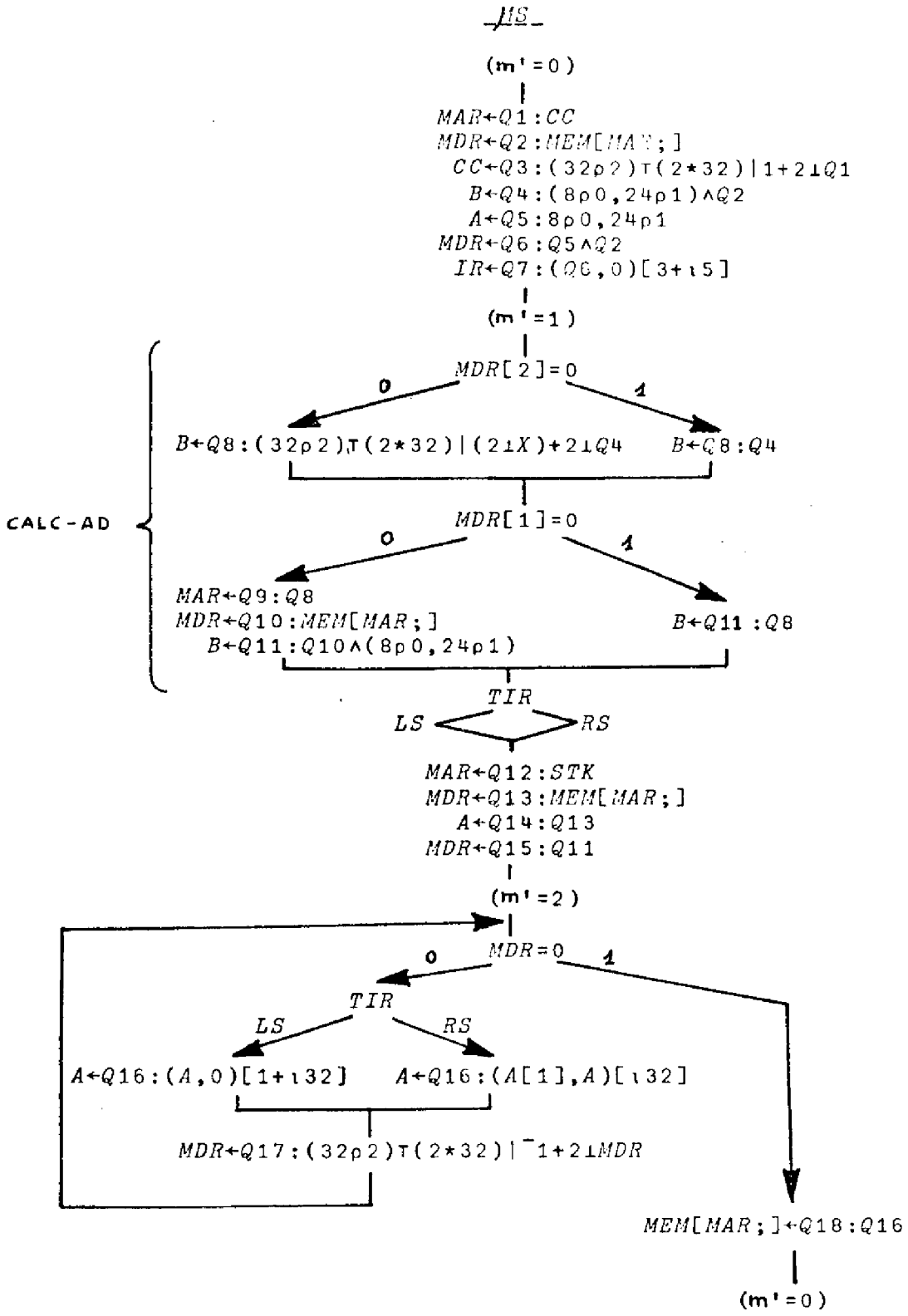

MS
(m'=0)
MAR←Q1:CC
MDR←Q2:MEM[MAR;]
CC←Q3:(32ρ2)⊤(2*32)|1+2⊥Q1
B←Q4:(8ρ0,24ρ1)∧Q2
A←Q5:8ρ0,24ρ1
MDR←Q6:Q5∧Q2
IR←Q7:(Q6,0)[3+ι5]
(m'=1)
MDR[2]=0
0
1
B←Q8:(32ρ2)⊤(2*32)|(2⊥X)+2⊥Q4
B←Q8:Q4
CALC-AD
MDR[1]=0
0
1
MAR←Q9:Q8
MDR←Q10:MEM[MAR;]
B←Q11:Q10∧(8ρ0,24ρ1)
B←Q11:Q8
TIR
LS
RS
MAR←Q12:STK
MDR←Q13:MEM[MAR;]
A←Q14:Q13
MDR←Q15:Q11
(m'=2)
MDR=0
0
1
TIR
LS
RS
A←Q16:(A,0)[1+ι32]
A←Q16:(A[1],A)[ι32]
MDR←Q17:(32ρ2)⊤(2*32)|¯1+2⊥MDR
MEM[MAR;]←Q18:Q16
(m'=0)

Figure II-8

On vérifie, en suivant le déroulement symbolique de S" et $\mathcal{MS}_{LSVRS}$, que les états résultant de l'application de F1 et F'1 ainsi que F2 et F'2 satisfont les expressions suivantes :

$$\forall r \in D_0, \forall \lambda \in D'_0 \quad (F_1(r), F'_1(\lambda)) \in R$$

$$\forall r \in D_1, \forall \lambda \in D'_1 \quad (F_2(r), F'_2(\lambda)) \in R$$

La relation R étant $R = R_0 \cup R_1 \cup R_2$

où

$$R_2 = \left\{ \begin{matrix} m, m' \mid m = m' = 2 \\ r, \lambda \mid r \in D ; \lambda \in D' ; \; MEM(r) = MEM(\lambda) ; \\ STK(r) = STK(\lambda) ; \; X(r) = X(\lambda) \; ; \\ AD(r) = B(\lambda), \; OP(r) \propto Ie(\lambda) \end{matrix} \right\}$$

Pour conclure la vérification de la simulation entre S" et $\mathcal{MS}_{LSVRS}$ il faut montrer l'équivalence de l'exécution symbolique entre le point de marquage ② et le ⓪ de chacun des deux programmes.

Pour faire ceci, nous allons dérouler la boucle qui apparaît au point de marquage ② de MS, comme s'il s'agissait d'un organigramme en "ligne droite". Dans la figure II-9, nous représentons en :

b) le déroulement de la boucle en $\mathcal{MS}$.

a) on montre le développement de l'expression qui représente le décalage de (ad) bits dans S". pi ← ad LS, pi-1 ou pi ← ad RS, pi-1.

Etant donné que cette manière de représentation n'est pas formelle, à cause surtout de l'indéfinition qu'elle présente, elle ne se prête pas facilement à une étude de vérification systématisée. C'est à cause de ce problème que nous nous sommes dirigés vers d'autres méthodes mieux adaptées au traitement des boucles. Nous les présentons au paragraphe suivant.

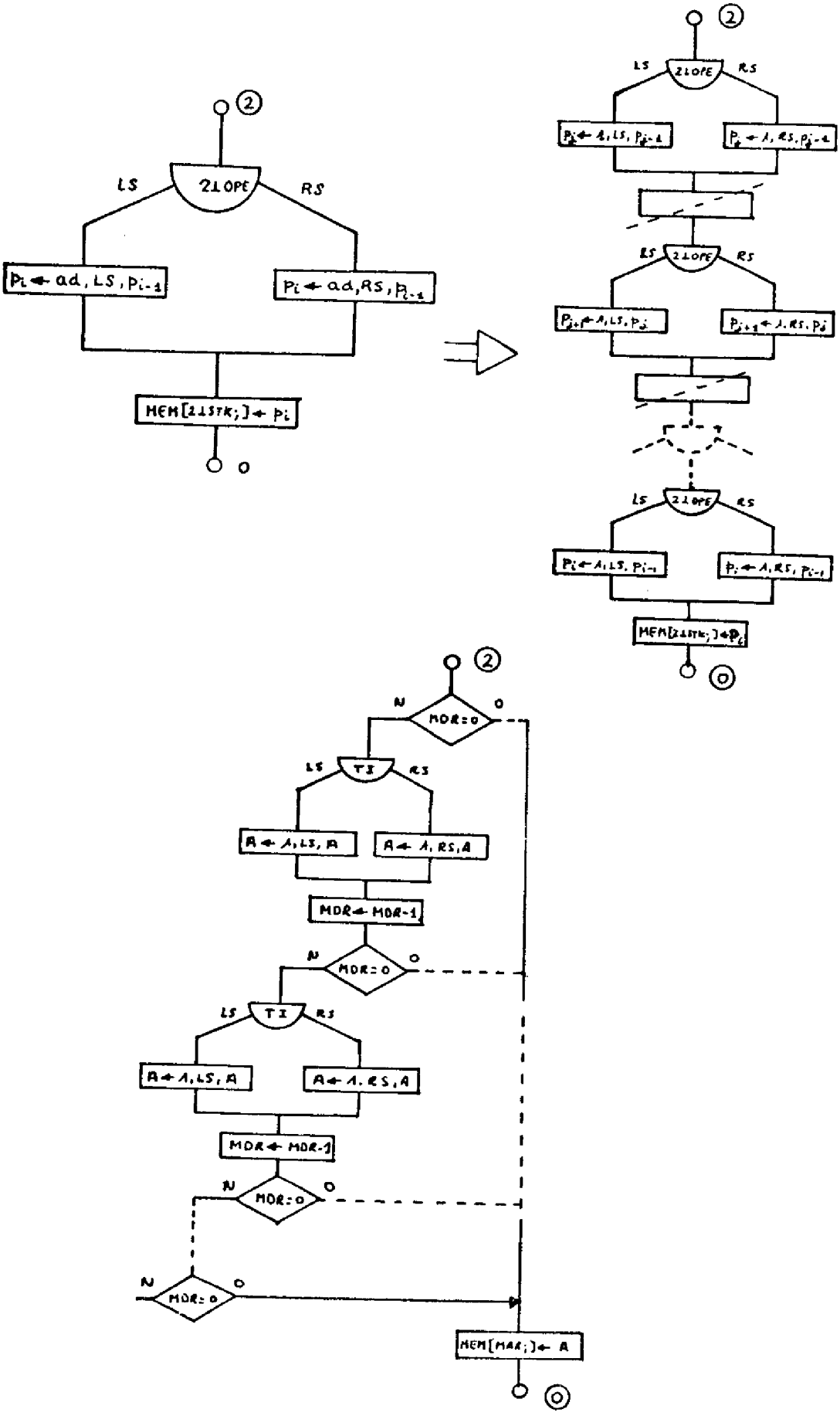

2
LS
2 L OPE
RS
Pi ← ad, LS, Pi-1
Pi ← ad, RS, Pi-1
MEM[2 L STK;] ← Pi
0
Pi ← A, LS, Pi-1
Pi ← A, RS, Pi-1
MEM[2 L STK;] ← Pi
N
MDR=0
O
TI
A ← A, LS, A
A ← A, RS, A
MDR ← MDR-1
MEM[MAR;] ← A

Figure II-9

II-2 CAS DE MICROPROGRAMMES CONTENANT DES BOUCLES : UTILISATION DE LA METHODE DES ASSERTIONS INVARIANTES

Le fait d'avoir une boucle dans la description du niveau "réalisation microprogrammée" et pas dans celle du niveau "spécification" nous a obligé à faire une vérification du programme comportant la boucle avant de comparer les résultats avec ceux de la description du niveau "spécification". Pour réaliser cette vérification, nous utilisons la méthode de Floyd.

II-2-1. Méthode de FLOYD

Avant de procéder à l'étude concrète de la boucle dans la partie du microprogramme correspondant aux instructions de décalage, nous allons présenter un résumé de la méthode de Floyd pour la vérification de programmes.

La notation que nous employons dans cette partie est celle utilisée par Mauna (16).

Nous tenons compte d'une classe simple d'organigrammes où l'on distingue trois types de variables (groupées comme des vecteurs).

- Vecteur des variables d'entrée $\bar{x} = (x1, x2, \ldots, xn)$

Ce sont des valeurs d'entrée et, par définition, elles ne changent pas de valeur pendant l'exécution du programme.

- Vecteur des variables intermédiaires (ou variables de programme) $\bar{y} = (y1, y2, \ldots, ym)$, utilisé comme enregistrement temporel des valeurs pendant l'exécution du programme.

- Vecteur des variables de sortie $\bar{z} = (z1, z2, \ldots, zp)$ qui contient les valeurs des variables de sortie quand le calcul est terminé.

Parallèlement on considère trois types de domaines non vides :

- Domaine d'entrée : $D\bar{x} : Dx1 \times Dx2 \times \ldots \times Dxn$

- Domaine des variables intermédiaires : $D\bar{y} = Dy1 \times Dy2 \ldots \times Dym$

- Domaine de sortie : $D\bar{z} = Dz1 \times Dz2 \times \ldots \times Dzp$

La vérification d'un programme est basée sur l'utilisation des deux prédicats suivants :

- Un prédicat $\varphi(\bar{x})$ sur $D\bar{x}$ nommé prédicat d'entrée qui définit les éléments de $D\bar{x}$ qui sont valides comme variables d'entrée. Dans le cas où tous les éléments de $D\bar{x}$ seraient acceptés par le programme, alors $\varphi(\bar{x})$ serait VRAI.

- Un prédicat $\Psi(\bar{x}, \bar{z})$ sur $D\bar{x} \times D\bar{z}$, dit prédicat de sortie qui indique les relations entre les variables d'entrée et de sortie qui doivent être satisfaites après l'exécution du programme.

A partir de ces éléments, on définit les points principaux de la vérification d'un programme P.

1) P <u>termine</u> sur φ si pour chaque entrée :

$$\bar{\xi} \mid \varphi(\bar{\xi}) = \text{VRAI}$$

le déroulement du programme termine.

2) P est <u>partiellement correct</u> par rapport à φ et Ψ, si pour chaque entrée :

$$\bar{\xi} \mid \varphi(\bar{\xi}) = \text{VRAI}$$

et que le programme termine, $\Psi(\bar{\xi}, P(\bar{\xi})) = \text{VRAI}$

3) P est <u>totalement correct</u> par rapport à φ et Ψ

si pour chaque $\bar{\xi} \mid \varphi(\bar{\xi}) = \text{VRAI}$, le déroulement du programme termine et $\Psi(\bar{\xi}, P(\bar{\xi})) = \text{VRAI}$.

Pour pouvoir étudier ces trois points sur un organigramme qui comporte des boucles, nous utilisons la méthode des assertions invariantes (ou inductives) de Floyd (6) qui peut être énoncée comme suit :

♦ Pour un organigramme donné P avec un prédicat d'entrée $\varphi(\bar{x})$ et un prédicat de sortie $\Psi(\bar{x}, \bar{z})$ on applique les étapes suivantes :

a) Couper les boucles (c'est-à-dire : introduire un point de marquage sur la boucle).

b) Trouver un ensemble approprié d'assertions invariantes.

c) Construire les conditions de la vérification et si toutes les conditions sont satisfaites alors P est partiellement correct par rapport à φ et ψ .

La construction de ces conditions de vérification est faite en tenant compte dans chaque chemin, entre deux points de marquage de l'organigramme, des actions et des conditions qui apparaissent et en allant dans le sens contraire à celui de l'exécution.

La figure II-10 montre sur un exemple la manière de construire les conditions de vérification.

Une manière de généraliser ceci est la suivante :

Soit $R_\alpha(\bar{x}, \bar{y})$ la condition qui doit être vérifiée pour que le chemin α soit parcouru (ce chemin α étant délimité par les points de marquage i et j).

Soit $r_\alpha(\bar{x}, \bar{y})$ la transformation subie par les variables $\bar{y}$ tout au long du chemin α .

Soit M l'ensemble des points de marquage.

- Les assertions $\{ p_i(\bar{x}, \bar{y}) \mid i \in M \}$ sont des assertions invariantes de P par rapport à φ si :

a) Pour chaque chemin α délimité par les points i et j tels que i soit le point de marquage correspondant au DEBUT du programme :

$$\forall \bar{x} \left[\varphi(\bar{x}) \wedge R_\alpha(\bar{x}) \supset p_j(\bar{x}, r_\alpha(\bar{x})) \right]$$

b) Pour chaque chemin délimité par les points i et j :

$$\forall \bar{x} \forall \bar{y} \left[p_i(\bar{x}, \bar{y}) \wedge R_\alpha(\bar{x}, \bar{y}) \supset p_j(\bar{x}, r_\alpha(\bar{x}, \bar{y})) \right]$$

Les expressions ci-dessus sont les conditions de vérification.

La partie a) se rapporte au chemin initial de l'organigramme où les variables $\bar{y}$ sont initialisées par des variables $\bar{x}$ ou par des fonctions de $\bar{x}$ seulement et non de $\bar{y}$.

Il faut noter que pour construire la condition de vérification du chemin de FIN du programme le prédicat de sortie peut être considéré de la même manière que les assertions invariantes.

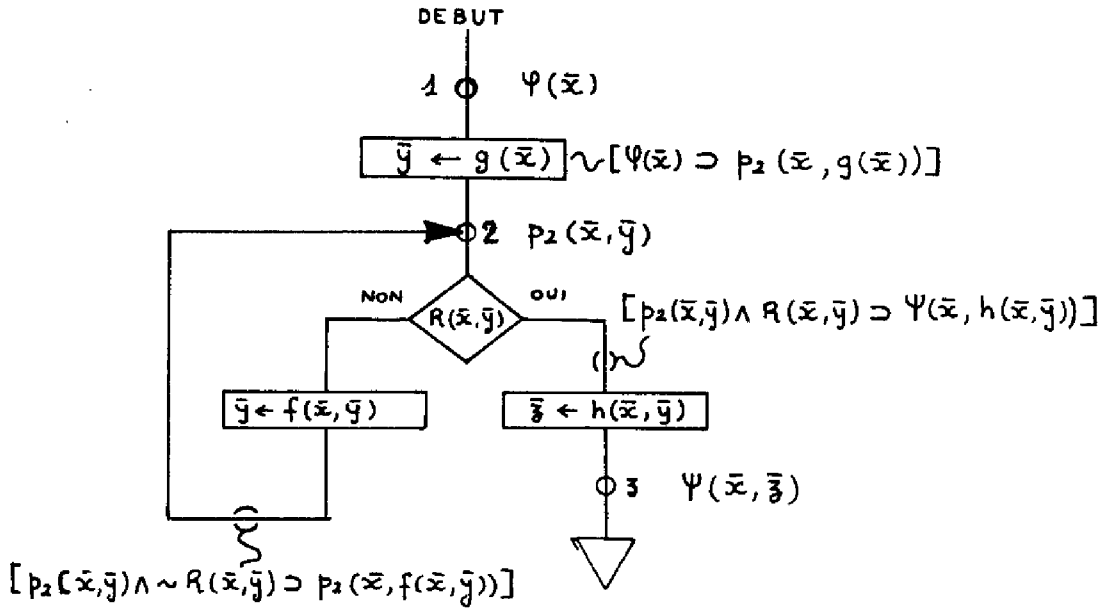


Figure II-10

TERMINAISON -

Pour pouvoir affirmer que P est totalement correct par rapport à φ et ψ, il faut vérifier que, pour toutes les valeurs d'entrée qui satisfont le prédicat φ, le programme termine.

La méthode de vérification de la terminaison d'un programme est quelque peu différente de la précédente et nous allons l'exposer séparément.

Cette méthode repose sur l'existence d'une relation, par l'intermédiaire de certaines fonctions appliquées aux variables dans les points de coupure des boucles, entre l'exécution du programme et un ensemble ordonné qui contient des séquences décroissantes finies.

L'ensemble ordonné le plus courant est celui des nombres entiers naturels, ordonnés par la relation $>$(plus grand que). Etant donné que pour n'importe quel nombre entier naturel n on a : $n > n-1 > \ldots > 2 > 1 > 0$ il n'existe pas de séquences décroissantes de longueur infinie.

Le nom employé par Floyd pour désigner ce type d'ensembles est celui d'"ensembles bien fondés".

En général, tous les ensembles "bien fondés" peuvent être utilisés dans le but de prouver la terminaison d'un programme.

DEFINITION -

Un ensemble partiellement ordonné $(\mathbb{D}, \prec)$ est un ensemble non vide avec une relation binaire $\prec$ définie sur les éléments de l'ensemble et qui satisfait les propriétés suivantes :

a) $\forall a \forall b \forall c \in \mathbb{D}$ si $a \prec b$ et $b \prec c$ alors $a \prec c$ (transitivité)

b) $\forall a \forall b \in \mathbb{D}$, si $a \prec b$, $b \not\prec a$ (antisymétrie)

c) $\forall a \in \mathbb{D}$, $a \not\prec a$ (non réflexivité)

On dit "ensemble partiellement ordonné" car la relation d'ordre n'est pas nécessairement définie pour tous les couples d'éléments : il peut exister des éléments a, b, pour lesquels ni $a \prec b$ ni $b \prec a$ n'est accomplie.

D'une manière générale, un ensemble partiellement ordonné $(\mathbb{D}, \prec)$ est dit "bien fondé" s'il contient des séquences finies de la forme :

$$a_0 \succ a_1 \succ a_2 \succ \cdots \quad , a_i \in \mathbb{D}$$

Par exemple l'ensemble de nombres entiers avec la relation d'ordre $<$ est partiellement ordonné mais il n'est pas "bien fondé", (il existe des séquences infinies comme :

$$0 > -1 > -2 > \ldots\ldots \quad)$$

La procédure simplifiée pour établir l'existence de la terminaison dans un programme comporte les étapes suivantes :

1) Choix d'un ensemble ordonné "bien fondé"

$$(\Phi, \prec)$$

2) Sélection d'un ensemble de points de marquage dans l'organigramme.

3) Dans le point de marquage des boucles, associer une fonction fi $(\bar{x}, \bar{y})$ qui applique $D\bar{x}$ X $D\bar{y}$ dans Φ , c'est-à-dire que :

$$\forall \bar{x} \forall \bar{y} \; [\, f_i(\bar{x}, \bar{y}) \in \Phi \,]$$

Si pour le chemin α délimité par les points i et j et qui forme partie de la boucle, l'expression :

$$[2] \qquad \forall \bar{x} \forall \bar{y} \; [\, \varphi(\bar{x}) \wedge R_\alpha(\bar{x}, \bar{y}) \supset f_i(\bar{x}, \bar{y}) \succ f_i(\bar{x}, r_\alpha(\bar{x}, \bar{y})) \,]$$

est vérifiée, alors P termine sur φ.

II-2-2. Application aux instructions LS et RS de la S-Machine

Nous avons montré dans la section II-1-2 que l'existence d'une boucle dans la réalisation microprogrammée des instructions LS et RS entraîne une mauvaise définition de l'interprétation symbolique qui rend difficile l'application de la procédure de vérification de la simulation entre "S" et "MS". Pour résoudre ce problème, nous allons appliquer la méthode exposée précédemment à la partie de l'organigramme de "MS" qui comporte la boucle.

Pour mieux montrer la manière dont cette application est effectuée

nous allons la faire sur l'organigramme de la figure II-11. qui est équivalent à celui de la figure II-5 (b) entre ② et ⓪

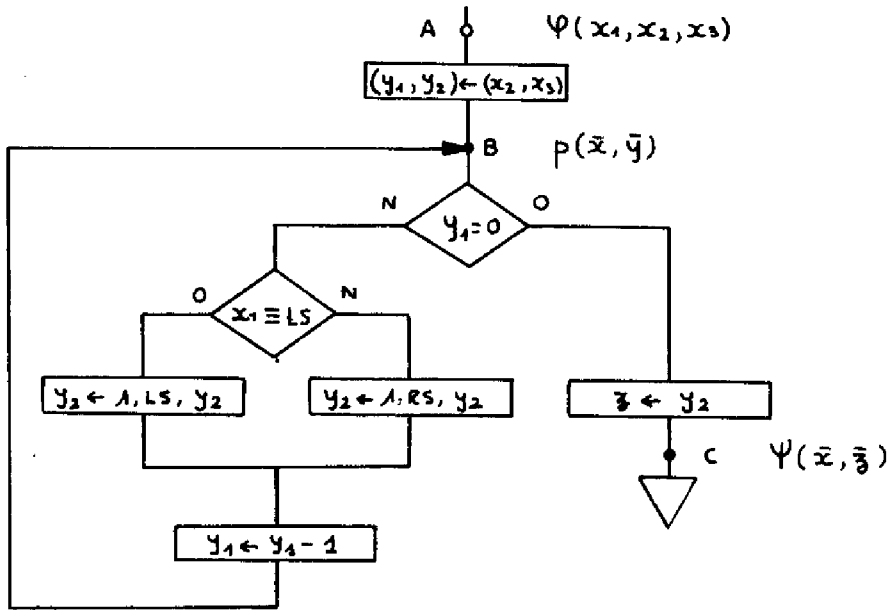


Figure II-11

Le changement de notation par rapport à la figure II-5 (b), est fait pour simplifier l'écriture en la rapprochant de celle qui a été utilisée pour exposer la méthode de Floyd.

Les équivalences sont (avec les variables de l'interprétation symbolique) :

x_1 = 2 ⊥ IR (code opération)
x_2 = 2 ⊥ MDR
x_3 = Q14
z = MEM [MAR;]

L'application de la méthode est présentée séparément pour LS et RS.

Instruction LS -

Les prédicats d'entrée et sortie sont :

$$\varphi(x_1, x_2, x_3) : \; x_1 = 24 \,,\; 0 \leq x_2 < 2^{32} \,,\; 0 \leq 2 \perp x_3 < 2^{32}$$

$$\psi_{LS}(\bar{x}, \bar{z}) \; : \; (x_3 \,,\, x_2 \,\rho\, 0)\,[x_2 + i\,32] = z$$

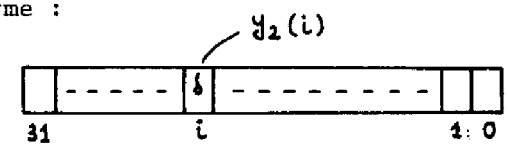
L'assertion invariante associée au point de coupure de la boucle B est orientée au niveau du bit ce qui veut dire que nous allons substituer l'expression de la figure II-11

$$y2 \leftarrow 1,\ LS,\ y2$$

par la suivante qui exprime aussi un décalage d'un bit :

$$\begin{cases} y2\ (i) \leftarrow y2\ (i-1),\ i = 1,\ 31 \\ y2\ (o) \leftarrow 0 \end{cases}$$

Dans la suite, nous tenons compte de y2 comme étant la représentation d'un registre de la forme :



Une assertion invariante au point B serait :

$$P_{LS}(\bar{x}, \bar{y}) : \left(\sum_{i=0}^{31} y_2(i) \cdot 2^i = \sum_{i=0}^{31-(x_2-y_1)} x_3(i) \cdot 2^{i+(x_2-y_1)}\right) \wedge \left(y_2(i) = x_3(i-(x_2-y_1))\right)$$

Le prédicat de sortie Ψ_{LS} orienté au niveau du bit équivalent à celui établi précédemment serait :

$$\psi_{LS}(\bar{x}, \bar{z}) : \sum_{i=0}^{31} z(i) \cdot 2^i = \sum_{i=0}^{31-x_2} x_3(i) \cdot 2^{i+x_2}$$

La construction des conditions de vérification est faite de la manière exposée dans la Section II-2-1 (cf. figure II-10).

- Chemin A - B :

$$\varphi(\bar{x}) \supset \left(\sum_{0}^{31} x_3(i) \cdot 2^{31} = \sum_{0}^{31} x_3(i) \cdot 2^{31}\right) \vee \left(x_3(i) = x_3(i-0)\right)$$

(condition évidemment vérifiée)

- Chemin B_{LS} - B_{LS} (Boucle) :

$$\left[\left(\sum_{0}^{31} y_2(i) \cdot 2^i = \sum_{0}^{31-(x_2-y_1)} x_3(i) \cdot 2^{i+(x_2-y_1)}\right) \wedge \left(y_2(i) = x_3(i-(x_3-y_1))\right) \wedge y_2 \neq 0 \wedge x_1 \equiv LS\right] \supset$$

$$\supset \sum_{0}^{31} y_2(i-1) \cdot 2^i = \sum_{0}^{31-(x_2-y_{i+1})} x_3(i) \cdot 2^{i+(x_2-y_{i+1})}$$

Nous allons transformer la deuxième partie de cette expression pour mettre en évidence que la condition est vérifiée.

$$\supset \sum_{-1}^{30} y_2(i)\cdot 2^{i+1} = 2\left(\sum_{0}^{31-(x_2-y_1)} x_3(i)\cdot 2^{i+(x_2-y_1)}\right) - 2x_3(31-(x_2-y_1))\cdot 2^{31}$$

$$\supset 2\sum_{0}^{31} y_2(i)\cdot 2^{i} = 2\,y_2(31)\cdot 2^{31} = 2\sum_{0}^{31-(x_2-y_1)} x_3(i)\cdot 2^{i+(x_2-y_1)} - 2\,x_3(31-(x_2-y_1))\cdot 2^{31}$$

(condition également remplie)

La difficulté principale de cette construction de la condition de vérification dans le chemin B-B vient du fait que le registre représenté par y2 a un nombre de bits limité. En conséquence dans les expressions qui déterminent le décalage il faut tenir compte de la perte du bit qui se trouve à l'extrémité du registre. L'expression est nettement simplifiée dans le cas hypothétique de registres de longueur illimitée.

- Chemin B - C :-

$$\left[\left(P_{LS}(\bar{x},\bar{y}) \wedge y_1 = 0\right) \supset \Psi_{LS}(\bar{x}, y_2)\right]$$

$$\left[\left(\sum_{0}^{31} y_2(i)\cdot 2^{i} = \sum_{0}^{31-(x_2-y_1)} x_3(i)\cdot 2^{i+(x_2-y_1)}\right) \wedge \left(y_2(i) = x_3(i-(x_2-y_1))\right) \wedge y_1 = 0\right] \supset$$

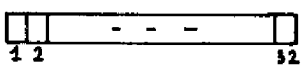
$$\supset \sum_{0}^{31} y_2(i)\cdot 2^{i} = \sum_{0}^{31-(x_2-y_1)} x_3(i)\cdot 2^{i+x_2}$$

(condition vérifiée)

Etant donné que les conditions de vérification associées à chacun des chemins de l'organigramme de la figure II-11. sont vérifiées, on en conclut que le programme représenté est partiellement correct (en ce qui concerne la partie LS) par rapport à Φ et Ψ .

Si l'on admet l'équivalence entre les expressions :

$$(y_2, 0)[\Delta+\iota 32] \equiv \begin{cases} y_2(i) \leftarrow y(i-1)\ ,\ i = \Delta, 31 \\ y_2(0) \leftarrow 0 \end{cases}$$

en tenant compte des changements d'indices. (En effet, pour l'expression APL on suppose un registre y2 de la forme : 

)

Le fait de montrer que la réalisation microprogrammée est correcte par rapport aux prédicats Φ et Ψ, implique que le décalage réalisé par ce programme est le même que celui réalisé dans le niveau "spécification" puisque

Ψ est un prédicat valable dans les deux descriptions.

Instruction RS -

Les prédicats d'entrée et sortie sont :

$\varphi(\bar{x})$: le même que pour LS

$$\Psi_{RS}(\bar{x},\bar{y}) : \sum_{0}^{31} y(i)\cdot 2^{i} = \sum_{x_2}^{31} x_3(i)\cdot 2^{i-x_2} + \sum_{1}^{x_2} x_3(31)\cdot 2^{31-x_2+i}$$

L'expression qui décrit le décalage à droite au niveau du bit serait :

y2 (i) ← y2 (i+1) , i = 0,30

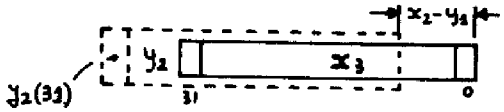


Figure II-12

à la place de celle qui dans la figure II-11. représente le décalage à droite de la forme

y2 ← 1, RS, y2

Le décalage à droite de (x2 - y1) bits réalisé sur le registre y2 dont la valeur initiale est x3 est représenté dans la figure II-12. Cette représentation aide à comprendre l'expression que nous avons choisie pour l'assertion invariante au point B :

$$P_{RS}(\bar{x},\bar{y}) : \left(\sum_{0}^{31} y_2(i)\cdot 2^{i} = \sum_{x_2-y_1}^{31} x_3(i)\cdot 2^{i-x_2+y_1} + \sum_{1}^{x_2-y_1} x_3(31)\cdot 2^{(31-x_2+y_1+i)}\right) \wedge$$

$$\wedge\ y_2(i) = x_3(i+x_2-y_1)\ ,\ \forall\, i \leq 31-x_2+y_1$$

$$\wedge\ y(i) = x_3(31)\ ,\ \forall\ i > 31-x_2+y_1$$

La construction des conditions de vérification est faite comme dans le cas de l'instruction LS : (cf : figure II-11).

- Chemin A - B :

$$\varphi(\bar{x}) \supset \sum_{0}^{31} x_3(i)\cdot 2^{i} = \sum_{x_2-x_2}^{31} x_3(i)\cdot 2^{i-x_2+x_2} + 0$$

(condition évidemment vérifiée)

- Chemin B_{RS} - B_{RS} (Boucle) :

$$[P_{RS}(\bar{x},\bar{y}) \wedge x_1 \not\equiv LS \wedge y_1 \neq 0] \supset \sum_{0}^{31} y_2(i+1)2^i = \sum_{x_2-y_1+1}^{31} x_3(i)\cdot 2^{i-x_2+y_1-1} + \sum_{1}^{x_2-y_1+1} x_3(31)\cdot 2^{31-x_2+y_1-1+i}$$

Dans ce cas aussi nous allons modifier l'expression qui se trouve à droite de la relation $\supset$ pour mettre en évidence que cette condition est vérifiée.

$$\supset \tfrac{1}{2}\sum_{0}^{31} y_2(i)\cdot 2^i - \tfrac{1}{2}y_2(0) + y_2(32)\cdot 2^{31} = \tfrac{1}{2}\sum_{x_2-y_1}^{31} x_3(i)\cdot 2^{i-x_2+y_1} - \tfrac{1}{2}x_3(x_2-y_1) +$$

$$+ \tfrac{1}{2}\sum_{0}^{x_2-y_1} x_3(31)\cdot 2^{31-x_2+y_1+i} + x_3(31)\cdot 2^{31}$$

$$\supset \sum_{0}^{31} y_2(i)\cdot 2^i = \sum_{x_2-y_1}^{31} x_3(i)\cdot 2^{i-x_2+y_1} + \sum_{1}^{x_2-y_1} x_3(31)\cdot 2^{31-x_2+y_1+i}$$

(condition vérifiée)

- Chemin B - C :

$$[P_{RS}(\bar{x},\bar{y}) \wedge y_1 = 0] \supset \sum_{0}^{31} y_2(i)\cdot 2^i = \sum_{x_2}^{31} x_3(i)\cdot 2^{i-x_2} + \sum_{1}^{x_2} x_3(31)\cdot 2^{31-x_2+i}$$

(condition vérifiée)

Comme pour l'instruction LS, la vérification de ces trois conditions permet de dire que l'organigramme de la figure II-11. est partiellement correct (pour LS et pour RS) par rapport à Φ et Ψ.

Pour RS, l'équivalence des expressions :

$$(y_2[1], y_2)[\iota 32] \equiv \quad y_2(i) \leftarrow y_2(i+1),\ i = 0,30$$

et le changement d'indices qui valide cette équivalence, étendent cette conclusion de correction partielle pour l'organigramme montré par la figure II-5 (b) (déroulement symbolique de "$\mathcal{MS}_{LSVRS}$").

Pour vérifier que l'organigramme de la figure II-11 est totalement correct et ayant montré qu'il est partiellement correct, il faut prouver que le programme termine pour toutes les valeurs qui vérifient

$$\Phi(\bar{x}) = \text{VRAI}.$$

Pour montrer l'existence de la terminaison nous appliquons la procédure exposée en II-1.

Dans cet organigramme correspondant à la réalisation microprogrammée des instructions LS et RS, on voit que la fonction la plus simple à appliquer en B pour réaliser l'association entre l'exécution des actions dans la boucle et l'ensemble des entiers naturels ordonnés par la relation $>$, est : $f_B(\overline{x},\overline{y}) = y1$

Il est évident que cette fonction vérifie la condition [2] (cf. terminaison) et par conséquent il nous est permis d'affirmer que le programme termine sur φ et qu'il est totalement correct par rapport à φ et Ψ (Ψ étant égal à $\Psi_{LS} \vee \Psi_{RS}$).

Le fait d'avoir vérifié que la partie de l'organigramme correspondant à la description des instructions LS et RS contenant une boucle est correcte par rapport à Ψ, implique l'équivalence entre la description des décalages du niveau "spécification" et celle du niveau "réalisation microprogrammée".

A propos de ce problème de la vérification de la terminaison dans un programme, il faudrait souligner qu'il se présente de manière différente par les descriptions globales des deux niveaux (figures I-7. et I-13.). En effet, dans ce cas la condition de terminaison est la mise à zéro de SW, ce qui implique que la terminaison de l'exécution des programmes abstraits "S" et "MS" exige le passage par une instruction STOP.

II-3 CONCLUSION

Nous avons montré dans ce chapitre quelques exemples permettant d'appliquer une procédure de vérification de la simulation entre deux programmes présentés sous la forme d'organigrammes. Cette procédure tend à établir une certaine systématisation du problème de la vérification ce qui entraîne dans certains cas, comme celui des boucles, l'introduction, dans l'étude, d'un degré de complexité qui peut sembler disproportionné par rapport aux résultats obtenus.

Nous serions de cet avis si nous n'avions pas vu que cette procédure aidait à déceler des erreurs de réalisation qui pourraient passer inaperçues si l'on se limitait à une simulation par ordinateur de la structure à vérifier. Au cours de l'étude de vérification de la S- Machine, menée par Birman (1) à partir de la description des deux niveaux en langage de définition VDL/APL, trois erreurs avaient été trouvées dans le microprogramme de la réalisation initiale de la S- Machine. Ces erreurs étaient :

1) Pour les instructions STX et ADX, il y avait une erreur qui entraînait l'exécution de SBX à la place des deux premières ;
2) Dans les instructions LS et RS si la partie adresse de l'instruction avait la valeur zéro, le microprogramme entrait dans une boucle qui réalisait $2^{24} - 1$ décalages.
3) Une erreur dans la partie recherche d'instruction dans le microprogramme qui ne se manifestait que dans le cas où l'adresse de l'instruction était $\geqslant 2^{23}$; si ceci survenait, aucune des instructions n'était exécutée correctement.

Il est certain que la première erreur pouvait être facilement décelée par simulation de la structure; cependant les deux autres avaient de grandes possibilités de rester masquées, d'une part parce que, pendant la simulation, on utilise une taille mémoire réduite et d'autre part parce que, pendant la simulation, les cas très particuliers comme LS, 0 ou RS, 0 risquent d'être non réalisés.

Dans la définition corrigée de la S- Machine, nous avons trouvé , sinon une erreur, plutôt une ambiguïté de la description dans la partie concernant le chargement du registre IR avec le code opération de l'instruction.

Nous avons remarqué aussi que dans la description du niveau "spécification" certaines instructions à une adresse comme LDI, LDXI, LS et RS

n'ont pas de possibilité d'adressage indirect ou indexé ; par conséquent, les deux premiers bits du mot correspondant à chacune de ces instructions peuvent, indifféremment, avoir la valeur 1 ou 0. Par contre, dans la réalisation microprogrammée le fait d'effectuer le calcul d'adresse (où l'on décide si l'adressage est indexé ou indirect) pour toutes les instructions à une adresse, implique que les mots correspondant aux instructions LDI, LDXI, LS et RS doivent avoir toujours les deux premiers bits à la valeur zéro (pour être en accord avec le fonctionnement spécifié de ces instructions).

L'exemple précédemment développé, appliqué au cas de la S- Machine, a montré en premier lieu que la représentation à partir d'organigramme permettrait une vérification formelle de la machine. Ceci nous a conduit à distinguer deux cas, selon que la microprogrammation de l'instruction comportait ou non une boucle. Le premier cas nécessite une preuve formelle par assertions et le deuxième est traité en recherchant l'équivalence entre les organigrammes décrivant les deux niveaux de spécification.

Le principal inconvénient de l'exemple réside dans le fait que la S- Machine n'est pas une machine réelle. Afin de lever les limitations qui pourraient en résulter, nous présenterons dans le chapitre suivant l'application de la procédure de vérification à la microprogrammation d'un microcalculateur réel, basé sur la famille AMD 2900.

CHAPITRE III

APPLICATION DE LA PROCEDURE DE VERIFICATION

A UN PROCESSEUR REEL

III-1 INTRODUCTION

Pour effectuer une application de la procédure de vérification exposée au chapitre précédent sur une structure réelle, nous avons sélectionné un système réalisé au L.A.A.S. dans le cadre d'une étude soutenue par un contrat DRME-ELECMA-LAAS (21).

Il s'agit d'un système organisé autour d'un microprocesseur cascadable, en technologie bipolaire LS (basse puissance Schottky) et dans lequel l'ensemble des instructions est microprogrammé. Cet ensemble d'instructions n'est pas prédéterminé, comme dans la plupart des autres microprocesseurs par les fabricants , c'est l'utilisateur qui doit l'implanter par microprogrammation.

- LE MICROPROCESSEUR -

Le microprocesseur utilisé dans ce système est le 2901 de AMD. La configuration interne de ce microprocesseur est représentée dans la figure III-1. Le chemin des données est d'une largeur de 4 bits. Cependant les différentes parties du microprocesseur ont été conçues de manière à rendre possible la concaténation ou mise en "cascade" de plusieurs boîtiers de façon à obtenir un ensemble microprocesseur ayant un chemin de données de 4 x n bits, n étant le nombre de boîters mis en cascade.

Les éléments principaux à l'intérieur du microprocesseur tels qu'on les voit sur la figure III-1 sont les suivants :

- Une mémoire à double accès (RAM) de 16 registres à 4 bits qui permet de lire en simultanéité le contenu de deux mots différents (ou le même deux fois) ; ces deux mots mémoire sont adressés par les adresses RA et RB. L'écriture permet de stocker un mot de 4 bits dans le registre adressé par la valeur RB. Ces actions de lecture et écriture peuvent être exécutées

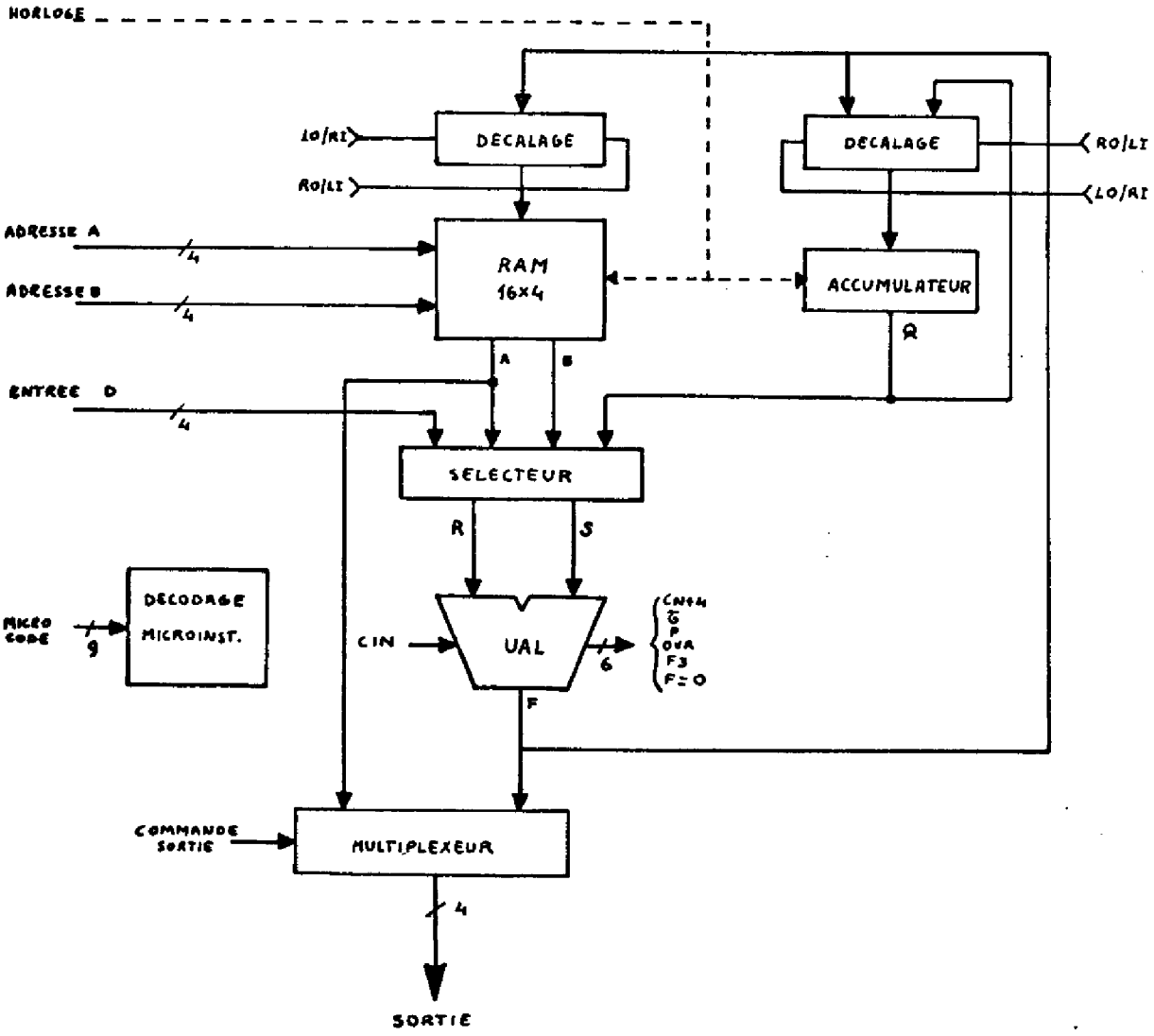

HORLOGE
LO/RI
RO/LI
DECALAGE
DECALAGE
RO/LI
LO/RI
ADRESSE A
ADRESSE B
4
4
RAM
16x4
ACCUMULATEUR
Q
A
B
ENTREE D
4
SELECTEUR
R
S
MICRO CODE
9
DECODAGE
MICROINST.
CIN
UAL
6
CN+4
G
P
OVR
F3
F=0
F
COMMANDE SORTIE
MULTIPLEXEUR
4
SORTIE

Figure III-1

dans le même cycle machine (la lecture est effectuée quand le signal d'horloge est à 1 et l'écriture quand il est à zéro, le résultat de la double lecture étant sauvegardé dans deux registres A et B pendant la valeur zéro du signal d'horloge).

- Une unité arithmétique et logique (UAL) capable de réaliser trois opérations arithmétiques : R + S, S - R et R - S et cinq opérations logiques RVS, RvS, $\overline{R} \wedge S$, $R \oplus S$, $\overline{R \oplus S}$.

L'UAL comporte plusieurs sorties pour la formation du mot d'état qui contient : signe, dépassement, zéro, retenue et de plus deux autres signaux nécessaires pour la formation de la retenue si le système conçu est constitué par plusieurs tranches en cascade comportant un générateur de retenue anticipée.

- Un registre accumulateur Q qui reçoit le résultat de l'UAL ; de plus, il est possible de commander un décalage d'un bit à gauche ou à droite de ce résultat avant de le mémoriser dans Q.

- Un sélecteur d'opérandes qui effectue le choix des deux opérandes, parmi les quatre possibles (A, B, Q, D), qui seront traités par l'UAL.

- Un circuit de décalage, similaire à celui de l'accumulateur, pour les vecteurs binaires à enregistrer dans la mémoire de registres.

- Un multiplexeur de sortie qui sélectionne un opérande parmi deux (A ou F).

Le fonctionnement de tous ces éléments, dans chaque cycle machine, est commandé par les différents champs de la microinstruction présente dans le registre MM montré dans la figure III-3. Il appartient au séquenceur de déterminer l'adresse de la microinstruction pour le cycle machine suivant.

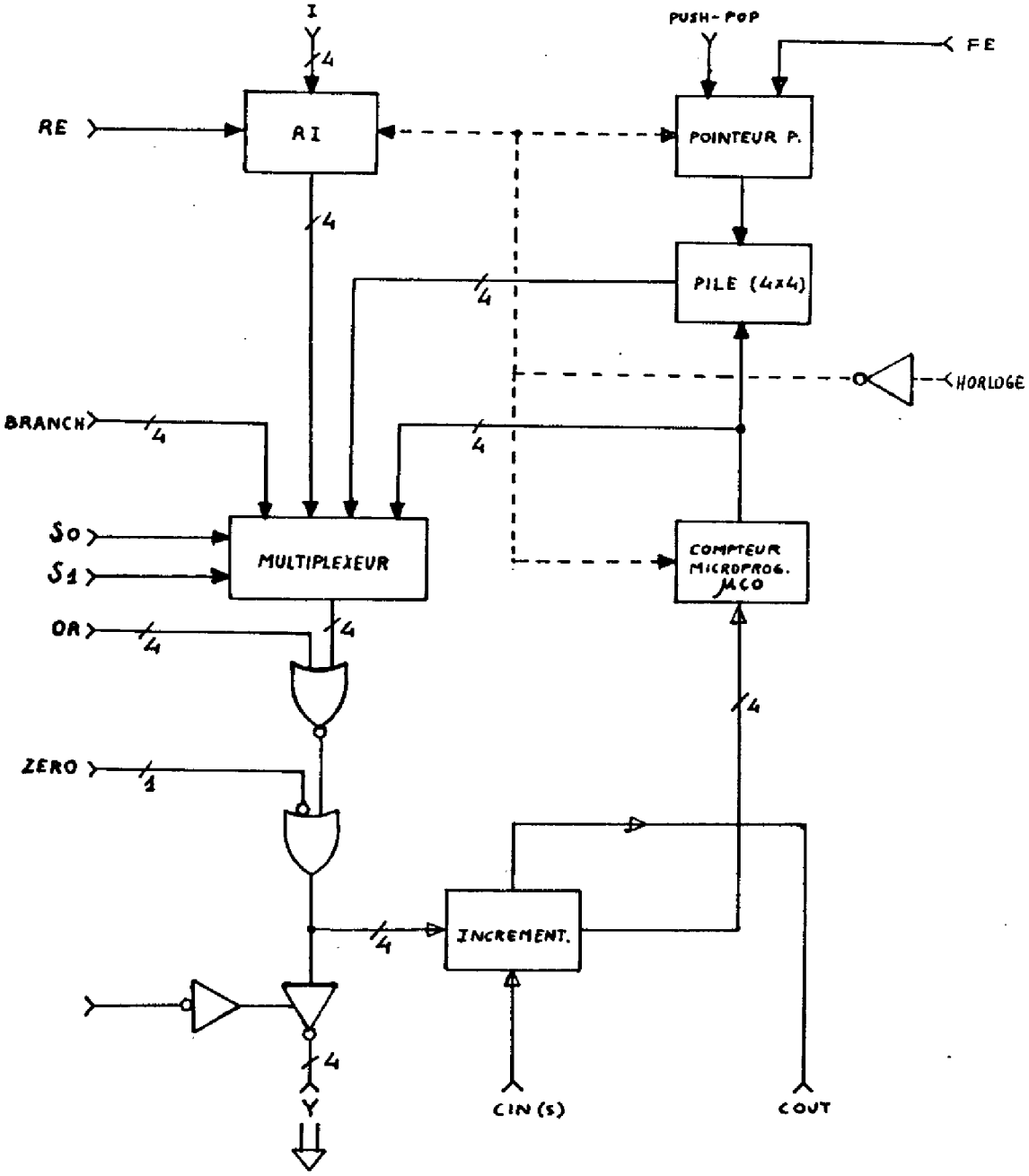

I
4
PUSH-POP
FE
RE
AI
POINTEUR P.
4
PILE (4x4)
4
HORLOGE
BRANCH
4
4
So
S1
MULTIPLEXEUR
COMPTEUR MICROPROG. μCO
OR
4
4
4
ZERO
1
INCREMENT.
4
4
Y
CIN (s)
COUT

Figure III-2

- LE SEQUENCEUR -

Le séquenceur de la famille AM2900 est présenté sous la forme d'un schéma bloc dans la figure III-2.

Comme nous venons de l'indiquer, le rôle essentiel de ce séquenceur est de déterminer l'adresse de la microinstruction qui doit être lue dans la mémoire du microprogramme à chaque cycle machine. Comme le microprocesseur, le séquenceur est construit avec un format de 4 bits par boîtier et il a aussi la possibilité d'être concaténé à d'autres boîtiers pour former une longueur de mot d'adresse de microprogramme de 4 x m bits, m étant le nombre de boîtiers concaténés.

Ainsi, la taille maximale de la mémoire du microprogramme détermine directement le nombre de tranches séquenceur (taille max. : $2^{4 \times m}$ mots). Dans ce séquenceur, le multiplexeur sélectionne, selon la valeur de deux bits So et S1, une des quatre voies d'entrée pour la diriger vers la sortie (les valeurs de So et S1 sont données par la microinstruction en cours d'exécution). Ces 4 voies sont (Figure III-2.) :

- BR (BRANCH), provenant de la microinstruction en cours d'exécution.
- RI, définissant le code opération d'une instruction.
- PILE, provenant d'une pile de 4 registres pour permettre la réalisation de sous microprogrammes.
- MCO, représentant l'adresse de la microinstruction en cours d'exécution incrémentée d'une unité.

Les différentes actions du séquenceur lui-même sont fonction des valeurs de certains bits de la microinstruction en cours d'exécution.

- LE SYSTEME -

Le système présenté ici traite des mots de 16 bits ce qui nécessite donc la concaténation de 4 tranches microprocesseur 2901 AMD).

D'autre part, la capacité mémoire de microprogramme choisie est de 512 mots ce qui fixe le nombre de tranches séquenceur à 3.

Le schéma de l'ensemble du système processeur est représenté par la figure III-3. Celui-ci n'est pas complet puisque, dans ce mémoire nous faisons maintenant abstraction de l'utilisation effective du système et nous y ferons allusion seulement pour justifier le rôle de certaines instructions. En conséquence, nous ne montrons dans la figure III-3. que la partie qui concerne directement notre étude.

Dans la figure III-3. MP représente la mémoire de programme (ROM) et MV une mémoire vive (RAM).

- LES MICROINSTRUCTIONS -

Les microinstructions sont constituées par des mots de 32 bits et peuvent être divisées en deux groupes selon leur rôle :

a) - Microinstructions de commande des transferts

b) - Microinstructions de branchement

Les microinstructions du groupe a) comportent les codes qui définissent la sélection des données à traiter (par le microprocesseur), le type d'opération à effectuer, le lieu de destination des résultats obtenus et les instants sélectionnés pour réaliser les lectures ou écritures en mémoire.

Les microinstructions du groupe b) permettent d'effectuer des branchements à des parties précises du microprogramme, autrement qu'en séquence ; le branchement peut être conditionnel ou inconditionnel. S'il est conditionnel un test est effectué sur l'état d'un bit sélectionné ; ainsi un saut dans le microprogramme peut être fonction de l'état des bits de :
signe, zéro, dépassement de capacité de Q, retenue.

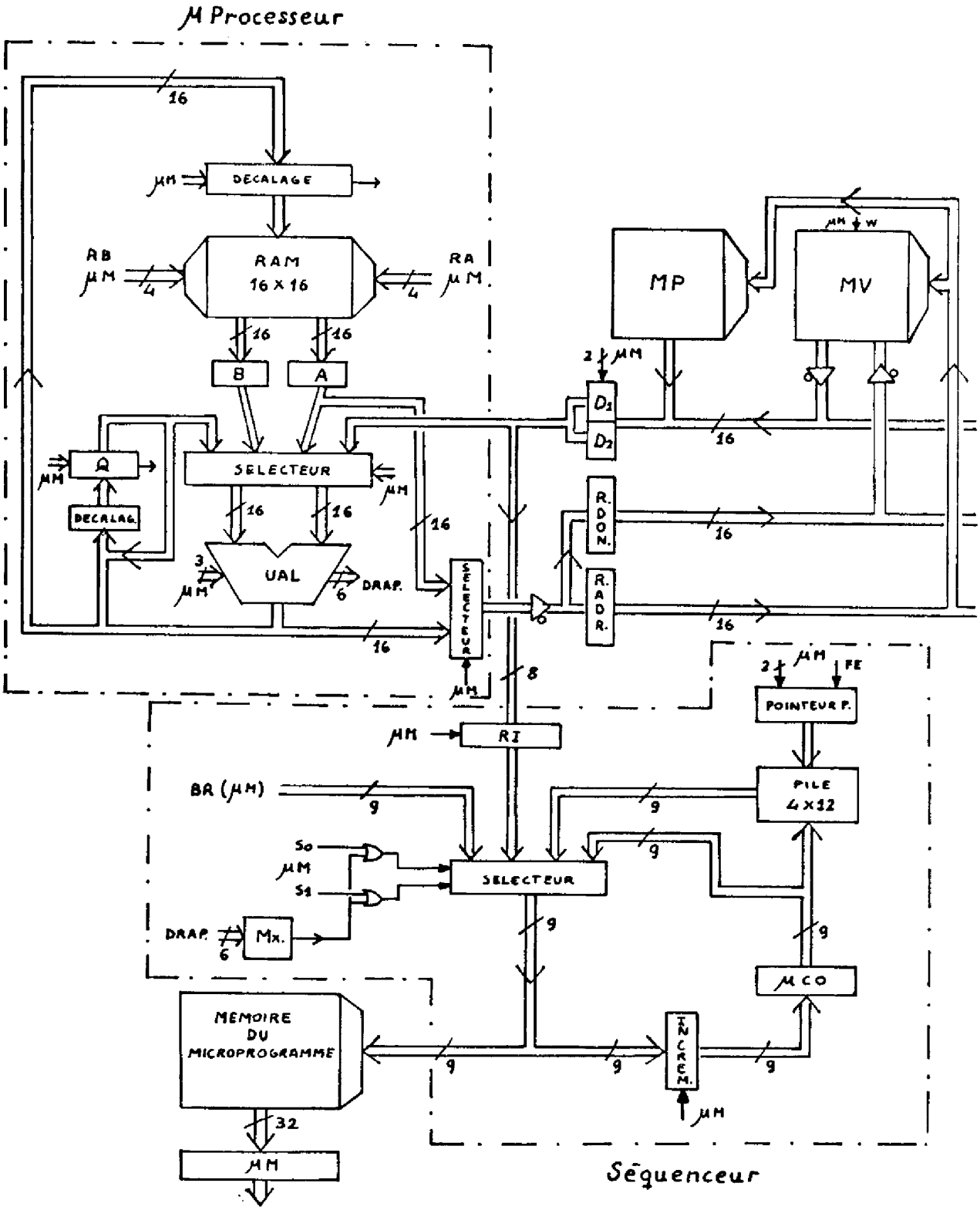
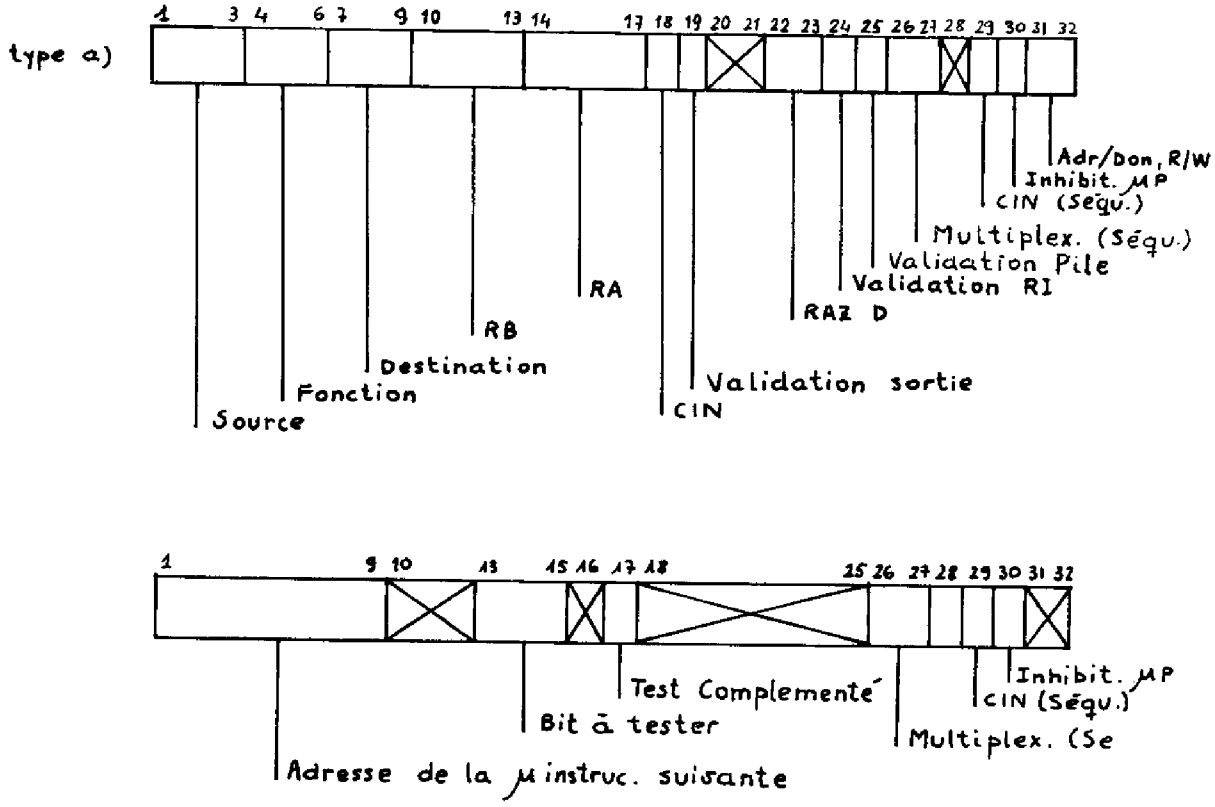

μ Processeur
16
μM
DECALAGE
RB
μM
4
RAM
16 X 16
4
RA
μM
16
16
B
A
Q
μM
DECALAG.
SELECTEUR
μM
16
16
3
μM
UAL
6
DRAP.
16
16
SELECTEUR
μM
8
MP
MV
μM W
2
μM
D1
D2
16
R. DON.
16
R. ADR.
16
2
μM
FE
POINTEUR P.
μM
RI
BR (μM)
9
9
PILE 4 X 12
So
μM
S1
SELECTEUR
9
9
DRAP.
6
Mx.
μ CO
MEMOIRE DU MICROPROGRAMME
9
9
INCREM.
9
μM
32
μM
Séquenceur

Figure III-3

Si le branchement est inconditionnel, l'adresse de la microinstruction suivante est donnée par les neuf premiers bits de la microinstruction en cours d'exécution ou bien par les huit premiers bits du registre RI.

Les formats de ces deux types de microinstructions sont donnés dans la figure suivante :



(Ces formats ont été choisis par les concepteurs du système).

- Instructions -

Les instructions définies par les réalisateurs de ce système en tenant compte de l'utilisation pour laquelle le microcalculateur est destiné,

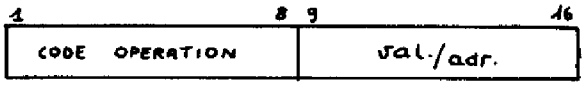
peuvent, aussi, être divisées en deux groupes, selon le nombre de mots de 16 bits qui les composent :

1) Instructions comportant deux mots de 16 bits.

2) Instructions comportant un seul mot de 16 bits.

Le premier mot de chaque instruction contient le code opération dans les huit premiers bits ; les huit bits suivants peuvent représenter une valeur ou une adresse (entre 0 et 255) ou bien ne sont pas utilisés. Le second mot des instructions du premier groupe constitue une adresse de 0 à $2^{16} - 1$.

Le format du premier mot de l'instruction est représenté par la figure suivante :



Les tableaux donnés ci-dessous indiquent les différentes instructions définies et la description informelle de leurs actions.

1) INSTRUCTIONS A DEUX MOTS -

MNEMONIQUE	DESCRIPTION
INIT val / ADR	- Initialisation d'un mot mémoire d'adresse ADR à la valeur "val" : M (ADR) val.
GET - / ADR	- Chargement du mot M (ADR) dans le registre R10.
PUT - / ADR	- Chargement du contenu du registre R10 dans le mot M (ADR).
IFINF val / ADR	- Si le contenu de R10 val, saut dans le programme à l'instruction d'adresse ADR.
IFSUP val / ADR	- Si le contenu de R10 val, saut dans le programme à l'instruction d'adresse ADR.

MNEMONIQUE	DESCRIPTION
IFEQ | val / ADR	- Si le contenu de R10 = val, saut dans le programme à l'instruction d'adresse ADR.
INC | val / ADR	- Incrementation du mot mémoire M (ADR) de la quantité val.
JUMP | - / ADR	- Saut dans le programme à l'instruction d'adresse ADR.
CALL | - / ADR	- Enregistrement de la dernière adresse de l'instruction à l'adresse 0 de la mémoire et saut du programme à l'instruction d'adresse ADR (1ère instruction d'un sous-programme).
ADR10 | - / ADR	- Incrémentation du contenu du mot mémoire M (ADR) de la quantité contenue dans R10.
SHL | val / ADR	- Décalage à gauche (div. par 2) du contenu du mot M (ADR) autant de fois que "val".
TEQ | val / ADR	- Si le contenu du mot M (ADR) = val, on saute l'instruction suivante (skip); sinon:en séquence.
MOVE | adr / ADR	- Assignation du contenu du mot M (adr) au mot M (ADR).
OUT | val / ADR	- Enregistrement de la quantité "val" dans le mot M (ADR).
READ | adr / ADR	- Enregistrement du mot dont l'adresse est le contenu du mot mémoire M (adr), dans le mot M (ADR). M (ADR) M (M(adr))
WRITE | adr / ADR	- Enregistrement du mot M (ADR) dans le mot mémoire dont l'adresse est le contenu du mot qui se trouve à l'adresse "adr". M (M(adr)) M (ADR)
INDEX | val / ADR	- Scrutation, dans la table qui commence à l'adresse ADR, de l'égalité entre le contenu des adresses impaires (local à la table) et le contenu de R10. Enregistrement dans R12 de l'adresse suivante (dans la table) en cas d'égalité. Sinon branchement à la 1ère instruction qui se trouve après la table.

2) INSTRUCTIONS A UN MOT -

MNEMONIQUE	
JRP | val	- Saut dans le programme à l'adresse CO + val.
JRN | val	- Saut dans le programme à l'adresse CO - val.
BAL | -	- Chargement du compteur ordinal avec le contenu du mot d'adresse zéro, incrémenté d'une unité.
JUMPI | adr	- Saut dans le programme à l'instruction d'adresse donnée par le contenu du mot M (adr).
SAVE | -	- Enregistrement de la valeur du compteur ordinal, (celle de l'adresse de l'inst. suivante), dans le mot d'adresse zéro.
RETURN | -	- Chargement du compteur ordinal avec la valeur FFFE.
CALL12 | -	- Chargement du compteur ordinal avec la valeur contenue dans R12.
DECR10 | -	- Décrémentation du contenu de R10 de la quantité "val".

Afin de déterminer si la description informelle représentée dans ces tableaux correspond à ce que l'on attend de chacune des instructions, nous nous sommes basés sur l'étude des programmes mis en place par les concepteurs.

III-1-1. Description du niveau "Spécification" -

Pour procéder plus tard à l'étude de vérification de la réalisation correcte du microprogramme, nous avons traduit cette description informelle sous forme d'organigramme.

Comme dans le cas de la S- Machine, nous utilisons certains opérateurs du langage APL pour définir les actions et opérations réalisées sur le contenu des différents registres.

Dans cette description du niveau "spécification" nous ne tenons compte que des éléments "visibles" par l'utilisateur (ou programmeur) du système.

Ces éléments sont les suivants :

- Registres à 16 bits : C0 ou R0, R10, R12 } Dans RAM du MP
  D (Formé par deux octets, D1 et D2)
  RDo
  RAdr

- Mémoire de capacité 64 K mots de 16 bits, différenciée en MV (RAM) et en MP (ROM mémoire programme).

La figure III-4 montre l'organigramme général de cette description ; la partie correspondant aux différentes instructions est donnée par les figures III-5 et III-6.

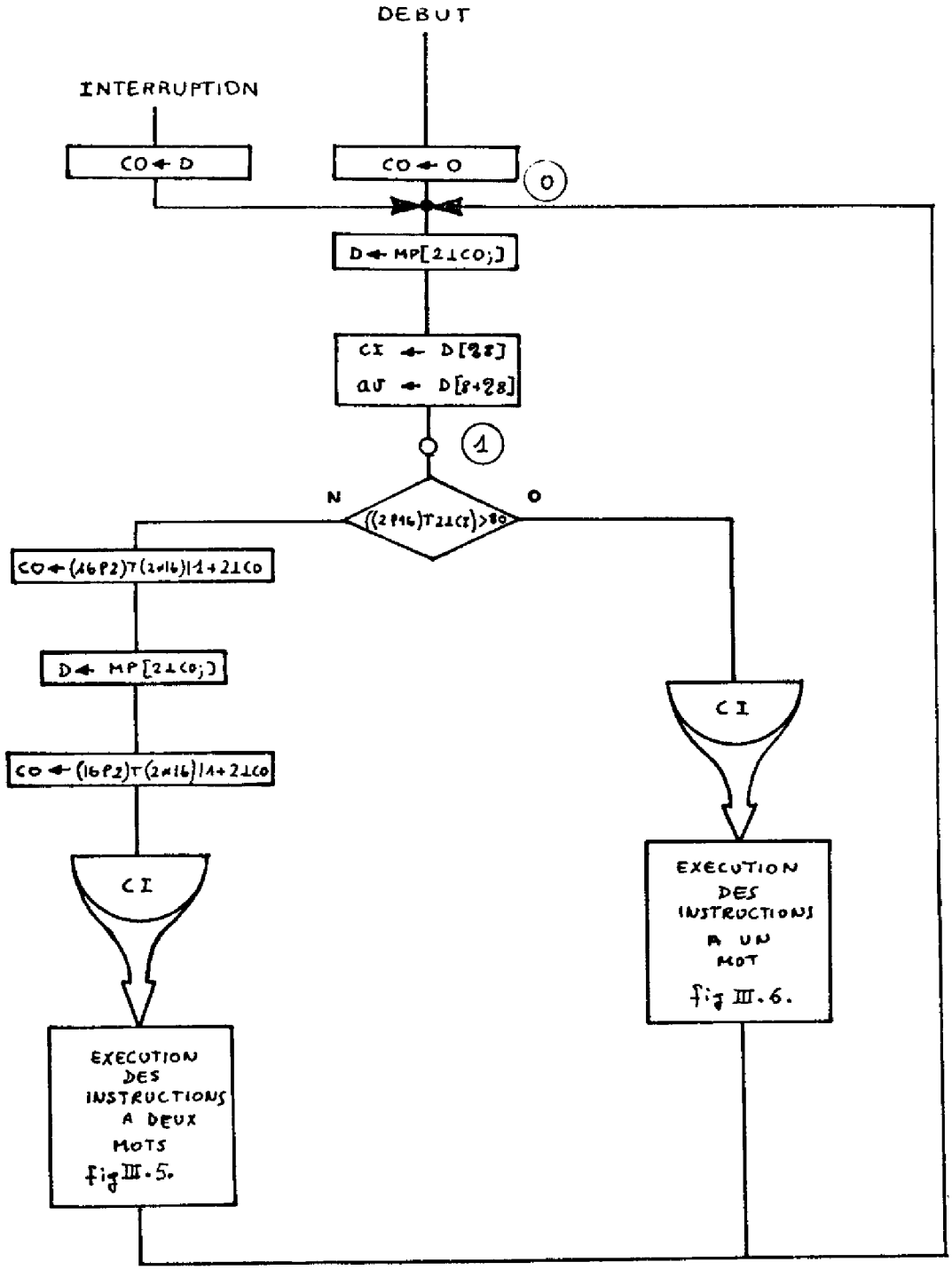

DEBUT
INTERRUPTION
CO ← D
CO ← 0
0
D ← MP[2⊥CO;]
CI ← D[28]
AU ← D[8+28]
1
N
O
((2*16)⊤2⊥CI)>80
CO ← (16P2)⊤(2*16)⊥1+2⊥CO
D ← MP[2⊥CO;]
CO ← (16P2)⊤(2*16)⊥1+2⊥CO
CI
EXECUTION DES INSTRUCTIONS A DEUX MOTS
fig III.5.
CI
EXECUTION DES INSTRUCTIONS A UN MOT
fig III.6.

Figure III-4

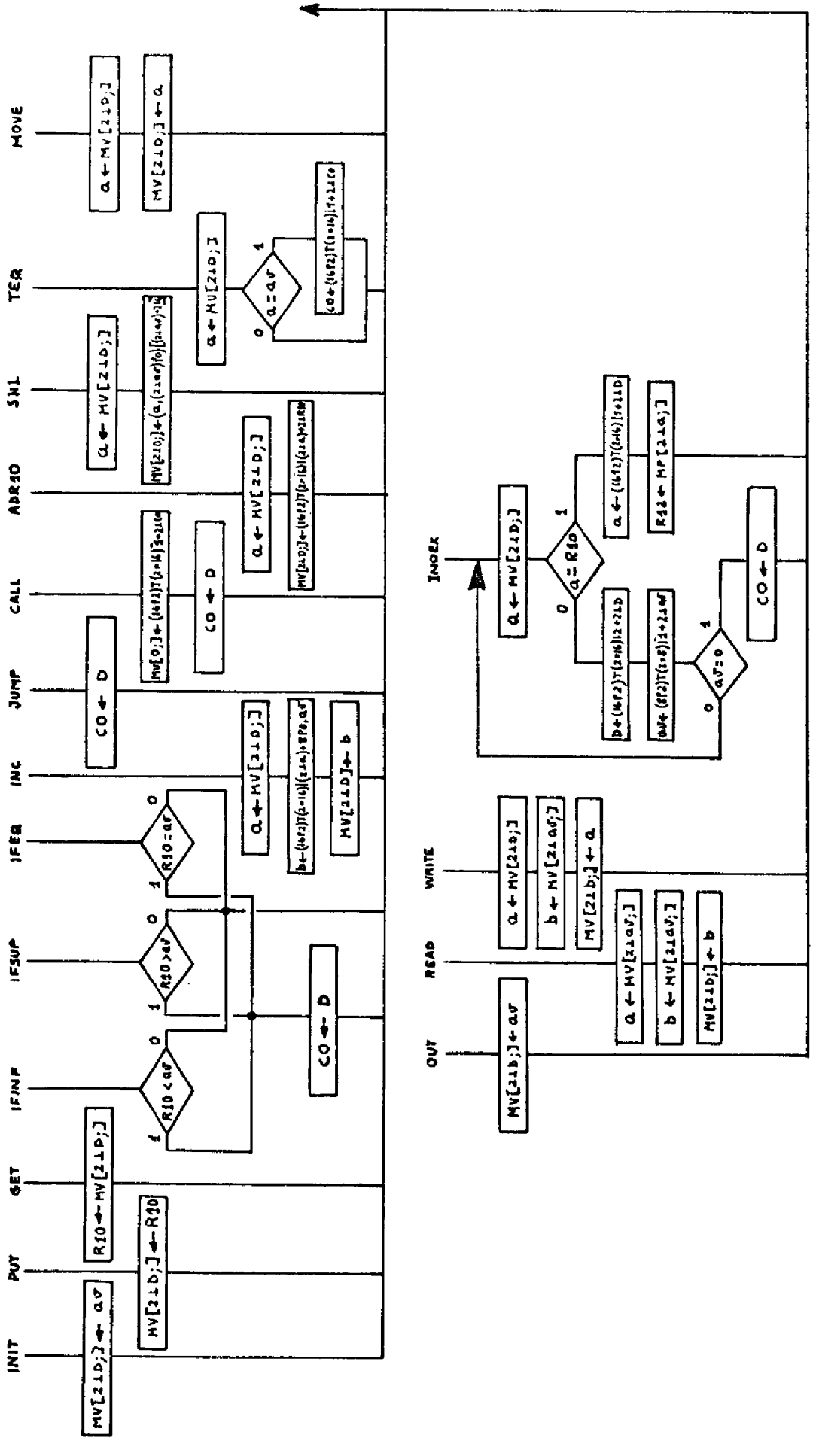

INIT
PUT
GET
IFINF
IFSUP
IFEQ
INC
JUMP
CALL
ADR10
SNL
TEQ
MOVE
INDEX
READ
WRITE
OUT

Figure III-5

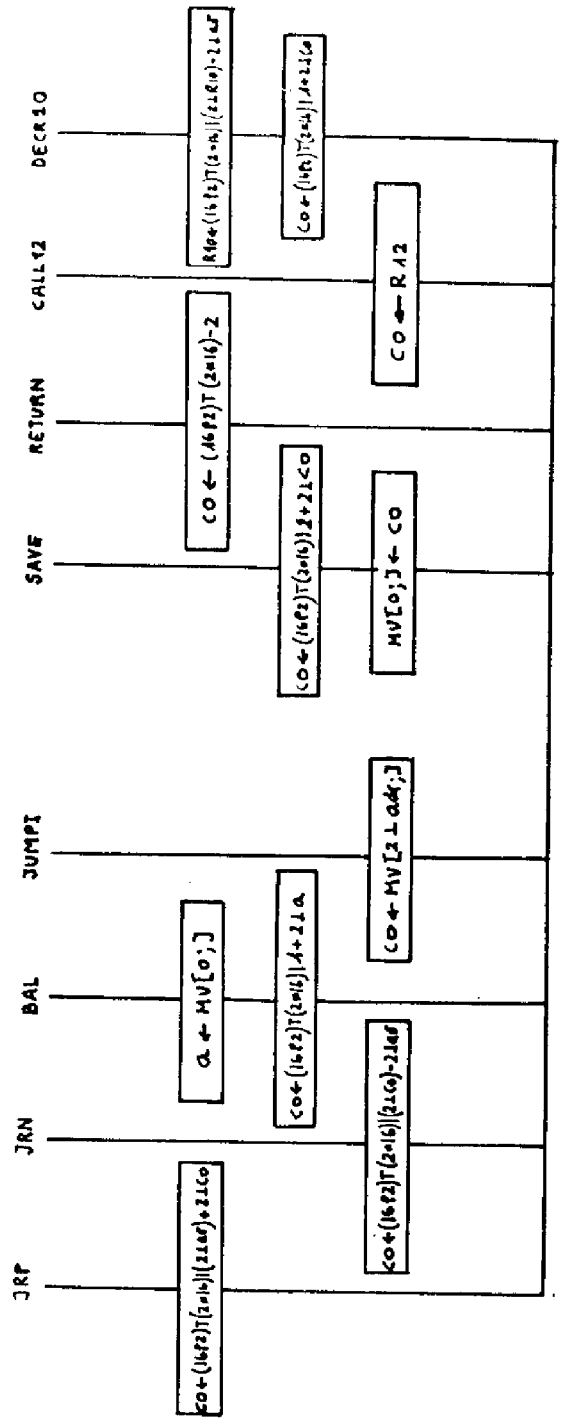

JRP
JRN
BAL
JUMPI
SAVE
RETURN
CALL12
DECR10
CO ← (A6F2)T(2+16)-2
CO ← R A2
A ← MV[0;]
MV[0;] ← CO
CO ← MV[2 ⊥ adr;]

Figure III-6

III-1-2. Description du niveau "Réalisation microprogrammée"

La description du niveau "réalisation microprogrammée" a été précisée à partir du microprogramme réalisé par les concepteurs du système.

Les éléments du système qui doivent être explicités dans la description de ce niveau sont les suivants :

- Mémoire Principale (MP et MV).
- Mémoire du microprogramme.
- Registres de 16 bits :

CO	R9	Registres dans RAM du microprocesseur.
R1	R10	
R2	R11	
R3	R12	
R4	R13	
R5	R14	
R8	R15	

D (D1, D2)
RDon
RAdr

- Registre de 32 bits, MM.

L'organigramme correspondant à la description de ce niveau est présenté dans les figures III-7, 8 et 9. Comme pour la description précédente, la figure III-7. correspond à la description de l'ensemble du microprogramme et les figures III-8. et 9 correspondent à la description des actions de chaque instruction.

Les expressions à l'intérieur de chaque rectangle dans ces organigrammes représentent les actions d'une seule microinstruction du microprogramme.

De la même manière que dans le chapitre I, l'organigramme du niveau "réalisation microprogrammée" est présenté en écriture simplifiée, c'est-à-dire que, pour des raisons de simplicité dans la présentation, nous avons écrit, par exemple, à la place de l'expression :

CO←(16↑2)T(2*16)|1 + 2⊥CO

qui décrit l'incrémentation du compteur ordinal, l'expression CO←CO + 1

où la forme vectorielle du registre n'est plus maintenue. Cependant, comme nous le verrons plus loin, la première expression sera celle utilisée dans l'étude de la vérification.

De plus, dans la description, nous différencions MP et MV (Mémoire morte de programme et mémoire de stockage des données).

En réalité, cette différenciation n'est faite qu'implicitement, selon la valeur de l'adresse RAdr ; autrement dit nous aurions dû écrire M [RAdr;] au lieu de MV [RAdr ;] ou MP [RAdr ;] mais nous avons préféré spécifier cette différenciation dans l'écriture pour mieux donner une idée de l'emplacement des vecteurs binaires dans les mémoires.

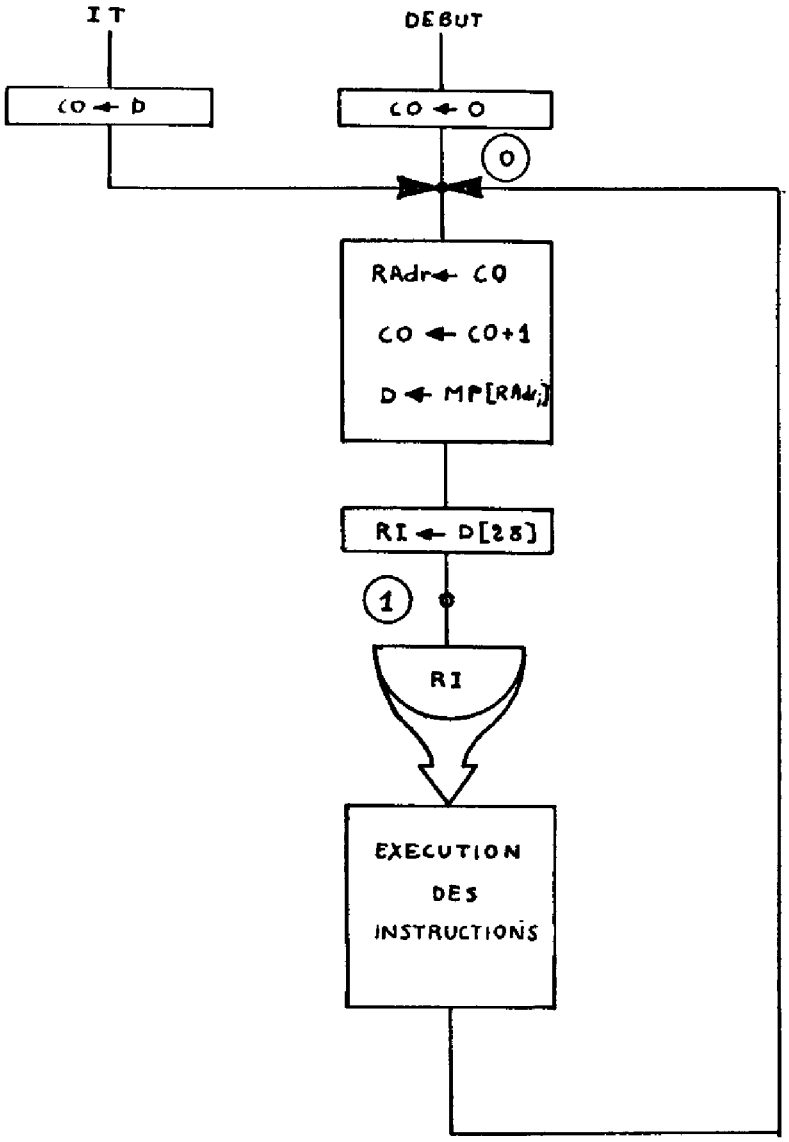

IT
DEBUT
CO ← D
CO ← 0
0
RAdr ← CO
CO ← CO+1
D ← MP[RAdr]
RI ← D[28]
1
RI
EXECUTION
DES
INSTRUCTIONS

Figure III-7

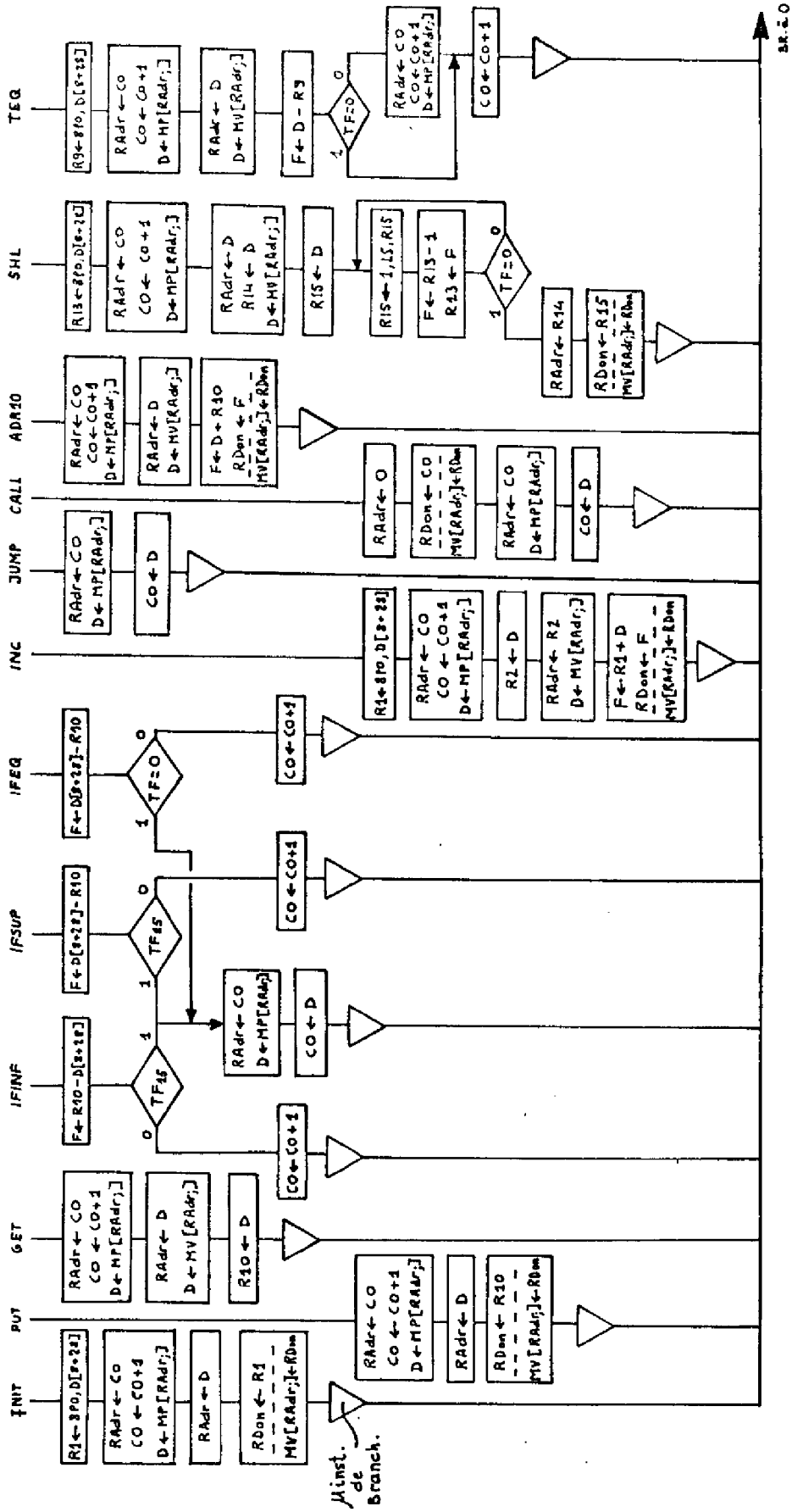

INIT
PUT
GET
IFINF
IFSUP
IFEQ
INC
JUMP
CALL
ADR10
SHL
TEQ
μinst. de Branch.
SR.C.O

Figure III-8

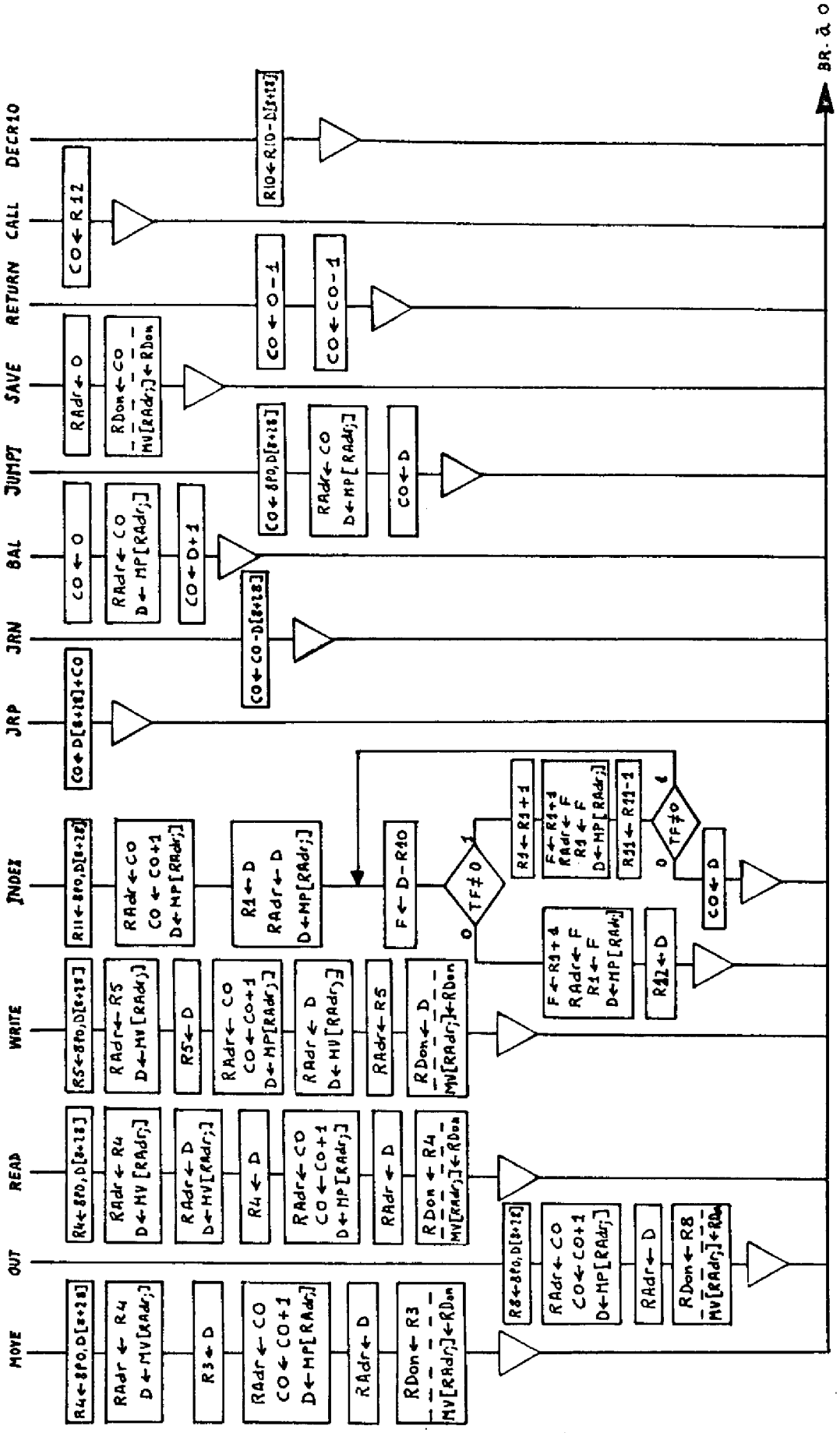
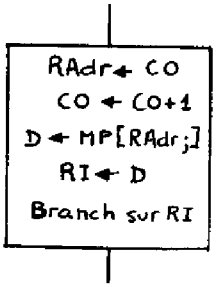

MOVE
OUT
READ
WRITE
INDEX
JRP
JRN
BAL
JUMPT
SAVE
RETURN
CALL
DECR10
BR. à 0
R4←8f0,D[8:18]
RAdr ← R4
D←MV[RAdr]
R3← D
RAdr← CO
CO← CO+1
D←MP[RAdr]
RAdr← D
RDon← R3
MV[RAdr]←RDon
R4←8f0,D[8:18]
RAdr ← R4
D←MV [RAdr]
RAdr← D
D←MV[RAdr]
R4← D
RAdr← CO
CO← CO+1
D←MP[RAdr]
RAdr← D
RDon ← R4
MV[RAdr]←RDon
R5←8f0,D[8:18]
RAdr← R5
D←MV [RAdr]
R5← D
RAdr← CO
CO← CO+1
D←MP[RAdr]
RAdr← D
D←MV [RAdr]
RAdr← R5
RDon← D
MV[RAdr]←RDon
R11←8f0,D[8:18]
RAdr← CO
CO ← CO+1
D←MP [RAdr]
R1← D
RAdr← D
D←MP[RAdr]
F← D-R10
TF≠0
R1← R1+1
F← R1+1
RAdr← F
R1← F
D←MP[RAdr]
R11← R11-1
TF≠0
CO← D
F← R1+1
RAdr← F
R1← F
D←MP[RAdr]
R12← D
CO←8f0,D[8:18]
RAdr← CO
D←MP[RAdr]
CO← D
CO← D[8:18]+CO
CO← CO-D[8:18]
CO← 0
RAdr← CO
D← MP[RAdr]
CO← D+1
CO←8f0,D[8:18]
RAdr← CO
D←MP[RAdr]
CO← D
RAdr← 0
RDon← CO
MV[RAdr]←RDon
CO← 0-1
CO← CO-1
CO← R12
R10← R10-D[8:18]

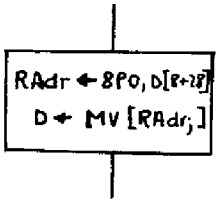
Figure III-9

Certaines microinstructions comportent plusieurs actions qui sont réalisées dans le même cycle machine ; ceci est représenté par plusieurs lignes à l'intérieur du même rectangle et ceci dans l'ordre où ces actions sont effectuées.

A ce propos un point important doit être signalé : notre étude suppose vérifiée la compatibilité entre les actions spécifiées dans une microinstruction et les instants où elles sont requises. Par exemple, dans une première implémentation, les réalisateurs ont trouvé qu'une microinstruction telle que celle représentée dans la figure suivante n'est pas compatible avec les signaux d'horloge nécessaires à son exécution.



Un autre exemple de ce problème, qui s'est présenté au cours de l'implémentation du microprogramme est celui de la microinstruction suivante :



Cette microinstruction permet de charger le registre RAdr avec les huit derniers bits du registre D pour effectuer par la suite le chargement de ce registre D avec le contenu du mot mémoire situé à l'adresse RAdr.

Cependant le signal qui provoque la mise à zéro du premier octet du registre D, reste actif pendant toute la durée du cycle machine et en conséquence, à la place de la valeur attendue D = MV[RAdr ;] on trouverait D = 8 f 0, (MV [RAdr ;]) [8 + ι8] .

Il existe donc un certain nombre d'erreurs que nous ne considérons pas dans ce mémoire. En effet, notre approche est orientée vers le côté logiciel pour la vérification. C'est pour cette raison que nous considérons l'implémentation réalisée, comme correcte du point de vue de la synchronisation et de la compatibilité des signaux d'horloge avec les actions spécifiées.

Avant de commencer l'étude de vérification, on peut remarquer dans la figure III-5. l'existence de deux instructions qui réalisent les mêmes actions : leur seule différence se situe au niveau de l'emploi de registres intermédiaires différents. Il s'agit des instructions INIT et OUT.

A première vue, il semble qu'une implémentation quasiment dupliquée ne puisse se produire, mais, pendant la période de conception de l'ensemble des instructions qui sont nécessaires pour réaliser les programmes spécifiques à traiter, la quantité de détails dont on doit tenir compte est très importante.

Il peut donc arriver, que pour réaliser une série d'actions, le fait d'avoir déjà établi une instruction qui pourrait les faire, passe inaperçu. Bien entendu, ce genre d'erreur est, en général, rapidement décelé, sans nécessiter des méthodes particulières. Néanmoins, la formalisation conduit à les mettre en évidence plus rapidement.

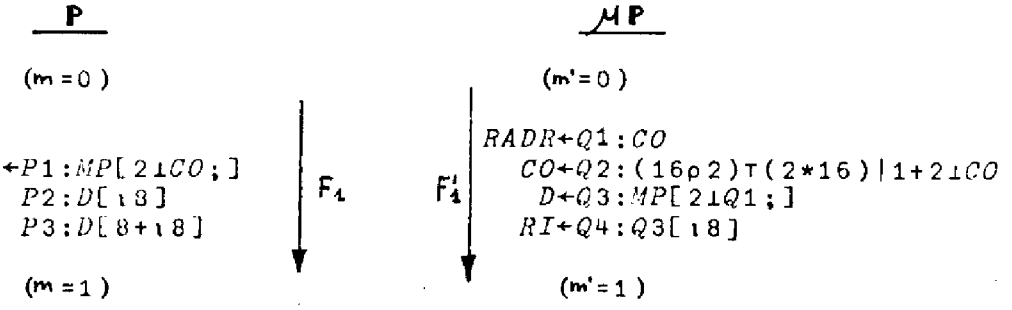
III-2 ETUDE DE VERIFICATION DE LA SIMULATION ENTRE LES DEUX NIVEAUX DE DESCRIPTION

Comme dans le chapitre II, cette étude sera réalisée en établissant des relations de simulation en différents points du déroulement des programmes abstraits dont on fait des interprétations symboliques en suivant les organigrammes des figures III-4, 5 et 6 pour ce qui concerne le niveau "spécification" et ceux des figures III-7, 8 et 9 pour le niveau "réalisation microprogrammée".

Pour ce système il n'existe pas d'instruction d'arrêt d'exécutions l'arrêt étant commandé par l'interrupteur de Marche-Arrêt (MA) qui établit ou coupe l'alimentation des circuits. Après la mise sous tension du système, l'exécution d'un programme débute en positionnant à zéro le compteur ordinal du microprogramme.

La microinstruction qui se trouve à cette adresse est donc la première à être exécutée.

Dans les organigrammes montrés sur les figures III-4 et III-7 (correspondant à la description générale de chacun des deux niveaux de description) on peut distinguer une partie commune à toutes les instructions, partie qui se trouve entre les points de marque ⓪ et ① et qui constitue la partie "recherche d'instruction". Les interprétations symboliques de cette partie pour chacun des deux niveaux sont les suivantes :



Les domaines des états des deux programmes abstraits P et MP sont respectivement D et D' définis par :

$$D = \{ m, r \mid MA(r) = 1 \ \vee \ MA(r) = 0 \}$$

MA = 1 Marche
MA = 0 Arrêt

$$D' = \{ m', s \mid MA(s) = 1 \ \vee \ MA(s) = 0 \}$$

où M et M' sont les ensembles des points de marquage (0 et 1 dans ce cas).

Les domaines initiaux Do et Do' sont :

$$Do = \{ 0, r \mid MA(r) = 1 \ , \ 0 \leq 2 \perp co(r) < 2^{16} \}$$

$$Do' = \{ 0, s \mid MA(s) = 1 \ , \ 0 \leq 2 \perp co(s) < 2^{16} \}$$

"r" représente un des états possibles de P et "s" un des états possibles de μP.

Nous établissons la relation R de simulation entre les deux niveaux comme étant : $R = Ro \cup RI$

où $Ro = \{ m, m' \mid m = m' = o$
$r, s \mid r \in Do ; s \in Do' ; MP(r) = MP(s) ; MV(r) = MV(s) ; CO(r) = CO(s) ;$
$RA0(r) = RA0(s) ; RA2(r) = RA2(s) ; MA(r) = MA(s) = 1 \}$

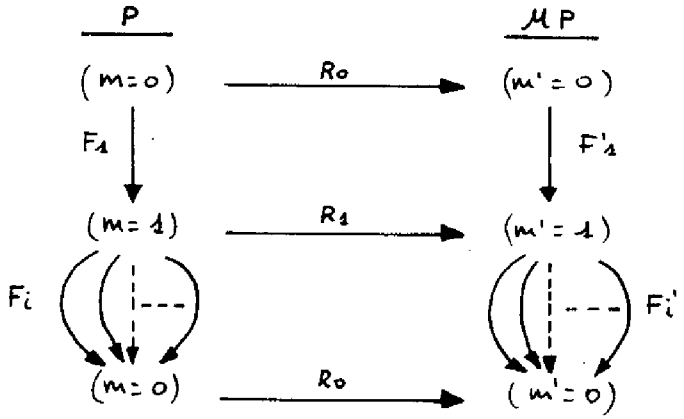
$RI = \{ m$
$r, s \mid r \in D ; s \in D ; MV(r) = MV(s) ; RA0(r) = RA0(s) ; RA2(r) = RA2(s) ;$
$CI(r) = RI(s) ; MA(r) = MA(s) = 1 \}$

Pour cette partie "recherche d'instruction" on peut vérifier que la condition C1 (cf. chapitre II) est satisfaite, c'est-à-dire que : partant d'un couple $(r,s) \in R$ tel que $r \in Do$ et $s \in Do'$, l'application des fonctions d'exécution F1 et F'1 à P et P' respectivement, conduit à un nouveau couple $(F_1 (r), F'_1 (s)) \in R$.

Bien entendu ceci est vrai pour r et s tels que

$< F_1 (r) >$ et $< F'_1 (s) >$.

Comme nous l'avons montré au chapitre II, il est possible d'étudier la vérification de la simulation de la manière représentée par le diagramme suivant :



Fi et F'i représentent les fonctions d'exécution de chaque instruction. Ces fonctions sont appliquées aux états machine après la partie "recherche d'instruction". Ces états sont regroupés dans les domaines DI et D'I :

$$DI = \{\lambda, r \mid MA(r)=1 ;\ 0 \leq 2 \perp CO(r) < 2^{16};\ 9 < ((2P16)T2 \perp CI) < 88\}$$

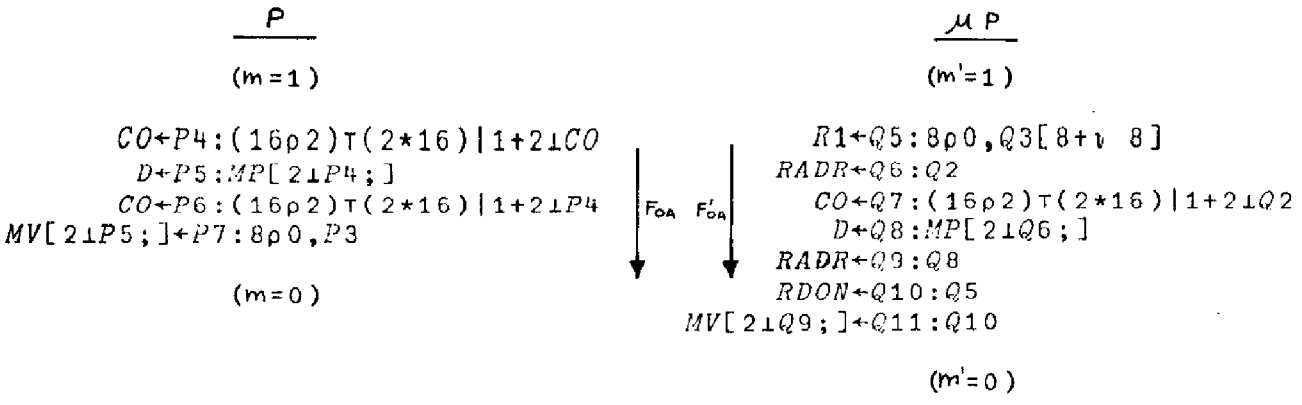
$$D'I = \{\lambda, s \mid MA(s)=1 ;\ 0 \leq 2 \perp CO(s) < 2^{16};\ 9 < ((2P16)T2 \perp RI) < 88\}$$

Au point de marquage ① une première différence entre les deux niveaux de description du système est la suivante : Au cours de l'application de F'_1 la valeur du compteur ordinal CO (r) est incrémentée d'une unité tandis que l'application F_1 ne change pas la valeur de CO (s). En conséquence, cette différence doit être incluse et traitée lors de l'analyse des fonctions Fi et F'i.

Dans la suite, nous allons présenter les interprétations symboliques de l'application des différentes Fi et F'i correspondant à chaque instruction.

L'étude de vérification est faite à partir de ces exécutions symboliques.

- INSTRUCTION "INIT", i = OA (cette valeur de i correspond au code héxadécimal de l'instruction) :-



On vérifie que : P6 = Q7
P7 = Q11

Par conséquent $(F_{OA}(d_1), F'_{OA}(d'_1)) \in R_o \subset R$ ce qui vérifie la condition $\mathcal{C}1$ (cf. chapitre II)

- INSTRUCTION "GET", i = OB :

P

(m=1)

```
CO←P4:(16ρ2)⊤(2*16)|1+2⊥CO
 D←P5:MP[2⊥P4;]
CO←P6:(16ρ2)⊤(2*16)|1+2⊥P4
R10←P7:MV[2⊥P5;]
```

(m=0)

F_{OB} / F'_{OB}

μP

(m'=1)

```
RADR←Q5:Q2
  CO←Q6:(16ρ2)⊤(2*16)|1+2⊥Q2
   D←Q7:MP[2⊥Q5;]
RADR←Q8:Q7
   D←Q9:MV[2⊥Q8;]
R10←Q10:Q9
```

(m'=0)

On vérifie que : P6 = Q7
P7 = Q10

- INSTRUCTION "PUT", i = OC :

P

(m=1)

```
CO←P4:(16ρ2)⊥(2*16)|1+2⊥CO
 D←P5:MP[2⊥P4;]
CO←P6:(16ρ2)⊤(2*16)|1+2⊥P4
M[2⊥P5;]←P7:R10
```

(m=0)

F_{OC} / F'_{OC}

μP

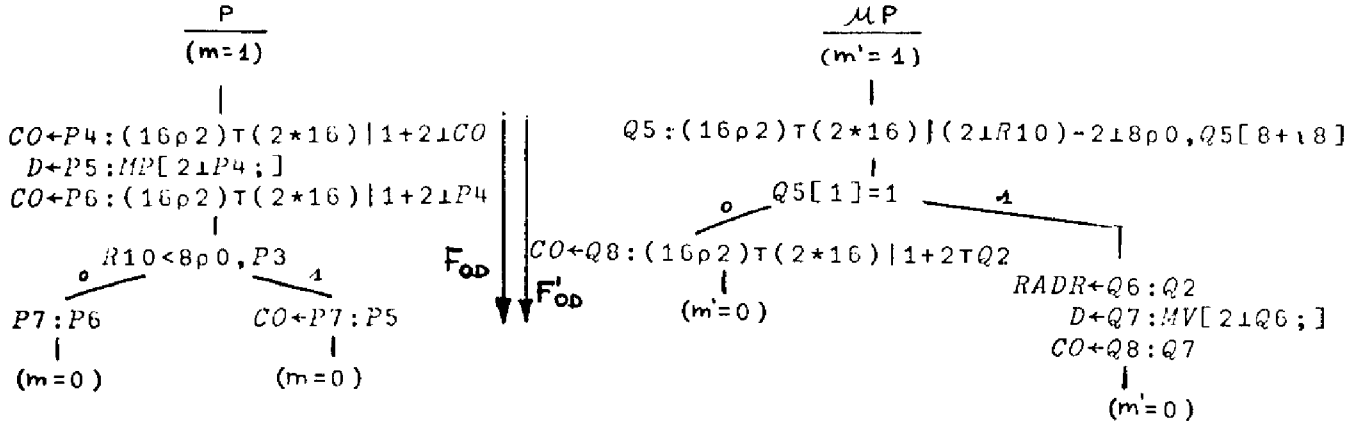
(m'=1)

```
RADR←Q5:Q2
  CO←Q6:(16ρ2)⊤(2*16)|1+2⊥Q2
   D←Q7:MP[2⊥Q5;]
RADR←Q8:Q7
RDON←Q9:R10
MV[2⊥Q8;]←Q10:Q9
```

(m'=0)

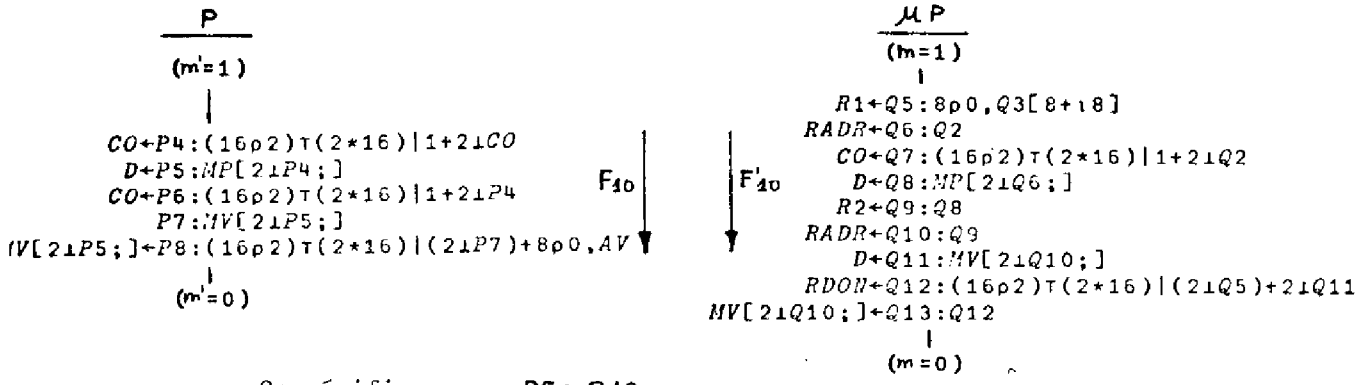
On vérifie que : P6 = Q6
P7 = Q10

- INSTRUCTION "IFINF", i = OD :



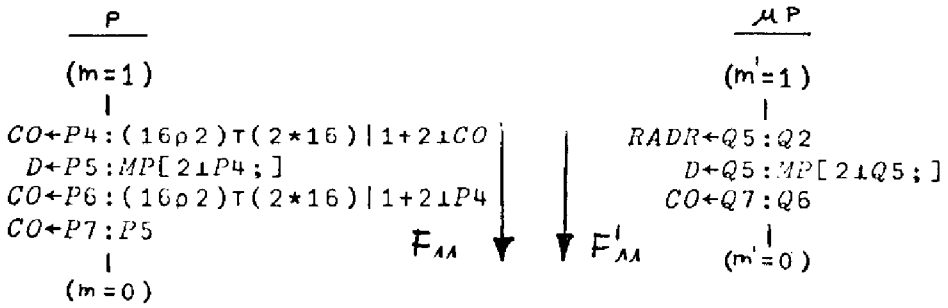
On vérifie que : P7 = Q8

- INSTRUCTION "INC", i = 10 :



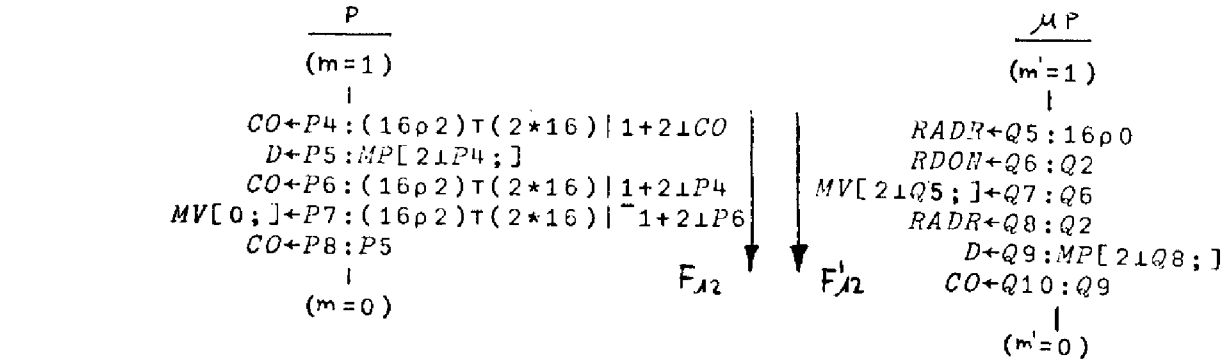
On vérifie que : P5 = Q10
P6 = Q7

- INSTRUCTION "JUMP", i = 11 :



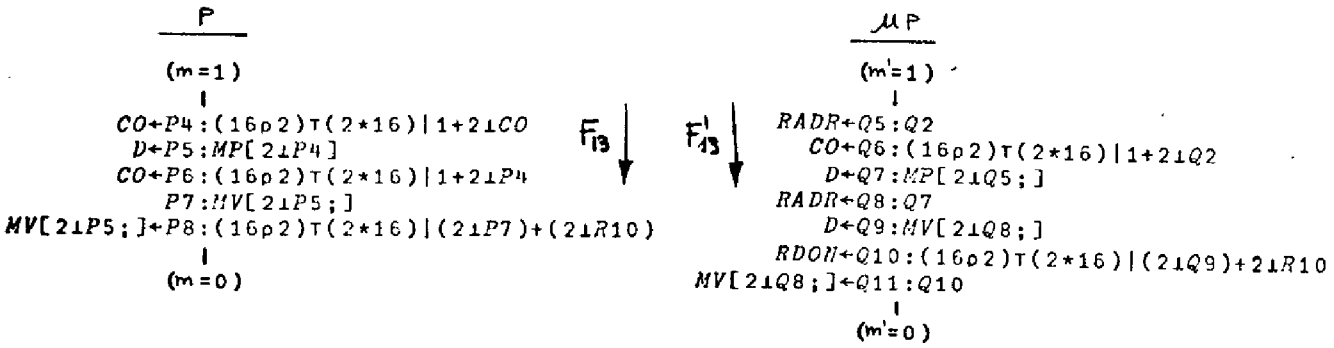
On vérifie que : P7 = Q7

- INSTRUCTION "CALL", i = 12 :



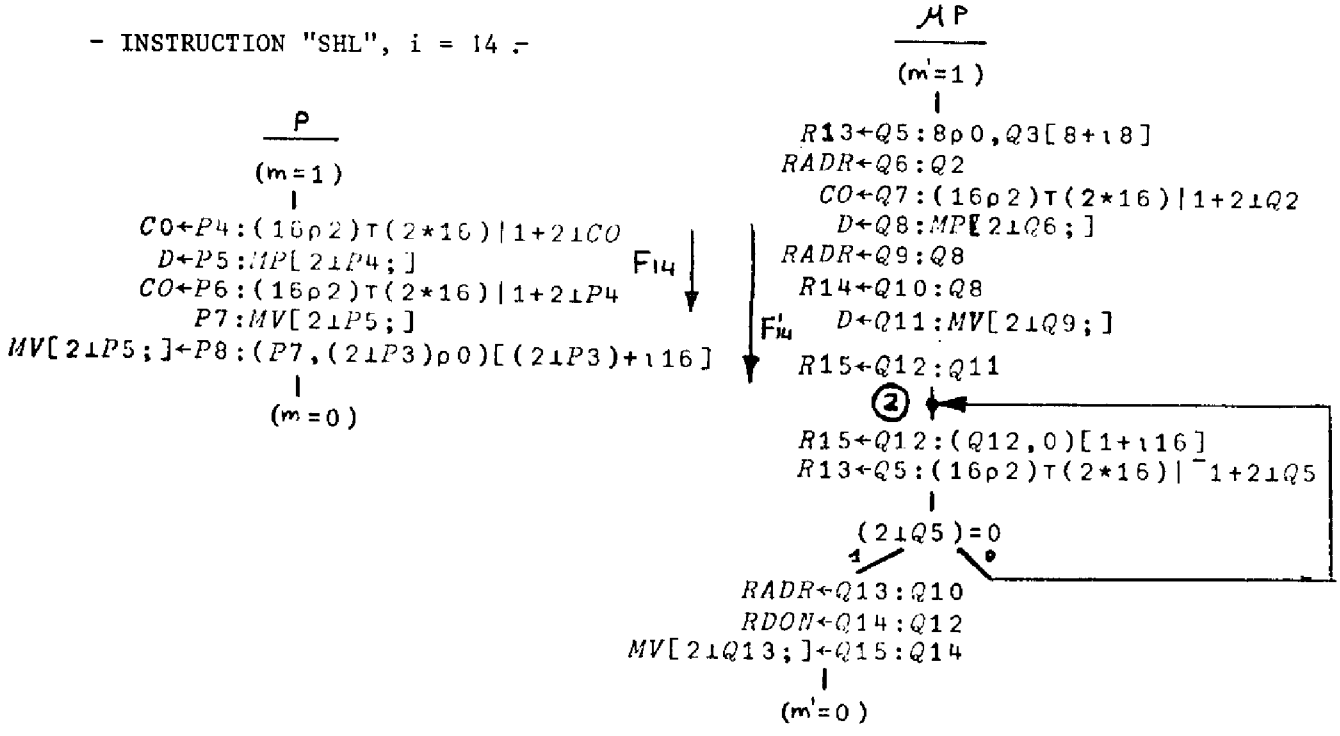
On vérifie que : P7 = Q7

- INSTRUCTION "ADR10", i = 13 :



On vérifie que : P6 = Q6
P8 = Q11
P5 = Q8

- INSTRUCTION "SHL", i = 14 :



La réalisation microprogrammée de cette instruction comporte une boucle qui exécute un décalage d'un bit à chaque passage. La structure de cette boucle est très proche de celle qui a été étudiée pour l'instruction LS de la S- Machine au Chapitre II ; nous n'allons pas reprendre ici la vérification de la partie de l'organigramme correspondant à cette boucle puisque celle-ci est pratiquement la même que la vérification pour l'instruction LS du Chapitre II.

D'après cette étude nous concluons que :

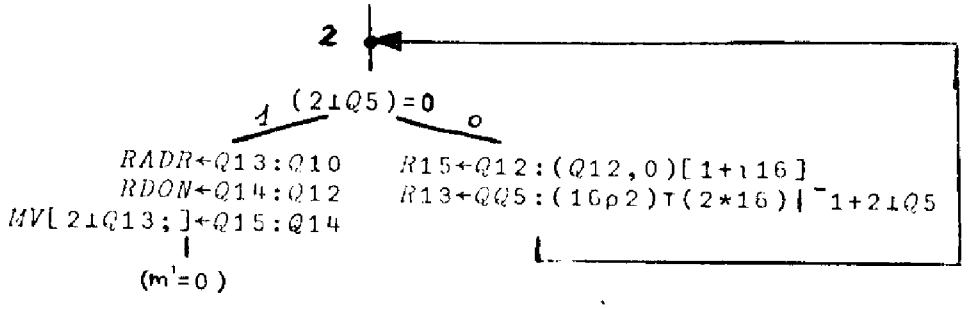
pour des valeurs de $Q5 \neq 0$ on vérifie que :

P6 = Q7
P5 = Q13
P8 = Q15

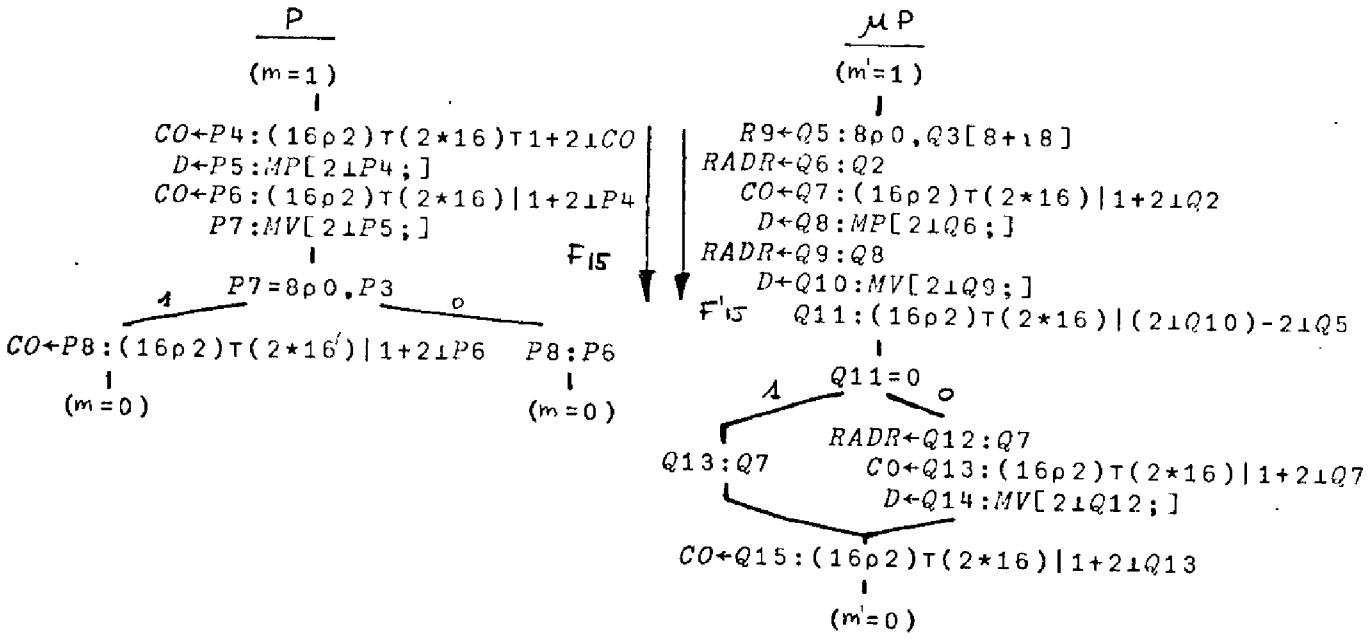
c'est à dire que la condition C1 est vérifiée et que la réalisation microprogrammée simule la description spécifiée pour cette instruction.

Cependant si Q5 = 0 (ce qui signifie que le deuxième octet du premier mot de l'instruction : val = 0) la relation d'égalité P8 = Q15 n'est pas vérifiée parce que dans ce cas la réalisation microprogrammée de cette instruction entre dans une boucle qui effectue 2^{16} décalages à gauche.

Pour éviter ce mauvais fonctionnement il suffit de modifier l'ordre de trois microinstructions et d'en ajouter une réalisant un branchement inconditionnel de façon à avoir une interprétation symbolique comme celle donnée ci-dessous. Ainsi, on vérifie que pour toute valeur de Q5 = P3 alors P8 = Q15.

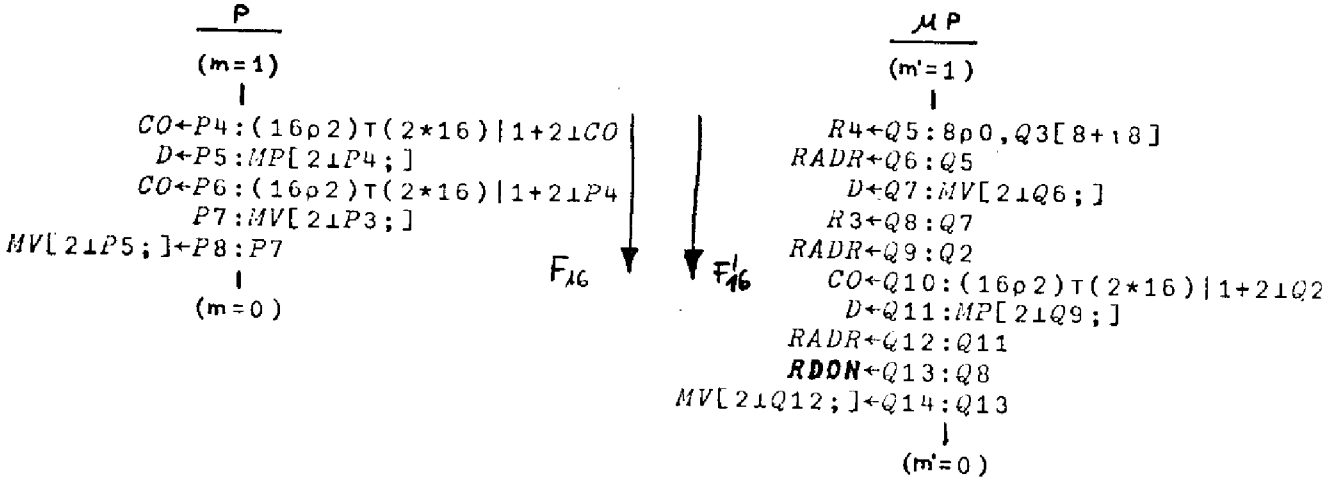


- INSTRUCTION "TEQ", i = 15 -



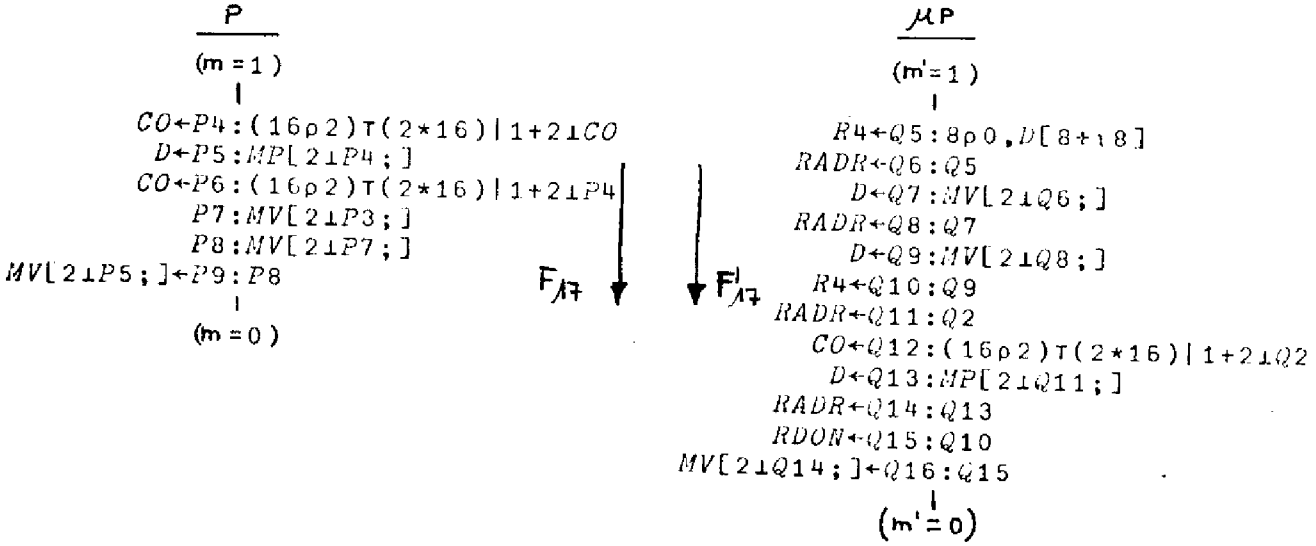
L'égalité P8 = Q15 n'est pas vérifiée ⇒ erreur (cf. III-3)

- INSTRUCTION "MOVE", i = 16 -



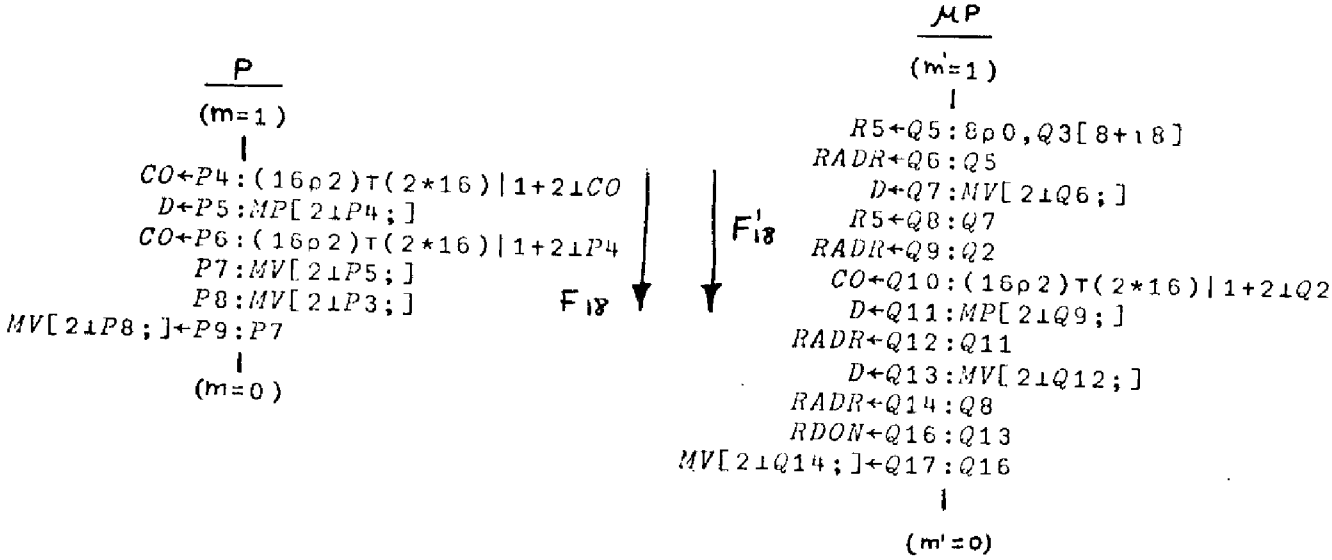
On vérifie que : P6 = Q10
P4 = Q9
P7 = Q8

- INSTRUCTION "READ", i = 17 :



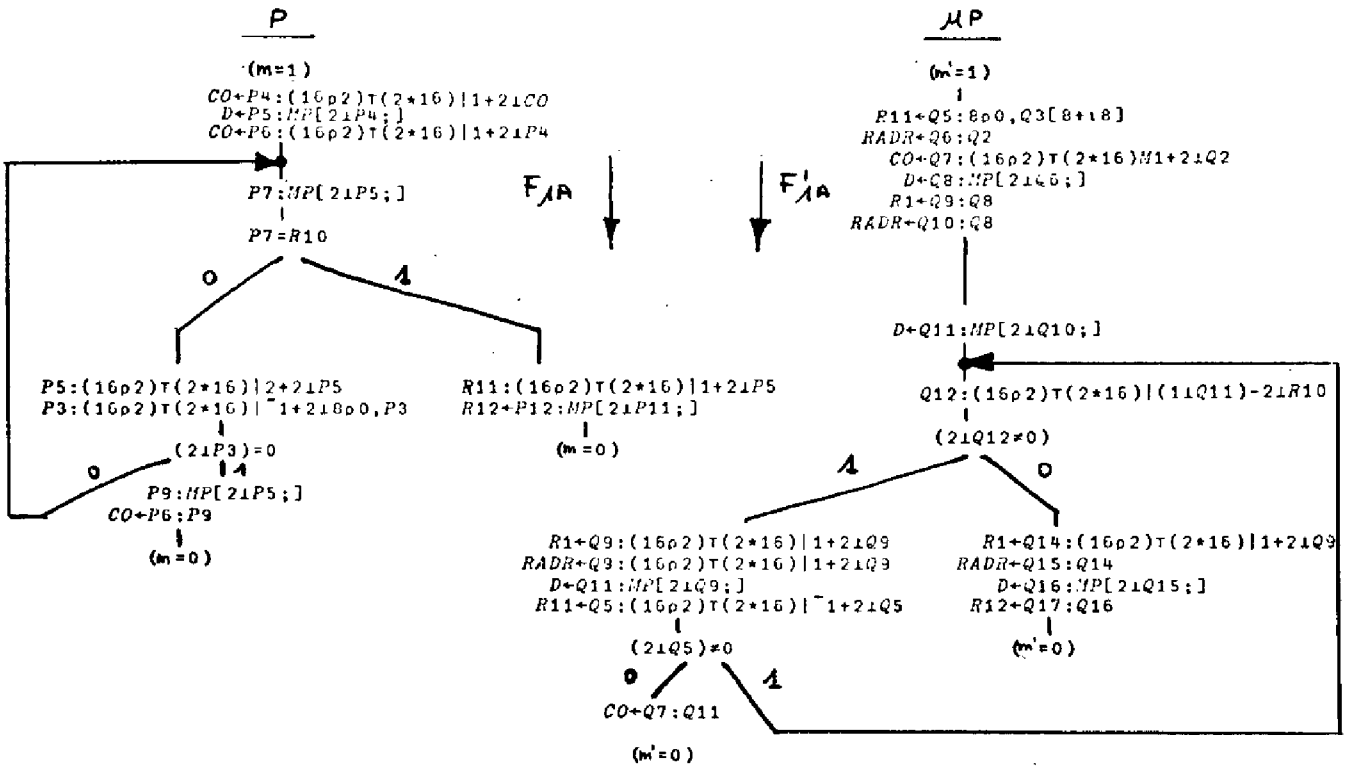
On vérifie que : P6 = Q12
P9 = Q16
P4 = Q11

- INSTRUCTION "WRITE", i = 18 :



On vérifie que : P6 = Q10
P9 = Q17
P8 = Q14

- INSTRUCTION "INDEX", i = 1A -



Il doit être vérifié que : P6 = Q7
P12 = Q17

Cette instruction est assez particulière : au cours de son exécution on recherche dans un tableau qui débute à l'adresse indiquée par P5 = Q8 = M (ADR) et de longueur (1 + 2 x 2⊥Q5) = (val + 1), s'il existe égalité entre le contenu du registre R10 et le contenu des mots qui se trouvent à des adresses impaires du tableau (en supposant que le premier mot du tableau se trouve à une adresse impaire, sinon le contraire). Si l'on trouve une égalité, la valeur de l'adresse du mot qui vérifie, incrémentée d'une unité, est enregistrée dans le registre R12. Si par contre, en fin de tableau il n'a pas été trouvé d'égalité, le registre CO est chargé avec le

dernier mot du tableau.

La figure III-10 montre un organigramme équivalent à celui correspondant à l'instruction INDEX dans MP.

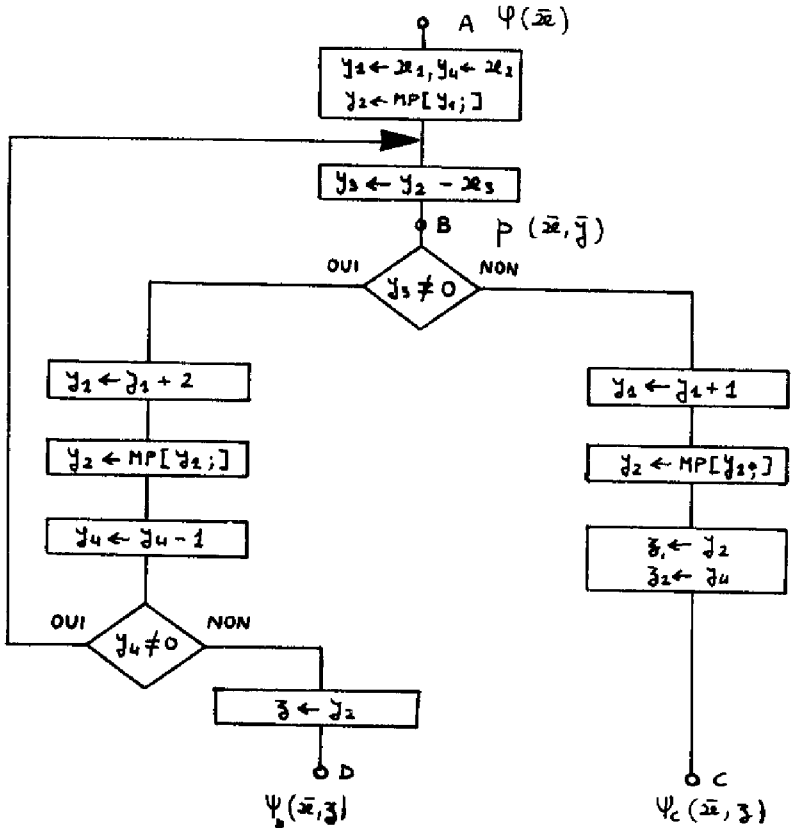


Figure III-10

Dans cet organigramme les valeurs de $\overline{x}$ représentent :

x1 ≡ 2 ⊥ R1
x2 ≡ 2 ⊥ R11
x3 ≡ 2 ⊥ R10

Etant donné l'existence de la boucle dans le programme nous allons étudier sa vérification sur l'organigramme de la figure III-10. et ceci selon

la méthode exposée en II-2. Dans ce cas, il faut souligner qu'il s'agit d'un programme particulier car il ne traite pas des opérations arithmétiques mais travaille sur un tableau.

Nous établissons les prédicats d'entrée et sortie suivants :

$$\varphi(\bar{x}) : 1 < 2 \cdot x_2 < 2^{16}, \quad 0 \leq x_3 < 2^{16}, \quad 0 < x_1 < 2^{16}$$

où $2 \cdot x2$ est la longueur du tableau et x1 la valeur de l'adresse de la tête du tableau.

$$\psi_C(\bar{x}, \bar{z}) : z_1 = MP[(x_1 + 2(x_2 - z_2) + 1);]$$

$$\psi_D(\bar{x}, z) : z = MP[(x_1 + 2x_2);]$$

Une assertion invariante au point B de coupure de la boucle est la suivante :

$$p(\bar{x}, \bar{y}) : y_4 > 0 \wedge (y_1 - x_1) = 2(x_2 - y_4)$$

Les conditions de vérification pour chaque chemin sont alors :

- CHEMIN A - B -

$$[\varphi(\bar{x}) \supset (x_2 > 0 \wedge x_1 - x_2 = 2(x_2 - x_2))]$$

Condition vérifiée.

- CHEMIN B - B -

$$[p(\bar{x}, \bar{y}) \wedge (y_4 - 1) \neq 0 \wedge y_3 \neq 0 \supset (y_4 - 1) > 0 \wedge (y_1 + 2 - x_1) = 2(x_2 - y_4 + 1)]$$

Condition vérifiée.

- CHEMIN B - C -

$$[p(\bar{x}, \bar{y}) \wedge y_3 = 0 \supset MP[(y_1 + 1);] = MP[(x_1 + 2(x_2 - y_4) + 1);]]$$

Condition vérifiée (puisque (y1 - x1) = 2(x2 - y4))

- CHEMIN B - D -

$$[p(\bar{x}, \bar{y}) \wedge y_3 \neq 0 \wedge y_4 = 0 \supset MP[(y_1 + 2);] = MP[(x_1 + 2x_2);]]$$

Condition vérifiée (puisque (y1 - x1) = 2 (x2 -(y4+1)) $\wedge$ y4 = 0)

Les conditions de vérification étant satisfaites, nous pouvons affirmer que cet organigramme est partiellement correct par rapport aux prédicats $\Psi(\bar{x})$, $\Psi_c(\bar{x},\bar{z})$ et $\Psi_d(\bar{x},z)$.

Pour montrer qu'il est totalement correct il faut maintenant vérifier que pour toutes les valeurs d'entrée satisfaisant le prédicat $\Psi(\bar{x})$, le programme termine. Pour cet organigramme il est facile de voir que, d'après ce qui a été exposé en II-2-1, un choix de $f_B(\bar{x},\bar{y})=y4$ comme fonction associée au point de coupure de la boucle, établit un rapport entre l'exécution des actions dans la boucle et un ensemble partiellement ordonné par la relation d'ordre $>$ dont les éléments sont des entiers naturels.

En conséquence le programme termine toujours par des valeurs d'entrée $\bar{x}$, tels que $\Psi(\bar{x})$ = VRAI.

Cette instruction est employée dans un programme d'utilisation du système avec un tableau comprenant 11 mots ($x_2 = 5$)

L'instruction OUT ne sera pas étudiée puisque, comme nous l'avons signalé, elle est similaire à l'instruction INIT.

- INSTRUCTION "JRP", i = 80 -

$$\underset{(m=1)}{\overset{P}{}} \qquad\qquad \underset{(m'=1)}{\overset{\mu P}{}}$$

$$CO \leftarrow P4:(16\rho 2)\tau(2*16)\,|\,(2\perp P3)+2\perp CO \qquad CO \leftarrow Q5:(16\rho 2)\tau(2*16)\,|\,(2\perp Q3[8+\iota 8]+2\perp Q2$$

$$\underset{(m=0)}{} \qquad\qquad \underset{(m'=0)}{}$$

On doit vérifier que P4 = Q5. Néanmoins nous avons signalé que au point de marquage m = m' = ①, CO (r) = CO (1) - 1 ; en conséquence pour que l'égalité P4 = Q5 soit vérifiée il faut changer soit la spécification de l'instruction (ce qui équivaut à changer l'utilisation de cette instruction dans les programmes) soit la réalisation microprogrammée.

- INSTRUCTION "JRN", i = 81 :-

```
        P                                        μP
    (m=1)                                    (m'=1)
      |                                        |
CO←Q5:(16ρ2)⊤(2*16)|(2⊥Q2)-2⊥Q3[8+ι8]   CO←P4:(16ρ2)⊤(2*16)|(2⊥CO)-2⊥P3
      |                                        |
    (m=0)                                    (m'=0)
```

Même remarque que pour JRP.

- INSTRUCTION "BAL", i = 82 :-

```
        P                                        μP
    (m=1)                                    (m'=1)
      |                                        |
   P4:MV[0;]                              CO←Q5:16ρ0
CO←P5:(16ρ2)⊤(2*16)|1+2⊥P4              RADR←Q6:Q5
      |                                    D←Q7:MP[2⊥Q6;]
    (m=0)                                CO←Q8:(16ρ2)⊤(2*16)|1+2⊥Q7
                                             |
                                           (m'=0)
```

On vérifie que : **P5 = Q8**

- INSTRUCTION "JUMPI", i = 83 :-

```
        P                                        μP
    (m=1)                                    (m'=1)
      |                                        |
CO←P4:MV[2⊥P3;]                          CO←Q5:8ρ0,Q3[8+ι8]
      |                                  RADR←Q6:Q5
    (m=0)                                  D←Q7:MP[2⊥Q6;]
                                         CO←Q8:Q7
                                             |
                                           (m'=0)
```

On vérifie que : **P4 = Q8**

- INSTRUCTION "SAVE", i = 84 :

```
        P                                   μP
    (m=1)                               (m'=1)
      |                                   |
CO P4 (16ρ2)⊤(2*16)|1+2⊥CO          RADR←Q5:16ρ0
MV[0;]←P5:P4                          RDON←Q6:Q2
      |                           MV[2⊥Q5;]←Q7:Q6
    (m=0)                               |
                                      (m'=0)
```

On vérifie que : P4 = Q2

- INSTRUCTION "RETURN", i = 85 :

```
        P                                   μP
    (m=1)                               (m'=1)
      |                                   |
CO←P4:(16ρ2)⊤(2*16)|¯2          CO←Q5:(16ρ2)⊤(2*16)|¯1
      |                           CO←Q6:(16ρ2)⊤(2*16)|(2⊥Q5)-1
    (m=0)                               |
                                      (m'=0)
```

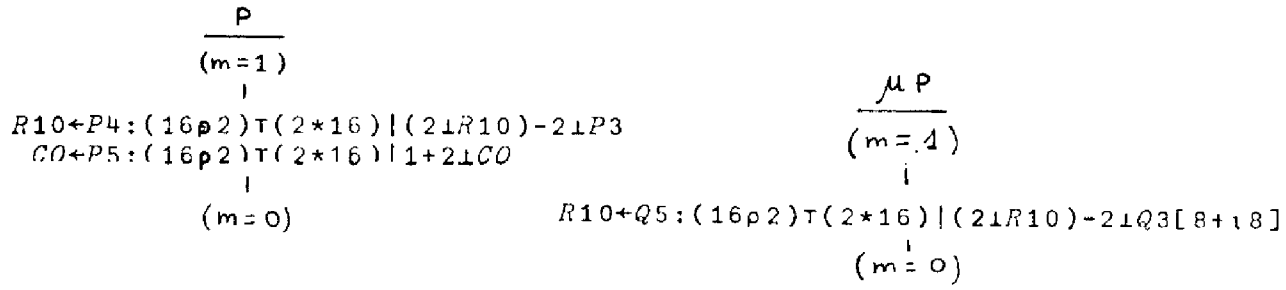
On vérifie que : P4 = Q6

- INSTRUCTION "CALL12", i = 86 :

```
        P                                   μP
    (m=1)                               (m'=1)
      |                                   |
CO←P4:R12                           CO←Q5:R12
      |                                   |
    (m=0)                               (m'=0)
```

On vérifie que : P4 = Q5

- INSTRUCTION "DECR10", i = 87 -



On vérifie que : P4 = Q5
P5 = Q2

Comme pour l'étude de la S- Machine, la vérification des conditions C2, C3 et C4 par la relation R de simulation est directe puisque celle-ci ne comporte que des relations d'égalité.

III-3 DISCUSSION

Au cours de cette étude de vérification nous avons trouvé les irrégularités suivantes :

- Une erreur dans la réalisation microprogrammée des instructions JRP et JRN dont l'effet était de provoquer un saut dans le programme à une instruction qui se trouve à l'adresse : (ad. de l'inst. en cours d'exécution) + av + 1 au lieu de l'adresse prévue par le programmeur qui est : (ad. de l'inst. en cours d'exécution) + av.

Il est évident que la manière la plus simple de corriger cette erreur est de changer dans le programme la valeur "av" associée à ces instructions par la valeur "av-1".

- Une implémentation imparfaite de SHL qui serait l'origine d'erreurs importantes dans le cas où le nombre de décalages demandés serait zéro.

La correction de cette imperfection est simple mais demande l'introduction d'une microinstruction de branchement supplémentaire ce qui entraîne un changement des codes opérations des instructions TEQ, MOVE, READ,

WRITE, INDEX.

- Une répétition du rôle de l'instruction INIT par l'instruction OUT, ce qui entraîne un accroissement non obligatoire de l'encombrement de la mémoire de microprogramme.

- Dans l'instruction TEQ dont le rôle est de sauter l'instruction suivante dans le programme ou de continuer en séquence selon le résultat du test de l'égalité entre "val" et M(ADR), nous avons trouvé que, dans le cas où val $\neq$ M(ADR), alors au lieu de poursuivre le programme en séquence (comme il est montré par le niveau "spécification") on effectue un saut de deux instructions.

Nous reproduisons dans la figure III-11. la partie de l'organigramme correspondant à l'instruction TEQ où se trouve l'erreur et nous montrons dans la figure III-12. la correction nécessaire qui nécessite simplement le changement d'une microinstruction dans le microprogramme.

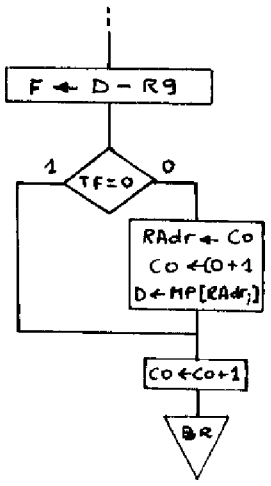


Figure III-11

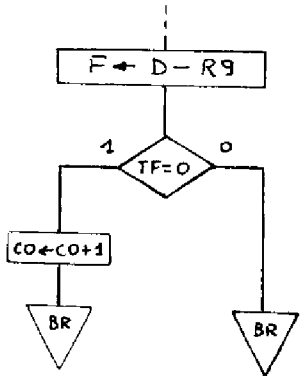


Figure III-12

Nous pouvons faire maintenant un bilan de l'ensemble de la procédure de vérification de systèmes microprogrammés pour ce cas particulier. Ce bilan peut être résumé de la manière suivante :

1) Etablissement d'une structure abstraite, dite description du niveau "spécification", en concordance avec les programmes qui définissent le fonctionnement global du système. Cette structure ne tient compte que d'une manière très générale de la véritable structure du système matériel ; par conséquent elle sera libérée de tout un ensemble de détails nécessaires pour le fonctionnement réel.

2) Traduction du contenu binaire de la mémoire de microprogramme sous une forme d'organigramme qui représente le niveau "réalisation microprogrammée". Ce niveau suit de près le fonctionnement du système et doit, donc, tenir compte de toutes les actions du processeur pour l'exécution de chaque instruction ainsi que du support matériel de ces actions : registres internes, opérations arithmétiques et logiques de l'UAL, différentes possibilités de test etc... Ceci entraîne une plus grande complexité de définition par rapport au niveau "spécification".

3) Etablissement, en tenant compte du fonctionnement attendu du processeur, d'une relation de simulation entre le niveau "spécification" et

le niveau "réalisation microprogrammée" et ensuite vérification de l'accord entre cette relation et certaines conditions fournies par la théorie de base de la simulation algébrique entre programmes.

Dans cette troisième partie de la procédure nous considérons que l'on doit s'astreindre à la plus grande rigueur théorique possible ; l'expérience montre que les procédures de vérification basées seulement sur des méthodes intuitives sont souvent relativement peu efficaces.

On retrouve ainsi un résultat plus général en programmation : le formalisme rend plus complexe et plus long l'établissement des spécifications mais le travail correspondant peut réduire considérablement les procédures et le temps de mise au point et de maintenance.

C O N C L U S I O N

Le travail exposé dans ce mémoire concerne la vérification des réalisations microprogrammées afin de déceler, par une approche rigoureuse, les erreurs de conception des microprogrammes, erreurs d'autant plus fréquentes que les microprogrammes sont optimisés.

Pour ce faire, il est en premier lieu nécessaire de disposer d'une double description du système et ceci selon deux niveaux : d'abord un niveau spécification fonctionnelle, ou spécification abstraite, déterminée par l'ensemble des primitives de haut niveau le caractérisant ; ensuite un niveau réalisation microprogrammée qui repose sur l'implémentation réelle du système et qui spécifie en particulier le chemin des données et le contenu de la mémoire de microprogramme. La vérification est alors menée en analysant les deux descriptions et en montrant qu'elles réalisent effectivement les mêmes opérations.

Afin de procéder à une vérification rigoureuse, il apparaît aussitôt nécessaire de disposer d'un outil de description des deux niveaux, lui-même précis et sans aucune ambiguïté. La première solution, développée par BIRMAN, repose sur l'utilisation du langage VDL. En fait une telle description demande afin d'être facilement manipulable, un effort de programmation considérable se traduisant par l'écriture d'un interpréteur symbolique. Devant cette difficulté, nous avons préféré utiliser des descriptions sous forme d'organigrammes qui dans le cas de processeurs à exécution non parallèle, fournissent une description simple et précise consolidée par l'utilisation des opérateurs APL pour spécifier les opérations de base. De plus, même si l'on ne cherche pas à réaliser une étude de vérification, il faut souligner qu'une description formelle du fonctionnement des processeurs paraît particulièrement intéressante afin de disposer d'une documentation complète et non ambiguë utilisable en particulier lors des étapes de mise au point et de modification du système.

A partir de ces organigrammes, la procédure de vérification que nous avons proposée consiste à modifier chacun des deux organigrammes de telle façon que les deux organigrammes résultants soient isomorphes.

Ceci est réalisé par une procédure, pas à pas, qui consiste à modifier l'un des organigrammes et après chacune des étapes, à prouver que l'organigramme modifié est une simulation de l'organigramme de départ. Cette procédure de vérification s'est avérée efficace et nous a permis de déceler certaines erreurs dans une mémoire de microprogramme.

Néanmoins, le travail correspondant est relativement lourd à effectuer de façon manuelle et il est certain qu'un système automatique de vérification serait d'une grande utilité. En fait, il semble possible de réaliser de façon automatique une grande partie de l'ensemble de la procédure. Ainsi, l'approche basée sur l'utilisation du langage VDL nécessite au minimum la programmation d'un exécuteur symbolique, capable de manipuler l'arbre de contrôle de la machine abstraite, ce qui nécessite un travail considérable mais allégerait la tâche de vérification d'une manière importante. Un certain nombre de travaux ont été réalisés dans ce domaine (3), (2), mais le problème de l'automatisation n'est pas encore entièrement résolu. De plus, l'existence de boucles dans les organigrammes complique encore le problème.

La solution automatique nous paraît passer, dans ce cas par l'utilisation d'un programme de démonstration de théorèmes afin de vérifier les implications nécessaires entre les différentes assertions et invariants.

Ainsi, si l'on considère que la validation automatique nécessite au minimum un exécuteur symbolique et un démonstrateur de théorèmes, alors, il nous semble que la procédure manuelle présentée ici est justifiée si l'on s'intéresse à des processeurs de taille et de complexité réduites tels que ceux qui sont traités dans ce mémoire.

A N N E X E S

A - Description VDL/APL

de la S- Machine

Definition of S

• *Abstract syntax of S*

is-S = (⟨*mem* : *is-mem*⟩,
⟨*stk* : *is-reg*⟩,
⟨*x*: *is-reg*⟩,
⟨*cc*: *is-reg*⟩,
⟨*sw*: *is-bit*⟩,
⟨*s-control*: *is-control*⟩,
⟨*s-lib*: *is-lib*⟩)

• *Initial state*

stk: (32ρ2)T2*24
s-control(*S*) : *exec-pgm*

• *Macro library*

1. *exec-pgm* =
is-run(*S*) → *exec-pgm*
exec-inst (*i*)
i: *fetch-inst*
else → Ω

2. *fetch-inst* =
PASS: *build-inst* (*a*)
a: *fetch-word* (*cc*)

3. *fetch-word* (*t*) =
PASS: *mem* [*m* :]
m: 2 ⊥ *t* [8 + ι 24]

4. *build-inst* (*t*) =
PASS: μ_0(⟨*s-id*:*id*⟩, ⟨*s-ix*:*ix*⟩,
⟨*s-op*:*op*⟩, ⟨*s-ad*:*ad*⟩)
id: *t*[0]
ix: *t*[1]
op: *t*[2 + ι6]
ad: *t*[8 + ι24]

5. *exec-inst* (*i*) =
is-load (*i*) → *load-stk* (*a*)
a: *fetch-word* (*b*)
b: *calc-addr* (*i*)
adv-ctr
is-ldi(*i*) → *load-stk* (8ρ0, *s-ad* (*i*))
adv-ctr
is-store(*i*) → *store-word* (*a*, *b*)
a: *pop-stk*
b: *calc-addr* (*i*)
adv-ctr
is-branch (*i*) → *cond* (*i*) → *cc*: *calc-addr* (*i*)
else → *adv-ctr*
is-enter (*i*) → *cc*: *calc-addr* (*i*)
load-stk (*cc*)
adv-ctr
is-ldx (*i*) → *x* : *fetch-word* (*a*)
a: *calc-addr* (*i*)
adv-ctr
is-ldxi (*i*) → *x* : 8ρ0, *s-ad* (*i*)
adv-ctr
is-loop (*i*) → *test-loop*(*i*)
incr-x
is-ls(*i*) → *store-word* (*a*, *stk*)
a: *ls* (*b*, 2 ⊥ *s-ad* (*i*))
b: *fetch-word* (*stk*)
adv-ctr
is-rs(*i*) → *store-word* (*a*, *stk*)
a: *rs* (*b*, 2 ⊥ *s-ad* (*i*))
b: *fetch-word* (*stk*)
adv-ctr
is-bin (*i*) → *store-word* (*a*, *stk*)
a: *bin* (*b*, *c*, *i*)
c: *fetch-word* (*stk*)
b: *pop-stk*
adv-ctr
is-not (*i*) → *store-word* (*a*, *stk*)
a: ∼ *fetch-word* (*a*, *stk*)
adv-ctr
is-xts (*i*) → *load-stk* (*x*)
adv-ctr
is-stx (*i*) → *x* : *pop-stk*
adv-ctr
is-adx (*i*) → *x*: *add* (*x*, *a*)
a: *pop-stk*
adv-ctr
is-sbx (*i*) → *x* : *sub* (*x*, *a*)
a : *pop-stk*
adv-ctr
is-ret (*i*) → *cc* : *pop-stk*
is-pop (*i*) → *pop-stk*
adv-ctr
is-stop (*i*) → *sw*(*s*) : 0
adv-ctr

6. *is-branch* (*i*) =
PASS : *is-tra* (*i*) ∨ *is-tpl* (*i*) ∨ *is-tmi* (*i*) ∨
is-tze (*i*) ∨ *is-tnz* (*i*)

7. *cond*(*i*) =
PASS: *is-tra* (*i*) ∨
(*is-tpl* (*i*) ∧ (*a*[0] = 0)) ∨
(*is-tmi* (*i*) ∧ (*a*[0] = 1)) ∨

```
        (is-tze (i) ∧ (0 = 2⊥a)) ∨
        (is-tnz (i) ∧ (0 ≠ 2⊥a))
          a : mem [2 ⊥ stk [8 + ι 24];]
```

8. *store-word* (a, t) =
   mem [2 ⊥ t[8 + ι 24];] : a

9. *load-stk* (a) = *store-word* (a, stk)
   push-stk

10. *push-stk* =
    stk : (32ρ2) ⊤ (2*32)| ¯1 + 2 ⊥ stk

11. *pop-stk* = stk : (32ρ2) ⊤ (2*32) | 1 + 2 ⊥ stk
    PASS : *fetch-word* (stk)

12. *adv-ctr* = cc : (32ρ2) ⊤ (2*32) | 1 + 2 ⊥ cc

13. *calc-addr* (i) = *calc-id* (i, a)
    a: *calc-ix(i)*

14. *calc-ix(i)* =
    (1 = *s-ix* (i)) → *PASS* : (32ρ2) ⊤ (2*32) | b + c
    b: 2 ⊥ x
    c: 2 ⊥ *s-ad(i)*
    else → *PASS*: 8ρ0, *s-ad(i)*

15. *calc-id* (a, i) =
    (1 = *s-id(i)*) → *PASS*: 8ρ0, b[8 + ι 24]
    b : *fetch-word* (a)
    else → *PASS*: a

16. *incr-x* =
    x : (32ρ2) ⊤ (2*32) | 1 + 2 ⊥ x

17. *test-loop* (i) =
    (0 ≠ 2 ⊥ x) → cc : *calc-addr* (i)
    else → *adv-ctr*

18. *ls* (a, N) =
    PASS: (a, Nρ0)[N + ι 32]

19. *rs* (a, N) =
    PASS: ((Nρa[0]), a) [ι 32]

20. *is-bin* (i) =
    PASS: *is-add* (i) ∨ *is-sub* (i) ∨ *is-and* (i) ∨
    is-or (i) ∨ *is-eor* (i)

21. *bin* (a, b, i) =
    is-add (i) → *PASS*: *add* (a, b)
    is-sub (i) → *PASS*: *sub* (a, b)
    is-and (i) → *PASS*: a ∧ b
    is-or (i) → *PASS*: a ∨ b
    is-eor (i) → *PASS*: (a ∧ ∼ b) ∨ (∼ a ∧ b)

22. *add* (a, b) =
    PASS: (32ρ2) ⊤ (2*32)|(2 ⊥ a) + 2 ⊥ b

23. *sub* (a, b) =
    PASS: (32ρ2) ⊤ (2*32) | (2 ⊥ a) − 2 ⊥ b

24. *is-run* (S) =
    PASS: sw(S) = 1

25. *is-load* (i) =
    PASS: 0 = 2 ⊥ *s-op* (i)
    is-ldi (i) =
    PASS: 1 = 2 ⊥ *s-op* (i)
    is-store (i) =
    PASS: 2 = 2 ⊥ *s-op* (i)
    is-tra (i) =
    PASS: 3 = 2 ⊥ *s-op* (i)
    is-tpl (i) =
    PASS: 4 = 2 ⊥ *s-op* (i)
    is-tmi (i) =
    PASS: 5 = 2 ⊥ *s-op* (i)
    is-tze (i) =
    PASS: 6 = 2 ⊥ *s-op* (i)
    is-tnz (i) =
    PASS: 7 = 2 ⊥ *s-op* (i)
    is-enter (i) =
    PASS: 8 = 2 ⊥ *s-op* (i)
    is-ldx (i) =
    PASS: 9 = 2 ⊥ *s-op* (i)
    is-ldxi (i) =
    PASS: 10 = 2 ⊥ *s-op* (i)
    is-loop (i) =
    PASS: 11 = 2 ⊥ *s-op* (i)
    is-ls (i) =
    PASS: 12 = 2 ⊥ *s-op* (i)
    is-rs (i) =
    PASS: 13 = 2 ⊥ *s-op* (i)
    is-add (i) =
    PASS: 32 = 2 ⊥ *s-op* (i)
    is-sub (i) =
    PASS: 33 = 2 ⊥ *s-op* (i)
    is-and (i) =
    PASS: 34 = 2 ⊥ *s-op* (i)
    is-or (i) =
    PASS: 35 = 2 ⊥ *s-op* (i)
    is-eor (i) =
    PASS: 36 = 2 ⊥ *s-op* (i)
    is-not (i) =
    PASS: 37 = 2 ⊥ *s-op* (i)
    is-xts (i) =
    PASS: 38 = 2 ⊥ *s-op* (i)
    is-stx (i) =
    PASS 39 = 2 ⊥ *s-op* (i)
    is-adx (i) =
    PASS: 40 = 2 ⊥ *s-op* (i)
    is-sbx (i) =
    PASS: 41 = 2 ⊥ *s-op* (i)
    is-ret (i) =
    PASS: 42 = 2 ⊥ *s-op* (i)
    is-pop (i) =
    PASS: 43 = 2 ⊥ *s-op* (i)
    is-stop (i) =
    PASS ≠ 44 = 2 ⊥ *s-op* (i)

26. *is-mem* (t) =
    is-binmatrix (t) → pt = (2*24 32) → *PASS* : 1
    else → *PASS* : 0
    else → *PASS* : 0
    is-reg (t) =
    is-binvector (t) → (pt) = 32 → *PASS*: 1
    else → *PASS*: 0
    else → *PASS*: 0

Definition of μS

• *Abstract syntax of μS*

```
is-μS = (⟨mem : is-mem⟩,
         ⟨stk  : is-reg⟩,
         ⟨cc   : is-reg⟩,
         ⟨sw   : is-bit⟩,
         ⟨cs   : is-controlstore⟩,
         ⟨mdr  : is-reg⟩,
         ⟨x    : is-reg⟩,
         ⟨a    : is-reg⟩,
         ⟨b    : is-reg⟩,
         ⟨mar  : is-adreg⟩,
         ⟨csar : is-csreg⟩,
         ⟨ir   : is-ireg⟩,
         ⟨s-control: is-control⟩,
         ⟨s-lib: is-lib⟩)
```

• *Initial state*

```
stk: (32ρ2) T 2*24
csar: 12ρ0
cs : MCODE1 (Appendix C)
s-control (μS): exec-μpgm
```

• *Macro library for μS*

```
1. exec-μpgm =
     is-run (μS)  → exec-μpgm
                    exec-irem
                      exec-μcycle
     else         → Ω
2. exec-irem =
     is-iexec(μS) → exec-irem
                    exec-μcycle
     else         → Ω
3. is-iexec (α) =
     PASS: (12ρ0) ≠ csar (α)
4. is-run (α) =
     PASS: 1 = sw(α)
5. exec-μcycle = exec-μi (i)
                 i : fetch-μi
6. fetch-μi = build μi (α)
              a : cs(μS)[2 ⊥ csar(μS);]
7. build-μi (t) =
     (1 = t[0]) → PASS: μ0(⟨s-branch : a⟩,
                           ⟨s-cond   : b⟩,
                           ⟨s-addr   : c⟩)
                    a : t [0]
                    b : t [1, 2, 3]
                    c : t [4 + ι 12]
     else       → PASS: μ0(⟨s-branch : a⟩,
                           ⟨s-memf   : b⟩,
                           ⟨s-in1    : c⟩,
                           ⟨s-in2    : d⟩,
                           ⟨s-f      : e⟩,
                           ⟨s-out    : f⟩)
                    a : t[0]
                    b : t[1, 2]
                    c : t[3, 4, 5]
                    d : t[6, 7, 8]
                    e : t[9 + ι 4]
                    f : t[13, 14, 15]
8. exec-μi (i) =
     (1 = s-branch (i)) → exec-branch (i)
     else               → exec-assign (i)
9. exec-branch (i)
     is-trm (i)  → csar (μS): s-addr (i)
     is-tφ (i)   → (mdr(μS)[0] = 0) → csar(μS) : s-addr (i)
                   else             → adv-csar
     is-t1 (i)   → (mdr(μS)[1] = 0) → csar(μS) : s-addr (i)
                   else             → adv-csar
     is-t2 (i)   → (mdr(μS)[2] = 0) → csar(μS): s-addr (i)
                   else             → adv-csar
     is-tmdr (i) → (mdr(μS) = 32ρ0) → csar(μS): s-addr (i)
                   else             → adv-csar
     is-ti(i)    → csar(μS) : s-addr(i) ∨ (7ρ0, ir(μS))
10. exec-assign (i) =
     is-stop (i) → sw(μS) : 0
                   csar(μS) : 12ρ0
     else        → exec-assign-1(i)
11. exec-assign-1(i) = mem (i)
                         set (i, a)
                         a: result (i, b, c)
                         b: select-1 (i)
                         c: select-2 (i)
                       adv-csar
12. select-1 (i)  =
     is-mdr-1 (i)  → PASS: mdr (μS)
     is-x-1 (i)    → PASS:   x (μS)
     is-mone-1 (i) → PASS: 32ρ1
     is-one-1 (i)  → PASS: 31ρ0, 1
     is-zero-1 (i) → PASS: 32ρ0
13. select-2 (i)  =
     is-one-2 (i)  → PASS: 31ρ0, 1
     is-zero-2 (i) → PASS: 32ρ0
     is-mask-2 (i) → PASS: 8ρ0, 24ρ1
     is-a-2 (i)    → PASS: a(μS)
```

```
      is-b-2 (i)      → PASS: b(μS)
      is-stk-2 (i)    → PASS: stk(μS)
      is-cc-2 (i)     → PASS: cc(μS)
14. result (i, a, b)  =
      is-add (i)      → PASS: (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                                              + 2 ⊥ b
      is-sub (i)      → PASS: (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                                              − 2 ⊥ b
      is-and (i)      → PASS: a ∧ b
      is-or (i)       → PASS: a ∨ b
      is-not (i)      → PASS: ∼ (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                                + 2 ⊥ b
      is-eor (i)      → PASS: (a ∧ ∼ b) ∨ (b ∧ ∼ a)
      is-rs (i)       → PASS: c[0], c[ι31]
                          c : (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                            + 2 ⊥ b
      is-ls (i)       → PASS: c[1 + ι31], 0
                          c : (32ρ2) ⊤ (2*32) | (2 ⊥ a)
                            + 2 ⊥ b
15. set (i, a)        =
      is-out-mar (i)  → mar(μS): a[8 + ι24]
      is-out-mdr (i)  → mdr(μS): a
      is-out-x (i)    → x(μS): a
      is-out-a (i)    → a(μS): a
      is-out-b (i)    → b(μS): a
      is-out-stk (i)  → stk(μS): a
      is-out-cc (i)   → cc(μS): a
      is-out-ir (i)   → ir(μS): a[3 + ι5]
16. mem (i)           =
      is-read (i)     → mdr(μS): mem(μS) [2 ⊥ mar(μS):]
      is-write (i)    → mem(μS)[2 ⊥ mar(μS):] : mdr (μS)
      is-p (i)        → Ω
17. adv-csar          =
      csar (μS): (12ρ2) ⊤ (2*12) | 1 + 2 ⊥ csar (μS)
18. is-trm (i)        = PASS: 0 = 2 ⊥ s-cond (i)
      is-tϕ (i)       = PASS: 1 = 2 ⊥ s-cond (i)
      is-t1 (i)       = PASS: 2 = 2 ⊥ s-cond (i)
      is-t2 (i)       = PASS: 3 = 2 ⊥ s-cond (i)
      is-tmdr (i)     = PASS: 4 = 2 ⊥ s-cond (i)
      is-ti (i)       = PASS: 7 = 2 ⊥ s-cond (i)
19. is-mdr-1 (i)      = PASS: 1 = 2 ⊥ s-in1 (i)
      is-x-1 (i)      = PASS: 2 = 2 ⊥ s-in1 (i)
      is-mone-1 (i)   = PASS: 3 = 2 ⊥ s-in1 (i)
      is-one-1 (i)    = PASS: 7 = 2 ⊥ s-in1 (i)
      is-zero-1 (i)   = PASS: 0 = 2 ⊥ s-in1 (i)
20. is-one-2 (i)      = PASS: 1 = 2 ⊥ s-in2 (i)
      is-zero-2 (i)   = PASS: 0 = 2 ⊥ s-in2 (i)
      is-mask-2 (i)   = PASS: 2 = 2 ⊥ s-in2 (i)
      is-a-2 (i)      = PASS: 3 = 2 ⊥ s-in2 (i)
      is-b-2 (i)      = PASS: 5 = 2 ⊥ s-in2 (i)
      is-stk-2 (i)    = PASS: 6 = 2 ⊥ s-in2 (i)
      is-cc-2 (i)     = PASS: 4 = 2 ⊥ in2 (i)
21. is-add (i)        = PASS: 0 = 2 ⊥ s-f (i)
      is-sub (i)      = PASS: 1 = 2 ⊥ s-f (i)
      is-and (i)      = PASS: 2 = 2 ⊥ s-f (i)
      is-or (i)       = PASS: 3 = 2 ⊥ s-f (i)
      is-not (i)      = PASS: 4 = 2 ⊥ s-f (i)
      is-eor (i)      = PASS: 5 = 2 ⊥ s-f (i)
      is-rs (i)       = PASS: 6 = 2 ⊥ s-f (i)
      is-ls (i)       = PASS: 7 = 2 ⊥ s-f (i)
      is-stop (i)     = PASS: 10 = 2 ⊥ s-f (i)
22. is-out-mar (i)    = PASS: 0 = 2 ⊥ s-out (i)
      is-out-mdr (i)  = PASS: 1 = 2 ⊥ s-out (i)
      is-out-x (i)    = PASS: 2 = 2 ⊥ s-out (i)
      is-out-a (i)    = PASS: 3 = 2 ⊥ s-out (i)
      is-out-b (i)    = PASS: 5 = 2 ⊥ s-out (i)
      is-out-stk (i)  = PASS: 6 = 2 ⊥ s-out (i)
      is-out-cc (i)   = PASS: 4 = 2 ⊥ s-out (i)
      is-out-ir (i)   = PASS: 7 = 2 ⊥ s-out (i)
23. is-read (i)       = PASS: 1 = 2 ⊥ memf (i)
      is-write (i)    = PASS: 2 = 2 ⊥ memf (i)
      is-p (i)        = PASS: 0 = 2 ⊥ memf (i)
24. is-mem (t) =
      is-binmatrix (t) → ρt = (2*24 32) → PASS: 1
                         else           → PASS: 0
      else             → PASS: 0
25. is-reg (t) =
      is-binvector(t) → (ρt) = 32 → PASS: 1
                        else      → PASS: 0
      else            → PASS: 0
26. is-controlstore(t) =
      is-binmatrix(t) → ρt = (2*12 16) → PASS: 1
                        else           → PASS: 0
      else            → PASS: 0
27. is-adreg(t) =
      is-binvector (t) → (ρt) = 24 → PASS: 1
                         else      → PASS: 0
      else             → PASS: 0
28. is-csreg(t) =
      is-binvector(t) → (ρt) = 12 → PASS: 1
                        else      → PASS: 0
      else            → PASS: 0
29. is-ireg(t) =
      is-binvector (t) → (ρt) = 5 → PASS: 1
                         else     → PASS: 0
      else             → PASS: 0
```

B - Microprogramme de la S- Machine

```
     V TAPARO
 [1]   ADDM,ZERO,CC,MAR,R
 [2]   ADDM,ONE,CC,CC,P
 [3]   TRM,(12ρ2)T168
 [4]   T2,(12ρ2)T7
 [5]   TI,(12ρ2)T32
 [6]   NADA
 [7]   T1,(12ρ2)T9
 [8]   ADDM,X,B,B,P
 [9]   T0,(12ρ2)T12
 [10]  ADDM,ZERO,B,MAR,R
 [11]  ANDM,MDR,MASK,B,P
 [12]  TI,(12ρ2)T64
 [13]  NADA
 [14]  NADA
 [15]  NADA
 [16]  NADA
 [17]  NADA
 [18]  NADA
 [19]  NADA
 [20]  ADDM,MONE,STK,STK,P
 [21]  ADDM,ZERO,STK,MAR,W
 [22]  TRM,(12ρ2)T1
 [23]  ADDM,MONE,STK,STK,P
 [24]  ADDM,ZERO,STK,MAR,W
 [25]  TRM,(12ρ2)T1
 [26]  ADDM,ZERO,B,MAR,W
 [27]  TRM,(12ρ2)T54
 [28]  NADA
 [29]  T0,(12ρ2)T70
 [30]  TRM,(12ρ2)T1
 [31]  NADA
 [32]  ADDM,ONE,STK,MAR,R
 [33]  TRM,(12ρ2)T115
 [34]  ADDM,ONE,STK,MAR,R
 [35]  TRM,(12ρ2)T115
 [36]  ADDM,ONE,STK,MAR,R
 [37]  TRM,(12ρ2)T115
 [38]  ADDM,ONE,STK,MAR,R
 [39]  TRM,(12ρ2)T115
 [40]  ADDM,ONE,STK,MAR,R
 [41]  TRM,(12ρ2)T115
 [42]  ADDM,ZERO,STK,MAR,R
 [43]  TRM,(12ρ2)T142
 [44]  ADDM,MONE,STK,STK,P
 [45]  TRM,(12ρ2)T144
 [46]  ADDM,ZERO,STK,MAR,R
 [47]  TRM,(12ρ2)T158
 [48]  ADDM,ZERO,STK,MAR,R
 [49]  TRM,(12ρ2)T147
 [50]  ADDM,ZERO,STK,MAR,R
 [51]  TRM,(12ρ2)T149
 [52]  ADDM,ZERO,STK,MAR,R
 [53]  TRM,(12ρ2)T156
 [54]  ADDM,ONE,STK,STK,P
 [55]  TRM,(12ρ2)T1
 [56]  STOPM,ZERO,ZERO,MAR,P
 [57]  NADA
 [58]  ADDM,ONE,STK,STK,P
 [59]  ADDM,ZERO,STK,MAR,R
 [60]  TMDR,(12ρ2)T126
 [61]  ADDM,ZERO,MDR,X,P
 [62]  ADDM,ZERO,ZERO,A,P
 [63]  TRM,(12ρ2)T92
 [64]  ADDM,ZERO,B,MAR,R
 [65]  TRM,(12ρ2)T20
 [66]  ADDM,ZERO,B,MDR,P
 [67]  TRM,(12ρ2)T23
 [68]  ADDM,ZERO,STK,MAR,R
 [69]  TRM,(12ρ2)T26
 [70]  ADDM,ZERO,B,CC,P
 [71]  TRM,(12ρ2)T1
 [72]  ADDM,ZERO,STK,MAR,R
 [73]  TRM,(12ρ2)T29
 [74]  ADDM,ZERO,STK,MAR,R
 [75]  TRM,(12ρ2)T97
 [76]  ADDM,ZERO,STK,MAR,R
 [77]  TRM,(12ρ2)T99
 [78]  ADDM,ZERO,STK,MAR,R
 [79]  TRM,(12ρ2)T101
 [80]  ADDM,ZERO,CC,MDR,P
 [81]  TRM,(12ρ2)T103
 [82]  ADDM,ZERO,B,MAR,R
 [83]  TRM,(12ρ2)T106
 [84]  ADDM,ZERO,B,X,P
 [85]  TRM,(12ρ2)T1
 [86]  ADDM,X,ONE,X,P
 [87]  TRM,(12ρ2)T108
 [88]  ADDM,ZERO,STK,MAR,R
 [89]  TRM,(12ρ2)T110
 [90]  ADDM,ZERO,STK,MAR,R
 [91]  TRM,(12ρ2)T110
 [92]  ADDM,MONE,STK,MAR,R
 [93]  ADDM,ZERO,STK,MAR,P
 [94]  TMDR,(12ρ2)T126
 [95]  TRM,(12ρ2)T138
 [96]  TRM,(12ρ2)T1
 [97]  T0,(12ρ2)T1
 [98]  TRM,(12ρ2)T70
 [99]  TMDR,(12ρ2)T70
 [100] TRM,(12ρ2)T1
 [101] TMDR,(12ρ2)T1
 [102] TRM,(12ρ2)T70
 [103] ADDM,MONE,STK,STK,P
 [104] ADDM,ZERO,STK,MAR,R
 [105] TRM,(12ρ2)T70
 [106] ADDM,MDR,ZERO,X,P
 [107] TRM,(12ρ2)T1
 [108] ADDM,X,ZERO,MDR,P
 [109] TRM,(12ρ2)T101
 [110] ADDM,MDR,ZERO,A,P
 [111] ADDM,ZERO,B,MDR,P
 [112] TRM,(12ρ2)T124
 [113] NADA
 [114] NADA
 [115] ADDM,MDR,ZERO,A,P
 [116] ADDM,ZERO,STK,MAR,R
 [117] ADDM,ONE,STK,MAR,R
 [118] TI,(12ρ2)T128
 [119] NADA
 [120] LSM,ZERO,A,A,P
 [121] TRM,(12ρ2)T123
 [122] RSM,ZERO,A,A,P
 [123] SUBM,MDR,ONE,MDR,P
 [124] TMDR,(12ρ2)T126
 [125] TI,(12ρ2)T120
 [126] ADDM,ZERO,A,MDR,W
 [127] TRM,(12ρ2)T1
 [128] ADDM,MDR,A,MDR,W
 [129] TRM,(12ρ2)T54
 [130] SUBM,MDR,A,MDR,W
 [131] TRM,(12ρ2)T54
 [132] ANDM,MDR,A,MDR,W
 [133] TRM,(12ρ2)T54
 [134] ORM,MDR,A,MDR,W
 [135] TRM,(12ρ2)T54
 [136] EORM,MDR,A,MDR,W
 [137] TRM,(12ρ2)T54
 [138] T32,(12ρ2)T140
 [139] ADDM,X,A,A,P
 [140] LSM,X,ZERO,X,P
 [141] TRM,(12ρ2)T151
 [142] NOTM,MDR,ZERO,MDR,W
 [143] TRM,(12ρ2)T1
 [144] ADDM,ZERO,STK,MAR,P
 [145] ADDM,X,ZERO,MDR,W
 [146] TRM,(12ρ2)T1
 [147] ADDM,MDR,ZERO,A,P
 [148] TRM,(12ρ2)T160
 [149] ADDM,MDR,ZERO,A,P
 [150] TRM,(12ρ2)T162
 [151] RSM,ZERO,MDR,MDR,P
 [152] TMDR,(12ρ2)T126
 [153] T32,(12ρ2)T140
 [154] TRM,(12ρ2)T139
 [155] NADA
 [156] ADDM,MDR,ZERO,CC,P
 [157] TRM,(12ρ2)T54
 [158] ADDM,MDR,ZERO,X,P
 [159] TRM,(12ρ2)T54
 [160] ADDM,X,A,X,P
 [161] TRM,(12ρ2)T54
 [162] SUBM,X,A,X,P
 [163] TRM,(12ρ2)T54
 [164] NADA
 [165] NADA
 [166] NADA
 [167] NADA
 [168] ANDM,MDR,MASK,B,P
 [169] NOTM,ZERO,MASK,A,P
 [170] ANDM,MDR,A,MDR,P
 [171] LSM,MDR,ZERO,IR,P
 [172] TRM,(12ρ2)T4
     V
```

C - Programme APL de simulation du niveau "spécification" de la S- Machine

```
     ∇ABSM[[]]∇
     ∇ ABSM
[1]   E1:→E2×⍳1=SW
[2]    →0
[3]   E2:'EXEC INST'
[4]    MDR←,MEM[2⊥CC;],0⍴V←0
[5]   'FETCH INST'
[6]   CC←,(32⍴2)⊤(2*32)|1+2⊥CC
[7]    'ADV CTR'
[8]    ID←,MDR[1],0⍴IX←,MDR[2],0⍴OP←,MDR[2+⍳6],0⍴AD←MDR[8+⍳24]
[9]    'BUILD INST'
[10]   →E3×⍳0=MDR[3]
[11]   'ZERO ADDRESS INST'
[12]   →E4×⍳36≥2⊥OP
[13]  ⍎OPR[1+2⊥OP;]
[14]  E11:' ***'
[15]   'END INST'
[16]  ' '
[17]   →E1
[18]  E4:A←,MEM[2⊥STK;],0⍴STK←(32⍴2)⊤(2*32)|1+2⊥STK,0⍴MDR←,MEM[2⊥STK;]
[19]   ⍎OPR[1+2⊥OP;]
[20]   MEM[2⊥STK;]←MDR
[21]   →E11
[22]  E3:'ONE ADDRESS INST'
[23]   'CALC ADD'
[24]   →E5×⍳1=IX
[25]   B←(8⍴0),AD
[26]  E6:→E7×⍳1=ID
[27]   →E8
[28]  E5:'INDEXATION'
[29]   B←(32⍴2)⊤(2*32)|(2⊥X)+2⊥AD
[30]   →E6
[31]  E7:'INDIRECTION'
[32]   B←;MEM[2⊥B;]
[33]  E8:⍎OPR[1+2⊥OP;]
[34]   →E9×⍳1=V
[35]   →E11
[36]  E9:CC←B,0⍴[←2⊥B,0⍴⍞←' **BRANCH TO '
[37]   →E11
     ∇
```

OPR

```
MEM[2⊥STK;]←MDR,0⍴STK←(32⍴2)⊤(2*32)|¯1+2⊥STK,0⍴MDR←MEM[2⊥B;],0⍴⍞←'LOAD'
MEM[2⊥STK;]←8⍴0,AD,0⍴STK←(32⍴2)⊤(2*32)|¯1+2⊥STK,0⍴⍞←'LDI'
MEM[2⊥B;]←MDR,0⍴STK←(32⍴2)⊤(2*32)|1+2⊥STK,0⍴MDR←MEM[2⊥STK;],0⍴⍞←'STORE'
CC←B,0⍴⍞←'TRA'
V←0=MDR[1],0⍴⍞←'TPL'
V←1=MDR[1],0⍴⍞←'TMI'
V←0=2⊥MDR,0⍴⍞←'TZE'
V←0≠2⊥MDR,0⍴⍞←'TNZ'
MEM[2⊥STK;]←MDR,0⍴STK←(32⍴2)⊤(2*32)|¯1+2⊥STK,0⍴MDR←CC,0⍴⍞←'ENTER'
X←MEM[2⊥B;],0⍴⍞←'LDX'
X←8⍴0,AD,0⍴⍞←'LDXI'
V←0≠2⊥X,0⍴X←(32⍴2)⊤(2*32)|¯1+2⊥X,0⍴⍞←'LOOP'
MEM[2⊥STK;]←MDR,0⍴MDR←(MDR,(2⊥AD)⍴0)[(2⊥AD)+⍳32],0⍴MDR←MEM[2⊥STK;],0⍴⍞←'LS'
MEM[2⊥STK;]←MDR,0⍴MDR←((2⊥AD)⍴0,MDR)[⍳32],0⍴MDR←MEM[2⊥STK;],0⍴⍞←'RS'

MDR←(32⍴2)⊤(2*32)|(2⊥A)+2⊥MDR,0⍴⍞←'ADD'
MDR←(32⍴2)⊤(2*32)|(2⊥A)-2⊥MDR,0⍴⍞←'SUB'
MDR←A∧MDR,0⍴⍞←'AND'
MDR←A∨MDR,0⍴⍞←'OR'
MDR←(A∧~MDR)∨(~A∧MDR),0⍴⍞←'EOR'
MEM[2⊥STK;]←MDR,0⍴MDR←~MEM[2⊥STK;],0⍴⍞←'NOT'
MEM[2⊥STK;]←X,0⍴STK←(32⍴2)⊤(2*32)|¯1+2⊥STK,0⍴⍞←'XTS'
STK←(32⍴2)⊤(2*32)|1+2⊥STK,0⍴X←MEM[2⊥STK;],0⍴⍞←'STX'
*(32⍴2)⊤(2*32)|(2⊥X)+2⊥MDR,0⍴STK←(32⍴2)⊤(2*32)|1+2⊥STK,0⍴MDR←MEM[2⊥STK;],0⍴⍞←'ADX'
X←(32⍴2)⊤(2*32)|(2⊥X)-2⊥MDR,0⍴STK←(32⍴2)⊤(2*32)|1+2⊥STK,0⍴MDR←MEM[2⊥STK;],0⍴⍞←'SBX'
STK←(32⍴2)⊤(2*32)|1+2⊥STK,0⍴CC←MEM[2⊥STK;],0⍴⍞←'RET'
STK←(32⍴2)⊤(2*32)|1+2⊥STK,0⍴MDR←MEM[2⊥STK;],0⍴⍞←'POP'
SW←0,0⍴⍞←'STOP'
```

D - Programme APL de simulation du niveau "réalisation microprogrammée" de la S- Machine

```
      ∇SEXM[⎕]∇
     ∇ STX;I
[1]   E1:→E2×⍳1=∨/SW
[2]    →0
[3]   E2:'***DEBUT MICRO***'
[4]    T←,T←CS[2⊥CSA?;]
[5]    →E3×⍳1=∨T[1]
[6]    FU←T[1+⍳4],0⍴R1←T[6 7 8],0⍴R2←T[9 10 11],0⍴RO←T[12 13 14],0⍴MT←T[15 16]
[7]    →E4×⍳8=2⊥FU,0⍴M←'OPERATION STOPS'
[8]    →E5×⍳10=2⊥FU
[9]    ±BUS1[∨/I;],0⍴I←I+1,0⍴I←2⊥51,0⍴M←'**SEL BUS1: '
[10]   ±BUS2[∨/J;],0⍴J←J+1,0⍴J←2⊥32,0⍴M←'**SEL BUS2: '
[11]   ±OPE[∨/K;],0⍴K←K+1,0⍴K←2⊥FU,0⍴M←'**OPE?:  '
[12]   ±BUSO[∨/L;],0⍴L←L+1,0⍴L←2⊥BO,0⍴M←'**SEL BUSO: '
[13]   ±MEMO[∨/M;],0⍴M←M+1,0⍴M←2⊥ME,0⍴M←'**OP MEM: '
[14]   CSAR←(12⍴2)⊤(2*12)|1+2⊥CSAR,0⍴M←'**ADV CSAR'
[15]   '***FIN MICRO ***'
[16]   →E1
[17]  E4:SW←0,0⍴CSAR←(12⍴2)⊤(2*12)|1+2⊥CSAR,0⍴M←'**WAITING'
[18]   →E1
[19]  E5:SW←0,0⍴CSAR←12⍴0,0⍴M←'**STOP'
[20]   →E1
[21]  E3:H←H+1,0⍴H←2⊥TE,0⍴TE←T[2 3 4],0⍴AD←T[4+⍳12]
[22]   ±TEST[∨/H;],0⍴M←'**TEST BRANCH: '
[23]   →E6×⍳2=∨/G
[24]   →E7×⍳1=∨/G
[25]   CSAR←(12⍴2)⊤(2*12)|1+2⊥CSAR,0⍴M←'**NO BRANCH'
[26]   ' '
[27]   →E1
[28]  E7:CSAR←AD,0⍴M←2⊥AD,0⍴M←'**BRANCH TO   '
[29]   →E1
[30]  E6:M←2⊥CSAR,0⍴M←'**BRANCH TO   '
[31]   →E1
      ∇
```

```
      BUS1
O1←32⍴0,0⍴M←'ZERO'
O1←MDR,0⍴M←'MDR'
O1←X,0⍴M←'X'
O1←32⍴1,0⍴M←'MONE'
O1←(31⍴0),1,0⍴M←'ONE'

      BUS2
O2←32⍴0,0⍴M←'ZERO'

O2←(8⍴0),24⍴1,0⍴M←'MASK'
O2←A,0⍴M←'A'
O2←CC,0⍴M←'CC'
O2←B,0⍴M←'B'
O2←STK,0⍴M←'STK'
O2←(31⍴0),1,0⍴M←'ONE'

      BUSO
MAR←R[8+⍳24],0⍴M←'MAR'
MDR←R,0⍴M←'MDR'
X←R,0⍴M←'X'
A←R,0⍴M←'A'
CC←R,0⍴M←'CC'
B←R,0⍴M←'STK←I',0⍴M←'STK'
R←R[3+⍳5],0⍴M←'I2'

      OPE
R←,(32⍴2)⊤(2*32)|(2⊥O1)+(2⊥O2),0⍴M←'ADD'
R←,(32⍴2)⊤(2*32)|(2⊥O1)-(2⊥O2),0⍴M←'SUB'
R←,O1∧O2,0⍴M←'AND'
R←,O1∨O2,0⍴M←'OR'
R←,~(32⍴2)⊤(2*32)|(2⊥O1)+(2⊥O2),0⍴M←'NOT'
R←,(O2∧~O1)∨(O1∧~O2),0⍴M←'SOU'
R←,0,R[⍳31],0⍴R←,(32⍴2)⊤(2*32)|(2⊥O1)+(2⊥O2),0⍴M←'LS'
R←,R[1+⍳31],0,0⍴R←,(32⍴2)⊤(2*32)|(2⊥O1)+(2⊥O2),0⍴M←'LS'

      MEMO
M←'NADA'
MDR←MEM[2⊥MAR;],0⍴M←'READ'
MEM[2⊥MAR;]←MDR,0⍴M←'WRITE'

      TEST
G←1,0⍴M←'TRU'
G←~MDR[1],0⍴M←'T0'
G←~MDR[2],0⍴M←'T1'
G←~MDR[3],0⍴M←'T2'
G←G←∨/MDR,0⍴M←'TMD?'
G←~MDR[32],0⍴M←'T31'

G←2,0⍴CSAR←AD∨(7⍴0),I2,0⍴M←'TI'
```

B I B L I O G R A P H I E

---

(1) BIRMAN A.

"On proving correctness of microprograms" IBM, J. Rech. Develop., vol 18, May 1974.

(2) BIRMAN A., JOYNER W.

"A problem-reduction approach to proving simulation between programs" IEEE Transactions on Software Engineering, June 1976.

(3) CARTER W.C., JOYNER W., LEEMAN G.B.

"Automated experiments in validating microprograms"

5th Fault Tolerant Comp. Symp., Paris, 1975.

(4) Mc CARTHY

"Towards a Mathematical theory of computation"

Proc. IFIP Congress 1962

North Holland Publ. Co. Amsterdam 1962.

(5) DAHL O. J., DIJKSTRA E. W., HOARE C. A. R.

"Structured programming"

Academic Press, London and New York, 1972.

(6) FLOYD R. W.

"Assigning meanings to programs"

Proc. of Symp. in Applied Math. Vol 19

American Mathematical Society, 1967.

(7) GEAR C. W.

"Computer organization and programming"

Mc Graw-Hill Book Co. INC New York, 1969.

(8) HARALSON K., POLIVKA R.
"Microprogram training - an APL application"
Proc. 4th. Int. APL Users Conf., 1962.

(9) HOARE C.
"Proof of a program : FIND"
Comm. of the ACM, Jan 1971, Vol 14, N1.

(10) IVERSON K.
"A Programming Language"
John Wiley et Sons, Inc. New York, 1962.

(11) JAWORSKY A.
"Un système d'aide à la programmation de microprocesseur"
Revue technique THOMPSON - CSF, Juin 1976.

(12) KATZ S., MANNA Z.
"Logical analysis of programs"
Comm. of the ACM, April 1976, Vol 19, N4.

(13) LUCAS P., LAVER P., STIGLEITNER H.
"Method and notation for the formal definition of programming languages"
IBM Lab. Vienna, Tech. Rep. TR 125 087. June 1968.

(14) LEE J.
"Computer Semantics"
Comp. Science Series, Van Nostrand Reinhold. 1972

(15) LEEMAN G. B.

"Some problems in certifying microprograms"

IEEE Transact. on computers, May 1975.

(16) MANNA Z.

"Mathematical theory of computation"

Mc Graw - Hill, Computer Science Series, 1974.

(17) MILNER R.

"An algebric définition of simulation between programs"

Report CS 205, Standford University, Calif, Feb 1971.

(18) RENALIER J., AZEMA P., VALETTE R.

"Programme de simulation et d'analyse de schémas à réseaux de Pétri, en langage APL"

Congres AFCET, Paris Mars 1977.

(19) RAULT J. C., DEMARS G., RUGGIU G.

"Le langage et les systèmes APL"

Masson et Cie. Paris, 1974.

(20) R. DE OLANO A., DIAZ M., AZEMA P.

"Description and verification of microprogrammed digital computers, using APL"

CAD Seminar - Budapest, Juillet 1976.

(21) BEOUNES C., CEREJA F., LAPRIE J.C., PONS J.M.

"Commande de processus à base de microprocesseurs et de circuits à haut degré d'intégration"

contrat DRME - SNECMA n° 74/431.

Lot n°3 - Avril 1977 publication LAAS n° 1553.

- AMD 2900 Tech. NOTE.

TABLE DES MATIERES