UNIVERSITÉ D'ALGER

ÉCOLE NATIONALE POLYTECHNIQUE

DEPARTEMENT ECONOMIE

THÈSE DE FIN D'ÉTUDES

SYSTEME MIX

SUJET

Proposé et dirigé par :
Mr N BOUMAHRAT
Mr F. D. ARMINGAUD

Etudié par :
Mlle H. AISSAOUI
Mrs M. ACHAIBOU
A. BENMANSOUR
B. SEDDIKI

Année 1974

A. SADJI Saïd

- Son étroite collaboration à la réalisation de ce travail nous a été d'un très grand secours.

Qu'il veuille bien trouver ici le témoignage de notre profonde gratitude et l'expression de notre vive amitié.

A NOTRE PROFESSEUR N. BOUMAHRAT :

Les cours d'informatique qu'il nous a dispensés nous ont permis d'aborder le présent travail avec beaucoup plus de sénérité et de maitrise .

Tout au long de nos études à l'ENPA , nous avons toujours trouvé au département économie dont il est le chef, une trés grande compréhension et une atmosphére accueillante.

Il nous a inspiré le sujet de cette thése et a veillé à sa réalisation avec un soin bienveillant .

Qu'il veuille bien trouver ici l'hommage de notre vive reconnaissance et de notre profond respect.

A François Dominique ARMINGAUD :

Il a dirigé la réalisation de cette thése et nous a, chaque jour, prodigué ses encouragements .

Durant notre scolarité de 5ème année, nous avons reçu son enseignement dont nous avons apprécié le réalisme et la rigueur scientifique .

Sa culture, son humanisme nous ont rappelé que l'ingénieur et le professeur, dans l'exercice quotidien de leur profession devaient faire preuve d'humilité .

Qu'il nous soit permis de lui exprimer aujourd'hui, toute notre reconnaissance et notre joie de le compter parmi nos amis, et qu'il daigne accepter l'hommage de notre admiration et de notre sincére attachement .

SYSTEME UNIX

S O M M A I R E

Pages

Introduction
Description de MIX

PREMIERE PARTIE : ASSEMBLEUR

CHAPITRE I: Langages symboliques et techniques de traduction

I-A : But de l'Assembleur MIX 1
I-B : Langages symboliques 1
I-C : Technique de Traduction 3

CHAPITRE II: Assembleurs et langages d'Assembleurs

II-A : Introduction 4
II-B : Principe d'un Assembleur 5
II-C : Codes et opérandes symboliques 6
II-D : Instructions 7
II-E : Pseudo-Instructions 8

CHAPITRE III: Le Langage d'Assemblage MIX 10

III-A : Exemple 1 10
III-B : Exemple 2 14
III-C : Régles 21

CHAPITRE IV: Assemblage 27

IV-A : Fonctionnement général 27
IV-B : Fonctions fondamentales 28

CHAPITRE V: Générations des Instructions

V-A : Format du langage d'Assemblage MIX 31
V-B : Schéma et description de l'Assembleur MIX .. 33
V-C : Sous-programme LECT 35

.../...

V-D : Table des Identificateurs ou table des symboles : PAS 1 35
V-E : Traduction du programme source (deuxiéme passage) 36
V-F : Sous-programme CODE 37
V-G : Sous-programme BIND 39
V-H : Sous-programme AUTOM 40

CHAPITRE VI: Simulation de l'Assembleur 43

DEUXIEME PARTIE : SIMULATEUR

CHAPITRE I: Présentation et Simulation des instructions MIX 53

* Section 0 : Sous-programmes de service 54
- Sous-programme ROME 54
- Sous-programme ETIRE 55
- Sous-programme PACTE 55
- Sous-programme SEULS 56
- Sous-programme LOCAL 56

* Section I : Opérateurs de chargement , module C0823 58
* Section II : Opérateurs de stockage 61
* Section III : Opérateurs Arithmétiques 66
* Section IV : Opérateurs de modification des registres, module C4855 81
* Section V : Opérateurs de comparaison 86
* Section VI : Opérateurs de saut 89
* Section VII : Opérateurs d'Entées-Sorties 95
* Section VIII : Opérateurs de conversion, module C05 109

.../...

* Section IX : Opérateurs divers 114
- Opérateur MOVE 114
- Opérateurs de décalages 115
- Opérateur NOP 120
- Opérateur HLT 120
- Table des codes 121

CHAPITRE II / Liaison des différentes sections 123

CONCLUSION 128

ANNEXE 1 : Extension de MIX 130
ANNEXE 2 : Exemple de programme en MIX 131

BIBLIOGRAPHIE 139

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

MIX est le premier calculateur insaturable du monde. Comme la plupart des machines, il a un nombre qui l'identifie: le 1009. Ce nombre fut trouvé en prenant 16 calculateurs actuels qui sont trés semblables à MIX et sur lesquels Mix peut être aisément simulé.

(360 + 650 + 709 + 7070 + U3 + SS80 + 1107 + 1604 + 620 + B 220 + S2000 + 920 + 601 + 4800 + PDP4 + II) / 16 = 1009

Ce nombre peut aussi être obtenu d'une façon plus simple, en prenant les chiffres romains: 1009 ≡ MIX. MIX a une propriété particuliére, en ce sens qu'il est à la fois binaire et décimal.

Le programmeur ne sait pas en fait, s'il est en train de programmer sur une machine en bases arithmétiques 2 ou 10.

Ceci a été fait pour que les algorithmes écrits en MIX puissent être utilisés sur n'importe quel type de machine avec peu de changement, et ainsi pour que MIX puisse être facilement simulé sur n'importe quel type de machine.

Les programmeurs habitués à une machine binaire peuvent considérer MIX comme binaire, ceux habitués à une décimale, peuvent considérer MIX comme décimal.

Les programmeurs d'une autre planète peuvent considérer MIX comme une machine "ternaire".

MIX

REGISTRE A

±	A1	A2	A3	A4	A5

REGISTRE X

±	X1	X2	X3	X4	X5

REGISTRE I1

±	I14	I15

REGISTRE I2

±	I24	I25

REGISTRE I3

±	I34	I35

REGISTRE I4

±	I44	I45

REGISTRE I5

±	I54	I55

REGISTRE I6

±	I64	I65

REGISTRE J

+	J4	J5

OVERFLOW TOGGLE

E L G INDICATEUR COMPARAISON

MÉMOIRE

0000:						
0001:						
0002:						
0003:						
3998:						
3999:						

D E S C R I P T I O N DE M I X

I - DEFINITION DU MOT MIX:

L'unité de base d'information est le byte.

Chaque byte contient une quantité d'information non spécifiée, mais il doit être capable de contenir au moins 64 valeurs distinctes. Or, nous savons que n'importe quel nombre entre 0 et 63, inclus, peut être contenu dans un byte.

D'autre part, chaque byte contient au plus 100 valeurs distinctes. Sur un calculateur binaire, un byte doit donc être composé de 6 bits.

Sur un calculateur décimal, nous avons deux chiffres par byte.

Les programmes exprimés en language MIX, doivent être écrits de telle sorte qu'un byte ne contienne pas plus de 64 valeurs. Si nous voulons traiter le nombre 80, il faut toujours utiliser 2 bytes adjacents pour l'exprimer, même si un byte est suffisant sur un calculateur décimal.

Un Algorithme en MIX doit s'exprimer correctement quelque soit la taille du byte.

Bien qu'il soit tout à fait possible d'écrire des programmes qui dépendent de la taille du byte , c'est un acte illégal qui ne sera pas toléré. Les seuls programmes légitimes sont ceux qui donnent des résultats corrects quelque soit la taille du byte.

- 2 bytes adjacents peuvent exprimer les nombres de:
  0 à 4095.
- 3 bytes adjacents peuvent exprimer les nombres de:
  0 à 262.143.
- 4 bytes adjacents peuvent exprimer les nombres de:
  0 à 16.777.215.
- 5 bytes adjacents peuvent exprimer les nombres de:
  0 à 1.073.741.823.

Un mot machine est constitué de 5 bytes et d'un signe. Le signe a deux valeurs possibles: " + " et " - ".

II - LES REGISTRES

Il existe 9 registres en MIX (voir fig. I.B.).

- Le registre A (Accumulateur) : 5 bytes plus le signe
- Le registre X (Extension) : également 5 bytes plus le signe.
- Les registres I (Registres d'Index) I1, I2, I3, I4, I5, I6, chacun d'eux tient sur 2 bytes plus le signe.
- Le registre J (adresse de saut) tient sur 2 bytes et le signe est toujours " + " .

Nous utiliserons la lettre " r " préfixée au nom , pour identifier un registre MIX . Ainsi " rA " signifie " Registre A " .

" Le Registre A " a beaucoup d'utilisations et spécialement pour les opérations arithmétiques et les opérations sur les données.Le "Registre X " est une extension du " second membre " de rA; il est utilisé en connection avec rA pour tenir sur 10 bytes, quand on veut éffectuer un produit ou une division.

Il peut être utiliser pour sauvegarder l'information que l'on peut shifter de rA.

Les " Registres Index" sont, utilisés en premier lieu de compteur et de références à des adresses mémoires variables.

Le " Registre J " contient toujours l'adresse de l'instruction suivant l'instruction " JUMP " précédente; il est utilisé principalement en connection avec les subroutines.

En plus des registres cités au dessus, MIX contient :

- Un " overflow toggle (un seul bit qui est soit sur " ON " soit sur " OFF ")

- Un " indicateur de comparaison":

  il prend 3 valeurs:

  plus petit, égal ou plus grand (< , = , >)
- Une mémoire (4000 mots de stockage, chaque mot étant constitué de 5 bytes plus un signe .

III - CHAMPS PARTIEL DES MOTS:

Les 5bytes et le signe d'un mot machine sont numérotés comme suit:

0	1	2	3	4	5
±	Byte	Byte	Byte	Byte	Byte

(1)

La plupart des instructions permettent au programmeur d'utiliser une partie seulement d'un mot s'il le désire. Dans ce sens " une spécification de champ" est donnée. Les champs permis sont ceux qui sont adjacents dans un mot machine, ils sont représentés par (L:R) , où L est le nombre de la partie gauche et R le nombre de la partie droite du champ.

EXEMPLES DE SPECIFICATIONS DE CHAMP:

(0:0), le signe seulement
(0:2), le signe et les 2 premiers bytes
(0:5), tout le mot, c'est la spécification de champ la plus courante.
(1:5), tout le mot, sauf le signe.
(4:4), le 4éme byte seulement.
(4:5), Les deux derniers bytes.

L'utilisation de ces spécifications de champ varie légérement d'une instruction à une autre; elle sera expliquée en détail pour chaque instruction .

Bien que ce ne soit généralement pas important pour le programmeur, le champ (L:R) est noté dans la machine par le seul nombre 8L + R, et ce nombre tiendra sur un byte.

IV - FORMAT D'UNE INSTRUCTION MIX:

Les mots machines utilisés pour les instructions ont la forme suivante:

0	1	2	3	4	5
±	A	A	I	F	C

(2)

Le byte le plus à droite, C, est le code opération . Il indique l'opération qui doit être faite.

Par exemple, C = 8 est l'opération LDA (" charger le Registre A ").

Le byte F contient une modification du code opération.F est habituellement une spécification de champ (L:R) = 8L + R; Par exemple si C = 8 et F = 11, l'opération est"charger le registre A " avec le champ (1:3). Quelquefois F est utilisé dans d'autres buts.

Pour les instructions d'entrée-sortie par exemple, F est le nombre de l'unité d'entée ou de sortie affectée.

La portion gauche de l'instruction, ± AA, est " l'adresse " .(Notons que le signe fait partie de l'adresse). Le champ I qui vient aprés l'adresse, est la"spécification d'Index ", qui peut être utilisée pour modifier l'adresse d'une instruction .

Si I = 0 , l'adresse ± AA est utilisée sans changement. Autrement I contiendrait un nombre i entre 1 et 6, et le contenu du Registred'Index Ii est additionné algébriquement à ± AA. Le résultat est utilisé comme adresse de l'instruction. Ce processus d'indexation a lieu pour chaque instruction. Nous utiliserons la lettre M aprés que n'importe quelle indexation spécifiée se soit produite. (Si l'addition d'un Registre d'Index à l'adresse ± AA donne un résultat qui ne tient pas sur 2 bytes, la valeur de M est alors indéfinie.).

Dans la plupart des instructions, M référera à une " cellule mémoire". Les termes " cellules mémoire " et " location mémoire" sont utilisés presque indifférement . Nous supposons qu'il y a 4000 " cellules mémoire " numérotées de 0 à 3999. Chaque location mémoire peut donc être adressée sur 2 bytes. Pour chaque instruction dans laquelle M se référe à une cellule mémoire, nous devons avoir $0 \leq M \leq 3999$, et dans ce cas nous écrirons contenu (M) pour noter la valeur stockée dans la location M

Pour certaines instructions, l'adresse " M " a une autre signification, et peut même être négative (cas des incrémentations).

Ainsi une instruction additionnant M à un Registre d'Index tient compte du signe de M.

V - NOTATION:

Pour parler des instructions d'une maniére lisible, nous utiliserons la notation:

OP ADDRESS, I(F)

CODE OPERATION OPERANDE

Pour indiquer une instruction comme (2).

- Ici OP est un nom symbolique qui est donné au code opération (partie C) d'une instruction;
- ADDRESS est la position $\pm$ AA.
- I et F représentent respectivement les champs I et F .
- Si I est nul , "I" est omis.
- Si F est la spécification F normale pour cet opérateur, le "F" n'a pas besoin d'être écrit. La spécification F normale pour presque tous les opérateurs est (0:5); elle représente tout le mot MIX .

Si un F différent est standart, il sera mentionné explicitement quand nous aborderons le cas d'un opérateur particulier. Par exemple, l'instruction qui charge un nombre dans l'accumulateur est appelée " LDA " et son code opération est C = 8 .

Exemples:

Représentation conventionnelle

Instruction numérique réelle

LDA 2000,2(0:3).....	+	2000	2	3	8
LDA 2000,2(1:3)	+	2000	2	11	8
LDA 2000(1:3)	+	2000	0	11	8
LDA 2000	+	2000	0	5	8
LDA -2000,4	-	2000	4	5	8

Pour expliciter ceci, l'instruction "LDA 2000, 2(0:3)" peut être lue. "charger A, du contenu de la mémoire 2000 indexée par le registre 2, avec le champ (0:3)".

Pour représenter les contenus numériques d'un mot MIX , nous utiliserons toujours une notation en cases comme celle au dessus.

Notons que dans le mot:

+	2000	2	3	8

le nombre + 2000 remplit 2 bytes adjacents et le signe . Le contenu réel du byte (1:1) et du byte (2:2) variera d'un mot mémoire MIX à un autre, puisque la taille d'un byte est variable.

Voici un autre exemple de cette notation pour des mots MIX:

-	10000	3000

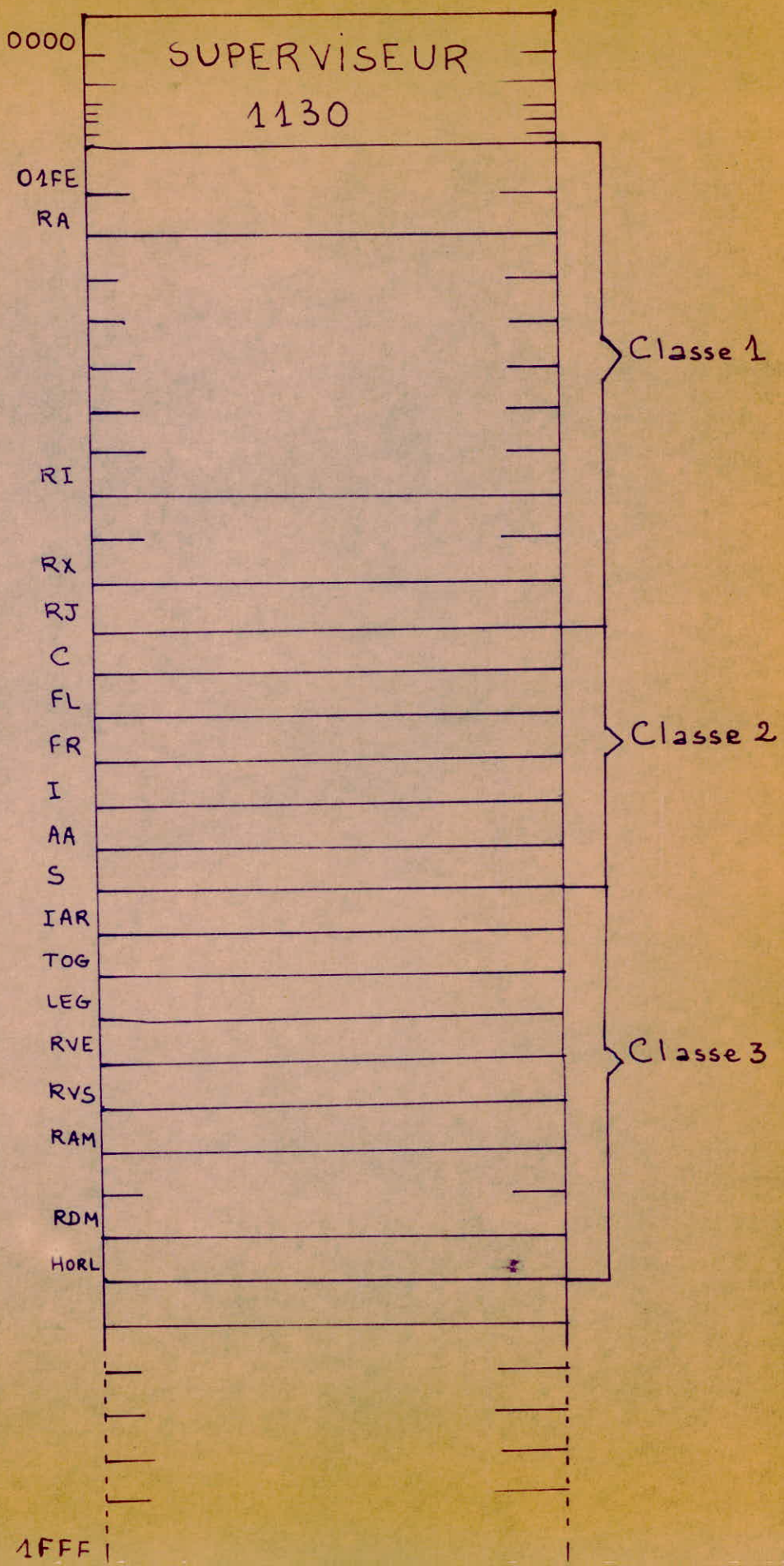
Ce diagramme représente un mot avec deux champs, un champ de 3 bytes plus le signe contenant - 10.000 et un champ de 2 bytes contenant 3000. Quand un mot est éclaté en plus d'un champ, on dit qu'il est tassé (" packed ") .

CARTE MEMOIRE

Une zone de mémoire fixe (510 à 536) a été reservée aux différents registres MIX (rA,rI1,rI2,rI3,rI4, rI5, rI6, rX, rJ) et à des registres de travail (S, AA, I, F, C, IAR...) .

Cette cartographie a été réalisée dans le but de permettre le transfert des paramètres entre les différentes instructions (registres MIX) et de faciliter la liaison entre les différentes parties du simulateur (registres auxiliaires) .

CARTE MÉMOIRE



Pour pouvoir programmer sur I B M 1130 en langage symbolique MIX, il est necessaire d'adapter ce langage sur ce type d'ordinateur.

Pour cela nous avons été amené à traiter un assembleur et un simulateur.

Aussi notre travail se décompose essentiellement de deux parties :

- l'assembleur
- le simulateur .

Dans une première partie nous présenterons l'assembleur, dans la deuxième le simulateur.

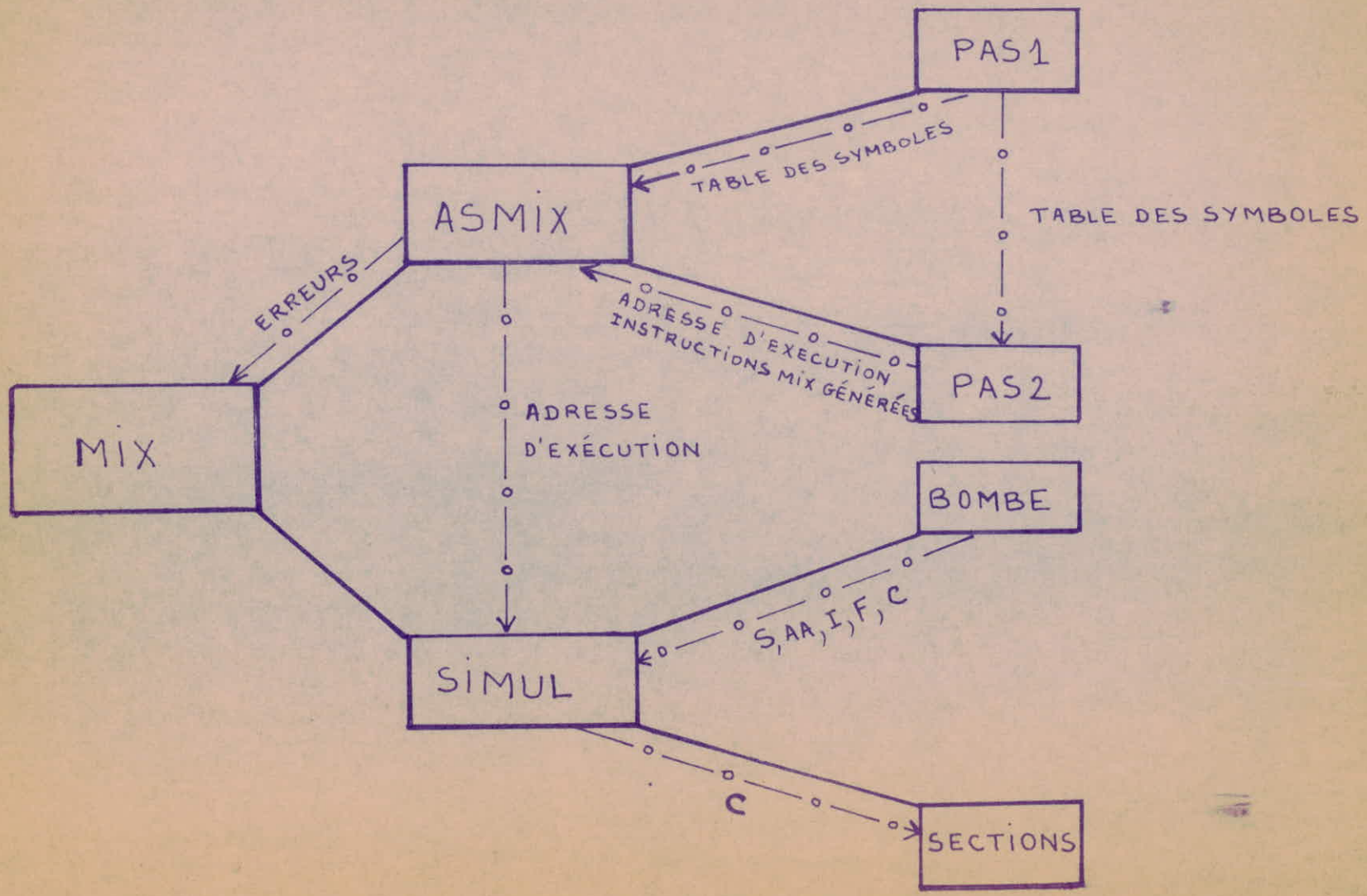


SCHÉMA GÉNÉRAL DU SYSTÈME MIX

PREMIERE PARTIE

L'ASSEMBLEUR

C H A P I T R E I

LANGAGES SYMBOLIQUES ET TECHNIQUES DE TRADUCTION

I - A : BUT DE L'ASSEMBLEUR MIX :

Etant donné un programme écrit dans le langage synthétique MIX, le but de l'assembleur est d'accepter comme données d'entrée ce programme et de produire en sortie ce même programme transposé en langage machine.

Ainsi le programme objet, généré par l'assembleur MIX,est à même d'être exécuté par le simulateur et de produire des résultats équivalents à ceux définis par le langage synthétique initial.

I - B : LANGAGES SYMBOLIQUES :

Ils sont encore trés proches du langage machine , mais ils suppriment les inconvénients liés à l'utilisation de codes et d'adresses numériques.
En effet le programmeur qui utilise un langage symbolique peut représenter ces codes et ces adresses par des symboles mnémotéchniques. Les symboles désignant les codes opération dans un langage donné sont définis une

.../...

fois pour toutes; par contre, les symboles désignant des adresses sont choisis par les programmeurs, étant entendu que certaines régles portant sur la formation de ces symboles doivent être respectées (nombre maximum de caractéres, obligation d'y faire apparaintre au moins un caractére alphabétique etc...) .

Malgré la grande similitude avec un programme écrit en langage machine, un programme symbolique ne peut être directement éxecuté par un ordinateur. Il doit être au préalable traduit .
Cette traduction est éffectuée par l'ordinateur sous le contrôle d'un programme spécialisé. Le programme écrit par le programmeur est appelé progme origine (programme source). Le programme obtenu aprés traduction est appelé programme résultant (programme objet) .

La programme traducteur utilisé pour passer du programme symbolique simple au programme langage machine est trés souvent appelé assembleur et, par extension, le langage symbolique simple, langage assembleur.

.../...

I - C : TECHNIQUES DE TRADUCTION :

Il y a deux grandes classes de techniques de traduction :

a) - Techniques d'interpretation .

Elle permettent l'execution directe de programmes écrits en langage symbolique.

b) - Techniques de compilation .

Elle réalisent la traduction en deux phases bien distinctes :

- Phase I : Le programme en langage évolué est transformé en un programme en langage machine.
- Phase 2 : Ce programme en langage machine est executé.

Les techniques d'interprétation correspondent à une traduction directe et dynamique caractérisée par le fait que le programme en langage symbolique est executé complétement au fur et à mesure de sa lecture , ce processus pouvant conduire soit à des éxecutions repetées, soit à la non-execution de certaines parties du programme, selon la nature du programme à traduire.

Les techniques de compilation correspondent à une traduction indirecte et statique caractérisée par le fait que leprogramme est transformé complétement en un programme en langage machine avant d'être éxecuté à ce dernier niveau.

C'est cette dernière technique que nous utiliserons dans notre étude.

.../...

C H A P I T R E II

ASSEMBLEURS ET LANGAGES D'ASSEMBLEURS

II - A : INTRODUCTION .

On désigne généralement par langage d'assembleur une forme évolué de langage machine symbolique caractérisé par :

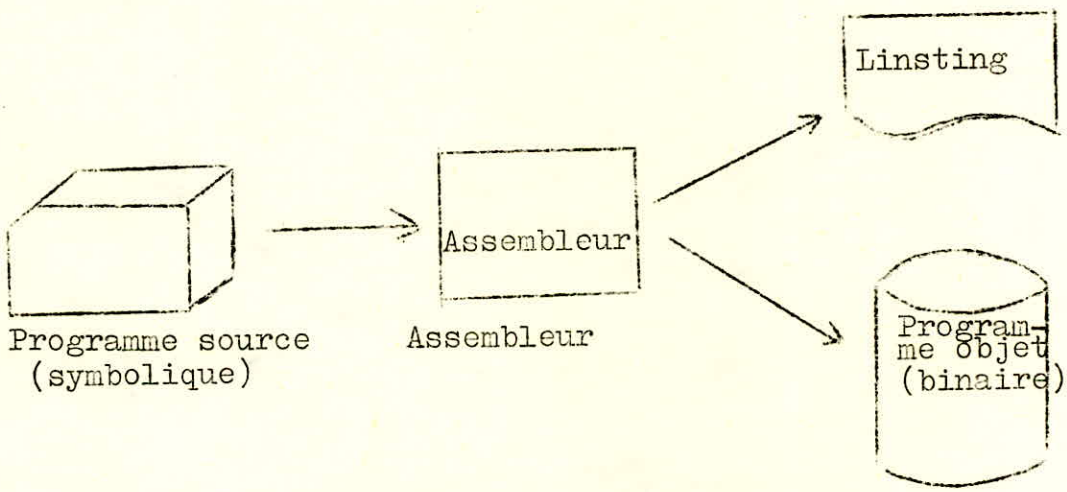
- L'utilisation systématique de symboles et d'expressions symboliques pour la représentation des codes opérations et des diverses adresses et indicateurs pour l'ensemble des instructions du langage machine (instructions machines symboliques) ;
- L'utilisation d'instructions d'assemblage (pseudo - instructions symboliques) ayant une forme identique à celle des instructions machine symboliques, et qui permettent de donner des directives à l'assembleur, c'est-à dire au programme chargé de la traduction des instruc - tions machine symboliques en instructions machine sous forme binaire. Ces pseudo-instructions sont traduites également en langage machine, mais la correspondance n'est pas systématiquement I ---o--- I (c'est à dire une ins - truction symbolique donne aprés traduction, une instruction binaire), elle peut être I ---o--- 0 (la pseudo-instruction n'est traduite par aucune instruction machine) ou I ---o--- N (La pseudo-instruction est traduite par N instructionsmachine) .

.../...

L'ensemble des instructions symboliques constitue le programme source (symbolique), l'ensemble des ins - tructions machine obtenues aprés traduction constitue le programme objet (binaire)

II - B : PRINCIPE D'UN ASSEMBLEUR .

Il est schématisé par la figure suivante :



Principe d'un assembleur

Remarque : Les programmes source et objet sont supposés définis sous forme de paquets de cartes à raison d'une instruction symbolique par carte. Lorsque les programmes source et objet ne sont pas définis sur cartes,on se raméne toujours à ce cas par le notion d'images de carte sur le dispositif de mémoire contenant le programme source et objet.

.../..

II - C : CODES ET OPERANDES SYMBOLIQUES .

Une première commodité du langage d'assembleur est de permettre de repérer un mot mémoire (qui peut contenir une instruction ou une donnée) par un nom symbolique, symbole ou identificateur ; on dit aussi adresse symbolique ou étiquette ou référence. Si une instruction est étiquetée, on peut, dans le programme, utiliser le symbole correspondant pour désigner l'adresse du mot-machine correspondant.
Un symbole est une suite de caractères obéissant à certaines régles de composition. Ces contraintes peuvent porter sur le nombre, la définition et la disposition des caractéres autorisés. Ainsi par exemple, un symbole est trés souvent formé d'une suite de caractéres alphanumériques, le premier étant une lettre.

Des attributs sont affectés à chaque identificateur pour le définir entièrement. Ainsi, dans une machine à caractères où les instructions sont de longeur variable, les attributs seraient : la longeur de la zone référencée, le type de cette zone (instruction, donnée alphabétique, donnée numérique, valeur absolue, ...). Dans la plupart des cas, à chaque symbole est affectée une valeur; cette valeur est l'adresse du premier mot référencé par ce symbole si ce dernier est une étiquette. On peut étendre la notion de symbole et avoir des identificateurs qui ne sont pas des étiquettes, mais peuvent représenter des constantes.

.../...

Une deuxième commodité du langage d'assembleur est l'utilisation de noms mnémoniques pour définir le code opération d'une instruction.
Ainsi, si le code opération d'une addition est IOIOOI en binaire, il est plus simple d'écrire ADD plutôt que IOIOOI, l'assembleur se chargeant de faire la traduction.
Désormais, quand nous parlerons du nom mnémonique de l'instruction, il faudra entendre : nom du code opération.

II - D : INSTRUCTIONS .

Nous avons vu qu'à chaque code opération (langage machine) correspond un nom mnémonique. Nous ne considérons ici que les assembleurs traitant un programme source écrit sur cartes.
Une instruction, en générale, est écrite sur une seule carte. La partie utile de la carte est divisée en 4 zones. Ces 4 zones sont soit situées en des emplacements prédéfinis, soit séparées par un caractère spécial et ont de ce fait un format variable.

- La première zone contient, facultativement, un identificateur qui est appelé étiquette est qui référence alors l'instruction qui suit.

- La deuxième zone contient le nom symbolique de l'instruction machine, ou celui de la pseudo-instruction.

.../...

ces divers noms doivent appartenir aux instructions ou pseudo-instructions reconnues par l'assembleur.
En régle générale, un nom symbolique n'appartenant à aucune de ces deux catégories entrainentune erreur.

- La troisième zone est réservée aux opérandes.
- Enfin la dernière est la partie réservée aux commentaires.

I	9	I4	45
Etiquette	Code opé.	Opérande	Commantaires

Exemple d'un format pour la représentation d'une instruction symbolique sur carte de 80 colonnes.

II - E : PSEUDO-INSTRUCTIONS .

Ce sont des indications particulières pour l'assembleur. Elles generent un nombre variable d'instructions machines. Ces pseudo-instructions appartienent à 3 grandes classes :

II-E-I : Indications de début et de fin d'assemblage.

Ces indications délimitent, pour l'assembleur, les cartes qui forment le programme à assembler.

.../...

En général, si l'indication de début physique de programme n'est pas obligatoire, celle de fin l'est.

Le symbole utilisé pour indiquer la fin est E N D en zone code opération (c'est une pseudo-instruction) .

II-E-2 : Affectation de valeur à un identificateur.

Une méthode pour affecter une valeur à un identificateur consiste à définir ce dernier comme étiquette d'une instruction ou d'une zone de mémoire. Une autre méthode utilise une pseudo-instruction spéciale, appelée E Q U.

L'identificateur à définir se trouve en zone étiquette, et sa valeur en zone opérande. Cette valeur peut être donnée par un nombre, ou une expression arithmétique.

Certains assembleurs n'autorisent en zone opérandes que des identificateurs déjà définis (c'est à dire apparus en zone étiquette auparavent) .

II - E - 3 : Pseudo-Instructions " ALF et CON " :

1°/ Pseudo-Instruction "ALF" :

L'opération " ALF" crée un code de caractéres alphanumériques MIX constant.

Exemple :

```
          9    14
          ↓    ↓
         ALF  ^HUND
```

Pour cette opération l'assembleur génére le mot :

+	00	08	24	15	04

Les codes MIX des caractéres se trouvant en opérande sont :

blanc (^) 00
H..... 08
U 24
N 15
D 04

Les opérations " ALF " sont utilisées principalement pour les titres.

2°/ Pseudo-Instruction " CON " :

L'opération " CON " est équivalente à la définition de constante (DC en Assembleur 1130)

Exemple :

```
BUFI    EQU  2000
        ---
        ---
        CON  BUFI+25
```

L'opération " CON BUFI+25 " assemblée, donne le mot :

+				2025

.../...

C H A P I T R E III
LE LANGAGE D'ASSEMBLAGE MIX

Un langage symbolique est utilisé pour faciliter considérablement la lecture et l'écriture des programmes MIX, pour éviter au programmeur des'occuper de détails qui ménent souvent à des erreurs.

Ce langage (MIXAL) est une éxtention de la notation utilisée dans la partie simulateur, éxtention caractériée notamment par l'utilisation optionnelle de noms alphabétiques pourreprésenter des nombres et un champ adresse pour associer à des noms des emplacements de mémoire.
MIXAL peut être facilement compris à partir d'un premier exemple : (sous programme de recherche du maximum de N éléments X(1)....X(N);ce programme est une partie d'un programme plus gros.

PROGRAMME M (RECHERCHE DU MAXIMUM).

INSTRUCTIONS ASSEMPLEES						N°Lig	ETIQUET	CODE OP	OPERANDES
						01	X	EQU	1000
						02		ORIG	3000
3000 :	+	3009	0	2	32	03	MAXIMUM	STJ	EXIT
3001 :	+	0	1	2	51	04	INIT	ENT3	0,1
3002 :	+	3005	0	0	39	05		JMP	CHANGEM
3003 :	+	1000	3	5	56	06	LOOP	CMPA	X,3
3004 :	+	3007	0	7	39	07		JGE	*+3
3005 :	+	0	3	2	50	08	CHANGEM	ENT2	0,3
3006 :	+	1000	3	5	08	09		LDA	X,3
3007 :	+	1	0	1	51	10		DEC3	1
3008 :	+	3003	0	2	43	11		J3P	LOOP
3009 :	+	3009	0	0	39	12	EXIT	JMP	*

.../...

Ce programme illustre plusieurs choses simultanément :

a) Les zones à en tête "ETIQUETTE ", " CODE OPERATION ", " OPERANDES " sont d'un intérêt primordial. Elles contiennent un programme en langage symbolique MIXAL que nous expliquerons plus loin en détails.

b) La zone intitulée " INSTRUCTIONS ASSEMBLEES " montre le langage machine numérique réel correspondant au programme MIXAL.
MIXAL a été conçu de façon qu'il soit relativement simple de traduire un programme MIXAL en son équivalent numérique en machine;
Cette opération est effectuée par un programme appelé " ASSEMBLEUR ". Dans ce cas, le programmeur peut écrire son programme en langage machine totalement en MIXAL sans jamais s'occuper de la traduction de ce programme en son équivalent numérique.

c) La zone intitulée " n° LIGNE " n'est pas essentielle, mais ne sert que pour des références ultérieures à des parties du programme.

d) Voyons maintenant en détail la partie MIXAL du programme :

- La ligne 1, "X EQU 1000 " indique que le symbole X est équivalent au nombre 1000 .
L'effet de ceci peut être vu à la ligne 6, où l'équivalent numérique de l'instruction : " CMPA X,3 " est :

+	1000	3	5	56

c'est à dire : " CMPA 1000,3 ".

- La ligne 2 indique que les emplacements réservés aux linges suivantes doivent être choisis séquentiellement à partir de l'adresse 3000 .
Le symbole "MAXIMUM" qui apparait dans la zone "ETIQUETTE" de la ligne 3 devient danc équivalent au nombre 3000.

.../...

- Dans les lignes 3 à 12 , le champ " CODE OPERATION " contient les noms symboliques des instructions MIX STJ , ENT3 etc ...
Le CODE OPERATION des lignes 1 et 2, contient EQU et ORIG qui ne sont pas des opérateurs MIX .
On les appelle des " pseudo-opérateurs " ou " pseudo-instructions " parce qu'ils n'apparaissent que dans le programme symbolique MIXAL .
Les pseudo-instructions sont utilisées pour spécifier la forme d'un programme symbolique ; ce ne sont pas des instructions du programme lui même .
Ainsi la ligne " X EQU 1000 " ne parle que du programme, et ne signifie aucunément qu'à une variable on doit donner la valeur 1000 lorsqu' on éxécute le programme .
A noter qu'aucune instruction n'est assemblée pour les lignes 1 et 2 .

La ligne 3 est un " store J " , instruction qui stocke le contenu du registre J dans le champ (0: 2) du mot d'adresse " EXIT " , c'est à dire dans la partie adresse de l'instruction qui se trouve à la ligne 12 .

Ainsi qu'il a été mentionné plus haut, le programme M est une partie d'un plus gros pragramme, autrement dit la sequence :

```
ENT1    100
JMP     MAXIMUM
STA     MAX
```

permettrait par exemple, de se brancher au programme M, N étant fixé à 100.

Le prpgramme M trouverait alors le plus grand des éléments X (1)..... X (N) et reviendrait à l'instruction :

```
STA     MAX,
```

la valeur maximum se trouvant dans rA et sa position, j, dans rI2 .

.../...

- La ligne 5 est un branchement à la ligne 8 .
- Les lignes 4,5,6 sont évidentes .
- La ligne 7 introduit une nouvelle notation:"*".

  Un astérisque référence l'adresse de la ligne en cours; " * + 3 " référence donc l'adresse de la troisième ligne après la ligne en cours. Puisque la ligne 7 correspond à l'adresse 3004, " * + 3 " référence l'adresse 3007 .

- Le reste du programme s'explique de lui-même. A Noter l'apparition d'un autre " * " à la ligne 12 (cette instruction permet le retour au programme principal).

.../...

EXEMPLE D'UN PROGRAMME ECRIT EN MIXAL

Cet exemple montre quelques autres caractéristiques du langage d'assemblage MIXAL.

Son but est d'imprimer les 500 premiers nombres premiers, avec 10 colonnes de 500 nombres chaqune. La table doit se présenter comme suit :

```
FIRST FIVE HUNDRED PRIMES
 0002 0233 0547 0877 1229 1597 1993 2371 2749 3187
 0003 0239 0557 0881 1231 1601 1997 2377 2753 3191
 0005 0241 0563 0883 1237 1607 1999 2381 2767 3203
  .                                               .
  .                                               .
  .                                               .
 0229 0541 0863 1223 1583 1987 2357 2741 3181 3571
```

PROGRAMME P :

Ce programme a été délibérement écrit de façon maladroite de façon à illustrer la plupart des caractéristiques de MIXAL dans un seul programme . rI1= j-500 ; rI2=N ; rI3=K ; rI4 indique B ; rI5 contient M plus les multiples de 50.

```
01   * EXEMPLE PROGRAMME ... TABLE DES NOMBRES PREMIERS
02   *
03   L        EQU  500
04   PRINTER  EQU  18
05   PRIME    EQU  -1
06   BUF0     EQU  2000
07   BUF1     EQU  BUF0+25
08            ORIG 3000
09   START    IOC  0(PRINTER)
10            LD1  =1-L=
11            LD2  =3=
12   2H       INC1 1
13            ST2  PRIME+L,1
14            JIZ  2F
```

.../...

```
15   4H      INC2 2
16           ENT3 2
17   6H      ENTA 0
18           ENTX 0,2
19           DIV  PRIME,3
20           JXZ  4B
21           CMPA PRIME,3
22           INC3 1
23           JG   6B
24           JMP  2B
25   2H      OUT  TITLE(PRINTER)
26           ENT4 BUF1+10
27           ENT5 -50
28   2H      INC5 L+1
29   4H      LDA  PRIME,5
30           CHAR
31           STX  0,4(1:4)
32           DEC4 1
33           DEC5 50
34           J5P  4B
35           OUT  0,4(PRINTER)
36           LD4  24,4
37           JRN  2B
38           HLT
39   * CONTENU INITIAL DES TABLES ET DES BUFFERS
40           ORIG PRIME+1
41           CON  2
42           ORIG BUF0-5
43   TITLE   ALF  FIRST
44           ALF  FIVE
45           ALF  HUND
46           ALF  RED P
47           ALF  RIMES
```

.../...

```
48          ORIG  BUF0+24
49          CON   BUF1+10
50          ORIG  BUF1+24
51          CON   BUF0+10
52          END   START
```

Dans une premier temps, nous expliquerons ce programme tel qu'il est considéré par DONALD E. KNUTH. Puis dans un deuxième temps nous donnerons les restrictions faites sur MIXAL. La différence sera l'extension qu'il sera souhaitable d'apporter à ce travail .

Pour ce programme, il est à noter princi - palement :

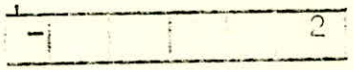
1) Les lignes 1,2 et 39 commencent par un astérisque : ceci est un commentaire purement explicatif, n'ayant aucun effet sur le programme assemblé.

2°) Comme dans le programme M, le " EQU " de la ligne 3 fixe l'équivalent d'un symbole; dans ce cas l'équivalent de L est 500 (dans le programme, L dans les lignes 10-24, représente le nombre de nombres premiers à calculer).
A noter que dans la ligne 5 le symbole PRIME a un équivalent négatif; l'équivalent d'un symbole peut être n'importe quelle configuration de 5 bytes plus un signe.
Dans la ligne 7 l'équivalent de BUF1 est calculé comme BUF0+25 c'est à dire 2025.
MIXAL permet un nombre limité de calculs arithmétiques.
Pour avoir un autre exemple, voir la ligne 13 où la valeur de PRIME+1 (dans ce cas 499) est calculée par l'assembleur.

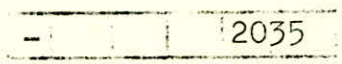
3°) Noter que le symbole PRINTER a été utilisé dans la zone F des lignes 25 et 35. La zone F, qui est toujours entre parenthéses, peut être un nombre ou un symbole, ou 2 nombres et / ou des symboles séparés par une virgule, comme le (1:4) de la ligne 31.

.../...

4°) MIXAL utilise plusieurs façons de spécifier des mots qui ne font pas partie du programme.
La ligne 41 montre une constante ordinnaire " 2 " utilisant le code opération " CON "; le résultat de la ligne 41 est d'assembler le mot :



La ligne 49 montre une constante un peu plus compliqué, " BUF1+10 " qui s'assemble comme :



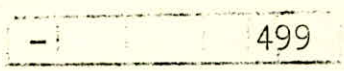
Une constante peut être mise entre les signes "=" et devient alors une " constante littérale " (voir lignes 10 et 11) . L'assembleur génère automatiquement des noms internes et insère des lignes " CON " pour les constantes littérales. Par exemple les lignes 10 et 11 du programme P seront changées en :

```
10          LD1   con1
11          LD2   con2
```

et alors, à la fin du programme, entre les lignes 51 et 52, les lignes

```
51a  con1   CON   1-L
51b  con2   CON   3
```

sont effectivement insérées comme partie de la procédure d'assemblage des constantes littérales.
La ligne 51a s'assemblera comme :



L'utilisation de constantes littérales est utile car elle signifie que le programmeur n'a pas à inventer de nom pour la constante et n'a pas à insérer cette constante en fin de programme; il peut garder à l'esprit les problèmes

.../...

importants et ne pas s'occuper de tels problèmes de routines. Bien sûr, dans le programme P, nous n'avons pas fait un très bon usage des constantes littérales puisque les lignes 10 et 11 pourraient s'écrire : " ENT1 1-L "; " ENT2 3 "!

5°) Un bon langage d'assemblage doit imiter la façon de penser d'un programmeur lœsqu'il écrit un programme , pour pouvoir s'exprimer courramment.
Un exemple de cette philosophie est l'utilisation de constantes littérales; un autre est l'utilisation du " * ", expliqué dans le programme M . Un troisième exemple est l'idée des " symboles locaux " tels les symboles 2H qui apparaissent dans le champ étiquette des lignes 12,25,28.

Les symboles locaux sont les symboles spéciaux dont les équivalents peuvent être redéfinis autant de fois que l'on veut. Un symbole tel que PRIME n'à qu'une seule signification au long du programme, et s'il apparaissait dans le champ étiquette de plus d'une ligne, une erreur serait générée par l'assembleur. Les symboles locaux sont de nature différente ; nous écrivons par exemple 2H dans le champ étiquette et 2F ou 2B dans le champ opérande d'une ligne MIXAL :

2F signifie la plus proche étiquette 2H précédente .
2B signifie la plus proche étiquette 2H suivante .

Par exemple, le 2F de la ligne 14 se refère à la ligne 25
le 2B de la ligne 24 se refère à la ligne 12
le 2B de la ligne 37 se refère à la ligne 28 .

Une adresse de 2F ou 2B ne se refère jamais à la même ligne, par exemple les trois lignes

```
2H      EQU   10
2H      MOVE  2F(2B)
2H      EQU   2B-3
```

sont virtuellement équivalentes à la seule ligne

```
        MOVE  *-3(10)
```

.../...

Les symboles 2F et 2B ne doivent jamais être utilisés dans le champ étiquette, de même que le symbole 2H n'est jamais utilisé dans le champ opérande.

Il y a 10 symboles locaux, qui peuvent être obtenus en remplacant le " 2 " de l'exemple précedent par n'importe quel nombre entre 0 et 9.

L'idée de symboles locaux fut introduite par M.E. CONWAY en 1958, en relation avec un assembleur pour l'UNIVAC 1. Les symboles locaux épargnent au programmeur la nécessité de donner un nom symbolique à une adresse lorsqu'il ne veut que référencer une instruction à quelques lignes de distance. Lorsqu'on référence une adresse proche, il n'y a souvent aucun nom approprié significatif; les programmeurs ont donc tendance à utiliser des symboles tels que X1,X2,X3 etc. ; ce qui conduit au danger d'utiliser 2 Fois le même symbole. C'est pourquoi le lecteur trouvera bientôt que l'utilisation de symboles locaux lui vient naturellement à l'esprit lorsqu'il écrit des programmes MIXAL, s'il n'est pas encore familiarisé avec cette idée.

6°) Aux lignes 30 et 38, le champ opérande est blanc, ce qui signifie que l'adresse est 0. De la même façon, nous aurions pu laisser le champ opérande blanc à la ligne 17, mais le programme aurait alors été moins lisible.

7°) Aux lignes 43-47, on utilise l'opération "ALF", qui crée une constante de 5 bytes dans le code alphanumérique de MIX. Par exemple la ligne 45 assemble le mot :

+	00	08	24	15	04

Ces lignes sont utilisées comme 25 premiers caractères de la ligne 'titre'. Toutes les positions de mémoire dont le contenu n'est pas spécifié dans le programme MIXAL sont généralement mises à 0 (sauf les mémoires 3700-3999 utilisées par le chargeur), il n'y a donc pas besoin de mettre les autres mots du 'titre' à"blanc".

.../...

8°) Il est à noter que des opérations arithméthiques peuvent être effectuées sur des instructions " ORIG "; par exemple voir lignes 40,42 et 48.

9°) La dernière ligne d'un programme MIXAL a toujours comme code opération " END " . La champ opérande de cette instruction contient l'adresse à laquelle commence l'éxécution du programme une fois chargé.

10°) Enfin on pourra observer comment a été effectué le codage de façon que les registres d'index sont incrémentés vers 0, et testés par rapport à 0, quand c'est possible. Par exemple, la quantité J-500, et non pas J , est gardée dans rI1 .
Les lignes 26-34 sont particulièrement caractèristiques, bien qu'un peu sophistiquéss peut être.

.../...

Nous avons vu ce qui pouvait être fait avec MIXAL. Nous concluons ce chapitre en décrivant les règles d'une façon plus précise, et nous verrons en particulier ce qui n'est pas permis en MIXAL.

Les règles relativement peu nombreuses qui suivent définisssent le langage.

REGLE 1 - Un symbole est une chaine de 1 à 7 lettres
Exemple : PRIME, TEMP, MAXIMUM...

REGLE 2 - Un nombre est une chaine de chiffres.

REGLE 3 - Chaque apparition d'un symbole dans un programme MIXAL est dite :"symbole défini" ou " référence future ". Un symbole défini est un symbole qui est apparu dans le champ étiquette d'une instruction précédente de ce programme MIXAL. Une référence future est un symbole qui n'a pas encore été défini de cette façon.

REGLE 4 - Une " expression atomique " est soit :
- Un nombre, ou
- Un symbole défini (concernant l'équivalent numérique de ce symbole, voir règle 13),ou
- Un astérisque (concernant la valeur de * voir règles 10 et 11)

REGLE 5 - Une expression est soit :
- Une expression atomique ou
- Un signe + ou - suivi d'une expression atomique ou
- Un expression suivie d'une opération binaire suivie d'une expression atomique. Les 4 opérations binaires permises sont : +, -, *, et :.

.../...

Ces opérations sont définies sur des mots MIX comme suit :

C = A+B LDA A; ADD B; STA C
C = A-B LDA A; SUB B; STA C
C = A*B LDA A; MUL B; STX C
C = A:B LDA A; MUL =8=; SLAX 5; ADD B; STA C.

Les opérations dans une expression sont évaluées de gauche à droite. Exemples :

- 1 + 5 = 4
1 : 3 égale 11 (utilisé d'habitude dans les spécifications de champ partiel)
* - 3 égale * moins 3

REGLE 6 - Un"champ A" (utilisé pour décrire le champ opérande d'une instruction MIX) est soit :
- Vide (représente la valeur 0) ou
- Une expression, ou
- Une référence future (représente l'éventuel équivalent du symbole, voir règle 13).

REGLE 7 - Un"champ index" (utilisé pour décrire le champ index d'une instruction MIX) est soit:
- Vide (représente la valeur 0),ou
- Une virgule suivie d'une expression (représente la valeur de cette expression).

REGLE 8 - Un "champ F " (utilisé pour décrire le champ F d'une instruction MIX) est soit :
- Vide (représente le champ F standard (0:5)
- Une parenthése ouvrante (() suivie d'une expression suivie d'une parenthése fermente ( )) .

.../...

REGLE 9 - Une " W - value " (utilisée pour décrire une constante MIX sur un mot entier) est soit :

a) Une expression suivie d'un champ F (dans ce cas un champ F vide représente (0:5)) soit

b) Une " W - value " suivie d'une virgule , suivie d'une W- value du type a) .

Une W-value représente la valeur d'un mot MIX numérique déterminée comme suit :
soit la W-value du type :

$$E_1\ (F_1), E_2(F_2)\ \ldots\ldots, E_n(F_n)$$

où $n \geq 1$, les " E " sont des expressions et les " F " sont des champs . Le résultat voulu est la valeur finale qui apparaitrait dans la mémoire CON si le programme hypothéqique suivant était exécuté :
STZ CON ; LDA C_1 ; STA CON (F_1) ; ....;
LDA C_n ; STA CON(F_n) .

Ici C_1 ; ...; C_n représentent des mémoires contenant les valeurs des expressions E_1 , ... E_n . Chaque F_i doit avoir la forme :

$$8L_i+R_i \quad \text{où } 0 \leq L_i \leq R_i \leq 5$$

Exemples :

1 .....

+				1
-	1000			1
+				1

REMARQUE:

Cette régle n'a pas été définie dans notre travail.

.../...

REGLE 10 - Le processus d'assemblage utilise une valeur * (appelée compteur d'adresse) qui est initialement mis à 0 .
La valeur de * doit être toujours un nombre non négatif qui puisse être codé sur 2 bytes. Lorsque le champ étiquette d'une ligne n'est pas vide il doit contenir un symbole qui n'a pas été précédemment défini. L'équivalent de ce symbole est alors défini comme étant la valeur "*" .

REGLE 11 - Aprés avoir traité le champ étiquette comme décrit à la règle 10, le processus d'assemblage dépend de la valeur de la zone étiquette. Il y a six possibilités pour ce champ :

a) La zone " code opération " est un opérateur symbolique MIX. Une table donne les valeurs standards de C et F pour chaque opérateur. Dans ce cas l'opérande doit être un champ A (règle 6) suivi d'un champ index (règle 7), suivi d'un champ F (règle 8).
Nous obtenons ainsi quatre valeurs C,F,A et I ; le résultat est l'assemblage du mot déterminé par la séquence :

```
LDA  C
STA  WORD
LDA  F
STA  WORD(4:4)
LDA  I
STA  WORD(3:3)
LDA  A
STA  WORD(0:2)
```

dans la mémoire spécifiée par "*" et d'incrémenter * de 1 .

.../...

b) La zone " code opération " est un " EQU " : la zone " opérande " doit être une W-value (règle 9); si la zone étiquette n'est pas vide l'équivalent du symbole qui y apparait est égalée à la valeur spécifiée dans la zone opérande. Cette règle est minoritaire par rapport à la règle 10. La valeur de * est inchangée.

c) La zone " code opération " est un " ORIG " : alors la zone étiquette doit être une W-value (règle 9) ; le compteur d'emplacement * est fixé à cette valeur (noter que, conformément à la règle 10, un symbole apparaissant dans la zone étiquette d'une carte " ORIG " a sa valeur fixée à celle de * avant que celui ci ne soit changé. Exemple :
TABLE ORIG *+100 fixe l'équivalent de TABLE à la premièredes 100 mémoires).

d) La zone " code opération " contient " CON ": la zone étiquette doit être une W-value; son effet est d'assembler un mot, ayant cette valeur, à l'adresse spécifiée par * et d'incrémenter * de 1 .

e) La zone " code opération " contient"ALF" : le résultat est d'assembler le mot de codes caractéres formé par les colonnes 14 - 18 de la carte; autrement, tout se passe comme pour CON.

f) La zone "code opération" est un "END": l'opérande doit être une W-value qui spécifie dans son champ (4:5) l'adresse à laquelle commence le programme.
La carte END signale la fin d'un programme MIXAL. Des lignes sont insérées en ordre arbitraire, et correspondent à tous les

.../...

symboles non définis et aux constantes littérales (voir régles 12 et 13).

REGLE 12 - Constantes littérales : Une W-value de 9 caractéres au moins peut être entourée de " = " et utilisée comme référence future. Le résultat est le même que si un nouveau symbole était crée et inséré juste avant END (voir remarque 4 du programme P).

REGLE 13 - Tout symbole a une et une seule valeur équivalente : c'est un mot MIX entier déterminé par l'apparition de ce symbole dans la zone étiquette conformément à la régle 10 ou à la régle 11(b) , ou alors une ligne, contenant dans la zone étiquette le nom du symbole avec CON pour code opération et "0" pour opérande , est éffectivement insérée avant la carte END.

NOTA: La conséquence la plus importante des régles précédentes est la restriction des références futures . Un symbole qui n'a pas été défini dans la zone étiquette d'une carte précédente ne peut pas être utilisée sauf dans le champ A d'une instruction. En particulier il ne peut pas être utilisé :

a) dans une expression arithmétique ou
b) dans la zone opérande de EQU, ORIG, ou CON. Par exemple :

```
LDA   2F+1
CON   3F
```

sont tous deux illégaux .

Cette restriction a été imposée pour permettre un assemblage plus efficace des programmes.

C H A P I T R E IV

ASSEMBLAGE

IV - A : FONCTIONNEMENT GENERAL :

L'assembleur est une fonction programmée pour transformer le programme source (synbolique) en un programme objet (binaire) éxécutable par le simulateur ; il fournit une liste d'assemblage . Il éxiste un grand nombre de procédes différents pour réaliser ceci. Ces procédés sont classés essentiellement en fonction du nombre de passages. Un passage se caractérise par un balayage complet du programme symbolique. La plupart des assembleurs (sans macro instructions) fonctionnent en deux passages :

- Le premier consiste à lire le programme symbolique pour rechercher tous les identificateurs (et leur valeur) et la validité des noms d'instructions(noms mnémoniques d'opérateurs et pseudo-instructions);

- Le deuxième passage effectue la traduction en binaire de chaque instruction.

Entre les premier et deuxième passage , le programme source est généralement conservé sur un support externe tel que le disque.

.../...

Nous allons examiner les diverses fonctions fonda - mentales d'un assembleur.

IV - B : FONCTIONS FONDAMENTALES .

On distingue trois fonctions fondamentales (édition, traduction, production du paquet binaire) qui peuvent être incl uses dans divers passages de l'assembleur.

IV - B - I : EDITION .

C'est la fonction qui consiste, à partir du programme source, à obtenir une liste d'assemblage dans laquelle les diverses parties des instructions sont en des emplacements fixes . On trouvera successivement les infor - mations suivantes :

- Traduction binaire de l'instruction,
- Instruction symbolique contenue dans la carte.

De plus la liste d'assemblage fournit des renseignements sur les erreurs du programme source.

Une autre information donnée par l'éditeur est une table des symboles utilisée dans le programme.

Cette table fournit en général les indications suivantes :

- Nom de l'identificateur,
- Valeur (ou indication de référence éxterne ou indica - tion de non définition).

.../...

IV - B - 2 : TRADUCTION .

C'est la fonction qui consiste à reconnaître les noms des instructions (recherche du code opération) et à affecter une valeur aux étiquettes.

En général, les noms des codes opérations sont en nombre limité et sont décrits dans une table spéciale. L'assembleur recherche donc dans cette table et, s'il ne trouve pas, fait appel à un sous-programme d'erreur. Cette table contient le nom mnémonique des codes opérations.

L'affectation d'une valeur aux étiquettes se fait par la fabrication d'une table dite table des identificateurs ou table des symboles.

IV - B - 3 : FORMATION DU PAQUET BINAIRE.

La dernière et principale fonction d'un assembleur est la formation d'un programme directement exécutable par la machine. On peut imaginer un assembleur implantant directement en mémoire centrale le résultat; cependant, cette solution est en général impensable par suite du problème de place. Ce programme objet sera donc écrit, temporairement, au fur et à mesure de sa formation , sur un support éxterne (disque).

Ce paquet comporte un certain nombre d'informations en plus du programme objet, ce sont natomment des cartes début et fin, des cartes contenant des

.../...

indications sur les symboles éxternes et sur les entrées, ou des cartes fixant l'adresse de début d'implantation en mémoire centrale dans le cas où cette adresse est fixée lors de l'assemblage (adressage absolu).

.../...

C H A P I T R E V

GENERATION DES INSTRUCTIONS MIX

A cause des références " en avant ", ceci impose que l'assembleur MIX utilise deux passages de lecture du programme :

- Le premier est une passage de définition pendant lequel est traité complétement l'évolution du compteur d'Emplacement ou compteur d'Adresse.
- Le second est un passage d'utilisation des valeurs obtenues au passage précédent.

La définition et l'utilisation des symboles correspond à la consultation d'un dictionnaire ou Table des symboles; (Les assembleurs utilisent donc largement les diverses téchniques de manipulation de tables, plus ou moins complexes suivant que le dictionnaire peut tenir en entier en mémoire rapide ou doit être fractionné en plusieurs parties rangées en mémoire auxiliaire non directement adressables).

V.A: Format du langage d'assemblage MIX :

Le langage d'assemblage représente symboliquement les instructions.

L'assembleur reste toujours lié à la machine et pense toujours en termes d'instructions élémentaires. Il utilise donc un modéle plus ou moins rigide pour leur représentation, avec des zones spécialisées assemblées par par union qui donnent ainsi la construction finale de l'instruction complète.

Le programme source MIX se compose de plusieurs lignes, chaque ligne comprenant soit une instruction machine, soit une directive d'assemblage.

..../...

Dans une ligne, on distingue plusieurs zones d'écriture :

- Zone symbole (ou Etiquette)
- Zone opération
- Zone opérande

Exemple : Assembleur MIX .

étiquette	opération	opérande
TOTO	STA	ZONE + 6,3 (PRINC)

1°) Les symboles qui représentent les emplacements de mémoires sont choisis par le programmeur qui peut les composer selon certaines règles :
Alphabet (les 26 lettres de l'Alphabet),
longeur (7 caractères)...

2°) Les symboles d'opération sont, par contre, imposés par le langage MIX et correspondent aux instructions machine :
' LDA ' pour chargement, 'ADD' pour addition,
' CMPA ' pour comparaison ...

3°) Les opérandes référencent:

- des symboles
- des expressions faisant intervenir des symboles.
  Ces symboles doivent apparaître une fois et une seule dans la zone symbole (ou zone étiquette).
- des modificateurs qui servent à indiquer des particuliarités dans l'instruction :
  registres, champs.

.../...

Exemple :

col. 1	Col.9	col.14
TOTO	STA	ZONE+6,3 (PRINT)

ZONE + 6 = expression
3 = registe } = opérande
(PRINT) = champ

V.B: Schéma et description de l'assembleur MIX :

V.B.1 : SChéma:

(Voir page suivante)

V.B.2 : Description du Schéma :

Cet assembleur doit effectuer 4 fonctions :

Fonction 1 :

Lecture des cartes constituant le programme écrit dans le langage symbolique MIX et mise sur disque de ce programme.

Fonction 2 :

A la demande de l'utilisateur, lecture du programme MIX à partir du disque et sortie sur imprimante.

Fonction 3 :

Construction de la Table des symboles qui se fait simultanément à la lacture des cartes. On donne à cette fonction le nom PASS 1.

Fonction 4 :

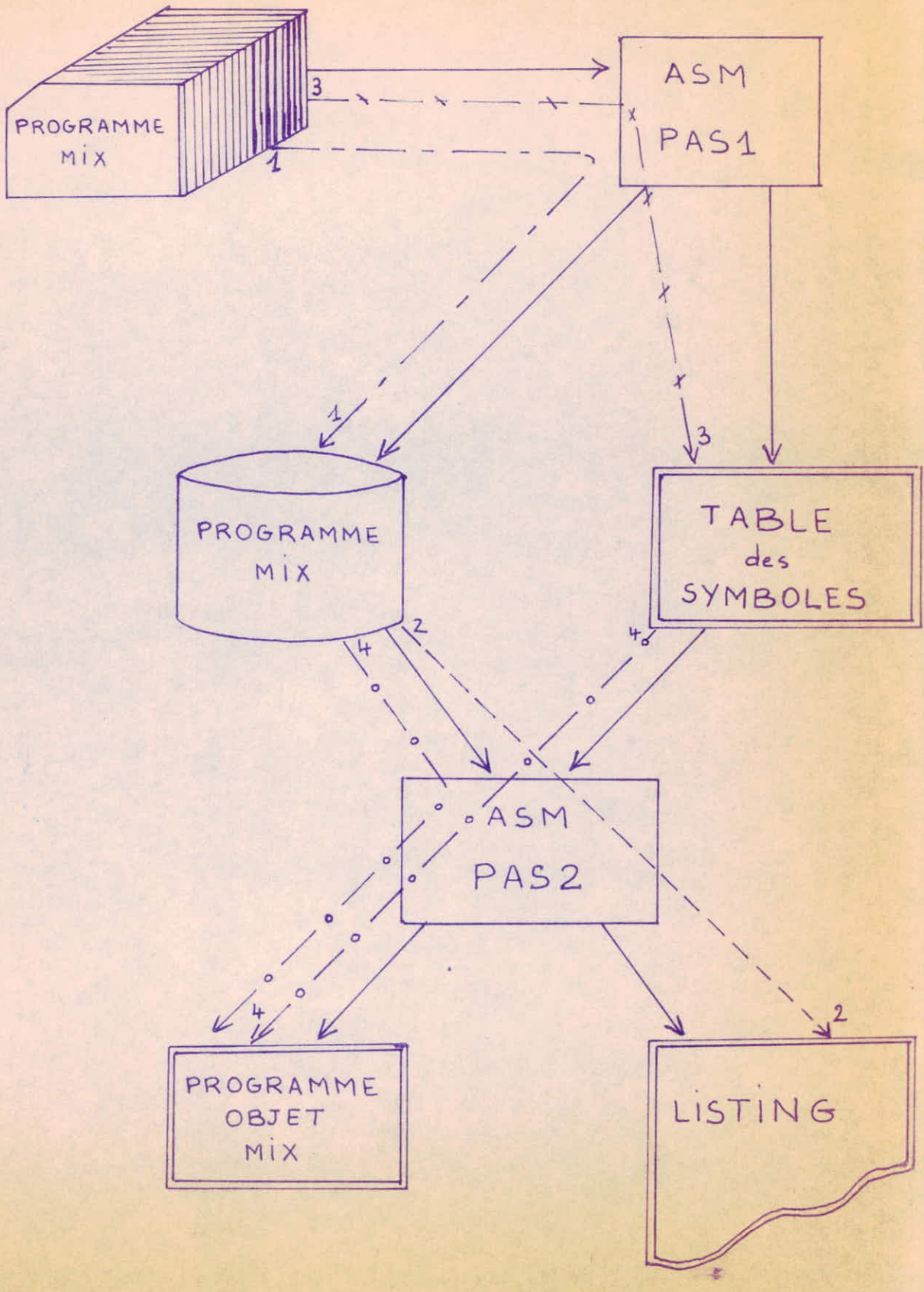
Cette fonction se compose:

-d'un automate d'états finis, qui reconnait un operande et l'évalue.

-d'un génèrateur de code opération.

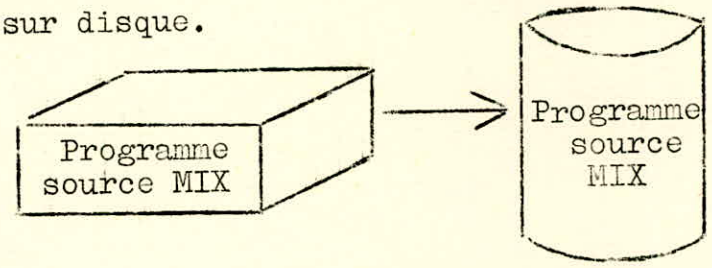
.../...

ASSEMBLEUR MIX



V. C : Sous programme LECT :

LECT a pour but de lire une carte du programme source (programme MIX sur cartes perforées) et de l'envoyer sur disque.



V. D : Table de identificateurs ou table des symboles : PAS 1

V.D.1 : But :

Le premier passage consiste à construire une table dans laquelle on met les étiquettes du programme MIX (zone occupant les colonnes 1 à 7 d'une carte). Ces étiquettes sont précédées de leur longueur et suivies de l'adresse à laquelle elles sont dans le programme écrit en langage symboliques MIX.

A chacune de ces étiquettes est associée une valeur qui correspond à l'emplacement de mémoire qu'elles représentent.

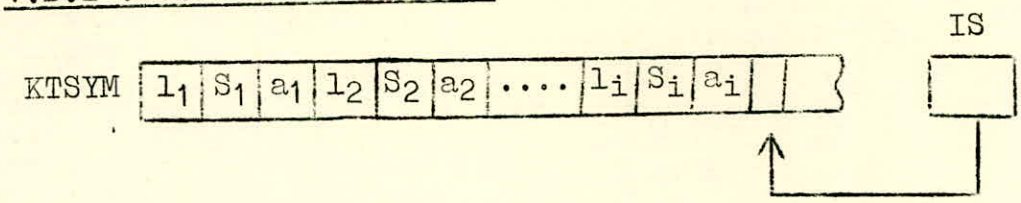
Cette valeur est determinée soit par l'évolution d'un compteur d'emplacement après chaque ligne assemblée, soit par la valeur de l'opérande pour les pseudo-instructions (EQU, ORIG, END ...).

Remarques :

- Un symbole utilisé plus d'une fois en zone étiquette conduit à une erreur.
- Seules les étiquettes alphabétiques de 7 caractères au maximum sont acceptées par l'assembleur.

.../...

V.D.2 : Structure associée :



On a 40 symboles (ou étiquettes) de 5 caractères environ dans un programme moyen.

La dimension de KSYM sera donc fixée à 256 mots 1130.

V.D.3 : Définitions :

- l_i : est constitué d'un mot mémoire.
  Il définit les longueur (nombre de caractères) du symbole ou étiquette S_i .
- S_i : Champ occupant l_i mots mémoire (la table des symboles KTSYM étant définie en format décompacté).
  S_i représente un symbole dans le format A1 (un caractère par mot).
- a_i : champ occupant 1 mot mémoire.
  Il représente l'adresse associée au symbole S_i.
- IS : A la table des symboles KTSYM est associé un indicateur IS.
  IS occupe un mot mémoire. Le contenu de IS indique la première place libre dans KTSYM.

V.E : Traduction du programme source : (2ém. passage).

Cette traduction porte le nom de PAS2.

V.E.1 : But :

Le but est de transformer complétement le programme MIX en un programme en langage machine afin qu'il puisse être executer par le simulateur.

Le programme obtenu après traduction est généré dans la zone COMMON de la mémoire du 1130.

.../..

Remarque : Les pseudo-instructions EQU et ORIG ainsi que les cartes commentaires ne sont pas traitées dans le deuxième passage de l'assembleur mais au cours du premier passage (PAS1), c'est à dire simultanément à la construction de la table des symboles.

V.E.2 : Sous programmes utilisés par PAS2 :

La formation du programme directement exécutable par la machine necessite :

- l'évaluation du nombre associé au nom mnémonique définissant le code opération d'une instruction. Cette évaluation est faite par le sous-programme CODE .
- l'évaluation de l'opérande (adresse , index, champ).
  Cette évaluation est faite par le sous-programme AUTOM .

Le mot MIX (2 mots 1130) est construit à l'aide du sous-programme BIND à partir des données fournies par CODE et AUTOM.

V.F : Sous-programme CODE :

Le sous-programme CODE reconnait le code d'une instruction MIX.

Lorsque des instructions ont même code opération, on les distingue par la valeur du champ F, dans ce cas F est donné par le sous programme CODE.

.../...

Exemple :

Les instructions de décalage ont même code opération, mais des champs F distincts :

C = 6
F = 0 instruction SLA
F = 1 instruction SRA
F = 2 instruction SLAX
F = 3 instruction SRAX
F = 4 instruction SLC
F = 5 instruction SRC

* Simulation :

Les instructions symboliques MIX sont mises dans différentes tables à raison d'un caractère par mot (format A1).

On distingue 12 Tables :

- KCOD = table de 256 mots mémoire où sont rangées par ordre croissant du code opération, les instructions dont le champ F est nul (C = 0 à 63).
- KCOD1: Table de 8 mots mémoire où sont rangées les instructions par ordre croissant de F (F=1à2) et dont le code opération est C = 5 .
- KCOD2 : Table de 20 mots où sont classées les instructions de décalage par ordre croissant de F (F = 1 à 5) avec C = 6 .
- KCOD3 : Table de 36 mots où sont classées les instructions de saut par ordre croissant de F (F = 1 à 9) avec C = 39 .
- KCOD4 : Table de 32 mots pour les instructions de décrementation par ordre croissant de C (C = 48 à 55) pour F = 1.

.../...

- KCOD5 : Table de 32 mots pour les instructions dont le code opération C = 48 à 55 pour F = 2.
- KCOD6 : Table de 32 mots pour les instructions dont le code opération est C = 48 à 55 pour F = 3.
- On distingue 5 tables de 32 mémoires chacune pour les instructions dont le code opération C = 40 à 47 pour des valeurs de F distinctes :

  - KCODB pour F = 1
  - KCODC pour F = 2
  - KCODD pour F = 3
  - KCODE pour F = 4
  - KCODF pour F = 5

Pour une instruction donnée, le sous-programme CODE consulte les tables citées au dessus pour donner le code opération de cette instruction et s'il y a lieu la valeur du champ F. Si cette instruction n'existe pas dans ces tables, CODE signale une erreur.

V.G : Sous programme BIND :

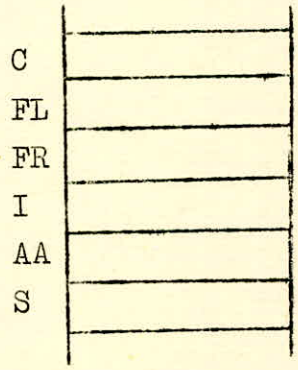
Ce sous programme a pour but de reconstituer une instruction MIX à partir des mots de la carte mémoire C, FL, FR, I, AA, AA et S.

Les contenus des mémoires :

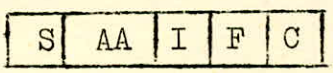
- I, AA, S sont donnés par le sous programme AUTOM
- C par le sous programme CODE
- FR soit par AUTOM, soit par code.

.../...

A partir des mots mémoire :

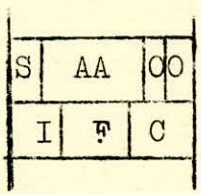


le sous-programme BIND nous donnera la configuration :



(2 mots 1130)

soit en mémoire la configuration suivante :



V - H : Sous-programme AUTOM:

V - H-1 : but :

AUTOM évalue toute la zone opérande :

- Le signe est mis dans le mot S
- L'adresse est mise dans le mot AA
- Le registre d'index est mis dans le mot I
- La spécification de champ dans le mot F

V - H-2 : Syntaxe générale d'un opérande :

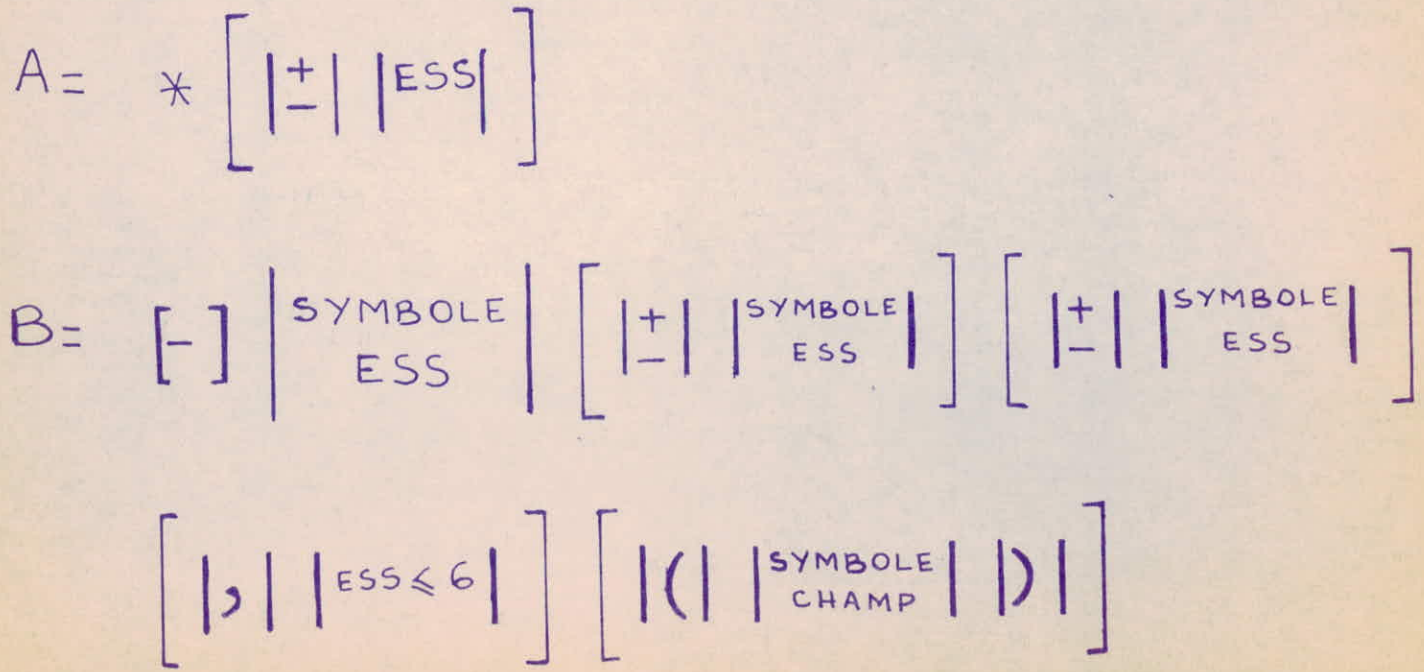
voir figure 1.

V - H-3 : Graphe :

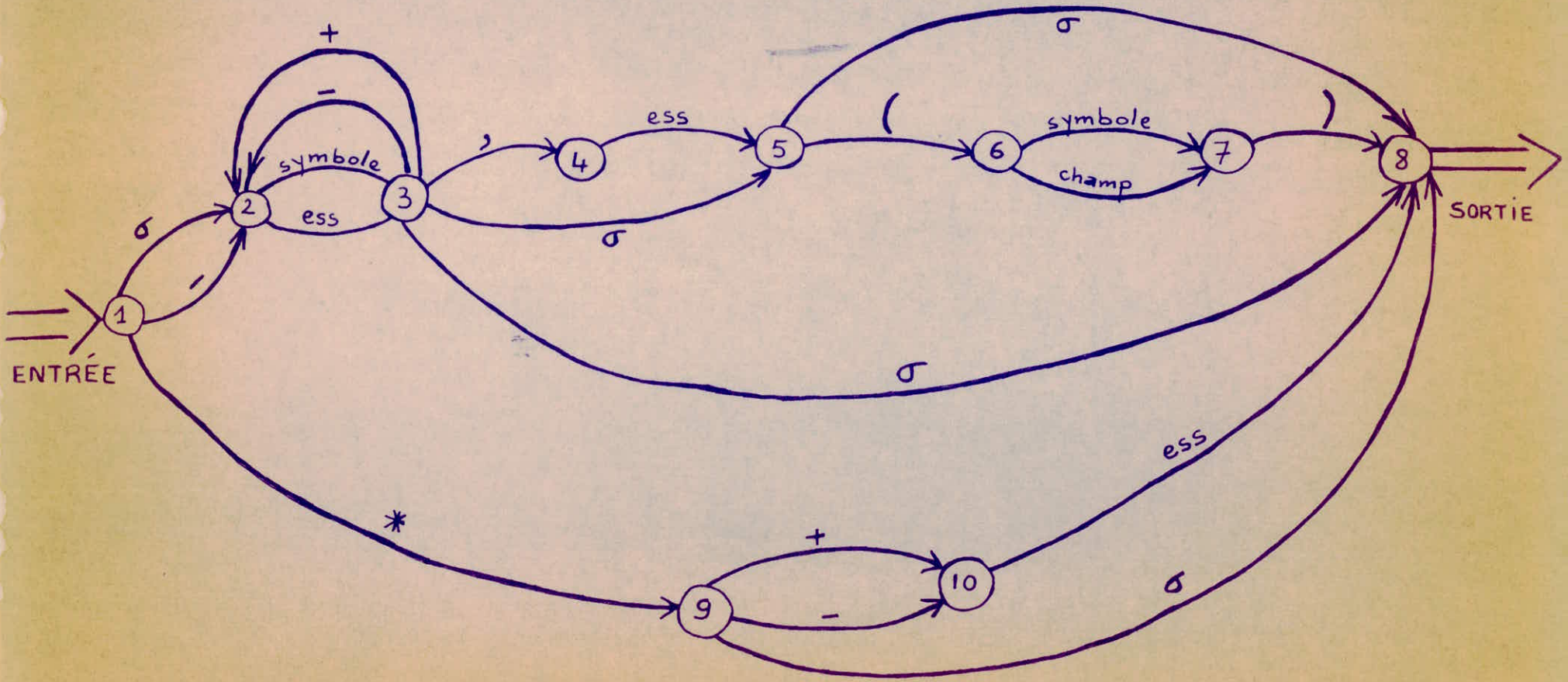
voir figure 2.

.../...

• SYNTAXE GÉNÉRALE D'UN OPÉRANDE •



GRAPHE


+
-
symbole
ess
σ
-
,
4
ess
5
σ
(
6
symbole
champ
7
)
8
SORTIE
σ
1
ENTRÉE
2
3
σ
*
9
+
-
10
ess
σ

C H A P I T R E V I

SIMULATION DE L'ASSEMBLEUR

Toute la simulation sera faite en utilisant la programmation structurée .

Notion de programmation structurée (1) :

C'est une discipline de programmation mise en jeu pour produire des programmes corrects et compréhensibles et qui soient aussi, faciles à maintenir et à modifier.

Le programme est composé en unités intellectuellement maniables (50 lignes de code environ) appelées modules ou segments .

L'essence même de la programmation structurée est plus une attitude qu'une recette : c'est la reconnaissance des limites de nos moyens . Cette prise de conscience peut être utilisée à notre avantage si nous prenons le soin de nous resteindre à l'écriture de programmes que nous puissions manier intellectuellement et dont nous puissions comprendre la totalité des implications.

Pour vaincre la complexité, notre principal outil intellectuel est l'abstraction. Un probléme complexe ne doit pas être immédiatement examiné en termes d'instructions machines, bits ou mots logiques, mais en termes d'entités naturelles au probléme lui-même, abstrait en un sens convenable. Chaque abstraction est ensuite décomposée en niveaux d'abstractions inférieures et ainsi récursivement jusqu'à atteindre la machine abstraite réelle . (1) .

(1) : M. GALINIER : Séminaire de programmation
(la programmation structurée : une introduction) -
Université Paul SABATIER.

PREMIER PASSAGE

- Lecture des cartes
- Envoi des images cartes sur disque
- Construction de la table des symboles

ASSEMBLEUR

1er Passage

2eme Passage

DEUXIEME PASSAGE

Implantation du binaire

dans la mémoire MMIX

LECTURE D'UNE CARTE

ENVOI SUR DISQUE

```
cont = 0
DO WHILE ( Not END )
      - lire une carte
      - Envoi de l'image
        carte sur disque
      - Traitement de la
        carte
END
```

TRAITEMENT DE LA CARTE

```
IF ( Pseudo-instruction )
   THEN  traitement de la Pseudo-
         instruction .
   ELSE  cont = cont + 1
END
IF ( étiquette )
   THEN  traitement de l'éti-
         quette .
END
```

TRAITEMENT DES PSEUDO-INSTRUCTIONS

```
CASE  pseudo-instructions   OF
   ORIG : cont = valeur de
          l'opérande
   EQU  :
   ALF  :   traitements spéci-
   ALF  :   fiques de ces pseudo-
            instructions
   CON  :
ESAC
```

TRAITEMENT DE L'ETIQUETTE

```
Reconnaissance de cette étiquette
IF ( étiquette existe dans table des symboles )
   THEN    erreur
   ELSE    li = longueur de l'étiquette
           Si = étiquette
           IF ( non pseudo-instruction )
              THEN  ai = valeur du compteur
                         ordinal
              ELSE  ai = valeur de l'opérande
           END
END
```

TABLE DES SYMBOLES

Une étiquette rencontrée est mise dans la table des symboles (si elle n'existe pas dèjà dans la table) précedée de sa lon - gueur et suivie de sa valeur d'adresse.

l_1	$Etiquette_1$	a_1	l_2	$Etiquette_2$	a_2

{ l_i $étiquette_i$ o_i }

l_i = 1 mot

$Etiquette_i$ = l_i mots

a_i = 1 mot

.../...

TRAITEMENT DE ALF

```
IF ( KPOPD = symbole de 5 caractéres " S1S2S3S4S5")
       THEN  DO  WHILE  ( I ≥ 5 )
                 Code EBCDIC S(I)=code EBCDIC S(I)-64
                  END
                  CALL  CONVR
                  AA = code MIX (S1)*64+code MIX(S2)
                  I = code MIX(S3)
                  FR = code MIX(S4)
                  C  = code MIX(S5)
                  S  = 0
                  CALL BIND
                  MMX(IJ) = RDM(1)
                  IJ = IJ + 1
                  MMX (IJ) = RDM(2)

        ELSE  erreur
END
```

Le sous-programme CONVR est décrit dans la partie simulateur .

TRAITEMENT DE CON

```
Mettre la valeur de l'opérande dans
un mot MIX . Ceci se fait au 2ème
passage .  Cont = Cont + 1
```

TRAITEMENT DE EQU

```
IF ( pas d'étiquette )
     THEN  erreur
     ELSE traitement d'étiquette

END
```

.../...

DEUXIEME PASSAGE

Implantation du binaire dans la mémoire MMIX ⟶ Programme POBJ

PROGRAMME POBJ

- Calcul de S, AA, I, F, C
- Génération du mot MIX

S,AA,I,F,C

- Automate donnant S,AA,I,F (AUTOM)
- Générateur du code donnant C (CODE)

GENERATION DU MOT MIX

Programme BIND qui regroupe S,AA,I,F,C en un mot MIX .

AUTOM

L'automate évalue l'opérande suivant la syntaxe définie précédemment.

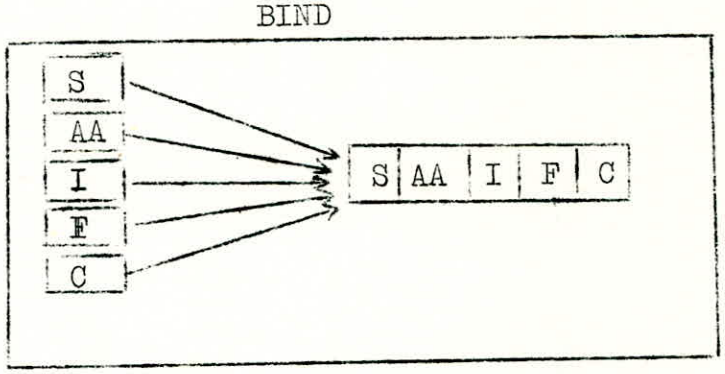
S = signe
AA = adresse
I = index
F = champ

CODE

```
C = valeur ( codop)
    (ceci sur simple
     consultation de tables)
IF ( opérateur étendu  )
    THEN   F = champ

END
```

BIND



MESSAGES D'ERREUR DE COMPILATION

Si une erreur est détectée à la compilation, l'assembleur le signalera à l'utilisateur en lui fournissant le type d'erreur et la ligne où cette erreur a été commise.

Nous distinguons 9 principaux types d'erreurs :

Type 1 : Syntaxe d'opérande refusée
Type 2 : Erreur de syntaxe sur le champ
Type 3 : Erreur de syntaxe sur l'index
Type 4 : Symbole indéfini
Type 5 : Erreur de syntaxe sur le code
Type 6 : Etiquette dépassant 7 caractéres
Type 7 : Seules les étiquettes alphabétiques sont autorisées.
Type 8 : Etiquette déclarée plus d'une fois .
Type 9 : Etiquette devant EQU obligatoire .

DEUXIEME PARTIE

SIMULATEUR

Quand le langage présenté à un programme interpréteur est le langage machine d'une autre, l'interpréteur est souvent appelé SIMULATEUR .

Pourquoi un simulateur ?

En général, ayant une nouvelle machine et des programmes écrits pour une ancienne, on veut que ces programmes soient acceptés par la nouvelle au lieu de réécrire de nouveaux programmes; ou bien ayant une machine, on veut donner aux utilisateurs la possibilité de programmer en un nouveau langage.

Cela a au moins une apllication. En effet un constructeur avant d'implanter un langage sur machine, est obligé de le simuler sur cette machine.

Il ne faut pas toutefois pousser cela jusqu'à arriver par exemple à simuler une machine avec un langage déjà simulé.

Cette simulation coûte cher dans la mesure où elle réduit considérablement la vitesse d'éxecution.

Il est certain, par exemple, qu'un programme MIX s'exécutera plus vite sur une machine MIX que sur un 1130 muni du système MIX

Les instructions à simuler ont été divisées en groupes d'opérateurs homogènes ou sections:

- Opérateurs de chargement.
- Opérateurs de stockage.
- Opérateurs arithmétiques.
- Opérateurs de modification des registres.
- Opérateurs de comparaison.
- Opérateurs de saut.
- Opérateurs d'entrées/sorties.
- Opérateurs de conversion.
- Opérateurs divers.

.../...

Dans une première partie, chacune de ces sections est présentée comme suit :

- Déscription des actions de chacun des opérateurs.
- Simulation de ces opérateurs. Il sera donné plutôt une méthode générale qui pourrait éventuellement être utilisée dans le sens d'une amélioration qu'une description exacte de la simulation qui pourrait conduire à des détails qui ne sont pas toujours utiles.

Dans une deuxième partie, le simulateur sera présenté comme un seul bloc et le cheminement depuis son appel jusqu'à l'exécution de la dernière instruction (rencontre de HLT) sera décrit .

Remarques : Certaines instructions ont été signalées bien qu'elles n'aient pas été simulées (car des opérateurs arithmétiques et des des comparaisons avec champ F =(0:6) indiquant des opérations en virgule flottante).
D'autres ont été simulées avec une différence par rapport à celles de MIX (cas de HLT - JRED) .

.../...

C H A P I T R E I

PRESENTATION ET SIMULATION DES INSTRUCTIONS MIX

S E C T I O N 0

SOUS PROGRAMME DE SERVICE

.Sous-programme ROME .

L'appel de ce sous-programme ce fait de la façon suivante :

```
CALL    ROME
DC      A
DC      X
```

A et X sont des étiquettes dans le programme appelant. I_1 utilise le mot du 1130 d'adresse 521.

* Si reste de $[521] / 8 = 0$ le retour se fait à l'instruction portant l'étiquette A.

* Si reste de $[521] / 8 = 7$ le retour se fait à l'instruction portant l'étiquette X.

* Sinon le retour se fait à l'instruction suivant DC X.

* Dans ce cas on a une valeur dans le registre 2 du 1130. Cette valeur est telle que :

$$\text{Rest} + [XR2] = 7 .$$

Cette valeur permet de récupérer le registre $r_I i$ sur lequelle porte l'instruction VEBi en tenant compte de la disposition des $r_I i$ sur la carte mémoire.

Exemple :

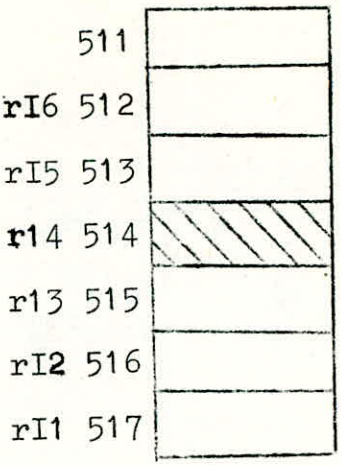
ENC4 C = 52 $[521] = 52$

Reste de $\frac{52}{8} = 4 \rightarrow$ il faut incrémenter $r_I 4$

$[XR2] = 7 - 4 = 3$

.../...

En indexant avec XR2
l'adresse de référence
étant 511 on accéde
à 511 + 3 = 514 → rI4



Sous-programme ETIRE

Forme de l'appel : CALL ETIRE .
Il suppose qu'il y a un index dans XR2 (fourni en général par ROME).

Le contenu du mot d'adresse 511 + XR2 c'est-à-dire l'un des rIi est étalé sur 2 mots du 1130, le résultat est stocké dans le registre accumulateur MIX : rA.

rIi | s | m | n |

Entrée

rA | s | 0 | 0 | 0 | m | n |

Résultat en Sortie

[XR2] = 7 - i

Sous-programme PACTE

Forme de l'appel : CALL PACTE.
Dans [XR2] = 7 - i
Il prend le mot MIX rA, le tasse (dans ce cas les bytes 1,2,3 de rA sont nuls) et le stocke dans le registre index MIX rIi.

rA | s | 0 | 0 | 0 | m | n |

Entrée

rIi | s | m | n |

Sortie

.../...

Sous-programme SEULS

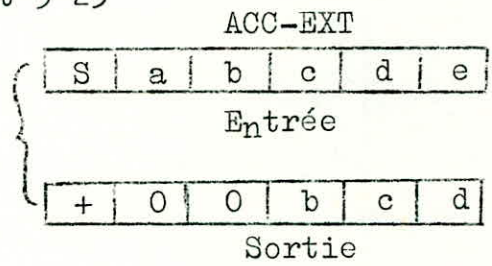
Etant donné une instruction portant spécification de champs (L : R), Le mot MIX étant chargé dans l'ACC-EXT du 1130 le sous-programme SEULS_ isole les bytes de rang L,L+1, ..., r

Les cadre à droite dans l'ACC-EXT. Si le nombres de bytes est inférieur à 5 il oomplète à l'avant par des zéros. Pour L = 0 le signe à la sortie est le même que celui de l'entrée.

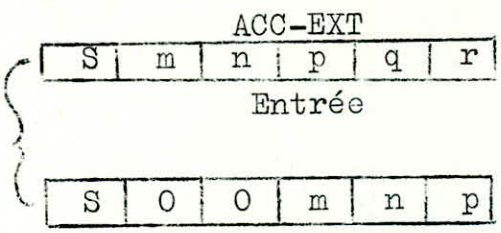
Pour L > 0 le signe à la sortie est plus (+). Les paramètres L et R sont respectivement dans les mots 1130 522 et 5 23

Exemple :

L = 2
R = 4



L = 0
R = 3



Sous-programme LOCAL

Forme de l'appel : CALL LOCAL
ETIQ DC *-*

ETI est facultative.

Ce sous-programme LOCAL part avec un mot MIX dans l'ACC-EXT du 1130 et les spécifications de champ L et R dans les mots 1130 d'adresse 522 et 523 respectivement.

.../...

Il appelle SEULS qui lui fournit à partir de ces données un résultat dans l'ACC-EXT.

Ce résultat est transformé en 2 :

- Le signe MIX de l'ACC (c'est-à-dire 0 pour + et 1 pour -) est stocké dans le mot d'adresse ETIQ .
- Le contenu de l'ACC-EXT qui perd son signe MIX et devient ou reste positif.

Exemple :

Résultat de SEULS

ACC-EXT

-	0	0	n	y	z

A la sortie de LOCAL

ACC-EXT

+	0	0	n	y	z

L'appel étant :

```
        CALL   LOCAL
SIGNE   DC
```

Dans le mot d'adresse signe on aura 1 aprés l'appel.

.../...

S E C T I O N I

OPERATEURS DE CHARGEMENT

MODULE C0823

I. - Les opérateurs :

a) LDA . C= 8 ; F = champs

Les champs spécifiés du mot MIX d'adresse M sont chargés dans le registre A.

Si dans la spécification de champs le signe est inclus (L = 0) le registre A prend aprés chargement le signe de M; sinon le signe + est sous-entendu. Les bytes chargés sont cadrés à droite dans rA, les bytes non chargés sont mis à zéro.

b) LDX . C = 15 ; F = champs

Identique au LDA sauf que rX est chargé à la place de rA.

c) LDi. C = 8 + i ($1 \leq i \leq 6$) F = champs

On charge rIi au lieu de rA (pour i=2, LD2, le registre index 2 est chargé selon les règles définies au LDA).

Les bytes 1,2,3 sont supposées contenir des 0 .

d) LDAN . C = 16 . F = champs

e) LDiN . C = 16 + i . F = champs

f) LDXN . C = 23 . F = champs

Un LD$N est équivalent à un LD$ suivi d'un changement de signe du registre $ ($ pouvant être A,X,1,2,3,4,5,6). Cela revient donc à charger l'opposé du registre spécifié.

.../...

II. - Forme de l'instruction :

```
ETIQ1    LD$     M,I(L:R)
ETIQ2    LD$N    M2,I2(L2:R2)
```

III. - Exemple :

```
         LDA     M,I(L:R)
         LDXN    M,I(L:R)
```

On obtient donc RA = - RX .

Soit le mot 2000

-	1	16	3	63	4

rA aprés chargement :

LDA	2000	-	1	16	3	63	4
LDA	2000(1:5)	+	1	16	3	63	4
LDA	2000(0:3)	-	0	0	1	16	3
LDA	2000(4:4)	+	0	0	0	0	63
LDA	2000(0:0)	-	0	0	0	0	0
LDA	2000(1:1)	+	0	0	0	0	1

IV. - Sous-programmes appelés :

- SEULS
- ROME
- PACTE

V. Simulation :

.../...

V. - Simulation :

- Sauvegarder rA
- IF (LD)
  THEN S2 = 0
  ELSE S2 = 16384
  END
- Récuperer le mot MIX
- Isoler les bytes à charger (SEULS)
- (ACC) + S2 et remettre le bit 0 à 0
- Stocker ACC-EXT dans rA
- IF (LDi)
  THEN traitement de rIi
  ELSE IF (LDX)
  THEN traitement rX
  ELSE aller à fin

Traitement rIi

- Prendre rA
- Tasser (supprimer les bytes 1,2,3 par PACTE)
- Stocker dans rIi
- Restaurer rA
- Aller fin

Traitement rX

- Mettre rA dans rX
- Restaurer rA
- Aller fin

.../...

S E C T I O N II

OPERATEURS DE STOCKAGE

I. - Fonction :

Ces opérateurs ont une signification opposée à celle des chargements.

1. STA (store rA). C=24. F = Champ.

Le contenu des champs spécifiés par F de la mémoire adressée est remplacé par le contenue de rA. Les bytes de rA qui vont dans cette mémoire sont toujours pris parmi les derniers. Les autres champs de la mémoire n'étant pas modifiés.

Le signe de la mémoire n'est pas affecté à moins qu'il fasse partie du champ et dans ce cas c'est alors le signe de rA.

Le contenu de rA n'est pas modifié.

F standard = (0 : 5)

Syntaxe : STA M (F) ou encore STA AA,I (F)

soit M l'adresse obtenue aprés indexation.

Contenu initial de M =	-	1	2	3	4	5
Contenu initial de rA=	+	6	7	8	9	0

Instruction	Contenu de M aprés exécution de cette instruction					
STA M (donc champ=(0:5))	+	6	7	8	9	0
STA M (1:5)	-	6	7	8	9	0
STA M (5:5)	-	1	2	3	4	0
STA M (2:2)	-	1	0	3	4	5
STA M (2:3)	-	1	9	0	4	5
STA M (0:1)	+	0	2	3	4	5

.../...

2. STX (store rX). C=31 - F = champ.

Même action que STA sauf que ça concerne rX au lieu de rA.

3. STi (store rI) C = 24 + i. F = champ.

Même action que STA excepté que c'est rIi qui est stocké en supoosant que les bytes 1,2 et 3 (fictifs) sont à zéro.

ainsi si rI1 contient | S | m | n |

c'est comme s'il contenait | S | 0 | 0 | 0 | m | n |

4. STJ (store rJ) C = 32 . F = champ

Même action que STi à part que c'est rJ qui est toujours considéré comme positif. Ceci est normal car rJ contient une adresse. La spécification standard de F est (0:2) et non (0:5) comme c'est pour les instructions précédentes.

5. STZ (store zéro) C = 33. F = champ.

Les champs spécifiés de la mémoire sont remplis de zéro.

Ces opérateurs sont simulés dans le module SPST.

II Simulation :

Le code opération des instructions traitées dans ce module est défini par :

$$24 \leq C \leq 33$$

1 Module SPST

```
    Case C OF
24: traitement du STA
26
 .  Traitement des ST1
 .
30:
31: traitement du STX
32: traitement du STJ
33: traitement du STZ
    ESAC
```

1.1 Traitement du STA

Sous-programme STORA

1.2 Traitement des STI

- CONVI
- Traitement du STA

1.3 Traitement du STX

- STORX
- Traitement du STA

1.4 Traitement de STJ

- STORJ
- Traitement du STA

1.5 Traitement STZ

rA (1) = 0
rA (2) = 0
Traitement STA

1.2.1 CONVI

Etendre rI sur 2 mots 1130 sous la forme :

± 0 0 0 m n

Ce double mot est mis dans rA

1.3.1 STOR X

rA (1) = rX (1)
rA (2) = rX (2)

.../...

1.4.1 STORJ

Etendre rJ sur 2 mots sous la forme :

+ 0 0 0 m n

signe toujours +

Ce double mot est mis dans rA.

Sous-programme STORA :

Voyons un exemple : STA M (1:2)

La selection se fera de la façon suivante :

rA	a	b	c	d	e	f

M	s	t	u	v	w	x

Pour cet exemple, le stockage consiste à mettre les champs c et f à la place des champs t et u; les champs s,v,w,x étant inchangés.

* Cas du signe :

Si le champ F est de la forme (0:FRà, cela signifie que l'on tiendra compte du signe du registre rA; le signe de rA, de même que les FR bytes les plus à droite seront stockés en M.

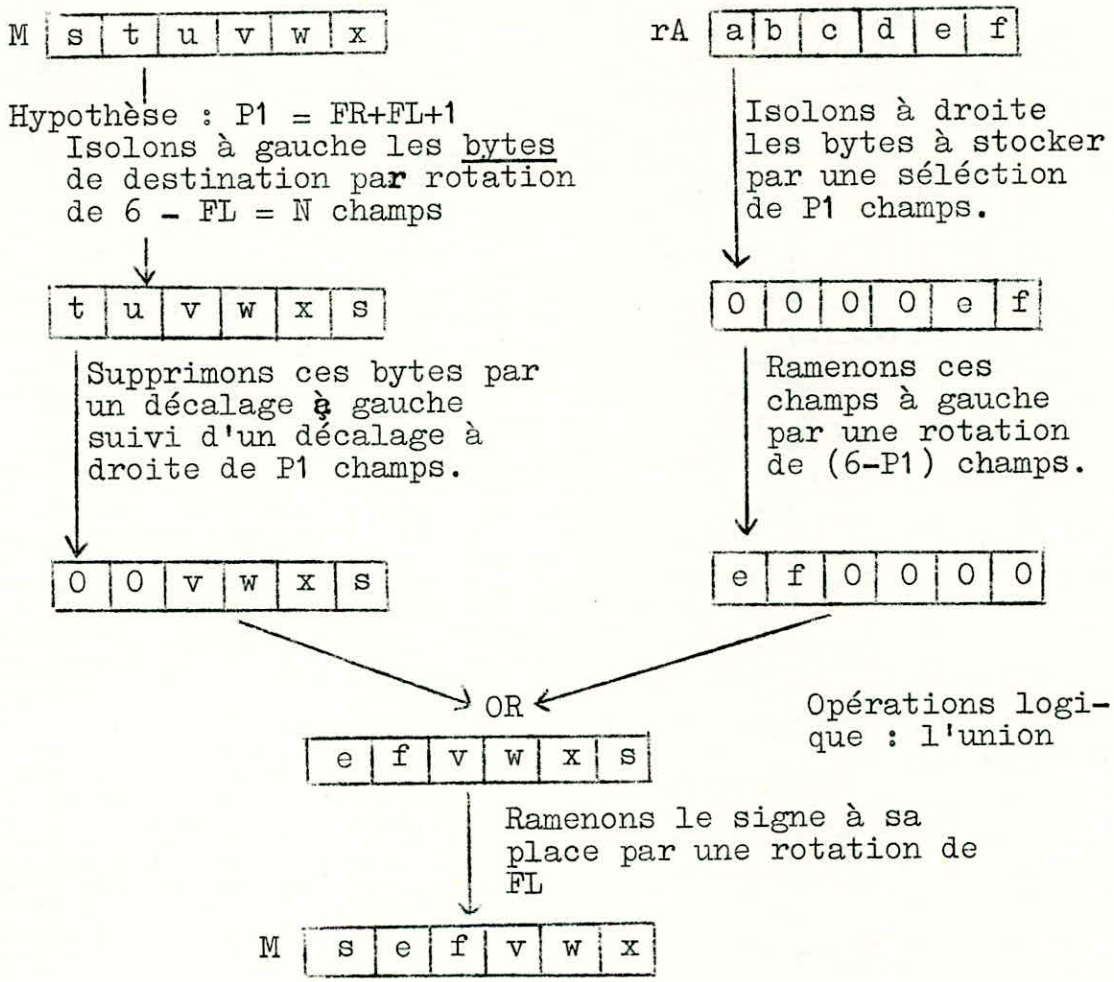
Cela revient à traiter :

F = (0:FR) ⟶ F = (0:0)+(1:FR)

L'instruction STA M (1:2), consite à stocker les deux bytes les plus à droite de rA dans les bytes 1 et 2 de la mémoire M.

.../...

S E L E C T I O N


M s t u v w x
Hypothèse : P1 = FR+FL+1
Isolons à gauche les bytes de destination par rotation de 6 - FL = N champs
t u v w x s
Supprimons ces bytes par un décalage à gauche suivi d'un décalage à droite de P1 champs.
0 0 v w x s
rA a b c d e f
Isolons à droite les bytes à stocker par une séléction de P1 champs.
0 0 0 0 e f
Ramenons ces champs à gauche par une rotation de (6-P1) champs.
e f 0 0 0 0
OR
Opérations logique : l'union
e f v w x s
Ramenons le signe à sa place par une rotation de FL
M s e f v w x

.../...

SECTION III

OPERATEURS ARITHMETIQUES

Ce sont les 4 instructions ADD, SUB, DIV, MUL. Une spécification de champ (0:6) peut être employé et désigne alors une opération en virgule flottante.

Ce cas-là n'a pas été simulé - l'assembleur y verra une erreur dans le champ.

La spécification standard du champ est (0:5). On désignera dans la suite par M l'adresse effective aprés indexation et V le contenu des champs de M spécifiés par l'instruction.

Exemple : dans OPR 2000,1(0:2) (OPR=Opération quelconque)

et si contenu de rI1=250

alors M=2250

et si 2250 contient :

+	04	05	62	41	49

alors :

V = | + | 00 | 00 | 00 | 04 | 05 |
|---|---|---|---|---|---|

I - L'addision: ADD - C=1,F=champ -

V est additionné au registre accumulateur.

Le résultat est dans le registre accumulateur.

Exemple : si rA = | + | 04 | 13 | 12 | 15 | 17 |
|---|---|---|---|---|---|

et M = | + | 32 | 63 | 12 | 24 | 39 |
|---|---|---|---|---|---|

ADD M, donne comme résultat dans le registre accumulateur

+	37	12	24	39	56

.../...

on remarque que lorsque le contenu d'un byte atteint ou dépasse 64, le reste est mis dans ce byte et une unité est ajoutée au byte précédent.

ADD M(2:3) donne d'abord V = | + | 00 | 00 | 00 | 63 | 12 |

et comme résultat final :

rA = | + | 04 | 13 | 13 | 14 | 39 |

ADD M(0:4) donne rA = | + | 04 | 46 | 11 | 27 | 41 |

autre exemple :

Additionner les 5 bytes du registre accumulateur.

```
STA   2000
LDA   2000(5:5)
ADD   2000(4:4)
ADD   2000(3:3)
ADD   2000(2:2)
ADD   2000(1:1)
```

II - La soustraction SUB. C=2, F=champ.

V est soustrait du registre accumulateur .
Le résultat est dans le registre accumulateur.

Exemple : supposons rA et M contenant les mêmes valeurs que celles dans l'addition.

SUB M donne | + | 04 | 13 | 12 | 15 | 17 |

— | + | 32 | 63 | 12 | 24 | 39 |

soit | - | 28 | 50 | 00 | 09 | 22 |

.../...

SUB M(2:3) donne | + | 04 | 13 | 12 | 15 | 17 |

\- | + | 00 | 00 | 00 | 63 | 12 |

et dans rA le résultat final :

| + | 04 | 13 | 11 | 16 | 05 |

REMARQUE SUR L'ADDITION ET LA SOUSTRACTION :

- Si le résultat de l'opération ne tient pas sur le registre accumulateur c'est-à-dire s'il est inférieur ou égal à -2^{30} ou supérieur ou égal à 2^{30}, l'overflow se met à ON. Sinon l'état de l'overflow reste inchangé.
- Si le résultat est nul, le signe de rA est inchangé.

  ainsi -4 +4 = - 0

  +4 -4 = + 0

Exemple dépassement

- Addition:

| - | 33 | 34 | 05 | 62 | 63 | + | - | 38 | 13 | 62 | 11 | 01 |

donne | - | 07 | 48 | 04 | 10 | 00 | et l'overflow se met à ON.

- Soustraction :

| + | 45 | 18 | 19 | 32 | 50 | - | - | 24 | 63 | 37 | 39 | 15 |

donne | + | 06 | 17 | 57 | 08 | 01 | et l'overflow se met à ON.

.../...

III - Siumulation de ADD et SUB .

Ces deux instructions sont simulées dans le module ADSOU. La simulation de ADSOU est facile du fait qu'il existe dans l'assembleur 11.30 deux instructions AD = addition double et SD = soustraction double additionnant ou soustrayant 2 mots consécutifs aux des registres accumulateur - extension.

ADSOU

```
TRAITEMENT - A
TRAITEMENT - B
TRAITEMENT - C
TRAITEMENT - D
TRAITEMENT - E
```

TRAITEMENT A

- Rameur rA de la représentation MIX à la représentation 11.30=ACC
- S1 = Signe de rA
- S2 = Signe du 2éme opérande
- RCM= Valeur absolue du 2éme opérande .

TRAITEMENT B

```
IF (code+S2=2)
        THEN RESULTAT = ACC-RCM
        ELSE RESULTAT = ACC+RCM
END
```

TRAITEMENT C

```
IF (Valeur absolue du résultat ≥ 2^30)
        THEN  TOG=1,
              prendre le résultat
              modulo 2^30
        ELSE  Rien
END
```

TRAITEMENT D

Ramener le résultat de la représentation

11.30 à la représentation MIX

TRAITEMENT E

```
IF (Résultat=0 )
      THEN  Résultat avec S1 comme
            signe
      ELSE  Rien
END
```

PASSAGE-REPRESENTATION MIX-REPRESENTATION 11.30

```
IF (nombre MIX <0 )
      THEN  prendre le complément à 2 de la
            valeur absolue
      ELSE  rien
END
```

PASSAGE-REPRESENTATION 11.30 - REPRESENTATION MIX

```
IF ( nombre 11.30 <0 )
      THEN  prendre la valeur absolue,
            accoler le signe (-)
      ELSE  rien
END
```

IV - L'opérateur produit - MUL C=3. F=champ

La quantité V est multipliée par rA.
Le résultat est sur 10 bytes : rA et rX.
rA et rX seront de même signe et ils ont :

- le signe (+) si V et rA précédent ont le même signe.
- sinon le signe (-) .

.../...

Exemples :

1. rA = | - | | | | 10 | 24 |

V = | + | 4 | 2 | 6 | 3 | 32 |

Le résultat sera $(10.64+24)\ (4.64^4+2.64^3+6.64^2+32)$

soit $41.64^5+53.64^4+46.64^3+52.64^2+20.64$

c'est-à-dire

rA

-	00	00	00	00	41

rX

-	53	46	52	20	00

2. rA = | - | 50 | 00 | 01 | 48 | 04 |

V = | - | 02 | 00 | 00 | 00 | 00 |

Le produit sera $50.64^4+64^2+48.64+4 \quad 2.64^4$

soit $100.64^8+2.64^6+96.64^5+8.64^4$

rA

+	100	00	03	32

rX

+	08	00	00	00	00

3. rA = | - | 05 | 00 | 03 | 02 | 33 |

V = | + | 00 | 00 | 00 | 00 | 10 |

Ce produit donne.

-	00	00	00	00	00

-	50	00	30	25	10

.../...

Cette multiplication suit le même principe que la multiplication habituelle:ainsi dans la multiplication des derniers bytes de rA et V

$33 \times 10 = 330 = 5.64+10.$

c'est à dire que 5 unités passent dans le byte précedent et 10 sont retenus dans ce byte.
Il en est de même pour le produit des différents bytes.

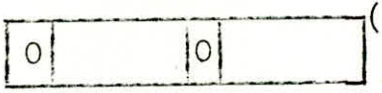
V - Simulation de la multiplication.

Nous donnons deux méthodes pour la simulation. seule la deuxième méthode a été simulée.

1°) - Première méthode :

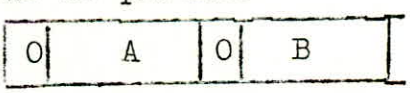
- ramener chacun des deux opérandes aprés avoir pris leur valeur absolue de la forme :

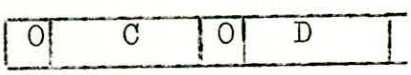
(un mot MIX),

à la forme 

(2 mots 1.130)

- Ainsi il revient à avoir le produit

si 1ér opérande = 

et 2ém opérande = 

$(A.\ 2^{15}+B)\ (C.2^{15}+D) = AC.2^{30}+(AD+BC)2^{15} + BD\ .$

or les différents produits AC, AD, BC, BD sont facilement obtenus à l'aide de l'instruction assembleur 1.130 .
(M (multiplication))
Le problème se pose lorsqu'on fait la multiplication $(AD+BC)\ 2^{15}$ et ce produit risque de dépasser la capacité de 2 mots 1.130 .

.../...

2°) - Deuxième méthode : calquée sur la méthode qu'on applique habituellement pour faire des produits à la main. Cette méthode est simulée dans MULDI dans sa deuxième partie - La première partie simulant la division.

MULTIPLICATION

```
TRAITEMENT A
TRAITEMENT B
TRAITEMENT C
```

TRAITEMENT A

- Localiser les champs du 2ém Opérande = V
- S1 = signe de rA.
- S2 = signe de V.
- Acc = |rA|
- RCM = |V|

TRAITEMENT B

- PROD=0
- i=1
- Traitement D

TRAITEMENT C

- Trouver le signe pour PROD
- PROD va dans rA et rX

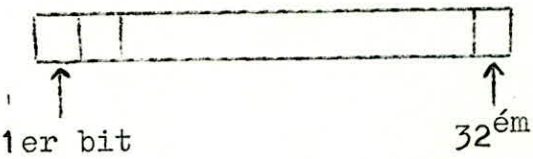
TRAITEMENT D

```
DO WHILE i ≤32
   IF (iéme bit = 1)
      THEN  BIDON=RCM
            décalé de 32-i
            PROD=PROD+BIDON
            i=i+1
    ELSE    i=i+1
   END
```

.../...

Remarque :

Un double mot 1130 est représenté comme suit :



VI - L'opération Division DIV - C=4. F=champ.

1. Fonction : la valeur de rA et rX, traité comme un nombre de 10 bytes avec le signe de rA, est divisée par V.
Si V=0 ou si le quotient ne tient pas sur 5 bytes (c'est-àdire lorsque $|rA| \geq |V|$), l'overflow se met à ON.
Sinon le quotient est mis dans rA avec le signe algébrique du quotient et le reste est mis dans rX avec le signe précédent de rA.
Dans le cas ou V=0, quotient=maximum possible c'est-à-dire :
$-(2^{30}-1)$ ou $2^{30}-1$ et reste =0 .
Dans le cas ou V≠0 et que le quotient n'arrive pas à tenir sur les 5 bytes de rA, on trouvera dans rA le quotient modulo 2^{30} et dans rX le reste exact.

2. Exemples :

a.) supposons

	rA					rX					
	+	0	0	0	0	0	+	0	0	0	0	17

V =

+					3

.../...

Le résultat sera:

rA = | + | 0 | 0 | 0 | 0 | 5 |

| + | 0 | 0 | 0 | 0 | 2 |

b.) Soit : rA = | - | 0 | 0 | 0 | 0 | 0 |

rX = | + | 1235 | 0 | 3 | 0 |

V = | - | 0 | 0 | 0 | 2 | 0 |

le quotient sera négatif car rA et rX sont de signes contraires . Le reste sera négatif car rA est négatif.

Aprés division rA et rX se présentent comme suit:

rA = | + | 0 | 617 | 32 | 0 |

rX = | - | 0 | 0 | 0 | 1 | 0 |

En effet il s'agissait de diviser (prenons les valeurs absolues) :

$1235.64^3 + 3.64$ par 2.64 ce qui donne :

$$\frac{1235}{2}\ 64^2 + \frac{3}{2} = 617.64^2 + \frac{1}{2}.64^2 + 1 + \frac{1}{2}$$

$$= 617.64^2 + 32.64 + 1 + \frac{1}{2}$$

on voit que : quotient $= 617.64^2 + 32.64 + 1$

reste $= \frac{1}{2} . 2.64 = 64$

c.) soit :

rA= | + | 0 | a | b | c | d |

rX= | + | e | 0 | 0 | 0 | 0 |

V = | + | a | b | c | d | e |

aprés division on a :

rA= | + | 1 | 0 | 0 | 0 | |

rX= | + | 0 | 0 | 0 | 0 | 0 |

d,) - Cas où le quotient ne tient pas sur rA.

rA= | + | 0 | 0 | 0 | 0 | 3 |

rX= | + | 0 | 0 | 0 | 0 | 1 |

V = | + | 0 | 0 | 0 | 0 | 2 |

Le résultat sera

$$\frac{3.64^5+1}{2}=\frac{3}{2}.64^5+\frac{1}{2}=64^5+32.64^4+\frac{1}{2}$$

quotient = $64^5+32.64^4$

reste = $\frac{1}{2}.2=1$

Dans ce eas rA et rX se présenteront comme suit :

.../...

rA= | + | 32 | 0 | 0 | 0 | 0 |

rX= | + | 0 | 0 | 0 | 0 | 1 |

le 64^5 n'apparait pas dans le quotient.

Pour éviter ça le programmeur peut suivre l'algorithme suivant :

but : Diviser (rA,rX) par V.

- sauvegarder rX dans SRX.
- IF |RA| > |V| THEN |ra| / |V| → quotient Q_1 et reste R_1

  mettre R_1 dans rA
  et restaurer rX.

  END
- Diviser (rA,rX) par V → quotient Q_2 et reste R_2.

Alors le résultat global sera :

$$\text{quotient} = Q_1 . 64^5 + Q_2$$
$$\text{reste} = R_2$$

VII. - Simulation de la division :

La dision est simulée dans MULDI.

posons le problème et raisonnons seulement dans le cas où dividende et diviseur positif.

Soit ACC, EXT, RCM les valeurs respectives de rA, rX et V (valeurs absolues).

Le but est donc de diviser :

$$ACC.2^{30} + EXT \text{ par } RCM$$

DIV1 = réaliser la division $ACC.2^{30}$ + EXT par RCM en supposant savoir diviser tout nombre inférieur à 2^{31} par RCM et cette 2ém. opération sera réalisée par,

DIV2 = la méthode utilisée pour DIV2 est celle qu'on emploie habituellement pour une division à la main.

.../...

Division

```
Traitement  A
Traitement  DIV1
Traitement  C
```

Traitement A

```
- Localiser les champs du 2ém. opérande = V
- S1  =  signe de rA
- ACC = |rA|
  EXT = |rX|
  RCM = |V|
  S2  =  signe de V
- IF  ( RCM = 0 )
      THEN  TOG = 1
            Message division par 0
      ELSE  R2 = rang du 1er. bit = 0
                 dans RCM
```

Traitement C

```
IF  ( QUOT ≥ 2^30 )
    THEN   TOG = 1
           message quotient trop élevé
           Trouver signe pour quotient et
           reste.
           quotient va dans rA.
           reste va dans rX.
END
```

.../...

Traitement DIV1

- QUOT1 = 0
- Traitement DIV2 (division ACC par RCM)
  QUOT1 = QUOT2.2^{30}
- i = 0
- DO WHILE (i < 30)
  DO WHILE (ACC.2^{i} < 2^{30})
  i = i + 1
  traitement DIV2
  BIDON = QUOT2 décalé de i bits
  vers la gauche
  QUOT1 = QUOT1 + QUOT2
  END
  END
- Traitement DIV2 pour (ACC + EXT) par RCM.

Traitement DIV2

QUOT = 0
Traitement D

Traitement D

R1 = Rang du 1er bit ≠ 0 dans ACC
D = R2 _ R1
Traitement E

.../...

Traitement E

```
IF  ( D<0 )
    THEN  fin traitement de la division 1
    ELSE  BIDON1 = RCM décalé de D bits vers
                   la gauche.
          REST = ACC - BIDON1
          IF ( REST<0 )
                THEN D = D - 1
                     traitement E.
                ELSE BIDON2 = 1 décalé de D
                              bits vers la
                              gauche.
                     QUOT2 = QUOT2 + BIDON2
                     ACC = REST
                     Traitement D
          END
END
```

.../...

SECTION IV

OPERATEURS DE MODIFICATION DES REGISTRES

MODULE C4855

Pour l'ensemble de ces opérations le code opération est un couple de valeur (C , F) .

l'opérande de l'instruction ne contient pas de spécification de champ.

I - OPERATIONS:

C \ F	0	1	2	3
48	INCA	DECA	ENTA	ENNA
48+i	INCi	DECi	ENTi	ENNi
55	INCX	DECX	ENTX	ENNX

$1 \leqslant i \leqslant 6$

INC : incrémenter le contenu du registre RA ou RX ou RIi de la valeur NB + RI(I)

DEC : Décrémenter.

ENT : Mettre la valeur NB + RI(I) dans RA ou RX ou RIi

ENN : Mettre la valeur - (NB + RI(I)) dans RA ou RX ou RIi.

II FORME DE L'INSTRUCTION

ETIQ OPER NB ,I

NB a une valeur de nombre et non d'adresse; $1 \leqslant I \leqslant 6$.

1.3

Traitement de RA

- IF (DEC ou ENN) se ramener à INC ou ENT

  END.

- IF (ENT) se ramener à INC

  END.

- RA = RA + MEJNC (ADSOU)

- IF (PAR = 1)
  THEN traitement 2 de RIi
  ELSE IF (PAR = -1)
  THEN traitement 2 de RX
  ELSE FIN
  END

  END

1.3.1

Traitement 2 de RIi

- Prendre contenu de RA
- Tasser (supprimer les bytes 1,2,3).
- Stocher dans RIi concerné
- Restaurer RA
- Aller FIN .

1.3.2

Traitement 2 de RX

- Prendre RA, le stocker dans RX
- Restaurer RA
- Aller à fin.

III - EXEMPLES:

3-1 : RX (avant) | - | 24731 |

RI3 | + | 94 |

RX (aprés) | - | 24661 |

INCX - 24,3

NB + RJ (J) = - 24 + 94 = 70

RX = - 24731 = 70 = 24661

3-2 : RA (Avant) | - | 327891 |

RA (Aprés) | + | 25 |

ENNA - 25

IV - SIMULATION:

Les sous-programmes qui sont appelés sont:

- ROME
- ETIRE
- ADSOU
- PACTE

présentation de la simulation par la méthode de la programmation structurée:

1

- Récupérer la valeur NB + RI (I) = MEINC
  par = 0
  IF (VERBI) then traitement de RIi
  else IF (VERBX)
  then traitement de RX
  ELSE traitement RA
  END

  END

1.1 -

Traitement RIi
- Récupérer RIi concerné
- Etendre RIi sur 2 mots 1130
- Mettre dans RA , PAR = 1
- Aller traitement de RA

1.2 -

Traitement de RX
- Mettre RX dans RA
- PAR = - 1
- Aller traitement RA

SECTION V

OPERATEURS DE COMPARAISON

I - FONCTION :

Permettent de comparer les champs spécifiés (dans l'instruction) de la mémoire M avec les champs correspondants d'un des huit registres (rA, rX ou rII avec I = 1 à 6).

Selon que c'est CMPA, CMPX ou CMPI, $1 \leq I \leq 6$, l'indicateur de comparaison ou LEG se met à l'état LESS Ou à l'état EQUAL ou encore à GREATER selon que la valeur pour le registre est plus petite, égale ou plus grande que la valeur pour le mot-mémoire.

Si le champ ne spécifie pas le signe, on comparera donc deux nombres positifs; sinon le signe est inclus dans la comparaison.

Si F = (0:0), l'indicateur de comparaison se met à l'état EQUAL quel que soit le signe des mémoires concernées.

C'est à dire que + 0 = - 0 .

Dans une comparaison concernant un registre index, les bytes 1,2 et 3 de ce registee sont traités comme des zéros.

Ainsi CMP1 2000, 2 (L:R)

supposons rI2 = 500

et 2500 :

+	12	13	18	20	32

r I1 :

+	14	19

Il revient à comparer les champs (L:R) de 2500 avec les mêmes champs de

+	00	00	00	14	19

l'indicateur de comparaison se positionnera à l'état L.

Exemple : RA

-	04	03	09	07	08

200

+	03	04	01	02	05

* CMPA 200 (3:4)

positionnera le LEG à G car 9.64 +7 > 64 + 2

* CMPA 200 (0:5)

positionnera le LEG à E car RA est négatif ≠ le contenu de 200 est positif .

* CMP2 300 (0:4)

avec RI2

+	01	07

300

-	02	08	02	04	08

positionne le LEG à G car

$1 > -\left[2.64^4 + 8.64^3 + 2.64^2 + 4.64 + 8\right]$

II - SIMULATION

Les comparaisons sont traitées dans le module C5663.

C5663

```
CASE        Code     OF
            56 :  traitement CMPA
            57 à 62 : traitement  CMPI
                      traitement  CMPA
            63 :      mettre rX dans rA
                      traitement CMPA

ESAC
```

Traitement ETIRE

- Trouvez le registre concerné I = Code - 56
- Etendre le contenu du registre sur 2 mots 11.30
- Mettre le résultat dans rA.

Traitement CMPA

- Isoler les champs du mot MIX ⟶ MECOM
- Isoler les champs du registre RA ⟶ ACC
- RESUL = ACC - MECOM
- IF (RESUL > 0)

```
        THEN     LEG  = 1
        ELSE     IF ( RESUL  = 0 )
                        THEN     LEG  = 0
                        ELSE     LEG  = - 1

                 END

  END
```

Les sous-programmes utilisés par C5663 sont

ROME
ETIRE
SEULS
ADSOU
PONE

} dont on a donné une explication détaillée au début.

S E C T I O N VI

OPERATEURS DE SAUT

I - Fonction :

Les instructions d'un programme sont généralement exécutées dans un ordre séquentiel. Les instructions de saut permettent d'interrompre cette séquence.

Toutes les fois qu'un saut arrive, le registre J contiendra l'adresse de l'instruction qui suit celle de saut c'est à dire la valeur du compteur ordinal puisque celui-ci contient l'adresse de la prochaine instruction à exécuter. La seule exception est pour l'instruction JSJ. Dans ce cas, la valeur du registre J est inchangée.

REMARQUE : Le registre J n'est chargé que quand un saut arrive (excepté STJ) et il n'y a aucune instruction nous permettant de mettre une valeur dans ce registre.

II - Les différents opérateurs de saut :

Il y a les opérateurs de saut inconditionnel et les opérateurs de saut conditionnel.

II.1. Opérateurs de saut inconditionnel.

a). JMP (jump). C = 39. F = 0

Cette instruction provoque un saut inconditionnel à l'adresse spécifiée dans l'instruction et sauvegarde la valeur du compteur ordinal en la mettent dans rJ.

Exemple : supposons que l'instruction JMP 3000,5 soit à l'adresse 4000 et que r15 contienne 250.

.../...

cette instruction :

- provoque un saut à l'instruction située à 3000 + 250 = 3250 .
- met la valeur 4001 dans le registre J, quel que soit le contenu précédent de celui-ci.

b). JSJ (jump save rJ) C = 39. F = 1
même chose que JMP seulement la valeur de rJ est inchangée.
ainsi : JSJ 3000,5 avec contenu de rI5 = 250 provoque uniquement un saut à 3250.

II.2. Opérateurs de saut conditionnel.

a). JOV (jump on overflow) C = 39. F = 2 .
si l'overflow est ON, cette instruction le remet à OFF et provoque avec sauvegarde de la valeur du compteur ordinal dans rJ .
sinon, il ne se passe rien. Soneffet devenant alors celui d'un NOP .

b). JNOV (jump on no overflow) C = 39. F = 3.
cette instruction provoque un saut à l'instruction d'adresse M si l'overflow est OFF.
si l'overflow est ON, il est mis à OFF uniquement.

c). JL (jump on less)..... C=39. F=4
JE (jump on equal)..... C=39. F=5
JG (jump on greater)..... C=39. F=6
JGE (jump on greater or equel)... C=39. F=7
JNE (jump on no equal)..... C=39. F=8
JLE (jump on less or equel)..... C=39. F=9

Pour toutes ces instructions il y a saut si l'indicateur de comparaison estàl'état indiqué.
l'indicateur de comparaison n'est pas affecté.

.../...

Exemple : supposons que l'instruction JNE 3000,1 soit à l'adresse 2000 et que rI1 contienne 900, ainsi :

- si l'indicateur est à l'état E, effet d'un NOP.
- si l'indicateur n'est pas à l'état, E on aura saut à l'instruction d'adresse 3900 et rJ = 2001 .

d). JAN (jump A negative) C=40. F=0
JAZ (jump A zéro) C=40. F=1
JAP (jump A positive) C=40. F=2
JANN(jump A nonnegative) C=40. F=3
JANZ(jamp A nonzéro) C=40. F=4
JANP(jump A nonpositive) C=40. F=5

Il y a saut si le contenu de rA satisfait la condition.

Ramarque : zéro signifie + zéro ou - zéro.

Exemple : JAZ 3000
provoque un saut à 3000 si rA est nul.

e). Les instructions décrites précédemment existent également pour les 6 registres index et pour le registre extension rX et elles ont le même rôle que celles concernant rA.
Ce sont :
JXN, JXZ, JXP, JXNN, JXNZ, JXNP
avec
F=0 F=1 F=2 F=3 F=4 F=5
respectivement et C=40 pour le registre extension.
et
JiN, JiZ, JiP, JiNN, JiNZ, JiNP
avec C=40+i $1 \leq i \leq 6$
et
F = 0 F=1 F=2 F=3 F=4 F,5 respectivement pour les différents registres.

.../...

III - Simulation des opérateurs de saut ayant pour code C = 39 -

opérations traitées dans le module SAU39

SAU39

```
CASE   F     OF
   0 :     BSPI
   1 :     IAR  = RAM
   2 :     IF ( TOG = 1 )
                 THEN  TOG=0
                       BSPI
                 ELSE  SORTIE
           END
   3 :     IF  ( TOG = 1 )
                 THEN  TOG=0
                 ELSE  BSPI
           END
   4,5 ou 6:
           IF  ( F - LEG = 5 )
                       THEN  BSPI
                       ELSE  SORTIE
           END
   7,8 ou 9:
           IF  ( F - LEG ≠ 8 )
                       THEN   BSPI
                       ELSE   SORTIE
           END
ESAC
```

traitement de BSPI

```
rJ = IAR
IAR= RAM
```

.../...

IV - Traitement des opérations de saut portant sur des tests sur les registres.

c'est-à-dire JØN, JO2, JØP, JØN2, JØNP

avec Ø = { A si le test porte sur le rA.
         { i avec 1 ≤ i ≤ 6 si le test porte sur le registre i.
         { X si le test porte sur le rX.

Ces opérations sont traitées dans le module C4047 -

C4047

```
CASE    C    OF
    40 :  ACEXT = rA
          TRAITEMENT A
    41 à 46 : AC EXT = rI ( C-40 )
          TRAITEMENT A
    47 :  ACEXT = rX
          TRAITEMENT A.
 ESAC
```

TRAITEMENT A.

```
TRAITEMENT  PONE
TRAITEMENT  B
```

TRAITEMENT PONG

```
IF ( ACEXT   négatif)
      THEN   INDICATEUR = - 1
      ELSE   IF ( ACEXT positif ) THEN  INDICATEUR = + 1
                                  ELSE  INDICATEUR = 0
             END
END
```

.../...

TRAITEMENT B.

```
CASE      F       OF
 0,1 ou 2 :      IF  (F - INDICATEUR = 2)
                              THEN      BSPI
                              ELSE      SORTIE
                 END
 3,4 ou 5 :      IF  (F - INDICATEUR = 4)
                              THENT     SORTIE
                              ELSE      BSPI
ESAC
```

BSPI

```
rJ = IAR
IAR= RAM
```

..../...

S E C T I O N VII

LES OPERATEURS D'ENTREES / SORTIES

I. - Les unités d'entrées / sorties .

Chaque unité a un numéro d'identification

N° de l'unité	Unité	Dimension de l'enregistrement
1	Console	16 mots - MIX
2	Lecteur-perforateur	16 mots - MIX
3	Imprimante	24 mots - MIX

II. - Les fonctions.

- d'entrées : IN } traitées dans le module INOUT
- de sorties: OUT }
- de test : JBUS - module C34
- de contrôle: IOC - module C35
- de dumpage de mémoires = JRED - module C38

III. - Module INOUT

1.- Les opérateurs IN et OUT :

Les entrées et les sorties sur cartes, console ou imprimante sont faites dans un code caractère où chaque byte représente un caractère alphaNUMErique .Ainsi 5 caractères sont transmis par mot-MIX.

D'aprés la table on voit que :

00 est le code de blanc.

01,02,.....,29 sont les codes de A,B,....Z

30,31,.....,39 sont les codes de 0,1,....9

40,41,.....,55 sont les codes de caractères spéciaux.

..../...

Tout code correspondant à un caractère admis dans MIX et n'existant pas sur l'une des unités du 1130 (exemple la lettre têta) ainsi que tout code ne correspondant à aucun caractère (exemple 60 ne correspond à aucun caractère) sera considéré par l'unité comme le code de ET Commercial.

Le signe des mots remplis par un IN est (+)
En sortie les signes sont ignorés.

* IN. C=36. F = numéro de l'unité.

Syntaxe IN M (F)

Cette instruction réalise le transfert d'information à partir de l'unité F vers les mots MIX M, M+1,...,M+15. Ainsi la longueur de l'enregistrement pour une unité est le nombre de mots transférés.

Exemple : IN 200 (2)
provoque la lecture d'une carte. Supposons que la carte contienne l'information suivante:

LIRE CETTE CARTE 12345

Cette information sera mise dans les mémoires 200,201,...,215.

200	+	13	09	19	05	00
201	+	03	05	23	23	05
202	+	00	03	01	19	23
203	+	05	00	31	32	33
204	+	34	35	00	00	00
205	+	00	00	00	00	00
215	+	00	00	00	00	00

Dans le mot 200, le byte 1 contient le code de L le 2ém. byte de I,...., le 5ém. byte le code de blanc.

.../...

* OUT. C=37. F = numéro de l'unité

OUT M (F) provoque l'impression sur console (F=1) ou imprimante (F=3) ou la perforation d'une carte (F=2) de l'information contenue dans M à M+15 (si F=1 ou 2) ou à M+23 (si F = 3).
Ainsi si l'on suppose la mémoire 200 contenant l'information (cf exemple IN),

OUT 200 (1) imprimera sur la console le message :

LIRE CETTE CARTE 12345

2. - Simulation :

Il a été nécessaire de faire des sous - programmes permettant de passer du code d'une unité du 1130 au code MIX et réciproquement. Ce sont :

CON25
CARMX
CONVR

Traitement de INOUT

```
IF ( C impair )
    THEN  traitement de OUT
    ELSE  traitement de IN
END
```

.../...

Traitement de OUT

```
CASE   F    OF
   1 - traitement OUT console
   2 - traitement perforateur
   3 - traitement imprimante
ESAC
```

Traitement de IN

```
CASE  F     OF
   1 - traitement IN console
   2 - traitement lecteur
ESAC
```

Traitement de OUT console

- CON25 : Conversion code MIX en code EBCDIC tassé
- EBPRT : Conversion code EBCDIC tassé en code console
- Impression.

Traitement perforateur

- CON25
- SPEED : (code EBCDIC en code carte)
- Perforation.

Traitement imprimante

- CON25
- Impression

.../...

Traitement IN console

- Lecteur de 80 caractères
- Test fin de lecture
- CARMX (conversion code carte en code MIX)

Traitement lecteur

- Lecture d'une carte
- Test de fin de lecture
- CARMX (conversion code carte en code MIX)

3. - Les sous-programmes utilisés par INOUT

3 - 1 S/P CON25

a) but

Ce sous-programme est utilisé dans le cas d'une instruction OUT. Sur l'imprimante 1132, la console et le perforateur 1442 il n'est pas possible de sortir directement avec du code MIX. Il faut convertir ces données en code approprié à l'unité de sortie.

CON25 convertit 2n mots MIX (l'adresse du 1er mot est spécifié en opérande de l'instruction OUT) en 5n mots 1130 contenant du code EBCDIC tassé (n=12 pour imprimande et n=8 pour perforateur et console).

L'appel de ce sous-programme se fait par :

```
CALL    CON25
DC      SORTI
DC      n
```

.../...

SORTI est l'adresse où sont rangés les codes EBCDIC tassé, définis dans le programme appelant CON25.
n : paramètre (cf fonction de CON25) .
CON25 appelle le sous-programme CONVR avec comme paramètres :

```
DC    ECLAT
DC    M
```

ECLAT est l'adresse d'une table de 10 mots 1130 définie dans le sous-programme CON25.
M = /0A01 indique la conversion de 10 codes MIX. en EBCDIC .

b) Simulation :

Traitement de CON25

- Récupérer l'adresse où stocker : ADSOR
- Récupérer le paramètre n
- I = 1
- DO WHILE (I ≤ n)
  - Eclater 2 mots MIX en 10 bytes et stocker ces 10 bytes dans ECLAT
  - Convertir ces 10 codes MIX en 10 codes EBCDIC non tassé.
  - Tasser les 10 codes EBCDIC contenus dans la table ECLAT en 5 mots que seront stockés dans ADSOR
- END

3 - 2:S/P CARMX

a) but

Ce sous-programme est utilisé par l'instruction d'E/S IN. Il convertit 80 codes cartes occupant 80 mots du 1130, en 80 codes caractéres MIX occupant 16 mots MIX.

Les mots MIX sont rangés à partir de l'adresse donnée par l'opérande de l'instruction :

```
IN   M,I ( unité )
```

M+ rI (I) = adresse où est mis le 1er. mot MIX.généré.

L'appel de CARMX se fait par :

```
CALL    CARMX
DC      ADR
```

sachant que dans le programme appelant on a :

```
ADR   BSS   80
```

CARMX appelle les sous-programmes :

- SPEED pour convertir du code carte en EBCDIC non tassé.
- CONVR pour convertir les codes EBCDIC non tassés en code MIX (à l'entrée et à la sortie de CONVR on a un code par mot).

b) Simulation :

Traitement de CARMX

- Récupérer l'adresse de la table contenant les codes cartes
- Conversion des 80 codes cartes en 80 codes EBCDIC non tassés (SPEED)
- Conversion des 80 codes EBCDIC non tassés en 80 codes MIX (CONVR)
- I = 1
- DO WHILE (I $\leqslant$ 16)

  Mettre 5 codes MIX occupant 5 mots 1130 dans un mot MIX

  I = I+1

  END

.../...

3.3 SOUS-PROGRAMME CONVR :

a) but

Ce sous-programme permet de passer du code caractère MIX au code caractère EBCDIC 1130 et également le passage inverse.

Dans les 2 passages et que ce soit en entrée ou en sortie, il suppose qu'il y a un code caractère par mot 1130. Si c'est un code caractère MIX, il est cadré à droite sinon il est cadré à gauche.

L'appel se fait de la façon suivante :

```
        CALL    CONVR
        DC      ADRES
        DC      M
avec M  DC      /XYOi
```

i = | 0 conversion EBCDIC - Code MIX
    | 1 conversion code MIX - Code EBCDIC

XY = nombre de caractères à convertir
(nombre en hexadécimal).

Pour la conversion, on a fait une table où l'on a mis les codes-caractères EBCDIC corespondant à l'ordre croissant des codes-caractères MIX. Comme le code caractère MIX tient sur 6 bits (pouvant donc prendre des valeurs de 0 à 63), on a une table de 64 éléments. Pour les caractères dont le code n'existe pas en EBCDIC (ex. la lettre grecque pi) et pour les codes ne correspondant à aucun caractère (exemple 60 ne correspond à aucun caractère), on a substitué le caractère ET Commercial .

.../...

b) Simulation :

Traitement CONVR

```
- j = 1
- DO  WHILE ( j ≤ nombre de caractères
             à convertir ( XY ) )

             CASE  i  OF
                  0 : Traitement EBCMIX
                  1 : Traitement MIXEBC
             ESAC
  END
```

Traitement EBCMIX

```
- i = 1
- DO  WHILE i ≤ 64 et iéme élément de TABLE
             différent du code EBCDIC donné

             i = i+1
  END
- Code MIX du code = i-1
- j = j+1
```

Traitement MIXEBC

```
- Code EBCDIC du code donné = élément
                de TABLE dont le rang est
                ( code+1 )

- j = j+1
```

.../...

IV. - L'opérateur JBUS - MODULE C34 :

Syntaxe: JBUS M(F)

M indique l'adresse à laquelle il faut se brancher si l'unité spécifiée par F est non prête (JBUS = Jump busy). Si l'unité est prête, le programme continue en séquence.

Remarque : Ce test doit se faire aprés une opération de sortie. Pour une opération d'entrée, il est inutile car il est fait dans CARMX. En effet il faut convertir les données en code-caractère MIX et cela n'est pas possible sans untest de fin de lecture.

Traitement C34

```
CASE  F   OF
   1.  IF ( console libre )
          THEN  Fin de traitement
          ELSE  traitement branchement
       END
   2.  IF ( perforateur libre )
          THEN  fin de traitement
          ELSE  traitement branchement
       END
   3.  IF ( imprimante libre )
          THEN  fin de traitement
          ELSE  traitement branchement

       END
ESAC
```

Traitement branchement

```
RJ = IAR
IAR = RAM
```

.../...

V.- CONTROLE E/S MODULE C35 :

1. Syntaxe :

IOC M (F)

M doit être une valeur et non un symbole

F = numéro de l'unité.

2. Signification par unité :

2.1. console : F=1

M ≠ blanc Aucun effet

2.2. perforateur : F=2

M = 3 : décalage d'une carte

M = 4 : séléction de stocker

2.3 imprimante : F=3

M = 1 : saut d'une page

M = 13: saut d'une ligne

M = 14: saut de deux lignes

M = 15: saut de trois lignes.

VI. - Dumpage mémoire :

Dans la version MIX, JRED est un opérateur de test.

JRED M (F)

provoque un saut si l'unité F est prête.

Opération rélisable avec JBUS; cela suggère de lui donner une autre version.

Nouvelle version :

L'opérateur JRED réalise un dumpage d'une zone de mémoire MIX. L'adresse du début de zone est donnée en opérande; celle de fin de zone est suposée mise dans le registre indec MIX 1.

Traitement JRED

- Premièr paramètre = RAM
- deuxième paramètre = rI1
- appel à DUMPA

.../...

SOUS-PROGRAMME DE DUMPAGE : DUMPA

Ce sous-programme permet de dumper une zone de mémoires MIX. L'appel se fait comme suit

```
CALL   DUMPA
DC     ADRS1
DC     ADRS2
```

toutes les mémoires dont les adresses MIX vont de ADRS1 à ADRS2 seront dumpées.

Prenons un exemple :

```
CALL   DUMPA
DC     255
DC     268
```

Le format des résultats se présente comme suit :

```
rA  00007441636  RX 12017540414  OVERFLOW OFF CINDICATOR E
rI1 00373   rI2   12000 ..... rI6  10404  rJ  1545
ADDR       *0          *1                      *7
0037*  11012140172  02357351761 ..... 10347400036
0038*  10101044050  16000743216 ..... 00000001440
0039*  00000000000  11006070012 ..... 10100550173
```

Explication :

Les résultats ainsi que les adresses sont donnés en base 8. On veut savoir par exemple le contenu de l'adresse 255. Cette adresse en base 8 s'écrit 377. Son contenu est donc 10347400036.

Le premier indique le signe de la mémoire. (1 pour négatif; 0 pour positif); les deux chiffres suivants le contenue du 1er byte,... et les deux derniers le contenu du 5ém. byte.

Donc la mémoire 255 contient :

1 00 47 40 00 36 soit | - | 03 | 35 | 32 | 00 | 30 |

.../...

Il en est de même pour rA et rX.
L'overflow peut avoir 2 états = OFF ou ON.
L'indicateur de comparaison 3 états = L,E ou G.
5 chiffres (base 8) définissent l'état de chacun des 6 registres index.
Le premier chiffre pour le signe, les 4 autres pour les deux bytes.
Pour rJ, 4 chiffres définissent son état puisque son signe est toujours (+).

Exemple :

rI1	00373	signifie	rI1 =	+	03	31
rI2	12000	signifie	rI2 =	-	16	00
rJ	1545	signifie	rJ =	+	13	37

DUMPA

- Traitement préparation (PREPA)
- Impression de ADDR *0 *1...*7
- Dans quel bloc se trouve la 1er.adresse.
- Trouver le nombre de blocs à dumper
- i = 1
- DO WHILE(i ≤ nombre de blocs à dumper).
  traitement du bloc i
  END

.../...

Traitement PREPA

- Traitement conversion en base 8 pour rA
- Traitement conversion en base 9 pour rX
- IF (TOG = 0)
  THEN mettre OFF
  ELSE mettre ON
- IF (LEG < 0)
  THEN mettre L
  ELSE IF (LEG) = 0
  THEN mettre E
  ELSE mettre G
  END

  END
- Traitement conversion pour chacun des registres.
- Impression des états de rA, rX des indicateurs et des registres index.

Remarque : Un "bloc" est composé de mots MIX pouvant tenir sur une ligne imprimante (8 mots - MIX).

Traitement d'un bloc

- j = 1
- DO WHILE j ≤ 8
  .traitement conversion en base 8 pour le jiéme mot du bloc.
  .j = j + 1

  END
- Impression du bloc.

Traitement: conversion en base 8

- Traitement du signe
- Eclater les 30 derniers bits du mot MIX en 10 parties de 3 bits.
- Convertir chacune des parties en base 8.

.../...

S E C T I O N VIII

OPERATEURS DE CONVERSION

MODULE CO5

1. - L'opérateur NUM. C=5 . F = 0

1. - Fonction : NUM suppose que dans les 10 bytes de rA et rX, il y a les codes-caractères de 10 chiffres.
NUM convertit les 10 codes-caractères en une valeur numérique obtenue comme décrit dans le paragraphe suivant.

2. - Génération du nombre :

Dans chaque byte de rang i, $1 \leq i \leq 10$, on a une certaine valeur. Soit C_i le reste de sa division par 10.
L'application de NUM donne :

$$N = \sum_{i=1}^{10} C_i . 10^{10-i}$$

N ainsi constitué est mis dans rA.
Le contenu de rX ainsi que le signe de rA sont inchangés.
Dans le cas où N dépasse la capacité de rA ($N > 2^{30}-1$ soit $N > 1.073.741.823$) il est pris modulo 2^{30} et l'overflow se met à ON.
Un exemple fera bien ressortir la fonction de NUM :

rA =	-	0	30	36	32	41

rX =	-	39	50	52	34	63

.../...

l'application de NUM donne :

rA = | - | 62190243 |

rX = | - | 39 50 52 34 63 |

on peut remarquer, d'après l'exemple, que dans le cas où l'un des bytes contient un code caractère différent de celui des 10 chiffres, tout se passe comme si on avait le code caractère d'un chiffre. Ainsi pour le 4ém. byte de rA, le résultat n'aurait pas changé si à la place de 32 (code du chiffre 2) on avait 42 (code de la parenthèse ouvrante);on le voit d'ailleurs sur le 5ém. byte de rA.

3. - Utilité de l'opérateur NUM .

Dans le cas d'une entrée de données numériques, on ne peut y effectuer des traitements avant de leur faire subir l'opérateur NUM. Donc toute entrée de données numériques doit être suivie d'une conversion à l'aide de NUM.

II. - L'opérateur CHAR. C = 5 . F = 1

1. - Fonction :

CHAR convertit une valeur numérique contenue dans rA en 10 codes caractères qui seront mis dans rA et rX. Les signes de rA et rX sont inchangés.

2. - Obtention des 10 codes caractères :

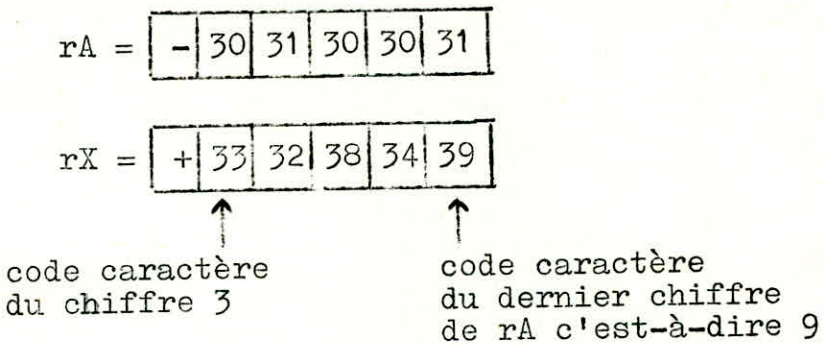
un exemple illustrera mieux cette opération.

rA = | - | 100132849 |

rX = | + | 432639512 |

.../...

aprés conversion par CHAR, rA et rX se présenteront comme suit :



3. - Utilité :

La conversion par CHAR est toujours préalable à une opération de sortie de valeurs numériques.

Remarques :

- forme de l'instruction :

ETIQUET	NUM
SYMBOL	CHAR

- pour ces deux instructions on n'a pas besoin d'un opérande.
- le champ F n'est pas spécifié car il sera généré par le producteur de codes à l'assemblage.

III. - Simulation de NUM et CHAR.

- I est le rang du byte
- CC(I) code caractère du byte I
- TABLE (I) : mot de 32 bits

La simulation schématisée sous forme de programmation structurée se présente sous la forme :

.../...

Traitement de C05

1

```
IF ( F = 0 )
   THEN  traitement de NUM.
   ELSE  IF ( F = 1 )
            THEN traitement de CHAR
            ELSE traitement de HLT
         END

   END
```

Traitement de NUM

1.1

- Isoler le signe de rA et le stocker dans SINUM.
- I = 1
- DO WHILE (I ≤ 10)
  TABLE (I) Reste de (CC(I) /10)
  END
- IF (TABLE (1) > 1)
  THEN TOG = 1
  END
- SOM = 0
- I = 1
- DO WHILE (I ≤ 10)
  SOM = SOM + TABLE(11-I)*10**(I-1)
  IF (DEPASSEMENT,)
  THEN TOG = 1
  END
  END
- Signe de SOM = signe de SINUM
- Stocker SOM dans rA .

.../...

Traitement A

S1 = Signe de rA
S2 = signe de rX
ACC= |rA|

Traitement B

```
i = 1
DO WHILE i < 10
   diviser ACC par 10
   ACC = quotient
   R (i) = reste
   code de R (i)=R(i)+30
   i = i + 1
END
```

Traitement de CHAR

Traitement A
Traitement B
Traitement C

Traitement C

- mettre les 5 premiers codes dans rA.
- mettre les 5 derniers codes dans rX
- restaurer les signes de rA et rX

La seule difficulté peut provenir de la division de ACC par 10 car : $0 \leqslant ACC \leqslant 2^{30}-1$ et le quotient peut ne pas tenir sur les 16 Bits d'un mot 1130. On résoud ce problème par des divisions successives.

Traitement de HLT

HLT est équivalent à l'instruction EXIT du 1130

HLT signifie la fin de l'éxécution; le contrôle est rendu au supérviseur.

.../...

S E C T I O N IX

OPERATEURS DIVERS

I. - L'opérateur MOVE. C= 7 - F = champ.

L'instruction s'écrit : MOVE M (F)

F mots MIX sont déplacés. L'adresse d'arrivée est donnée par M et l'adresse de départ par le registre index 1.

Le champ standard est F = 1.

Comme $0 \leqslant F \leqslant 63$, on ne peut déplacer plus de 63 mots MIX; à la fin de l'opération rI1 est incrémenté de la valeur de F.

Remarque :

- Si F = 0, l'effet de cet opérateur est celui d'un NOP.
- Il faut faire attention au chevauchement.

Exemples : F = 3. M = 1000 et rI1 contient 999.

Dans ce cas : le mot d'adresse 1000 va à l'adresse 999.

le mot d'adresse 1001 va à l'adresse 1000.

le mot d'adresse 1002 va à l'adresse 1001.

rien d'anormal n'arrive.

Supposons maintenant que rI1 contient 1001 et non 999 .

Alors tout se passe comme si le mot d'adresse 1000 va aux adresses 1001,1002,et 1003.

Autre exemple : F = 5. M = 200. rI1 contient 500.

le mot MIX d'adresse 200 est mis à l'adresse 500.

Le mot MIX d'adresse 201 est mis à l'adresse 501.

Le mot MIX d'adresse 204 est mis à l'adresse 504.

A la sortie rI1 contient 505.

MOVE est simulé dans le module C07 .

.../...

II. - Les opérateurs de décalages C = 6.

SLA (Shift left rA) F=0
SRA (Shift right rA) F=1
SLAX (Shift left AX) F=2
SRAX (Shift right AX) F=3
SLC (Shift left AX circulary) F = 4
SRC (Shift right AX circulary) F = 5

Les signes de rA et rX ne sont affectés par aucun des opérateurs. M spécifie le nombre de bytes dont il faut décaler.
SLA et SRA n'affectent pas rX.
Les autres opérateurs affectent rA et rX comme si l'ensemble se comporte comme un registre de 10 bytes.
Avec SLA,SRA,SLAX,SRAX des bytes disparaissent d'un coté et des zéros rentrent de l'autre.
SRC et SLC sont des décalages rotatoires dans lesquels les bytes qui sortent d'une extrémité rentrent de l'autre.

Exemples :

		rA						rX					
Contenu initial		+	1	2	3	4	5	-	6	7	8	9	10
SRAX	1	+	0	1	2	3	4	-	5	6	7	8	9
SLA	2	+	2	3	4	0	0	-	5	6	7	8	9
SRC	4	+	6	7	8	9	2	-	3	4	0	0	5
SRA	2	+	0	0	6	7	8	-	3	4	0	0	5
SLC	501	+	0	6	7	8	3	-	4	0	0	5	0
ou SLC	1												

.../...

III. - Simulation des opérateurs de décalage.

Le code opération des instructions de SHIFT est C = 6

1. Module SHIFM

```
Case F OF
0 : Traitement du SLA
1 : Traitement du SRA
2 : Traitement du SLAX
3 : Traitement du SRAX
4 : Traitement du SLC
5 : Traitement du SRC
ESAC
```

1.1 Sauvegarde des signes des registres

```
SIGNA = Signe du registre rA
SIGNX = Signe du registre rX
```

Quelque soit l'instruction de décalage, les signes de registres MIX rA et rX sont inchangés après le Shift.

Les instructions SLA et SRA travaille sur le registre rA fixé en carte mémoire, son signe est inchagé et le nombre n de bytes à décaler est inférieur ou égal à 5.

Les instructions SLAX, SRAX, SLC et SRC utilisent les registres rA et rX, leurs signes sont inchangés et le nombre des bytes à décaler est :

$1 \leqslant n \leqslant 10$ pour SLAX et SRAX

$n \geqslant 1$ pour SLC et SRC.

.../...

* Sélection :

. Les instructions SLA et SRA du langage MIX sont traitées de la même façon que celles du langage Assembleur 1130.

. Pour SLAX et SRAX, même traitement que les instructions SLT et SRT de l'assembleur 1130 sauf qu'on travaille sur l'accumulateur MIX (rA) occupant 2 mots du 1130 et l'extension MIX (rX) occupant 2 mots du 1130 .

* Exemples :

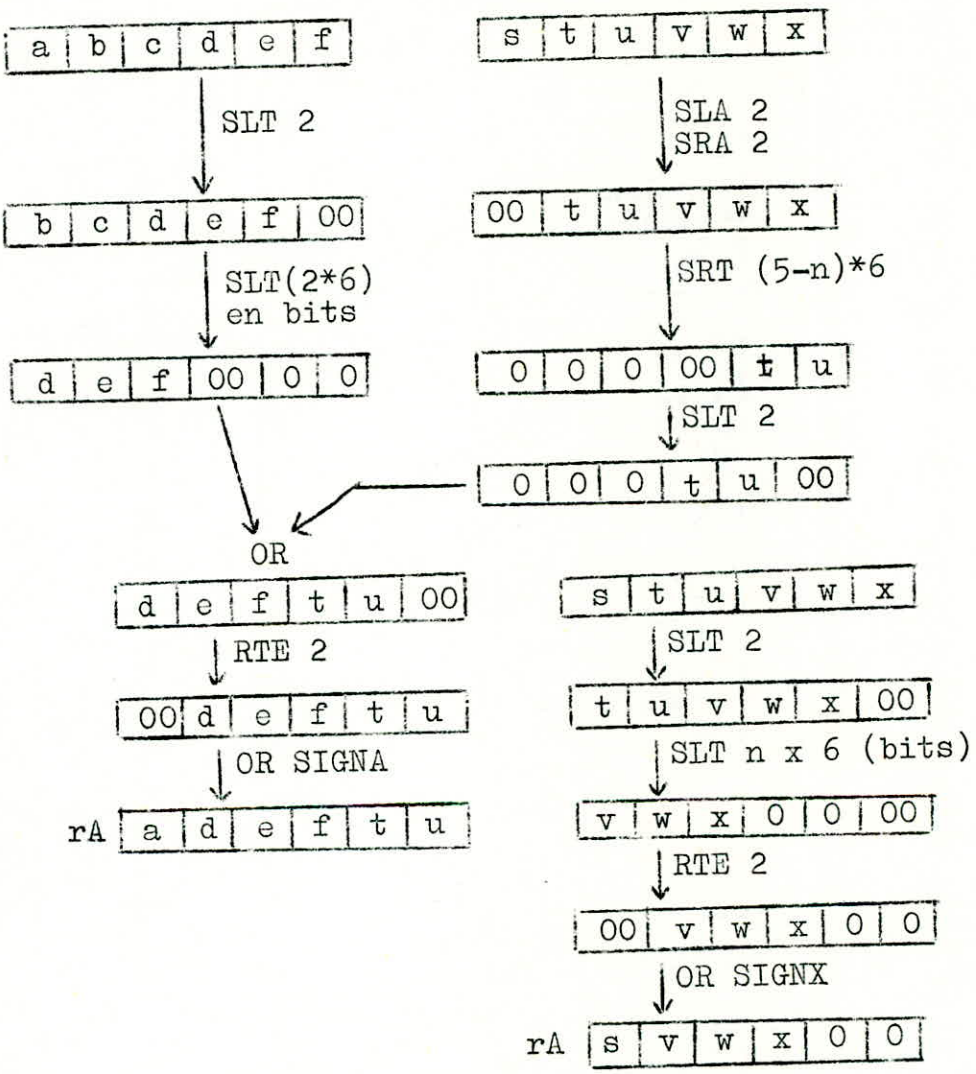
1°) SLAX 2

contenus initiaux :

Résultats :

.../...

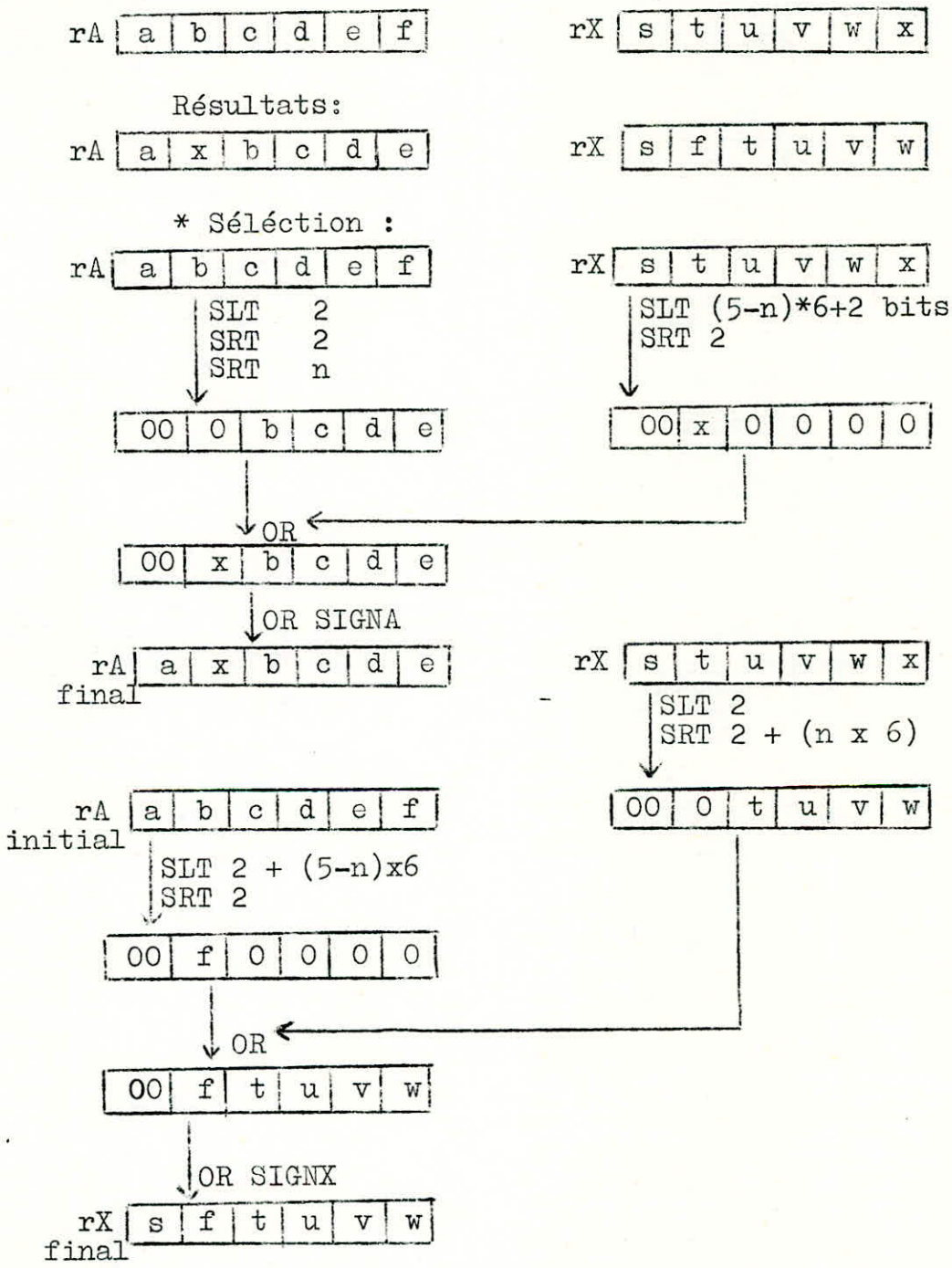
SLAX 2


a b c d e f
SLT 2
b c d e f 00
SLT(2*6) en bits
d e f 00 0 0
s t u v w x
SLA 2
SRA 2
00 t u v w x
SRT (5-n)*6
0 0 0 00 t u
SLT 2
0 0 0 t u 00
OR
d e f t u 00
RTE 2
00 d e f t u
OR SIGNA
rA a d e f t u
s t u v w x
SLT 2
t u v w x 00
SLT n x 6 (bits)
v w x 0 0 00
RTE 2
00 v w x 0 0
OR SIGNX
rA s v w x 0 0

.../...

2°) SRC 501 ≡ SRC 1

* Contenus initiaux :



.../...

IV. - L'opérateur NOP - C = 0

Aucune opération n'arrive. O_n passe à l'instruction suivante :

F et M sont ignorés.

V. - L'opérateur HLT . C = 5 . F = 2.

D'aprés la version MIX, il y a arrêt physique de la machine. Si l'opérateur appuie sur " départ programme ", son effet est celui d'un NOP.

Dans la simulation, cette version n'a pas été suivie. La rencontre de cet opérateur signifie la fin d'exécution du programme en cours et le contrôle est rendu au superviseur.

.../...

TABLE DES CODES MIX

00	1	01	2	02	2	03	10	04	12	05	1	06	2	07	1 + 2F
No operation NOP(0)		rA ← rA + V ADD(0:5) FADD(6)		rA ← rA − V SUB(0:5) FSUB(6)		rAX ← rA × V MUL(0:5) FMUL(6)		rA ← rAX/V rX ← remainder DIV(0:5) FDIV(6)		Special NUM(0) CHAR(1) HLT(2)		Shift M bytes SLA(0) SRA(1) SLAX(2) SRAX(3) SLC(4) SRC(5)		Move F words from M to rI1 MOVE(1)	
08	2	09	2	10	2	11	2	12	2	13	2	14	2	15	2
rA ← V LDA(0:5)		rI1 ← V LD1(0:5)		rI2 ← V LD2(0:5)		rI3 ← V LD3(0:5)		rI4 ← V LD4(0:5)		rI5 ← V LD5(0:5)		rI6 ← V LD6(0:5)		rX ← V LDX(0:5)	
16	2	17	2	18	2	19	2	20	2	21	2	22	2	23	2
rA ← −V LDAN(0:5)		rI1 ← −V LD1N(0:5)		rI2 ← −V LD2N(0:5)		rI3 ← −V LD3N(0:5)		rI4 ← −V LD4N(0:5)		rI5 ← −V LD5N(0:5)		rI6 ← −V LD6N(0:5)		rX ← −V LDXN(0:5)	
24	2	25	2	26	2	27	2	28	2	29	2	30	2	31	2
F(M) ← rA STA(0:5)		F(M) ← rI1 ST1(0:5)		F(M) ← rI2 ST2(0:5)		F(M) ← rI3 ST3(0:5)		F(M) ← rI4 ST4(0:5)		F(M) ← rI5 ST5(0:5)		F(M) ← rI6 ST6(0:5)		F(M) ← rX STX(0:5)	
32	2	33	2	34	1	35	1 + T	36	1 + T	37	1 + T	38	1	39	1
F(M) ← rJ STJ(0:2)		F(M) ← 0 STZ(0:5)		Unit F busy? JBUS(0)		Control, unit F IOC(0)		Input, unit F IN(0)		Output, unit F OUT(0)		Unit F ready? JRED(0)		Jumps JMP(0) JSJ(1) JOV(2) JNOV(3) also [*] below	
40	1	41	1	42	1	43	1	44	1	45	1	46	1	47	1
rA:0, jump JA[+]		rI1:0, jump J1[+]		rI2:0, jump J2[+]		rI3:0, jump J3[+]		rI4:0, jump J4[+]		rI5:0, jump J5[+]		rI6:0, jump J6[+]		rX:0, jump JX[+]	
48	1	49	1	50	1	51	1	52	1	53	1	54	1	55	1
rA ← [rA]? ± M INCA(0) DECA(1) ENTA(2) ENNA(3)		rI1 ← [rI1]? ± M INC1(0) DEC1(1) ENT1(2) ENN1(3)		rI2 ← [rI2]? ± M INC2(0) DEC2(1) ENT2(2) ENN2(3)		rI3 ← [rI3]? ± M INC3(0) DEC3(1) ENT3(2) ENN3(3)		rI4 ← [rI4]? ± M INC4(0) DEC4(1) ENT4(2) ENN4(3)		rI5 ← [rI5]? ± M INC5(0) DEC5(1) ENT5(2) ENN5(3)		rI6 ← [rI6]? ± M INC6(0) DEC6(1) ENT6(2) ENN6(3)		rX ← [rX]? ± M INCX(0) DECX(1) ENTX(2) ENNX(3)	
56	2	57	2	58	2	59	2	60	2	61	2	62	2	63	2
rA(F):V → CI CMPA(0:5) FCMP(6)		rI1(F):V → CI CMP1(0:5)		rI2(F):V → CI CMP2(0:5)		rI3(F):V → CI CMP3(0:5)		rI4(F):V → CI CMP4(0:5)		rI5(F):V → CI CMP5(0:5)		rI6(F):V → CI CMP6(0:5)		rX(F):V → CI CMPX(0:5)	

General form:

C	t
Description OP(F)	

C = operation code, (5:5) field of instruction
F = op variant, (4:4) field of instruction
M = address of instruction after indexing
V = F(M) = contents of F field of location M
OP = symbolic name for operation
(F) = standard F setting
t = execution time; T = interlock time

rA = register A
rX = register X
rAX = registers AX as one
rIi = index reg. i, $1 \leq i \leq 6$
rJ = register J
CI = comparison indicator

[*]:		[+]:	
JL(4)	<	N(0)	
JE(5)	=	Z(1)	
JG(6)	>	P(2)	
JGE(7)	≥	NN(3)	
JNE(8)	≠	NZ(4)	
JLE(9)	≤	NP(5)	

C H A P I T R E II

LIAISON DES DIFFERENTES SECTIONS

L'orientation, vers les modules décrits, est faite dans le sous-programme SIMUL (qu'on peut considérer comme le programme principal de la partie simulateur).

Avant que SIMUL n'appelle un des modules il y a un travail préparatoire qui doit se faire. A l'entrée on initialise les indicateurs LEG et TOG, ceci est fait une fois seulement; mais il y a aussi un travail qui se fait dans SIMUL pour chaque instruction. Cette préparation apparait dans le traitement de SIMUL (voir plus loin).

En se référant au traitement de SIMUL on voit qu'on boucle dans SIMUL (sachant que le retour d'un sous-programme se fait au programme appelant).

La sortie se fait du module C05 (C = 5 le seul qui ne revient pas toujours au programme qui l'appelle) si F = 2.

Pour C = 5 et F = 2 l'instruction est un HLT qui signifie pour le simulateur MIX la fin de l'exécution.

A partir de ces constatations on peut affirmer:

> Dans un programme MIX il doit y avoir <u>toujours</u> l'instruction HLT 0 , qui indique la fin de l'exécution; sinon le programme ne s'arretera pas.

Parmi les modules appelés par SIMUL il y a 2 (C0107 et C3239) qui ne sont pas des modules directement exécutant. Ils permettent seulement d'affiner l'orientation vers l'un des modules déjà décrits. Ces modules " rond-point" ont été nécessaires pour $1 \leqslant C \leqslant 7$ et $32 \leqslant C \leqslant 39$; un regroupement ne parait pas judicieux du fait que les instructions correspondantes sont hétérogénes (exemple : pour C = 5 conversion ou sortie et C = 6 décalage alors que pour $8 \leqslant C \leqslant 23$ chargement).

Pour l'utilisateur ils ne présentent aucun intéret, c'est des modules internes.

Leur programmation sera présentée dans les pages qui suivent car il n'y a rien d'autre à signaler à leur sujet.

.../...

SIMUL

- Mettre l'adresse d'exécution dans le IAR
- Mettre TOG et LEG à zéro
- Traitement INSTRUCTION

Traitement Instruction

- Traitement bombe
- IAR = IAR + 1
- Traitement EXECUT

Traitement Bombe

- Eclater l'instruction dont l'adresse est dans le IAR
- IF (I=0) THEN RAM=AA
  ELSE RAM=AA+RI(I)

END

Traitement EXECUT

```
IF C = 0
   THEN traitement Instruction
   ELSE QUOT=partie entière
                de C/8
        CASE QUOT OF
          0:traitement C0107
            traitement instruction
          1 traitement C0823
          2 traitement instruction
          3:traitement SPST
            traitement instruction
          4:traitement C3239
            traitement instruction
          5: traitement S4047
             traitement instructior
          6: traitement C4855
             traitement instructior
          7:traitement C5663
            traitement instruction
        ESAC
END
```

.../...

MODULE C0107

Il permet de se brancher à un des modules décrits déjà dans le cas où $1 \leqslant C \leqslant 7$

Traitement de C0107

```
CASE ⩽C⩽ OF
        1-2 : ADSOU
        3-4 : MULDI
          5 : C05
          6 : SHIFM
          7 : C07
ESAC
```

MODULE C3239

Identique à C0107 sauf que $32 \leqslant C \leqslant 39$

Traitement de C3239

```
RES = reste de ( C/8 )
CASE  RES   OF
        0-1 : SPST
          2 : C34
          3 : C35
        4-5 : INOUT
          6 : C38
          7 : SAU39
ESAC
```

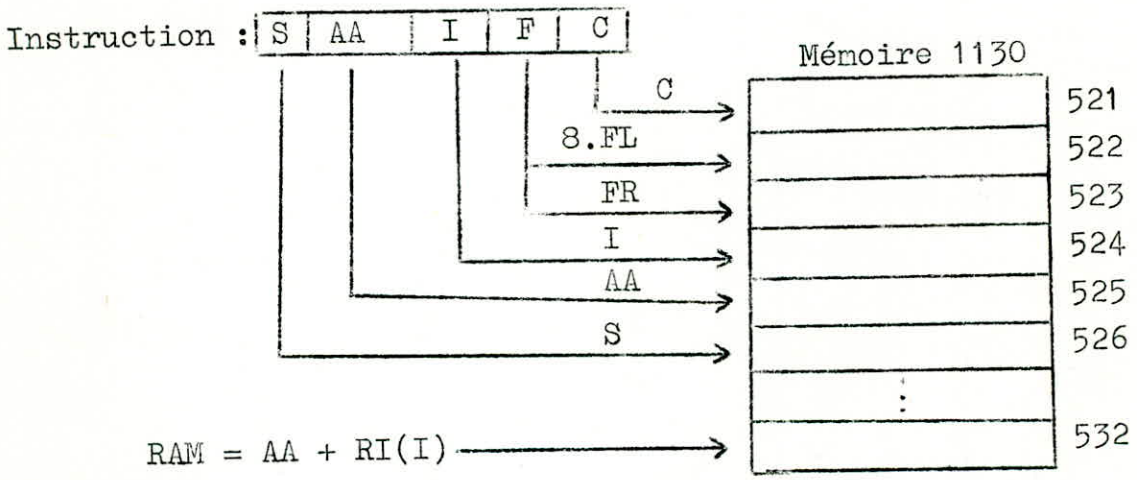
.../...

SOUS-PROGRAMME BOMBE

Il éclate l'instruction MIX dont l'adresse est dans le IAR (adresse absolue 1130 = 527) en ses différents bytes.

Bombe met dans les mémoires réservées à C (code opération), FL (quotient de F par 8), FR (reste de la division de F par 8), I (numéro du registre index), S (le signe de l'instruction), AA (l'adresse) et RAM (Registre adresse-mémoire) leurs valeurs respectives.

Celà peut être résumé par le schéma suivant:



Exemple : Supposons que IAR = 1000

et qu'à l'adresse-MIX 1000 on ait :

0	3001	2	45	32

et que rI2 contienne :

0	145

alors BOMBE mettra dans :

C la valeur 32
FL la valeur 5
FR la valeur 5
I la valeur 2
AA la valeur 3001
RAM la valeur 3001 + 145 = 3146

.../...

CONCLUSION

==*=*=*=*=*=*=*=*=*=*=*

Pour un éventuel utilisateur de MIX, il est demandé de donner le programme écrit en MIXAL (langage MIX) de la façon suivante :

```
// JOB
// XEQ MIX
    programme source en MIXAL
```

1°) Le programme assembleur est stocké sur disque sous le nom de MIX. Ainsi la rencontre de la carte :

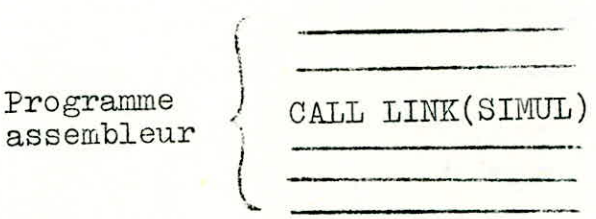
// XEQ MIX

provoque le chargement du programme assembleur MIX en mémoire et son exécution.

2°) Le chargement du programme simulateur se fait grâce à l'instruction :

CALL LINK (SIMUL)

que le système rencontrera dans le programme assembleur. Cette instruction à été necessaire pour réaliser le lien entre assembleur et simulateur, faute de superviseur.



Cette instruction permet de donner le contrôle au simulateur c'est-à-dire de ramener en mémoire

.../...

ce dernier à la place de l'assembleur (dans le cas où il n'y a pas d'erreurs de compilation) et surtout de sauvegarder le produit de l'assembleur placé en zone COMMON.

Ce produit est une donnée de travail pour le simulateur.

L'utilisateur rencontrera un inconvenient du fait qu'il ne pourra pas écrire des programmes assez longs (plus de 80 instructions MIX). Cela est dû au au fait du manque de place mémoire.

Une technique permettant de lever cet obstacle est l'utilisation d'une mémoire virtuelle. Il s'agit d'une astuce technique permettant de stocker en mémoire centrale uniquement les parties du programme. necessaires à un moment déterminé du traitement.

Cette mémoire virtuelle ainsi que les extensions prévues (voir annexe1) pourrait faire l'objet d'un projet futur .

A N N E X E 1

==*=*=*=*=*=*=*=*=*

Idées suggérées par D. E KNUTH pour une extension de MIX.

1°) Augmentation de la mémoire et des unités d'E/S de MIX.

2°) Le champ I pourra être utilisé :

- pour indexer le registre J.
- pour une indexation multiple (le champ I pouvant être utilisé pour 2 registres index).
- pour un adressage indirect.

ou pour une combinaison quelconque des utilisations mentionnées plus haut.

3°) rIi et rJ peuvent être étendues à 5 bytes. Cela signifie des locations (à valeur très élevée des adresses) auxquelles on ne pourra se référer que par indexation; mais ceci peut être comptatible avec l'idée 2°) d'indexation multiple.

4°) Une capacité d'interruption peut être ajoutée (quand certaines conditions arrivent, les registres sont stockés dans des mémoires particulières et un contrôle est donné à un programme qui plus trad restaure les registres et retourne au programme appelant par utilisation d'un nouveau code opération)

5°) Des opérations logiques peuvent être ajoutées dans la version binaire de MIX.

6°) Des shifts binaires peuvent être prévus.

ANNEXE 2

-=-=-=-=-=-=-=-=-=-=

PROGRAMME 1

Lecture de 2 cartes et leur impression sur pupitre console et sur imprimante.

```
// JOB
// XEQ MIX
LILI    EQU  7
        ORIG 3955
TOTO    ENT2 2          mettre 2 dans rI2
        STZ  DEB+17     mettre à zéro la mémoire DEB+17
* Mise à zéro de 7 mots consécutifs par l'instruction
* MOVE avec adresse de départ=DEB+17 et d'arrivée
* DEB+18. Nombre de mots à transporter=LILI=7.
        ENT1 DEB+18
        MOVE DEB+17(LILI)
* Lecture d'une carte.
RET     IN   DEB+1(0 2)
* son impression sur console
        OUT  DEB-1(0 1)
A       JBUS A(0 1)     test si impression finie
* son impression sur imprimante.
        OUT  DEB-1(0 3)
B       JBUS B(0 3)
* décrémentation de rI2 de 1 et test si contenu
* de rI2 positif. Si oui alors carte suivante
* sinon fin.
        DEC2 1
        J2P  RET
        HLT  0          Sortie
* Définition de caractères
        ALF  MIX M
DEB     ALF  ARCHE
```

.../...

```
* Définition de la zone d'impression.
        ORIG *+22
        END  TOTO
```

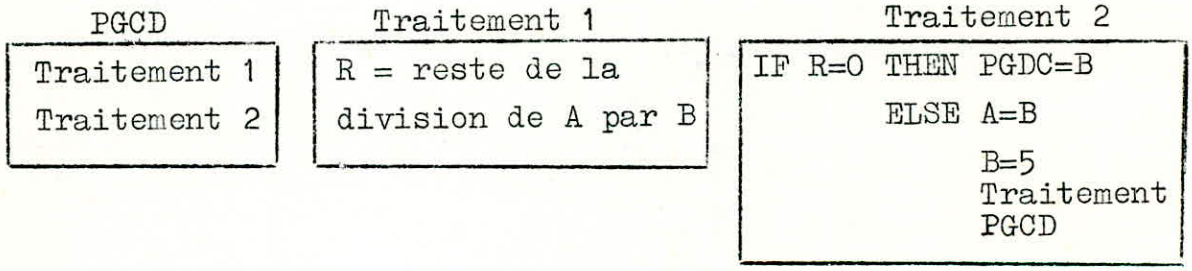
Puis les 2 cartes contenant le message

```
** MONSIEUR LE PRESIDENT
** MESSIEURS LES MEMBRES DU JURY . BONJOUR
```

PROGRAMME 2

Programme recherchant le PGCD de 2 nombres A et B et imrimant : " LE PGCD DES NOMBRES DONNES A ET B est __"

Algorithme de cette recherche:




```
// JOB
// XEQ MIX
        ORIG  3955
* Conversion en code caractère de A et B pour impression
DEB     LDA   A
        CHAR  0
        STX   AA
        LDA   B
        CHAR  0
        STX   BB
*Division de A par B
        LDA   A
```

.../...

```
        SRAX  5
        DIV   B
        SLAX  5
        JAZ   FIN
* Reste non nul - alors permutation
        STA   C
        LDA   B
        STA   A
        LDA   C
        STA   B
        JSJ   DEB
* Reste=0-pgcd dans B. Conversion en code caractère.
FIN     LDA   B
        CHAR  0
        STX   B
        OUT   MES(0 3)
AC      JBUS  AC(0 3)
        HLT   0
MES     ALF   LE PG
        ALF   CD DE
        ALF   S DEU
        ALF   X NOM
        ALF   BRES
        ALF
AA      CON   0
        ALF
        ALF     ET
        ALF
BB      CON   0
        ALF
        ALF    EST
        ALF
B       CON   1504
        ALF
```

.../...

```
CON  0
CON  0
CON  0
CON  0
CON  0
CON  0
CON  0
CON  0
END  DEB
```

PROGRAMME 3

Ce programme range 12 nombres par ordre croissant. I_1 fait appel à 2 sous-programmes MAXIMUM et SORTI.

L'agorithme est le suivant : trouver le maximum des 12 nombres, le permuter avec celui se trouvant au bas de la table. Trouver ensuite le maximum des 11 premiers, le permuter avec l'avant dernier de la table et ainsi de suite...

MAXIMUM trouve le maximum des n premiers nombres d'une table X, (n est supposé dans rI1), Le maximum est donné dans rA et le rang du maximum dans la table est dans rI2.

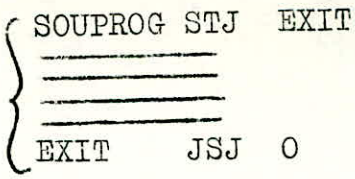
SORTI imprime une table de 12 nombres sur imprimante.

Remarque concernant les sous-programmes.

Lorsqu'on fait appel à un sous-programme, on doit sauvegarder l'adresse de l'instruction qui suit celle de l'appel (s'il n'y a pas de paramètres) pour pouvoir retourner au programme appelant.

Cela se fait dans le programme principal par JMP SOUPROG et dans le sous-programme par:

.../...



Donc tout sous-programme a la forme donnée ci-dessus.

* Programme de rangement de 12 nombres

```
         ORIG  3955
DEB      JMP   SORTI
         ENT1  12
RET      JMP   MAXIMUM
         STA   MAX
         LDA   X-1,1
         STA   X-1,2
         LDA   MAX
         STA   X-1,1
         JMP   SORTI
         DEC1  1
         CMP1  UN
         JG    RET
         HLT   0
```

* Sous-programme de recherche du maximum
* rI1 contient le nombre d'éléments.
* Le maximum sera dans rA.

```
MAXIMUM  STJ   EXIT
INIT     ENT3  0,1
         JMP   CHANGEM
LOOP     CMPA  X-1,3
         JGE   *+3
CHANGEM  ENT2  0,3
         LDA   X_1,3
         DEC3  1
         J3P   LOOP
EXIT     JMP   *
```

.../...

```
* définition de la table X et des constantes

UN      CON   1
X       CON   9000
        CON   6199
        CON   2011
        CON   7800
        CON   1240
        CON   3005
        CON   2000
        CON   1973
        CON   4032
        CON   1849
        CON   6199

* Réservation de zones

MAX     CON   0
Y       CON   0
        ORIG  *+23

* Sous-programme d'impression

SORTI   STJ   FIN
        ENT2  -12
        ENN1  24
BCL     LDA   X+12,2
        CHAR  0
        STX   Y+24,1
        STZ   Y+25,1
        DEC1  -2
        DEC2  -1
        J2N   BCL
        OUT   X(0 3)
A       JBUS  A(0 3)
FIN     JMP   *
        END   DEB
```

.../...

PROGRAMME 4

Programme qui ordonne 24 nombres par ordre croissant par la méthode du bubble-sort.

```
* Mise à zéro de la zone d'impression
F       EQU   23
DEB     ENT1  TAB+41
        STZ   TAB+40
        MOVE  TAB+40(F)
* Lecture de 3 cartes contenant 24 nombres
* et impression de ces nombres à raison d'un par ligne.
        ENT3  3
        ENT4  0
AC      IN    TAB(2)
        ENT2  -16
AB      LDA   TAB+16,2
        STA   TAB+40+11
        LDX   TAB+17,2
        STX   TAB+40+12
        OUT   TAB+40(0 3)
A       JBUS  A(0 3)
        NUM   0
        STA   TAB+16,4
        INC2  2
        DEC4  -1
        J2N   AB
        DEC3  1
        J3P   AC
```

.../..

```
* Triage des nombres
          ENT1  24
HA        ST1   BO(1 2)
          ENT2  1
          ENT1  0
          JMP   BO
HB        LDA   TAB+15,2
          CMPA  TAB+16,2
          JLE   HD
          LDX   TAB+16,2
          STX   TAB+15,2
          STA   TAB+16,2
          ENT1  0,2
HD        INC2  1
BO        ENTX  -BO,2
          JXN   HB
          J1P   HA
* Impression des 24 nombres: 1 nombre par ligne
* ordonné par ordre croissant.
AD        ENT6  -24
          LDA   TAB+39+24,6
          CHAR  0
          STA   TAB+40+11
          STX   TAB+40+12
          OUT   TAB+40(0 3)
B         JBUS  B(0 3)
          IOC   13(3)
          DEC6  -1
          J6N   AD
TAB       ORIG  *+64
          END   DEB
```

BIBLIOGRAPHIE

^^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*

- DONALD E. KNUTH Stanford University
" THE ART OF COMPUTER PROGRAMMING "
ADDISON-WESLEY PUBLISHING COMPANY (1973)

- M. ADIBA
Manuel de cours d'Informatique (ENPA)

- O. LECARNE
" SYSTEMES D'EXPLOITATION "
Cours dispensé à Grenoble

- HOPGOOD
" TECHNIQUES DE COMPILATION "

- Documentation I B M