INSTITUT NATIONAL POLYTECHNIQUE
DE LORRAINE

THESE

pour obtenir le diplome de docteur-ingénieur
spécialité informatique

COMPILATEUR DU LANGAGE PASCAL POUR MINIORDINATEURS
REALISATION SUR SOLAR 16

par

Alain TISSERANT

Soutenue le 16 Décembre 1977

Jury : Président C.PAIR
Examinateurs D.COULON
J.C.DERNIAME
M.GRIFFITHS
S.KRAKOWIAK

INSTITUT NATIONAL POLYTECHNIQUE
DE LORRAINE

THESE

pour obtenir le diplome de docteur-ingénieur

spécialité informatique

COMPILATEUR DU LANGAGE PASCAL POUR MINIORDINATEURS
REALISATION SUR SOLAR 16

par

Alain TISSERANT

Soutenue le 16 Décembre 1977

Jury : Président C.PAIR
Examinateurs D.COULON
J.C.DERNIAME
M.GRIFFITHS
S.KRAKOWIAK

THESE DE DOCTEUR-INGENIEUR

A.TISSERANT

COMPILATEUR DU LANGAGE PASCAL POUR MINIORDINATEURS.

REALISATION SUR SOLAR 16.

Résumé : Après avoir rappelé les techniques adaptées à l'implantation de compilateurs du langage Pascal, ce travail propose une méthode de transport pour miniordinateurs.
Le compilateur utilise un langage intermédiaire, Babel, reflétant les caractéristiques communes des miniordinateurs usuels. Un moniteur d'exécution gère une mémoire virtuelle segmentée, levant ainsi les restrictions de taille des programmes. Une première réalisation est décrite pour Solar 16.

CRIN 77-R-046 Décembre 1977

REMERCIEMENTS

Je tiens à exprimer ma gratitude à

Monsieur PAIR, professeur à l'INPL, qui m'a permis de faire cette thèse, m'a conseillé et guidé tout au long de ce projet. Je tiens à le remercier pour toutes les facilités de travail qu'il m'a accordées.

Monsieur COULON, Maître de Conférences à l'Ecole des Mines de Nancy, pour les conseils reçus, et qui m'a permis de travailler dans d'excellentes conditions.

Messieurs KRAKOWIAK, Professeur à l'Université de Grenoble, DERNIAME, Maître de Conférences à Nancy I, GRIFFITHS, Professeur à l'IUT de Nancy, qui se sont intéressés à ce travail, et ont bien voulu le juger.

Messieurs BASSELIN, DESCHAMP, GODEFROY, LECAS, MERCIER et Madame SCHWAAB qui ont contribué à la réalisation effective de ce projet.

Madame BEAURAIN qui a décrypté le manuscrit, et en a remarquablement assuré la frappe et la mise en page.

L'IRIA, qui, par une contribution financière du SESORI, a permis au travail de s'effectuer dans de bonnes conditions.

Et aussi toutes les personnes non citées qui ont contribué à la réalisation technique et matérielle de cette thèse.

SOMMAIRE

introduction

Parallèlement à l'apparition des miniordinateurs et à l'évolution de leur architecture ces dernières années, de nouveaux langages de programmation ont été définis, mettant en application les principes de la programmation structurée [DHD72,Inf72,Inf76]. Dans les deux cas, l'évolution s'est réalisée vers des structures de programmes imposées aux programmeurs :

- pour les langages [Dij73,Wir73b,Wir76b],notons la programmation par raffinements successifs [Wir71c, Wir73a], l'usage intensif de procédures, les structures de données assez éloignées de la machine, les instructions puissantes rendant inutile l'usage de l'instruction *aller à* [Dij68],... Cette évolution s'est donc faite essentiellement vers la facilité d'écriture et la lisibilité des programmes, conduisant à la fiabilité.

- pour les machines (§ I B et III C), citons les instructions d'appel de sous-programmes permettant directement la récursivité, l'adressage basé limitant l'accès aux données, les protections entre sous-programmes ou tâches, la puissance de certaines instructions remplaçant des séquence de code répétitives, et certains langages d'assemblage qui sont réellement des langages évolués [For75]. Cette évolution a donc eu comme objectifs la fiabilité des programmes, et leur performance.

Nous nous proposons d'utiliser l'évolution commune de ces deux voies de recherche pour étudier les problèmes d'implantation de langage sur miniordinateur, problèmes tant de compilation (§II B) que de système d'exploitation (§ II A et II C), et en particulier examiner la représentation des structures d'information et de contrôle (§II B), les problèmes de gestion de la mémoire (§ II A), et l'adéquation des instructions de la machine et des modes d'adressage.

Nous nous proposons aussi d'illustrer et valider cette recherche par la mise en oeuvre d'un langage issu des idées de la programmation structurée, PASCAL [Wir71a, Wir72,Wir75,HoW73,JeW74,CGZ76] (§ I A 1 et I A 2), sur un miniordinateur récent, sophistiqué et largement répandu, SEMS SOLAR 16 [Tel75] (§ I B).

A notre connaissance, le parallélisme d'évolution que nous constatons entre langages évolués et architectures de miniordinateurs n'a pas encore été utilisé pour la réalisation d'une implantation complète de Pascal sur ce type de machines.

Voulant que cette expérience débouche sur un produir réellement utilisable (§ I C), en particulier pour des applications d'enseignement [Lec74], nous nous attacherons

à réaliser un système original de compilation, complet et cohérent, ne nécessitant pas la connection constante à une grosse machine, acceptant le langage Pascal standard sans restrictions de nature à en limiter l'usage, restrictions tant sur le langage que sur la taille des programmes, et fournissant des outils évolués de mise au point (§ I C).

Désirant pouvoir réutiliser cette expérience pour d'autres miniordinateurs, nous essayerons de définir un compilateur adaptable à toute une classe de machines [Lec75,NaJ74] (§ III C); l'aspect "transport" en conditionnera une partie des spécifications techniques.

I - spécification

du problème

A - LE LANGAGE

"The desire for a new language for the purpose of teaching programming is due to my deep dissatisfaction with the presently used major languages whose features and constructs too often cannot be explained logically and convincingly and which too often represent an insult to minds trained in systematic reasoning."

N. Wirth

1 - HISTORIQUE

En 1965, Niklaus Wirth proposa un successeur au langage Algol 60 [Nau63], et qui en était essentiellement une extension. Cette proposition, qui fut rejetée par l'IFIP WG 2.1 en faveur de ce qui devint Algol 68, fut développée sous le nom d'Algol W.

Pascal n'est pas une extension d'Algol W, mais en est le successeur, défini à partir d'idées de Hoare sur la structuration des données[DHD72] et d'idées de Wirth sur la méthodologie de la programmation [Wir71c et Wir73a, Wir73b, Wir76b] . Son développement commencé en 1968 aboutit à une première version [Wir71a] implémentée à Zürich sur CDC 6500 en 1970 [Wir71b]. Par la suite le langage fut modifié (essentiellement au niveau syntaxique) pour aboutir au'Revised Report on the Programming Language Pascal' [Wir72] complété par une définition axiomatique de la sémantique [HOW73], et à de nombreuses implémentations [PUG].

Conception du langage

Les buts initiaux de Wirth furent de développer un langage suffisamment simple et systématiquement cohérent qui permettrait d'exprimer les structures élémentaires d'algorithmes et de données d'une manière claire et naturelle ; le langage devait être suffisamment lisible, concis et puissant pour permettre l'enseignement de la programmation d'une manière scientifique, et pour permettre aussi la construction de programmes complexes, tels qu'un compilateur. En même temps, le langage devait faciliter une compilation efficace sur les ordinateurs existants, sans que sa définition ne fasse référence à une machine particulière.

Ce conflit apparent entre indépendance de la machine et implémentation efficace fut résolu en basant le langage Pascal sur un ensemble de possibilités élémentaires de structuration universellement applicables et librement combinables pour les instructions et les données. Les propriétés dépendantes de la machine sont introduites en définissant un ensemble de types de données de base adaptés. Par exemple le type standard entier introduit l'ensemble fini ordonné des nombres entiers utilisables sur une machine donnée. Les possibilités de structuration de données, et leur combinabilité, permettent une bonne économie d'espace mémoire.

Notons enfin que la notion de type de donnée en Pascal recouvre les notions mathématiques de "type" (la façon d'interpréter l'arrangement des bits), "d'ensemble des valeurs" (l'ensemble des valeurs que peut prendre une variable d'un type donné) et de "structure" (un modèle pour ranger les données en mémoire).

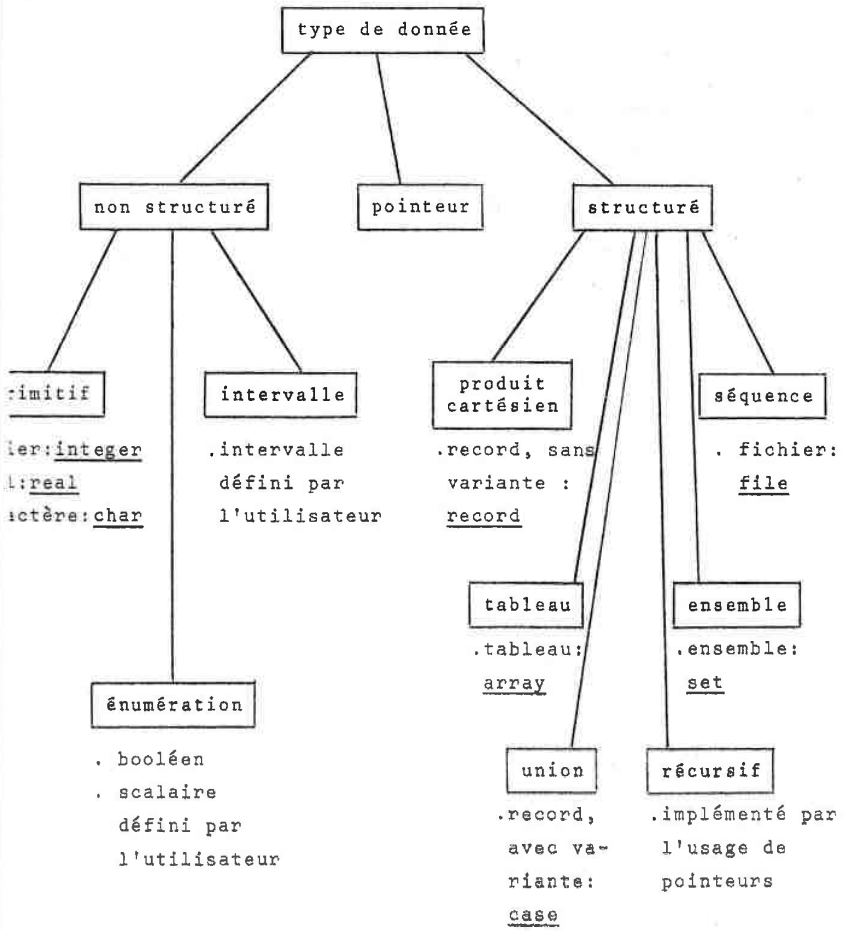
2 - DESCRIPTION DU LANGAGE

Cette brève description ne vise qu'à poser des points du langage importants pour la compréhension de la suite du texte. Le lecteur est prié de se reporter à [JeW74,CGZ76] pour une première approche du langage, à [Wir72] pour une description de sa syntaxe, et à [HoW73] pour une description de sa sémantique.

a) LES DONNEES ET LEUR STRUCTURATION

En Pascal, à chaque variable est attaché un type qui définit l'ensemble de ses valeurs possibles.

La hiérarchie des types de Pascal peut être représentée par le diagramme suivant, inspiré d'une terminologie introduite par Hoare [DHD72] :



rimitif :

es types prédéfinis ont leurs valeurs notées par des nombres et es caractères entre apostrophe, notations syntaxiquement différentes des identificateurs. Leurs ensembles de valeurs dépendent à l'implémentation : les entiers et les réels sont souvent eprésentés sur un mot, les caractères étant ceux disponibles ur l'installation, ou le Display Code de CDC. On trouvera en annexe

Enumération :

La définition d'un type scalaire lui associe un ensemble ordonné de valeurs, représentées par des identificateurs.

Intervalle :

Un intervalle est défini à partir d'un type scalaire en en indiquant la plus petite et la plus grande valeur. Les opérations possibles sur une variable de type intervalle sont celles définies sur le type de départ.

Pointeur :

Les variables déclarées dans des déclarations explicites sont statiques : la déclaration associe à la variable un identificateur qui permet ensuite de la référencer. Des variables peuvent aussi être générées par une instruction, lors de l'exécution du programme: elles sont dynamiques. On peut alors les référencer par l'intermédiaire d'un pointeur, qui se substitue à un identificateur explicite. Un pointeur ne peut référencer que des variables d'un même type T unique, ou aussi ne référencer aucune variable (valeur nil).

Produit cartésien :

Dans une structure de record les composants (appelés champs) ne sont pas nécessairement tous du même type (au contraire d'un tableau).

fin que le type des champs accédés soit évident dès la compilation,
n sélecteur référençant un champ d'un record n'est pas une valeur
alculable, mais un identificateur, déclaré lors de la définition
u record. Ainsi un record est une structure à accès direct.

zbleau :

ci tous les composants sont de même type. Un sélecteur de compo-
ant est un indice calculable, dont le type est donné à la définition
ı tableau, et doit être scalaire. Chaque variable tableau peut être
onsidérée comme une application du type de l'indice sur le type
es composants. C'est une structure à accès direct, comme les
ecord.

ion :

'union explicite de types n'est pas licite en Pascal, cependant
ı type record peut être défini comme constitué de plusieurs
ariantes. Ce qui implique que différentes variables, bien que
ı même type, peuvent avoir une structure différante : cette
lfférence de structure peut être une différence sur le nombre
ı le type des champs. La variante choisie pour la valeur effective
'une variable record est indiqué par un champ commun à toutes lss
ariantes.

Récursif :

Ce type est implanté en Pascal par l'intermédiaire de pointeurs, puisque des variables pointeur peuvent apparaître comme composants de variables structurées, elles-mêmes étant générées dynamiquement.

Ensemble :

Une structure d'ensemble définit l'ensemble des parties d'un type scalaire de base.

Séquence :

Une structure de fichier est une séquence d'éléments de même type. Un ordre est défini naturellement sur les éléments par la séquence. A chaque instant, un seul des éléments est directement accessible ; les autres composants sont rendus accessibles en progressant séquentiellement dans le fichier. Comme un fichier est généré en lui concaténant séquentiellement des éléments à sa fin, la taille d'un fichier n'est pas définie à sa déclaration.

Les données sont représentées en Pascal par les valeurs des variables. Toute variable utilisée dans un programme doit au préalable être introduite par une déclaration de variable qui lui associe un identificateur et un type. Ce type peut être décrit explicitement dans la déclaration de variable, ou peut être référencé par un identificateur de type, défini dans une déclaration de type explicite

Dans une description de type, les méthodes de structuration sont librement combinables entre elles ('orthogonalité du langage'), moyennant cependant certaines restrictions d'implémentation (telles que les fichiers de fichiers, ou les fichiers de pointeurs).

b) STRUCTURES DE CONTRÔLE

Nous décrivons brièvement la syntaxe des instructions par des diagrammes de Conway [Con63,Wir72].

L'instruction fondamentale est l'affectation, qui transmet à une variable la valeur calculée d'une expression ; variable et expression doivent être de même type (ainsi on ne peut affecter une valeur réelle à une variable entière).

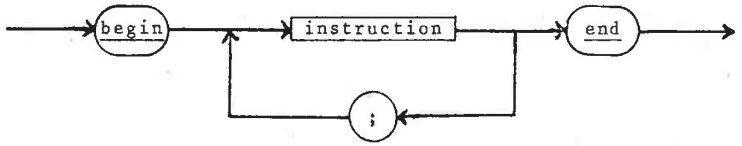
syntaxe :

A partir d'un nombre réduit de structures de contrôle, librement combinables entre elles, Pascal permet l'écriture de programmes lisibles et simples, suivant les "règles" de la programmation structurée.

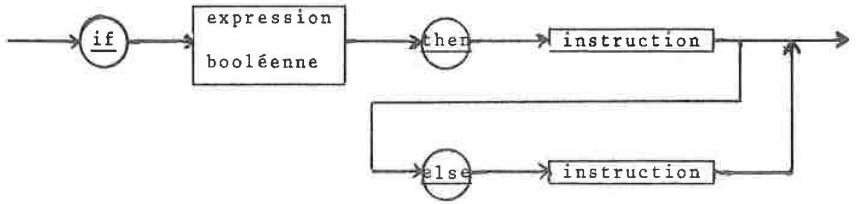
Les principales instructions de contrôle sont les suivantes :

Composée :

Elle permet de réunir une liste d'instructions en une seule instruction : ce n'est pas ici l'instruction de définition de bloc (au contraire de PL/I et Algol 60).

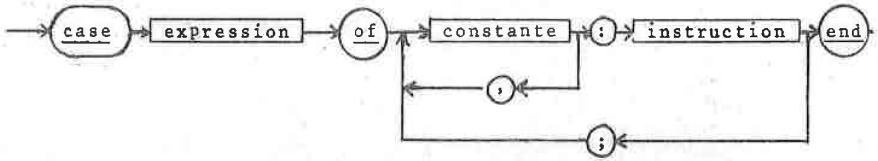


Conditionnelle :



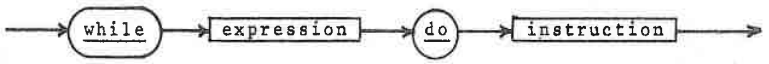
Cas :

C'est une généralisation de l'instruction conditionnelle.

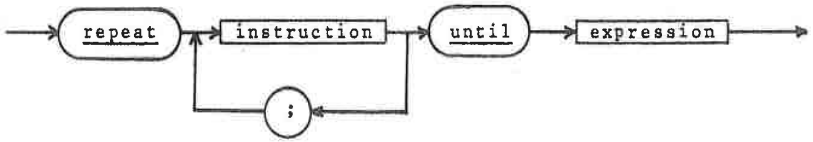


Itérations :

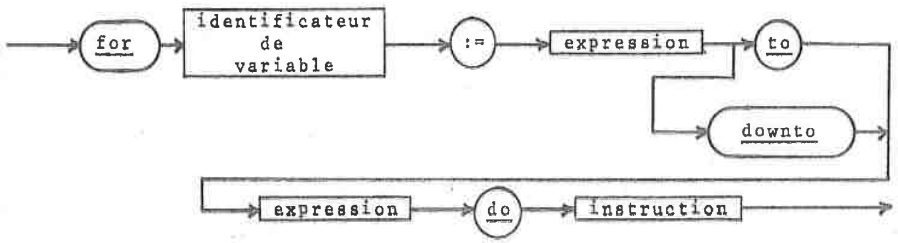
a) si le test d'arrêt doit être effectué avant chaque exécution de l'instruction à répéter : forme tantque...répéter...



b) si le test d'arrêt doit être effectué après chaque exécution de l'instruction à répéter : forme répéter...jusqu'à...



c) si le nombre d'itérations est connu a priori : forme pour i de n à m faire ...



Enfin notons parmi les autres instructions du langage Pascal :

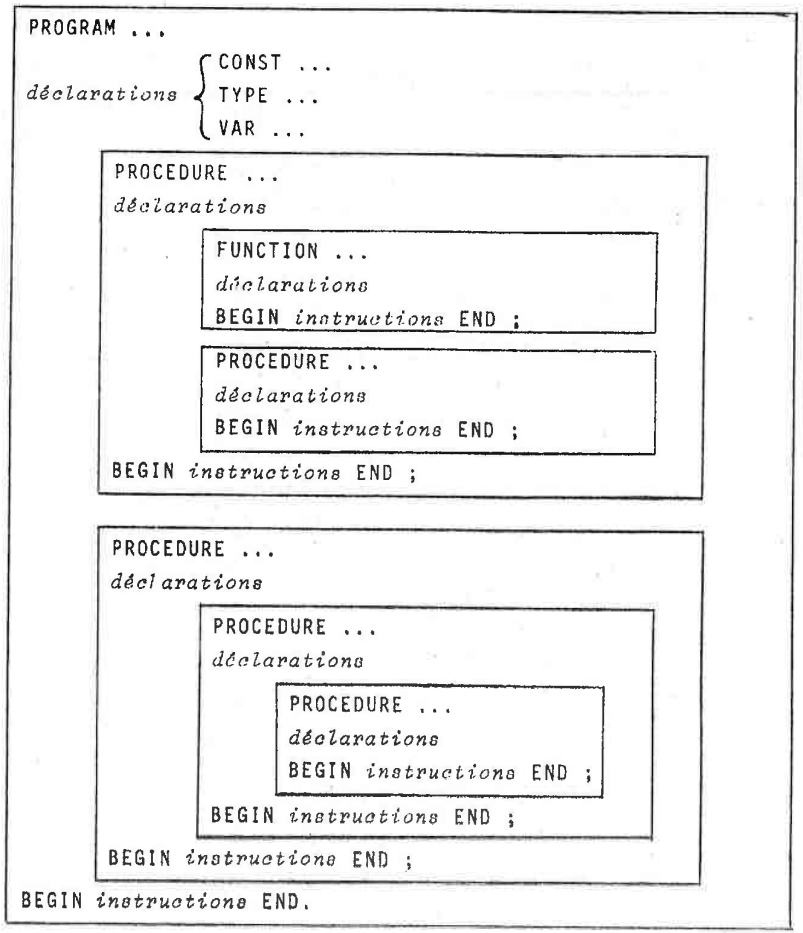
- l'instruction d'appel de procédure ;
- l'instruction de rupture de séquence inconditionnelle, goto, dont la sémantique n'a pas été axiomatisée, de même que pour l'arithmétique des nombres réels...
- les instructions d'entrée-sortie formatée sur les fichiers de

c) LA STRUCTURE DES PROGRAMMES

L'on peut donner un nom (identificateur) à une instruction et la référencer par cet identificateur. L'instruction est alors appelée une procédure, sa déclaration une déclaration de procédure, et sa référence une instruction procédure. Une telle déclaration de procédure peut aussi contenir des déclarations de variables, de types, et d'autres déclarations de procédures. Les objets ainsi déclarés ne peuvent être référencés qu'à l'intérieur de la procédure elle-même : ils lui sont dits locaux. Leurs identificateurs n'ont de signification que dans le texte qui constitue la déclaration de procédure, et qui est appelé la portée de ces identificateurs.

Comme des procédures peuvent être déclarées locales à d'autres procédures, les portées peuvent être imbriquées. Les objets qui sont accessibles à une procédure sans lui être locaux, lui sont dits globaux.

Ainsi la structure générale d'un programme peut s'illustrer par le schéma de la page suivante.



Du fait de la règle de localité d'un identificateur, les procédures Pascal sont des blocs -au sens Algol 60 [Pai72,RaR64]- pour la gestion de la mémoire. Mais la conception du langage, et son adaptation à la programmation structurée, permettent d'envisager un traitement de ces blocs moins général qu'en Algol : les seuls objets globaux accédés dans une procédure étant les paramètres déclarés transmis par référence, et des objets globaux assez peu nombreux. Nous utiliserons cette remarque pour adapter notre système

de compilation au critère mémoire centrale de taille réduite. (§ II A).

Enfin, pour terminer ce survol du langage, nous allons examiner brièvement quelques points litigieux de Pascal, qui nous ont amenés à apporter quelques extensions et à lever certaines restrictions d'implémentation.

d) QUELQUES POINTS CRITIQUES DU LANGAGE

Langage défini, d'abord expérimentalement, par une seule personne, puis plusieurs fois modifié au fil des implémentations sur CDC 6000 à Zürich et des éditions de rapports [Wir71a,Wir72, HoW73,...], Pascal souffre de l'absence d'un standard précis ;ses atouts principaux, faible taille et simplicité, peuvent faire regretter l'absence de certains éléments et notions (cf. PL/I, souvent appelé "langage dinosaure") ; enfin la facilité de modifications permise par un compilateur structuré,court et écrit en Pascal [Amm73], ainsi que sa large diffusion dans les milieux universitaires, ont entraîné nombre d'extensions et modifications au langage [Ten75].

- *Initialisation de variables à la compilation* :

La richesse des structures de données rend cette initialisation difficile à définir (elle est cependant réalisée en LIS [Cii75], très proche de Pascal, et partiellement dans certains dialectes de Pascal). Dans l'attente d'une standardisation nous ne définirons pas d'extension en ce sens.

- *Types scalaires symboliques* :

Pascal ne fournit pas, dans sa version officielle, de possibilités directes d'entrée ou de sortie pour des variables de type scalaire symbolique. Cela semble être là une incohérence du langage, très gênante pour les applications d'enseignement. Nous étendrons donc les instructions d'entrée-sortie aux scalaires symboliques.

- *Tableaux dynamiques* :

Ils sont interdits en Pascal. Assez peu coûteux effectivement en temps d'exécution, ils ajoutent cependant une complexité qui doit être mise en regard des facilités de programmation qu'ils permettent. Pascal permet de tourner assez aisément cette interdiction en fournissant des définitions de constantes et de types très puissants dans la pratique (mis à part le cas des tableaux dynamiques paramètres de procédures).

- *Absence de tableaux à bornes dynamiques en paramètres de procédures* :

Comme les bornes d'un tableau font partie du type de ses

tableaux dont les bornes ne sont pas connues à la compilation. Cette restriction, qui n'est contraignante dans la pratique de la programmation en Pascal que rarement, est un handicap sérieux du langage pour l'utilisation de bibliothèques de programmes. En l'absence de standard, nous n'implanterons pas cette extension.

- *Etiquettes et instruction aller à* :

Pascal ne tranche pas dans la polémique sur le goto [Dij68,DHD72,Inf76] en en autorisant l'emploi, mais dans des limites très contraignantes. En l'absence d'autres structures, goto reste très utile par exemple pour le traitement des erreurs syntaxiques dans un compilateur.

- *Types structurés* :

Un objet structuré peut habituellement être manipulé comme un objet non structuré ; ainsi par exemple un tableau peut être un composant d'un fichier, et l'on peut effectuer en une seule instruction des affectations entre de tels objets. Les diverses implémentations introduisent de nombreuses restrictions à ce sujet (fichiers de fichiers, fichiers d'objets dynamiques...), que nous reprendrons en général.

- *Cardinal maximum des ensembles* :

Chaque implémenteur doit définir ce point du langage. Pour des raisons d'efficacité, un ensemble se représente usuellement sur un mot (CDC 6000) ou un double-mot (IBM 370, CII IRIS 80); les ensembles sont ainsi limités à une soixantaine d'éléments. Ce nombre très faible, qui ne permet même pas l'utilisation du type set of char, ne peut être modifié qu'au prix d'une forte perte de performance à l'exécution, et de difficultés de compilation. Nous avons cependant choisi d'implémenter les ensembles sur un nombre de mots (de 16 bits) variable (jusqu'à 2048).

- *Chaînes de caractères* :

Elles ne font partie du langage que sous la forme de tableaux compactés de caractères : c'est une source de difficultés de programmation, mais aussi de facilités de compilation.

En conclusion, le faible nombre de déficiences constatées dans des articles tels que [Hab73, LeD74, LeD75] , plaide en faveur du langage Pascal, ces déficiences étant de plus compensées par les points très positifs.

Pascal, bien qu'imparfait, et appelant un successeur, reste l'un des langages connaissant une des plus fortes croissances de diffusion et d'utilisation actuellement. S'opposant vivement à PL/I, Algol 68 et APL, il a certainement comblé un vide.

3 - *IMPLEMENTATIONS*

Après avoir examiné rapidement l'histoire des compilateurs Pascal développés à Zürich, nous étudierons les familles de techniques d'implémentation.

a) *HISTORIQUE*

Le travail d'implémentation d'une première version de Pascal commença en 68. Le compilateur fut écrit en Fortran, avec l'intention de le traduire ensuite en Pascal. Le choix de Fortran s'avéra être une erreur grave et N.Wirth décida de recommencer le développement d'un compilateur à partir de zéro, en l'écrivant directement cette fois en Pascal. Son développement, commencé fin 69, conduisit à une version initiale n'acceptant qu'un langage réduit, mais suffisant pour écrire un compilateur (pas d'ensembles, de records compactés, de fonctions, de paramètres procédure ou fonction, d'arithmétique réelle). Le compilateur fut très rapidement traduit "à la main" en un langage de bas niveau, puis les éléments absents y furent successivement introduits [Wir71b].

La méthode de compilation utilisée est une technique de génération de code dirigée par l'analyse de la syntaxe [LeS68, Pai72, Knu71, Gri69, Gri71], la syntaxe du langage Pascal ayant été définie au départ de manière à permettre

une telle compilation en une seule passe (avec lecture d'un symbole à l'avance) [Con63].

Le compilateur Pascal 0 ainsi obtenu a de très bonnes performances (il analyse environ 250 lignes de texte source par seconde sur CDC 6600) et atteint un niveau de lisibilité remarquable pour un programme de cette taille. Générant du code objet absolu, il ne permet pas l'utilisation de bibliothèques de programmes.

Après parution du 'revised report' introduisant quelques modifications de syntaxe, le compilateur fut modifié, et prit le nom de Pascal 1 en 72.

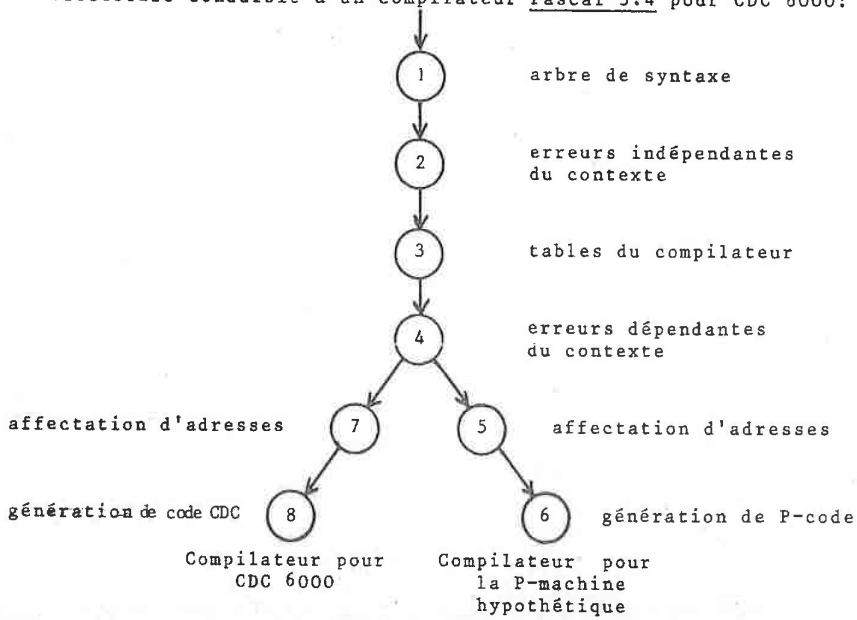
Par la suite un compilateur Pascal 2 fut écrit en 74, toujours à Zürich sur CDC 6000, introduisant quelques modifications du langage (disparition des classes et de la partie valeur), générant du code objet translatable et permettant l'édition de liens avec des sous-programmes Fortran.

Ces compilateurs Pascal furent transportés sur au moins trois machines : sur ICL 1900 à Belfast [WeQ71] et sur CII 10070 à Paris [MaT74] par une réécriture des routines sémantiques directement dans le compilateur et en effectuant le processus d'amorçage sur CDC 6500 ; sur IBM 360 à Stanford [RuS76] en utilisant PL/I pour l'amorçage (technique que les auteurs ne recommandent plus).

En 73, un compilateur portable fut écrit à Zürich par U. Ammann. Développé par étapes, selon les techniques de la programmation structurée, il génère du code objet (P-code [Gau74, NAJ74]) pour une machine hypothétique à piles et à mots de 60 bits. En le recompilant par lui-même, on l'obtient écrit en P-code. Il suffit alors de disposer d'un interpréteur de P-code pour l'exécuter sur toute machine.

Ce compilateur Pascal P subit plusieurs mises à niveau à partir de 74, la dernière version en date étant Pascal P4 (76). Il a été porté sur de nombreuses machines, dont CDC 3600 [FaG75], PDP 11 [BrD76], Dec System 10 [GlN76], CII IRIS 50 [LeC77], etc.

Parallèlement, le même processus de rafinements successifs conduisit à un compilateur Pascal 3.4 pour CDC 6000:



Toujours à Zürich, H.H. Nägeli écrit un compilateur Pascal Tronc, indépendant de toute machine, et devant être complété par les routines de génération de code appropriées pour chaque implantation. Il a été utilisé pour le développement de Pascal sur Hitachi 8000.

Un autre compilateur, réputé entièrement portable, Pascal J a été écrit à l'Université de Boulder, Colorado en 75. Il génère un code intermédiaire, Janus, destiné au macroprocesseur Stage 2.

Enfin, Pascal a donné naissance à plusieurs descendants et collatéraux, parmi lesquels nous citerons :

- Pascal S [Wir76a] , sous-ensemble strict de Pascal, implémenté sur CDC 6000, et Honeywell Multics .
- Modula [Wir76c], tourné vers la programmation système,en cours d'implémentation sur PDP 11.
- Concurrent Pascal [Bri73, Bri75], tourné vers l'écriture de systèmes d'exploitation, et implémenté sur plusieurs machines (au départ PDP 11).
- Simone [BKL76], tourné vers la programmation système et la simulation, fournissant le concept de moniteur de Hoare, et implémenté sur ICL 1900 et CII IRIS 80.
- Lis [CII75], tourné vers la programmation système et implémenté sur CII IRIS 80.

Notons que Pascal est souvent choisi comme langage support de recherches sur la programmation système et la programmation modulaire [CCK76, Bri73, Lec75,SGP, Wir76c, Bri76].

b) *TECHNIQUES D'IMPLEMENTATION*

Découlant du paragraphe précédent, nous pouvons recenser comme techniques utilisées :

α) *Ecriture manuelle complète du compilateur* :

Employée lors de la génération du premier système Pascal, elle n'est utilisée qu'en cas de refonte complète des algorithmes de compilation, décompilation en plusieurs passes, de modifications profondes du langage (Pascal 2) ou de travaux de recherche universitaires (IBM 370). Le langage d'écriture qui s'impose est généralement Pascal (cf. les expériences avec Fortran et PL/I).

β) *Réécriture partielle du compilateur, en conservant les routines d'analyse syntaxique et de gestion de la table de contexte* :

Dans la mesure où un compilateur Pascal lisible, fiable et performant existe, cette méthode conduit à une implantation rapide : ICL 1900, CII IRIS 80.

γ) *Utilisation de langages intermédiaires (P-code Janus) et interprétation* :

Méthode conduisant très rapidement (2 mois en moyenne) à une implantation sans avoir à écrire réellement de compilateur (utilisation de Pascal P ou de Pascal J), elle a l'inconvénient de ne permettre ensuite que le passage de courts programmes :

expériences sur CDC 3600 à Paris VI et sur Philips P1200 à Saint-Etienne, où recompiler le compilateur Pascal P prenait 11 heures de temps d'unité centrale... Cette méthode peut devenir acceptable si le compilateur est destiné à la production de logiciel de base : Pascal STEP à Toulouse sur CII IRIS 80, ou si la machine support est suffisamment puissante (Burroughs B6700, Data General Eclipse : recompilation en une heure).

δ) *Utilisation de langage intermédiaire et macroassemblage du code intermédiaire* :

Cette technique se ramène en fait à une compilation en deux passes. Elle conduit rapidement à un compilateur suffisamment efficace (Burroughs B4700 : recompilation en 5 mn).

ε) *Raffinements successifs du compilateur Pascal P* :

Il s'agit d'appliquer les étapes 7 et 8 de la méthode d'Ammann pour une machine donnée. On obtient ainsi une implémentation d'excellente qualité en un temps assez court (Pascal 3.4 sur CDC 6000). Cependant l'orientation de Pascal P vers une machine à piles et à mots de 60 bits peut être une gêne considérable selon l'architecture de la machine cible.

ζ) *Utilisation du compilateur Pascal-Tronc* :

Ce compilateur étant adapté du compilateur Pascal-P, la technique préconisée de simple écriture des routines sémantiques présente beaucoup d'analogie avec la méthode ε, et semble plus simple.

En conclusion, nous noterons que ces méthodes d'implémentation ne concernent que le compilateur même, et que l'implémentation effective nécessite très fréquemment l'écriture d'un moniteur d'exécution, gérant par exemple les entrées-sorties et l'allocation dynamique de mémoire pour la "pile" et le "tas" [RaR64], ou simulant une machine hypothétique (interprétation du P-code). Or ce moniteur est usuellement un assez gros programme peu portable. Nous pensons qu'on ne peut effectivement dissocier compilateur et système d'exploitation (donc aussi architecture de la machine) dans une approche cohérente ; l'écriture d'un compilateur ne peut alors se concevoir seule. En effet l'exécution de gros programmes (tels qu'un compilateur) sur une machine où l'espace de mémoire centrale adressable est limité fait appel à un mécanisme d'échanges avec une mémoire secondaire (recouvrement, segmentation ou pagination) : le choix d'un tel mécanisme n'est pas indépendant de la structure du langage.

Une alternative à cette conception de systèmes de compilation peut être l'utilisation de compilateurs croisés : le compilateur réside sur une grosse machine, et produit du code pour une plus petite machine qui lui est connectée (nombreuses implémentations de ce type sur PDP 11, CDC STAR 100),

tous les gros programmes étant exécutés sur la machine centrale. C'est là certainement une solution d'avenir, mais qui pénalise lourdement toute une classe d'utilisateurs de mini- et microordinateurs.

B - LA MACHINE

Nous allons ici décrire brièvement l'architecture de la gamme de miniordinateurs Solar 16 et TI600. Nous ne présenterons que les aspects utiles à la compréhension du chapitre II, en renvoyant le lecteur pour plus de détails aux manuels du constructeur [Tél73]. Puis nous en dégagerons les points communs à d'autres machines, en vue d'une éventuelle portabilité du compilateur Solar 16. Le transport sur d'autres minis ou micro-ordinateurs sera en fait traité au chapitre III.

1 - LA MEMOIRE PRINCIPALE ET LES REGISTRES

Solar 16 est une machine à mots de 16 bits (accès au mot en 375 ns à 1,2 µs), où l'adressage se fait au niveau du mot, sauf pour quelques instructions référençant des bits, octets ou double-mots. La taille des mots conduit à un espace adressable de 32 K mots ; cependant la taille d'une adresse sur le bus mémoire - 20 bits - permet l'utilisation de 1024 K mots de mémoire. Le mécanisme d'extension d'adressage est ici l'utilisation de "tâches software esclaves" référençant chacune une zone d'au maximum 32 K mots (pas de pagination ni de mapping).

Les registres programmables, au nombre de 12, sont pour chaque tâche spécialisés de la manière suivante :

A accumulateur : opérations arithmétiques et logiques

B extension de l'accumulateur : opérations sur 32 bits

X index : pour l'adressage des chaînes de bits, d'octets ou de mots

Y travail : pour mémoriser des valeurs temporaires

C base commune : registre de base donnant accès à 256 mots de données

L base locale : registre de base donnant accès à 256 mots de données

W base de travail : registre de base donnant accès à 256 mots de données

K pile : pointeur de pile (en mémoire)

P compteur ordinal

ST registre d'état

SLØ début de zone esclave

SLE fin de zone esclave

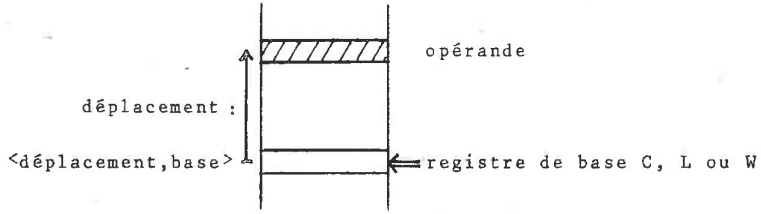
Les quatre registres A, B, X et Y sont banalisés pour les opérations du type chargement et rangement ; l'accumulateur est nécessaire pour les opérations, et le traitement des octets et bits.

2 - L'ADRESSAGE

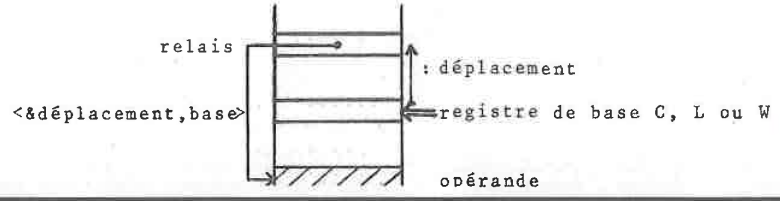
Il se fait uniquement dans la zone de mémoire affectée à la tâche, et bornée par les registres SLØ et SLE, relativement au registre de base SLØ : aucune translation d'adresses n'est nécessaire lors du chargement d'une tâche esclave.

Les instructions avec référence mémoire utilisent un adressage basé : l'instruction sélectionne un des registres de base C, L ou W, puis indique un déplacement de ± 128 mots par rapport à l'adresse pointée par le registre. Il existe trois modes d'adressage d'opérandes en mémoire :

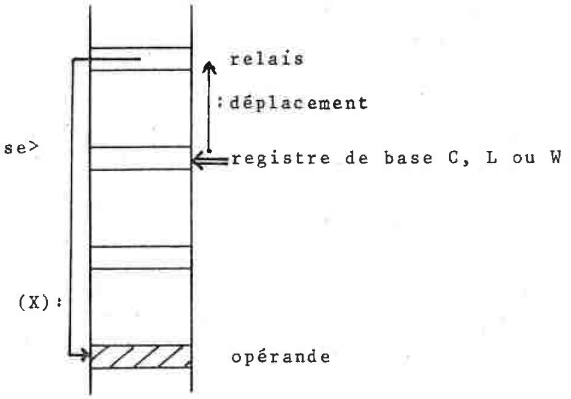
- adressage direct : l'adresse effective de l'opérande est donnée par la somme des valeurs d'un registre de base et d'un déplacement.



- adressage indirect : l'adresse de l'opérande est obtenue par un adressage direct.



- adressage indirect post-indexé : l'adresse de l'opérande est obtenue par un déplacement supplémentaire, donné par le registre d'index X, après un adressage indirect.



L'indexation ne peut donc se faire directement, sans relais d'indirection.

La distinction entre ces deux derniers adressages est faite par la valeur du premier bit du relais, qui est positionné à 1 pour indiquer l'indexation.

Ainsi une tâche peut accéder directement à 3 × 256 mots au maximum à chaque instant. L'adressage d'opérandes situés hors d'une de ces trois sections de données se fait par :

- une modification d'un des registres de base
- ou un adressage indirect en utilisant un relais placé dans une section de données accessible.

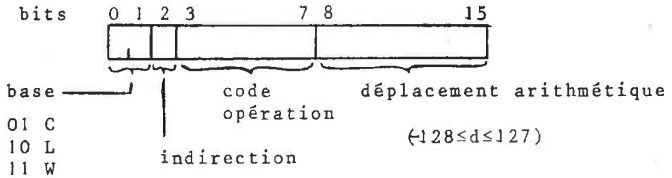
L'adressage des tables (chaînes de bits, d'octets, de mots ou de double-mots) nécessite une indirection préalable : les tables ne seront donc pas situées dans des sections de données accessibles par un des registres de base. De même l'utilisation de l'adressage basé mène à séparer physiquement en mémoire les sections de données des sections de programme.

Le registre K pointe sur une zone de mémoire gérée comme une pile ; l'accès s'y effectue par l'intermédiaire d'instructions spécialisées. Cette pile est notamment utilisée par le superviseur, et lors des appels-retours de sous-programmes.

3 - LE CODE D'ORDRE

Le jeu d'instruction particulièrement riche (140 instructions - voir en annexe) comporte essentiellement des instructions à une adresse, et quelques instructions sans adressage, entre registres ou spéciales. Toutes les instructions tiennent sur 16 bits, sauf celles concernant l'arithmétique flottante (sur 32 bits). Nous en donnons ci-dessous quelques exemples.

- Instructions avec référence mémoire :



exemples :

LA LB LX LY LBY chargement de registres

XM échange d'un mot mémoire avec l'accumulateur

AD SB MP DV opérations arithmétiques

CP comparaison entre mot mémoire et accumulateur

- Instructions avec opérande immédiat :

exemples:

LAI LBI chargements immédiats de registres

ADRI addition immédiate à un registre

- Instructions entre registres :

exemples :

LR XR échanges entre registres

LRP chargement de P dans un registre

ADR CPR addition, comparaison entre registres

LRM chargement de 1 à 8 registres par les valeurs suivant l'instruction

- Instructions de saut :

exemples :

JIX JDX saut conditionnel après incrémentation ou décrémentation du registre d'index

JMP JC JV ... saut conditionnel aux indicateurs report et débordement

Ces instructions permettent un saut de ± 128 mots par rapport à l'instruction : ainsi le code d'un programme n'a pas à subir de translation à chaque chargement.

- Instructions de traitement d'octets :

exemples :

LBY STBY CPBY chargement, rangement, comparaison

SBS recherche d'un octet dans une chaîne

- Instructions de traitement de bits :

exemples :

SLRS SARD ... décalages arithmétique, logique ou circulaire, à droite ou à gauche, dans A (16 bits) ou AB

RBT SBT IBT TBT mise à 0, mise à 1, inversion et test d'un bit dans AB

DBT recherche du premier bit à 1 dans AB

RBTM SBTM mise à 0, mise à 1 d'un bit d'un mot en mémoire

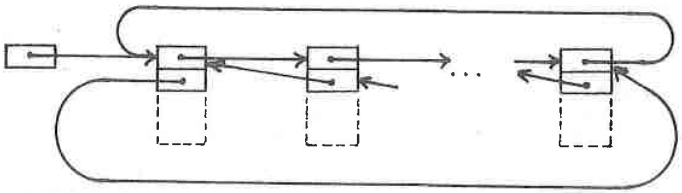
DRBM recherche du premier bit à 1 d'un mot en mémoire, et remise à 0 de ce bit

- Instructions de traitement de pile :

exemples :

PSR PLR empilement ou dépilement de 1 à 8 registres dans la pile pointée par K

- Instruction de traitement de liste :



exemples :

INSQ insertion dans une liste

SFQ suppression en tête de liste

SUPQ suppression dans une liste

SLQ suppression en queue de liste

- Instructions de traitement de sous-programmes :

exemples :

BSR branchement à un sous-programme et empilement de l'adresse de retour dans la pile K

RSR retour de sous-programme, l'adresse de retour étant dépilée de la pile K

SVC RSV appel et retour d'une routine du superviseur, avec empilement de l'adresse de retour

Ces instructions sont bien adaptées à l'emploi de la récursivité. La transmission des paramètres s'effectue généralement par l'intermédiaire de la pile pointée par K ; une économie de temps d'exécution et d'espace mémoire (nombre d'instructions nécessaires à la transmission d'un paramètre) aurait été réalisée par la microprogrammation d'instructions de transmission de paramètres (par valeur et par référence).

- Autres instructions :

exemples :

MOVE transfert d'une zone mémoire de plusieurs mots

MVTS MVTM transfert de zones mémoire entre zones maîtres ou esclaves

DBP SBP RBP recherche, positionnement ou effacement de points d'arrêt

IPI interruption entre processeurs

RQST RLSE primitives P et V sur sémaphore d'exclusion

WAIT ACT primitives P et V sur sémaphore privé

Nous examinerons au paragraphe suivant les instructions ARM

4 - LES TACHES ET LEUR ORDONNANCEMENT

Les processus séquentiels parallèles sont représentés sur Solar par des "tâches software esclaves". Il peut en exister au maximum 128, chacune affectée d'un niveau de priorité de 0 à 127. Leur ordonnancement est réalisé par un microprogramme - le scheduler - en fonction :

- du contexte de chaque tâche : valeur des douze registres A, B, X, Y, C, L, W, K, P, ST, SLØ et SLE au moment de la dernière interruption de la tâche. SLØ et SLE limitent l'espace de mémoire alloué à la tâche.

- de l'état de la tâche, précisé par trois files de 128 bits :
  1) file a : chaque bit à 1 indique une tâche armée par l'instruction ARM.
  2) file e : chaque bit à 0 indique une tâche en attente derrière un sémaphore : attente de ressource derrière un sémaphore d'exclusion, ou attente de synchronisation derrière un sémaphore privé.
  3) file r : chaque bit à 0 indique une tâche non résidente, qui doit donc être rechargée depuis la mémoire secondaire avant exécution.

  Le scheduler scrute ces trois files, dans l'ordre des priorités décroissantes 0 → 127, et active la première tâche armée et non bloquée :
  - . profil a = 1 e = 1 r = 1
  - . ou profil a = 1 e = 1 r = 0. Dans ce cas une tâche spécialisée "complément de scheduling" est préalablement activée par le scheduler pour recharger la tâche sauvegardée en mémoire secondaire.

Deux instructions agissent sur la file a :

ARM n , qui positionne le bit n de la file a : armement d'une tâche

QUIT , qui remet à 0 dans la file a le bit correspondant à la tâche active.

La tâche active perd alors le contrôle de l'unité centrale, son contexte (registres A, B, ... , SLE) est sauvegardé, et le scheduler microprogrammé recherche la tâche activable la plus prioritaire. Cette commutation de tâches se réalise en 6 µs environ, et peut donc survenir fréquemment sans ralentir ostensiblement le déroulement des programmes.

5 - ASPECTS COMMUNS ENTRE SOLAR 16 ET D'AUTRES CALCULATEURS

Une comparaison rapide entre les architectures du Solar 16 et d'autres miniordinateurs - Mitra 15, PDP 11, Nova, TI 990, HP 21000 - conduit à retenir :

- la taille du mot de mémoire : 16 bits, bien que certaines machines réalisent l'adressage au niveau de l'octet.

- l'espace adressable (32K mots) et la taille de la mémoire (jusqu'à 1024 K) utilisable par l'intermédiaire de tâches, de pagination ou de mapping.

- la notion d'accumulateur pour l'exécution des opérations arithmétiques et logiques. Suivant la machine, il existe jusqu'à 16 accumulateurs ; nous pouvons donc transporter le compilateur Solar en n'utilisant qu'un de ces accumulateurs.

- la notion de registre d'index (même remarque que précédemment).

- l'adressage basé : suivant la machine, deux (Mitra 15) ou trois (Solar 16) segments de données sont accessibles par des registres de base, ou deux pages (page courante et page 0) à huit pages sont accessibles directement.

- le code d'ordre riche, pouvant contenir des instructions

- les instructions d'appel-retour de sous-programme, réalisant directement la récursivité (sauf pour Nova et HP2100).

Les différences porteront donc pour nous surtout sur l'adressage indirect qui est généralement à plusieurs niveaux, sur l'indexation, assez particulière sur Solar. Les tâches que nous utiliserons sur Solar n'existent pas ailleurs sous la même forme évoluée, mais peuvent cependant être réalisées au moins par des moyens logiciels. A tous points de vue, la machine la plus proche du Solar 16 est le Mitra 15.

C - CONTRAINTES EXTERNES

En dehors des contraintes de compilation dues au langage Pascal et à l'architecture du Solar 16, nous nous imposons d'orienter notre travail vers une bonne portabilité et vers une utilisation aisée.

1 - PORTABILITE

Un aspect important de Pascal est la forte portabilité des programmes entre les diverses installations, et la portabilité élevée des compilateurs, tant dépendant d'une machine (compilateur Pascal 1 pour CDC CYBER porté sur ICL 1900 et CII IRIS 80) qu'indépendant (compilateur Pascal P, phases d'analyse syntaxique et de génération de code intermédiaire, porté parfois en moins de deux mois). Notre travail d'adaptation au Solar 16 doit pouvoir permettre un transport simple du compilateur pour une classe de machines : miniordinateurs, adressage basé, 32K mots de 16 bits, registres accumulateur et d'index. Nous définirons donc ce compilateur en tenant compte à la fois du Solar 16 et des autres machines, et nous essayerons de proposer une technique simple de transport, telle que paramétrisation de certaines parties du compilateur, et réécriture de routines isolées et bien définies. Nous reviendrons sur cet aspect au chapitre III.

2 - SYSTEME DE COMPILATION TOURNE VERS L'UTILISATEUR

A notre sens un bon compilateur ne se limite pas à une détection efficace des erreurs de syntaxe et à une production de code optimisé : il doit aussi être facilement utilisable par un utilisateur peu expérimenté, et faire partie d'un système de compilation fournissant des facilités d'écriture et de mise au point des programmes. Les machines que nous visons particulièrement - Solar 16 et Mitra 15 - étant largement répandues dans las universités et lycées, nous essayerons de fournir un système de compilation orienté vers l'enseignement. Le choix du langage Pascal permet de résoudre les critères de facilité d'écriture des programmes, et d'adaptation à l'enseignement de la programmation. La mise au point des programmes à l'exécution sera facilitée par la détection, par un moniteur spécialisé, des erreurs d'exécution et par la fourniture d'une édition sous forme naturelle des valeurs prises par les variables en cas d'arrêt intempestif de l'exécution ("post-mortem-dump"). Enfin nous tenterons de fournir aux utilisateurs une documentation lisible !

II - implémentation

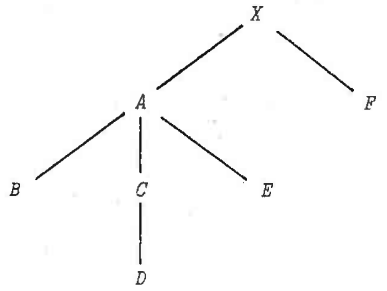
A - ALLOCATION DE MEMOIRES

1 - MECANISME D'EXECUTION D'UN PROGRAMME

Un langage tel que Pascal impose au programmeur, par l'intermédiaire des règles de visibilité des noms, une structure arborescente pour l'écriture des programmes [DHD72].
Soit, par exemple, un programme structuré de la façon suivante :

program X
déclaration de X
procedure A
déclarations de A
procedure B
déclaration de B
corps de B
procedure C
déclaration de C
procedure D
déclaration de D
corps de D
corps de C
procedure E
déclaration de E
corps de E
corps de A
procedure F
déclarations de F
corps de F
corps de X

La structure de l'imbrication des déclarations de procédures, que nous appellerons structure statique, peut se représenter par l'arborescence :



où chaque procédure située à un noeud quelconque peut utiliser les objets déclarés au préalable dans les procédures situées aux noeuds aïeux. Ainsi la procédure D connaît les objets déclarés dans C, A et X à l'exception des procédures E et F qui sont déclarées après D.

Lors de l'exécution de cette procédure D, tous les objets de D, C, A et X devront être directement accessibles.

On appellera niveau statique d'une procédure son niveau dans l'arborescence des déclarations : X est au niveau 0, A et F au niveau 1, ...

L'exécution d'un programme sera un parcours dans l'arborescence des déclarations statiques, et l'on pourra adresser des objets par leur niveau statique relatif puisque les seuls objets accessibles par une procédure sont les objets

2 - PILE GENERALE

Pour la gestion de la mémoire, une procédure correspond à la notion habituelle de bloc : la durée de vie des objets locaux à la procédure est limitée par le mécanisme appel-retour de procédure. Nous devons alors envisager une gestion de la mémoire en pile [Pai72, Sch75, Gri69, GoH76, RaR64]

- à l'appel d'une procédure, il y a allocation au sommet de la pile générale d'un bloc contenant les données locales de la procédure

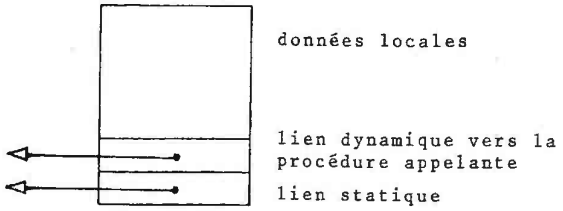
- au retour d'une procédure, il y a désallocation, au sommet de la pile générale, du bloc correspondant à cette procédure.

La gestion de ces blocs en pile nécessite l'emploi de liens dynamiques correspondant à l'enchaînement des appels de procédure. Si la pile générale est effectivement gérée par une méthode de pile, l'accès doit cependant pouvoir s'y faire directement selon deux autres méthodes :

- pour les objets adressés par leur niveau statique relatif, l'accès se fera par un parcours de liens statiques liant les blocs correspondants à des procédures situées dans la même branche de l'arborescence des déclarations

- pour les objets tels que les paramètres passés par référence ou les pointeurs, l'accès doit pouvoir se faire par un adressage direct dans la pile générale.

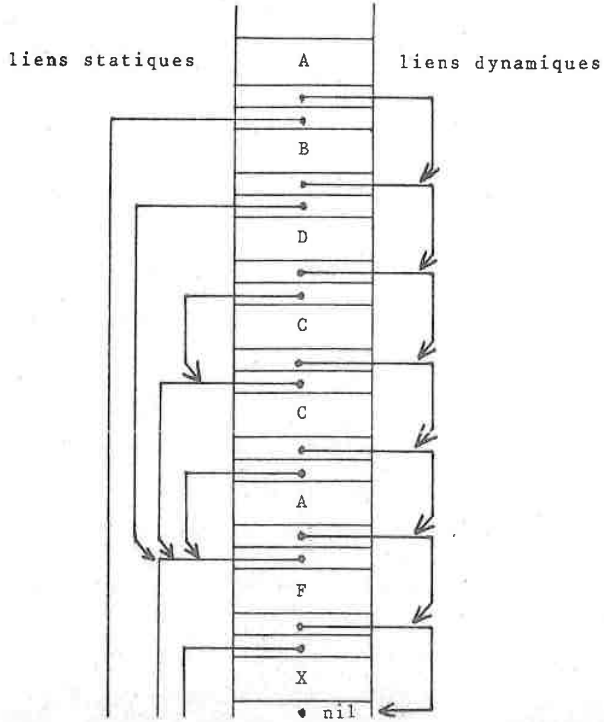
Ainsi pour une pile générale résidant en mémoire principale chaque bloc aura la structure



En reprenant l'exemple précédent, l'enchaînement des appels de procédure

$X \rightarrow F \rightarrow A \rightarrow C \rightarrow C \rightarrow D \rightarrow B \rightarrow A$

sera mémorisé dans la pile générale de la façon suivante



3 - MEMOIRE VIRTUELLE

Désirant pouvoir exécuter sur des miniordinateurs, donc avec une mémoire centrale de taille réduite (de l'ordre de 16 à 32 K mots de 16 bits), de gros programmes - tels qu'un compilateur - sans avoir à prévoir de segmentation préalable manuelle, nous ferons de cette pile générale une mémoire virtuelle segmentée [Bri73, Cro75].

Du fait des liens statiques, donc de la structure de blocs, l'adressage des données se fait :

a) pour des données locales à la procédure active, par une adresse relative dans le bloc sommet de la pile générale.

b) pour des données globales, par un niveau statique relatif (accès à la dernière activation d'une procédure englobante à la déclaration), et une adresse dans le bloc ainsi concerné dans la pile générale.

c) pour des données gérées dynamiquement (variables accessibles par des pointeurs), et pour les paramètres des procédures transmis par référence, le bloc de la pile générale contenant la donnée ne correspond pas nécessairement à la dernière activation d'une procédure englobante ; il n'est alors pas accessible par un chaînage statique. Nous devons pouvoir accéder à tout bloc de la pile directement, par un nom de bloc (= d'activation de procédure) et une adresse dans le bloc.

Ainsi soit le fragment de programme

```
procedure A ;
    var x : integer ;
        procedure B (var y : char) ;
        var z : boolean ;
            begin     x := ...
                      y := ...
                      z := ...
            end ;
```

L'affectation z:= donne lieu à un adressage local, par une adresse simple.

L'affectation x:= nécessite un adressage global, par un chaînage statique vers le bloc correspondant à la dernière activation de la procédure A, puis une adresse dans ce bloc.

L'affectation y:= utilise un adressage direct de bloc : le compilateur ne connait ni le nom de la procédure où est déclaré l'objet effectif y, ni l'activation de cette procédure contenant l'objet effectif.

Pour accélérer l'accès aux données, nous rendrons résidentes en mémoire principale les données les plus fréquemment adressées. Dans le cas du langage Pascal des mesures [For75] ont montré, ainsi que la structure imposée des programmes le laissait prévoir, que les données les plus utilisées par une procédure sont ses données locales, puis les données globales au niveau du programme principal, et finalement les données

globales à des niveaux intermédiaires, assez peu fréquemment référencées.

Ainsi nous rendons résidents en mémoire principale

- les données du programme principal
- les données de la procédure active, c'est-à-dire une copie du bloc sommet de la pile d'exécution
- et, pour pouvoir effectivement l'exécuter, le code de la procédure active.

Le code des autres procédures, et la pile générale n'ont pas à être chargés en mémoire principale : la taille de mémoire nécessaire pour exécuter un programme sera celle de la plus grosse procédure, plus la zone nécessaire aux données de la procédure principale.

4 - SEGMENTS

Cependant, afin de pouvoir exécuter des programmes comportant de 'gros' objets (tableaux, variables dynamiques...), nous distinguerons à l'intérieur de la zone des données d'une procédure, deux parties :

- l'une comportant les gros objets, et gérée comme une mémoire paginée ;
- l'autre comportant les objets de taille plus réduite, et chargée directement en mémoire principale.

De plus nous distinguerons dans cette seconde partie de la zone de données quatre segments pour faciliter les sauvegardes et rechargements avec la mémoire secondaire des procédures interrompues par l'appel d'une autre procédure, et préparer l'utilisation des bases d'adressage :

- un segment comportant uniquement des constantes, invariables lors de l'exécution de la procédure ;
- un segment comportant uniquement des variables et relais ;
- un segment comportant de 'petits' objets structurés, adressables par des relais placés dans le segment précédent ;
- un segment correspondant à la pile pointée par le registre K de la machine.

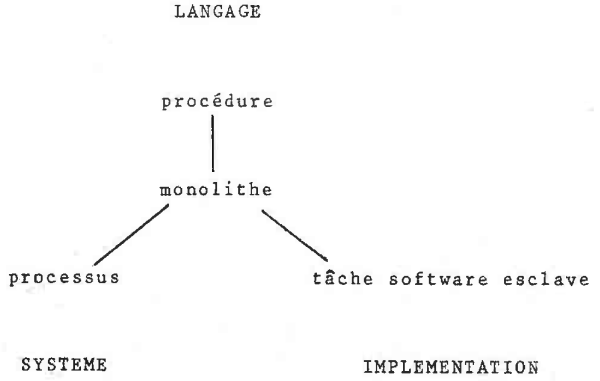
Ainsi la zone affectée à l'exécution d'une procédure sera partagée en :

Segment de Code
Segment C : constantes
Segment R : variables résidentes
Segment T : tables
Segment P : tampon de pagination

5 - MONOLITHES

Par la suite, nous appellerons monolithe une procédure en cours d'exécution. Nous distinguerons parmi tous les monolithes, c'est-à-dire toutes les procédures dont l'exécution a été demandée par une instruction Pascal d'appel de procédure et dont l'instruction Pascal "fin de procédure" n'a pas été exécutée, le monolithe actif, correspondant à la procédure effectivement active. Nous appellerons également monolithe résident, tout monolithe dont l'exécution peut être reprise immédiatement, sans nécessiter d'échanger avec la mémoire secondaire.

L'aspect langage des monolithes, en tant qu'objets représentant des procédures, se complète par un aspect système : une procédure Pascal est en fait un bloc, au sens algolique du terme, dont nous ferons un processus pour résoudre naturellement les problèmes de gestion de la mémoire principale, ainsi que nous le verrons par la suite. Enfin l'aspect implémentation des monolithes se traduira par l'utilisation de tâches représentant ces processus.



Ce "partage du travail" effectif entre le compilateur et le moniteur d'exécution vise à répondre à des soucis de méthodologie, de simplicité et de fiabilité. De plus, la présence de microprogrammes d'ordonnancement des tâches software esclave sur Solar 16 doit permettre la réalisation d'un système d'exécution de programmes Pascal performant sur cette gamme de machines.

6 - RESIDENCE DES MONOLITHES

L'exécution d'un programme Pascal se ramène ainsi pour la gestion de la mémoire à une suite d'appels de procédures, se traduisant par des créations de monolithes. Tant que l'espace de mémoire principale alloué à l'exécution du programme n'est pas saturé, les monolithes inactifs -correspondant à des procédures ayant exécuté une instruction appel de procédure, et attendant la terminaison des procédures appelées- peuvent résider dans cet espace mémoire, afin de réduire les échanges avec la mémoire secondaire. Lorsqu'il ne reste plus d'espace suffisant pour créer de nouveaux monolithes, des sauvegardes en mémoire secondaire doivent être exécutées. Afin de limiter les rechargements depuis la mémoire secondaire, nous effectuerons ces sauvegardes dans l'ordre :

a) segments contenant des données de type variable : variables résidentes, tables non constantes, tampon de pagination, pile du monolithe ;
b) segments invariants : code, constantes, tables invariantes.

Pour reprendre l'exécution d'un monolithe, tous ses segments doivent être chargés en mémoire principale. Ainsi l'algorithme d'ordonnancement des processus associés aux monolithes devra tenir compte de l'état de résidence. Nous examinerons cet algorithme au paragraphe C.

La préparation de l'exécution des monolithes, c'est-à-dire le découpage des procédures en divers segments dans le code objet, doit se faire lors de la compilation des programmes. Pour produire du code objet, le compilateur tiendra donc compte des

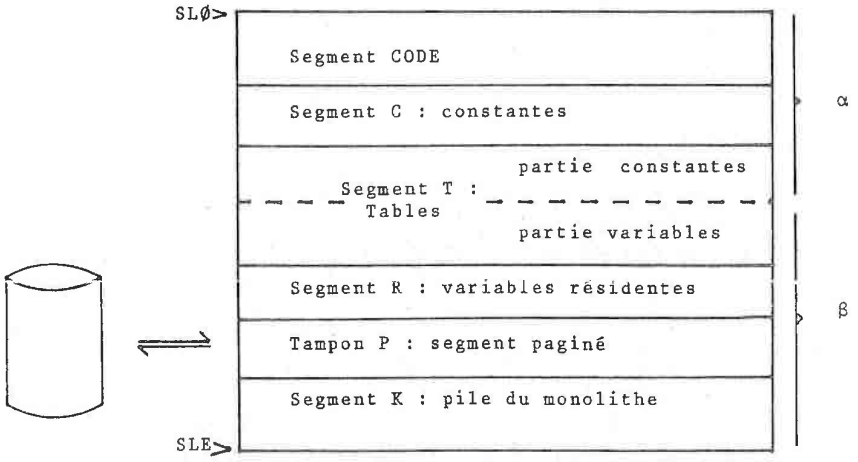
particularités d'adressage des données dans le monolithe (accès au segment, puis au mot dans le segment) et de l'indépendance des monolithes (accès au monolithe par son nom, ou par un niveau statique relatif).

7 - IMPLANTATION DES MONOLITHES

Pour le programme Pascal, un monolithe est une procédure en cours d'exécution ; pour le système d'exploitation c'est un processus séquentiel. Sur Solar 16 nous les représenterons par des tâches software esclaves. L'étroit parallélisme entre ces notions de bloc (= procédure), de processus et de tâche software, ainsi que la présence de microprogrammes de gestion de ces tâches, permet d'espérer de bonnes performances du système.

Notons que cette représentation sur Solar 16 nécessite la contiguïté de tous les segments de chaque monolithe, puisque chaque tâche software esclave ne peut adresser qu'une zone de mémoire limitée par les registres SLO (Slave Origin) et SLE (Slave End).

L'implantation sur Solar 16 d'un monolithe en mémoire principale prend alors la configuration suivante :



a) le segment CODE débute à l'adresse pointée par le registre SLO, donc à l'adresse 0 de la tâche : nous éliminons ainsi tout problème de translation d'adresse lors des sauvegardes-rechargements.

b) la zone α ne contient que du code ou des données invariantes ; elle ne dépend donc pas du monolithe, mais de la procédure correspondante, et n'a alors pas à être sauvegardée sur la pile générale.

c) la zone β, contenant des données non invariantes, correspond au bloc de la pile générale. Elle subira les sauvegardes-rechargements globalement, et non au niveau des segments la constituant.

d) le segment P réside en mémoire secondaire ; un tampon en mémoire principale est alloué au système de pagination, ce qui permettra l'utilisation de la pagination matérielle du Solar 16.

e) le segment K se termine à l'adresse pointée par le registre SLE : tout débordement de la pile du monolithe provoquera une alarme au moniteur d'exécution.

B - COMPILATION

1 - CHOIX DU LANGAGE D'ECRITURE

Pour obéir à des soucis de facilité d'écriture, de lisibilité, de maintenabilité, de fiabilité et de portabilité, nous avons choisi d'écrire le compilateur en un langage évolué plutôt qu'en un langage du niveau assembleur. La perte de performance à l'exécution des programmes, nécessairement induite par ce choix nous a semblé acceptable par rapport au gain d'adaptation aux critères cités ci-dessus.

Le langage choisi pour l'écriture du compilateur doit permettre la réalisation de gros programmes - tels qu'un compilateur - tout en restant simple et suffisamment concis (lisibilité, maintenabilité) ; devant permettre la description de structures complexes, pour la compilation des types de données Pascal, il doit aussi pouvoir refléter naturellement l'imbrication récursive des structures de contrôle ; enfin ce langage d'écriture du compilateur doit être accepté sur des machines variées (portabilité, diffusion et développements de notre compilateur).

Pour ces raisons évoquées brièvement, et développées dans la littérature [Wir71], nous avons choisi d'écrire le compilateur en Pascal, dont une implantation est disponible sur l'ordinateur CII IRIS 80 [MaT74], auquel nous avons accès.

2 - COMPILATION EN DEUX PASSES

Pascal a été conçu pour se compiler en une seule passe [Wir75], ce que réalisent la plupart des implémentations. Cependant, pour des raisons de portabilité, de méthodologie et de facilité nous choisissons une compilation en deux passes, avec utilisation d'un langage intermédiaire.

a) portabilité. Si le langage intermédiaire utilisé est à la fois suffisamment évolué, pour ne pas dépendre d'une machine particulière, et suffisamment de bas niveau, pour que l'effort de compilation soit essentiellement reporté sur la première passe, alors le transport du compilateur entre deux ordinateurs ne nécessitant l'adaptation à la machine cible que de la seconde passe peut représenter un travail de faible envergure [FaG75]. Le choix d'un code intermédiaire adéquat permet alors une bonne portabilité du compilateur.

b) méthodologie. L'évolution récente des langages vers la "programmation structurée" peut se mettre en parallèle avec l'évolution des architectures de miniordinateurs :

- la conception modulaire des programmes se traduit par l'utilisation de registres de base, et par l'existence d'instructions rapides d'appel de sous-programmes et de passage de paramètres.
- la récursivité se traduit par des instructions d'appel de sous-programmes directement récursives.
- l'implantation des types de données tels que les ensembles et les tableaux est facilité par des instructions de manipulation de chaînes de bits, d'octets ou de mots.
- etc.

En définissant un langage de complexité intermédiaire entre Pascal et les langages-machine, nous souhaitons préciser et formaliser cette évolution : notre langage intermédiaire devra refléter les éléments communs aux machines et aux langages. Restant indépendant d'un mini-ordinateur particulier, il permettra de préciser les ressemblances entre les machines, donc de préciser l'évolution de leur architecture. Restant langage intermédiaire, il facilitera la détermination des divergences d'évolution entre machines et langages dits structurés.

Finalement ce langage intermédiaire peut être le point de départ d'une réflexion sur la conception des langages de programmation ainsi que des architectures de miniordinateurs.

c) facilité. L'existence du compilateur Pascal-P permet d'envisager la réalisation rapide de la première passe de notre compilateur, les routines d'analyse lexicologique et syntaxique pouvant généralement être conservées telles quelles. La suite logique de l'adaptation du compilateur Pascal-P à notre réalisation aurait pu être l'affinage des instructions générées jusqu'à obtention directe de code machine. Pour des raisons déjà évoquées, nous préférons effectuer une seconde passe de compilation.

Par la suite nous donnerons le nom de Babel 19 au langage intermédiaire choisi. Pour éviter une description fastidieuse de ses instructions, nous examinerons plutôt une machine hypothétique acceptant directement le langage Babel 19 (ce langage est décrit plus en détails en annexes et en [GoT76, GMT77a]).

3 - MACHINE BABEL

MEMOIRE D'EXECUTION

C'est une mémoire à accès direct, organisée en mots de 16 bits. Sa taille doit être suffisante pour contenir le système d'exploitation, le monolithe le plus gros du programme (code et segments de données) et les segments de donnée du monolithe de niveau statique absolu 0 (c'est-à-dire le programme principal).

ADRESSAGE

A chaque monolithe sont associés quatre registres de base : C, R, T et P qui permettent l'accès aux données contenues respectivement dans les segments constant, résident, tables et paginé.

Chacun de ces segments de données est spécialisé :

- le segment Constant contient des constantes et des relais d'indirection, générés par le compilateur.
- le segment Tables contient des tableaux et des chaînes de caractères ; ces objets ne sont accessibles que par des relais d'indirection placés dans un autre segment. Une partie initialisée du segment tables (adresses négatives par rapport au registre T) est générée par le compilateur ; l'autre partie contient des variables.
- le segment Résident contient des objets non initialisés, modifiables à l'exécution.
- le segment Paginé a le même rôle que le segment résident, mais est optionnel : si la taille des segments résident ou tables est limitée par les possibilités de la machine support, alors des objets sont placés dans ce segment paginé, qui peut être ou non réellement paginé.

Dans le cas le plus simple, une adresse est donc un triplet <N S D> , N étant le niveau statique du monolithe contenant l'objet référencé, S le nom du registre de base concerné et D un déplacement en mots par rapport à ce registre de base.

Afin de réaliser un compromis entre le langage PASCAL et les possibilités de certains miniordinateurs, les niveaux d'adressage peuvent être :

- le mot
- le bit dans le mot
- l'octet dans le mot
- le mot multiple
- le mot dans une chaîne de mots
- l'octet dans une chaîne d'octets.

Les contraintes d'adressage imposées par le langage Pascal et par les architectures de mini-ordinateurs, mènent à introduire les notions d'adressage :

- local ou global
- direct, indirect ou indirect post-indexé (mais pas uniquement indexé).

REPRESENTATION DES MONOLITHES

Pour la machine BABEL, un monolithe est formé de cinq segments : code, constant, tables, résident et paginé. Les contraintes logiques imposées par BABEL sont les suivantes:

- les segments constant, résident et paginé sont directement adressables, par l'intermédiaire d'un registre de base C, R ou P ; les déplacements par rapport à ce registre sont des entiers positifs ou nuls, indiquant un déplacement en nombre de mots.

- le segment table n'est pas adressable directement. Seuls des relais d'indirection placés dans un autre segment de données permettent d'y accéder. Les déplacements par rapport au registre T sont des nombres entiers, positifs lorsque les objets sont des variables d'exécution, négatifs lorsque les objets sont des constantes initialisées à la compilation.

- les segments code, constant, et adresses négatives dans le segment tables, sont initialisés à la compilation, et non modifiés par la suite ; ils n'ont donc pas à subir de sauvegarde en mémoire secondaire avant écrasement.

- le segment paginé peut être entièrement résident ou être en mémoire virtuelle. Dans ce dernier cas un ou plusieurs tampons en mémoire principale permettent la pagination.

Sur une machine support particulière, ces segments peuvent être réunis, accédés par hard ou soft ; ils peuvent être contigüs ou disjoints ; d'autres segments peuvent leur être ajoutés (piles...)...

REGISTRES ET ACCUMULATEURS

La machine Babel possède 2054 registres chargeables par l'usager, dont les 2052 premiers ont fonction d'accumulateurs :

- C taille : 1 bit
- H taille : 1 octet
- W taille : 1 mot
- D taille : 2 mots
- M accumulateurs de taille multiple du mot (2048 registres M, de taille 1 mot, 2 mots, ..., 2048 mots), nommés 1, 2, ..., 2048
- X registre de post-indexation au niveau du mot
- Y registre de post-indexation au niveau de l'octet ou du bit

Accumulateur C (prononcer : chouïa) : alloué aux scalaires d'au plus deux éléments. Ex : booléens.

Accumulateur H (prononcer : half-word) : alloué aux scalaires de 3 à 64 éléments. Ex : caractères.

Accumulateur W (prononcer : word) : alloué aux scalaires de 65 à 32767 éléments, et aux entiers (-32768 ... +32767).

Accumulateur D (prononcer : double word) : alloué aux nombres réels.

Accumulateur M (prononcer : multiple) : alloués aux ensembles, et scalaires de plus de 32767 éléments. L'accumulateur n est destiné au traîtement des ensembles de cardinal compris entre 16(n-1) et 16 n. La taille maximale d'un ensemble est donc de 32768 éléments.

Ces registres sont utilisés de la manière suivante :

- Une opération arithmétique ou logique monadique porte sur un opérande contenu dans l'accumulateur désigné. Le résultat est remis dans l'accumulateur.

- Une opération arithmétique ou logique dyadique porte sur un premier opérande contenu dans l'accumulateur désigné, et sur un second opérande contenu en mémoire. Le résultat est remis dans l'accumulateur.

- Une suite d'opérations ne peut porter que sur un seul accumulateur : on ne peut pas utiliser un accu si un autre contient une information à conserver.

- Seule l'instruction de calcul d'adresse modifie le contenu des registres X et Y.

- Lors d'un adressage indirect post-indexé, le registre X contient un déplacement en mot par rapport au contenu d'un relais ; le registre Y contient éventuellement un déplacement en octet ou bits à l'intérieur d'un mot.

4 - REPRESENTATION EN BABEL DES STRUCTURES DE DONNEES PASCAL

Scalaire :

suivant le nombre n de valeurs possibles du scalaire, il sera représenté sur

1 bit si $n \leq 2$

1 octet si $n \leq 256$

1 mot si $n \leq 32767$

m mots si $n \leq 32767\ m$

Entier :

sur un mot ; codage binaire avec complément à 2. Pour faciliter les opérations arithmétiques, on ne cherchera pas à compacter la représentation des intervalles d'entiers : array [0..15] of 0..1 sera représenté sur 15 mots.

Booléen :

sur un bit : array [0..15] of boolean sera représenté sur un mot.

Caractère :

sur un octet. Le codage adopté est le code ASCII sans bit de parité.

Réel :

sur deux mots.

Ensemble :

considéré comme une chaîne de bits, un ensemble occupe un nombre entier de mots, selon son cardinal maximal.

Fichier :

représenté par un nom interne (entier positif), un tampon dont la taille est celle des éléments, et par des requêtes d'accès.

Tableau et agrégat :

selon la taille de leurs éléments et champs, ils seront compactés au maximum. Toutefois, pour en faciliter la manipulation, seuls les éléments déclarés au niveau le plus bas pourront être représentés sur des portions de mots ; tous les éléments situés à d'autres niveaux seront alignés sur des frontières de mots.

Variable dynamique :

elle réside toujours dans un segment paginé.

Pointeur :

représenté sur trois mots, il contient :

- un déplacement dans le segment paginé
- le niveau statique relatif du monolithe contenant l'objet pointé, ou le nom du monolithe
- la taille en mots de l'objet.

5 - L'ADRESSAGE DES OBJETS BABEL

La représentation des objets Pascal mène à des niveaux d'accès tels que le bit, l'octet,... Les modes d'adressage sur miniordinateur, la structuration interne des objets imposent des méthodes d'accès telles que l'indirection et l'indirection post-indexée. De plus la notion de monolithe introduit une distinction fondamentale entre adressage local et adressage global.

Afin que Babel soit de complexité intermédiaire entre Pascal et les langages machine de miniordinateurs, nous distinguerons sept types d'accès :

- L0 : adressage direct local, pour les objets simples, totalement déterminés avant l'exécution, appartenant au monolithe actif.
- L1 : adressage indirect local, pour les objets simples, dynamiques, appartenant au monolithe actif.
- L2 : adressage indexé local, pour certains objets structurés appartenant au monolithe actif.
- G1S : adressage indirect global simple, pour les objets simples, n'appartenant pas au monolithe actif ; le niveau d'indirection introduit ici facilite le traitement des objets non locaux.
- G1D : adressage indirect global double, pour les objets simples, dynamiques, n'appartenant pas au monolithe actif.
- G2S : adressage indexé global simple, pour les objets structurés n'appartenant pas au monolithe actif, nécessitant une indexation.
- G2D : adressage indexé global double, pour les objets structurés n'appartenant pas au monolithe actif, et dont l'accès local est de type L2.

a) Adressage de type L0 : local direct.

Ce type d'adressage est utilisé pour l'accès aux objets simples totalement déterminés avant l'exécution et appartenant au monolithe actif (niveau statique relatif égal à 0).

Ainsi, soient les déclarations Pascal :

```
var i : integer ;
    j : array [-4..7] of real ;
    k : record a, b, c : boolean end ;
```

alors les références i, j[2] et k.c se résolvent par un adressage direct.

S : nom du segment contenant l'objet (C, R ou P)

D : adresse relative de l'objet par rapport au registre de base du segment

Z : indications complémentaires éventuelles (déplacement en bits dans le mot référencé, ou nom de l'accumulateur multiple concerné)

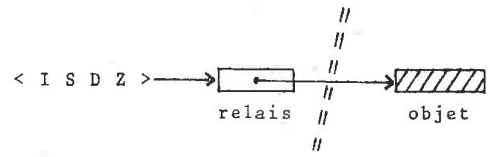
b) Adressage de type L1 : local indirect.

Ce type d'adressage est utilisé pour l'accès aux objets simples de structure totalement déterminée, mais dont l'emplacement dans le segment ne sera connu qu'à l'exécution, et appartenant au monolithe actif (niveau statique relatif égal à zéro). Ces objets étant dynamiques seront générés dans le segment paginé ; par contre le relais peut se trouver dans un autre segment.

Ainsi, soient les déclarations Pascal :

```
type i = set of 'a' .. 'z' ;
     l = char ;
var p : ↑i ;
    q : ↑l ;
```

alors les références p↑ et q↑ sont des adressages de type L1.



S : nom du segment contenant le relais

D : adresse relative du relais par rapport au registre de base du segment

Z : indication complémentaire éventuelle (registres M et C)

L'adresse référencée par <S D> est celle d'un relais sur un objet

c) *Adressage de type L2 : local indirect post-indexé.*

Ce type d'adressage est utilisé pour l'accès aux éléments d'objets structurés nécessitant une indexation, ces objets étant placés dans le monolithe actif (niveau statique relatif égal à zéro). C'est en particulier le cas des tableaux.

Ainsi, soient les déclarations Pascal :

```
type a = record i : array [char] of integer ;
                b : boolean
         end ;
var t : array [char] of char ;
    c : char ; r : a ; p : ↑a ;
```

alors les références t[c], t['b'], r.i[c], p↑.b et p↑.i['s'] sont des adressages post-indexés :

- t[c] et t['b'] : accès à un octet dans une chaîne, nécessitant donc un relais d'indirection et une indexation
- r.i[c] : accès à un élément inconnu d'un tableau dans un agrégat
- p↑.i['s'] et p↑.b : p est un relais d'indirection sur un agrégat, relais de valeur connue seulement à l'exécution

Par contre t['a'], p↑ et r.i['h'] sont des adressages directs.

Pour un adressage de bits :

Le préfixe 'X' indique la post-indexation. L'adresse référencée par <S D> est celle d'un relais sur un objet structuré se trouvant dans le même segment du même monolithe. Le registre d'index X contient un déplacement supplémentaire dans l'objet structuré. Le registre Y contient un déplacement (de bits) dans le mot référencé.

(L'instruction INDX permet le chargement préalable des registres d'index.)

d) Adressage de type G1S : global indirect simple.

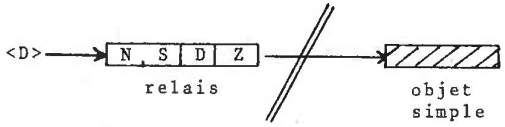
Ce type d'adressage est utilisé pour l'accès à des objets simples n'appartenant pas au monolithe qui les référence.

Ainsi, soient les déclarations Pascal :

```
type c = char ; p = ↑c ;
var a : c ;
procedure h(r:p) ;
```

alors, dans la procédure h, les références r↑ et a sont des adressages de type G1S.

L'adresse référencée par <D> dans le segment constant ou par <@D> dans le segment résident du monolithe actif est celle d'un relais sur l'objet simple :



où N > 0 N est le niveau statique du monolithe contenant l'objet adressé ;

N < 0 -N est le nom du monolithe ;

S est le nom du segment contenant l'objet ;

D est un déplacement dans ce segment S ;

Z est une éventuelle indication complémentaire.

Cet adressage sera généralement traduit par une requête au système d'exploitation, en lui passant en 'bloc de contrôle de requête' le relais généré par le compilateur.

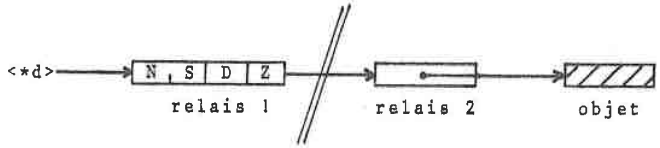
e) Adressage de type G1D : global indirect double.

Ce type d'adressage correspond à des objets simples n'appartenant pas au monolithe qui les référence, et dont l'accès local est de type L1.

Ainsi, soient les déclarations Pascal :

```
type a = char ; p = ↑a ;
var 1 : p ;
procedure h(var q : p) ;
```

alors, dans la procédure h, les références q et 1↑ se résolvent par un adressage de type G1D.



Le préfixe '*' indique la double indirection. d est l'adresse d'un relais situé dans le segment constant du monolithe actif, ou *@d si le relais est dans le segment résident.

N > 0 N est le niveau statique du monolithe contenant le second relais ;

N < 0 -N est le nom du monolithe ;

S est le nom du segment contenant le second relais ;

D est un déplacement dans le segment S.

Le second relais et l'objet sont situés dans le même segment du même monolithe.

Z est une indication complémentaire éventuelle.

Cet adressage sera généralement traduit par une requête au système d' exploitation, en lui passant en 'bloc de contrôle de requête' le premier relais (relais 1) généré par le compilateur.

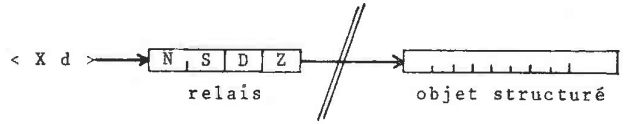
6) *Adressage de type G2S : global indirect post-indexé simple.*

Ce type d'adressage est utilisé pour l'accès à des objets structurés nécessitant une indexation, et n'appartenant pas au monolithe qui les référence.

Ainsi, soient les déclarations Pascal :

```
type l = record a : char ;
                b, c : boolean ;
                d : array [char] of real
         end ;
     z = array [boolean] of char ;
     p = ↑l ;
var z1 : z ; l1 : l ;
 procédure h (w : p ; var v : z) ;
```

alors, dans la procédure h, les références z1[true], l1.c, v[x] et w↑.c se résolvent par un adressage de type G2S.



où N > 0 N est le niveau statique du monolithe contenant l'objet structuré ;

N < 0 -N est le nom du monolithe ;

S est le nom du segment contenant l'objet ;

D est un déplacement dans le segment S ;

Z est une indication complémentaire éventuelle.

(Le préfixe 'X' indique l'indexation.)

D est un déplacement dans le segment constant ou résident (<X@D>) du monolithe actif, précisant le relais utilisé.

Le registre d'index X contient un déplacement supplémentaire dans l'objet adressé. (L'instruction INDX permet le chargement préalable des registres d'index.)

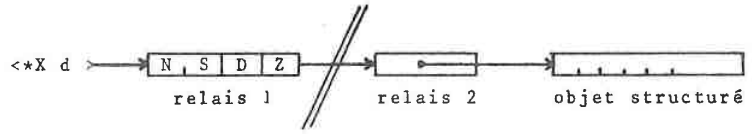
g) *Adressage de type G2D : global indirect post-indexé double.*

Ce type d'adressage est utilisé pour l'accès à des objets structurés n'appartenant pas au monolithe, et dont l'accès local est de type L2.

Ainsi, soient les déclarations Pascal :

```
type p = ↑l ;
     l = record a : char ;
                b, c : boolean ;
                d : array [char] of real
         end ;
var g : p ;
 procedure h(var u : p) ;
```

alors, dans la procédure h, les références u↑.c et g↑.c se résolvent par un adressage de type G2D.



Le préfixe '*' indique la double indirection.

d est l'adresse d'un relais situé dans le segment constant ou résident (<*X@d>) du monolithe actif.

'X' indique l'indexation.

N > 0 N est le niveau statique du monolithe contenant le second relais.

N < 0 -N est le nom du monolithe.

S est le nom du segment contenant le second relais.

D est un déplacement dans le segment S. Le second relais et l'objet sont situés dans le même segment du même

Z est une indication complémentaire éventuelle.

Le registre d'index X contient un déplacement supplémentaire dans l'objet adressé. L'instruction INDX permet le chargement préalable des registres d'index.

Cet adressage sera généralement traduit par une requête au système d'exploitation, en lui passant en 'bloc de contrôle de requête' le premier relais (relais 1) généré par le compilateur.

6 - PROGRAMMATION EN BABEL

Un programme Babel est composé d'instructions et de directives, groupées en déclarations de monolithes.

Les instructions expriment des actions à entreprendre à l'exécution du programme, les principales classes étant :

- chargements et rangements d'accumulateur
- calcul d'adresse dans les types structurés
- opérations monadiques
- opérations dyadiques
- comparaisons
- ruptures de séquence
- opérations sur monolithes
- opérations d'entrée/sortie

Les directives décrivent des informations nécessaires au fonctionnement de la machine Babel (déclarations d'étiquettes, de procédures compilées séparément, ...), ou à l'environnement d'exécution (déclarations de noms en vue d'édition de post mortem dumps, ...).

a) Chargements et rangements :

Chargements immédiats

Leur utilité est de permettre l'utilisation des opérations immédiates sur miniordinateurs.

Exemple : LDCN load constant nil : chargement dans W du pointeur nul

Chargements et rangements adressés

Exemple :	bit :	LODC STOC	
	demi-mot :	LODH STOH	
	mot :	LODW STOW	adressage de type
	double-mot:	LODD STOD	L0, L1, L2,
	multiple :	LODM STOM	G1S, G2S, G1D ou G2D

b) *Comparaisons* :

Ces instructions effectuent la comparaison de deux valeurs de même type scalaire.

Le résultat de la comparaison est rendu dans l'accumulateur C, et pourra être utilisé par l'instruction de rupture de séquence conditionnelle.

Les comparaisons possibles sont :

	<	<=	=	/=	>=	>
sur bit	LESC	LEQC	EQUC	NEQC	GEQC	GRTC
sur octet	LESH	LEQH	EQUH	NEQH	GEQH	GRTH
sur mot	LESW	LEQW	EQUW	NEQW	GEQW	GRTW
sur double-mot	LESD	LEQD	EQUD	NEQD	GEQD	GRTD
sur multiple de mot	LESM	LEQM	EQUM	NEQM	GEQM	GRTM

c) *Ruptures de séquence* :

Rupture de séquence inconditionnelle [Dij68]

GOTO E

E étant une étiquette déclarée dans la procédure.

Rupture de séquence conditionnelle

FJMP E (false jump)

Un branchement à l'étiquette E (déclarée dans la procédure) est réalisé si l'accu C contient la valeur false.

Rupture de séquence conditionnelle à issues multiples

XJMP R E (indexed jump)

R est le nom d'un accumulateur contenant une valeur d'index, E l'étiquette référençant une table d'étiquettes. Le branchement se produit à la (R) ième étiquette. Cette instruction correspond à l'instruction Pascal case.

d) *Opérations monadiques* :

L'opérande se trouve dans l'accumulateur spécifié. Le résultat est remis dans un accumulateur. Il n'y a pas de référence mémoire.

Exemple :

accumulateur opérande	accumulateur résultat	instruction	signification
W ou M	W ou M	ABSI [Z]	valeur absolue d'un entier
D	D	ABSR	valeur absolue d'un réel
D	D	ATAN	arctangente
R	n	SING R n	génération de singleton (R = C, H ou W)
W ou M	W ou M	SQRI[Z]	carré d'entier

Fonctions spéciales

TIME heure sous forme hh mn ss

e) *Opérations dyadiques* :

L'opération s'effectue entre un accumulateur et un élément de même type en mémoire. Le résultat est remis dans un accumulateur.

Exemple : ADDI addition entière
ADDR addition réelle
AND 'and' booléen
SUBS différence d'ensembles

f) *Opération immédiate* :

Exemple : INCR R n addition immédiate de la valeur entière n au registre R

g) *Opérations sur fichiers* :

Un fichier Babel est connu par son nom (entier positif). Tous les fichiers doivent être déclarés par une ouverture, et sont de type consécutif, accessibles en écriture.

Ouverture :

Fichier permanent : FILE n
(déjà connu du système d'exploitation)

Fichier temporaire : VEDA n
(dont la durée de vie ne dépasse pas celle du monolithe où il est déclaré).

Fermeture de fichier temporaire :

KALI n

Rembobinage : RSET n et RWRT n
effectuent les opérations Pascal reset et rewrite

Accès : GET n l'opérande ou résultat se trouve dans
PUT n l'accumulateur concerné.

Tests : EOF n et EOLN n
gèrent les conditions Pascal correspondantes.

Fichiers de type Pascal text :

Ex: WRTP n NOM écriture d'un ensemble de type NOM sur le fichier n

WRTL n adr écriture d'une chaîne de caractères sur le fichier n, contenue à l'adresse indiquée

PAGE n saut de page sur le fichier n

CADR facteur de cadrage, contenu dans le registre W

par exemple, writeln(reel:7:2)
se traduit par :

```
LØDD reel
WRTR output
LDCW 7
CADR
LDCW 2
CADR
WRTE output
```


h) Opérations sur monolithes :

Chaque procédure reçoit un nom interne à sa déclaration dans le programme (directives PROG, EXT ou instruction RATI). L'utilisation des monolithes correspondant à l'exécution se fait à l'aide de ce nom interne.

Opérations vitales

Ex : CIVA n p incarnation du monolithe correspondant à la procédure n, avec liens statiques vers la procédure p. (sans rapport avec [Der75])

NIRV désincarnation

XIPE réincarnation

RAMA désincarnation de tous les monolithes (instruction Pascal Halt)

Passage de paramètres

Ex : PVAL R paramètre valeur

PVAR adr paramètre référence

PMON n paramètre procédure ou fonction

i) Opérations d'adressage dans les types structurés, de vérification, de génération d'agrégats :

Ex : NEW n s d génération d'objet dynamique

Vérification de bornes pour les scalaires et les intervalles

MANU accu B1 B2

Si la valeur contenue dans l'accumulateur désigné appartient à l'intervalle B1 .. B2, il y a saut d'une instruction (en général BUG), sinon on continue en séquence.

Traitement des erreurs d'exécution

BUG n

Le système traite le cas d'erreur numéro n.

Accès aux tableaux

INDX A T B [+]

Cette instruction rend dans le registre X une adresse relative en mots et dans le registre Y une adresse relative en bits ou octets à l'intérieur d'un mot.

Au départ, W contient le résultat du calcul de l'indice dans le tableau. T est la taille de chaque élément, B la borne inférieure de l'indice, A l'adresse relative du tableau dans l'objet structuré le contenant.

Alors l'adresse relative de l'élément est : T(W-B) + A.

Cette instruction sera examinée plus en détail à la rubrique 'gestion des types structurés'.

j) Directives :

Directives de compilation

Ex :	L n	numéro de ligne source
	:n	étiquette
	END n	fin de compilation de la procédure n
	!e liste!	liste d'étiquettes pour XJMP
	CNST{R, I, H} valeur[,valeur...]	initialisation de constante réelle, entière ou hexadécimale
	TABL{R, I, H, "chaîne"} valeur[,valeur...]	initialisation de table

Directives de dump

Ex :	TYPE nom = ... ;	déclaration de type

En tête de procédure

Ex : FUNC n N nom : type ; S D

(n : nom interne
N : niveau statique)

Référence externe

PRØC { P, F } n N nom ...
FUNC { A }

P : PASCAL
F : FORTRAN
A : ASM

k) Gestion des types structurés :

Les types structurés Pascal

Dans un programme Pascal, les bornes des tableaux sont statiques et ainsi, dans chaque segment, les adresses relatives des diverses variables sont déterminées à la compilation. Le rapport Pascal précise qu'un tableau est identique à un agrégat où les différents champs sont référencés par les indices, et qu'un tableau à n indices d'éléments de type T est identique à un tableau à un indice dont les éléments sont des tableaux à n-1 indices de type T. Ainsi l'accès à un élément d'un objet structuré complexe se réduit à une suite d'accès dans les objets structurés à un seul niveau : T[I] ou A.C .

Représentation Babel

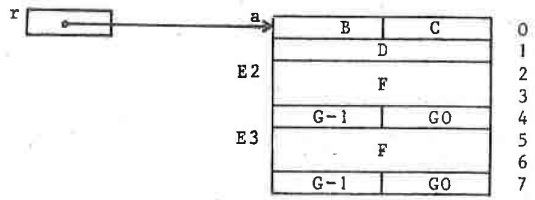
En traduisant Pascal en Babel l'on distingue l'accès à des objets d'adresse connue à la compilation (références explicites) de l'accès à des objets de structure connue, mais de location connue seulement à l'exécution (par exemple des paramètres passés par référence, des agrégats référencés par des variables de type pointeur, une variable indice de tableau).

Dans le premier cas, le calcul d'adresse pour un type structuré est fait à la compilation. Dans le second cas, un relais d'indirection, pointant sur le premier mot mémoire occupé par la variable, permet un adressage indirect post-indexé, ainsi qu'il est d'usage sur miniordinateurs. Les instructions INDX et LØDX permettent le calcul du déplacement dans l'objet.

Par exemple, soit la déclaration de variable :

```
A : record B, C : char ;
              D : integer ;
              E : array [2 .. 3] of record F : real ;
                                           G : array [-1 .. 0]of char
                                  end
    end ;
```

objet représenté en mémoire par :



Double indexation

L'indexation doit permettre l'accès à tous les objets élémentaires du langage, qu'ils soient représentés par des bits, des octets, des mots, ou des multiples mots. Or, sur miniordinateur, les données ne sont usuellement accessibles qu'au niveau du mot. En Babel on différencie alors l'indexation pour accéder à des niveaux d'information plus fins (registre Y).

Instructions d'indexation

INDX A T B [+]	indexation
LØDX valeur	chargement du registre X

A : adresse relative du tableau dans l'objet structuré de niveau supérieur

T : taille en nombre de mots de chaque élément

ou H : élément = demi-mot

ou C : élément = bit

B : borne inférieure de l'indice

+ : si présent, X contient déjà un déplacement, résultat de calculs préliminaires

Le registre W contient la valeur de l'indice.

1) T numérique : X ← A + T(W - B) [+ X]

2) T = H : alors α = 2

W ← W - B

Y ← W mod α

3) T = C : alors α = 16

W ← W div α

X ← A + W [+ X]

En reprenant la déclaration de variable précédente, l'on traduira

A.C	par	LØDX 1 LØDH X S r
A.D	par	LØDW S a+1
A.E[2]	par	LØDM S a+2 3
A.E[2].F	par	LØDD S a+2

A.E[I].F	par	LØDW(I) MANU W 2 3 BUG 0 INDX 2 3 2 LØDD X S r
A.E[I]. G[J]	par	LØDW (I) MANU W 2 3 INDX 2 3 2 LØDW (J) MANU W -1 0 BUG 0 INDX 2 H -1 + LØDH X S r

Cette instruction INDX est donc bien adaptée à la fois à Pascal et aux langages machine de miniordinateurs tels que Solar 16.

7 - LA SECONDE PASSE DE COMPILATION

Ce programme est chargé de l'adaptation du code Babel à la machine choisie (Sur Solar 16, se référer à [GMT77b]).

a) Les instructions Babel correspondant à des instructions de la machine -comme l'addition, l'accès à des données locales résidentes- sont directement traduites.

Exemples sur Solar 16 :

STOD — rangement du registre D (double mot)

Fonction : (adrd) := (D)

Compilation :

			L0	L1	L2
code	FST D,base	rangement direct étendu	1		
code	FST &D,base	rangement indirect étendu	1	1	1

NEGI — négation d'entier

Fonction : (W) := -(W)

Compilation :

code	NGR A,A	(A) := -(A)

INCR — addition immédiate à un registre

Fonction : addition d'une valeur numérique au registre spécifié ; cette valeur est comprise entre -128 et +127

Compilation :

(accu /= multiple)

code	ADRI valeur,A	addition immédiate au registre RA

) Certaines instructions Babel n'ayant pas d'équivalent dans les instructions de la machine -comme le traitement des ensembles, l'accès à des données structurées locales résidentes, les vérifications de bonne exécution, ...- peuvent être traduites par une courte séquence.

Exemples sur Solar 16 :

EXPO — exposant d'un réel

Fonction : (W) := exposant de (D)

Compilation :

code	LBI 'FF	chargement de 'FF dans l'octet droit de RB
code	TBT 8	chargement du carry par le bit de signe de l'exposant
code	JC $+3	saut si exposant négatif
code	ANDR B,A	mise à zéro de l'octet gauche de RA
code	JMP $+3	saut instruction suivante
code	SWBR B,B	échange des octets de RB
code	ORR B,A	mise à 1 des bits de l'octet gauche de RA

DIVI — division entière

Fonction : (accu) := (accu) / (adri)

Compilation :

			L0	L1	L2
code	SARD 16	positionnement dans RB	1	1	1
code	DV D,base	division directe	2		
code	DV &D,base	division indirecte		2	2

code	JNV 3	saut si pas de débordement
code	LAI 10	overflow algébrique division
code	SVC svcerr	erreur

MANU — vérification des bornes

Fonction :

vérifie que la valeur contenue dans l'accumulateur est comprise entre les bornes précisées ;
dans ce cas il y a alors saut d'une instruction Babel.

Compilation :

(accu /= multiple)

1) si -256 ≤ borninf ≤ 255 alors (2) sinon (3)

2)

code	CPI borninf	comparaison immédiate

puis (4)

3)

CNST	gconst (borninf)	génération de borninf en zone constante
code	CP D,L	comparaison à borninf

puis (4)

4)

code	JL 3	saut si accumulateur < borninf

si -256 ≤ bornsup ≤ 255 alors (5) sinon (6)

5)

code	CPI bornsup	comparaison immédiate

puis (7)

6)

CNST	gconst (bornsup)	génération de bornsup en zone constante
code	CP D,L	comparaison à bornsup

puis (7)

7)

code	JLE 2	saut à 2 instructions si accu inférieur ou égal à bornsup

c) Les autres instructions Babel sont "interprétées" lors de l'exécution : elles sont traduites par un appel à une routine spécialisée du moniteur, qui effectuera l'action souhaitée. Dans cette classe d'instructions, nous trouvons les instructions sur monolithes et sur fichiers, les adressages de données globales ou paginées, les instructions d'entrée-sortie, ...

Exemples sur Solar 16 :

BUG — branchement sur erreur

Fonction : traitement éventuel de l'erreur numéro n

Compilation :

code	LAI n	chargement dans A du numéro d'erreur
code	SVC SVCERR	requête d'erreur

NEW — génération d'objet dynamique

Fonction : génération d'un objet dynamique de taille en mots précisée par W
l'adresse de cet objet et sa taille sont rangées dans le pointeur adressé

Compilation :

const	gconst(code)	réservation d'un mot contenant le niveau statique et le segment du pointeur adressé codés ensemble
code	LB D,L	chargement dans RB de ce code
code	LYI Dep	chargement immédiat dans RY du déplacement du pointeur dans le segment
code	SVC newreq	requête de génération

STOD (avec requête)

			L0	L1	L2	G1S	G1D	G2S	G2D
code	FST RCBA,C	sauvegarde de l'accu étendu	1	1	1	1	1	1	1
code	STX RCBX,C	sauvegarde de l'index			2			2	2
code	loadreg(ACD,false) SVC svcad	écriture d'un double mot en mémoire	2	2	3	2	2	3	3
code	LV RCBX,C	restitution de l'index			4			4	4

d) Les directives Babel de déclaration de variables et de constantes donnent lieu à des allocations de mémoire dans les divers segments du monolithe -la ventilation des objets entre les segments ayant eu lieu lors de la première passe- ; les adresses des variables sont notées sur un fichier qui sera dépouillé lors d'un dump éventuel.

Exemples sur Solar 16 :

TABL — directive table

Fonction : initialisation de constante dans le segment table

Compilation :

si constante = caractère alors
  tant que caractère
    cadrage de 2 caractères par mot
    et réservation dans le segment T
  fin tant que
sinon (valeurs réelles, entières ou hexa)
  tant que valeur
    réservation et initialisation
    de la valeur dans le segment T
  fin tant que

directive liste d'étiquette

Fonction : définition d'une liste d'étiquette utilisée par l'instruction XJMP

Compilation :

- mise à jour du relai en zone constante
- pour la deuxième étiquette jusqu'à la dernière
  faire
    écrire cette étiquette dans la zone Table
  fin faire

e) Les directives Babel de déclarations de type sont transmises sur un fichier utilisé par l'utilitaire de dump post-mortem.

La structure du compilateur est alors la suivante : pour chaque procédure, les directives et instructions Babel sont décodées et traitées de façon linéaire. Les programmes donnés en pâture à cette seconde passe du compilateur ont été générés par la première passe, et sont donc syntaxiquement corrects. Il est à noter que c'est cette seconde passe qui résout l'imbrication des procédures, et prépare ainsi la gestion des monolithes par le moniteur d'exécution.

8 - LA PREMIERE PASSE DU COMPILATEUR

Ce programme est chargé :

a) de la compilation des déclarations de données :

Les déclarations de types et de constantes sont entièrement prises en compte (à chaque identificateur un type explicite est associé : voir en annexe) et les déclarations de variables sont traduites en Babel, ce qui implique que la première passe effectue toutes les vérifications de syntaxe et de cohérence entre types, ainsi que l'allocation en mémoire des données statiques.

b) de la traduction en Babel des instructions du programme Pascal :

Génération des instructions de calcul et d'accès aux données, et production d'un code correspondant à la linéarisation des structures de contrôle.

Il travaille en une passe, selon une méthode de production de code conduite par une analyse syntaxique descendante [Knu71, Pai72, Wir71c, Amm73, Wir71b].

Ce programme, écrit en Pascal, a été écrit en s'inspirant du compilateur Pascal-P de Zürich (cf. le paragraphe sur les implémentations de Pascal). Ainsi les routines d'analyse syntaxique ont-

un travail déjà effectué par ailleurs ; les routines sémantiques ont subi une large réécriture, due aux différences fondamentales existant entre les machines Babel [GoT76] et Pascal-P [NAJ74,Gau74] utilisation d'accumulateurs en place de piles, accès aux données très différents.

La qualité du compilateur Pascal-P, tant en fiabilité qu'en lisibilité, a permis d'envisager et d'effectuer aisément ce travail de réécriture (les méthodes de compilation employées sont succinctement décrites en annexe).

C - MONITEUR D' EXECUTION SUR SOLAR 16

Certaines instructions Babel n'ont pu être traduites
n une séquence de quelques instructions machine : la seconde
asse de compilation en a alors fait des requêtes (instruction
olar SVC : Supervisor Call) qui seront traitées à l'exécution
ar des routines spécialisées du système d'exploitation. C'est
ne manière simple d'étendre le jeu d'instructions de la machine,
ans avoir à en modifier la microprogrammation. Pour ne pas
odifier le superviseur existant, nous introduisons lors de
'exécution de programmes Pascal un moniteur d'exécution traitant
es requêtes supplémentaires (§1), et plus généralement réalisant
'interface nécessaire entre les programmes et le superviseur :

- allocation en mémoire secondaire (§3)
- allocation en mémoire principale, ramasse-miettes (§3)
- ordonnancement des tâches représentant les monolithes (§4)
- sauvegarde en mémoire secondaire des monolithes inactifs, et restauration (§5)
- gestion de l'état de résidence des monolithes (§5)
- création et destruction de monolithes (§6)

uis nous examinerons le découpage du moniteur en tâches exécutées
n manière quasi-parallèle (§7), les problèmes de l'édition de
ens (§8) et du dump symbolique (§9).

1 - LES REQUETES

a) REQUETES SUR MONOLITHES

Les transactions entre monolithes sont de deux types :

- opérations vitales, qui seront détaillées au paragraphe 6, telles que :

  CIVA incarnation

  NIRV désincarnation

  Nous introduisons également une requête XIPE, réincarnation, correspondant à un appel récursif de procédure, pour faciliter la gestion de la mémoire.

- et passage de paramètres. Le langage Pascal distingue quatre sortes de paramètres :

  1. paramètre constante, transmis par valeur en entrée de procédure par une requête PVAL
  2. paramètre variable, dont l'adresse est transmise par une requête PVAR
  3. paramètre procédure, dont le nom est transmis par une requête PMON
  4. valeur résultat de l'exécution d'une fonction, transmis par une requête SFCT.

  A l'entrée d'une procédure, une requête RATI permet la transmission effective des paramètres déjà mémorisés par le moniteur lors des requêtes PVAL, PVAR et PMON. C'est à ce moment qu'ont lieu les recopies d'adresses ou de valeurs dans les zones de données de la procédure.

b) REQUETES D'ACCES AUX DONNEES

Tout adressage d'une variable globale ou paginée donne lieu à une requête : le moniteur d'exécution gère seul et entièrement l'allocation de la mémoire principale aux sections de données des monolithes. Dans une version ultérieure, il est prévu d'utiliser le mécanisme matériel de pagination du Solar 16 pour les accès au segment paginé (ce module matériel n'étant pas disponible actuellement).

Deux requêtes, NEW et DISP, permettent respectivement l'allocation et la désallocation des objets dynamiques dans le segment paginé.

Enfin, les accès aux données locales non paginées ne donnent pas lieu à des requêtes, ayant été entièrement traduits en langage machine. Ainsi les accès les plus fréquents (aux objets locaux) sont compilés, alors que les accès plus rares (aux objets globaux) sont interprétés à l'exécution grâce à des requêtes et sont donc plus lents, pouvant même nécessiter des échanges avec la mémoire secondaire. Les accès aux variables globales au niveau du programme principal sont fréquents, mais interprétés grâce à des requêtes ; ce choix ayant été introduit par notre politique de gestion de la mémoire principale, et par l'utilisation de tâches software. Toutefois ils ne nécessitent pas d'échanges avec la mémoire secondaire, le moniteur gardant à cet effet résidents les segments de donnée du monolithe 0.

c) REQUETES DE SERVICE

Il s'agit ici d'instructions Babel sans équivalent simple en langage machine. Notons succinctement :

- les opérations mathématiques : fonctions trigonométriques, logarithmes...
- les services du superviseur : date et heure...
- les traitements d'erreurs d'exécution : requête BUG.

d) REQUETES D'ENTREE-SORTIE

Un fichier Pascal se représente par un tampon et des requêtes d'accès, qui sont :

FILE	ouverture d'un fichier permanent, déjà connu du système d'exploitation
VEDA	création d'un fichier temporaire
KALI	destruction d'un fichier temporaire
RSET et RWRT	rembobinage en lecture, ou en écriture
GET et PUT	lecture ou écriture d'un tampon
EOF et EOLN	test des conditions fin de fichier, et fin de ligne (pour un fichier de type text)

2 - ENTREES-SORTIES FORMATTEES

Nous autorisons les opérations suivantes sur les fichiers de type text :

en lecture	en écriture
caractère	caractère
entier	entier
réel	réel
ensemble	ensemble
scalaire symbolique	scalaire symbolique
	chaîne de caractères
	saut à la page
saut à la carte suivante	saut à la ligne

Chaque opération de formattage étant effectuée par une procédure ajoutée au programme à la seconde passe de compilation. Les monolithes correspondant à l'activation de ces procédures ne seront ainsi rendus résidents en mémoire principale que lors des formattages effectifs.

3 - OCCUPATION DES MEMOIRES

Un des rôles principaux du moniteur d'exécution est la gestion de l'allocation des mémoires secondaire et principale.

La mémoire secondaire contient le code exécutable des procédures, et à chaque instant la zone nécessaire aux blocs de la pile générale. Les blocs sont remplis segment par segment à mesure des sauvegardes de la mémoire principale, et des allocations d'objets dynamiques par la requête NEW. La saturation de cette mémoire secondaire ne doit jamais se produire.

La mémoire principale contient au moins :

- le superviseur résident
- les tables du superviseur

et dans la zone réservée au système Pascal

- le moniteur d'exécution
- les segments de donnée du monolithe 0 (programme principal)
- le monolithe actif
- éventuellement des monolithes inactifs, si l'espace mémoire est suffisamment vaste.

Les monolithes inactifs (procédures dont l'exécution est suspendue par l'appel d'une autre procédure) ne sont en effet pas systématiquement sauvegardés dans la pile générale. La sauvegarde, segment par segment, est décidée par le moniteur lorsque l'espace libre en mémoire principale devient insuffisant pour l'incarnation d'un nouveau monolithe. Des sauvegardes anticipées sont également décidées lorsqu'aucun échange entre mémoires n'est

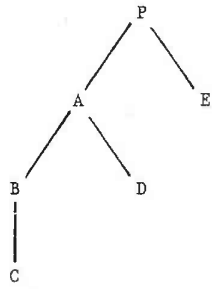
n cours, afin de faire fonctionner en simultanéité les échanges
vec la mémoire secondaire et l'exécution du code généré : l'on
eut ainsi espérer approcher d'une gestion optimale de la mémoire
rincipale, et obtenir une adaptation automatique de la résidence
es monolithes à la taille de la mémoire.

4 - ORDONNANCEMENT DES TACHES ASSOCIEES AUX MONOLITHES

a) REPRESENTATION DES MONOLITHES POUR L'ORDONNANCEUR

Comme nous l'avons vu précédemment (§ II.A.7), les monolithes se représentent sur Solar 16 par des tâches software. Chaque monolithe possède un nom qui est un entier positif. Ces noms, affectés en séquence dans l'ordre croissant, au fur et à mesure des appels de procédure, et restitués en fin d'activation, schématisent l'empilement des monolithes. On peut remarquer qu'à un instant donné, le nom de monolithe le plus élevé correspond au nombre d'activations de procédure en cours.

Ainsi soit par exemple un programme où l'arborescence des déclarations de procédures a la forme



pour les appels de procédure

P → A → B → B → B → C appels

← ← retours

→ C appel

les noms des monolithes correspondant aux procédures seront

procédure	nom du monolithe
P	1
↓	
A	2
↓	
B	3
↓	
B	4
↓ ↑ ↓	
B C	5
↓ ↑	
C	6

Pour s'activer, un monolithe est représenté par une tâche software esclave. Ces tâches sont numérotées dans l'ordre des priorités décroissantes et leur nombre est limité (le scheduler microprogrammé peut en gérer 128 au maximum). Dans le sous-système Pascal, les numéros de priorité des tâches représentant des monolithes prendront leurs valeurs dans l'intervalle $[\alpha,\beta]$ avec

$$0 < \alpha < \beta \leq 127$$

L'affectation de niveaux de priorité software aux monolithes est réalisé selon le principe suivant :

monolithe 1 ⟶ tâche α
monolithe 2 ⟶ tâche $\alpha + 1$
... ...
monolithe $\beta-\alpha+1$ ⟶ tâche β

Quand le parcours dans l'arborescence des déclarations conduit à un numéro de monolithe supérieur à $\beta-\alpha+1$, il y a déclanchement d'une opération de réinitialisation, qui consiste à réaffecter les numéros de tâches, sans modifier l'état de résidence des monolithes. Les monolithes non résidents ne sont alors plus représentés par des tâches software. Dans la pratique, cette opération ne se produit qu'en cas de programmes récursifs.

b) RESIDENCE ET FILES D'ORDONNANCEMENT DES MONOLITHES

Les sauvegardes en mémoire secondaire étant faites au niveau des segments d'un monolithe, le moniteur doit connaître à tout moment l'état de résidence des monolithes.

Pour chaque monolithe, cinq bits appartenant à des files du système, indiquent l'état de la tâche associée :

- le bit a précise si l'exécution de la tâche a été demandée (c'est la file armed du scheduler du Solar 16) ;
- le bit r est à 1 si la tâche est entièrement résidente (c'est la file ready du scheduler) ;
- le bit c indique que le segment de code est présent en mémoire ;
- le bit t indique si la place libérée par la sauvegarde de la zone des données des monolithes a été récupérée par le ramasse-miettes.

De plus, afin de mémoriser l'état des sauvegardes anticipées qui sont décidées dynamiquement par le moniteur, la file r est doublée par la file r' :

- le bit r' indique si la zone des données a déjà été sauvegardée sur la pile générale.

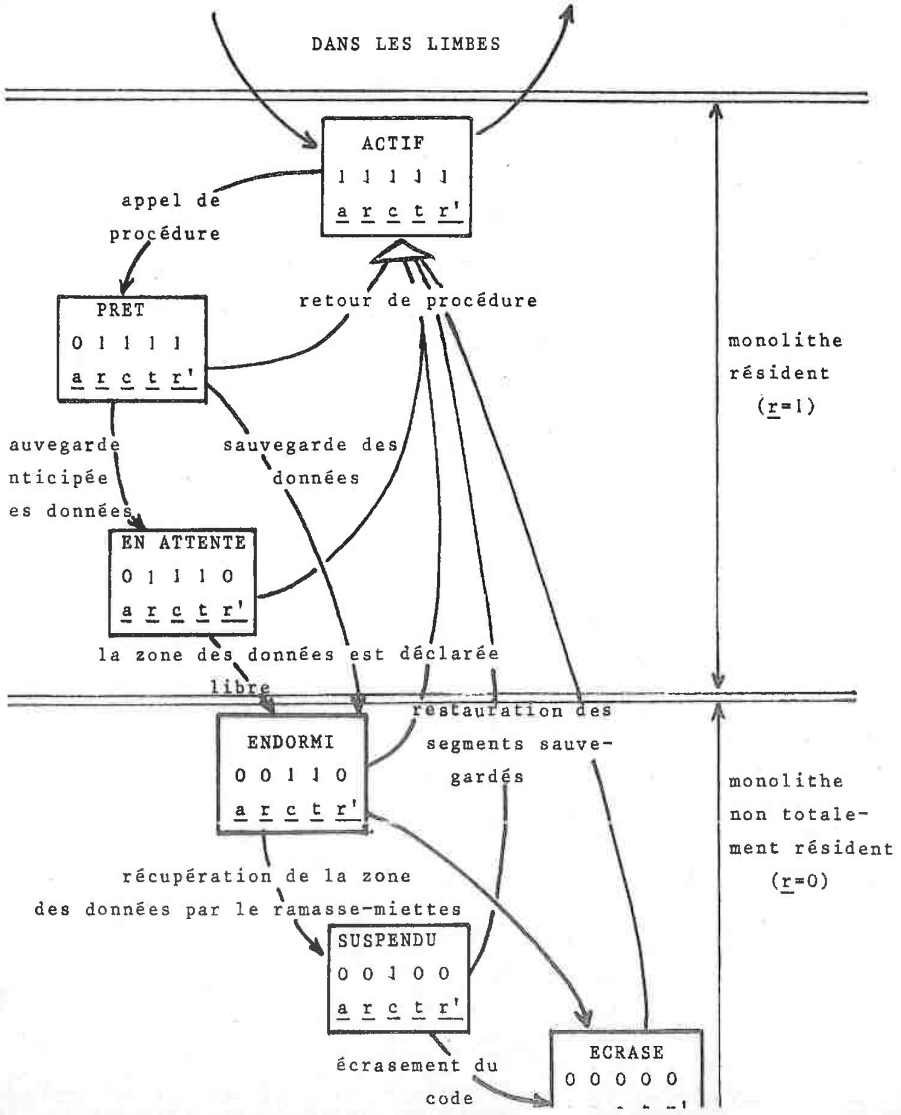
Notons que l'algorithme d'ordonnancement, sous-jacent à l'usage de ces files, utilise deux des trois files du scheduler microprogrammé, les files a et r : ce microprogramme effectuera rapidement une part importante du travail d'ordonnancement en

- recherchant la première tâche dont le bit a est à 1 ;
- puis en activant soit cette tâche si elle est entièrement résidente (bit r à 1), soit une tâche spécialisée du moniteur (tâche software 0) dont le rôle sera de recharger en mémoire principale le ou les segments sauvegardés (selon l'état des files c, t et r'), puis de rendre la main de nouveau au scheduler.

La troisième file du scheduler, enabled, correspondant à la synchronisation de tâches quasi-parallèles, n'est pas utilisée dans ce contexte où le monolithe actif s'exécute seul. Sa mise en oeuvre n'est à envisager que pour d'éventuels développements ultérieurs, tels qu'une implantation du langage SIMONE.

- TRANSITIONS D'ETAT DE RESIDENCE DES MONOLITHES

Connaissant la spécialisation des cinq files du système, es transitions d'états possibles pour un monolithe deviennent alors es suivantes :



Avec pour signification des états :

Dans les limbes. Le monolithe n'existe pas ; le code de la procédure correspondante se trouve dans un fichier du moniteur.

Actif (1 1 1 1 1). Le monolithe est en cours d'exécution ; entièrement résident en mémoire principale, il a le contrôle de l'unité centrale.

Prêt (0 1 1 1 1). Le monolithe a exécuté une requête appel de procédure, et a perdu le contrôle de l'unité centrale (a=0). Entièrement résident, son exécution sera reprise sans délai à la désincarnation du monolithe appelé.

En attente (0 1 1 1 0). Mêmes propriétés que l'état prêt : la zone des données a été sauvegardée sur la pile générale (r'=0) par une décision à titre préventif du moniteur, mais la zone allouée de mémoire principale n'a pas été déclarée libre (r=1) ; le retour éventuel vers l'état actif s'effectue sans délai. En cas de saturation de la mémoire principale, la transition vers l'état endormi s'effectue sans nouvel échange avec la mémoire secondaire; ce type de transition revêt en effet un caractère urgent, qu'il est possible de satisfaire grâce à l'usage de l'état en attente.

Endormi (0 0 1 1 0). La zone des données du monolithe étant sauvegardée sur la pile générale (r'=0), l'espace ainsi libéré en mémoire principale est rendu au moniteur, mais n'est nullement désalloué. Le moniteur peut ensuite, soit effectivement désallouer cet espace et le récupérer par

le ramasse-miettes en cas de saturation de la mémoire principale, soit l'affecter à un autre monolithe représentant la même procédure : le code et les constantes de la procédure n'auront pas à être chargés depuis la mémoire secondaire puisque déjà résidents ; toutefois les données ne sont alors plus celles du monolithe endormi, et auront à être rechargées. D'où la nécessité de déclarer au scheduler ce monolithe non résident (r=0).

Suspendu (0 0 1 0 0). Après que la zone des données du monolithe aie été sauvegardée sur la pile générale (r'=0), l'espace libéré en mémoire principale a été déclaré récupérable par le ramasse-miettes, et n'appartient alors plus au monolithe (t=0 , et donc r=0) Le retour vers l'état actif nécessitera une réallocation de mémoire, puis un rechargement des données.

Ecrasé (0 0 0 0 0). Après sauvegardes sur la pile générale (r'=0), toutes les zones de mémoire principale ont été désallouées (t=c=0, donc r=0 également).

6 - INCARNATION, REINCARNATION ET DESINCARNATION

Nous pouvons désormais préciser l'effet des requêtes CIVA, XIPE et RAMA.

L'appel d'une procédure se fait par la requête CIVA. Comme il s'agit à ce moment d'une incarnation de monolithe, l'instruction "arm" du Solar 16 ne peut être utilisée directement, car il y aurait lancement de la tâche complément de scheduling qui tenterait de restaurer le monolithe à partir de la pile générale.

Trois cas peuvent se présenter :

- la procédure appelée n'a encore jamais été utilisée : c'est une incarnation normale de monolithe, et tous ses segments doivent être initialisés.

- la procédure a déjà été exécutée entièrement, et les segments de code et de constantes sont encore résidents en mémoire. Si l'espace libre nécessaire aux données leur est contigü, on réutilise la même zone de mémoire ; sinon, on déplace les segments de code et de constantes, en réservant une zone pour les données.

- la procédure a été interrompue pendant son exécution. Elle est donc représentée par un monolithe dont l'état de résidence est donné par les files du système. Si le monolithe est entièrement résident, l'espace mémoire qu'il occupe pourra être directement utilisé après sauvegarde de la zone des données dans

Ensuite, le moniteur alloue un nouveau bloc sur la pile générale, met à jour les tables du système, initialise les bits des files du système pour la tâche appelée, acquitte la tâche appelante (par une instruction "quit"), transmet les paramètres éventuels et active le monolithe (par une instruction "arm").

Pour des appels récursifs de procédure, effectués par la requête XIPE, deux zones de mémoire sont utilisées en bascule, afin d'accélérer le processus d'allocation de mémoire en évitant la sauvegarde systématique de la zone des données au début du traitement de la requête. Le travail du moniteur se poursuit ensuite comme pour un appel simple.

Une fin de procédure se traduit par une requête RAMA. Le monolithe associé à la procédure qui se termine disparaît. Le moniteur effectue la transmission d'une valeur de retour s'il s'agissait d'une fonction Pascal, désalloue le bloc au sommet de la pile générale, mémorise l'état de résidence des segments de code et de constantes, et réactive le monolithe interrompu.

7 - DECOUPAGE DU MONITEUR EN TACHES QUASI-PARALLELES

Nous avons vu que le moniteur d'exécution décidait et effectuait des sauvegardes en mémoire secondaire, l'espace ainsi libéré en mémoire principale étant rendu au gestionnaire de mémoire. Cette pratique conduit à un émiettement de la mémoire libre n'en permettant pas la réutilisation immédiate. Un programme, appelé ramasse-miettes, est chargé de la translation des zones occupées, afin de juxtaposer et fusionner les espaces libres.

Pendant les attentes dues aux échanges avec la mémoire secondaire, la tâche ramasse-miettes[1] est activée. Elle effectue la réorganisation de l'espace occupé à partir de la carte de la mémoire fournie par les registres SLO-SLE (adresses origine et fin de tâche) des tâches connues du moniteur.Les zones de données qui ont été libérées après sauvegarde, sont remises dans l'espace libre et certains segments de code résidents sont déplacés, afin de rendre les zones libres adjacentes. Cette politique d'activation du ramasse-miettes pendant les "temps morts" en rend le temps de fonctionnement peu préjudiciable au temps total d'exécution des programmes Pascal, et permet l'utilisation d'un algorithme de compactage très simple.

(1) Un ramasse-miettes est au logiciel ce qu'un garbage collector est au software.

D'autres tâches du moniteur réalisent les "compléments de scheduling" (c'est-à-dire les rechargements de segments à partir de la pile générale), les sauvegardes anticipées et les entrées-sorties. Une autre tâche est spécialisée dans la récupération d'erreurs d'exécution du programme ou de fonctionnement du moniteur, le dépouillement des tables du système, la remontée des chaînages dynamiques et statiques, l'édition de "post-mortem-dumps symboliques" et de statistiques.

Toutes ces tâches sont exécutées de manière quasi-parallèle.

L'allocation des priorités à ces tâches est la suivante, par ordre de priorité décroissante :

- complément de scheduling
- entrées-sorties
- récupération d'erreurs
- sauvegardes anticipées
- tâches utilisateur (affectées aux monolithes, une seule est active à la fois)
- ramasse-miettes : tâche toujours activable.

8 - COMPILATIONS SEPAREES

Dans un système de programmation orienté vers l'utilisateur, il apparaît important de donner la possibilité de compiler séparément, et aussi d'utiliser des modules écrits en d'autres langages que Pascal (bibliothèques Fortran, routines en assembleur). Nous autorisons donc des compilations de procédures séparément, en imposant toutefois les règles suivantes, pour des raisons de méthodologie (une procédure n'est pas un module):

- les seuls objets accessibles dans une procédure compilée séparément sont les objets locaux et les paramètres.

- les déclarations de paramètres formels et effectifs doivent se correspondre (une vérification syntaxique sera réalisée lors de l'édition de liens).

- la déclaration d'une routine qui sera compilée séparément doit comporter l'indication du type de l'interface à réaliser par le compilateur : pascal, fortran ou asm.

- les routines compilées séparément en d'autres langages que Pascal ne doivent pas perturber le fonctionnement du moniteur, par des entrées-sorties intempestives, ou par la création de tâches par exemple.

Le mode d'exécution des programmes permet d'envisager une édition de liens reportée à la phase initiale de chaque exécution. En effet, l'édition de liens consiste à créer à partir du code généré pour chaque procédure un fichier à accès direct où un article contient le corps de la procédure de même nom interne. Le faible travail nécessité par une telle édition de liens rend envisageable sa réalisation avant chaque exécution, et non une fois pour toute ; l'édition de liens n'est donc plus purement statique.

9 - POST-MORTEM-DUMP SYMBOLIQUE

Toujours dans un but de facilité d'utilisation, un vidage de la mémoire, et une remise sous forme symbolique, est prévu en cas d'erreur d'exécution dans un programme Pascal.

Pour préparer ce dump, un fichier du système doit contenir les renseignements nécessaires : nom en clair des variables, leur localisation en mémoire, et leur type. Ces renseignements ont alors dû être recueillis lors de la première passe de compilation, et complétés dans la seconde passe. Ils sont fournis sous forme de directives du langage Babel (cf. description du langage Babel en annexe).

10 - SECURITE DE FONCTIONNEMENT

Cette sécurité est essentiellement obtenue sur Solar par l'utilisation de tâches software esclaves pour représenter les monolithes : chaque monolithe se voit allouer une zone de mémoire dont il ne peut franchir les bornes sans provoquer d'alarme au moniteur.

Le découpage du moniteur en tâches asynchrones permet également d'en vérifier le bon fonctionnement, et d'isoler ses erreurs.

Enfin, le langage Pascal même facilite la détection d'erreurs de programmation, tant à la compilation qu'à l'exécution.

D - AMORCAGE

Les deux passes du compilateur sont des programmes Pascal tout-à-fait standard. Leur utilisation sur Solar 16 ne pose donc pas de problème particulier, sauf tout de même pour leur première implantation.

A cet effet, nous utilisons le compilateur Pascal existant déjà sur Cii Iris 80 : après écriture des deux compilateurs Solar en Pascal, et compilations successives de ces programmes sur Iris 80, on en obtient une version générée en langage machine Solar, et donc directement utilisable.

Notons par p le langage Pascal
i le langage machine Iris 80
s le langage machine Solar 16
b le langage Babel 19

Un compilateur $\overline{\omega}$ du langage Δ, produisant des programmes en langage ∇, et écrit en langage $\square$, sera noté :

$$\overline{\omega} = \{\Delta \rightarrow \nabla \mid \square\}$$

et une compilation d'un programme Δ, fournissant un programme traduit ∇, sera représentée par :

$$\Delta \; \alpha \; \overline{\omega} \subset \nabla$$

)

Si Δ et ∇ sont également des compilateurs, $\Delta = \{\beta \to \gamma \mid \delta\}$ et $\nabla = \{\delta \to \xi \mid \zeta\}$, alors sur la machine ζ on a la propriété :

$\{\beta \to \gamma \mid \delta\} \alpha \{\delta \to \xi \mid \zeta\} \subset \{\beta \to \gamma \mid \xi\}$.

Il s'agit donc d'obtenir les deux compilateurs Pascal sur Solar 16 $\{b \to s \mid s\}$ et $\{p \to b \mid s\}$, en partant du compilateur Pascal sur Iris 80 $\{p \to i \mid i\}$ et des deux compilateurs Solar en version source $\{b \to s \mid p\}$ et $\{p \to b \mid p\}$.

La séquence de compilations emboitées suivante permet d'obtenir ce résultat :

1. $\underline{\{p \to b \mid p\}} \alpha \underline{\{p \to i \mid i\}} \subset \{p \to b \mid i\}$
2. $\underline{\{p \to b \mid p\}} \alpha \{p \to b \mid i\} \subset \{p \to b \mid b\}$
3. $\underline{\{b \to s \mid p\}} \alpha \underline{\{p \to i \mid i\}} \subset \{b \to s \mid i\}$
4. $\underline{\{b \to s \mid p\}} \alpha \{p \to b \mid i\} \subset \{b \to s \mid b\}$
5. $\{b \to s \mid b\} \alpha \{b \to s \mid i\} \subset \boxed{\{b \to s \mid s\}}$
6. $\{p \quad b \mid b \quad \alpha \{b \to s \mid i\} \subset \boxed{\{p \to b \mid s\}}$

Par cette méthode de transport, on s'affranchit de l'écriture fastidieuse des "bootstraps" successifs, d'où une économie de temps et de moyens importante.

L'utilisation d'une grosse machine pour la première génération d'un logiciel destiné à un miniordinateur, méthode qui tend à devenir d'usage courant, peut se généraliser par la réalisation de compilateurs-croisés : le compilateur produisant du code pour le miniordinateur réside toujours sur la grosse machine. Cette technique peu onéreuse à la réalisation présente cependant l'inconvénient d'exiger la présence d'une grosse machine lors de la compilation de chaque programme. Nous avons jugé cette contrainte incompatible avec certains modes d'utili-

III - résultats

A - POINTS ESSENTIELS

Nous venons de décrire une implantation du langage Pascal sur miniordinateur qui nous a amenés à

- définir un langage de complexité intermédiaire entre Pascal et les langages machine de la classe de calculateurs visée, facilitant le transport du compilateur ;

- définir une technique d'espace virtuel bien adaptée au langage et permettant l'exécution de gros programmes, tels qu'un compilateur, avec une mémoire principale exigüe, et pouvant ainsi mener à de bonnes performances ;

- établir un parallèle entre des notions du langage (procédure), de la machine (tâches, segments) et plus formelles (processus) ;

- concevoir un compilateur en parallèle avec un système d'exploitation, les deux aspects se complétant afin d'améliorer la fiabilité et la performance du produit ;

- utiliser au mieux les microprogrammes évolués d'ordonnancement de tâches, les techniques d'adressage d'une machine.

Ces aspects sont illustrés par la réalisation du compilateur Pascal pour Solar 16, en voie d'achèvement. La réflexion précédente sur la méthodologie d'implantation, la haute qualité du compilateur Pascal P et les qualités du langage Pascal nous ont permis d'entreprendre facilement la réalisation validant cette recherche, et d'en espérer de bons résultats.

B - PERSPECTIVES DE DEVELOPPEMENT

La distribution du compilateur Solar 16 dans les universités et lycées sera assurée par l'intermédiaire de l'IRIA et de la SEMS. A court terme, ce produit devra être adapté au traitement de petits programmes, comportant fréquemment des fautes de syntaxe (profil des programmes de débutants) : génération par le compilateur de code machine absolu, directement chargeable et exécutable, sans utiliser la notion de monolithe. Cette adaptation a été prévue dans la phase de création du compilateur, et devrait donc être réalisable facilement.

D'autres développements jugés probables ont également conditionné partiellement les spécifications techniques du compilateur : exécution des programmes dans un contexte de temps partagé ou de quasi-parallélisme. Le développement futur ayant le plus influencé la conception du compilateur en reste cependant le transport.

C - TRANSPORT DU COMPILATEUR

En vue de transports ultérieurs du compilateur nous allons dans ce paragraphe préciser l'adéquation des caractéristiques de quelques machines courantes à nos hypothèses de travail (machine à mots de 16 bits et à accumulateur, adressage basé, etc...)

Le transport du compilateur Solar sur une autre machine de la même classe s'effectue en :

1) paramétrant convenablement le compilateur première passe : définition des constantes de taille maximale des sections de données et de programme, format des relais d'indirection, indexation, ...

2) réécrivant les routines sémantiques du compilateur seconde passe, afin de générer les séquences de code machine nécessaires pour chaque instruction Babel.

3) traduisant manuellement le moniteur d'exploitation dans un langage disponible sur la machine cible, et en l'adaptant au système d'exploitation.

4) testant le produit obtenu, à l'aide d'un jeu de programmes d'essais réalisés pour l'implantation sur Solar 16.

Notre compilateur se présente ainsi comme une alternative à Pascal-Tronc, pour les miniordinateurs.

a) SEMS Mitra 15 :

. La mémoire : 32K mots de 16 bits ; l'adressage est réalisé au niveau de l'octet.

. Les registres : le contexte d'un programme est représenté par les registres rapides

- P compteur ordinal
- L base locale
- G base générale
- A accumulateur
- E extension
- X index

On retrouve ici la même structure que sur Solar : registre d'index, registres accumulateur et extension.

. L'adressage : deux registres de base L et G permettent un accès direct à deux zones de 256 mots. De même que sur Solar, on distingue des adressages :

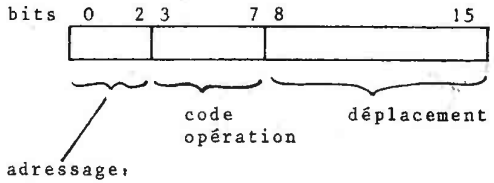
- direct : <base,déplacement>
- indirect : par l'intermédiaire d'un relais placé dans la section de données locale.
- indexé : par l'intermédiaire d'un relais d'indirection.

A la différence du Solar, l'indexation est précisée dans l'instruction, et non dans le relais d'indirection ;

un paramètre du compilateur première passe permet de lever facilement ce problème de transport. Le nombre de bases d'adressage, et leur portée, correspondent à ce qui est prévu pour le Solar ; seule l'interdiction d'adressage indirect par la base générale diffère.

. Les instructions : au nombre de 87, généralement à une adresse, elles se répartissent en :

- 40 avec référence mémoire



000 direct, base locale

001 immédiat : l'opérande est le déplacement

010 direct, base générale

011 indirect, base locale

100 indexé, base générale

101 indexé, base locale

- 29 entre registres

- 12 de décalages

- 6 spéciales

Le code d'ordre comparable à celui du Solar n'a cependant pas la même richesse (pas d'opérations sur bits, zones mémoire, ...).

. Les sous-programmes : les instructions d'appel CLS et de retour RTS de sous-programme, et d'appel CSV et de retour RSV de routine du superviseur réalisent la sauvegarde et la réinitialisation de l'adresse de retour et du registre de base locale, au début de la zone locale appelée ; la récursivité n'est ici pas réalisée directement, mais l'on retrouve la même structure que sur Solar quant aux appels au superviseur.
Cette machine est donc très proche du Solar. Le transport du compilateur ne pose de difficultés que pour le traitement des tâches, qui sera réalisé en logiciel, d'où un ralentissement de l'exécution des programmes et sans doute une mauvaise protection inter-tâches.

b) *Hewlett Packard 2100* :

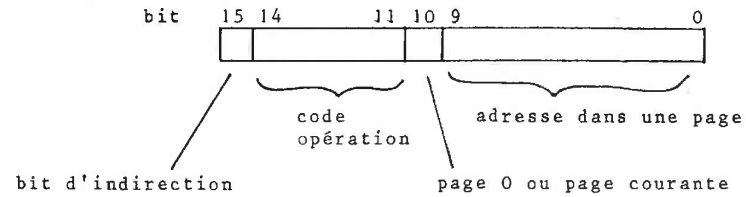
. La mémoire : 32K mots de 16 bits, l'adressage est réalisé au niveau du mot.

. Les registres : en sus du compteur ordinal P, on dispose de deux accumulateurs A et B complètement indépendants pour les opérations arithmétiques, et qui peuvent être liés pour des opérations de décalage ou d'autres opérations sur 32 bits. Ces accumulateurs sont les mots d'adresse 0 et 1, et peuvent être référencés comme tels.

. L'adressage : la mémoire est découpée en pages de 1K mots. Chaque instruction avec référence mémoire peut adresser directement un opérande dans une des deux pages : page courante et page zéro ; la partie adresse occupe donc 10 bits (pour adresser 1K mots). Les objets non adressables directement, sont accessibles par l'intermédiaire d'un relais d'indirection, placé dans l'une des pages référençable; cette indirection peut se faire en cascade : tout relais dont le premier bit est à 1 référence de nouveau un relais d'indirection.
L'adressage est donc très différent de celui du Solar. En particulier l'absence d'indexation peut créer des difficultés pour un éventuel transport.

- Les instructions : les 80 instructions peuvent se répartir en :

  - référence mémoire :



  - entre registres : décalages et opérations logiques sur les accumulateurs et le "carry"

  - entrées-sorties

  - arithmétique longue.

- Les sous-programmes : l'instruction JSB d'appel de sous-programme provoque le rangement de l'adresse de retour dans le premier mot du programme appelé. Sans distinction mode maître/mode esclave, il n'y a pas d'instruction d'appel de routine du superviseur, ni de dispositif de protection de mémoire maître.

Le HP 2100 apparaît donc beaucoup moins sophistiqué que le Solar 16, et d'architecture suffisamment différente pour poser des problèmes de transport.

2

c) *Texas Instrument 990/10* :

Il s'agit du haut de gamme d'une série de machines compatibles s'étendant jusqu'au microprocesseur mono-puce TMS9900.

- La mémoire : 32K mots de 16 bits sont directement adressables par octet ou par mot. Un dispositif de mapping permet l'extension de la mémoire à 1024 K (adresse sur 20 bits).

- Les registres : à chaque tâche ou sous-programme est attaché un "workspace" de 16 registres :

  R0 compteur de décalages
  R1 à R10 accumulateurs, pouvant tous servir d'index
  R11 adresse de retour de sous-programme
  R12 servant aux entrées-sorties
  R13 à R15 servant de lien pour les changements de contexte :
    R13 pointeur de "workspace" WP
    R14 compteur ordinal PC
    R15 registre d'état ST

9 registres peuvent donc être utilisés comme accumulateur, pointeur ou index.

- L'adressage :
  . adressage de registre : le registre contient l'opérande
  . adressage indirect de registre : le registre contient l'adresse de l'opérande
  . adressage direct : le mot suivant l'instruction contient l'adresse de l'opérande
  . adressage indexé : le mot suivant l'instruction contient l'adresse de base, et un registre contient l'index
  . adressage immédiat : le mot suivant l'instruction contient l'opérande
  . adressage avec autoincrémentation : un registre contient l'adresse de l'opérande. Après exécution de l'instruction, le registre est incrémenté.

La richesse des adressages peut permettre la simulation d'adressages basés, directs, indirects et post-indexés, en débanalisant certains registres.

- Les instructions : Au nombre de 69, dont 34 avec adressage, plus 16 instructions étendues (XØP - équivalent de SVC sur Solar) elles sont en général à deux adresses. Sans que l'on retrouve toute la richesse du code Solar, elles permettent une génération de code compact.

- Les sous-programmes : leur appel par l'instruction BLWP provoque un changement de contexte (10 µs) : les registres WP et PC (adresse de retour) sont affectés par un vecteur de transfert, leurs anciennes valeurs, ainsi que celle du registre d'état, sauvegardées dans les registres 13 à 15 du nouveau "workspace". La récursivité n'est donc pas réalisée directement, et ce changement de contexte s'apparente à celle du Mitra 15, en étant plus rapide.

Ces particularités, et l'usage du mapping (trois registres de base, et trois registres de limite) permettent d'envisager un transport assez rapide du compilateur.

d) *Digital Equipment Corporation PDP 11/45* :

La gamme PDP 11 s'étend jusqu'à un microprocesseur LSI 11 compatible.

- La mémoire : jusqu'à 124 K mots de 16 bits, adressable par octet ou mot. L'extension d'adressage est réalisée par pages, de taille variable (≤ 4K mots).

- Les registres : au nombre de 16, ils peuvent être utilisés comme accumulateur, pointeur, index et pointeur avec autoincrémentation ou décrémentation. Leur affectation est la suivante :

  R0 à R5 registres banalisés. Il en existe deux jeux ; le jeu en cours d'utilisation étant déterminé par le mot d'état du programme

  R6 pointeur de pile SP. Il en existe simultanément trois, correspondant aux trois modes de fonctionnement de la machine

  R7 compteur ordinal PC.

Douze registres peuvent donc être utilisés, comme sur TI 990, comme accumulateur, pointeur ou index.

- L'adressage :

  . Direct, avec registre : le registre contient l'opérande

  avec autoincrémentation : le registre contient

l'adresse de l'opérande, et sera incrémenté

avec autodécrémentation : le registre contient l'adresse de l'opérande, après décrémentation

indexé : le registre contient un index, ajouté à l'adresse de base prise dans le mot qui suit l'instruction

. Indirect : mêmes modes d'adressage, avec un niveau d'indirection supplémentaire.

Les registres PC et SP pouvant être utilisés au même titre que les registres banalisés, l'on obtient une grande richesse d'adressages, permettant de simuler ceux du Solar 16.

- Les instructions : peu nombreuses, souvent à deux adresses, et d'un seul type elles permettent cependant la génération d'un code très compact.

- Les sous-programmes : les instructions d'appel JSR, et de retour RTS empilent et dépilent l'adresse de retour, et permettent directement la récursivité.

Toutes les adresses dans un programme étant relatives au compteur ordinal courant, le code est translatable directement.

Ces caractéristiques, ajoutées à la pagination et à la séparation des instructions et des données (8 pages pour chaque), permettent d'envisager un transport assez rapide du compilateur.

;8

e) *Data General Nova* :

C'est une gamme compatible comprenant les miniordinateurs Nova2, Nova3, Supernova, Nova1200, Nova800, Supernova SC et le microprocesseur Micro-Nova.

- La mémoire : 32K mots de 16 bits, adressée au niveau du mot.

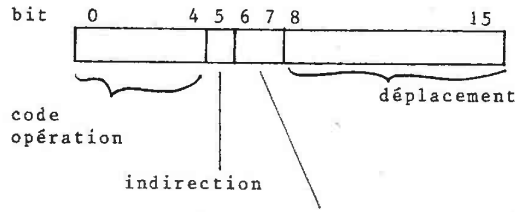
- Les registres : en plus du compteur ordinal PC et du report, il existe quatre accumulateurs, dont deux peuvent servir d'index.

- L'adressage : la mémoire est divisée en pages de 256 mots ; quatre zones sont directement accessibles : la page courante, la page zéro et deux zones par les registres AC2 et AC3. Dans la page courante les registres d'index et PC permettent d'effectuer des adressages indexés.

Les indirections éventuelles peuvent se faire en cascade.

Les mémoires 20_8 à 27_8 sont à autoincrémentation, 30_8 à 37_8 sont à auto-décrémentation.

- Les instructions : assez compactes, elles permettent une écriture artistique. Le format des instructions avec référence mémoire est le suivant :



00 page 0

01 déplacement relatif au compteur ordinal

10 déplacement relatif à la base AC2

11 déplacement relatif à la base AC3

L'adressage basé est donc ici directement possible.

- Les sous-programmes : l'instruction d'appel JMP sauvegarde l'adresse de retour dans le registre AC3 : la récursivité n'est pas directement réalisée.

Cette machine, de structure assez simple, permet cependant une grande variété d'adressages et d'opérations. Le transport du compilateur semble donc relativement rapide, malgré l'absence de tâches.

conclusion

La définition et la réalisation de ce compilateur furent menées de manière inhabituelle pour ce type de travail. Cela tient aux points suivants :

Le langage implanté, Pascal, est adapté à une compilation aisée et performante, malgré sa puissance de description d'algorithmes et de données ; les choix de N. Wirth lors de sa définition en facilitent l'analyse par une méthode descendante récursive en une passe, correspondant à la structure même du langage.

Le langage d'écriture du compilateur, Pascal, a permis la réalisation d'un compilateur par U. Ammann, performant, fiable et très lisible, que nous avons pu réutiliser au prix d'un effort minime.

La définition d'un langage de complexité intermédiaire entre Pascal et les langages machine de miniordinateurs, Babel, et son utilisation dans notre compilateur, permettent d' en envisager le transport sur toute une gamme de machines.

L'analogie que nous avons remarquée entre les notions de procédure, de processus et de tâches a mené à l'utilisation intensive de l'architecture du Solar 16, garantissant les performances du compilateur, et à la levée des restrictions habituelles concernant la taille des programmes exécutables, satisfaisant ainsi notre volonté de réaliser un produit réellement et facilement utilisable.

Enfin nous avons essayé d'illustrer la nécessité de concevoir en parallèle les compilateurs et les systèmes d'exploitation.

Il nous semble que cette recherche peut se poursuivre vers la portabilité des logiciels d'exploitation d'une machine, et vers la définition d'un jeu d'instructions et d'une architecture pour mini ou micro-ordinateurs adaptés aux langages procéduraux.

bibliographie

Amm73 U.AMMANN - The Method of Structured Programming Applied to the Development of a Compiler - International Computing Symposium 1973

BKL76 J.BEZIVIN, W.H.KAUBISCH, A.LEROY, J.L.NEBUT, R.RANNOU - Simone : manuel de référence - IRISA Rennes (Novembre 76)

BrD76 C.BRON, W.DE VRIES - A Pascal Compiler for PDP 11 Minicomputers - Software Practice and Experience, 6, 109-116 (76)

Bri73 P.BRINCH HANSEN - Operating Systems Principles - Prentice Hall (73)

Bri75 P. BRINCH HANSEN - The Programming Language Concurrent Pascal - Software Engineering, 1, 2, 199-207 (Juin 75)

Bri76 P.BRINCH HANSEN - Concurrent Pascal Report ; Sequential Pascal Report ; The Solo Operating System ; A Real-Time Scheduler ; Concurrent Pascal Machine ; Concurrent Pascal Implementation Notes ; The Job Stream System - Caltech (Juin 75 - Janv.76)

CCK76 J.L.CHEVAL, F.CRISTIAN, S.KRAKOWIAK, M.LUCAS, J.MONTUELLE, J.MOSSIERE - Un système d'aide à l'écriture des systèmes d'exploitation - Congrès AFCET 76, 625-634 (Nov. 76)

GZ76 R.CONWAY, D.GRIES, E.C.ZIMMERMAN - A primer on Pascal - Winthrop publishers (76)

ii75 CII - The System Implementation language LIS, Reference Manual-(Juillet 75)

on63 M.E.CONWAY - Design of a Separable Transition Diagram Compiler - CACM 6, 7, 396-408

ro75 CROCUS (nom collectif) - Systèmes d'exploitation des ordinateurs - Dunod (75)

er75 J.C.DERNIAME- Le projet CIVA, un système de programmation modulaire - Thèse, Nancy (75)

HD72 E.W.DIJKSTRA, C.A.R.HOARE, O.DAHL - Structured Programming - Academic Press (72)

ij68 E.W.DIJKSTRA - Goto Statement Considered Harmful - CACM, 11, 3 (Mars 68)

ij73 E.W.DIJKSTRA - The Humble Programmer - CACM 15, 10 (Oct.73)

aG75 J.FARRE, M.GAUTHIER - Transplantation par optimisations successives d'un compilateur Pascal - Journées Pascal, Universités de Nice et Paris VI (Juin 75)

or75 R.FORTIER - Etude et définition du langage intermédiaire et d'une machine formelle multiprocesseurs orientée vers l'exécution du langage Pascal - Thèse de 3ème cycle, INPG

Gau74 M.GAUTHIER - Le P-code Pascal - Institut de Programmation Paris VI (Juin 74)

G1N76 C.O.GROSSE-LINDEMANN, H.H.NAGEL - Postlude to a PASCAL Compiler Bootstrap on a DEC System 10 - Software Practice and Experience, 6, 29-42 (76)

GoH76 G.GOOS, J.HARTMAN (éditeurs) - Compiler Construction, An Advanced Course - Springer Verlag (76)

GoT76 E.GODEFROY, A.TISSERANT - Le langage Babel 19 - CRIN 76-R-043, Nancy (Nov.76)

GMT77a E.GODEFROY, P.MERCIER, A.TISSERANT - Directives et instructions du langage Babel 19 - CRIN 77-R-005, Nancy (Fev.77)

GMT77b E.GODEFROY, P.MERCIER, A.TISSERANT - Compilation de Babel sur Solar 16 - CRIN 77-R-036, Nancy (Nov.77)

Gri69 M.GRIFFITHS - Analyse déterministe et compilateurs - Thèse, Université de Grenoble (Oct.69)

Gri71 D.GRIES - Compiler construction for digital computers - Wiley (71)

Hab73 A.N.HABERMAN - Critical Comments on the Programming Language Pascal - Acta Informatica 3, 47-57 (73)

HoW73 C.A.R.HOARE, N.WIRTH - An Axiomatic Definition of the Programming Language Pascal - Acta Informatica 2, 335-355 (73)

Inf 72 High Level Languages - Infotech State of the Art Report 7 (72)

Inf 76 Structured Programming - Infotech State of the Art Report (76)

JeW74 K.JENSEN, N.WIRTH - A User Manual for Pascal - ETH Zürich (Avril 74)

Knu71 D.E.KNUTH - Top-down Syntax Analysis - Acta Informatica, 1,2, 79-110 (71)

Lec74 O.LECARME - Structured Programming, Programming Teaching and the Language Pascal : a Bibliography - Université de Montréal (74)

Lec75 O.LECARME - Le langage Pascal comme outil d'écriture de programmes transportables - Journées Pascal, Universités de Nice et Paris VI (Juin 75)

Lec77 O.LECARME - Development of a Pascal Compiler for the CI IRIS 50. A partial History - Pascal Newsletter 8, 8- (Mai 77)

LeD74 O.LECARME, P.DESJARDINS - Reply to a paper by A.N. Haberman on the programming language Pascal - Sigplan Notices, 9, 10, 21-27 (Oct. 74)

LeD75 O.LECARME, P.DESJARDINS - More comments on the programming language Pascal - Acta Informatica, 4, 231-243 (75)

LeS68 P.M.LEWIS, R.E.STEARNS - Syntax directed Transduction - JACM 15, 3, 465-88 (68)

MaT74 P.MANCEL, D.THIBAULT - Transport d'un compilateur Pascal,écrit en Pascal, d'un CDC 6400 sur un CII IRIS 80 - Thèse de Docteur-Ingénieur, Institut de Programmation, Paris VI (Sept.74)

NAJ74 K.V.NORI, U.AMMANN, K.JENSEN, H.H.NAGELI - The Pascal <P> Compiler : Implementation Notes - ETH Zürich (Déc.74)

Nau63 P.NAUR (ed) - Revised Report on the Algorithmic Language Algol 60 - CACM 6, 1-17 (Janv. 63)

Pai72 C.PAIR - Cours de compilation - Ecole d'Eté d'Informatiqu de l'AFCET à Neuchâtel, Masson (72)

PUG Pascal News - Pascal User's Group, G.RICHMOND, puis A. MICKEL éditeurs, Université du Minnesota

RaR64 B.RANDELL, L.J.RUSSEL - Algol 60 Implementation - Academic Press (64)

uS76 D.L.RUSSEL, J.Y.SUE - Implementation of a Pascal Compiler for the IBM 360 - Software Practice and Experience, 6, 371-376 (76)

ch75 M.SCHWAAB - Implémentation de Pascal sur mini-ordinateur T1600 : le moniteur d'exécution - Rapport de DEA, INPL Nancy (Mai 75)

cT75 M.SCHWAAB, A.TISSERANT - Implémentation de Pascal sur miniordinateur T1600 - Journées IRIA sur la recherche en miniinformatique (Oct. 75)

GP Bulletin de liaison du sous-groupe Pascal - Groupe programmation et langages de l'AFCET, O. LECARME éditeur, Université de Nice

e175 Manuel de présentation série Solar 16 - Télémécanique informatique (75)

en75 R.D. TENNENT - Pasqual : a proposed generalization of Pascal - Queen's University, Kingston, Ontario, Canada (Fév. 75)

is76 A.TISSERANT - Présentation d'un projet de compilateur Pascal pour miniordinateurs - CRIN Nancy R-76-021 (Juin 76)

eQ71 J.WELSH, C.QUINN - A Pascal Compiler for the ICL 1900 Series Computers - Software Practice and Experience, 1, 309-333 (71)

Wir71a N.WIRTH - The Programming Language Pascal - Acta Informatica 1, 35-63 (71)

Wir71b N.WIRTH - The Design of a Pascal Compiler - Software Practice and Experience, 1, 309-333 (Oct. 71)

Wir71c N.WIRTH - Program development by stepwise refinement - CACM 14,4 (1971)

Wir72 N.WIRTH - The Programming Language Pascal (Revised Report) - ETH Zürich (Nov. 72)

Wir73a N.WIRTH - Systematic Programming - An Introduction - Prentice Hall (73)

Wir73b N.WIRTH - From Programming Techniques to Programming Methods - International Computing Symposium 73

Wir75 N.WIRTH - An assessment of the programming language Pascal - Software Engineering, 1, 2, 192-198 (75)

Wir76a N.WIRTH - Pascal S : A Subset and its Implementation - ETH Zürich (76)

Wir76b N.WIRTH - Algorithms + Data Structures = Programs - Prentice Hall (76)

Wir76c N.WIRTH - Modula : A language for modular multi-programming ETH Zürich (Mars 76)

annexes

TABLE DE CONTEXTE

ANALYSE LEXICOLOGIQUE :

La procédure insymbol effectue toutes les entrées de texte source. Elle lit caractère par caractère un symbole de base du langage (mot clé, identificateur, constante numérique, symbole spécial...) et le compare éventuellement aux tables du compilateur pour en déterminer la nature et la signification. Etant seule à accéder au texte source, cette procédure est également chargée du traitement des blancs non significatifs, des commentaires, des options de compilation, des fins de ligne et aussi de la mise en page de la liste du programme compilé.

COMPILATION DES DECLARATIONS D'IDENTIFICATEURS ET DE TYPES :

A chaque niveau d'imbrication statique de procédure, repéré par la variable top, les identificateurs déclarés locaux sont rangés dans un arbre binaire, de façon à être triés alpha-numériquement par un parcours en ordre symétrique ; les racines de ces arbres étant repérées par la variable tableau display :

```
var top : 0 .. 20 ;
    display : array [0..20] of record
                    fname : ↑identifier;{pointeur sur l'identificateur}
                    case occur:(blck,crec,vrec,rec) of
                         crec:(clev:0..maxlevel;
                               cdspl:0..32767);
                         vrec:(vdspl:0..32767)
              end;
```

(occur = blck : l'identificateur est un identificateur de variable
crec : l'identificateur est un identificateur de champ dans un record avec adresse constante
vrec : l'identificateur est un identificateur de champ dans un record avec adresse variable.

Les noms d'identificateurs et leur implantation éventuelle en mémoire sont décrits par des agrégats de type identifier :

```
type identifier=record
                name:alfa;
                llink,rlink:↑identifier;
                idtype:↑structure;
                next:↑identifier;
                case klass:idclass of
                  types:( );
                  konst:(values:record case intval:boolean of
                                  true:(ival:integer);
                                  false:(valp:↑constant)
                                  end);
                  vars:(vkind:(actual,formal);
                        vlev:0..maxlevel;
                        vaddr:record
                                segment:(cbase,rbase,pbase,tbase);
                                wordaddr:0..32767;
                                halfaddr:0..1;
                                chouiaaddr:0..15
                              end);
                  field:(fldaddr:record
                                  segment:(cbase,rbase,pbase,tbase);
                                  wordaddr:0..32767;
                                  halfaddr:0..1;
                                  chouiaaddr:0..15
                                end);
                  proc,func:(case pfdeckind:declkind of
                              standard:(key:1..15);
                              declared:(pflev:0..maxlevel;
                                        pfname:integer;
                                        case pfkind:(actual,formal) of
                                          actual:(forwdecl,
                                                  externfor,
                                                  externasm,
                                                  externpas:boolean)))
end;
```


où name est le nom de l'identificateur

llink et rlink sont les liens dans l'arbre binaire des identificateurs de même niveau statique

idtype est un pointeur sur la description du type de donnée associé à l'identificateur

next sert au chaînage des identificateurs décrivant une même structure : liste des identificateurs d'un type scalaire symbolique, de champs dans un record ...

klass est la classe de l'identificateur :

- identificateur de type : on lui associe uniquement la description du type
- identificateur de constante : on lui associe également sa valeur (entière, réelle, ...)
- identificateur de variable : on indique son adresse en mémoire (segment, déplacement en mots, adresse d'octet ou de bit dans un mot)
- identificateur de champ
- identificateur de fonction ou procédure

Description de type de donnée :

```
type structure=record
                size:record accu : (chouia,half,word,double,
                                    multiple,any);
                                z : 0..32767
                    end;
                case form : structform of
                  scalar:(case scalkind:declkind of
                            declared:(fconst:↑identifier);
                            standard:(none:↑structure));
                  subrange:(rangetype:↑structure;
                            min,max:record case intval:boolean of
                                      true:(ival:integer);
                                      false:(valp:↑constant)
                                      end);
                  pointer:(eltype:↑structure);
                  power:(elset:↑structure;
                         add:integer);
                  arrays:(aeltype,inxtype:↑structure);
                  records:(fstfld:↑identifier;
                           recvar:↑structure);
                  files:(filtype:↑structure);
                  tagfld:(tagfieldp:↑identifier;
                          fstvar:↑structure);
                  variant:(nxtvar,subvar:↑structure;
                           varval:record case intval:boolean of
                                    true:(ival:integer);
                                    false:(valp:↑constant)
                                  end)
end;
```


où size indique la taille de la zone de mémoire nécessaire à un objet de type décrit : 1 bit, 1 octet, 1 mot, plusieurs mots (z < 32K), et l'accumulateur permettant éventuellement de manipuler l'objet

puis suivant la forme de structuration (scalaire, intervalle, tableau, ...) des pointeurs renvoient sur la description des objets de niveau inférieur :

- scalar, subrange : liste ordonnée des identificateurs dans un type scalaire ou intervalle

- pointer : type de l'objet pointé

- power : type de l'intervalle sur lequel est construit l'ensemble

- arrays : type de l'indice, et de l'élément de base dans le tableau (un tableau à plusieurs indices est traité comme un tableau de tableau, chacun à un seul indice)

- records : liste des identificateurs de champs, et type de l'agrégat

...

TYPES PREDECLARES :

La description des types de données prédéclarées (integer, real, char, boolean, text, et valeur nil) est accessible par l'intermédiaire des variables globales :

```
var intptr,
    realptr,
    charptr,
    boolptr,
    nilptr,
    textptr : †structure ;
```

IDENTIFICATEURS PREDECLARES :

En début d'exécution du compilateur, les identificateurs prédéclarés (noms des types prédéclarés, noms des procédures et fonctions standards...) sont définis par la création d'agrégats identifier au niveau statique 0. Les identificateurs déclarés dans le programme sont alors introduits à partir du niveau statique 1.

DIRECTIVES ET INSTRUCTIONS BABEL

CHARGEMENTS ET RANGEMENTS

LDCN	Chargement du pointeur nul
LDCC	Chargement immédiat dans C
LDCW	Chargement immédiat dans W
LDCH	Chargement immédiat dans H
LDCM	Chargement immédiat multiple
LODC	Chargement dans C
LODH	Chargement dans H
LODW	Chargement dans W
LODD	Chargement dans D
LODM	Chargement multiple
STOC	Rangement de C
STOH	Rangement de H
STOW	Rangement de W
STOD	Rangement de D
STOM	Rangement multiple

COMPARAISONS

LESC	Inférieur strict sur C
LESH	Inférieur strict sur H
LESW	Inférieur strict sur W
LESD	Inférieur strict sur D
LESM	Inférieur strict sur M
LEQC	Inférieur ou égal sur C
LEQH	Inférieur ou égal sur H
LEQW	Inférieur ou égal sur W
LEQD	Inférieur ou égal sur D
LEQM	Inférieur ou égal sur M
EQUC	Egal sur C

EQUH	Egal sur H
EQUW	Egal sur W
EQUD	Egal sur D
EQUM	Egal sur M
NEQC	Différent sur C
NEQH	Différent sur H
NEQW	Différent sur W
NEQD	Différent sur D
NEQM	Différent sur M
GEQC	Supérieur ou égal sur C
GEQH	Supérieur ou égal sur H
GEQW	Supérieur ou égal sur W
GEQD	Supérieur ou égal sur D
GEQM	Supérieur ou égal sur M
GRTC	Supérieur strict sur C
GRTH	Supérieur strict sur H
GRTW	Supérieur strict sur W
GRTD	Supérieur strict sur D
GRTM	Supérieur strict sur M
TODD	Parité

RUPTURES DE SEQUENCE

GOTO	Branchement inconditionnel
FJMP	Branchement conditionnel
XJMP	Branchement à issues multiples

OPERATIONS MONADIQUES

ABSI	Valeur absolue d'un entier
ABSR	Valeur absolue d'un réel
ATAN	Arc tangente
COS	Cosinus
EXP	Exponentielle
FLOAT	Conversion d'un entier en réel
LOG	Logarithme népérien

NEGB	Not booléen
NEGI	Négation d'entier
NEGR	Négation de réel
SIN	Sinus
SING	Génération d'un singleton
SQRI	Carré d'entier
SQRR	Carré de réel
SQRT	Racine carrée
TRNC	Conversion de réel en entier
CARD	Cardinal
EXPO	Exposant d'un réel

FONCTIONS SPECIALES

TIME	Heure
DATE	Date
CLOK	Temps machine
MESS	Message à l'opérateur
PACK	Compactage
UNPK	Décompactage

OPERATIONS DYADIQUES

ADDI	Addition antière
ADDR	Addition réelle
AND	Et booléen
SUBS	Différence d'ensembles
DIVI	Division entière
DIVR	Division réelle
MULS	Intersection d'ensembles
OR	Union booléenne
MEMB	Appartenance à un ensemble
MOD	Modulo
MULI	Multiplication entière
MULR	Multiplication réelle
SUBI	Soustraction entière
SUBR	Soustraction réelle
ADDS	Réunion d'ensembles

OPERATION IMMEDIATE

INCR	Addition Immédiate

OPERATIONS SUR FICHIERS

FILE	Ouverture de fichier permanent
VEDA	Ouverture de fichier temporaire
KALI	Fermeture de fichier temporaire
RSET	Rembobinage pour lecture
RWRT	Rembobinage pour écriture
GET	Lecture
PUT	Ecriture
EOF	Test fin de fichier
EOLN	Test fin de ligne

OPERATIONS SUR FICHIERS DE TYPE TEXT

RIDC	Lecture d'un caractère
RIDI	Lecture d'un entier
RIDR	Lecture d'un réel
RIDE	Passage à la carte suivante
RIDS	Lecture d'un scalaire
RIDP	Lecture d'un ensemble
RIDL	Lecture d'une chaîne de caractères
WRTC	Ecriture d'un caractère
WRTI	Ecriture d'un entier
WRTE	Fin de ligne
WRTS	Ecriture d'un scalaire
WRTP	Ecriture d'un ensemble
WRTL	Ecriture d'une chaîne de carcatères
PAGE	Saut à la page
CADR	Facteur de cadrage

OPERATIONS SUR MONOLITHES

CIVA	Incarnation
NIRV	Désincarnation

XIPE	Réincarnation
RAMA	Fin des temps
PVAL	Passage de paramètre par valeur
PVAR	Passage de paramètre référence
PMON	Passage de paramètre procédure ou fonction
RATI	Récupération de paramètre
NAME	Construction de relais dynamique
RFCT	Récupération du résultat d'une fonction
SFCT	Transmission du résultat d'une fonction

OPERATIONS SUR LES OBJETS DYNAMIQUES

NEW	Génération
DISP	Destruction

TRAITEMENT D'ERREUR

MANU	Vérification de bornes
BUG	Branchement sur erreur

INDEXATION

INDX	Indexation
LODX	Chargement du registre X

DIRECTIVES

L	Numéro de ligne
:	Définition d'étiquette
END	Fin de procédure
!	Liste d'étiquettes
CNST	Constante
RLAI	Relais d'indirection
TABL	Table
#T	Taille de segment T
#R	Taille de segment R
#P	Taille de segment P

TYPE	Déclaration de type
VAR	Déclaration de variable
PROG	En-tête de procédure
FUNC	En-tête de fonction
VPAR	Définition de paramètre formel valeur
RPAR	Définition de paramètre formel référence
FPAR	Paramètre formel fonction
PPAR	Définition de paramètre formel procédure
PROG	En-tête de programme

REGISTRES, ADRESSAGE ET FORMATS D'INSTRUCTIONS T1600

L'unité d'adressage est le mot de 16 bits. 12 registres rapides (accès en 60 ns) sont accessibles au programmeur :

A – REGISTRE ACCUMULATEUR

A est le registre destination de la plupart des instructions ayant un mot mémoire pour opérande.

A peut être chargé à partir de la mémoire ou rangé en mémoire.

A est un des registres opérandes possibles des instructions registre-registre.

A est le registre qui sert aux échanges avec les périphériques.

A est sauvegardé et restauré lors des changements de contexte.

B – EXTENSION DE L'ACCUMULATEUR

B sert d'extension à l'accumulateur et constitue avec celui-ci un registre 32 bits :

- pour des opérandes arithmétiques en double longueurs (multiplication, division, décalage numérique) ;
- pour des opérandes logiques en double longueurs (décalages logiques et circulaires, opérations sur bits) ;
- pour les opérandes en virgule flottante

B peut être chargé à partir de la mémoire ou rangé en mémoire

B est un des registres opérandes possibles des instructions registre-registre.

B est sauvegardé et restauré lors des changements de contexte.

X – REGISTRE INDEX

X a le rôle d'index dans l'adressage indirect des opérandes mémoire : lorsqu'une constante adresse comporte le bit 0 à 1 les 16 bits de l'index sont ajoutés aux 15 bits poids faible de la constante adresse.

X permet d'indexer les instructions de décalage, d'opération sur bit et l'instruction ARM : Son contenu est ajouté aux bits 11 à 15 (9 à 15 pour ARM) de l'instruction.

X peut être chargé à partir de la mémoire et rangé en mémoire.

X est un des registres opérandes possibles des instructions registre-registre.

X est sauvegardé et restauré lors des changements de contexte.

Y – REGISTRE INTERMÉDIAIRE

Y peut être chargé à partir de la mémoire et rangé en mémoire

Y est un des opérandes possibles des instructions registre-registre.

De préférence aux registres A B et X qui ont des rôles spécifiques et qu'il est souvent intéressant de ne pas modifier, Y est destiné à servir de registre intermédiaire.

Par exemple l'utilisation de Y est conseillée pour :

charger à partir de la mémoire ou ranger en mémoire C, L, W ou K ;

transférer le contenu d'une mémoire dans une autre mémoire ;

faire des opérations arithmétiques, logiques, de comparaison entre la mémoire et un des registres B, X, C, L, W, K sans modifier l'accumulateur.

Y est sauvegardé et restauré lors des changements de contexte.

C – BASE COMMUNE

L – BASE LOCALE

W – BASE DE TRAVAIL

C, W et L ont un fonctionnement identique. Le choix des appelations respectives de ces registres (qui correspond à l'utilisation qui en est faite dans l'assembleur et PLT1600) est arbitraire.

C, L et W sont utilisés en tant que bases dans l'adressage de la mémoire : les 16 bits de ces registres permettant aux instructions à référence mémoire (qui comportent l'indication d'une base et un déplacement de 8 bits) d'accéder à trois zones mémoire de 256 mots chacunes.

C, L et W sont des opérandes possibles pour les instructions registre-registre.

C, L et W sont sauvegardés et restaurés lors des changements de contexte.

K – POINTEUR DE PILE

Les 16 bits de K sont utilisés pour pointer sur une zone mémoire fonctionnant en pile. Les instructions BSR et SVC utilisent cette pile pour sauvegarder le registre P, les instructions RSR et RSV pour le restaurer. De même l'instruction PSR utilise cette pile pour sauvegarder les registres A, B, X, Y, C, L, W, K et l'instruction PLR pour le restaurer.

Les instructions de sauvegarde incrémentent K avant chaque rangement et les instructions de restauration le décrémentent après chaque chargement, de telle manière qu'il pointe toujours sur la mémoire occupée de la pile d'adresse la plus grande. Si la pile est vide K pointe sur la mémoire précédant immédiatement la pile.

K est un des opérandes possibles des instructions registre-registre.

K est sauvegardé et restauré lors des changements de contexte.

Il doit y avoir une pile pour chaque tâche hard ou soft, étant donné que ces tâches peuvent être mises en attente et ne se terminent donc pas toujours dans l'ordre hiérarchique des priorités.

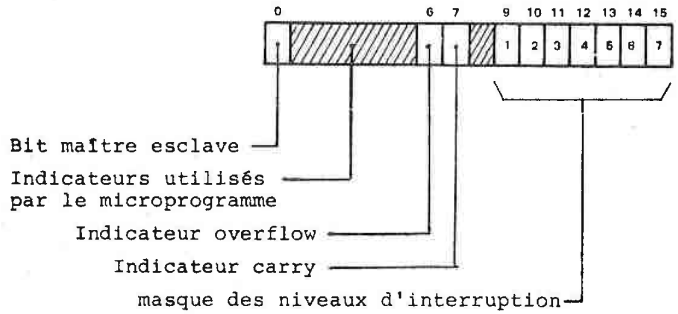
P – POINTEUR D'INSTRUCTION

Les 16 bits de ce registre indiquent l'adresse de la prochaine instruction à exécuter.

P est sauvegardé et restauré lors des changements de contexte.

S – REGISTRE D'ÉTAT

S regroupe différents indicateurs et masques :



Bit maître/esclave :

Quand ce bit vaut 1 le calculateur est en mode maitre (toutes les instructions sont exécutables ; la translation et la protection mémoire sont ineffectives).

Quant ce bit vaut 0 le calculateur est en mode esclave (certaines instructions provoquent une alarme ; la translation et la protection mémoire sont effectives si l'option DRPS existe sur le calculateur). Le bit maitre/esclave est positionné par les instructions SVC et RSV. Le bit maitre/esclave est sauvegardé et restauré avec les bits de l'octet gauche de S, lors des changements de contexte.

Indicateur V et indicateur C :

Ces deux indicateurs sont positionnés par certaines instructions, avec des significations différentes selon l'instruction, et peuvent ensuite être testés par des jumps conditionnels. (Les opérations arithmétiques, les comparaisons, les décalages, certaines instructions sur bits, SIØ, etc. . , positionnent ces deux indicateurs).

L'exécution de l'instruction SCY (qui positionne l'indicateur C à 1 sans modifier l'indicateur V) est le seul cas où un des indicateurs est modifié sans que l'autre le soit.
Dans tous les autres cas ou bien l'état de chacun des deux indicateurs a une signification, ou bien seul l'un des indicateurs a une signification et l'autre est remis à zéro.

L'indicateur V et l'indicateur C sont sauvegardés et restaurés, avec les bits de l'octet gauche de S, lors des changements de contexte.

Masque des interruptions :

Les bits 9 à 15 de S, lorsqu'ils sont mis à 1, masquent sélectivement les interruptions des niveaux 1 à 7. Le niveau hard 0 (alarmes) ne peut pas être masqué.

Les bits 9 à 15 de S peuvent être changés par l'instruction XIMR.

Les bits 8 à 15 de S ne sont ni sauvegardés ni restaurés lors des changements de contexte.

SLO – ORIGINE DE LA PROTECTION MÉMOIRE ET DE LA TRANSLATION

SLE – FIN DE LA PROTECTION MÉMOIRE

SLO contient, sur 16 bits, l'adresse réelle de la mémoire qui est adressée, en tant que mémoire 0, par un programme en mode esclave.

SLE contient, sur 16 bits, l'adresse réelle de la mémoire au delà de laquelle un programme en mode esclave n'a plus accès.

SLO et SLE sont sauvegardés et restaurés lors d'un changement de contexte.

FORMAT DES INSTRUCTIONS - ADRESSAGE

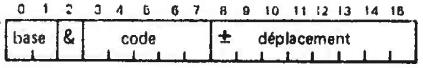
FORMAT DES INSTRUCTIONS

Les instructions peuvent être classées, selon les différents champs qui les composent, en 9 formats différents (à l'exception de PSR, PLR, SVC et ARM qui utilisent chacune un format propre).

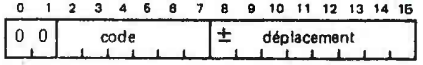
A chacun des différents formats correspond un ou plusieurs modes d'adressage spécifiques.

Notations : base = base C, L ou W
& = indirection
Rsce = registre source
Rdest. = registre destination

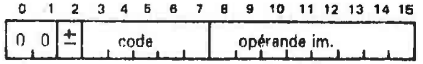
⊕ Instructions avec opérande en mémoire (mot ou octet) adressage basé direct, indirect, indirect post indexé



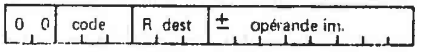
σ Instructions de saut adressage par rapport à P



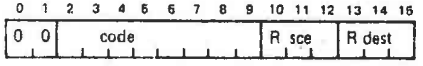
♦ Instructions avec opérande immédiat 9 bits adressage immédiat



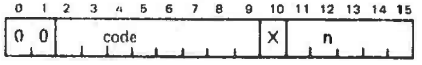
● Instructions avec opérande immédiat 8 bits adressage immédiat



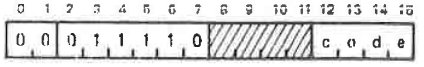
η Instructions registre-registre 8 registres adressables en tant que registres banalisés



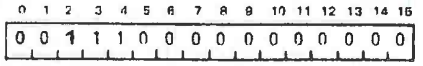
ψ Décalages et opérations sur bits non indexés ou indexés



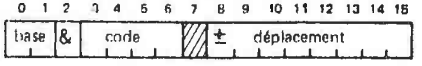
ω Instructions sans indication d'opérande



ζ Instructions virgule flottante avec opérande mémoire, adressage basé direct, indirect, indirect post indexé

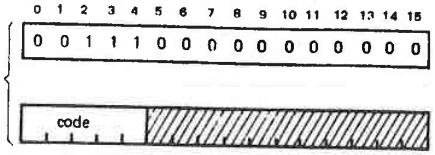


Ces instructions occupent deux mots de 16 bits, le premier mot identifiant l'instruction comme faisant partie de l'extension flottant



Instructions virgule flottante sans opérande de mémoire

Ces instructions occupent deux mots de 16 bits, le premier mot identifiant l'instruction comme faisant partie de l'extension flottant.



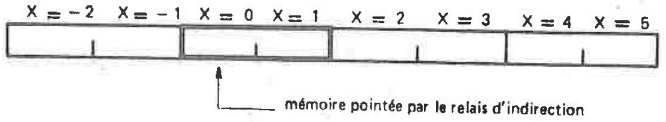
INSTRUCTIONS AVEC OPÉRANDE EN MÉMOIRE : ⊕

Les bits 0 et 1 comportent l'indication d'une base avec le codage indiqué ci-contre. Le bit 2, lorsqu'il est à un, indique l'adressage indirect.

base C	01
L	10
W	11

Les bits 8 à 15 comportent un déplacement compris entre – 128 et + 127. Certaines de ces instructions ont pour opérande un mot, d'autre un octet. Elles utilisent l'adressage direct, indirect et indirect post indexé, avec dans tous les cas d'adressage basé, dont le fonctionnement est le suivant : le déplacement est additionné au contenu du registre de base indiqué par l'instruction, ce qui permet d'avoir accès à chacun des 256 mots dont l'adresse est comprise entre "valeur de la base – 128" et "valeur de la base + 127."

- Adressage direct de mot : l'opérande est le mot mémoire adressé par la base et le déplacement.
- Adressage indirect de mot : l'opérande est le mot mémoire dont l'adresse est donnée par les 15 bits poids faibles d'un relais d'indirection, ce relais d'indirection étant le mot mémoire adressé par la base et le déplacement. Dans ce cas le bit 0 du relais d'indirection vaut 0. Ce mode d'adressage ne donne accès qu'a 32 K mots (adresses comprises entre SLO et SLO + 32 767).
- Adressage indirect post-indexé de mot : l'opérande est le mot mémoire dont l'adresse est donnée par la somme des 16 bits du registre X, et des 15 bits poids faible du relais d'indirection adressé par la base et le déplacement. Dans ce cas, le bit 0 du relais d'indirection vaut 1.
- Adressage direct d'octet : l'opérande est obligatoirement l'octet gauche du mot mémoire adressé par la base et le déplacement. Ce mode d'adressage ne peut être utilisé que dans des cas particuliers.
- Adressage indirect d'octet : l'opérande est obligatoirement l'octet gauche du mot mémoire dont l'adresse est donnée par les 15 bits poids faible du relais d'indirection adressé par la base et le déplacement. Dans ce cas le bit 0 du relais d'indirection vaut 0. Ce mode d'adressage ne donne accès qu'aux octets gauche des mots mémoire dont l'adresse est comprise entre SLO et SLO + 32 767. Ce mode d'adressage ne peut être utilisé que dans des cas particuliers.
- Adressage indirect post indexé d'octet;dans ce cas le bit 0 du relais d'indirection vaut 1. L'opérande est l'octet gauche du mot mémoire si la valeur du registre X est paire, l'octet droit si la valeur du registre X est impaire. L'adresse du mot est donnée par la somme du registre X décalé algébriquement à droite d'une position, et des 15 bits poids faibles du relais d'indirection adressé par la base et le déplacement. En faisant varier l'index par pas de 1 on peut ainsi avoir accès à des octets rangés successivement en mémoire :



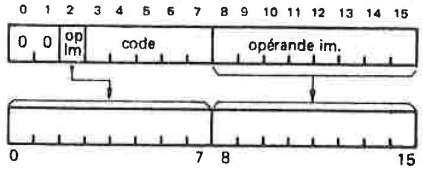
INSTRUCTIONS DE SAUT : σ

Ces instructions utilisent l'adressage par rapport à P : les bits 8 à 15 comportent un déplacement qui est additionné au registre P pour obtenir l'adresse vers laquelle le saut est effectué. Au moment de l'exécution P pointe sur l'instruction de saut : on peut atteindre de cette manière les 128 instructions qui la précèdent et les 127 instructions qui la suivent.

INSTRUCTIONS AVEC OPÉRANDE IMMÉDIAT 9 BITS : ♦

L'opérande effectif comporte 16 bits obtenus de la manière suivante

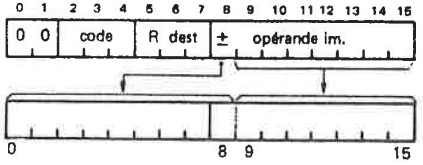
les 8 bits poids fort sont identiques au bit 2 de l'instruction, les 8 bits poids faible aux bits 8 - 15 de l'instruction



INSTRUCTION AVEC OPÉRANDE IMMÉDIAT 8 BITS : ●

L'opérande effectif comporte 16 bits obtenus de la manière suivante :

les 9 bits poids fort sont identiques au bit 8 de l'instruction, les 7 bits poids faible aux bits 9 - 15 de l'instruction.



Le codage du registre concerné est celui utilisé dans les instructions registre-registre.

INSTRUCTIONS REGISTRE-REGITRE : η

Le codage du registre source et du registre destination est le suivant :

A	B	X	Y	C	L	W	K
000	001	010	011	100	101	110	111

Pour certaines instructions le registre source ou le registre destination est implicite, dans ce cas le contenu des bits correspondants de l'instruction est sans signification.

L'opération est effectuée entre les contenus du registre source et du registre destination, et le résultat rangé dans le registre destination.

Il est toujours possible de spécifier le même registre comme registre source et destination.

DÉCALAGES ET OPÉRATIONS SUR BITS : ψ

Les 5 bits poids faibles de l'instruction indiquent un nombre de pas de décalage, ou un numéro de bit (de 0 pour Ao à 31 pour B_{15}).

Lorsque l'instruction est indexée (bit 10 = 1) les 5 bits poids faibles de l'instruction sont additionnés au registre X pour donner sur 5 bits le nombre de pas ou le n° de bit effectif. Cette opération s'effectue modulo 32.

INSTRUCTIONS VIRGULE FLOTTANTE : ζ ξ

Le premier mot est identique pour toutes les instructions en virgule flottante, il comporte le code '3800 et sert à les distinguer des autres instructions (voir codes réservés aux extensions page 139). S'il s'agit d'une instruction avec opérande en mémoire les bits 0 et 1 du second mot comportent l'indication de la base, le bit 2 indique l'adressage indirect, les bits 3 à 7 comportent le code opération et les bits 8 à 15 un déplacement.

S'il s'agit d'une instruction dont le ou les opérandes sont dans des registres, le second mot ne comporte que le code opération.

Selon l'option choisie ces instructions sont soit exécutées directement par le calculateur (option flottant hardware) soit exécutées par un ensemble de sous-programmes appelés par le calculateur lorsqu'il rencontre une instruction virgule flottante (option flottant programmé).

LISTE DES INSTRUCTIONS T1600

CHARGEMENTS (LA LB LX LY LAI LBI LXI LYI LR LRP LAD LBY PLR PULL)
RANGEMENTS (STA STB STX STY STZ STBY PSR PUSH)
ECHANGES (XR XIMR SWBR)
OPERATIONS ARITHMETIQUES
- Additions (AD ADRI ADR ADCR IC ADRP)
- Soustractions (SB SBR SBCR DC)
- Division - Multiplication - Nègation (DV MP NGR)

OPERATIONS LOGIQUES
- Intersections (AND ANDI ANDR)
- Unions (OR ORI ORR)
- Disjonctions (EOR EORI EORR)
- Masque - Complément (CLSR CMR)

DECALAGES
- Logiques (SLRS SLRD SLLS SLLD)
- Circulaires (SCRS SCRD SCLS SCLD)
- Arithmétiques (SARS SARD)

COMPARAISONS (CP CPI CPR CPZR CPZ CPBY)
SOUS PROGRAMMES (BSR RSR SVC RSV)
BRANCHEMENTS INCONDITIONNELS (BR JMP ADRP)
BOUCLES (JIX JDX)
SAUTS SUR TEST ACCUMULATEUR (JAL JAE JAG JAGE JALE JANE)
SAUTS APRES COMPARAISON (JL JE JG JGE JLE JNE)
SAUTS SUR TEST INDICATEURS (JC JV JNC JNV JCV JNCv)
MANIPULATION DE BITS (RBT SBT IBT TBT DBT)
MANIPULATION D'OCTETS (LBY STBY CPBY SWBR SBS PTY)
MANIPULATION DE PILES (PSR PLR PUSH PULL)
OPERATIONS DIVERSES (STZ IC DC LAD ADRP SCY PTY MOVE)
ENTREES/SORTIES (SIO BIO)
INTERRUPTIONS (ACQ ACK ACTD XIMR HALT)
INSTRUCTIONS SPECIALES (ACTD STEP ROMB)
TACHES (ARM QUIT)
SEMAPHORES (RQST RLSE WAIT ACT)
FLOTTANTS (FLD FST FAD FSB FNEG FABS FMP FDV FCAM FCAZ FCMZ FIX FLT NORM CDF CDI CFD CID ATAN COS

CHARGEMENTS

type	mnémonique	signification
⊕	LA MEM	Load A ¤ *Chargement du mot mémoire dans l'accumulateur*
⊕	LB MEM	Load B ¤ *Chargement du mot mémoire dans le registre B*
⊕	LX MEM	Load X ¤ *Chargement du mot mémoire dans le registre X*
⊕	LY MEM	Load Y ¤ *Chargement du mot mémoire dans le registre Y*
♁	LAI -256..255	Load A Immediate ¤ *Chargement de l'opérande immédiat dans l'accumulateur*
♁	LBI -256..255	Load B Immediate ¤ *Chargement de l'opérande immédiat dans le registre B*
♁	LXI -256..255	Load X Immediate ¤ *Chargement de l'opérande immédiat dans le registre X*
♁	LYI -256..255	Load Y Immediate ¤ *Chargement de l'opérande immédiat dans le registre Y*
η	LR R_s,R_d	Load Register by Register ¤ *Chargement du registre R_d par le registre R_s*
η	LRP R	Load Register with P ¤ *Chargement du registre par le registre P*
⊕	LAD MEM	Load Address ¤ *Chargement de l'accumulateur par l'adresse mémoire*
⊕	LBY MEM	Load Byte ¤ *Chargement de l'octet mémoire dans l'accumulateur*
◆	PLR R [,R...]	Pull Registers ¤ *Dépilement de registres (pile pointée par K)*
ω	PULL	Pull ¤ *Dépilement de l'accumulateur (pile pointée par Y)*

RANGEMENTS

type	mnémonique	signification
⊕	STA MEM	Store A ★ *Rangement de l'accumulateur dans la mémoire*
⊕	STB MEM	Store B ★ *Rangement du registre B dans la mémoire*
⊕	STX MEM	Store X ★ *Rangement du registre X dans la mémoire*
⊕	STY MEM	Store Y ★ *Rangement du registre Y dans la mémoire*
⊕	STZ MEM	Store Zero ★ *Remise à zéro d'un mot mémoire*
⊕	STBY MEM	Store Byte ★ *Rangement de l'accumulateur dans l'octet mémoire*
▲	PSR R [R,...]	Push Registers ★ *Empilement de registres (pile pointée par K)*
ω	PUSH	Push ★ *Empilement de l'accumulateur (pile pointée par Y)*

ECHANGES

type	mnémonique	signification
η	XR R,R	Exchange Registers ★ *Echange du contenu des deux registres*
η ©	XIMR R	Exchange Interrupt Mask with Register ★ *Echange des registres R et S*
η	SWBR R_s,R_d	Swap Byte Registers ★ *Les deux octets du registre source sont invertis et placés dans le registre destination*

© mode maître

OPERATIONS ARITHMETIQUES

Additions

type	mnémonique	signification
⊕	AD MEM	Add ★ *Addition de la mémoire à l'accumulateur*
•	ADRI -128..127,R	Add Register Immediate ★ *Addition de l'opérande immédiat au registre*
η	ADR R_s,R_d	Add Registers ★ *Addition du registre R_s au registre R_d*
η	ADCR R	Add Carry to Register ★ *Addition du carry au regsitre*
⊕	IC MEM	Increase Memory ★ *Addition de 1 à la mémoire, et comparaison à 0*
η	ADRP R	Add Register to P ★ *Addition du registre au registre P*

Soustractions

⊕	SB MEM	Substract ★ *Soustraction de la mémoire à l'accumulateur*
η	SBR R_s,R_d	Substract Registers ★ *Soustraction du registre R_s au registre R_d*
η	SBCR R	Substract Carry from Register ★ *Scustraction du carry au registre*
⊕	DC MEM	Decrement Memory ★ *Soustraction de 1 à la mémoire, et comparaison à 0*

Négation - Multiplication - Division

type	mnémonique	signification
⊕	DV MEM	Divide ★ *Division de AB par la mémoire; quotient dans A; reste dans B*
⊕	MP MEM	Multiply ★ *Multiplication de l'accumulateur par la mémoire; produit AB*
η	NGR R_s,R_d	Negate Registers ★ *L'opposé du registre source est chargé dans R_d*

OPERATIONS LOGIQUES

Intersection

⊕	AND MEM	And ★ *Intersection de l'accumulateur et de la mémoire*
•	ANDI -256..255	And Immediate ★ *Intersection de l'opérande immédiat et de l'accumulateur*
η	ANDR R_s,R_d	And Registers ★ *Intersection du registre R_s et du regsitre R_d*

Union

⊕	OR MEM	Or ★ *Union de l'accumulateur et de la mémoire*
•	ORI -256..255	Or Immediate ★ *Union de l'opérande immédiat et de l'accumulateur*
η	ORR R_s,R_d	Or Registers ★ *Union du registre R_s et du registre destination*

type	mnémonique	signification
⊕	EOR MEM	Exclusive Or ★ *Disjonction de l'accumulateur et de la mémoire*
●	EORI -256..255	Exclusive Or Immediate ★ *Disjonction de l'opérande imm. et de l'accu.*
η	EORR R_s,R_d	Exclusive Or Registers ★ *Disjonction du registre R_s et du registre R_d*

Masque - Complément

η	CLSR R_s,R_d	Clear Selective Registers ★ *Remise à zéro des bits du registre destination correspondants aux bits à un du registre source*
η	CMR R_s,R_d	Complement Registers ★ *Chargement du registre R_d par le complément de R_s*

DECALAGES

Logique

ψ	SLRS 0..31	Shift Logical Right Single ★ *Décalage logique droit dans A*
ψ	SLRD 0..31	Shift Logical Right Double ★ *Décalage logique droit dans AB*
ψ	SLLS 0..31	Shift Logical Left Single ★ *Décalage logique gauche dans A*
ψ	SLLD 0..31	Shift Logical Left Double ★ *Décalage logique gauche dans AB*

Circulaire

type	mnémonique	signification
ψ	SCRS 0..31	Shift Circular Right Single ★ *Décalage circulaire droit dans A*
ψ	SCRD 0..31	Shift Circular Right Double ★ *Décalage circulaire droit dans AB*
ψ	SCLS 0..31	Shift Circular Left Single ★ *Décalage circulaire gauche dans A*
ψ	SCLD 0..31	Shift Circular Left Double ★ *Décalage circulaire gauche dans AB*

Arithmétique

ψ	SARS 0..31	Shift Arithmetical Single ★ *Décalage arithmétique dans A*
ψ	SARD 0..31	Shift Arithmetical Double ★ *Décalage arithmétique dans AB*

COMPARAISONS ALGEBRIQUES

⊕	CP MEM	Compare Word ★ *Comparaison de l'accumulateur à la mémoire*
●	CPI -256..255	Compare Immediate ★ *Comparaison de l'accumulateur à l'opérande immédiat*
η	CPR R_s,R_d	Compare Registers ★ *Comparaison du second registre au premier registre*
η	CPZR R	Compare Zero to Register ★ *Comparaison du registre avec zéro*
⊕	CPZ MEM	Compare to Zero ★ *Comparaison de la mémoire à zéro*
⊕	CPBY MEM	Compare Byte ★ *Comparaison de l'accumulateur à l'octet mémoire*

SOUS PROGRAMMES

type	mnémonique	signification
⊕	BSR MEM	Branch Subroutine ★ *Sauvergarde de l'adresse de retour et branchement*
ω	RSR	Return from Subroutine ★ *Retour à l'adresse sauvegardée par un BSR*
○	SVC 0..255	Supervisor Call ★ *Appel d'une routine superviseur*
ω ©	RSV	Return from Supervisor ★ *Retour d'une routine superviseur*

BRANCHEMENTS INCONDITIONNELS

⊕	BR MEM	Branch ★ *Branchement à l'adresse donnée par le contenu de la mémoire*
σ	JMP $-128..$+127	Jump ★ *Saut inconditionnel*
η	ADRP R	Add Register to P ★ *Addition du registre au registre P*

BOUCLES

σ	JIX $-128..$+127	Jump Incrementing X ★ *Addition de 1 au registre X, et saut si X<0*
σ	JDX $-128..$+127	Jump Decrementing X ★ *Soustraction de 1 au registre X, et saut si X>0*

© mode maître

SAUTS SUR TEST ACCUMULATEUR

type	mnémonique	signification
σ	JAL $-128..$+127	Jump on A Less Zero ★ *Saut si Accumulateur<0*
σ	JAE $-128..$+127	Jump on A Equal Zero ★ *Saut si Accumulateur=0*
σ	JAG $-128..$+127	Jump on A Greater Zero ★ *Saut si Accumulateur>0*
σ	JAGE $-128..$+127	Jump on A Greater Or Equal Zero ★ *Saut si Accumulateur≥0*
σ	JANE $-128..$+127	Jump on A Not Equal Zero ★ *Saut si Accumulateur≠0*
σ	JALE $-128..$+127	Jump on A Less Or Equal Zero ★ *Saut si Accumulateur≤0*

SAUTS APRES COMPARAISON

σ	JL $-128..$+127	Jump on Less ★ *Saut si inférieur*
σ	JE $-128..$+127	Jump on Equal ★ *Saut si égal*
σ	JG $-128..$+127	Jump on Greater ★ *Saut si supérieur*
σ	JGE $-128..$+127	Jump on Greater Or Equal ★ *Saut si supérieur ou égal*
σ	JNE $-128..$+127	Jump on Not Equal ★ *Saut si différent*
σ	JLE $-128..$+127	Jump on Less Or Equal ★ *Saut si inférieur ou égal*

SAUTS SUR TEST INDICATEURS

type	mnémonique		signification
σ	JC	$-128..$+127	Jump On Carry ★ *Saut si le carry vaut 1*
σ	JV	$-128..$+127	Jump On Overflow ★ *Saut si l'overflow vaut 1*
σ	JNC	$-128..$+127	Jump On Not Carry ★ *Saut si le carry vaut 0*
σ	JNV	$-128..$+127	Jump On Not Overflow ★ *Saut si l'overflow vaut 0*
σ	JCV	$-128..$+127	Jump On Carry Or Overflow ★ *Saut si le carry vaut 1 ou l'overflow vaut 0*
σ	JNCV	$-128..$+127	Jump On Not Carry And Overflow ★ *Saut si carry vaut 0 et overflow vaut 0*

MANIPULATION DE BITS

ψ	RBT	0..31	Reset Bit ★ *Mise à zéro d'un bit du registre AB*
ψ	SBT	0..31	Set Bit ★ *Mise à 1 d'un bit du registre AB*
ψ	IBT	0..31	Inverse Bit ★ *Inversion d'un bit du registre AB*
ψ	TBT	0..31	Test Bit ★ *Test d'un bit du registre AB*
ω	DBT		Discover Bit ★ *Chargement du regsitre X par le rang du bit le plus à gauche dans le registre AB*

MANIPULATION D'OCTETS

type	mnémonique		signification
⊕	LBY	MEM	Load Byte ★ *Chargement de l'octet mémoire dans l'accumulateur*
⊕	STBY	MEM	Store Byte ★ *Rangement de l'accumulateur dans l'octet mémoire*
⊖	CPBY	MEM	Compare Byte ★ *Comparaison de l'accumulateur et de l'octet mémoire*
η	SWBR	R_s,R_d	Swap Byte Registers ★ *Les deux octets du registre source sont invertis et placés dans le registre destination*
ω	SBS		Search Byte String ★ *Recherche d'un octet identique à celui contenu dans l'accumulateur, dans la chaîne dont l'adresse est donnée par le registre B et la longueur par le regsitre Y, à partir de l'octet dont le rang est donné par le regsitre X*
ω	PTY		Parity ★ *Test de la parité de l'octet droit de l'accumulateur*

MANIPULATION DE PILES

▲	PSR	R [,R...]	Push Registers ★ *Empilement des registres (pile pointée par K)*
◆	PLR	R [,R...]	Pull Registers ★ *Dépilement des registres (pile pointée par K)*
ω	PUSH		Push ★ *Empilement de l'accumulateur (pile pointée par Y)*
ω	PULL		Pull ★ *Dépilement de l'accumulateur (pile pointée par Y)*

OPERATIONS DIVERSES

type	mnémonique	signification
⊕	STZ MEM	Store Zero ★ *Mise à zéro du mot mémoire*
⊕	IC MEM	Increment Memory ★ *Addition de 1 au mot mémoire*
⊕	DC MEM	Decrement Memory ★ *Soustraction de 1 au mot mémoire*
⊕	LAD MEM	Load Address ★ *Chargement de l'accumulateur par l'adresse de l'opérande*
η	ADRP R	Add Register to P ★ *Addition du registre au registre P*
ω	MOVE	Move ★ *Transport du contenu d'une zone mémoire dont l'adresse est donnée par l'accumulateur et la longueur par le registre Y, dans une zone mémoire dont l'adresse est donnée par le registre B*
ω	SCY	Set Carry ★ *Mise à 1 du carry, sans modification de l'overflow*
ω	PTY	Parity ★ *Test de la parité de l'octet droit de l'accumulateur*

ENTREES/SORTIES

⊕ ©	SIO MEM	Start Input/Output ★ *Lancement d'une opération d'entrée/sortie avec un périphérique (mode programmé simple ou prioritaire, microcanal ou ADM)*
⊕ ©	BIO MEM	Bus Input/Output ★ *Echange de signaux avec le bus général*

) mode maître

INTERRUPTIONS

type	mnémonique	signification
ω ©	ACQ	Acquit ★ *Fin de tâche hardware*
ω ©	ACK	Acknowledge ★ *Reconnaissance et annulation d'une interruption en attente*
η ©	XIMR R	Exchange Interrupt Mask With Register ★ *Echange des registres R et S*
ω	HALT	Halt ★ *Arrêt de l'exécution des instructions jusqu'à la prochaine IT*

INSTRUCTIONS SPECIALES

ω	ACTD	Activate Debug ★ *Interruption de niveau 0, sous niveau 2*
ω ©	STEP	Step ★ *Exécution d'une instruction et IT de niveau 0, sous niveau 3*
ω ©	ROMB	ROM Branch ★ *Déclanchement de l'exécution d'un microprogramme*

TACHES

□ ©	ARM 0..127 [,X]	Arm ★ *Armement d'une tâche software*
ω	QUIT	Quit ★ *Fin de tâche software*

Ə mode maître

SEMAPHORES

type	mnémonique	signification
⊕	© RQST MEM	Request ★ *Primitive P sur sémaphore d'exclusion*
⊕	© RLSE MEM	Release ★ *Primitive V sur sémaphore d'exclusion*
⊕	© WAIT MEM	Wait ★ *Primitive P sur sémaphore de synchronisation*
⊕	© ACT MEM	Activate ★ *Primitive V sur sémaphore de synchronisation*

FLOTTANTS

ζ	FLD MEM	Floating Point Load ★ *Chargement*
ζ	FST MEM	Floating Point Store ★ *Rangement*
ζ	FAD MEM	Floating Point Add ★ *Addition*
ζ	FSB MEM	Floating Point Substract ★ *Soustraction*
ξ	FNEG	Floating Point Negate ★ *Négation*
ξ	FABS	Floating Point Absolute Value ★ *Valeur absolue*
ζ	FMP MEM	Floating Point Multiply ★ *Multiplication*
ζ	FDV MEM	Floating Point Divide ★ *Division*
ζ	FCAM MEM	Floating Point Compare Accumulator to Memory ★ *Comparaison à la mémoire*
ζ	FCAZ MEM	Floating Point Compare Accumulator to Zero ★ *Comparaison à zéro*

© mode maître

type	mnémonique	signification
ζ	FCMZ MEM	Floating Point Compare Memory to Zero ★ *Comparaison de la mémoire à zéro*
ξ	FIX	Floating Point Integer Conversion ★ *Flottant ⇒ Entier*
ξ	FLT	Float ★ *Entier ⇒ Flottant*
ξ	NORM	Normalize ★ *Normalisation*
ζ	CDF MEM	Convert Decimal to Float ★ *Décimal ⇒ Flottant*
ζ	CDI MEM	Convert Decimal to Integer ★ *Décimal ⇒ Entier*
ζ	CFD MEM	Convert Float to Decimal ★ *Flottant ⇒ Décimal*
ζ	CID MEM	Convert Integer to Decimal ★ *Entier ⇒ Décimal*
ξ	ATAN	Arc Tangent ★ *Arc tangente*
ξ	COS	Cosine ★ *Cosinus*
ξ	EXP	Exponential ★ *Exponentielle*
ξ	SIN	Sine ★ *Sinus*
ξ	SQRT	Square Root ★ *Racine carrée*
ξ	TANH	Hyperbolic Tangent ★ *Tangente hyperbolique*
ξ	LOG	Natural Logarithm ★ *Logarithme népérien*

?
■

σ	NOP $-123..$+127	No Operation ★ *Addition de 1 au registre P*

INSTRUCTIONS PRIVILEGIEES

(NON UTILISABLES EN MODE ESCLAVE)

ACK	*ACKNOWLEDGE*
ACQ	*ACQUIT*
ACT	*ACTIVATE*
ARM	*ARM*
BIO	*BUS INPUT\OUTPUT*
RLSE	*RELEASE*
ROMB	*READ ONLY MEMORY BRANCH*
RQST	*REQUEST*
RSV	*RETURN FROM SUPERVISOR*
SIO	*START INPUT\OUTPUT*
WAIT	*WAIT*
XIMR	*EXCHANGE INTERRUPT MASK WITH REGISTER*

TRADUCTION DE BABEL SUR T1600: IMPLANTATION DES MONOLITHES, BRANCHEMENTS, REGISTRES, REQUETES AU SUPERVISEUR

Correspondance monolithe-tâche software Solar 16

Tout monolithe actif est géré comme une tâche software esclave et par conséquent la zone qu'il occupe est délimitée par le contenu des registres SLO-SLE.

Il se compose de deux parties :

- une partie indépendante des diverses activations de la procédure constante ;
- une partie variable spécifique à l'activation concernée.

Ces deux parties sont traitées différemment par le moniteur d'exécution afin d'éviter des recopies inutiles de la partie invariante lors d'appels récursifs ou simplement rapprochés d'une procédure.

Un monolithe, d'autre part, se décompose en plusieurs segments dont l'accès est différent

- le code parcouru par le pointeur d'instruction ;

(C) - la zone de données constantes engendrées par les passes 1 et 2 sur laquelle pointe le registre (RL) Solar ;

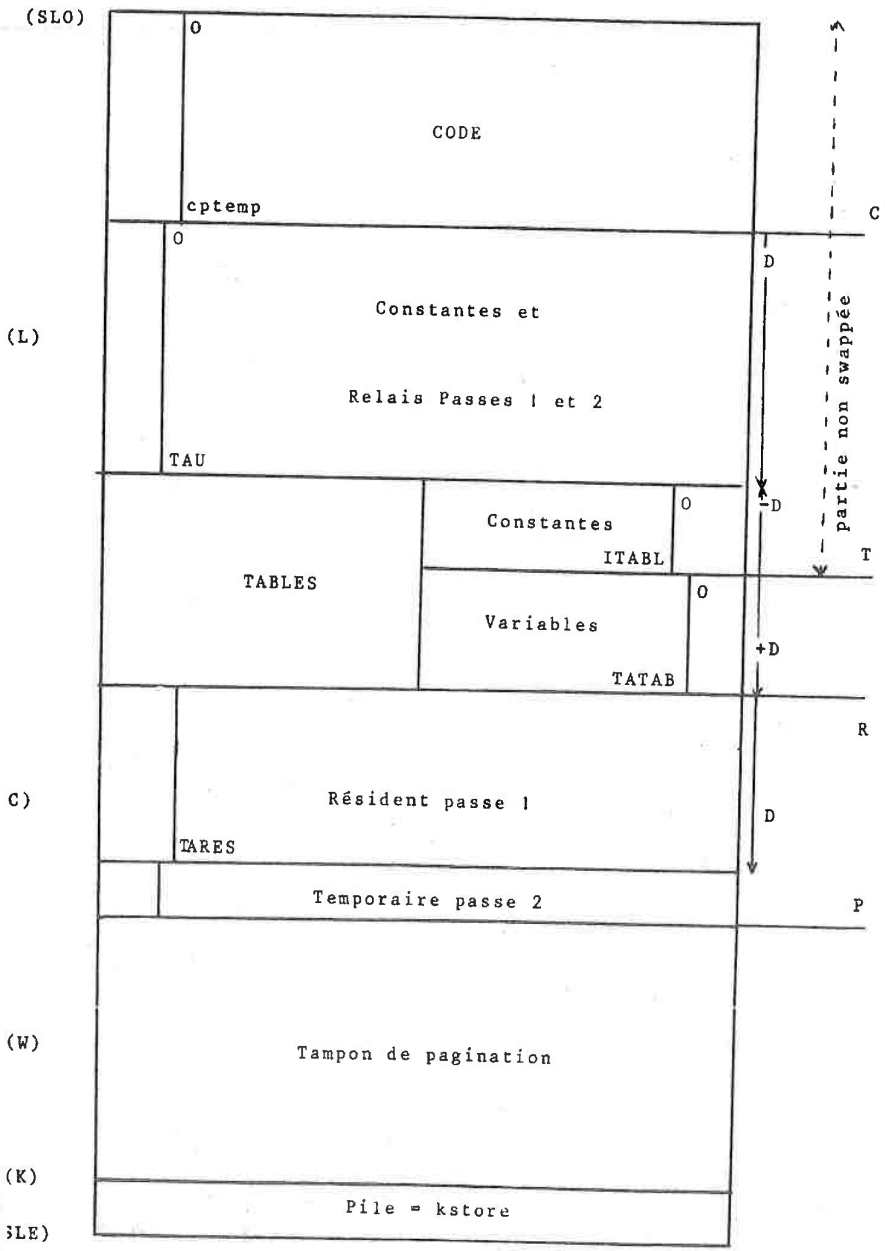
(T) - la zone de Tables à accès indirect au moyen de relais en zone constante ou résidente ;
Avec la partie constante de cette zone se termine la partie invariante du monolithe.

(R) - la zone résidente comprenant les variables simples de la procédure ainsi que les variables temporaires d'exécution et une zone de sauvegarde des registres ; le registre (RC) pointe sur cette zone.

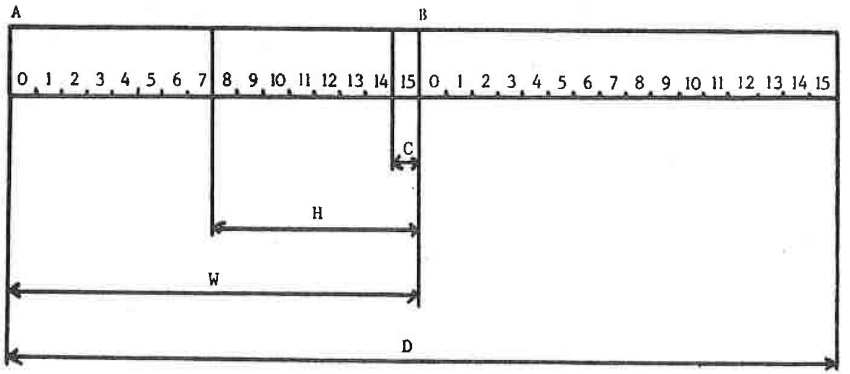
(P) - le tampon de pagination contenant une page de mémoire virtuelle, auquel on accède à l'aide de requêtes spécialisées;

T1600

BABEL 19



Equivalence entre les registres C, H, W et D de BABEL et les registres A et B de SOLAR 16.



les registres (et accumulateurs) Babel sont alloués à divers types d'objets Pascal.

Registre C : alloué aux scalaires d'au plus 2 éléments. Il est représenté par le dernier bit de l'accumulateur RA.

Registre H : alloué aux scalaires de 3 à 256 éléments. Il est représenté par le second octet de RA.

Registre W : alloué aux scalaires de 257 à 32767 éléments et aux entiers (-32768 .. +32767). Il est représenté par l'accumulateur RA.

Registre D : alloué aux nombres réels. Il est représenté par le registre étendu RA,RB.

Registres M : alloués aux ensembles et scalaires de plus de 32767 éléments. Ils subissent un traitement spécial.

I - Résolution des branchements étiquetés

a) *nomenclature*

On utilise 3 tableaux

TABETI : array (. 0.. maxeti.) of integer

contient le compteur d'emplacement correspondant à l'étiquette lorsque celle-ci a été définie par une directive.
Initialisé à -1 en début de procédure.

TIREF : array (..1..maxref,1..2.)of integer

fait la liaison entre :
colonne 1 : la valeur des étiquettes référencée au cours de la procédure.
colonne 2 : la valeur du compteur d'emplacement au moment de la référence à une étiquette non encore définie.

ETIREF(.-,1.),ETIREF(.-,2.) sont initialisés à -1 en début de procédure.

ETIREF(.-,2.) reprend la valeur -1 lorsque

a) l'étiquette est définie
b) le branchement court doit être transformé en branchement long.

TRESOL : array (.0..maxeti.) of integer

contient l'adresse, dans le buffer du segment constant, des relais indirects correspondant aux branchements longs.

initialisé à -1 en début de procédure.

b) *fonctionnement*

- A la rencontre d'un branchement étiqueté, deux cas se présentent :

1. l'étiquette a déjà été déclarée

On peut déterminer alors s'il faut un branchement long (par relais) ou court (direct).

. branchement court
saut arrière simple

. branchement long
- si l'étiquette est référencée pour la première fois
  * on crée un relais pointant sur cette étiquette
  * on enregistre ce relais dans TNRESOL
  * on génère le branchement long
- si l'étiquette a déjà été référencée
  * on génère le branchement sur le relais déjà existant

2. l'étiquette n'a pas encore été déclarée

- on génère un branchement court en avant quitte à le transformer en branchement long si l'étiquette n'est pas déclarée avant la limite de 127 mots
- on enregistre la référence à l'étiquette dans le tableau ETIREF

- A la rencontre d'une étiquette :

. on enregistre la déclaration d'étiquette dans le tableau TABETI.
. si des branchements longs ont été créés (TNRESOL en témoigne) on les résout en rangeant dans le relais correspondant le compteur ordinal de l'instruction suivant cette déclaration.
. on résout les branchements courts mémorisés par ETIREF à l'aide des informations qu'il contient.
. ETIREF (.-,2.) est réinitialisé à -1.

I - branchements courts - branchements longs

a) *les divers types de branchements*

Les branchements peuvent être regroupés en plusieurs types:

- à l'intérieur d'une instruction Babel

certains branchements sont nécessaires dans la compilation elle-même d'une instruction Babel. Ils sont alors courts et leur valeur de saut est connue à la compilation; leur code peut donc être généré immédiatement.

- étiquetés

(voir page 9)

dans le cas d'un branchement sur condition (instruction Babel FJMP) on génère une séquence de trois instructions destinée à ramener ce cas au cas général. Par exemple:

```
        JAE   $+2
        JMP   $+2
        JMP   $+30     saut étiqueté court
```

ou

```
        JAE   $+2
        JMP   $+2
        BR    &-12,L   branchement long
```

en effet l'instruction BR étant inconditionnelle on ne pourrait transformer directement le branchement court en branchement long. Seuls les branchements courts arrière n'ont pas à être transformés.

b) *incrémentation du compteur d'emplacement*

A chaque écriture d'un mot de code le compteur d'emplacement est incrémenté d'une unité.

A chaque incrémentation du compteur d'emplacement on parcourt la table des références non résolues (tableau ETIREF). Si un branchement en avant vient à dépasser la limite des sauts courts (saut simple JMP du solar 16 limité à l'intervalle -128..+127) il faut changer le code provisoire généré en un branchement indirect avec relais en zone constante.

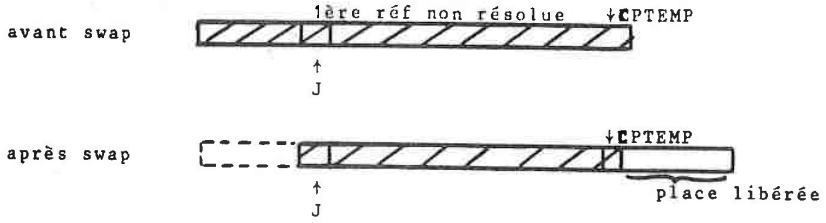
Le relais contient l'adresse de l'étiquette référencée dont la valeur est calculée à la rencontre de cette étiquette.

L'existence d'un buffer contenant plus de 128 mots de code permet la transformation du code provisoire. (voir: écriture du code)

III - Ecriture du code

Chaque instruction générée est écrite dans un buffer de taille supérieure à 127 mots (BUFCOD). Lorsque ce buffer est rempli la partie du buffer où les références sont toutes résolues est écrite dans la zone attribuée au code de la procédure. Elle est désormais inaccessible.

BUFCOD dispose donc d'un compteur (J) qui pointe sur la première référence non résolue trouvée, et qui permet la translation du code libérant à nouveau de la place.



V - Relais

Ils sont de deux sortes :

. relais Babel :

comportant 1, 2 ou 3 mots et servant à l'adressage. Générés dans le segment constant, les relais de 1 mot doivent être complétés en fin de procédure lorsqu'on peut calculer l'adresse exacte sur laquelle ils pointent.

. relais générés par le compilateur :

- relais de branchement indirect:
  leur valeur est déterminée au cours de la procédure.
- relais de l'instruction XJMP:
  doit être complété en fin de procédure.
- relais portant sur les variables temporaires:
  doivent être complétés en fin de procédure.

les relais nécessitant d'être complétés doivent donc être mémorisés lorsqu'on les rencontre.

On range donc :
- leur emplacement dans le segment constant ;
- la zone sur laquelle ils pointent.

En fin de procédure

On connaît alors la taille des différentes zones du monolithe constituant la procédure. On calcule alors pour chaque relais la valeur exacte de l'adresse sur laquelle il pointe.

Le système des Requêtes

Les requêtes sont des appels au moniteur d'exécution. Leur fonctionnement est décrit dans le moniteur d'exécution. On décrit ici l'interface compilateur-moniteur et les opérations qu'il nécessite dans le compilateur 2ème passe.

Les requêtes sont de divers types :

a) *Les requêtes d'adressage*

Elles sont nécessaires chaque fois que l'on veut accéder à un objet extérieur au monolithe actif.

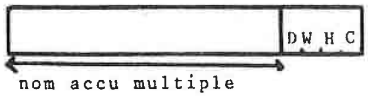
Elles se présentent donc :
- pour tout adressage global ;
- pour un adressage local lorsque l'objet appartient au segment paginé.

Ces deux cas sont distincts mais traités de manière peu différentes.

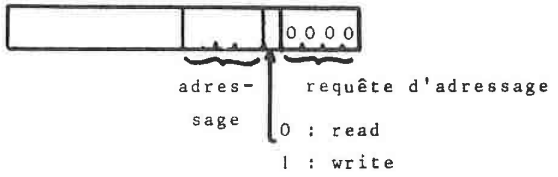
Afin d'optimiser l'accès auxdonnées, l'interface se fait au moyen de registres du T1600. On utilise RA, RB et RY. Il faudra donc sauvegarder le contenu de ces registres s'il est à conserver ainsi que celui du registre d'index RX qui peut être détruit par la requête. On utilisera la kstore pour empiler ces registres.

Les renseignements nécessaires à une requête sont :
- pour un adressage local
  - . D : le déplacement de l'objet dans le segment paginé
  - . le nom de l'accumulateur concerné
  - . le nom de l'adressage : L0, L1, L2
- pour un adressage global
  - . l'adresse du relais d'adressage dans le monolithe
  - . le nom de l'accumulateur concerné (C,H,W,D)
  - . le nom de l'adressage : G1S, G1D, G2S, G2D

On choisit d'enregistrer ces renseignements de la manière suivante :
- dans RA : le nom de l'accumulateur concerné.
  Il est codé de la manière suivante :



- dans RB : le déplacement (adressage local) ou l'adresse absolue du relais d'adressage (adressage global).
- dans RY : le nom de la requête.
  Avec en plus la précision écriture ou lecture.
  Ceci est codé de la manière suivante :



Code d'adressage :

L0	000
L1	001
L2	010
G1S	011
G1D	100
G2S	101
G2D	110

Ces divers renseignements sont obtenus de l'instruction Babel faisant appel à l'adressage.
Lorsque le codage dans les trois registres RA, RB et RY est terminé la requête proprement dite est faite par un appel au superviseur dont le numéro est réservé aux requêtes d'adressage. La valeur transmise se trouve soit dans les registres RA et RB soit dans la partie RCB du segment T pour les accumulateurs multiples. Les registres RX et RY ont été empilés (section k) si nécessaire.

b) *les requêtes pour objets dynamiques*

Les instructions NEW et DISP permettent respectivement la génération et la destruction d'objets dynamiques déterminés par un pointeur contenant taille et adresse. Ce pointeur est adressé par les paramètres de l'instruction :

N niveau statique du pointeur
S segment contenant le pointeur
D déplacement dans le segment S

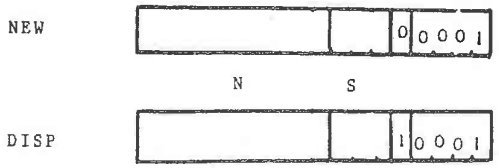
D'autre part, la taille de l'élément est contenue dans le registre W (RA du T1600).

On choisit donc d'utiliser en plus les registres RB et RY pour l'interface.

dans RA : taille de l'objet

dans RB : déplacement

dans RY : niveau statique N, segment S, New ou Dispose, et numéro de requête codés de la manière suivante

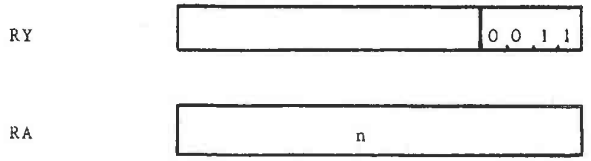


c) *les requêtes de service*

Elles sont de deux sortes :
- pour les directives CLOK et MESS
- pour la directive BUG

Dans le premier cas seul le registre RY est utilisé
Dans le second cas on utilise deux registres

- RY : pour le numéro de requête cadré dans les quatre derniers bits.
- RA : pour le numéro n de l'erreur détectée à la première passe



Pour les instructions CLOK et MESS le numéro de requête est 2 .
Les instructions TIME et DATE sont compilées.

d) *les requêtes sur monolithes*

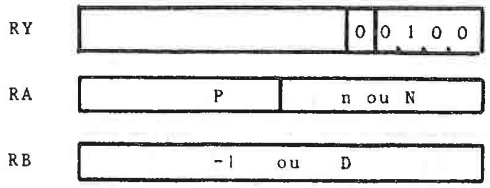
Elles correspondent à deux types de transactions sur monolithe : les opérations vitales et le passage de paramètres.

1 - Opérations vitales

- CIVA Incarnation de monolithe

Deux cas sont possibles :
- l'instruction Babel fournit le nom interne n de la procédure
- l'instruction Babel fournit l'adresse du mot contenant le nom interne de la procédure : N D

D'autre part P exprime le lien statique avec la procédure P.
On choisit de passer ces renseignements de la manière suivante :



- XIPE Réincarnation de monolithe

Instruction spécialement adaptée à la récursivité. On garde le même numéro de requête que pour CIVA, le bit 11 faisant la différence. Un seul registre suffit.

- NIRV Désincarnation

RAMA Désincarnation totale

Ces deux instructions utilisent le même numéro de requête. Un seul registre suffit.

- NAME Construction de relais dynamique

Le numéro de requête est 14, le registre RY suffit

2 - Passage des paramètres

Le passage des paramètres se décompose en deux phases

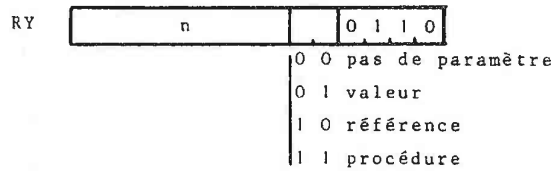
- Envoi des paramètres

cette phase est entièrement compilée : PVAR, PVAL et PMON ont pour effet d'empiler trois mots dans la pile du monolithe actif correspondant aux trois mots d'un relais d'adressage dans le cas général.

- Récupération des paramètres

L'instruction RATI utilise une requête pour laquelle il faut préciser:

- le type du passage de paramètre: valeur, référence, procédure
- le déplacement D dans le segment résident pour PVAR et PMON
- l'accumulateur concerné R pour PVAL



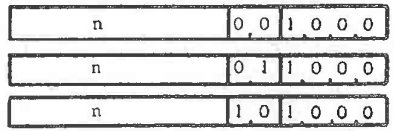
Pour les fonctions la transmission du résultat est compilée en empilant l'accumulateur concerné R.
La récupération utilise une requête précisant le nom de l'accumulateur concerné
SFCT est donc compilée
RFCT utilise le registre RY

e) *Les requêtes d'entrée-sortie sur fichier*

Un fichier est connu par son nom (entier positif). On regroupe ensemble les instructions d'après le type d'opération qu'elles effectuent.

- Ouverture FILE (permanent), VEDA (temporaire)

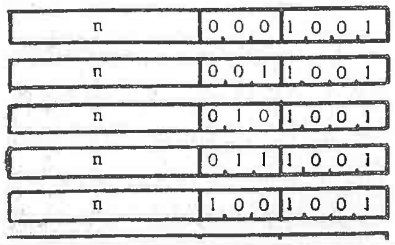
Fermeture KALI



- Rembobinage RSET, RWRT

Accès GET, PUT

Test EOF, EOLN



6) opérations sur fichier text

Un nombre important d'instruction est réservé aux fichiers :ext. On regroupe les lectures d'une part et les écritures l'autre part.

Les lectures sont de trois types

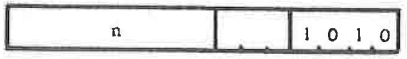
a) RIDC, RIDR, RIDE pour lesquelles il suffit de passer le nom de fichier

b) RIDI pour laquelle il faut préciser l'accumulateur concerné

c) RIDS, RIDP pour lesquelles il faut préciser le type du scalaire ou de l'ensemble

d) RIDL pour laquelle il faut préciser l'adresse du pointeur sur la chaine d'octets concernée

n utilise le registre RY



0 0 0	RIDC
0 0 1	RIDR
0 1 0	RIDE
0 1 1	RIDI
1 0 0	RIDS
1 0 1	RIDP
1 1 0	RIDL

vec en plus pour les cas b), c) et d)

b) dans RA l'accumulateur concerné

c) le type du scalaire ou de l'ensemble qui est empilé sur 4 mots

d) une chaine d'octets pointée par un pointeur adressé par l'instruction

. l'adressage est local
on empile 2 mots correspondant au relais d'adressage issu de l'opérande de l'instruction

. l'adressage est global
on empile le relais d'adressage sur 2 mots

De même on regroupe les écritures qui sont aussi de trois sortes:

a) WRTC, WRTR, WRTE, PAGE pour lesquelles il suffit de passer le nom de fichier

b) WRTI pour laquelle il faut préciser l'accumulateur concerné

c) WRTS, WRTP pour lesquelles il faut préciser le type du scalaire ou de l'ensemble

d) WRTL pour laquelle il faut préciser l'adresse du pointeur sur la chaine d'octets concernée

On utilise le registre RY

0 0 0	WRTC
0 0 1	WRTR
0 1 0	WRTE
0 1 1	WRTI
1 0 0	WRTS
1 0 1	WRTP
1 1 0	WRTL
1 1 1	PAGE

Avec en plus pour les cas b), c) et d)

b) dans RA l'accumulateur concerné

RA | accu multiple | D W H C |

c) le type du scalaire ou de l'ensemble qui est empilé sur 4 mots

d) une chaine d'octets pointée par un pointeur adressé par l'instruction

. l'adressage est local
on empile 2 mots correspondant au relais d'adressage issu de l'opérande de l'instruction

. l'adressage est global
on empile le relais d'adressage sur 2 mots

En fin l'instruction CADR pour le passage des facteurs de cadrage de certaines de ces instructions à laquelle on affecte le numéro 12

g) *erreurs à l'éxécution*

Une requête leur est réservée qui porte le numéro 13.
RY contient le numéro de requête
RA contient le numéro de l'erreur détectée

EXEMPLES DE TRADUCTION PASCAL→BABEL

EXEMPLE 1

Lire une suite de nombres, et pour chaque terme imprimer la valeur lue et la somme de la suite:

```
program X ;
var s,i : integer ;
begin s:=0 ;
    repeat read(i) ;
           s:=s+i ;
           writeln(i,s:30)
    until eof(input)
end .
```

Implantation en mémoire :

Procédure 0 : Segment R @0 s : 1 mot
@1 i : 1 mot

Traduction :

```
                program X ;
PRØG X
                var s,i : integer ;
VAR   S : INTEGER ; R 0
VAR   I : INTEGER ; R 1
                begin s := 0 ;
RATI  0
LDCW  0
STØW  R   0
```

```
                repeat read(i) ;
:0
RIDI   input
STØW   R   1
                        s := s+i ;
LØDW   R   0
ADDI   R   1
STØW   R   0
                        writeln(i,s:30)
LØDW   R   1
WRTI   output
LDCW   10
CADR
LØDW   R   0
WRTI   output
LDCW   30
CADR
WRTE
                until eof (input)
EØF    input
FJMP   0
                end.
NIRV
END    0
```


EXEMPLE 2

Calcul de la fonction d'Ackermann, pour les valeurs 3 et 7 :

```
program ackermann ;

function ack(i,j:integer):integer ;
begin
    if i=0
      then ack:=j+1
      else if j=0
             then ack:=ack(i-1,1)
             else ack:=ack(i-1,ack(i,j-1))
end ;

begin writeln('ack(3,7)=', ack(3,7))  end .
```

Implémentation en mémoire :

```
Procédure 0 :  Segment T   @0 : chaîne de 5 mots
Procédure 1 :  Segment C   @0 : variable temporaire
               Segment R   @0 : ack : 1 mot
                           @1 :   i : 1 mot
                           @2 :   j : 1 mot
```

Traduction :

```
                program ackermann;
PRØG  ACKERMAN
                function ack(i,j:integer):integer;
FUNC  1    1 ACK : INTEGER ; R  0
VPAR  I : INTEGER ; R 1
VPAR  J : INTEGER ; R 2
                begin if i = 0 then ack := j + 1
RATI  1,V W R 1,V W R 2
LØDW  R 1
EQUW
FJMP  0
LØDW  R 2
INCR  W 1
STØW  R 0
                else if j = 0 then ack := ack(i-1,1)
GØTØ  1
:0
LØDW  R 2
EQUW
FJMP  2
LØDW  R 1
INCR  W -1
PVAL  W
LDCW  1
PVAL  W
CIVA  1
RFCT  W
STØW  R 0
```



```
                else ack := ack(i-1,ack(i,j-1))
GØTØ  1
:2
LØDW  R 1
PVAL  W
LØDW  R 2
INCR  W -1
PVAL
CIVA  1
RFCT  W
STØW  C 0
LØDW  R 1
INCR  W -1
PVAL  W
LØDW  C 0
PVAL  W
CIVA  1
RFCT  W
STØW  R 0
                end;
:1
LØDW  R 0
SFCT  W
NIRV
END   1
                begin
RATI  0
                      writeln (' ack(3,7)=' , ack(3,7))
WRTL  n 0
TABL  ' ACK(3,7)='
LDCW  10
CADR
LDCW  3
PVAL  W
LDCW  7
PVAL  W
CIVA  1
RFCT  W
WRTI  n
LDCW  10
CADR
WRTE  n
                end.
NIRV
END   0
```

EXEMPLES DE TRADUCTION DE BABEL SUR SOLAR 16

Dans les exemples suivants, dont la succession n'a pas de signification par elle-même, on trouvera pour chaque instruction Babel traduite le code machine généré, sous formes hexadécimale et symbolique, précédé de la valeur du compteur d'emplacement.

DISP	1 R 102	destruction d'objet dynamique pointé par le relais adressé (requête)
L	7	directive numéro de ligne source Pascal
INDX	2 3 2	calcul d'indice dans un tableau d'adresse relative 2 dans l'objet de niveau supérieur, dont les éléments ont une taille de 3 mots, et où la borne inférieure de l'indice est 2.

```
DISP 1 R 102
              007C    9694    LY      -108,L
              007D    1766    LBI     102
              007E    1C00    SVC     '0

 L  7
 INDX 2 3 2
              007F    37FE    LBI     -2
              0080    2C08    ADR     B,A
              0081    1703    LBI     3
              0082    AC90    STB     £-112,L
              0083    AE90    MP      £-112,L
              0084    28D0    TBT     16
              0085    2D80    ADCR    A
              0086    2600    JAE     $+4
              0087    1006    LAI     6
              0088    160A    LYI     10
              0089    1C00    SVC     '0
              008A    1002    LAI     2
              008B    2C08    ADR     B,A
              008C    2BC2    LR      A,X
```

LDCC	1	chargement immédiat de bit
GEQW	I P 1000	comparaison supérieur ou égal de mots, adressage local indirect dans le segment paginé
LDCH	3	chargement immédiat d'octet
LDCW	-13	chargement immédiat de mot
GOTO	18	branchement à l'étiquette 18 (branchement en avant)

```
LDCC 1
              008D    1001    LAI     1
GEQW I P 1000
              008E    AD90    STA     E-112,L
              008F    9795    LB      -107,L
              0090    1620    LYI     32
              0091    1004    LAI     4
              0092    1C00    SVC     '0
              0093    B590    CP      E-112,L
              0094    2001    JMP     $+1
              0095    A596    BR      E-106,L
              0096    0700    JLE     $+3
              0097    2001    JMP     $+1
              0098    A596    BR      E-106,L
              0099    1000    LAI     0
              009A    2001    JMP     $+1
              009B    A596    BR      E-106,L
              009C    A597    BR      E-105,L
              009D    2000    JMP     $+2
              009E    1001    LAI     1
LDCH 3
              009F    1003    LAI     3
LDCW -13
              00A0    30F3    LAI     -13
GOTO 18
              00A1    2000    JMP     $+0
```


LODH	R 20	chargement d'octet, adressage local direct dans le segment résident
INDX	2 H -1 +	calcul d'indice dans un tableau d'octets
STOC	X P 1234	rangement de bit, adressage local indirect post-indexé dans le segment paginé (requête)
LODW	C 73	chargement de mot, adressage direct local dans le segment constant
LODC	I R 44 8	chargement de bit, adressage local indirect dans le segment résident
LODD	I C 1	chargement de double mot, adressage local indirect dans le segment sonstant

```
LODH R 20
              00A3    4096    LBY     -106,C
 INDX 2 H -1 +
              00A4    1701    LBI     1
              00A5    2C08    ADR     B,A
              00A6    28D0    TBT     16
              00A7    2D9B    ADCR    Y
              00A8    2801    SARS    1
              00A9    1702    LBI     2
              00AA    2C08    ADR     B,A
              00AB    2C02    ADR     A,X
STOC X P 1234
              00AC    1A10    PSR     A,X,Y
              00AD    9798    LB      -104,L
              00AE    1650    LYI     80
              00AF    1001    LAI     1
              00B0    1C00    SVC     '0
              00B1    1B08    PLR     A,X,Y
LODW C 73
              00B2    90C9    LA      -55,L
LODC I R 44 8
              00B3    70AE    LA      E-82,C
              00B4    2001    JMP     $+1
              00B5    A599    BR      E-103,L
              00B6    A59A    BR      E-102,L
              00B7    A59B    BR      E-101,L
              00B8    28C8    TBT     8
              00B9    1000    LAI     0
              00BA    2D80    ADCR    A
LODD I C 1
              00BB    3800
              00BC    A081    FLD     E-127,L
```

```
:14                 définition d'étiquette
LODH    I C 100     chargement d'octet, adressage local indirect
                    dans le segment constant
MANU    W -1785 -1      vérification que la valeur contenue
                    dans le registre W appartient à l'intervalle
                    -1785..-1 ;  instruction suivie d'une
                    instruction BUG
BUG     8           erreur d'exécution numéro 8 (requête)
TIME    R 12        demande de l'heure au système
INCR    C 1         addition immédiate au registre C (fonction
                    Pascal succ)
GOTO    19          branchement : l'étiquette 19 est située
                    trop loin en arrière pour un branchement
                    court, on génère un relais
LODW    X C 13      chargement de mot, adressage local indirect
                    post-indexé dans le segment constant
```

```
: 14
  LODH I C  100
              02F8    A0E4    LBY     £-28,L

MANU W -1785 -1
              02F9    95C1    CP      -63,L
              02FA    0500    JL      $+3
              02FB    35FF    CPI     -1
              02FC    0700    JLE     $+2

BUG 8
              02FD    1008    LAI     8
              02FE    1603    LYI     3
              02FF    1C00    SVC     '0

TIME R 12
              0300    9D9D    LAD     -99,L
              0301    1C35    SVC     '35
              0302    5D8E    LAD     -114,C
              0303    2BC1    LR      A,B
              0304    BD9D    LAD     £-99,L
              0305    0803    ADRI    3,A
              0306    1103    LXI     3
              0307    1E09    MOVE

INCR C 1
              0308    0801    ADRI    1,A

GOTO 19
              0309    A5C2    BR      £-62,L

   LODW X C  13
              030A    B08D    LA      £-115,L
```



```
SUBI    83          soustraction entière, adressage global
                    indirect simple
LESW    I C 77      comparaison inférieur ou égal sur mot,
                    adressage local indirect dans le segment C
GOTO    7           branchement : l'étiquette est située trop
                    loin en arrière pour un branchement court
NEQH    C 13        comparaison différent sur octets, adressage
                    local direct dans le segment constant
```

```
SUBI 83
              030B    1A80    PSR     A
              030C    9DD3    LAD     -45,L
              030D    2BC1    LR      A,B
              030E    1660    LYI     96
              030F    1004    LAI     4
              0310    1C00    SVC     '0
              0311    AD90    STA     £-112,L
              0312    1B01    PLR     A
              0313    A890    SB      £-112,L
              0314    0200    JNV     $+4
              0315    1009    LAI     9
              0316    160A    LYI     10
              0317    2001    JMP     $+1
              0318    A599    BR      £-103,L
              0319    1C00    SVC     '0

   LESW I C 77
              031A    B5CD    CP      £-51,L
              031B    0500    JL      $+3
              031C    1000    LAI     0
              031D    2000    JMP     $+2
              031E    1001    LAI     1
GOTO 7
              031F    A5C3    BR      £-61,L

   NEQH C 13
              0320    8280    CPBY    -115,L
              0321    0200    JNE     $+3
              0322    1000    LAI     0
              0323    2000    JMP     $+2
              0324    1001    LAI     1
```

```
GRTC    I C 22 15   comparaison supérieur sur bits, adressage
                    local indirect dans le segment constant
:2                  définition d'étiquette
STOC    * X 17      rangement de bit, adressage global indirect
                    double post-indexé
INCR    W 127       addition immédiate au registre W
GOTO    9           branchement (en avant)
INDX    10 H 1      calcul d'indice dans un tableau d'octets
LDCN                chargement de la constante nil
```

```
    GRTC I C  22 15
              0325    B796    LB      £-106,L
              0326    28DF    TBT     31
              0327    1700    LBI     0
              0328    2D89    ADCR    B
              0329    2EC1    CPR     A,B
              032A    0300    JG      $+3
              032B    1000    LAI     0
              032C    2000    JMP     $+2
              032D    1001    LAI     1

     : 2
STOC * X 17
              032E    1A10    PSR     A,X,Y
              032F    9D91    LAD     -111,L
              0330    2BC1    LR      A,B
              0331    16D0    LYI     208
              0332    1001    LAI     1
              0333    1C00    SVC     '0
              0334    1B08    PLR     A,X,Y
INCR W 127
              0335    087F    ADRI    127,A


GOTO 9
              0336    2000    JMP     $+0
 INDX 10 H 1
              0337    37FF    LBI     -1
              0338    2C08    ADR     B,A
              0339    28D0    TBT     16
              033A    2D9B    ADCR    Y
              033B    2B01    SARS    1
              033C    170A    LBI     10
              033D    2C08    ADR     B,A
              033E    2BC2    LR      A,X
LDCN
              033F    30FF    LAI     -1
```



```
L       11          définition de numéro de ligne source Pascal
DIVR    I R 20      division réelle, adressage local indirect
                    dans le segment résident
LDCW    -13         chargement immédiat de mot
LODD    I C 1       chargement de double mot, adressage local
                    indirect dans le segment constant
ADDR    X C 30      addition flottante, adressage global indirect
                    simple post-indexé
GOTO    14          branchement (en arrière)
```

```
  L  11
DIVR I R 20
              0353    3800
              0354    6696    FDV     £-106,C
              0355    0500    JC      $+4
              0356    0200    JNV     $+6
              0357    1007    LAI     7
              0358    2000    JMP     $+2
              0359    1008    LAI     8
              035A    160A    LYI     10
              035B    1C00    SVC     '0


LDCW -13
              035C    30F3    LAI     -13
LODD I C 1
              035D    3800
              035E    A081    FLD     £-127,L
ADDR X C 30
              035F    3800
              0360    A29E    FAD     £-98,L
              0361    0500    JC      $+4
              0362    0200    JNV     $+6
              0363    1007    LAI     7
              0364    2000    JMP     $+2
              0365    1008    LAI     8
              0366    160A    LYI     10
              0367    1C00    SVC     '0
GOTO 14
              0368    2090    JMP     $-112
  ABSI
              034D    2100    JAGE    $+2
              034E    2E00    NGR     A,A
```

BIBLIOTHECAIRE

Les textes sources des deux passes du compilateur, et du moniteur d'exécution ont été stockés sur des disques T1600. Pour réaliser la mise à jour de ces fichiers, nous avons défini et réalisé un utilitaire de maintenance de textes sources, appelé bibliothécaire, et inspiré du produit UPDATE de CDC.

Comme son nom l'indique, un bibliothécaire est un utilitaire permettant de créer, d'utiliser et de mettre à jour des bibliothèques de programmes, qui peuvent être des bibliothèques du système ou des bibliothèques d'utilisateurs . Le contenu de ces bibliothèques peut être constitué de données, mais est en général constitué de programmes sous la forme source que le bibliothécaire permet de modifier et de présenter en entrée de compilateurs. Le bibliothécaire garde trace de toutes les modifications apportées, et évite donc de devoir conserver les anciennes versions d'un programme, comme ce serait le cas avec un éditeur de textes.

La bibliothèque de programmes que peut traiter le bibliothécaire est un fichier contenant des images de cartes sous forme condensée, et un historique relatif à celles-ci . Le niveau de travail le plus fin pour le bibliothécaire est la carte .

Chaque carte possède un identificateur unique constitué

- d'un nom identifiant la modification ayant introduit cette carte,
- d'un numéro de séquence différenciant les cartes ayant un même nom de modification .

Les modifications successives apportées à une bibliothèque de programmes ne sont généralement pas définitives : il est possible de revenir à un état antérieur .

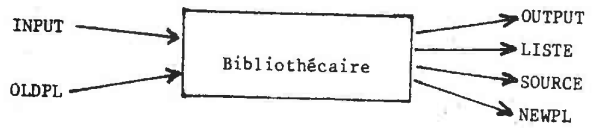
Le bibliothécaire est ainsi un système de mise à jour automatique, conservant l'historique complet des modifications apportées successivement au texte original . C'est un outil indispensable lors de l'écriture et la mise au point de gros programmes .

UTILISATION DU BIBLIOTHECAIRE.

Les éléments fournis au bibliothécaire sont de trois types :
- une bibliothèque à mettre à jour,
- des directives,
- des données.

Les directives se distinguent des données dans le flot d'entrée par la présence d'un caractère spécial en colonne 1 (en général /). Toute image de carte , repérée par un identificateur unique, peut se trouver dans deux états différents : active ou inactive . Les cartes en état inactif ne sont pas transmises sur les fichiers destinés à un listing ou à une entrée de compilateur .

LES FICHIERS UTILISES.



Chaque traitement ne nécessite pas obligatoirement l'apparition de tous ces fichiers . Seuls les fichiers INPUT et OUTPUT sont implicites . Les autres sont demandés explicitement par l'utilisateur.

OLDPL — Ce fichier contient la bibliothèque de programmes à modifier .

NEWPL — Contiendra la nouvelle bibliothèque mise à jour .

INPUT — Contient la description des modifications à apporter à OLDPL.

OUTPUT — Fichier sur lequel le bibliothécaire rend compte de toutes les opérations qu'il a effectué, avec ou sans succès.

SOURCE — Contiendra les cartes actives de NEWPL sous forme directement assimilable par un compilateur .

LISTE — Contiendra les cartes actives du fichier NEWPL, avec leur identification, dans un format destiné à une imprimante.

ØLDPL et NEWPL sont des fichiers indexés I(2), INPUT, ØUTPUT, SØURCE et LISTE sont soit des fichiers séquentiels, soit des unités symboliques.

IDENTIFICATION DES CARTES.

A chaque carte est associé de façon permanente un couple nom de modification.numéro de séquence .

Le nom de modification (20 caractères au plus) peut être :
- le nom du deck auquel appartient la carte . Il est fixé par une directive DECK.
- le nom de la modification si elle a été introduite ultérieurement. Ce nom est fixé par une directive IDENT.

Le numéro est un numéro de séquence à l'intérieur d'un deck ou d'une modification.

FORMAT DES DIRECTIVES.

/mot-clé paramètres

En colonne 1 figure un caractère de contrôle signalant une directive au bibliothécaire . Sa valeur par défaut est /.

Puis apparait un mot-clé en colonne 2, se terminant par un ou plusieurs blancs . Des paramètres peuvent suivre .

La dernière directive prise en compte par le bibliothécaire lors d'un traitement doit être /END

Par la suite, les conventions suivantes seront utilisées :

fnum — désigne un numéro de fichier . Le bibliothécaire ne fera que des lectures ou écritures sur ce fichier : c'est à l'utilisateur d'ouvrir, de fermer ou d'effacer ce fichier.

fnam désigne un nom de fichier, sous l'une des quatre formes :

- NOM
- NOM-CATALOGUE
- NOM,FU
- NOM-CATALOGUE,FU

par défaut : catalogue = [illegible]
fu = D4

SU désigne une unité symbolique

id est un nom de modification .

n est un numéro de séquence à l'intérieur d'une modification.

(Se reporter au manuel d'utilisation du système)

DIRECTIVES DE DECLARATION DE TRAITEMENT.

Ce sont les premières directives d'un traitement .

```
OLDPL fnum
OLDPL fnam
```

Traitement spécifié : mise à jour d'une ancienne bibliothèque .
Si le paramètre est 0, il n'y a pas d'ancienne bibliothèque.
Traitement par défaut : absence d'ancienne bibliothèque.

Paramètre : nom de l'ancienne bibliothèque .

```
EWPL 0
EWPL fnum
EWPL fnam
EWPL
```


Traitement spécifié : création d'une nouvelle bibliothèque . Si le paramètre est 0, pas de création .
Traitement par défaut : /NEWPL
Paramètre : nom de la nouvelle bibliothèque .
Valeur par défaut du paramètre : fichier spécifié par /ØLDPL

```
/INPUT fnum
/INPUT fnam
/INPUT SU
```

Traitement spécifié : les directives et les données sont à prendre sur le fichier ou l'unité spécifié .
Traitement par défaut : /INPUT SI
Paramètre : nom du fichier ou de l'unité d'entrée.

```
/OUTPUT fnum
/OUTPUT fnam
/OUTPUT SU
/OUTPUT 0
```

Traitement spécifié : un état sera imprimé sur le fichier ou l'unité spécifié .
Traitement par défaut : /OUTPUT LØ
Paramètre : fichier ou unité de sortie . Si 0, pas d'état imprimé.

```
/SOURCE

/SOURCE fnum
/SOURCE fnam
/SOURCE SU
```

Traitement spécifié : création d'un fichier destiné à un compilateur, reflétant l'état final de la NEWPL.
Traitement par défaut : pas de création de fichier.
Paramètre : nom d'unité ou de fichier .

Paramètre : nom de fichier ou d'unité .
Valeur par défaut du paramètre : SØ

```
/LISTE

/LISTE fnum
/LISTE fnam
/LISTE SU
```

Traitement spécifié : création d'un fichier destiné à une imprimante , contenant les cartes actives et leur identification, reflétant l'état final de la NEWPL.
Traitement par défaut : pas de liste.
Paramètre : nom de fichier ou d'unité .
Valeur par défaut du paramètre : LØ

La directive `/=c` apparaissant à un emplacement quelconque dans le flot d'entrée, indique que le caractère de contrôle prend la valeur c . La valeur par défaut est /

La dernière directive d'un traitement doit être : `/END`

CREATION D'UNE BIBLIOTHEQUE.

```
/DECK id
/DECK id fnum
/DECK id fnam
```

Cette directive permet d'initialiser la bibliothèque. Les cartes lues sur SI (1ère forme) jusqu'à la rencontre d'une directive /IDENT ou /END, sont introduites dans la bibliothèque. Si un nom de fichier est spécifié (2ème forme), toutes les cartes figurant sur ce fichier seront introduites dans la bibliothèque.

MODIFICATION D'UNE BIBLIOTHEQUE.

```
/IDENT id
```

Cette directive précède un jeu de directives de modification . Elle associe à la modification un nom (id) d'au plus 20 caractères . Ce nom ne doit pas encore être connu dans la bibliothèque ; il s'applique à toutes les images de cartes qui suivent, jusqu'à une directive /IDENT ou /END

AJOUT A PARTIR D'UNE AUTRE UNITE QUE INPUT.

```
/MERGE id.n fnum
/MERGE id.n fnam
```

Cette directive insère après la carte id.n toutes les cartes figurant dans le fichier spécifié par son fnum ou son fnam. D'éventuelles directives figurant sur ce fichier seront considérées comme des cartes de donnée.

CORRECTIONS.

```
/DELETE id
/DELETE id.n
/DELETE id.n id.n
```

Une modification, une carte ou une séquence de cartes peuvent être désactivées par cette directive . Les cartes de données suivant cette directive sont insérées après la dernière carte désactivée.

```
/RESTORE id
/RESTORE id.n
/RESTORE id.n id.n
```

Les cartes nommées sont activées à nouveau . D'autres cartes peuvent être insérées à la suite .

```
/BEFORE id.n
/AFTER id.n
```

Les cartes qui suivent (jusqu'à la prochaine directive autre que /=) sont isérées avant (première forme) ou après (deuxième forme) la carte désignée par id.n .

ETAT D'UNE CARTE.

```
/ETAT id
/ETAT id.n
/ETAT id.n id.n
```

Edition sur OUTPUT de toutes les modifications concernant la ou les cartes nommées .

SUPPRESSION PERMANENTE D'UNE MODIFICATION.

```
/PURGE id
/PURGE id *
```

Cette directive permet de supprimer de façon irréversible une modification . La forme id * supprime toutes les modifications apportées à la bibliothèque depuis la modification id . Cette forme permet de retrouver l'état de la bibliothèque à un moment donné, sauf si des modifications irréversibles y ont été apportées.

REORGANISATION D'UNE BIBLIOTHEQUE.

```
/CLEAN
```

Le fichier NEWPL ne contiendra que les images de cartes actives . L'on perd ainsi toutes les cartes inactives et l'historique des cartes actives . Cette réorganisation permet de réduire la taille physique d'une bibliothèque ayant atteint un certain niveau de stabilité.

COMPTE RENDU DE TRAITEMENT.

Le bibliothécaire transmet en fin de traitement un compte rendu au système d'exploitation . La commande IF permet de tester par la suite la validité du traitement . On trouvera

- 0 si le traitement s'est bien déroulé
- 1 si le bibliothécaire n'a pu effectuer tous les traitements demandés
- 2 si le bibliothécaire a été amené à corriger des directives erronées pour effectuer le traitement
- 3 si des erreurs graves se sont produites
- 9 en cas de désatre complet et irréversible

UTILISATION DU BIBLIOTHECAIRE.

```
CALL ARCHIV
```

XEMPLES

. La bibliothèque BRTZEL-QQ contient un programme PL16 que l'on veut compiler.

```
/CALL ARCHIV          }   appel du bibliothécaire
/ØLDPL BRTZEL-QQ      }
/SØURCE               }   création d'un fichier sur SØ
/END                  }   destiné à un compilateur
/CALL PL              }   transformation SØ → SI
/IPLC                 }   compilation
```

La bibliothèque de fnum 3 contient le texte :

```
IMPLICIT CØMPLEX(A-Z)          G       .1
BUFFER IN Q,D                  G       .2
END                            G       .3
```

et l'on veut insérer après G.2 la carte

```
DØ2I=Q+D
```

pour obtenir la bibliothèque de fnum 17, que l'on listera

```
/CALL ARCHIV

/ØLDPL 3
/NEWPL 17
/LISTE
/IDENT MØDIF1
/AFTER G.2
      DØ2I=Q+D
/END
```


Le fichier OUTPUT contiendra :

```
      >BIBLIOTHECAIRE : DEBUT
/OLDPL 3
>DECKS : G
/NEWPL 17
/LISTE
/IDENT MODIF1
/AFTER G.2
      DO2I=Q+D                          MODIF1          .1
/END

      >PAS D'ERREUR
      >NEWPL CREE : 17
      >LISTE : LO

      IMPLICIT COMPLEX(A-Z)             G               .1
      BUFFER IN Q,D                     G               .2
      DO2I=Q+D                          MODIF1          .1
      END                               G               .3
      >ACTIVES : 4
      >INACTIVES : O
      >DECKS : G        MODIF1
      >BIBLIOTHECAIRE : FIN
```

Directives de déclaration de traitement.

OLDPL	ancienne bibliothèque existante
NEWPL	nouvelle bibliothèque à créer
INPUT	unité de lecture des directives et données
OUTPUT	unité de sortie des messages
SOURCE	fichier destiné à une entrée de compilateur
LISTE	listing du contenu actif de la bibliothèque

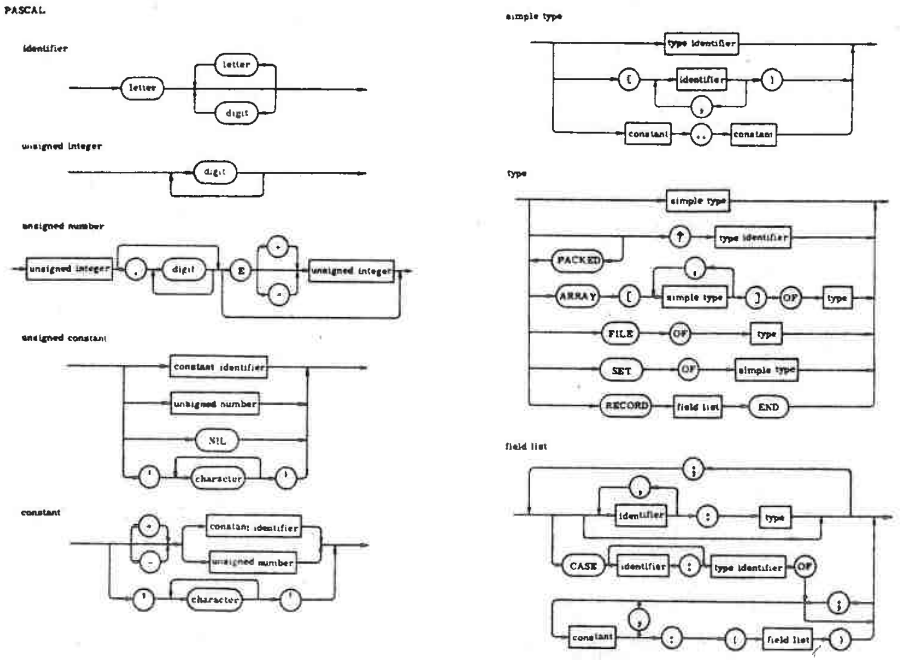
Directives de contrôle.

=	modification du caractère de contrôle
END	fin de traitement
CLEAN	effacement des cartes inactives dans la NEWPL

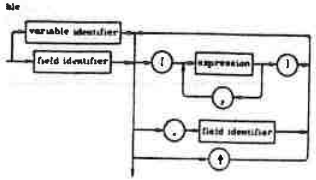
Directives de traitement.

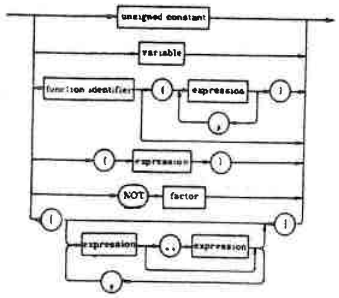
DECK	création d'une bibliothèque
IDENT	déclaration de modifications à une bibliothèque existante
PURGE	suppression permanente de DECKS
MERGE	fusion d'un fichier et de la bibliothèque
DELETE	désactivation de DECK ou de cartes
RESTORE	réactivation de DECK ou de cartes
BEFORE	ajout de cartes
AFTER	ajout de cartes
ETAT	édition des modifications successives subies par des cartes

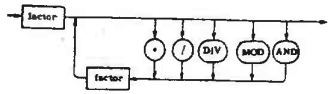
SYNTAXE DE PASCAL



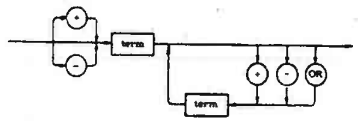
ble



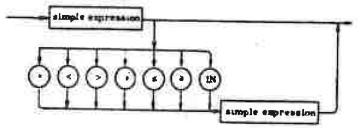




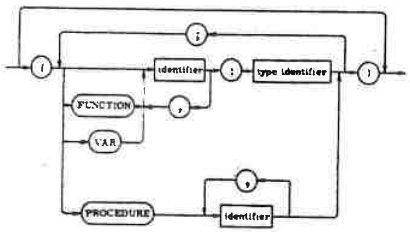
simple expression

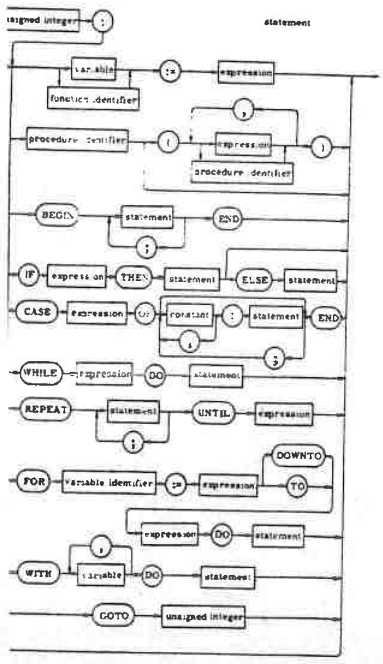


expression

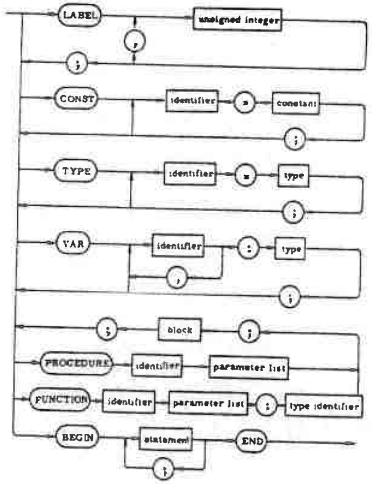


parameter list





block



program