Université de Montréal

Un générateur de code machine pour le compilateur Gambit

par

Sylvain Beaulieu

Département d'informatique et
de recherche opérationnelle

Faculté des arts et sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M. Sc.)
en informatique

Juin 1999

Page d'identification du jury

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Un générateur de code machine pour le compilateur Gambit

Présenté par :

Sylvain Beaulieu

a été évalué par un jury composé des personnes suivantes :

Guy Lapalme
(président du jury)

Michel Boyer

Marc Feeley
(directeur de maîtrise)

Mémoire accepté le : _____

Sommaire

Notre travail vise à étudier la possibilité d'avoir un ordonnanceur d'instructions général pouvant être utilisé avec plusieurs processeurs modernes. De plus, nous désirons écrire des générateurs de code natif pour le compilateur Gambit-RTL et y utiliser notre ordonnanceur. Avec ceci, nous désirons regarder les gains de performance que l'ordonnanceur nous offre dans le cadre d'un compilateur de code natif pour Scheme.

De plus, nous allons étudier quelques séquences de code pour en trouver des plus rapides. Entre autre, nous examinerons la création des constantes sur DEC Alpha et les sauts à l'aide d'un registre sur i386. Nous étudierons aussi la plupart des processeurs commerciaux sur différents angles. Nous allons examiner les unités de traitement, les registres, les branchements et quelques autres aspects.

Nous allons finir par montrer, à l'aide de tableaux, les résultats encourageants que nous avons obtenus selon les différentes optimisations utilisées (ordonnanceur, ancre et régions).

Liste des tableaux

Liste des Figures

1) Introduction

L'architecture des processeurs a évolué énormément depuis plusieurs années. Puisque le fait de simplement augmenter la vitesse de l'horloge n'est pas une chose élémentaire, les concepteurs de processeurs ont dû utiliser des techniques de traitement parallèle afin d'exécuter plus d'instructions en un temps donné.

Les processeurs modernes peuvent exécuter de 2 à 6 instructions en parallèle à chaque cycle d'horloge. De plus, les opérations à exécuter sont divisées en petites étapes, selon le principe de pipeline, ainsi le résultat d'une opération peut être disponible plusieurs cycles après le début de son traitement.

La recherche que nous avons effectuée vise à montrer qu'il est possible d'avoir un ordonnanceur d'instructions unique pour la plupart des processeurs modernes tout en étant relativement performant et simple.

1.1) Objectifs

Ce mémoire a pour but de montrer l'attrait d'avoir des générateurs de code natif ainsi qu'un ordonnanceur d'instructions universel dans le cadre du compilateur Gambit-RTL et d'un interpréteur Scheme multi-plateforme.

Un objectif secondaire est de trouver les meilleures séquences d'instructions machine pour chacun des générateurs de code pour optimiser les opérations qu'on retrouve dans des programmes Scheme typiques.

1.2) Motivation

Le système de programmation Gambit est constitué d'un compilateur et d'un interprète Scheme. La première version fut écrite en 1987. Cette version génère du code natif pour le processeur Motorola 68000. Évidemment, Gambit-68K était limité aux machines à base de ce processeur. Donc, en 1994, Gambit-C fut créé. Cette variante de Gambit génère du code C standard. Ceci permet de compiler le code généré sur n'importe quelle architecture disposant d'un compilateur C. Le fait de générer du code C a cependant provoqué, en moyenne, un ralentissement par un facteur de 2 du programme exécutable par rapport au compilateur Gambit-68K car certaines formes d'expressions Scheme ne se traduisent pas facilement en C. De plus, le compilateur Gambit-C est presque entièrement écrit en Scheme et génère des programmes en langage C ce qui lui permet de se compiler lui-même et d'être utilisé sur plusieurs plates-formes.

L'interprète Scheme exécute une boucle que l'on appelle *Read-Eval-Print*. Cette boucle débute par la lecture d'une expression et l'évalue. Lorsqu'il a terminé l'évaluation, l'interprète affiche le résultat et il retourne à l'étape de la lecture. Quant à lui, le compilateur, fait la lecture d'un fichier contenant un module Scheme et il génère du code en langage C. Il faut donc compiler le code C dans une deuxième étape et faire l'édition de lien avec d'autres modules compilés pour obtenir un programme exécutable.

Le grand avantage du compilateur par rapport à l'interprète est la vitesse d'exécution qui peut être de l'ordre de dix à cent fois plus rapide. Par contre, le compilateur Gambit-C est présentement totalement dépendant d'un compilateur C.

Pour des raisons évidentes de performance, nous désirons générer du code machine directement au lieu de générer un source en langage C et d'être obligé de compiler ensuite ce fichier. La compilation sera plus rapide et le programme exécutable aussi. Ceci impose d'avoir un générateur de code différent pour chacun des processeurs cibles mais ceci permet la génération de code spécialisé. C'est-à-dire que le compilateur peut générer des séquences de code machine qu'il serait impossible d'obtenir en passant par un compilateur C (ou autre).

L'utilisation d'un interprète permet une mise au point interactive mais l'exécution n'est pas rapide. Il serait avantageux de modifier l'interprète pour qu'il génère du code machine et exécute ce code si cela peut être fait assez rapidement. De cette façon, nous avons les avantages du mode interactif de l'interprète ainsi que la vitesse d'exécution du compilateur. Nous devons donc disposer de générateurs de code rapides pour les machines cibles.

Générer un code naïf et lent (ce qui pourrait se faire rapidement) ne serait pas intéressant puisque nous n'aurions aucun avantage par rapport au fait de générer du code C. Donc, nous devons aussi inclure des optimisations pour tous les processeurs dont on aura un générateur de code natif.

Le choix du langage Scheme pour les générateurs de code et pour les étapes d'optimisation s'est imposé pour des raisons de portabilité et de « *bootstrap* ». Une grande partie du compilateur Gambit-C et de l'interprète est écrit en Scheme. Donc, les nouveaux modules doivent être écrits dans le même langage si l'on veut que le tout soit portable.

De plus, il serait bon d'avoir tous les générateurs de code simultanément disponibles sur toutes les plate-formes. C'est-à-dire que le fait d'avoir un « *cross-*

compiler » est utile pour plusieurs raisons et nous devons voir à ce que tous les générateurs de code puissent être utilisables quelle que soit la machine hôte.

Pour les fins d'évaluation de performances et de faisabilité nous voulons expérimenter avec des processeurs cibles assez diversifiés. Par exemple, des processeurs avec des mots de longueur différente sont utilisés car l'alignement et les instructions requises pour les accès à la mémoire sont différents. De plus, des processeurs de types CISC et RISC doivent être examinés à cause des différences fondamentales entre ces types :

- Le nombre d'instructions disponibles est relativement différent.
- La longueur des instructions (en octet) est fixe pour les RISC et variable pour les CISC.
- Les modes d'adressages disponibles ne sont pas les mêmes.
- Normalement, sur les RISC, les opérations ne sont effectuées qu'entre registres tandis que sur les CISC, les opérations peuvent être aussi exécutées entre les registres et la mémoire.
- Les instructions doivent généralement être alignées sur un certain multiple pour les RISC mais ceci n'est pas applicable pour les processeurs CISC.

1.3) Problématique

Un ordonnanceur est un programme qui change l'ordre des instructions d'une séquence d'instructions pour en améliorer la vitesse d'exécution. L'ordonnanceur que nous voulons obtenir doit répondre à certains critères de base. Il est à noter que les générateurs de code doivent répondre aux mêmes critères. Nos critères sont les suivants :

- *Réutilisabilité* : Nous voulons que l'ordonnanceur puisse être utilisé dans tout autre projet qui a recours à un algorithme d'ordonnancement. Ceci pourrait être un autre compilateur ou bien un programme qui a besoin d'ordonnancer des tâches. Ceci doit pouvoir se faire sans avoir à modifier (ou très peu) le source de l'ordonnanceur.
- *Applicabilité à Scheme* : Nous savons que le style de programmation employé avec le langage Scheme est différent des langages conventionnels comme le C ou le COBOL. Nous devons tenir compte de ce fait lors de l'écriture de l'ordonnanceur pour qu'il soit utilisable principalement dans le cadre d'un compilateur Scheme.
- *Performance* : L'ordonnanceur doit exploiter au maximum le processeur cible pour obtenir un code rapide. De plus, le temps pris par l'ordonnanceur pour effectuer sa tâche ne doit pas être trop élevé.

Il existe quelques algorithmes d'ordonnancement publiés mais ils comportent un bon nombre de lacunes relatives à nos critères de base. Certains sont spécialisés pour un type de processeurs. D'autres peuvent demander beaucoup d'informations (matrice de latence d'instructions, graphe de flot de données, graphe de flot d'exécution, ...), possiblement non disponibles, de la part de celui qui veut utiliser l'ordonnanceur ce qui aurait comme effet de limiter la réutilisabilité à d'autres processeurs.

Les modules de génération de code du compilateur GCC, un compilateur C des plus utilisé, sont écrits en C et ils sont très dépendants de la structure du langage C. Ils contiennent aussi du code C à insérer dans le compilateur pour l'aider lors de la génération du code. De plus, ils génèrent du code assembleur ce qui impose d'avoir aussi un assembleur pour chaque processeur.

Nous devons donc trouver une façon d'écrire des générateurs de code modulaires. De plus, il nous faut trouver un principe pour l'écriture de l'ordonnanceur d'instructions pour qu'il soit simple à intégrer dans les générateurs de code futurs ou dans d'autres projets.

1.4) Méthodologie

Nous avons débuté par une étude approfondie de certains processeurs modernes, de types RISC et CISC. Il nous a fallu étudier plusieurs caractéristiques comme le nombre de registres disponibles, le nombre d'unités de traitement, l'exécution en ordre et en désordre des instructions ainsi que plusieurs autres aspects mineurs. Après cette étude, nous avons pu choisir six processeurs différents pour l'écriture de générateurs de code. Les processeurs choisis sont le Pentium (i386 et ses successeurs), le M68k de Motorola, le DEC Alpha 21X64, le Sparc de Sun, le MIPS R2000 ainsi que le PowerPC. Nous pensons que ces six processeurs sont assez différents entre eux et qu'ils incluent l'ensemble des caractéristiques de base des processeurs. Par la suite, nous avons procédé à l'écriture de ces six générateurs de code natif.

Après avoir effectué quelques essais sur chacun de ces processeurs, nous avons cherché des séquences d'instructions machine plus rapides pour remplacer les plus problématiques. Nous avons principalement dû trouver une façon d'optimiser les accès à la mémoire sur DEC Alpha ainsi que les branchements sur Pentium.

Nous avons ensuite procédé à la recherche de programmes d'ordonnancement d'instructions. Nous avons étudié plusieurs textes traitant d'ordonnanceurs ainsi que l'ordonnanceur utilisé par le compilateur GCC. Suite à cette étude, nous sommes passés à l'étape de l'écriture d'un ordonnanceur général en Scheme. Cet

ordonnanceur a subi plusieurs modifications au cours des tests pour en améliorer les performances.

Lorsque le tout était stable, nous avons inclus l'ordonnanceur dans une version épurée du compilateur Gambit et nous l'avons appelé Gambit-RTL. Ce compilateur ne contient pas de glaneur de cellule (GC) pour récupérer l'espace mémoire et la librairie de fonctions ne contient qu'un petit sous-ensemble des fonctions prédéfinies de Scheme. Nous avons aussi inséré l'ordonnanceur dans les générateurs de code et nous avons procédé à quelques essais sur des petits programmes pour trouver les points faibles des générateurs de code et les corriger. Finalement, nous avons testé le système complet sur un jeu de programmes pour en analyser les performances, en particulier comparativement à Gambit-C.

2) Description de l'objectif

Dans ce chapitre, nous donnons un aperçu de l'environnement dans lequel les recherches se sont effectuées. Nous commençons par une description des processeurs et, en particulier, ceux que nous avons étudiés. Ensuite, nous allons présenter les instructions virtuelles disponibles dans les générateurs de code. Finalement, nous montrerons les principaux problèmes, causés par le code virtuel, que nous devons résoudre lors de l'écriture des générateurs de code.

2.1) Les processeurs

Les processeurs évoluent très rapidement. Ils ont passé de dizaine de milliers de transistors à plusieurs millions en une décennie. De nos jours, il est courant qu'un processeur puisse effectuer des calculs matriciels ainsi que des calculs reliés à l'infographie. De plus, la dimension de leur mémoire cache est plus grande que la mémoire principale des ordinateurs utilisés au début des années 1980.

Les processeurs les plus complexes peuvent exécuter les instructions dans un ordre autre que celui qui est inscrit dans le programme (exécution « en désordre »). Ces processeurs ont une unité de traitement spéciale qu'on appelle « bassin d'instructions » et qui contient une liste d'instructions qui doivent être exécutées. À chaque cycle, l'unité extrait une ou plusieurs instructions prêtes pour l'exécution. Dans la limite du possible, ces instructions sont choisies de façon à ne pas causer un arrêt momentané des pipelines. De plus, les dépendances de données entre les instructions doivent toujours être respectées. Tous les processeurs ne possèdent pas nécessairement une unité d'exécution en désordre mais celle-ci semble devenir de plus en plus courante.

Certains processeurs ont des registres internes en plus grand nombre que les registres visibles au programmeur. Ces processeurs associent dynamiquement les registres internes aux registres visibles (avec possiblement plus d'un registre interne par registre visible), ce qui permet de réduire le nombre de dépendances et ainsi accélérer l'exécution du code. Dans l'exemple qui suit, nous pouvons voir que la séquence de code originale, à gauche, doit être exécutée séquentiellement à cause du choix des registres. Nous constatons que la séquence de droite peut être exécutée en parallèle sur plusieurs unités de traitement.

	Code original	Code avec changements de registres IRx : Registre interne.
1	R3 ← MEM(0)	IR3 ← MEM(0)
2	R1 ← MEM(1)	IR1 ← MEM(1)
3	R2 ← R3 + R1	IR2 ← IR3 + IR1
4	MEM(2) ← R2	MEM(2) ← IR2
5	R2 ← MEM(3)	IR4 ← MEM(3)
6	R4 ← R3 + R2	IR5 ← IR3 + IR4
7	MEM(4) ← R4	MEM(4) ← IR5

Dans le code original, nous voyons que l'instruction 5 doit attendre que l'instruction 4 ait été exécutée avant de débuter. Ceci n'est plus vrai dans la colonne de droite ce qui permet de lancer les instructions 5 à 7 sans avoir besoin d'attendre l'instruction 4.

Presque tous les processeurs modernes possèdent une unité de prédiction des branchements. Le coût associé à un arrêt et une reprise du pipeline à cause d'un branchement est fort élevé. Pour qu'un branchement conditionnel puisse être exécuté, le résultat de la condition doit être connu. Sur certains processeurs, l'instruction de comparaison précède de peu l'instruction de branchement. Dans ce cas, le résultat ne sera probablement connu qu'après l'entrée dans un des pipelines de l'instruction de branchement. À l'étape de la prise de décision, si le branchement est pris, le pipeline doit être vidé pour reprendre l'exécution à l'endroit demandé par le branchement. Ceci implique donc d'aller chercher les nouvelles instructions à la nouvelle adresse, les mettre dans la queue d'exécution et ensuite recommencer l'exécution avec les nouvelles instructions. À cause de ceci, il peut y avoir beaucoup de délais dans l'exécution. Donc, une unité de prétraitement des instructions est souvent ajoutée. Cette unité examine les instructions avant qu'elles entrent dans un des pipelines pour trouver les instructions de branchement. Lorsque l'unité en trouve une, elle essaie de prévoir le résultat de la comparaison avant qu'elle en connaisse le résultat. Il existe quelques algorithmes qui sont utilisés pour prédire les branchements :

- *Historique* : L'unité peut garder un historique des derniers sauts. Lorsqu'une instruction de branchement est analysée, l'unité détermine, en consultant l'historique, si ce saut a récemment été exécuté et, dans ce cas, continue l'exécution à la même adresse de destination que la dernière fois.
- *Prédéfini*: Selon le type de branchement, l'unité prédit que le saut sera pris si l'adresse de la destination du branchement est inférieure à l'adresse de l'instruction courante tandis que l'exécution se poursuit normalement dans le cas contraire.
- *Compilateur* : Des bits d'informations sont inclus dans les instructions de branchement pour aider le processeur à faire ses prédictions.

Les processeurs utilisent normalement une combinaison d'algorithmes. Selon le résultat de la prédiction, l'unité continue de charger les instructions à l'adresse prédite. Lorsque l'unité ne fait pas la bonne prédiction, la méthode traditionnelle est utilisée, c'est-à-dire de vider les pipelines et de recharger les nouvelles instructions à partir de la bonne adresse.

Un petit nombre de processeurs utilisent la technique du branchement à retardement « *delayed-branch* » pour minimiser les pertes de cycles imposées par les instructions de sauts. Cette technique consiste à placer un certain nombre d'instructions après l'instruction de saut en sachant qu'elles seront exécutées avant que le saut prenne effet. Puisque l'obtention d'un résultat pour une instruction peut prendre plus d'un cycle, les quelques instructions qui suivent l'instruction de saut peuvent avoir débutées leur exécution dans les pipelines. Au lieu de rejeter ces instructions, le processeur les garde et continue l'exécution à la position demandée par le saut. Dans l'exemple qui suit, nous voyons qu'il est possible d'exécuter deux instructions avant que l'adresse de branchement soit connue :

JMP dest	Lecture	Exécution	Écriture		
ADD R1,R2		Lecture	Exécution	Écriture	
ADD R3,R4			Lecture	Exécution	Écriture

Figure 1 : Pipeline d'exécution

2.1.1) Les registres

Tous les processeurs possèdent des registres. Certains en ont plus que d'autres et ils peuvent être plus ou moins spécialisés. Les trois types de registres principaux sont :

- *Général* : Ces registres sont utilisés pour les opérations principales comme l'addition et les opérations logiques.
- *Dédié* : Pour certaines opérations, nous devons utiliser des registres spécifiques. Parfois ces registres peuvent être dédiés à une ou plusieurs opérations tandis qu'ils peuvent être considérés comme des registres généraux pour les autres opérations.
- *Spécialisé* : Ces registres sont utilisés pour des tâches spécifiques comme la gestion de la pile et la sauvegarde des résultats d'une comparaison.

Les processeurs de type RISC ont généralement 32 registres de base et les CISC les plus répandus ont entre 8 et 16 registres. Par exemple, le Pentium n'a que 8 registres de 32 bits qui sont assez particuliers car les opérandes de plusieurs

instructions comme la division et le décalage de bits doivent absolument être situées dans un des registres dédiés [LHR92]. Pour l'instruction de décalage de bit par un nombre variable nous devons absolument mettre ce nombre dans le registre ECX (« *counter* »). D'un autre coté, le DEC ALPHA 21064 possède 32 registres 64 bits très généraux [AAH]. Le seul registre qui est spécialisé est le R31 qui contient toujours la valeur 0 même après une écriture. Les autres registres peuvent être utilisés pour toutes les opérations.

Le système d'exploitation peut limiter un peu le choix des registres. Il peut diviser les registres en plusieurs classes et imposer certaines règles lors de l'utilisation des registres. Les classes couramment définies sont :

- *Temporaire* : Ces registres sont utilisés pour stocker des résultats intermédiaires d'un calcul.
- *Temporaire sauvegardé* : Ces registres sont comme les registres « temporaires » mais en plus, ils ne doivent pas être modifiés par les fonctions appelées. Si une fonction a besoin de modifier un de ces registres, elle doit sauvegarder la valeur du registre et la restaurer à la fin.
- *Arguments* : Ces registres servent à passer les paramètres d'un appel de fonction dans des registres. La fonction appelée peut modifier ces registres au besoin.
- *Réservé* : Le système d'exploitation peut se réserver un ou plusieurs registre(s) pour son usage interne.

Il est clair qu'à l'intérieur d'un programme, nous pouvons nous déroger de ces classes et utiliser la plupart des registres comme bon nous semble. Aussitôt que nous voulons appeler une fonction externe au programme qui utilise les conventions des systèmes d'exploitation, nous sommes tenus de respecter les classes.

2.1.2) ALU

En plus d'avoir mis en pipeline les unités de traitements, les concepteurs de processeurs ont ajouté plusieurs unités aux nouvelles versions de leurs processeurs. Ces nouvelles unités peuvent être le résultat d'un duplicata d'une unité déjà existante, la création d'une unité spécialisée ou bien la division d'une unité existante en deux unités spécialisées. Par exemple, le 80486 d'Intel a une unité de traitement qui s'occupe de toutes les opérations sur les entiers. Dans le Pentium, Intel a dupliqué cette unité en deux pipelines (U et V) non symétriques. Le pipeline U a la possibilité de faire des rotations de bits contrairement au pipeline V. Puisque les deux unités ne sont pas symétriques, le code généré doit être ordonnancé de façon à placer le bon type d'instructions dans le bon pipeline. Dans le Pentium II, Intel a modifié ses pipelines de façon à les rendre symétriques et il a aussi doublé l'unité de

traitement des nombres à virgule flottante. De cette façon, aussitôt qu'une unité d'un certain type devient libre, une instruction du même type peut y entrer.

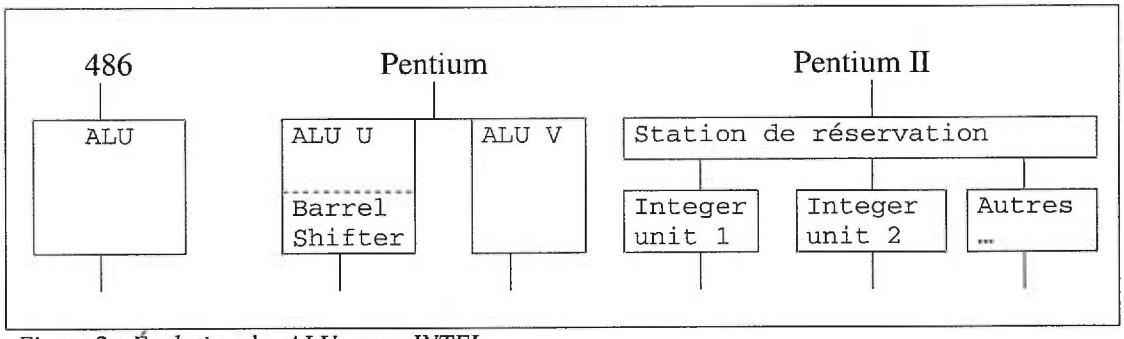

Figure 2 : Évolution des ALUs pour INTEL

Le processeur 68060 de Motorola est presque dans la même situation que le Pentium. Les instructions doivent être regroupées par paire lorsque ceci est applicable. De plus, comme pour le Pentium, quelques instructions plus complexes ne peuvent aller que dans un des deux pipelines [MUM]. Dans le PowerPC 604, trois unités se chargent des calculs sur les entiers [PPC1]. Les deux premières unités s'occupent des instructions qui s'exécutent en un cycle et la dernière unité s'occupe des instructions qui s'exécutent en plus d'un cycle (par exemple la division et la multiplication).

2.1.3) Expéditeur « *Dispatcher* »

Le module d'expédition des instructions est devenu assez complexe à cause de l'évolution des unités de traitements. Il doit déterminer la destination de plusieurs instructions simultanément tout en empêchant les conflits entre les instructions. Le processeur peut procéder de trois façons différentes :

1. Exécution par bloc fixe
2. Exécution par bloc dynamique
3. Exécution par instruction

Ceci dépend principalement du type de processeurs (CISC ou RISC).

Généralement, les processeurs RISC analysent les instructions par groupe de taille fixe. Lorsqu'il n'y a pas de conflits entre les instructions des groupes, il exécute toutes les instructions dans le même cycle. Par contre, dans le cas contraire, il exécute dans le premier cycle le préfixe qui précède le premier conflit et il recommence les vérifications avec les instructions restantes du groupe dans le prochain cycle. Par exemple, pour le bout de code de 12 instructions suivant, nous pourrions obtenir cette exécution si le processeur est capable d'exécuter trois instructions par cycle :

Code (op dest,source1,source2)	Cycle
1. R1 ← 10 2. R2 ← 20 3. R3 ← 30	1
4. **R4** ← R1 + R2 5. **R5** ← R2 *XOR* R3	2
6. R6 ← **R4** - **R5**	3
7. **R7** ← R4 * R5	4
8. R8 ← **R7** + R2 9. R9 ← R3 + R2	5
10. R10 ← R7 *XOR* R1 11. R11 ← R8 - R2 12. R12 ← R9 + R3	6

Tableau 1 : Exemple d'exécution par bloc fixe

Nous pouvons voir que les instructions 4 et 5 sont en conflit avec l'instruction 6 à cause des registres R4 et R5. Dans le 3e bloc, nous avons un conflit entre les instructions 7 et 8 à cause du registre R7. Donc, l'instruction 8 doit attendre un cycle avant d'être exécutée ainsi que l'instruction 9.

Les processeurs CISC exécutent, normalement, les instructions par blocs dynamiques. La vérification des conflits est la même mais la différence est au niveau de la reprise lors du cycle suivant. Au lieu de continuer les vérifications avec les instructions restantes dans le bloc, les CISC peuvent généralement recommencer avec un bloc complet. Ceci permet donc plus de parallélisme au niveau du code. Pour le même exemple que tantôt, nous obtiendrions l'exécution suivante (pour des blocs de taille 3) :

Code	Cycle
1. R1 ← 10 2. R2 ← 20 3. R3 ← 30	1
4. **R4** ← R1 + R2 5. **R5** ← R2 *XOR* R3	2
6. R6 ← **R4** - **R5** 7. **R7** ← R4 * R5	3
8. R8 ← **R7** + R2 9. R9 ← R3 + R2 10. R10 ← R7 *XOR* R1	4
11. R11 ← R8 - R2 12. R12 ← R9 + R3	5

Tableau 2 : Exemple d'exécution par bloc dynamique

Après le premier conflit de registres à l'instruction 6, l'expéditeur examine le bloc de trois instructions débutant avec l'instruction 6. C'est la même chose pour le deuxième conflit à l'instruction 8. Le processeur examine ensuite les instructions 8 à 10.

L'exécution par instructions est utilisée dans les processeurs plus anciens et plus simples. Chaque instruction est prise à tour de rôle et elle est exécutée une à la fois. Ceci est égal à une exécution par blocs de taille un.

2.1.4) Unité de décodage

L'unité de décodage des instructions sur les RISC est relativement simple. Celle dans les CISC est un peu plus complexe dû au format des instructions qui est souvent irrégulier. Il existe un processeur CISC qui sort totalement des normes : le Pentium II qui est très différent des autres. Le cœur du processeur est presque RISC mais le format des instructions en entrée est très variable en longueur ainsi qu'au niveau du décodage des bits. Par exemple, il y a des instructions qui prennent un octet (sans compter les paramètres) et d'autres qui prennent deux octets. Une instruction, en comptant les paramètres, peut avoir une longueur d'un octet jusqu'à une quinzaine d'octets. Pour palier à cette situation, Intel a ajouté un niveau de décodage. Ce niveau transforme les instructions complexes en plusieurs instructions très simples. Pour ce faire, Intel a conçu trois unités de conversion des instructions CISC vers des « *micro-ops* ». La première unité est capable de convertir les instructions simples (par exemple, une addition de registre à registre) et complexes (par exemple, un chargement en mémoire). Elle peut émettre jusqu'à quatre « *micro-ops* » à chaque cycle. Les deux autres unités ne peuvent accepter que des instructions simples qui ne requièrent qu'une « *micro-ops* ». De plus, ces trois unités sont ordonnées « complexe, simple, simple » ce qui implique que nous devons ordonnancer le code de façon à respecter cet ordre si nous voulons maintenir l'unité de décodage des instructions à sa pleine vitesse.

Lors du décodage, quelques processeurs en profitent pour changer les registres utilisés par des registres internes. Le 21264 contient 80 registres internes pour les entiers et 72 pour les nombres flottants. De son coté, le Pentium II possède un total de 40 registres internes qui peuvent être utilisés autant pour les nombres flottants que pour les entiers.

2.1.5) Exécution

Anciennement, les processeurs exécutaient une instruction à la fois. Ils regardaient tout simplement l'instruction suivante et l'exécutaient. Par contre, les nouveaux processeurs tendent à grouper les instructions et à en choisir un sous-

ensemble (pas nécessairement un préfixe) qui peuvent être exécutées, selon certains critères, simultanément. Le 21264 de Digital et le Pentium II d'Intel font partie de ceux qui ont la capacité d'exécuter dans un ordre autre que celui spécifié par le programme.

Le Pentium II regarde dans un pool de « *micro-ops* » pour trouver les *instructions* qui n'ont pas de dépendances. À chaque cycle, il peut distribuer jusqu'à cinq « *micro-ops* » aux unités d'exécution. Lorsque le résultat est obtenu, il est envoyé dans une section du pool pour ensuite être retiré dans l'ordre normal du programme (pour respecter la logique séquentielle du programme). Trois résultats à la fois peuvent être retirés du pool [IIP].

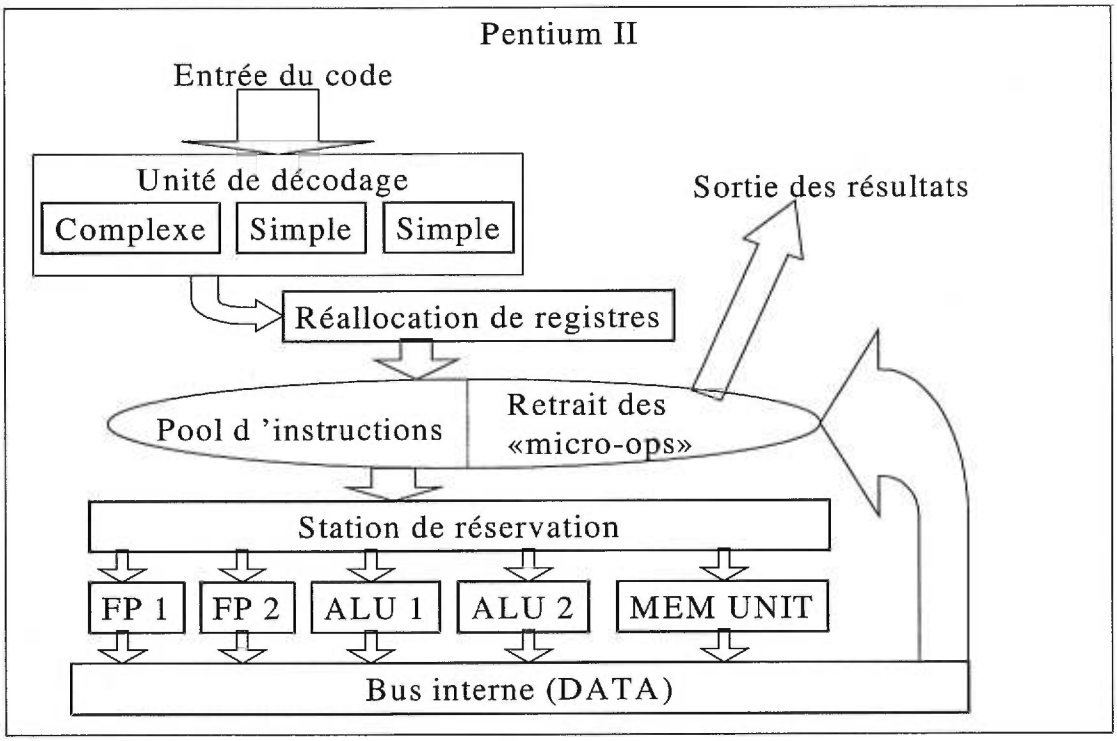

Figure 3 : Unité d'exécution du Pentium II

De son coté, le 21264 décode quatre instructions à la fois et les distribue aux deux queues d'exécution. La première queue s'occupe des instructions sur les entiers et la deuxième queue s'occupe des instructions sur les nombres flottants. Puisqu'il y a quatre unités de traitement des entiers et deux unités pour les nombres flottants, un maximum de six instructions peuvent débuter leur exécution à chaque cycle. Ces instructions sont choisies d'une façon à favoriser les instructions les plus âgées.

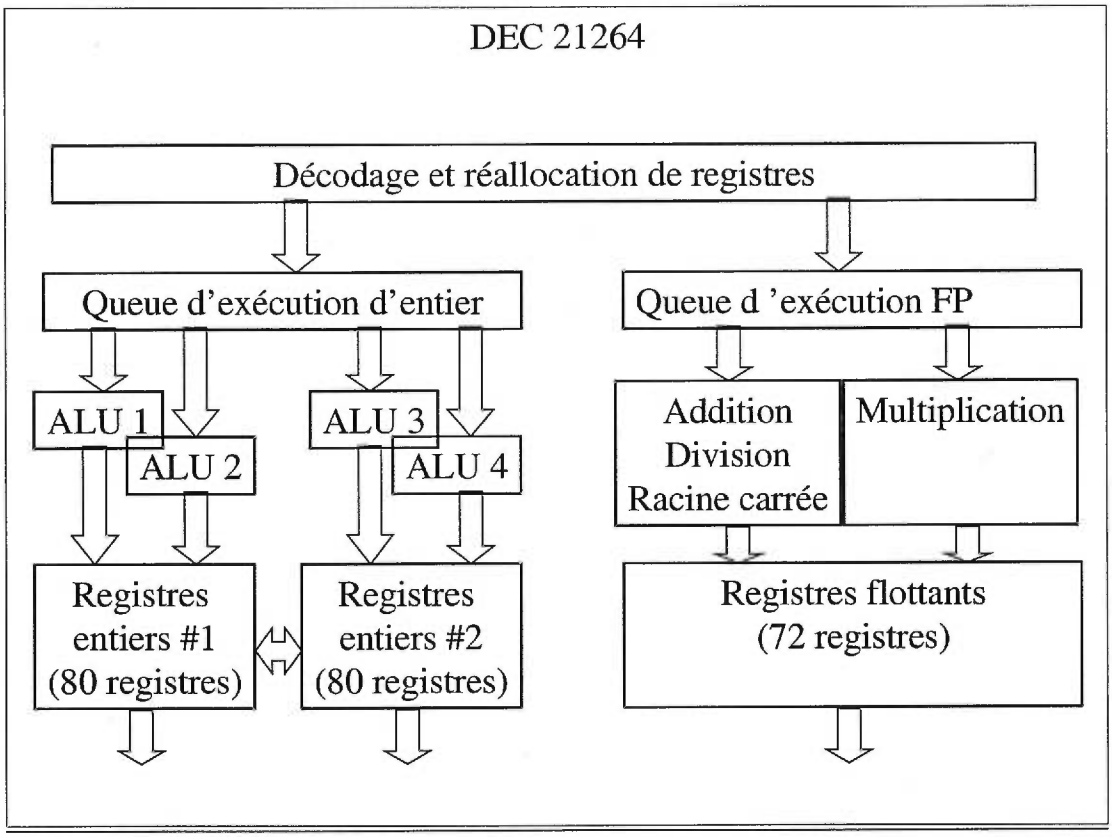

Figure 4 : Unité d'exécution du 21264

Il est à noter que les deux unités qui renferment les 80 registres d'entiers sont identiques. Lorsqu'une unité modifie un des 80 registres, elle demande à l'autre unité d'en faire de même. Ceci a été fait ainsi pour réduire la taille des bus internes [MR96].

2.2) Contraintes propres à Scheme

Le langage Scheme est non typé ce qui implique qu'un système doit permettre de détecter le type des variables lors de l'exécution et des GC. Dans le compilateur Gambit-RTL, les deux bits les moins significatifs sont réservés pour identifier le type principal des objets. Les quatre types possibles sont :

- 00 (fixnums) : Les entiers qui entrent sur un mot machine (moins les deux bits requis pour le type).
- 01 (sous-type) : Tous les autres types (vecteurs, procédures, ...).
- 10 (spécial) : Contient les constantes Scheme (#t, #f, (), #!eof, ...) et les objets de type caractères.

- 11 (pair) : Pointeur vers un objet de type paire.

Les procédures sont des objets Scheme et doivent donc être représentées par des pointeurs typés. De plus, le GC traverse la pile et les registres où peuvent se trouver des adresses de retour qui doivent donc aussi être représentées par des pointeurs typés. Cela rend l'implantation des branchements aux procédures et adresses de retour plus complexe que dans les langages conventionnels.

Le standard Scheme demande que les symboles soient alloués de façon unique. C'est-à-dire que si deux modules (`M1` et `M2`) utilisent le symbole `X` comme une constante, il faut que `M1` et `M2` contiennent des références vers le symbole `X` alloué à un endroit unique. De cette façon, `(eq? V 'X)` se compile en une comparaison d'adresse.

2.3) RTL

Dans le but de simplifier l'écriture des générateurs de code, un langage pseudo assembleur « RTL » a été conçu. RTL se situe au même niveau que l'assembleur de la plupart des machines. Le mot RTL est l'abréviation de « *Register Transfer Language* ». Il comprend des opérations entre registres, des branchements conditionnels et inconditionnels, des accès à la mémoire ainsi que des instructions de *gestion* du code. Chaque générateur de code doit implanter toutes les instructions. Les générateurs de code doivent générer des séquences d'octets qui représentent le code machine requis, sur la machine cible, pour exécuter l'instruction RTL. Les instructions RTL sont faites de façon à être très proches des machines RISC. La plupart des instructions RTL correspondent à une seule instruction machine sur les RISC.

Il est à noter que l'implantation des instructions RTL est laissée au développeur ce qui implique que les optimisations au niveau du code machine doivent être faites dans les générateurs de code machine.

Ainsi, la compilation d'un programme est composée de trois étapes :

1. Traduction de Scheme à GVM (la machine virtuelle de Gambit).
2. Traduction de GVM à RTL.
3. Traduction de RTL au code machine de la machine cible. Cette étape fait aussi l'écriture du code machine dans un fichier ainsi que l'écriture de la table d'édition de liens.

La figure qui suit nous montre les étapes pour compiler trois modules Scheme.

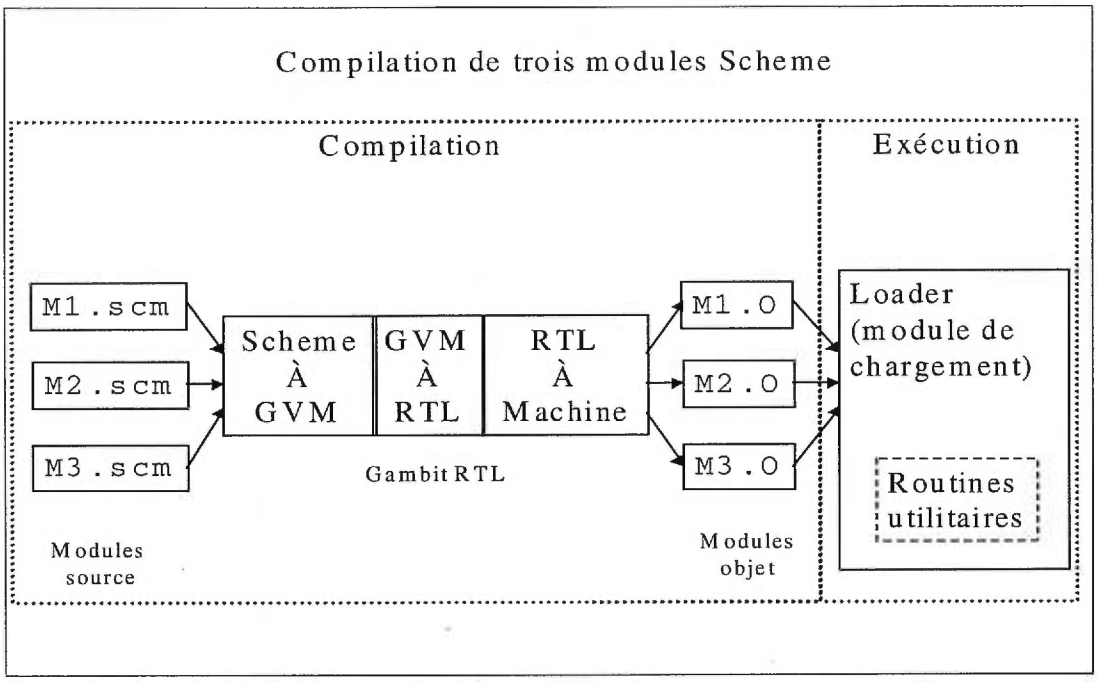

Figure 5 : Compilation de trois modules Scheme

2.3.1) Le loader (Module de chargement)

Le loader est un programme écrit en C qui effectue le chargement des modules compilés, les prépare pour l'exécution et lance l'exécution. De plus, le loader contient des routines utilitaires de bas niveau ainsi qu'un glaneur de cellules.

Pour chaque générateur de code disponible, nous devons avoir une section dépendante de ce générateur dans le loader. Ces sections s'occupent d'initialiser les modules et de les exécuter. La liste des fonctions qui doivent être définies pour chacun des générateurs de code est la suivante :

`Target_dependent_setup`	Permet d'initialiser certaines variables qui peuvent être utilisées par le module. Par exemple, on peut créer une variable globale qui donne le point d'entrée d'une fonction pour les opérations arithmétiques d'un processeur.
`Start_scheme`	Initialise les registres et lance l'exécution d'un module compilé. Cette fonction est très dépendante du code généré et doit être codée en assembleur.
`Stop_scheme`	Arrête l'exécution du module.
`Cache_flush`	Vide le contenu des caches. Puisque l'édition des liens se fait au chargement, la mémoire cache doit être invalidée avant de lancer l'exécution.

Tableau 3 : Instructions dépendantes.

Le loader débute par le chargement des modules. Pour chaque module, une zone de mémoire est allouée et le module est chargé dans cette zone de mémoire. La section pour l'édition des liens (étape ultérieure) est chargée dans une zone différente du code.

La fonction d'initialisation des variables pour ce processeur est appelée (`Target_dependent_setup`). Ceci permet d'initialiser des variables ou d'effectuer des opérations qui sont propres à ce processeur.

Après avoir chargé tous les modules en mémoire, l'édition de liens ajuste dans le code les pointeurs vers des variables globales ainsi que vers les étiquettes publiques dans chacun des modules. Ceci se fait en deux étapes. L'étape un consiste à la récolte de toutes les variables/symboles globaux ainsi qu'à la création d'une case mémoire pour chacune des variables. Ceci est fait pour tous les modules avant de passer à l'étape deux.

La seconde étape construit les adresses demandées par les tables de liaison de liens et écrit ces adresses directement dans le code des modules. Selon le processeur et la section (code ou data), la construction des adresses peut varier. Par exemple, sur DEC Alpha, une adresse complète (64 bits) qui doit être inscrite dans le code nécessite un fractionnement en quatre parties de 16 bits (chaque partie vient se placer dans une des quatre instructions générées par le compilateur). Par contre, le Pentium est capable de recevoir une adresse de 32 bits, sans fractionnement, dans une instruction.

Ensuite, les caches d'instruction et de données sont invalidées. Puisque le code est modifié directement en mémoire, nous devons vider les caches de données et d'instructions pour qu'ils tiennent compte des nouveaux changements.

La dernière étape consiste au lancement des modules. Pour ce faire, la procédure Scheme «;kernel» est recherchée parmi tous les symboles globaux et l'exécution débute à son adresse.

2.3.2) Entête

Pour aider la génération du code RTL, le compilateur doit avoir certaines informations sur la machine cible. Ces informations sont fournies par le module qui implante le traducteur de RTL au code de la machine cible. Le nombre de registres doit être connu pour faire l'allocation des registres. Le registre utilisé pour la gestion de la pile doit être spécifié. Étant donné que chaque processeur code ses instructions avec un nombre fixe de bits, nous devons indiquer la distance relative maximale permise pour les accès à la mémoire. Par exemple, si les instructions d'un processeur disposent de 16 bits pour faire un accès à la mémoire relativement à un registre, nous serons limités à une distance de –32768 à 32767 (pour une distance sur un nombre signé) octets d'un certain registre si nous ne voulons utiliser qu'une instruction RTL. Cette information doit être connue du compilateur pour empêcher qu'il demande de générer du code inefficace. Il est possible qu'un processeur demande à avoir un registre « *code anchor* » qui pointe sur l'adresse de début d'un bloc de base tout au long de l'exécution. Ceci peut lui permettre de calculer une adresse plus simplement. Si tel est le cas, le générateur de code doit spécifier le registre utilisé à cette fin. Ensuite, nous devons spécifier les registres utilisables par le générateur de code RTL ainsi que les registres utilisables pour la génération de code machine. Les registres réservés pour la génération de code servent de registres temporaires pour les opérations plus complexes. Par exemple, si nous voulons additionner deux nombres situés en mémoire et que ce n'est pas possible directement sur le processeur, le générateur de code RTL va avoir besoin de mettre ces deux nombres dans des registres temporaires et faire l'addition après. Pour finir, nous devons spécifier la liste des registres pour les nombres à virgule flottante.

2.3.3) Format et paramètres

Comme il est mentionné plus haut, les instructions RTL sont assez proches des processeurs RISC. Il y a beaucoup de processeurs RISC qui utilisent trois registres pour les opérations arithmétiques donc le code RTL utilise généralement trois registres. Le premier registre sert de destination pour l'opération tandis que les deux autres registres servent de source pour le calcul. Donc, si nous avons l'instruction RTL `op(r1,r2,r3)`, ceci veut dire `r1` ← `r2 op r3`. Selon le type d'instructions, les paramètres peuvent être variés. Les choix possibles sont les suivants :

- Registre : Ceci est un numéro de registre pour les nombres entiers ou les nombres flottants.
- Adresse : Un nom qui symbolise l'adresse mémoire d'un objet alloué statiquement (par exemple le nom d'une variable globale).
- Constante : Un nombre quelconque. Le nombre doit entrer sur un nombre de bits fixé d'avance. Selon l'instruction RTL et la machine cible, le nombre de bits est fixe.
- Label : Un objet qui représente une étiquette dans le code du programme. Cet objet sera utilisé au moment de l'édition de liens.
- Symbole : Un symbole Scheme.
- «*keyword*» : Un « *keyword* » Scheme. Une extension à Scheme propre à Gambit et qui à toute fin pratique est comme un symbole.

Le type des paramètres est fixé d'avance pour toutes les opérations, par exemple l'instruction d'addition générale recevra toujours trois registres entiers en paramètres.

2.3.4) Instructions arithmétiques

Les instructions arithmétiques RTL peuvent être divisées en trois catégories. Les deux premières catégories utilisent des paramètres de type entier et l'autre catégorie n'utilise que des registres «flottants». La première catégorie comprend les instructions qui ont un paramètre de type constante. Ces instructions sont les suivantes :

Logique		Décalage		Multiplication/Division	
ANDwc	ET logique	SRAwc	Décalage arithmétique à droite	MULwc	Multiplication
IORwc	OU inclusif	SRLwc	Décalage logique à droite	DIVwc	Division
XORwc	OU exclusif	SLLwc	Décalage logique à gauche	REMwc	Reste après division

Autre	
ADDwc	Addition

Tableau 4 : Instructions RTL entre registres entiers et constantes

Toutes ces opérations sont entre une constante et un registre. Le « c » indique la constante et le « w » indique que le tout doit être fait sur des mots machines (*Word*). Donc, même si le nombre peut être encodé avec moins de bits que la longueur des mots de la machine, il doit être utilisé comme mot. Une instruction de

soustraction n'est pas présente car il suffit d'additionner l'inverse additif de la constante.

Le deuxième type d'opérations arithmétiques n'utilise que des registres. Les instructions RTL sont les suivantes :

Logique		Décalage		Multiplication/Division	
ANDwr	ET logique	SRAwr	Décalage arithmétique à droite	MULwr	Multiplication
IORwr	OU inclusif	SRLwr	Décalage logique à droite	DIVwr	Division
XORwr	OU exclusif	SLLwr	Décalage logique à gauche	REMwr	Reste de la division

Un seul paramètre		Autre	
NEGwr	Négation	ADDwr	Addition
NOTwr	Inversion des bits	SUBwr	Soustraction

Tableau 5 : Instructions RTL entre registres entiers

Toutes les instructions utilisent deux registres sources et un registre destination à l'exception de la négation et de l'inversion qui n'ont besoin que d'un registre source et d'un registre destination.

La dernière catégorie contient les instructions à nombres flottants. La syntaxe des instructions est normalement un registre pour la source et un pour la destination à l'exception des quatre instructions arithmétiques de base. Les instructions sont les suivantes :

Trigonométriques			
SINfr	Sinus	ACOfr	Arc cosinus
COSfr	Cosinus	ATAfr	Arc Tangente
TANfr	Tangente	LOGfr	Logarithme
ASIfr	Arc sinus	EXPfr	Exponentielle

Autres			
ADDfr	Addition	ABSfr	Valeur absolue
SUBfr	Soustraction	TRUfr	Partie entière
MULfr	Multiplication	ROUfr	Entier le plus près
DIVfr	Division	SQRfr	Racine carrée
NEGfr	Négation		

Tableau 6 : Instructions RTL avec registres flottants

Pour les processeurs qui ne peuvent exécuter ces instructions directement, il faut les simuler par une séquence moyennement longue d'instructions simples ou en appelant des routines de librairie (par exemple écrites en C).

2.3.5) Instructions de branchement

Les branchements arrivent très fréquemment dans un langage comme Scheme car c'est un langage souvent utilisé pour le traitement symbolique. Les branchements sont classés en deux catégories. La première et la plus petite comprend les branchements inconditionnels et la deuxième catégorie comprend tous les types de branchement conditionnel. Les branchements inconditionnels sont les suivants :

JUMPl	Branchement à une étiquette locale au module.
JUMPp	Branchement à une étiquette publique, c'est-à-dire qui est possiblement dans un autre module.
JUMPi	Branchement à l'adresse contenue dans un registre.

Tableau 7 : Instructions RTL de branchements inconditionnels

Ces trois instructions de branchement ont deux paramètres. Le premier est l'objet qui contient l'étiquette ou le numéro du registre et le deuxième paramètre est une distance qui est ajoutée au premier paramètre pour trouver la destination du branchement. Cette valeur peut être utile dans bien des langages mais elle est presque essentielle pour un compilateur Scheme pour retirer l'information de type sur les objets procédure. Ceci permet aux processeurs qui en ont la capacité de faire le saut et l'ajustement pour le type en une seule instruction machine.

Les instructions de comparaison suivantes peuvent s'appliquer dans les trois situations suivantes :

- Comparaison entre une constante et un registre (suffixe « wc »)
- Comparaison entre deux registres entiers (suffixe « wr »)
- Comparaison entre deux registres flottants (suffixe « fr »)

Les préfixes sont les suivants :

JEQ	Égalité	JNE	Pas égal
JLT	Plus petit	JGT	Plus grand
JLE	Plus petit ou égal	JGE	Plus grand ou égal

Tableau 8 : Sauts conditionnels

En utilisant les trois suffixes et les six préfixes, nous obtenons 18 instructions de sauts conditionnels.

2.3.6) Accès à la mémoire

Il existe trois classes d'instructions et directives qui ont un lien avec la mémoire. La première classe comporte des directives qui permettent de définir des zones de mémoire initialisée dans le code généré. La seconde classe d'instructions sert aux accès à la mémoire (lecture et écriture) et la dernière classe d'instructions permet d'obtenir l'adresse de certains objets.

La première classe comporte huit directives qui ajoutent des données brutes dans le code. Ces données doivent être préalablement alignées sur des adresses multiples de leur taille. Puisque ces données peuvent être inscrites directement dans le code généré, elles doivent avoir un sens pour le générateur de code, c'est-à-dire que les données doivent être placées de façon à ne pas entrer en conflit avec le code. Elles devraient être placées entre deux blocs de base ou dans le code mais après un saut inconditionnel pour éviter tout problème lors de l'exécution du code. Ces instructions sont les suivantes :

DATA1	Initialise un octet avec un entier 8 bits	DATAF	Initialise un nombre flottant sur 32 bits
DATA2	Initialise deux octets avec un entier 16 bits	DATAD	Initialise un nombre flottant sur 64 bits
DATA4	Initialise quatre octets avec un entier 32 bits	DATAK	Initialise un mot avec l'adresse d'un « *keyword* »
DATAW	Initialise un mot (relatif au processeur utilisé)	DATAL	Initialise un mot avec l'adresse d'une étiquette locale (inclus un déplacement)
DATAS	Initialise un mot avec l'adresse d'un symbole	DATAP	Initialise un mot avec l'adresse d'une étiquette publique (inclus un déplacement)

Tableau 9 : Instructions RTL pour la définition de données

La deuxième classe d'instructions permet de faire des accès en lecture et en écriture de la mémoire. Cette classe comprend les instructions suivantes :

Écriture		Lecture	
STO1I	Écriture indirecte d'un octet	LOA1I	Lecture indirecte d'un octet
STO2I	Écriture indirecte de deux octets	LOA2I	Lecture indirecte de deux octets
STO4I	Écriture indirecte de quatre octets	LOA4I	Lecture indirecte de quatre octets
STOWI	Écriture indirecte d'un mot	LOAWI	Lecture indirecte d'un mot
STOFI	Écriture indirecte d'un nombre flottant 32 bits	LOAFI	Lecture indirecte d'un nombre flottant 32 bits
STODI	Écriture indirecte d'un nombre flottant 64 bits	LOADI	Lecture indirecte d'un nombre flottant 64 bits
STOWG	Écriture d'une valeur dans une variable global	LOAWG	Lecture d'une valeur dans une variable global
STOWP	Écriture d'une valeur à une étiquette publique	LOAWP	Lecture d'une valeur à une étiquette publique

Tableau 10 : Lecture et écriture en mémoire

Les accès indirects se font à l'aide d'un registre. Ce registre contient l'adresse de la source (dans le cas d'une lecture) ou de la destination (dans le cas d'une écriture). Chacune de ces fonctions reçoit un objet représentant une région (par exemple un symbole) en paramètre. La région indique la zone de mémoire utilisée pour l'accès. Par exemple, nous pouvons avoir les zones suivantes : pile, tas, variables globales. Cette information est utile pour le réordonnanceur.

Certaines des instructions utilisent un nom d'objet (comme `stowg` qui reçoit en paramètre le nom d'une variable globale) pour les accès à la mémoire. Puisque les adresses des objets globaux et publics ne sont connues qu'à l'édition de liens (dans le loader), le nom de l'objet doit être sauvegardé dans la table d'édition des liens.

La troisième série permet de mettre l'adresse d'un objet dans un registre entier. Ces instructions sont les suivantes :

MOVWL	Met l'adresse d'une étiquette locale dans un registre entier.
MOVWP	Met l'adresse d'une étiquette publique dans un registre entier.
MOVWS	Met l'adresse d'un objet symbole dans un registre entier.
MOVWK	Met l'adresse d'un objet mot clef dans un registre entier.

Tableau 11 : Instructions RTL d'adresses

2.3.7) Autres

Les autres instructions et directives permettent de faire des opérations simples comme déplacer une valeur d'un registre à un autre, de convertir un registre entier en

un registre flottant et inversement, d'aligner le code, d'insérer des commentaires dans le code généré et de définir des étiquettes :

COMNT	Met un commentaire dans la liste textuelle du code.	ALIGN	Aligne le code qui va suivre cette instruction.
LOCAL	Définit une étiquette locale dans le code.	PUBLI	Définit une étiquette publique dans le code.
MOVWR	Copie le contenu d'un registre entier dans un autre registre entier.	MOVFR	Copie le contenu d'un registre flottant à un autre registre flottant
MOVWC	Assigne une constante à un registre entier.		
CONWR	Copie la valeur d'un registre flottant dans un registre entier.	CONFR	Copie la valeur d'un registre entier dans un registre flottant.

Tableau 12 : Instructions RTL générales

Les processeurs demandent normalement que les accès à la mémoire soient alignés sur un multiple de la taille en octets de la donnée. Par exemple, les accès à des données de deux octets doivent être alignés sur les multiples de 16 bits, c'est-à-dire que le bit le moins significatif de l'adresse de la valeur devrait être à zéro. Ceci n'est pas imposé par tous les processeurs mais pour des raisons de rapidité, il est préférable d'utiliser un tel alignement.

Les étiquettes locales sont utilisées dans les sauts conditionnels ainsi que pour la définition des objets locaux. La même étiquette peut donc être réutilisée dans un autre module sans risque de conflit.

Dans l'instruction de conversion entre un registre flottant et un registre entier, il est évident qu'il peut y avoir de la perte d'information. La partie fractionnaire est perdue dans la conversion.

2.4) RTL et les processeurs

Certaines instructions RTL posent des problèmes d'implantation sur certaines architectures. Cette section discute des difficultés importantes qui sont survenues lors de l'écriture des générateurs de code.

2.4.1) Saut indirect

L'instruction «`JUMPI reg, deplacement`» est utilisée pour brancher à une procédure ou à un point de retour d'un appel de procédure. Le déplacement permet d'annuler l'information de type placée sur le pointeur de procédure ou adresse de

retour. Dans un premier temps, nous avons utilisé la représentation simple suivante pour les procédures et adresses de retour : un pointeur vers le code machine exécutable, aligné sur un multiple de quatre et taggé avec le type 1 (sous-type).

Le processeur Pentium d'Intel possède une instruction pour faire un saut inconditionnel à l'adresse contenue dans un registre mais cette instruction ne permet pas d'ajouter un déplacement constant à la valeur du registre dans le saut. Évidemment dans le cas où le déplacement est égal à zéro, l'instruction Pentium «`JUMP %R`» est suffisante. Prenons l'exemple de l'instruction RTL : `jumpi %R1,2`. Cette instruction demande de calculer deux de plus que le contenu du registre `%R1` et ensuite, de continuer l'exécution du programme à cette adresse. Vu que le nombre de registres est très faible sur le Pentium, nous ne pouvons nous permettre de copier la valeur du registre `%R1` dans un autre registre, faire l'addition et finir par le branchement car nous ne sommes pas sûrs d'avoir toujours un registre de libre. Nous ne pouvons non plus changer la valeur du registre `%R1` par l'addition de ce registre et de la valeur car le registre `%R1` peut être lu ultérieurement et son contenu ne sera plus celui qui était prévu.

La première façon de contourner le problème est très simple à implanter mais elle est très peu performante. Le bout de code suivant est inséré dans le code machine lors de la traduction de l'instruction `jumpi` :

« JUMPI Reg, Valeur »	
`PUSH Reg`	Met le contenu du registre `Reg` sur la pile
`ADD [SP],Valeur`	Additionne la valeur désirée directement sur la pile
`RETURN`	Continue l'exécution à la valeur au sommet de la pile (c'est-à-dire à `Reg+Valeur`)

Tableau 13 : JUMPI sur Pentium(1)

Cette méthode n'est pas efficace à cause des quatre accès à la mémoire (deux accès pour l'addition, un pour le `PUSH` et un pour le `RETURN`). L'unité de prédiction de saut est incapable de prédire la bonne adresse de retour pour ce genre de séquence. Le nombre de cycles machine requis par l'instruction `RETURN` est donc très élevé. De plus, ces trois instructions sont très dépendantes entre elles ce qui les rend difficiles à ordonnancer.

La deuxième méthode n'est qu'expérimentale car elle ne répond pas à une des normes du RTL. Nous avons effectué des tests pour savoir quelle accélération est possible si nous n'utilisons pas la pile pour faire le saut. Nous modifions par contre la valeur du registre passé en paramètre. La nouvelle séquence de code ressemble à ceci :

« JUMPI Reg, Valeur »	
`ADD Reg,Valeur`	Additionne au registre `Reg` la constante `Valeur` pour qu'il contienne aussi le déplacement.
`JUMP Reg`	Branche à l'adresse demandée

Tableau 14 : JUMPI sur Pentium(2)

Comme nous le voyons, le registre `Reg` est modifié mais, dans nos tests, nous n'avons utilisé que des séquences de code qui n'utilisent pas la valeur du registre `Reg` après le saut. Cette façon de procéder nous permet d'avoir les accélérations suivantes (sur les *benchmark* `fib` et `tak`) :

- Entre 1 et 7 pour cent sur un Pentium
- Entre 6 et 20 pour cent sur un Pentium Pro

Les séquences de codes utilisées pour les tests sont relativement petites pour avoir le plus de sauts possible. Dans ces tests, le déplacement est –1.

La dernière alternative est très dépendante du compilateur mais elle donne de très bons résultats. Puisque nous utilisons les générateurs de code pour Gambit-RTL, le déplacement demandé dans l'instruction `jumpi` est toujours petit (un nombre entre zéro et moins trois) et nous indique le type de l'objet. Nous pouvons ajouter un certain nombre d'octets bidons lors de l'écriture des étiquettes pour ainsi enlever le besoin de soustraire la valeur du type lors des sauts. Par exemple, si les objets de type code (procédures et adresses de retour) ont un type égal à 1, la valeur contenue dans le registre `Reg` lors des sauts sera toujours un octet plus loin que l'étiquette. Donc, en insérant un octet bidon au début du code, nous n'avons plus besoin de faire la soustraction de un avant le saut. Ceci nous permet donc de coder l'instruction de saut avec une seule instruction machine. Grâce à ceci, nous obtenons les accélérations suivantes[1] :

- Entre 2 et 14 pour-cent sur un Pentium
- Entre 9 et 24 pour-cent sur un Pentium Pro

Nous arrivons donc avec les trois séquences de code suivantes :

[1] Ces résultats sont par rapport à la méthode originale.

Instruction RTL	Alternative 1	Alternative 2	Alternative 3
`MOVWL R1,L21`	`MOVL $L21+1,%R1`	`MOVL $L21+1,%R1`	`MOVL $L21+1,%R1`

`JUMPI R1,-1`	`PUSHL %R1`
	`DECL (%SP)`
	`RET`
		`DECL %R1`
		`JUMP %R1`
			`JUMP %R1`

`LOCAL L21`	`.ALIGN 4`
	`.LONG 4369`
	`L21:`
	...
		`.ALIGN 4`
		`.LONG 4369`
		`L21:`
		...
			`.ALIGN 4`
			`.LONG 4369`
			`L21:`
			`NOP`
			...

Tableau 15 : JUMPI, les trois alternatives

La directive « `.ALIGN 4` » permet de s'assurer que l'étiquette `L21` est bien alignée sur un multiple de quatre et que les deux bits les moins significatifs sont à zéro. Ensuite, la directive « `.LONG 4369` » n'est que de l'information pour le GC. Cette information lui permet d'identifier les sous types des objets qui sont utilisés.

Pour le DEC Alpha, les branchements indirects (relatif au type) peuvent se faire en une seule instruction. Puisque les instructions sont toujours alignées sur des multiples de quatre octets, les deux bits les moins significatifs du registre PC sont ignorés par le processeur. L'information de type est donc enlevée automatiquement par le processeur à coût zéro.

Après avoir exécuté des tests de performance sur les branchements, nous avons changé la représentation des procédures pour supporter les fermetures. De plus, la représentation des adresses de retour a également été modifiée. Maintenant, les branchements se font toujours avec un déplacement nul. Les procédures et les blocs de retour sont maintenant représentés de cette façon :

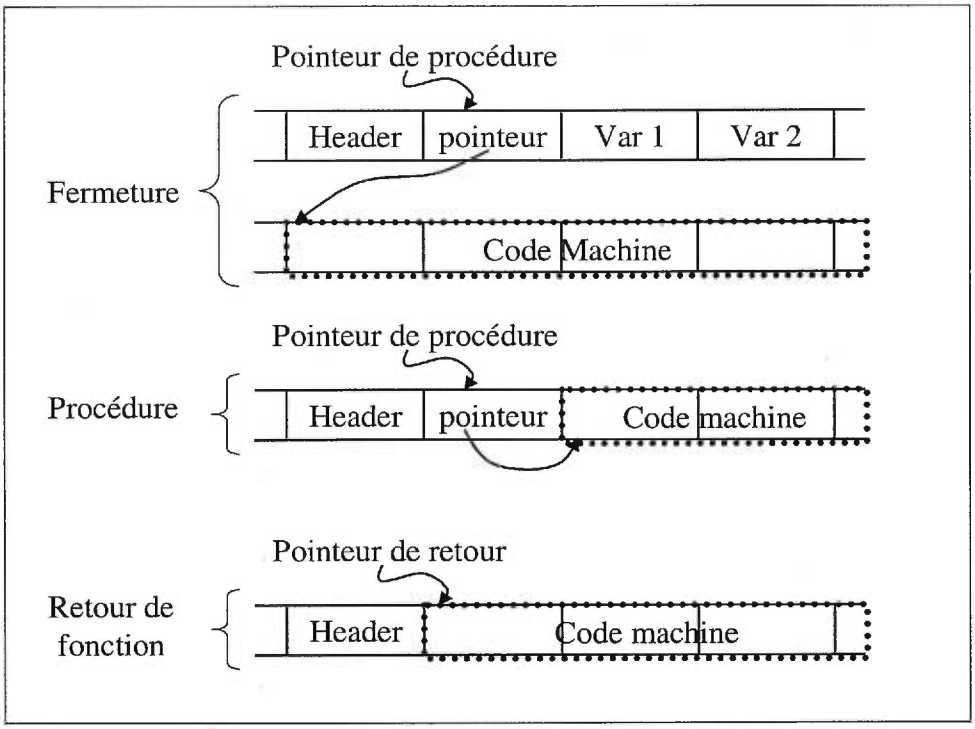

Figure 6 : Fermetures et Retours

Pour faire un appel à une procédure, il faut charger l'adresse du code en utilisant le pointeur de procédure. Ce dernier pointe vers le second octet du deuxième mot de la fermeture. Étant donné que les procédures sont alignées sur des multiples de quatre octets et que le *Header* est de la dimension d'un mot machine (4 ou 8 octets), le pointeur de procédure a comme type 01 (les deux bits les moins significatifs). Nous utilisons les deux instructions RTL suivantes pour faire un saut à une procédure :

`LOAWI %RegDest, %RegSrc, -1, 'tas`	Chargement du pointeur vers le code machine. `%RegSrc` pointe vers une fermeture dans le tas.
`JUMPI %RegDest, 0`	Branchement au code machine de la procédure.

Tableau 16 : Instructions RTL pour un appel de procédure

Puisque le GC n'a pas besoin de traverser les blocs de retour de procédures, nous pouvons utiliser les types 00 et 10 (fixnum et spécial) pour les pointeurs vers ces blocs. De cette façon, nous pouvons faire pointer le registre directement sur le début du bloc. De plus, nous pouvons débuter les blocs à n'importe quelle adresse paire lorsque le processeur le permet, sinon, à un multiple permis (normalement 4).

Grâce à ceci, nous pouvons effectuer les retours de fonctions avec un simple branchement à l'adresse spécifiée dans un registre.

2.4.2) Accès à la mémoire sur 64bits

L'architecture des DEC Alpha 21064 a un bus de donnée de 64 bits. De plus, puisque les instructions du DEC Alpha 21064 sont codées sur 32 bits et qu'il n'y a que 16 bits de disponibles pour les constantes, nous devons utiliser six instructions, dans le cas général, pour construire une valeur de 64 bits (constante, adresse). Nous commençons par obtenir les bits 0 à 15 ainsi que les bits 16 à 31. Ensuite, nous devons obtenir les bits 32 à 47 ainsi que les bits 48 à 63. Après ceci, il faut faire une décalage de 32 bits des deux dernières valeurs obtenues car elles sont situées dans les bits 0 à 31 d'un registre. Pour finir, il faut additionner les deux valeurs. La séquence de code qui est requise est la suivante :

`LDA t0,{00_15}(zero)`	Écriture des bits 0 à 15 dans le registre t0
`LDAH t0,{16_31}(t0)`	Écriture des bits 16 à 31 dans le registre t0 dans la partie haute des 32 bits les moins significatifs
`LDA t1,{32_47}(zero)`	Écriture des bits 32 à 47 dans le registre t1 dans les 16 premiers bits du registre t1
`LDAH t1,{48_63}(t1)`	Écriture des bits 48 à 63 dans le registre t1 dans la partie haute des 32 bits les moins significatifs.
`SLL t1,32,t1`	Décalage de 32 bits du registre t1
`ADDQ t0,t1,t0`	Addition des deux valeurs

Tableau 17 : Création d'une adresse 64 bits sur DEC Alpha

Par exemple, si la constante que nous voulons créer est 0123456789ABCDEF, nous avons le code suivant :

Instructions	Registre	
	T0	T1
`LDA t0,{00_15}(zero`[2]`)`	`0000 0000 0000 CDEF`	***Inconnue***
`LDAH t0,{16_31}(t0)`	`0000 0000 89AB CDEF`	***Inconnue***
`LDA t1,{32_47}(zero)`	...	`0000 0000 0000 4567`
`LDAH t1,{48_63}(t1)`	...	`0000 0000 0123 4567`
`SLL t1,32,t1`	...	`0123 4567 0000 0000`
`ADDQ t0,t1,t0`	`0123 4567 89AB CDEF`	...

Tableau 18 : Exemple de construction d'adresse

Nous constatons que ceci est très coûteux en temps et en espace étant donné que le nombre de constructions est élevé. Chaque accès direct à la mémoire (sans

[2] Registre qui a toujours la valeur *zéro*.

passer par un registre) demande une construction d'adresse. Dans le cas de constantes entières, nous pouvons souvent éviter quelques instructions puisque les bits du haut sont souvent tous égaux (soit 1 ou 0).

Un cas fréquent de construction d'adresses, c'est la construction de l'adresse de retour d'un appel de procédure. Dans ce cas, il faut charger un registre avec l'adresse du point de retour de l'appel (une étiquette dans le code).

Le compilateur GCC[3] sur le DEC Alpha utilise une autre approche pour minimiser le nombre d'instructions lors des constructions d'adresses ainsi que pour les constructions de constantes de 64 bits. Lors de la compilation d'un module, une table de constantes est créée et placée dans le code. Avec cette table, la construction d'une constante se fait en seulement trois instructions. Nous avons besoin de deux instructions pour obtenir l'adresse de la constante dans la table et d'une instruction pour faire la lecture de la constante de la table. Les instructions sont les suivantes :

`LDAH tempREG, XXXX(GP)` `LDA tempREG, YYYY(tempREG)`	Le tableau se trouve à `XXXXYYYY` octets du registre GP. L'équivalent dans le langage C de ces instructions est `tempREG=&Tableau[Index].`
`LDQ REG2,0(tempREG)`	Lecture du tableau

Tableau 19 : Création d'une constante avec GCC

Ce tableau n'est pas pointé directement par le registre GP. La distance entre ce pointeur et le tableau doit entrer dans 32 bits. Ces trois instructions sont une solution plus compacte que les six instructions originales mais ils demandent d'avoir un accès à la mémoire supplémentaire. Normalement, lorsque nous construisons une adresse, c'est que nous voulons faire une lecture ou une écriture à cette adresse. Nous avons donc besoin de deux accès à la mémoire au lieu d'une ce qui ralentit l'exécution du code. Nous devons aussi ajuster le registre GP lors de l'entrée dans une fonction (si cette fonction est appelée d'un autre module) ainsi qu'au retour d'un appel d'une fonction appartenant à un autre module.

Dans le contexte de Gambit-RTL, les branchements à des procédures sont normalement effectués à l'aide d'un registre (le registre contient l'adresse de destination). Nous avons donc créé un registre virtuel (implanté à l'aide d'un registre général) qui pointe toujours vers un des blocs de base de la procédure courante (plus précisément un point d'entrée ou de retour). Ce registre (CA pour « *code anchor* ») est additionné à la distance entre l'adresse de l'étiquette désirée et le bloc vers lequel pointe CA. Lorsque l'étiquette désirée est à une distance

[3] Version 2.8.0

inférieure à 32768, seule une instruction d'addition est requise pour le DEC Alpha (ADDwc).

Pour mettre à jour le registre CA lors d'un appel à une procédure et d'un retour de procédure, nous utilisons CA pour charger l'adresse de destination du branchement. Ce principe nous évite d'avoir à faire une mise à jour du registre CA aux points d'entrée de procédures (début et retour de procédures). Ainsi le compilateur sait que lors de l'exécution du code dans un bloc de base qui est un point d'entrée ou de retour, CA pointe vers le bloc en question.

Un problème survient cependant pour les autres blocs de base. Appelons « parent de B » tout bloc de base qui contient un branchement vers le bloc de base B. Si B n'a qu'un parent, il est logique de lui attribuer une valeur pour le registre CA qui est identique à son parent (aucune modification du registre CA ne sera nécessaire lors du branchement à B). Mais si B a plus d'un parent et que ceux-ci n'ont pas la même valeur dans le registre CA, il faudra ajuster le registre pour certains des branchements à B. Dans l'exemple qui suit, nous avons une procédure (F) qui boucle sur une certaine condition et qui appelle F' à chaque tour de boucle. Les blocs A (début de la procédure) et B (retour d'un appel) sont des « ancres », c'est-à-dire des blocs dont l'adresse est assignable au registre CA :

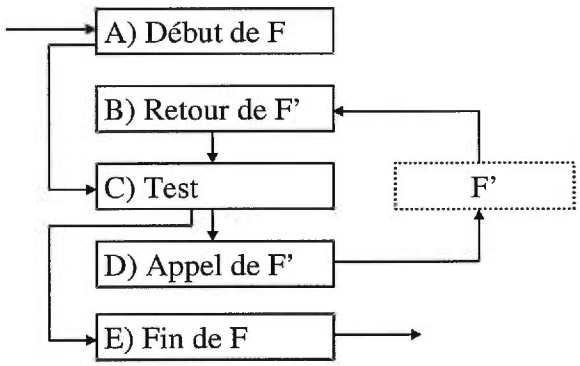

Figure 7 : Attribution des ancres

À l'exécution du bloc C, le registre CA peut pointer vers A ou vers B. Nous devons donc ajuster le registre CA pour qu'il pointe toujours à la même position (à l'intérieur d'un bloc). Nous avons les deux choix suivants :

1. Ajuster le registre CA à la fin de A pour qu'il pointe sur B
2. Ajuster le registre CA à la fin de B pour qu'il pointe sur A.

Puisque le bloc B risque d'être exécuté plus souvent que le bloc A, nous préférons faire l'ajustement 1. L'ajustement consiste à additionner la différence d'adresse entre deux étiquettes au registre CA.

L'algorithme utilisé pour attribuer une ancre aux blocs de base est une boucle qui fait la propagation de l'ancre vers le bas (selon le graphe de flot d'exécution). Cette boucle est exécutée jusqu'à ce que tous les blocs de base aient une ancre.

```
Pour tout bloc de base B
    Si B est un point d'entrée  ou de retour
        B.Ancre ← B.
    Sinon
        B.Ancre ← ∅
    Fin du si
Fin du Pour tout
Tant qu'il existe un bloc de base B t.q. (B.Ancre = ∅)
    Tmp ← ∅
    Pour tout bloc de base P parent de B  (selon le graphe de flot d'exécution)
        Si la distance entre B et P < Tmp
            Tmp ← P
        Fin du si
    Fin du Pour tout
    B.Ancre ← Tmp
Fin du Tant que
```

Tableau 20 : Distribution des ancres

Lors de la génération du code machine, un ajustement sera fait avant les branchements si l'ancre du bloc destinataire n'est pas le même que le bloc courant. Il serait possible d'éviter un ajustement dans les cas où les blocs de destination n'ont pas besoin du registre CA mais cela n'a pas été implanté.

Pour finir, nous avons réservé un second registre (GV pour «*Global variables*») qui pointe vers les variables globales. Les variables globales de tous les modules sont relatives à ce pointeur.

2.4.3) Nombres à virgule flottante

Les calculs sur les nombres flottants ont posé problème lors de l'écriture de certains générateurs de code. Parfois, seulement un sous-ensemble des instructions point flottant est implanté sur le processeur ou bien il n'existe pas de registres point flottant.

Le SPARC n'implante qu'un très petit sous-ensemble des instructions point flottant. Il supporte les opérations de base comme l'addition, la soustraction, la multiplication, la division, la racine carrée ainsi que des instructions de conversion et de comparaison. Les instructions trigonométriques ne sont pas calculées en matériel.

Nous devons donc utiliser une librairie de fonctions mathématiques pour exécuter les opérations trigonométriques. Nous avons choisi d'utiliser la librairie mathématique du langage C. Nous avons ajouté une section dans le loader(pour le SPARC) pour les opérations à point flottant manquantes. Lors de l'initialisation du *loader* (fonction `Target_dependent_setup`), deux variables globales sont ajoutées. Ces variables sont accessibles aux modules Scheme. La première variable est un pointeur vers du code C, la deuxième est un pointeur vers un nombre flottant. Le module Scheme met le paramètre de l'instruction trigonométrique dans la bonne cellule mémoire et appelle le module C pour qu'il effectue l'opération. Au retour, il reprend le résultat à la même adresse que l'opérande. Pour éviter tout conflit, le module C doit sauvegarder tous les registres utilisés par le module Scheme. Ceci pourrait être fait par le module Scheme mais cela augmenterait grandement la taille du code Scheme inutilement.

Le processeur Pentium d'Intel n'utilise pas les registres flottants de la même façon que les autres processeurs. Ils sont sous forme d'une pile de petite taille (8). Toutes les instructions utilisent le sommet de la pile comme source. Par exemple, pour effectuer une addition, nous devons mettre un des deux nombres au sommet de la pile et ensuite nous pouvons effectuer l'addition. De plus, quelques fonctions comme $\tan^{-1}(y/x)$ exigent que ces deux paramètres soient au sommet de la pile. Ceci pose un petit problème parce que les registres flottants de Gambit-RTL sont adressables directement. Nous devons donc émuler une banque de registres à l'aide de cette pile en copiant un des paramètres de l'addition au sommet de la pile avant d'effectuer l'opération. Pour l'arc tangente partielle, nous devons mettre ces deux paramètres au sommet de la pile. À cause de ceci, nous devons toujours avoir au moins deux registres flottants consécutifs libres. Voici un exemple de l'arc tangente, qui calcule $\tan^{-1}$`(freg1)`→`freg2` :

`FLOAD FREG1`	Empile le paramètre au sommet de la pile point flottant
`FLOAD 1`	Empile 1 au sommet de la pile (le registre `FREG1` est donc en deuxième position).
`FPATAN`	Calcule `ATAN (FREG1/1)`
`STORE FREG2+1`	Met le résultat dans le registre `FREG2`. Puisque cette instruction effectue un `FPOP` après le *store*, nous devons additionner 1 au numéro de registre pour que le résultat soit à la bonne position.

Tableau 21 : Exemple avec FPATAN.

3) Fonctionnement d'un ordonnanceur

L'ordonnanceur d'instructions prend une série d'instructions et il essaie de trouver une façon de placer ces instructions pour que leur exécution soit plus rapide. Malheureusement, l'algorithme optimal est NP-complet [HG83]. Nous devons donc utiliser des heuristiques. Un des algorithmes le plus connu est le « list scheduling » [AEAK92]. L'avantage de cet algorithme est qu'il est assez simple à implanter et qu'il donne de bons résultats. Avant de faire le « list scheduling », nous avons besoin de créer le graphe de dépendance (liens entre les instructions) ainsi que d'exécuter les algorithmes ALAP (*as late as possible*) et ASAP (*as soon as possible*) pour savoir quelles instructions favoriser.

3.1) Graphe de dépendance

Un graphe de dépendance exprime les contraintes d'ordre d'exécution des instructions. Un graphe de dépendance est représenté par la liste des instructions (LI) ainsi qu'une liste de dépendances (LD). La liste LD est composée de couples de numéros d'instructions. Chacun des couples exprime une contrainte sur l'ordre d'exécution des instructions. Par exemple, le couple (A, B) indique que A doit être exécutée avant l'instruction B.

Une instruction dépend d'une autre si un des quatre cas suivants est présent :
- *Anti-dépendance* : Il y a lecture d'une variable (registre ou emplacement mémoire) par une instruction et cette variable est modifiée par une instruction qui suit.
- *Dépendance de flux de données* : Une variable est modifiée par une instruction et elle est lue par une instruction qui suit.
- *Dépendance de sortie* : Une variable est modifiée par une instruction et cette même variable est aussi modifiée par une instruction qui suit. Il est à noter que l'ordonnanceur n'est pas en mesure de déterminer si nous pouvons éliminer la première instruction qui peut sembler (et même être) inutile dans certains cas.
- *Dépendance de contrôle* : Les instructions de branchements ont une dépendance avec les autres instructions.

Pour construire le graphe de dépendances, nous devons relier les instructions qui ont une dépendance entre elles. Pour ce faire, nous utilisons un algorithme comme celui-ci :

```
Pour A=1 à #LI
      Pour B=A+1 à #LI
            S'il existe une dépendance entre LI[A] et LI[B]
                  Ajouter un arc de A à B.
            Fin du Si
      Fin du Pour
Fin du Pour

#LI    : Nombre d'instructions dans la liste d'instructions
LI[X]  : Xe instruction de la liste des instructions
```

Tableau 22 : Création du graphe de dépendance

Nous devons vérifier les dépendances pour tous les *paramètres* des instructions LI[A] et LI[B]. Le nombre de paramètres étant borné, cet algorithme s'exécute en $O(n^2)$ où **n** est le nombre d'instructions dans LI. De plus, nous pouvons utiliser la propriété de transitivité pour réduire le nombre d'arc dans le graphe de dépendances.

L'exemple qui suit nous montre un bout de code en assembleur ainsi que les dépendances associées :

Pseudo code	Assembleur	Dépendances
`A = T1(0)*P1+T2(0)`	1. `%F0 ← MEM(0+%R16)`	(1,5), (1,9)
`B = T1(1)*P2+T2(1)`	2. `%F8 ← MEM(0+%R0)`	(2,6), (2,11)
`T3(0)=A`	3. `%F1 ← MEM(8+%R16)`	(3,7), (3,9)
`T3(1)=B`	4. `%F9 ← MEM(8+%R0)`	(4,8), (4,11)
`IF P3 < T1(2) THEN`	5. `%F0 ← F5 * F0`	(5,6)
`  GOTO Place`	6. `%F8 ← F0 + F8`	(6,12)
`END IF`	7. `%F1 ← F5 * F1`	(7,8)
	8. `%F9 ← F1 + F9`	(8,13)
`P1,P2,P3` sont des paramètres pour le fragment de code.	9. `%R16 ← &MEM(10+%R16)`	(9,10)
	10. `%R11 ← %R16 < %R10`	(10,14)
`T1,T2,T3` sont des tableaux.	11. `%R0 ← &MEM(10+%R0)`	(11,12),(11,13)
	12. `MEM(0+%R0) ← %F8`	
	13. `MEM(8+%R0) ← %F9`	
	14. `BNE %R11,Place`	(1,14)…(13,14)

Tableau 23 : Exemple de graphe de dépendance

L'instruction 14 crée une dépendance de contrôle avec toutes les autres instructions car ce branchement doit absolument être exécuté après toutes les autres instructions. Nous pouvons transformer la dépendance de contrôle en Anti-dépendance en ajoutant une lecture du registre PC à toutes les instructions et en ajoutant une écriture du registre PC seulement pour les branchements.

3.2) List Scheduling

L'algorithme du « list scheduling » est normalement utilisé comme cœur d'un ordonnanceur d'instructions. Il y a plusieurs raisons qui motivent ce choix. Premièrement, l'algorithme est peu complexe et très facile à comprendre. Deuxièmement, il s'adapte facilement à diverses applications d'ordonnancement. Finalement, il a une structure modulaire ce qui facilite la maintenance.

Voici l'algorithme général :

```
ListScheduling ( LI, LD, fct_Select)
      Schedule ← ∅
      Tant que LI ≠ ∅
            Tête ← { I ∈ LI | ¬∃ (P,I) ∈ LD }
            I ← fct_Select(Tête)
            Schedule ← Schedule ++ I
            LI ← LI - {I}
            LD ← { (x,y) ∈ LD | x ≠ I }
      Fin
      Retourner Schedule
```

¬∃ : N'existe pas
++ : Ajouter à la fin de la liste.

Tableau 24 : L'algorithme du List-Scheduling

En sortie, *Schedule* contient l'ordre des instructions à exécuter. L'algorithme débute par initialiser cette liste à *vide*. Ensuite, il faut boucler jusqu'à ce que toutes les instructions aient été ordonnancées. Nous assignons à la variable *Tête* toutes les instructions libres (celles qui n'ont plus de dépendance dans LD). Ensuite, nous devons choisir une instruction parmi cette liste d'instructions. Après coups, nous l'ajoutons à la liste *Schedule*. Pour finir, nous enlevons l'instruction qui a été choisie de la liste LI ainsi que des dépendances LD.

La difficulté de l'algorithme se situe dans le choix de la fonction fct_Select. Selon l'implantation de cette fonction, nous pouvons avoir des résultats tout à fait différents. Il existe plusieurs algorithmes qui peuvent êtres utilisés [AEAK92]. Les quatre qui suivent sont parmi les plus intéressant :

1. *Longueur maximum* : Nous choisissons l'instruction qui a le chemin le plus long.
2. *Nombre de cycles* : Nous choisissons l'instruction qui s'exécute en le moins (ou le plus) de cycles.
3. *Dépendance maximal* : Nous choisissons l'instruction qui enlève le plus de dépendance.
4. *Âge maximal* : Nous choisissons l'instruction qui est la plus âgée.

3.3) Calcul de la longueur du chemin(ALAP/ASAP)

Pour nous aider à choisir entre plusieurs instructions qui sont disponibles en même temps, nous pouvons utiliser la latence (nombre de cycles que prend une instruction pour s'exécuter) des instructions pour nous guider. La latence est le temps que prend cette instruction ainsi que ses successeurs à s'exécuter. Pour calculer la latence maximale (nombre de cycles que prend une instruction et ses successeurs pour s'exécuter), nous pouvons utiliser un algorithme qui se nomme ALAP (*As Late As Possible*) [DM94].

```
ALAP( LI, LD, λ )
    LI[#LI].Latence = λ
    Tant qu'il reste des latences à trouver
        Sélection d'une instruction (I) dont toutes les latences de ses successeurs ont été calculées.
        LI[#I].Latence = Latence Minimum des successeurs de I moins durée de LI[#I].
    Fin du tant que.
```

Tableau 25 : Calcul de la longueur du chemin

Si nous utilisons l'algorithme ALAP sur l'exemple qui suit, nous obtenons les résultats suivants :

Assembleur	Étapes	Instructions Disponibles	Latence maximale des instructions
1. %F0 ← MEM(0+%R0) 2. %F1 ← MEM(8+%R0) 3. %F2 ← MEM(10+%R0) 4. %F0 ← %F1 * %F0 5. %F0 ← %F2 + %F0 6. %R0 ← &MEM(20+%R0) 7. %R11 ← %R0 < %R10 8. BNE %R11,Place	1	8	λ=5*
	2	7 5	5-1=4 5-1=4
	3	6	4-1=3
	4	4	3-1=2
	5	1 2 3	Min(2,3)-1=1 Min(2,3)-1=1 Min(4,3)-1=2

Tableau 26 : Exemple de calcul de longueur du chemin

* : Le nombre 5 est tout à fait arbitraire. Toutes les opérations sont considérées comme ayant une latence de 1.

Nous obtenons l'exécution suivante :

Cycle	Unité 1	Unité 2
1	%F0 ← MEM(0+%R0)	%F1 ← MEM(8+%R0)
2	%F2 ← MEM(10+%R0)	%F0 ← %F1 * %F0
3	%R0 ← &MEM(20+%R0)	
4	%F0 ← %F2 * %F0	%R11 ← %R0 < %R10
5	BNE %R11,Place	

Tableau 27 : Exemple d'ordonnancement

Avec ceci, nous pouvons savoir que le chemin critique est de cinq cycles lorsque l'on dispose d'au moins deux unités de traitement. Comme nous le constatons, cet algorithme peut servir d'ordonnanceur d'instructions lorsque le nombre d'unités de traitement est considéré comme infini.

3.4) Principaux types d'ordonnanceurs

Dans le cadre de la compilation, il est possible d'utiliser l'ordonnanceur de plusieurs façons. Selon la complexité du compilateur et la performance désirée, les choix suivants se présentent :

- *Ordonnancement avant l'allocation de registres* : Puisque les registres à ce stade sont des registres virtuels, il y a peu de dépendances entre les calculs. Nous avons ici le problème de la durée de vie des variables. Leur durée de vie peut devenir trop élevée ce qui va provoquer un manque de registres lors de l'allocation des registres. Nous pouvons donc ajouter des « spills » (code de sauvegarde des registres en mémoire) où il n'y en aurait pas eu.
- *Ordonnancement après l'allocation de registres* : Lorsque nous faisons l'ordonnancement après l'allocation de registres, nous pouvons avoir un manque d'efficacité car le graphe de dépendances peut devenir assez dense lorsque le nombre de registres n'est pas assez élevé. Plus le graphe est dense, moins l'ordonnanceur a de marge de manœuvre pour déplacer les instructions. Nous pouvons, aussi, avoir un module d'allocation de registres qui est sensible à l'ordonnanceur [CNLP2]. Ceci permet d'obtenir un graphe de dépendance moins dense.
- *Ordonnancement avant et après l'allocation de registres* : Nous avons l'avantage de l'ordonnancement avant l'allocation de registres ainsi que l'avantage de pouvoir ordonnancer les instructions qui peuvent avoir été rajoutées pour le « *spilling* ».
- *Ordonnancement entrelacé avec l'allocation de registres* : Il y a plusieurs façons d'entrelacer l'ordonnanceur avec l'allocation de registres [SSP93][CN95][PSR95][BEH91]. Cette façon de procéder a l'avantage de tenir compte du « *spilling* » ainsi que du niveau de parallélisme dans le

code. Par contre, la complexité du module « Ordonnancement + Allocation » est grandement accentuée.

3.5) Facteurs influants

Plusieurs facteurs peuvent influencer les résultats d'un ordonnanceur.

- *Le nombre de registres* : Le nombre de registres est le facteur le plus important dans l'ordonnanceur. Plus le nombre de registres est faible, plus le graphe de dépendances peut être dense. De plus, si nous disposons d'un nombre insuffisant de registres, du « spilling » surviendra ce qui va rendre le graphe plus dense et ajoutera des instructions fort dépendantes.
- *Types de mémoire* : Pour diminuer la densité du graphe de dépendances, il est bien de distinguer la mémoire générale de la pile, des variables globales, et autres. Ceci ne peut être fait que si chaque type de mémoire est dans une zone bien distincte. Grâce à ceci, nous pourrons entrelacer plus facilement les instructions. Dans l'exemple qui suit, nous pouvons constater que les registres `%R1` et `%R2` disposent maintenant de deux cycles avant d'être utilisés.

Code original, sans distinction de types de mémoire	Avec distinction de types de mémoire

`%R1 ← Mem(0)`
`SommetPile ← %R1`
`%R2 ← Mem(1)`
`%R1 ← %R1 + %R2`
	`%R1 ← Mem(0)`
	`%R2 ← Mem(1)`
	`SommetPile ← %R1`
	`%R1 ← %R1 + %R2`

Tableau 28 : Distinction entre les types de mémoires

- *Conflit impossible, possible, assuré* : Pour les mêmes raisons que pour la distinction entre les types de mémoires, nous devons pouvoir distinguer les conflits lors des accès à la mémoire. S'il est *impossible* d'avoir un conflit entre deux accès à la mémoire, nous n'ajoutons pas d'arc au graphe. Lorsque le conflit est *possible*, nous restons conservateurs et nous considérons l'accès comme *assuré*.

Impossible	Possible	Assuré
`%R1 ← MEM(0+%R2)` `MEM(20+%R2) ← %R3`	`%R1 ← MEM(0+%R2)` `MEM(20+%R4) ← %R3`	`%R1 ← MEM(0+%R2)` `MEM(0+%R2) ← %R3`

Tableau 29 : Conflit impossible, possible et assuré

- *Réallocation de registres* : Lorsque le nombre de registres le permet, nous pouvons réallouer les registres de façon à diminuer la densité du graphe de

dépendances. Ceci a comme inconvénient d'exiger d'avoir un ordonnanceur très dépendant de l'architecture de la machine cible.

Code original	Code avec réallocation
a ← b + c	a ← b + c
e ← a + f	e ← a + f
a ← e + f	**z** ← e + f
d ← **a** + e	d ← **z** + e

Tableau 30 : Réallocation des registres

4) Les ordonnanceurs existants

Il existe plusieurs ordonnanceurs. Ils ont chacun leurs avantages et inconvénients. La plupart ne sont qu'expérimentaux ou peu utilisés. Par contre, il en existe quelques-uns qui sont connus et largement utilisés. Nous allons débuter par expliquer la différence entre deux types d'ordonnanceurs possibles et nous allons continuer par une description de plusieurs ordonnanceurs.

4.1) Global Scheduling

Lorsque nous ordonnançons le code, nous pouvons utiliser principalement deux philosophies différentes [CN95].

La première façon de procéder est de n'ordonnancer que les instructions d'un bloc de code à la fois. Les blocs de code sont définis comme suit :

- *Un seul point d'entrée* : Il ne doit pas être possible de faire un saut, à partir de ce bloc ou de tout autre bloc, sur une instruction autre que la première.
- *Un seul point de sortie* : Il ne peut y avoir qu'un saut (conditionnel ou non) et ce saut doit être la dernière instruction du bloc.

Il est assez simple de fonctionner de cette façon. Peu de mémoire est requise car la mémoire est libérée après l'ordonnancement de chaque bloc. Normalement, les blocs sont assez petits.

La seconde façon de procéder est de prendre tous les blocs de base et de déplacer les instructions dans un même bloc ou deux blocs différents. Ceci est plus complexe car le choix des instructions à déplacer entre les blocs n'est pas simple. Nous devons trouver les instructions indépendantes pouvant être déplacées dans le ou les blocs précédants/suivants sans poser problèmes. Par exemple, si nous avons les trois blocs de codes suivants :

```
1. BLOC1:
2. J = 12
3. K = MEM(1)
4. L = J*K
5. GOTO BLOC3
```

```
6. BLOC2:
7. A = 15
8. B = MEM(2)
9. C = A*B
10. GOTO BLOC3
```

```
11. BLOC3:
12. X = 99
13. MEM(3) = X
14. Y = X+1
```

Tableau 31 : Ordonnancement global

Dans ce code, les instructions 4 et 9 ne peuvent être exécutées sans délai à cause de la dépendance avec l'instruction précédente. Le fait d'ordonnancer globalement nous permettrait de déplacer l'instruction 12 entre les instructions 3 et 4 ainsi qu'entre les instructions 8 et 9. Ceci aurait l'avantage de permettre d'augmenter le niveau de parallélisme dans les deux premiers blocs avec un léger accroissement de la taille du code. De cette façon, les instructions 4 et 9 pourraient être exécutées avec pas ou peu de délais.

4.2) L'ordonnanceur de GCC

Le compilateur C de GNU, GCC, est utilisé sur beaucoup de plates-formes différentes. De plus, pour plusieurs processeurs, il existe plusieurs versions du compilateur selon le système d'exploitation qui est utilisé. Ce compilateur a été écrit de façon à ce qu'il soit facilement portable. Pour ce faire, il travaille avec des instructions virtuelles, c'est-à-dire qu'une fois que le source est lu en mémoire, il est converti en instructions RTL (très différentes de celles que l'on utilise pour Gambit-RTL). Par la suite, toutes les optimisations s'opèrent sur les instructions RTL. Pour finir, un module de définition doit être écrit pour chaque version pour la génération du code. La génération de code se fait avec du filtrage (« *pattern matching* »).

L'ordonnanceur qui est inclus dans GCC est très générique et il est exécuté après l'analyse de flux de données et il fonctionne comme suit :

- Assignation des priorités (numéro de l'instruction) aux instructions (première : 0, dernière : X)
- List Scheduling (en partant de la fin de la liste)
  - Extrait les instructions prêtes dans une liste
  - Choisit une des instructions de la liste des instructions prêtes selon l'ordre suivant :
    1. L'instruction qui a un coût de conflit peu élevé.
    2. L'instruction qui a le chemin le plus long jusqu'à la fin du bloc.
    3. L'instruction qui diminue le plus le nombre de registres utilisés.

4. L'instruction qui a le plus de conflits avec celles de la liste des instructions prêtes.

- Mise à jour des prédécesseurs des instructions choisies (ajustement des dépendances)

L'ordonnanceur est appelé avant et après l'allocation des registres.

Il est à noter que l'ordonnanceur est désactivé sur les processeurs i386 et ces successeurs lorsque l'on demande l'optimisation de niveau 2 et plus (`-O2`). Ceci est fait parce que les performances obtenues, après avoir ordonnancé le code sur i386, ne sont pas bonnes. Nous pouvons observer que l'exécution du code est affectée négativement par l'ordonnancement. Ceci est dû aux raisons suivantes :

- Manque évident de registres
- Dépendances entre les instructions et les registres
- Les processeurs Pentium II et III (les plus modernes présentement) exécutent les instructions en les réordonnant.

4.2.1) Ordonnanceur balancé

Normalement, les ordonnanceurs qui utilisent l'algorithme du « *list scheduling* » considèrent que la durée d'exécution d'une instruction prend toujours le même temps, quelles que soient les circonstances. Lors de la création du graphe de dépendances, cette durée est assignée aux arêtes et utilisée pour le restant des traitements. Ce principe fonctionne bien pour des processeurs qui ne sont pas joints à une mémoire cache ou bien ceux qui arrêtent l'exécution de toutes les instructions lorsqu'un des paramètres n'est pas disponible pour une des instructions qui veut être exécutée.

Cette façon de procéder ne tient pas compte des nouvelles réalités des processeurs. Les processeurs modernes peuvent continuer à exécuter des instructions même s'il y en a une qui est arrêtée à cause d'un paramètre non disponible, de plus, le temps d'accès à la mémoire n'est plus constant dû au fait que les données peuvent être dans une des mémoires caches disponibles dans le processeur (ou près).

[DRK92] propose une solution à ce petit problème. Cette solution peut être intégrée dans l'algorithme du « *list scheduling* » assez facilement. Ce qui est proposé est de calculer les poids pour l'exécution des chargements dynamiquement. Ceci peut être fait à l'aide d'une mesure qui est appelée « *niveau de parallélisme des chargements* ». Donc, pour ce faire, le poids de chaque chargement est calculé de façon indépendante selon le nombre d'instructions (autres que chargement) qui

peuvent débuter leur exécution durant le chargement ainsi que le nombre de chargements qui peuvent être utilisés pour masquer la latence des instructions. La formule pour connaître le niveau de parallélisme à une instruction donnée est la suivante :

$$1 + \frac{\text{Nombre d'instructions indépendantes qui suivent}}{\text{Nombre de chargements en série qui suivent}}$$

Équation 1 : Niveau de parallélisme des chargements

Le nombre « 1 » est le coût associé au « lancement » de l'instruction de chargement. Le nombre d'instructions indépendantes est la somme des instructions qui n'ont pas de lien de dépendances avec l'instruction de chargement courante. Le nombre de chargements en série est le nombre de chargements qui dépendent du chargement courant. Pour nous aider à mieux comprendre, regardons les exemples qui suivent (les résultats sont relatifs au premier chargement (`%L1`)) :

Chargement en série	Chargement en parallèle

`%L1 ← MEM(0)`
`%L2 ← MEM(%L1)`
`%L3 ← MEM(%L2)`
Nombre de chargements en série : 3
	`%L1 ← MEM(0)`
	`%L2 ← MEM(4)`
	`%L3 ← MEM(8)`
	Nombre de chargements en série : 1

`%L1 ← MEM(0)`
`%L2 ← MEM(%L1+0)`
`%L3 ← MEM(%L1+4)`
Nombre de chargements en série : 3
	`%L1 ← MEM(0)`
	`%L2 ← MEM(%L4)`
	`%L3 ← MEM(%L2)`
	Nombre de chargements en série : 1

Tableau 32 : Exemples de chargement en série et en parallèle

Nous pouvons voir que le chargement fait partie de la série. À l'aide d'un exemple, nous allons voir que cette façon de calculer a un impact sur le poids qui est associé au chargement :

Code	Chargement	Formule	Poids
`1. %L1 ← MEM(0)`	1	1+5/2	3.5
`2. %L2 ← MEM(4)`	2	1+6/1	7

`3. %L3 ← MEM(%L1)`
`4. %L4 ← 5`
`5. %L5 ← %L4 + 10`
`6. %L6 ← 15`
`7. %L7 ← %L6 + 20`
	3	1+5/1	6

Tableau 33 : Niveau de parallélisme

Nous pouvons voir que le niveau de parallélisme est moins élevé pour le premier chargement. Ceci est normal puisque cette instruction doit être ordonnancée tôt pour ne pas poser problème au chargement numéro trois.

4.2.2) Ordonnancement par trace

L'ordonnancement par trace utilise un graphe de contrôle du flux d'exécution et optimise le parcours que le compilateur estime que le programme empruntera [CN95]. Donc l'ordonnanceur prend toutes les instructions du chemin choisi et les considère comme faisant partie d'un bloc unique. De cette façon, il est possible d'augmenter le parallélisme de certaines parties du code et de diminuer certaines parties trop parallèles pour le processeur en cours d'utilisation. L'inconvénient majeur de cet algorithme est que le programmeur doit aider l'ordonnanceur dans ses choix ou bien il doit exécuter le logiciel dans un environnement contrôlé pour capturer un profil d'une exécution type.

4.2.3) Ordonnancement par percolation

Cet algorithme utilise le graphe de contrôle de flux pour déplacer des instructions vers des blocs précédents [CN95]. Ceci est fait pour une instruction à la fois et se termine lorsque l'on ne peut plus rien effectuer comme changement ou bien lorsque l'optimisation à l'aide de l'ordonnancement n'apportera plus rien. L'exemple suivant nous montre ce que nous pouvons avoir comme résultats :

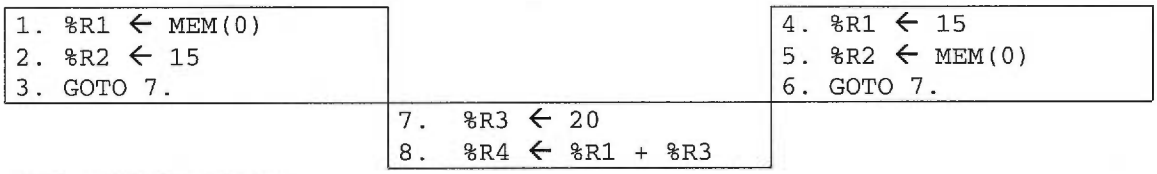

Tableau 34 : Percolation

Dans cet exemple, nous pouvons faire monter l'instruction 7 dans les deux blocs précédents sans problème. Une fois l'instruction 7 rendue dans les deux blocs supérieurs, elle pourrait être déplacée dans les blocs précédents aux besoins.

4.2.4) Autres

Il existe quelques autres algorithmes pour ordonnancer les instructions. Il est à noter que ces algorithmes ne sont, pour la plupart, que des modifications mineures des algorithmes présentés.

5) Implantation

Nous avons pu voir au cours des chapitres précédents qu'il existe beaucoup d'ordonnanceurs différents et qu'ils ont tous des qualités et des défauts. De plus, il y a sur le marché plusieurs processeurs différents. Nous savons aussi que les processeurs sont en constante évolution et qu'il devrait y avoir encore énormément de changements dans l'architecture interne des futurs processeurs. Donc, pour ces raisons, nous nous devons d'utiliser un ordonnanceur qui n'est ni trop spécialisé ni trop complexe pour éviter d'être obligé de réécrire entièrement. Il doit aussi pouvoir accepter facilement les changements futurs des processeurs tout en minimisant l'impact sur les générateurs de code RTL pour éviter d'avoir à les éditer à chaque fois.

5.1) Terminologie utilisée

Plusieurs termes sont utilisés dans ce chapitre et nous devons les expliquer.

Les ***ressources physiques*** sont des composantes qui permettent au processeur de fonctionner. Par exemple, les unités de traitements et les bus internes.

Les ***ressources logiques*** expriment des concepts présents dans un processeur. Par exemple, le tas (en tant qu'une partie de la mémoire générale), la pile et l'espace pour les variables globales sont des ressources logiques.

Les ***ressources***, utilisées par l'ordonnanceur, sont les ressources physiques et logiques utilisées par un générateur de code machine. Chaque ressource doit avoir un nom unique. Donc, si deux ressources sont identiques dans un processeur (par exemple deux unités de traitements pareilles), nous devons leur donner des noms différents (comme ALU1 et ALU2) pour que l'ordonnanceur puisse les distinguer. Nous pouvons aussi définir des ressources propres à certains processeurs. Par exemple, nous avons créé les ressources «simple1», «simple2» et «complexe», pour le Pentium, afin de distinguer les instructions pouvant être décodées en une «*micro-ops*» des autres.

Les ***ressources de dépendance*** sont celles qui doivent respecter les règles de dépendance. Les registres, la mémoire, le tas et la pile font partie de ce type. Ces ressources lui permettent de créer le graphe de dépendance.

Les ***ressources de réservation*** sont celles qui sont utilisées par l'ordonnanceur pour aider à choisir une instruction lorsque plus d'une est disponible. Chacune des instructions à ordonnancer peut réserver des ressources pour une certaine durée.

Donc les instructions dont les ressources demandées ne sont pas réservées sont préférées.

Les ***ressources lues*** (*«read-res»*) sont celles qui sont utilisées en lecture par l'instruction courante. Ceci peut être un accès en lecture en mémoire ainsi que la lecture d'un registre. Ces ressources sont du type 'ressources de dépendance' puisqu'elles sont utilisées lors de la création du graphe de dépendance.

Les ***ressources modifiées*** (*«write-res»*) sont celles qui sont modifiées par l'instruction courante. La sauvegarde d'une valeur dans un registre, sur la pile ou en mémoire sont des exemples. Ces ressources sont aussi du type 'ressources de dépendance' pour les mêmes raisons.

Les ***ressources utilisées*** (*«use-res»*) sont celles qui ne sont ni lues ni modifiées par l'instruction courante. Par exemple, il y a les unités de traitement et les bus de données. Ces ressources sont de type 'ressources de réservation'.

L'étape de la ***réservation*** consiste à réserver une ressource pour un nombre de cycles donné. Ceci permet d'ordonnancer des instructions qui sont indépendantes des ressources déjà réservées lorsque ceci est possible.

La ***durée*** de réservation d'une ressource s'exprime en cycles. Nous pouvons donner le nombre de cycle moyen ou maximal lorsque celui-ci n'est pas constant (par exemple, un accès à la mémoire). Ce choix dépend de plusieurs facteurs comme le processeur utilisé, le type de programme qui est compilé et le programmeur qui utilise l'ordonnanceur.

Les ***instructions suivantes*** sont celles qui doivent absolument être exécutées après l'instruction courante.

Un ***thunk*** est une procédure sans paramètre. Normalement une fermeture.

5.2) Informations utilisées par le réordonnanceur

Deux types d'information sont utilisés par le réordonnanceur. Le premier type lui permet d'avoir de l'information sur le processeur cible et le second type lui donne de l'information sur les instructions qui sont ajoutées.

5.2.1) Initialisation

Nous devons donner quelques informations sur le type d'ordonnancement désiré. Ceci comprend le type de bloc (fixe, dynamique, par instructions) et la

dimension des blocs. De plus, nous devons donner de l'information sur le support du branchement à retardement.

5.2.2) Ajout d'instructions

Lorsqu'on ajoute une instruction à la liste des instructions déjà générées, nous devons donner la liste des ressources lues, modifiées et utilisées. De plus, pour chacune d'elles, nous devons donner la durée que prendra la lecture, la modification et l'utilisation de ces ressources.

5.3) Fonctions de base

Dans le but de garder l'ordonnanceur le plus indépendant possible, nous n'avons créé que quatre fonctions publiques. Les deux premières fonctions sont utilisées pour l'initialisation de l'ordonnanceur et les deux autres fonctions sont utilisées par les générateurs de code. Les fonctions sont les suivantes :

1. ***OPT-Basic-Block-Init***
2. ***OPT-Set-Options*** `Type-Block Block-Size isDelayBranch?`
3. ***OPT-Add-Ins*** `thunk read-lst write-lst use-lst link-with-next?`
4. ***OPT-Basic-Block***

La première fonction est appelée par le compilateur avant de débuter la génération de code. Cette fonction sert à initialiser les structures internes de l'ordonnanceur.

La deuxième fonction peut être appelée par les générateurs de code pour donner de l'information sur le type d'ordonnanceur requis pour le processeur courant. Ses trois paramètres permettent de changer les options suivantes :

- *Type-Block* : Ce paramètre peut être un des symboles suivants :
  - *Block* : Indique à l'ordonnanceur de prendre les instructions par bloc. Tel que mentionné plus haut, les processeurs RISC ont tendance à exécuter un nombre d'instructions par cycle et d'exécuter les instructions restantes du bloc dans le (ou les) prochain(s) cycle(s) selon la disponibilité des unités de traitement.
  - *NoBlock* : Nous indiquons à l'ordonnanceur de ne pas essayer de remplir le bloc d'instructions avant de passer au prochain bloc. De cette façon, nous pouvons avoir des blocs de tailles différentes. Cette technique s'applique à certains processeurs CISC comme le Pentium.

  - ♦ *Basic* : Cette technique peut être utilisée sur les processeurs qui n'exécutent qu'une instruction à la fois.
- *Block-Size* : Ce paramètre donne la taille des blocs qui doivent être utilisés par les types `NoBlock` et `Block`.
- *IsDelayBranch?* : Lorsque ceci est applicable, la technique du branchement à retardement est utilisé. Ceci peut s'appliquer aux trois méthodes d'ordonnancement vues plus haut.

La troisième fonction sert à ajouter des instructions dans le bloc de base courant. Les paramètres sont les suivants :

- *Thunk* : Après avoir ordonnancé le code, tous les **Thunk** sont exécutés dans l'ordre demandé par l'ordonnanceur. Voici un exemple de **Thunk** présent dans le générateur de code pour Pentium :

```
(lambda ()  (asm-8 #x90)
            (if rtl-listing? (out "nop")))
```

  Ce Thunk ajoute l'instruction NOP (octet 90h) au code déjà existant ainsi qu'un "nop" au fichier liste (si ce fichier est demandé).
- *read-res* : Nous devons donner la liste de toutes les ressources (registres, unités, ...) qui sont lues par le `Thunk` ainsi que le temps demandé pour compléter l'opération. Ceci se fait à l'aide d'une liste de paire : `((unité durée) ...)`. Par exemple, l'instruction d'écriture d'un mot en mémoire sur DEC Alpha (`MEM(r2)`←`r1`), définit la liste suivante :

```
((r1 1) (r2 1))
```

  Ceci indique que les registres `r1` (contient la valeur à écrire en mémoire) et `r2` (contient l'adresse mémoire) sont lus et qu'il ne peuvent être réutilisés avant le cycle qui suit.
- *write-res* : Nous devons aussi donner la liste des ressources qui sont accédées en écriture par le **Thunk**. Cette liste sert à empêcher qu'une unité soit accédée alors que le résultat n'est pas encore rendu. Par exemple, lors d'un chargement d'une donnée, le résultat peut n'être disponible que plusieurs cycles après le lancement de l'opération. La liste suivante (utilisée dans l'instruction `PUSH reg` du générateur de code pour Pentium) :

```
((i386_ESP 2) (stack 3))
```

  nous indique que la pile sera utilisée pour trois cycles et que le registre SP ne sera pas disponible avant deux cycles.
- *use-res* : Cette liste nous donne les ressources utilisées par le **Thunk**. Contrairement aux deux listes précédentes, celle-ci n'est pas utilisée pour la construction du graphe de dépendances (deux instructions qui utilisent une même unité de traitement peuvent être inversées sans changer la sémantique du programme). Cette liste n'est utilisée que pour connaître la disponibilité des ressources lors de l'ordonnancement.

Si une ressource n'est pas unique (par exemple, deux unités de calcul identiques), nous devons pouvoir en informer l'ordonnanceur. Dans ce cas, nous utilisons l'opérateur OR. Cet opérateur permet d'exprimer le choix **d'une** ressource parmi plusieurs. Sa sémantique est la suivante : `(or (ressource1 durée1) (ressource2 durée2) ...)`. L'ordonnanceur regarde chacune des ressources de la liste à l'intérieur du OR et il utilise la première qui n'est pas réservée. Cette liste :

```
((or (ALU1 1) (ALU2 1)) (BUS1 1))
```

indique que l'ordonnanceur a le choix entre la ressource ALU1 et ALU2 pour l'exécution. De plus, cette instruction réserve toujours la ressource BUS1 pour un cycle.

Nous avons mentionné dans le chapitre 2 que le Pentium II effectue l'étape du décodage d'une façon particulière. Nous devons utiliser les unités de décodage dans l'ordre suivant : « complexe, simple, simple ». Donc, pour ce faire, nous ajoutons les listes qui suivent aux instructions :

simples : `(or (COMPLEXE 1) (SIMPLE1 1)) (SIMPLE2 1))`

Ceci permet d'ordonnancer cette instruction sur n'importe quelle unité de décodage disponible tout en commençant par l'unité *complexe*.

complexes : `((COMPLEXE 1))`

Cette liste exige un décodage sur l'unité des instructions complexes.

- *`link-with-next?`* : Ceci permet d'attacher plusieurs instructions successives ensemble. Elles vont être considérées comme une seule instruction pour toute l'étape d'ordonnancement. Ceci peut être utilisé pour protéger des séquences d'instructions qui doivent absolument être exécutées successivement. Nous pouvons aussi utiliser cette fonctionnalité pour forcer les commentaires à suivre les instructions pendant l'ordonnancement.

La dernière fonction sert à démarrer l'ordonnanceur. Cette fonction peut être appelée à chaque moment jugé opportun. Puisque l'ordonnanceur fonctionne par bloc de base, il est nécessaire d'appeler cette fonction lors des circonstances suivantes :

- *Avant une étiquette* : L'ajout d'une étiquette indique une possibilité d'entrée dans un bloc de code. Nous devons donc empêcher que des instructions soient déplacées entre ces deux blocs. Dans l'exemple suivant, nous pouvons constater que le fait de déplacer l'instruction 3 après l'étiquette changera la sémantique du programme :

Code original	Code erroné
1. MOVE R1,10	1. MOVE R1,10
2. BLE L1	2. BLE L1
3. MOVE R1,15	3. LOAD R2,MEM(0)
4. LOAD R2,MEM(0)	4. L1:
5. L1:	5. MOVE R1,15
6. ADDL R1,R2	6. ADDL R1,R2

Tableau 35 : Changement de sémantique

- *Après une instruction de branchement* : Il faut tout d'abord mentionner que la sémantique du code ordonnancé doit être préservée lors d'ajout d'instructions de branchement. Pour ce faire, lors de l'ajout d'une instruction de branchement, nous devons spécifier que le registre PC (*program counter*) peut être modifié par cette instruction (changement autre que l'incrémentation normale après chaque instruction). De plus, puisque toutes les autres instructions impliquent une certaine lecture du registre PC, nous ajoutons automatiquement le registre PC dans la liste des ressources lues. Après coup, nous devons appeler l'ordonnanceur pour qu'il génère les instructions du bloc.
- *Avant et après une donnée dans le code* : Le fait d'ajouter une donnée dans le code nous impose d'appeler l'ordonnanceur avant et après l'ajout. Ceci est dû au fait que la donnée n'est dépendante d'aucune instruction donc elle peut être relocalisée facilement. Nous pouvons évidemment forcer une dépendance en inscrivant le registre PC dans la liste des ressources modifiées mais ceci n'est pas une solution élégante.

5.4) Collecte d'instructions

La récolte des instructions se fait à l'aide de la fonction `OPT-Add-Ins`. Nous débutons par la construction d'un vecteur qui contient tous les champs nécessaires pour l'ordonnancement. Ce vecteur est ensuite ajouté à une liste. Le vecteur contient les champs suivants :

- *Minimum* : Numéro du cycle minimal pour exécuter l'instruction, c'est-à-dire que cette instruction ne doit pas être ordonnancée avant ce cycle sinon la sémantique du code ne sera pas respectée.
- *Maximum* : Numéro du cycle maximal pour exécuter l'instruction. Nous devons donc ordonnancer cette instruction entre *Minimum* et *Maximum*.
- *Thunk* : Nous avons ici l'instruction ou les instructions à exécuter.
- *Liste-ressources* : Pour le reste de l'exécution, nous n'avons pas besoin de faire une distinction entre les ressources lues, modifiées et utilisées. À

cause de ceci, nous concaténons ces trois listes. Cette liste sera utilisée lors de l'ordonnancement pour mettre à jour la table de réservation des ressources.

- *Précédents* : La création du graphe de dépendances implique la création d'arêtes. Nous avons ici la liste des arêtes qui pointent sur des instructions précédentes. Par exemple, si nous avons les instructions suivantes :

Instructions	Précédents	Suivants
1. LOAD R1,MEM(0) 2. MOVE R2,R1 3. ADDL R1,10 4. STORE MEM(0),R1	1. vide 2. 1 3. 2 4. 1 et 3	1. 2 et 4 2. 3 3. 4 4. vide

Tableau 36 : Dépendances. Suivants et précédents

Nous pouvons voir que l'instruction deux est dépendante de la première à cause du registre R1. Cette liste sera utilisée par l'algorithme du ALAP et du ASAP.

- *Suivants* : Nous avons le complément du champ *Précédents*. Ces deux champs sont en fait une liste de paires contenant les deux champs suivants : Numéro de l'instruction précédente ou suivante ainsi que la durée associée à l'arc.
- *N* : Numéro de l'instruction courante.

Notes pour les champs *Minimum* et *Maximum* :

- Ils sont initialisés à la valeur 0 lors de la création du vecteur. Ces valeurs ne peuvent être obtenues que lorsque toutes les instructions sont dans la liste. Ils sont donc calculés dans la deuxième partie de l'algorithme d'ordonnancement.
- Ces deux valeurs vont contenir le numéro du cycle optimal. Elles vont être calculées selon un environnement sans limite de ressources. Par exemple, si nous avons cinq instructions consécutives d'addition et qu'elles sont indépendantes entre elles, alors elles vont avoir le même *Minimum* et *Maximum*.

5.4.1) Création des paramètres suivants et précédents.

Lors de l'initialisation de l'ordonnanceur, deux listes sont créées. Elles contiennent de l'information sur les dernières instructions ajoutées. La première liste contient les renseignements sur les ressources qui sont lues et la seconde liste est pour les ressources qui sont modifiées. Les trois types d'information sauvegardée

sont les ressources, le numéro de l'instruction ainsi que la durée requise pour libérer la ressource. Donc, si la seconde liste (écriture) contient les données suivantes :

```
((R1 3 1) (MEM(0) 4 3)
```

ceci nous indique que le registre `R1` est utilisé en écriture par la troisième instruction et qu'il va être réservé pour un cycle. Ensuite, la ressource `MEM(0)` est utilisée pour une écriture par l'instruction numéro quatre et cette ressource ne pourra pas être utilisée avant quatre cycles.

Ces deux listes nous permettent de trouver facilement les dépendances entre les instructions. Nous n'avons qu'à faire l'intersection entre une des deux listes et un des deux paramètres (*read-lst* et *write-lst*) passés à la fonction `OPT-Add-Ins`. Selon le type d'intersection, nous pouvons obtenir un type différent de dépendance. Par la suite, nous pouvons créer les arcs tout simplement en liant le numéro des instructions trouvées par l'intersection et le numéro de l'instruction courante. Nous pouvons obtenir les trois types de dépendances suivants :

Paramètre		Liste des ressources		Type
Read-lst	Write-lst	Lue	Modifié	
#A			#B	Anti-Dépendance
	#A	#B		Dépendance de flux de données
	#A		#B	Dépendance de sortie

Tableau 37 : Intersection entre les quatre listes

La création des arcs se fait en ajoutant le bon numéro d'instruction à la bonne liste. Nous devons modifier la variable *Précédents* de l'instruction courante pour y ajouter le numéro de l'instruction qui crée la dépendance ainsi qu'ajouter le numéro de l'instruction courante au paramètre *Suivant* de l'autre instruction.

Après avoir créé les arcs, nous n'avons qu'à ajuster les deux listes pour qu'elles tiennent maintenant compte de l'ajout de l'instruction. Nous effectuons, dans l'ordre, les opérations suivantes :

- Ajout à la liste des ressources lues de toutes les ressources lues par l'instruction courante.
- Retrait de la liste des ressources lues de toutes les ressources modifiées par l'instruction courante.
- Mise à jour et ajout des ressources modifiées par l'instruction courante à la liste des ressources modifiées.

5.5) Algorithme utilisé pour l'ordonnancement

Lorsque toutes les instructions d'un bloc de base ont été données à l'ordonnanceur, il doit effectuer quelques étapes préparatoires avant de vraiment

démarrer l'ordonnancement. Les étapes préparatoires consistent à calculer les cycles minimaux et maximaux pour l'exécution (*ALAP* et *ASAP*) et un tri sur les cycles maximaux. Après ces trois étapes, nous pouvons effectuer l'ordonnancement.

5.5.1) Cycles minimaux et maximaux

Pour le calcul du cycle minimal, un simple balayage des instructions est effectué. En débutant par la première instruction, la somme du poids de chacun des arcs appartenant au champs *Suivants* et du numéro du cycle courant est affectée aux instructions destinataires des arcs.

```
POUR i ∈ {1 .. |Instructions|}
      POUR (j durée) ∈ Suivant[i]
            Minimum[j]=MAX(Minimum[j], Minimum[i]+durée)
```

Tableau 38 : Cycles mininaux

De cette façon, nous obtenons aussi le nombre de cycles requis pour l'exécution du bloc dans une situation où nous n'avons pas de limite de ressources. Après avoir obtenu le cycle minimum, nous calculons le cycle maximum de la même façon sauf en partant de la fin de la liste. Nous utilisons le cycle le plus élevé qui a été obtenu dans la première partie comme nombre de départ.

Une fois les cycles minimal et maximal obtenus, nous pouvons facilement reconnaître les instructions qui font partie du chemin critique. Ces instructions ont la même valeur dans les deux champs. Donc, elles ne démarrent pas avant un certain cycle et ne peuvent démarrer après ce même cycle. L'exemple suivant montre les résultats obtenus après ces deux calculs (le coût des chargements est de 3 cycles et toutes les autres opérations sont considérées comme unitaires) :

Code	Suivants	Précédents	Cycle	
			Minimum	Maximum
1. LOAD R1,MEM(0)	(3 3) (5 1[4]) (7 1)	AUCUN	0	0
2. MOVE R2,10	(4 1) (7 1)	AUCUN	0	3
3. ADDL R3,R1	(4 1) (7 1)	(1 3)	3	3
4. ADDL R3,R2	(7 1)	(2 1) (3 1)	4	4
5. STORE MEM(1),R4	(6 1) (7 1)	(1 1)[5]	1	3
6. MOVE R4,20	(7 1)	(5 1)	2	4
7. JUMP ETI2	AUCUN	(1 1) ... (6 1)[6]	5	5

Tableau 39 : ASAP et ALAP

Nous constatons que les instructions 1, 3, 4 et 7 font partie du chemin critique. Si plus d'un chemin critique est présent, l'ordonnanceur sera moins performant puisque le choix d'instructions à chaque cycle sera moins élevé.

Après avoir obtenu ces résultats, nous effectuons un tri des instructions selon le cycle maximum. Le tri nous permet de savoir quelles instructions nous devons ordonnancer en priorité. Si nous avons le choix entre deux instructions dont les cycles maximums sont différents, nous devons choisir celle dont ce numéro est le plus petit pour éviter de faire un trou dans le pipeline.

5.5.2) Ordonnancement

Après avoir obtenu les résultats des calculs précédents, nous pouvons maintenant débuter l'ordonnancement. Selon le type d'ordonnancement sélectionné par la fonction `OPT-Set-Options`, nous effectuons le choix des instructions.

Le premier principe, le plus général, prend l'instruction la plus prioritaire et l'ordonnance si toutes les ressources demandées sont disponibles. Chacune des instructions est testée de cette façon jusqu'à ce que l'on en trouve une qui réponde aux exigences. Si nous ne trouvons aucune instruction, nous passons tout simplement au cycle suivant sans rien ajouter à la liste des instructions ordonnancées. Cette méthode de fonctionner est principalement pour les processeurs qui n'exécutent qu'une instruction à la fois mais elle peut aussi servir comme canevas de base, dû à sa grande simplicité, pour l'écriture d'un autre type d'ordonnancement.

---

[4] La ressource mémoire est libérée après un cycle à cause du principe de pipeline.

[5] Nous supposons que MEM(0) et MEM(1) sont de la catégorie « conflit possible ».

[6] Toutes les instructions sont dépendantes à cause du registre PC qui est modifié par l'instruction numéro 6.

Le second principe est principalement pour les processeurs de type CISC. Il est à noter que ce principe a déjà été étudié dans une section précédente. En résumé, il est demandé par le générateur de code d'ordonnancer un maximum de N instructions par cycles. Même si nous ne pouvons ordonnancer les N instructions dans le cycle courant, au cycle suivant, nous recommençons la recherche pour trouver N instructions indépendantes et dont les ressources sont toutes libres. Ce principe s'applique très bien au processeur de type Pentium II car un maximum de trois instructions peuvent être ordonnancées par cycle et ceci dépend du type d'instructions (soit simple ou complexe) que nous avons.

Le dernier principe a été créé pour les processeurs de type RISC. Cette méthode a aussi été étudiée dans une section du chapitre 2. Ici, nous devons ordonnancer N instructions par bloc. Lorsque nous avons moins de N instructions de disponibles, nous pouvons finir de remplir par des NOP. Nous pouvons aussi finir de remplir le bloc avec d'autres instructions mais celles-ci ne vont être exécutées que dans le cycle suivant.

Les trois méthodes que nous venons de mentionner sont codées de la même façon à l'exception de quelques petits détails. Les grandes étapes sont les suivantes :

- Demande une instruction à la fonction de *sélection*.
  - Les ressources demandées par l'instruction doivent être libres.
  - Le cycle minimal pour débuter l'instruction doit être respecté.
  - Les dépendances doivent être respectées.
- Mise à jour de la liste des ressources utilisées.
- Mise à jour des dépendances (enlèvement d'arcs dans le graphe de dépendances).
- Ajout de l'instruction choisie dans la liste final.
- Boucle (selon la méthode)
  1. Incrémente le cycle à tout coup
  2. Incrémente le cycle si N est atteint ou que nous n'avons pas trouvé d'instructions (on repart avec N instructions).
  3. Incrémente le cycle si N est atteint ou que nous n'avons pas trouvé d'instructions (on ne change pas de bloc, on doit le compléter).

Après que ceci ait eu lieu, tous les *thunk* sont exécutés dans l'ordre demandé par une des trois méthodes. Pour finir, toutes les structures sont réinitialisées.

6) Résultats

Pour les tests de performances, nous avons utilisé les six générateurs de code. Lorsque ceci était possible, nous avons exécuté les tests sur plus d'une version du processeur. Nous allons débuter par une petite description des processeurs utilisés qui sera suivie de la liste des programmes utilisés pour les tests de performance et nous allons finir par les résultats des tests.

Nous avons utilisé les huit machines suivantes pour effectuer les tests de performance :

Générateur de code	Processeur	Mémoire	Cache L1	Horloge	Colonne
Alpha	Alpha 21064	155M	16K	133Mhz	Alpha0
Alpha	Alpha 21164	*N.A.*	16K	500Mhz	Alpha1
I386	Pentium Pro	512M	16K	150Mhz	PII
I386	Pentium	30M	16K	133Mhz	PI
M68k	68LC040	20M	8K	33Mhz	M68k
Mips	Mips	512M	64K	150Mhz	Mips
Ppc	IBM RS6000	*N.A.*	32K	80Mhz	PPC
Sparc	Ultra Sparc	128M	32K	167Mhz	Usparc
Sparc	Sparc	352M	32K	50Mhz	Sparc

Tableau 40 : Processeurs utilisés

Les programmes que nous avons utilisés sont les suivants :

Nom	Description
`(assq 1000000)`	Ce programme effectue un grand nombre de recherches dans une liste d'entiers. Il y a donc beaucoup d'accès à la mémoire ainsi que des comparaisons.
`(cpstak 22 14 4)`	Nous avons ici le programme tak qui a subi une transformation CPS. Il crée donc abondamment de fermetures et la mémoire (le tas) est constamment utilisée.
`(nrev 110 55)`	Ce programme crée une liste de 110 nombres et inverse cette liste 55 fois.
`(deter 8 8)`	Ce programme calcule le déterminant d'une matrice 8 par 8 de façon récursive. À chaque niveau de récursivité, une nouvelle matrice est créée.
`(fib 37)`	Ce programme effectue un nombre considérable de récursions. Les blocs de base sont très petits et il y a

	beaucoup de branchements.
`(mazefun 17 17)`	Ce programme crée un labyrinthe (17 par 17) avec une liste de listes. Chaque élément des listes est soit #f ou #t pour représenter un mur ou un passage.
`(nqueens 10)`	Ce programme compte le nombre de possibilités de placer 10 reines sur un damier de 10 par 10. Des listes sont utilisées.
`(primes 500)`	Ce programme retourne la liste des nombres premiers inférieurs à 500. Le programme débute par créer une liste avec les nombres de 2 à 500 et il enlève tous les nombres non premiers ainsi que ses multiples.
`(quotient 24)`	Ce programme effectue principalement des opérations arithmétiques de base (addition, soustraction et division) un grand nombre de fois.
`(Sort 100)`	Ce programme implante l'algorithme du tri par insertion. Il trie les 388 nombres en ordre croissant et décroissant en alternant 100 fois. Le tableau de nombres est alloué de façon statique.
`(Tak 22 14 4)`	Ce programme est le programme Tak standard. À chaque niveau de récursivité, l'ordre des paramètres est changé.
`(takl 15 9 3)`	Ce programme est comme Tak sauf qu'il utilise des listes au lieu des soustractions. Les listes contiennent les nombres de 15 à 0, de 9 à 0 et de 3 à 0. Au lieu de faire les soustractions, un `cdr` est effectué.

Tableau 41 : Liste des programmes utilisés

Nous avons effectué les tests avec les options suivantes :

1. Aucune option (temps de base).
2. Ajout de l'ordonnanceur.
3. Ajout du déroulement de boucles.
4. Ajout du déroulement de boucles et de l'ordonnanceur.
5. Ajout du registre CA.
6. Activation des régions.
7. Activation des régions et du déroulement de boucles.
8. Comparaison avec Gambit-C.

À l'exception du tableau 6.1, les nombres représentent le taux d'accélération. Le calcul utilisé est le suivant :

$$\frac{\texttt{[Vitesse de base]} - \texttt{[Vitesse avec option]}}{\texttt{[Vitesse avec option]}} + 1$$

Un nombre égal à un indique que le programme s'est exécuté en un temps identique et un nombre supérieur à 1 indique une accélération.

La moyenne algébrique (M. Alg) est calculée comme suit : $\frac{\sum_{i=1}^{n} X_i}{n}$.

La moyenne géométrique (M. Géo) est calculée comme suit : $\sqrt[n]{\prod_{i=1}^{n} X_i}$.

6.1) Temps de base

Le tableau qui suit nous donne le temps qu'ont pris les programmes pour s'exécuter sur chacun des processeurs. Aucune option d'optimisation n'a été utilisée. Ces temps serviront de base pour le calcul des taux d'accélération des autres tableaux.

	Alpha0	Alpha1	Mips	PPC	Sparc	Usparc	PI	PII	M68k
`Takl`	35.24	8.25	34.83	552.80	57.00	29.00	22.12	9.30	408.40
`Nqueens`	42.89	9.97	96.53	125.29	58.93	28.33	22.71	7.16	191.76
`Fib`	47.72	10.40	39.24	93.04	64.80	45.60	25.42	11.05	72.64
`Tak`	49.51	10.08	45.54	92.10	64.80	40.80	24.62	10.86	91.38
`Cpstak`	55.14	15.75	38.03	234.03	61.83	21.00	27.16	9.59	115.85
`Mazefun`	62.79	14.40	52.34	293.60	82.67	37.33	33.00	10.88	260.40
`Nrev`	63.10	23.00	54.67	289.80	82.00	30.00	36.64	10.76	238.93
`Deter`	63.67	15.33	50.84	285.73	85.33	44.00	34.72	10.24	223.80
`Assq`	86.92	16.83	81.00	169.20	106.00	47.00	34.58	10.48	374.30
`Sort`	100.41	18.30	54.00	116.25	102.00	58.50	29.72	9.18	338.85
`Quotient`	223.14	30.80	46.52	39.74	49.60	44.00	38.94	9.28	37.96
`Primes`	277.08	46.80	69.83	477.00	91.50	52.50	51.23	10.13	249.90

Tableau 42 : Temps de base (en secondes)

Remarques :

- Le générateur de code pour le processeur Alpha utilise la division de la librairie mathématique du langage C. C'est pour cette raison que les tests `quotient` et `primes` sont beaucoup plus lents que les autres.
- Le PII détient 75% des temps les plus rapides. Les autres temps ne sont pas très loins des plus rapides.
- Le M68k performe mieux lorsque le programme et ses données entrent au complet dans la mémoire cache (`fib`, `quotient` et `tak`).

6.2) Code de base ordonnancé

Le tableau qui suit nous montre les accélérations obtenues après l'ajout du module d'ordonnancement.

	Alpha0	Alpha1	Mips	PPC	Sparc	Usparc	PI	PII	M68k
Tak1	0.70	1.00	1.03	1.00	1.06	1.00	1.01	1.01	1.16
Cpstak	0.92	1.05	1.08	1.00	0.96	1.00	1.00	1.05	1.02
Mazefun	0.97	1.01	1.05	1.00	1.03	0.93	1.01	1.00	1.03
Nrev	0.97	1.02	1.07	1.00	1.03	1.00	0.99	1.03	1.06
Sort	0.99	1.34	1.22	1.07	1.00	1.08	1.11	1.16	1.12
Primes	1.03	1.03	1.04	0.99	1.02	1.09	1.01	0.99	1.03
quotient	1.03	1.04	1.02	1.01	1.04	1.38	1.01	1.01	1.80
Deter	1.04	1.06	1.11	1.01	1.07	1.14	1.00	1.01	1.10
Nqueens	1.06	1.06	1.11	1.07	1.04	1.04	1.04	1.03	1.08
Assq	1.12	1.23	1.00	1.01	1.01	1.02	1.00	1.00	1.02
Fib	1.13	1.10	1.16	1.03	1.05	1.07	1.00	0.98	1.07
Tak	1.20	1.25	1.06	1.07	1.04	1.06	1.00	1.11	1.19
M. Alg.	1.01	1.10	1.08	1.02	1.03	1.07	1.02	1.03	1.14
M. Geo.	1.01	1.09	1.08	1.02	1.03	1.06	1.01	1.03	1.13

Tableau 43 : Ajout de l'ordonnanceur

Remarques :

- Le code du Pentium (I et II) est très difficile à ordonnancer car le nombre de registres est trop petit.
- Le programme Sort est celui qui a le plus profité de l'ordonnanceur (moyenne de 1.12 du taux d'accélération). Ceci est probablement dû aux mémoires caches qui sont bien construites sur les nouvelles versions des processeurs (PII et Alpha1).

6.3) Déroulement des boucles

Le tableau qui suit nous donne l'accélération obtenue lorsque nous ajoutons l'option de déroulement des boucles à la compilation de base.

	Alpha0	Alpha1	Mips	PPC	Sparc	Usparc	PI	PII	M68k
quotient	0.75	1.03	1.01	1.05	1.02	1.04	0.99	0.95	1.60
Mazefun	0.97	1.02	1.02	1.02	1.07	1.17	0.98	1.06	0.96
Nrev	0.99	1.04	1.08	1.03	1.05	1.15	0.94	1.03	1.16
Takl	0.99	1.00	1.00	1.00	1.00	0.97	1.06	1.07	1.57
Nqueens	1.01	1.02	1.10	1.10	1.06	1.00	0.91	0.97	1.02
Primes	1.02	1.01	1.02	0.99	1.02	1.06	0.98	0.91	0.98
Sort	1.02	1.08	0.89	0.96	0.99	1.11	0.95	1.16	1.81
Deter	1.04	1.06	1.08	1.02	1.07	1.14	1.03	1.10	1.10
Cpstak	1.07	1.00	1.13	1.04	1.06	1.06	1.04	1.07	1.04
Tak	1.09	1.09	1.06	1.07	1.10	1.17	1.12	1.10	1.27
Fib	1.26	1.18	1.20	1.28	1.23	1.31	1.03	1.12	1.29
Assq	1.38	1.36	1.41	1.60	1.36	1.15	1.20	1.35	1.01
M. Alg.	1.05	1.07	1.08	1.10	1.09	1.11	1.02	1.07	1.23
M. Geo.	1.04	1.07	1.08	1.09	1.08	1.11	1.02	1.07	1.21

Tableau 44 : Ajout du déroulement de boucles

Nous pouvons remarquer que les tests n'effectuant aucune allocation dynamique ont un taux d'accélération plus élevé que les autres. Ces programmes profitent plus du déroulement de boucles.

6.4) Déroulement de boucles et ordonnanceur

Dans le tableau qui suit, nous avons les taux d'accélération lorsque nous activons les options pour le déroulement de boucles ainsi que l'ordonnanceur.

	Alpha0	Alpha1	Mips	PPC	Sparc	Usparc	PI	PII	M68k
Takl	0.71	1.05	1.06	1.00	1.06	1.00	1.07	1.07	1.60
Mazefun	0.96	1.04	1.09	0.97	1.07	1.17	0.99	1.06	0.99
Nrev	0.98	1.04	1.16	1.04	1.05	1.25	0.96	1.29	1.21
quotient	1.03	1.05	1.03	1.05	1.07	1.40	1.01	0.96	1.74
Primes	1.04	1.03	1.04	1.00	1.03	1.09	0.98	0.92	1.02
Cpstak	1.05	1.02	1.20	1.04	1.04	1.00	1.04	1.09	1.08
Deter	1.08	1.11	1.18	1.03	1.14	1.32	1.04	1.11	1.14
Nqueens	1.11	1.13	1.22	1.23	1.11	1.04	0.98	1.00	1.06
Sort	1.11	1.39	1.25	1.03	1.01	1.34	1.08	1.23	2.10
Fib	1.32	1.24	1.30	1.36	1.25	1.33	1.06	1.10	1.24
Tak	1.35	1.40	1.16	1.15	1.16	1.24	1.12	1.20	1.39
Assq	1.42	1.36	1.42	1.61	1.36	1.15	1.21	1.34	1.02
M. Alg.	1.10	1.16	1.18	1.13	1.11	1.19	1.05	1.11	1.30
M. Geo.	1.08	1.15	1.17	1.11	1.11	1.19	1.04	1.11	1.26

Tableau 45 : Ordonnanceur et déroulement de boucles

Remarques :

- Nous voyons que l'accélération est assez importante. Ceci est dû au fait que l'ordonnanceur a beaucoup plus de liberté puisque les blocs de base sont plus gros.

- Le PII a une accélération beaucoup plus forte que le PI car la longueur des blocs ainsi que la réallocation de registres («*register renaming*») effectuée par le processeur lui permet de diminuer les dépendances entre les instructions.

6.5) Utilisation du registre CA

Le tableau suivant nous montre l'accélération obtenue par l'ajout du registre CA («*code anchor*») au code de base. Les processeurs PI, PII et M68k ne profitent pas de cette option car la construction d'adresses se fait en une seule instruction.

	Alpha0	Alpha1	Mips	PPC	Sparc	Usparc
Mazefun	0.93	0.97	0.98	0.99	1.00	1.08
Assq	0.98	1.00	1.00	1.00	1.00	0.81
Nqueens	0.98	0.94	1.00	1.00	0.98	0.96
quotient	0.98	1.00	1.00	1.00	1.01	1.01
Cpstak	1.00	1.00	1.01	1.01	1.00	1.00
Tak	1.00	0.99	0.99	0.98	0.96	1.03
Takl	1.00	0.99	0.99	1.00	0.98	0.94
Deter	1.01	1.01	0.99	1.00	1.00	1.00
Nrev	1.01	1.00	1.00	1.01	1.00	0.94
Primes	1.01	1.00	1.00	0.99	1.00	0.97
Sort	1.05	1.03	1.00	1.00	0.97	0.98
Fib	1.07	1.04	1.03	1.04	1.06	1.00
M. Alg.	1.00	1.00	1.00	1.00	1.00	0.98
M. Geo.	1.00	1.00	1.00	1.00	1.00	0.97

Tableau 46 : Utilisation du registre CA

Remarques :

- Les accélérations obtenues ne sont pas importantes car la gestion du registre CA demande d'ajouter *trop* d'instructions. La taille des fichiers générés lorsque nous utilisons cette option est supérieure (très rarement égale) lorsque nous utilisons cette technique.
- Les tests qui profitent vraiment de cette option (Sort et Fib) sont ceux qui sont très petits. Pour ces tests, presque tous les blocs de base construisent une adresse (utilisent le registre CA).

6.6) Régions

Dans le tableau qui suit, nous avons les taux d'accélération lorsque nous discriminons entre les zones (régions) de mémoire. Les régions sont des zones comme la pile, le tas et les variables globales.

	Alpha0	Alpha1	Mips	PPC	Sparc	Usparc	PI	PII
Assq	0.80	0.90	0.70	1.01	0.73	0.76	0.83	0.76
Cpstak	0.86	1.05	1.07	1.00	0.92	0.97	0.97	0.97
Deter	0.96	0.96	1.12	1.01	0.98	0.91	0.96	0.94
Fib	0.82	0.88	1.12	0.96	0.84	0.80	0.96	0.88
Mazefun	1.01	1.00	1.07	1.00	1.00	1.00	1.02	0.95
Nqueens	0.97	0.96	1.02	1.01	0.96	1.03	1.08	1.06
Nrev	1.07	0.99	1.07	1.00	0.99	0.90	1.05	1.05
Primes	1.01	1.01	1.03	1.00	1.01	1.02	1.03	1.09
Sort	0.92	1.18	1.22	1.00	1.02	0.99	1.10	1.00
quotient	1.01	1.00	1.02	1.00	0.97	1.01	1.01	1.06
Tak	0.91	0.99	1.07	1.01	0.93	0.90	0.91	0.98
Takl	0.73	0.97	1.05	1.00	1.06	1.01	0.97	0.93
M. Alg.	0.92	0.99	1.05	1.00	0.95	0.94	0.99	0.97
M. Geo.	0.92	0.99	1.04	1.00	0.95	0.94	0.99	0.97

Tableau 47 : Régions

Les accélérations de ce tableau sont généralement inférieures à un. Ceci est principalement dû au fait que le principe de localité des données s'applique moins et que ceci nuit lorsque l'on utilise des mémoires caches (entrelacement des variables en mémoire avec ceux sur la pile, ...).

6.7) Régions et déroulement de boucles

Dans le tableau qui suit, nous avons les taux d'accélération lorsque nous discriminons entre les zones (régions) de mémoire et que l'option de déroulement de boucles est activée.

Les résultats sont par rapport à l'option de déroulement de boucles et non au temps de base.

	Alpha0	Alpha1	Mips	PPC	Sparc	Usparc	PI	PII
Assq	1.32	1.21	1.00	1.00	1.36	1.44	1.21	1.33
Cpstak	1.04	1.01	1.05	0.99	1.05	1.04	1.04	1.10
Deter	1.03	1.05	1.11	1.01	1.11	1.16	1.03	1.11
Fib	1.04	1.04	1.15	1.04	1.15	1.32	1.09	1.13
Mazefun	1.03	1.04	1.16	1.00	1.05	1.04	0.99	1.08
Nqueens	1.07	1.11	1.09	1.01	1.10	1.06	0.96	0.99
Nrev	0.97	1.04	1.10	1.00	1.07	1.30	0.97	1.33
Primes	1.03	1.02	1.04	1.00	1.03	1.00	0.97	0.92
Sort	1.03	1.35	1.40	1.02	1.01	1.36	1.05	1.14
quotient	1.02	1.04	1.02	1.01	1.02	1.08	1.00	0.95
Tak	1.10	1.19	1.12	1.02	1.11	1.17	1.16	1.17
Takl	0.71	1.05	1.10	1.00	1.04	1.04	1.06	1.05
M. Alg.	1.03	1.10	1.11	1.01	1.09	1.17	1.04	1.11
M. Geo.	1.02	1.09	1.11	1.01	1.09	1.16	1.04	1.10

Tableau 48 : Régions et déroulement de boucles

L'accélération est plus importante de cette façon. Puisque les blocs de base sont plus gros, il contiennent plusieurs accès à la mémoire d'un même type de régions et leur ordonnancement se fait mieux.

6.8) Gambit-C et Gambit-RTL

Le tableau qui suit fait la comparaison entre Gambit-C et Gambit-RTL. Nous avons utilisé la fonction « `time` » de Gambit-C pour obtenir le temps d'exécution des tests. Cette fonction nous donne le temps total ainsi que le temps utilisé par le GC. Donc, pour comparer les deux compilateurs, nous avons soustrait le temps pris par le GC. Il est à noter que le chargeur (loader) de Gambit-RTL permet d'exécuter les programmes un certain nombre de fois en réinitialisant le tas à chaque itération.

Nous avons compilé les tests avec la commande « `gsc -dynamique test.scm` » et nous les avons exécutés avec la commande « `gsi test.o1` ».

Les résultats supérieurs à un indique que Gambit-C est plus rapide que Gambit-RTL.

	Alpha0	Alpha1	Mips	PPC	Sparc	Usparc	PI	PII	M68k
Deter	0.10	0.11	0.07	0.14	0.07	0.08	0.04	0.04	0.03
Tak	0.60	0.56	0.69	0.91	0.53	0.70	0.29	0.33	0.26
Fib	0.67	0.75	0.71	0.85	0.68	0.46	0.43	0.53	0.22
Mazefun	0.94	1.07	0.66	1.53	0.75	0.36	0.30	0.37	0.35
Assq	1.00	1.10	0.73	0.85	0.74	0.60	0.31	0.29	0.98
Nqueens	1.02	1.15	1.62	1.65	0.86	0.86	3.64	1.35	0.94
Cpstak	1.12	1.48	0.55	3.78	0.87	0.37	0.44	0.45	0.58
Takl	1.13	0.81	0.65	3.18	0.68	0.81	0.29	0.35	0.41
Nrev	1.18	1.67	0.80	3.27	0.83	0.65	0.50	0.44	0.70
Sort	1.75	1.53	0.83	1.22	0.86	0.92	0.46	0.48	0.49
Primes	2.57	2.07	0.56	4.82	0.55	0.67	0.45	0.38	0.46
Quotient	3.65	2.35	0.82	0.95	0.69	0.82	0.84	0.69	0.07
M. Alg.	1.31	1.22	0.72	1.93	0.68	0.61	0.67	0.48	0.46
M. Geo.	1.00	0.99	0.61	1.39	0.59	0.53	0.40	0.38	0.33

Tableau 49 : Gambit-C vs Gambit-RTL

- Nous voyons que les ratios sont généralement en faveur de Gambit-RTL (inférieur à 1).
- Les programmes `primes` et `quotients` sont beaucoup plus rapides avec Gambit-C sur DEC Alpha. Le code généré par Gambit-RTL fait un appel au module C (loader) qui lui fait un appel à la librairie mathématique. De plus, le loader fait une sauvegarde de tous les registres utilisés par le code RTL avant d'appeler la librairie mathématique.

7) L'état actuel des travaux

Au moment de la rédaction de ce mémoire, nous avions complété six générateurs de code différents. Nous pouvons donc utiliser Gambit-RTL avec les six générateurs suivants :

- RTL-Alpha
- RTL-i386
- RTL-M68k
- RTL-Mips
- RTL-Ppc
- RTL-Sparc

Les instructions RTL pour les nombres flottants ont été codées dans le i386, M68k et dans le Sparc. Le restant des instructions RTL a été implanté dans tous les générateurs de code.

De plus, nous avons créé un programme qui permet de tester automatiquement les générateurs de code. Ce programme fait appel à chacune des instructions dans plusieurs contextes et compare le résultat de l'opération avec celui généré par Scheme. Ce programme nous a permis d'accélérer le débogage des générateurs de code car les programmes générés sont spécialisés et ils utilisent très peu de fonctions RTL (par exemple : initialisation de deux registres, une addition entre ces registres, sauvegarde de la valeur).

De plus, nous avons créé un ordonnanceur assez général pour être utilisé dans le cadre de différents processeurs. Selon les paramètres utilisés, nous pouvons changer le comportement de l'ordonnanceur. Trois comportements différents ont été codés (blocs de taille fixe, blocs dynamiques, par instructions). Il sont normalement liés au type de processeurs utilisé. Ce programme a été conçu de façon très modulaire pour permettre d'effectuer des modifications assez facilement sans affecter les programmes qui utilisent déjà l'ordonnanceur.

Pour pouvoir utiliser l'ordonnanceur ainsi que les quatre générateurs de code dans le compilateur Gambit-RTL, quelqu'un doit compléter la conversion du code GVM (Gambit Virtual Machine) vers le code RTL. Seulement une partie de la conversion GVM à RTL a été effectuée. Nous avons codé l'ensemble minimal d'instructions GVM dont nous avions besoin pour tester l'ordonnanceur.

7.1) Améliorations possibles

Il est évident qu'il existe un nombre infini d'améliorations possibles. Nous allons donc énumérer quelques-unes de ces améliorations auxquelles nous avons pensées :

- Permettre d'avoir un ordonnancement plus efficace en entrelaçant l'allocation de registres avec l'ordonnanceur. Ceci pourrait donner un peu de flexibilité pour l'ordonnancement sur les processeurs disposant de peu de registres comme le i386. Par contre, l'utilisation de processeurs contenant très peu de registres tend à diminuer depuis quelques années. Il est à noter que cette modification demanderait d'effectuer de gros changements dans le compilateur Gambit-RTL.

- Il serait bien d'ordonnancer les instructions en utilisant plus d'un bloc de base à la fois. Puisque l'ordonnanceur est principalement utilisé dans le cadre de Scheme (les blocs de base sont petits), le fait d'ordonnancer plusieurs blocs de base consécutifs (selon le graphe de flot d'exécution), pourrait augmenter le parallélisme dans les zones où il en manque. Il est évident que cette amélioration n'est pas simple à implanter mais les résultats peuvent être intéressants.

- Pour permettre d'exprimer des concepts plus forts, l'opérateur OR devrait être modifié pour que nous puissions choisir plus d'une ressource dans une liste. Par exemple, la liste suivante pourrait être utile : `((or ((ALU1 1) (BUS1 1)) ((ALU2 1) (BUS2 1)) (MEM 2))`. Cet exemple permet d'avoir un lien entre les unités et les bus. Ce concept peut être utilisé dans certains processeurs.

- La gestion du registre CA (« *code anchor* ») demande d'ajouter trop d'instructions. Ce n'est pas tous les blocs de base qui ont besoin de ce registre. Donc, il serait bien d'avoir une analyse de haut niveau pour détecter et enlever les ajustements inutiles.

8) Conclusion

Ce travail se veut une étude des optimisations pouvant être utilisées dans le cadre de plusieurs générateurs de code. La plus grande partie de cette étude est la création d'un ordonnanceur unique utilisable dans nos générateurs de code. Le restant de l'étude visait à trouver des séquences de code plus rapides.

Les aspect étudiés au cours de notre travail sont les suivants :

- Les processeurs
  - Unité de traitement
  - Unité de branchement
  - Les registres
  - Expéditeur
- Les ordonnanceurs
- Les accès à la mémoire et la construction d'adresses

Nous avons vu que l'utilisation de l'ordonnanceur donne une amélioration de performance intéressante dans la plupart des cas. De plus, il est facilement possible de l'inclure dans les générateurs de code futurs.

De plus, les séquences d'instructions permettent une exécution plus rapide des programmes. Nous avons accéléré l'exécution des séquences de branchement avec un registre sur le i386 ainsi que la construction des adresses et des constantes sur le processeur DEC Alpha (passant de 6 instructions à 1 ou 2 selon le cas).

Malgré le fait que les processeurs soient de plus en plus puissants (vitesse et fonctionnalité), nous avons pu voir qu'il est profitable d'optimiser le code généré par les compilateurs. Nous avons montré que :

- le choix des instructions machine était d'une importance considérable et qu'il pouvait faire varier grandement les résultats.
- Les changements dans les modèles (représentation des objets, ancre, ...) utilisés nous ont permis d'avoir du code qui se rapproche plus des processeurs actuellement utilisés.
- les optimisations de haut niveau permettent de générer du code qui utilise plus les points forts et moins les points faibles des processeurs.

Les processeurs plus anciens et moins puissants sont souvent utilisés dans des projets où plusieurs restrictions ont été imposées. Le code généré pour ces projets doit donc être le plus compacte et le plus rapide possible. Si les optimisations ne

sont pas assez importantes, l'intérêt d'utiliser un langage de plus haut niveau que le langage machine est presque nul.

Pour les raisons ci-haut mentionnées, nous croyons que nos objectifs ont été atteints.

Bibliographie

[AAH] *Alpha architecture handbook*, Digital, 1992, 215p.

[ADTJ0] A *200-MHz 64-bit Dual-issue CMOS Microprocessor*, Digital Technical Journal – 21064, Vol. 4, No. 4, Special Issue 1992.

[ADTJ1] C*ircuit Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU*, Digital Technical Journal – 21164, Vol. 7, No. 1, 1995

[ADTJ2] I*nternal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor*, Digital Technical Journal - 21164, Vol. 7, No 1, 1995

[AEAK92] ERTL, M. Anton, Krall, Andreas (1992), *Instruction Scheduling for complex pipelines*, Compiler Construction (CC'92), Springer LNCS 641, p207 à 218

[BEH91] BRADLEE, David G., Eggers, Susan J., Henry, Robert R. (1991), *Integrating register allocation and instruction scheduling for RISCs*, SIGPLAN notices 26(4), juin, p122 à 131.

[CJ84] CHAILLOUX, Jérôme (1984), *La machine LLM3*, (Mai 1984)

[CN95] NORRIS, Cindy (1995), *COOPERATIVE REGISTER ALLOCATION AND INSTRUCTION SCHEDULING*, University of Delaware, 107p.

[CNLP1] NORRIS, Cindy, Pollock, Lori (1993), *An experimental study of several cooperative register allocation & instruction scheduling strategies*, University of Delaware et Appalachian state university, 11p.

[CNLP2] NORRIS, Cindy, Pollock, Lori (1993), *A scheduler-sensitive global register allocator*, University of Delaware et Appalachian state university, 10p.

[DD96] DUBE, Danny (1996), *Un système de programmation Scheme pour micro-contrôleur*, Université de Montréal, Thèse (M. Sc.), QA 76 U54 1996 v.032.
Source : Http://www.iro.umontreal.ca/~dube/memoire.ps.gz

[DEC] D*EC OSF/1 : Assembly language programmer's guide*, Digital, AA-PS31A-TE

[DM94] DE MICHELI, Giovanni (1994), *Synthesis and optimization of digital circuits*, McGraw-Hill, ISBN 0-07-016333-2, 580p.

[DRK92] KERNS, Daniel R. (1992), *Balanced scheduling : Instruction scheduling when memory latency is uncertain*, University of washington, technical report 92-11-03, 33p.

[HZ96] HOOVER, Roger, Zadeck, Kenneth (1996), *Generating machine specific optimizing compilers*, Conference Record of POPL '96: The ACM SIGPLAN-SIGACT

Symposium on Principles of Programming Languages, pp. 219-229, 21-24 January.

[IIP] *Architectural features of the Pentium Processor Family*, Intel, Source : http://www.intel.com, document : 24319001.pdf

[JFB89] BARLETT, Joel F. (1989), *Scheme->C a Portable Scheme-to-C Compiler*, wrl Research Report 89/1, Western Research Laboratory, Digital, 25 p.

[JGHW] GOODMAN, James R., Wei-chung Hsu (1988), *Code Scheduling and register allocation in large basic blocks*, Conference Proceedings 1988 International Conference on Supercomputing, July 1988, pp. 442-452.

[JLSE] LO, Jack L., Eggers, Susan J. (1995), *Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism*, University of Washington, Department of Computer Science and engineering, 12p.

[KRHPA] KELSEY, Richard, Jonathan Rees, Paul Hudak, James Philbin, Norman Adams (1986) , *ORBIT: An Optimizing Compiler for Scheme*, SIGPLAN Notices, July 1986, 21(7), pp. 219-233.

[LHR92] L. HUMMEL, Robert (1992) , *The processor and coprocessor*, Emeryville, Ziff-Davis PRESS, 761 p.

[MF97] FEELEY, Marc (1997), *The gambit virtual machine*, Version 2.0.2, Université de Montréal, 8 janvier 1997, 29p.

[MFJM90] FEELEY, Marc, James S. Miller, *A Parallel Virtual Machine for Efficient Scheme Compilation*, Proceedings of the 1990 ACM SIGPLAN Conference on Lisp and Functional Programming, Nice, France, June 1990, pp. 119-130.

[MH95] HOPPER, Michael A. (1994), *Register allocation*, School of Electrical engineering, Georgia Institute of Technology, Qualifying Exam, Mars 1994, 28p.
Source : http://www.ee.gatech.edu/users/mhopper/papers/qual.ps

[MIPS1] M*IPS R4000 Microprocessor User's Manual*, Source : http://www.mips.com

[MIPS2] KANE, Gerry (1988), *Mips RISC ARCHITECTURE*, Prentice Hall, ISBN 0-13-584749-4

[MPSR95] MOTWANI, Rajeev, Palem, Krishna V., Sarkar, Vivek, Reyen, Salem (1995), *Combining Register allocation and instruction scheduling*, Courant Institute, TR698, Juin 1995.

[MR96] *Digital 21264 Sets new standard*, Microprocessor report, Vol. 10, Issue 14, 28 octobre 1996.

[MUM] M*68060 User's Manual*, Motorola, Source :
Http://www.mot.com/SPS/HPESD/aesop/680X0/040/040UM.pdf

[PPC1] *PowerPC 604 RISC Microprocessor User's Manual*, Source : http://www.chips.ibm.com:80/products/powerpc/chips/604_um.pdf

[PPC2] *PowerPC 601 RISC Microprocessor User's Manual*, Motorola, MPC601UM/AD,

[R^{4}RS] CLINGER, William, Rees, Jonathan (editeurs) (1991), *Revised ^{4} Report on the Algorithmic Language Scheme*, 55p.

[SJB87] BEATY, Steven J. (1987), *THESIS : Register allocation and assignment in a retargetable microcode compiler using graph coloring*, Technical Report, Computer Science Department, Colorado State University, May, 50p.

[SSP93] PINTER, Shlomit S. (1993), *Register Allocation with Instruction Scheduling : a New Approach*, ACM SIGPLAN Notices, 28(6), pp. 248-257, June 1993.

[TD95] TABAK,DANIEL (1995) , *ADVANCED MICROPROCESSORS*, McGraw-Hill, Inc., 523 p.

[ZHHL] ZORN, Benjamin, Paul Hilfinger, Kinson Ho et James Larus (1987), *SPUR Lisp : Design and Implementation*, Computer Science Division-EECS, University of California at Berkeley, Technical Report No. UCB/CSD 87/373, September 1987.

Appendice A : Source du de l'ordonnanceur

```
;;; Scheduler generique.
;;; Fonctions de type "public" :
;;;
;;;  1- (OPT-Set-Options Type-Block Block-Size isDelayBranch?)
;;;  2- (OPT-Basic-Block-Init)
;;;  3- (OPT-Add-Ins thunk read-lst write-lst use-lst link-with-next?)
;;;  4- (OPT-Basic-Block)
;;;
;;; Variables de type "public" :
;;;
;;; 0- OPT-INFO-BASE    : 'Block 'NoBlock 'Basic
;;; 1- OPT-INFO-N       : 2
;;; 2- OPT-INFO-DELAY   : #t ou #f ; Delay branch
;;;
;;; Format de OPT-Set-Options :
;;;   Type-Block : si #f, ne change rien. Sinon, met la variable OPT-INFO-BASE a
;;;                Type-Block.
;;;   Block-Size : si #f, ne change rien. Sinon, met la variable OPT-INFO-N a
;;;                OPT-INFO-N
;;;   isDelayBranch? : Est-ce que le cpu est de type delay-branch? Si oui,
;;;                on DOIT toujours finir le basic block de cette facon :
;;;                ...
;;;                (OPT-Add-Ins thunk-jump ... (pc) ... )
;;;                (OPT-Add-Ins thunk-noop ... (pc) ... )
;;;                (OPT-Basci-Block)
;;;
;;; Format de OPT-Add-Ins :
;;;  thunk     : Le code qui va generer l'asm.
;;;              ex: (lambda () (asm-16 #x90) (if list (asm-list "nop")))
;;;  read-lst  : List de paire de 'unit/reg' qui sont lu. (unit/reg delay)
;;;              ex: '( (r1 0) (memory 3))
;;;                   r1 est lu mais peut etre utilise 0 cycle plus tard...
;;;                   memory est lu et bloque pour 3 cycles
;;;  write-lst : Comme read-lst mais en ecriture.
;;;              ex: '( (stack 2) (r8 34) (f15 4))
;;;                  stack modifie et "bloque" pour 2 cycles
;;;                  r8 et f15 modifie.
;;;  use-lst   : Liste de units qui sont utilise. Utile dans le cas ou
;;;              les units sont pipeline. Genre le ALU est bloque pour
;;;              1 cycle mais le resultat sera disponible dans 5 cycles.
;;;              ex : '((ALU 1))
;;;  link-with-next? : Est ce que cette instruction DOIT etre suivit
;;;              de la suivante.  Par ex sur INTEL, si on modifie les
;;;              registres SS:SP, il faut les faire un a la suite de
;;;              l'autre. Sur Sparc, il faut qu'un NOOP suive le
;;;              JUMP a cause du delay-branch (dans le cas d'un opt
;;;              poche!).
;;;
;;; Algo :
;;;  1- Creation du graph d'interference de facon incremental
;;;     a chaque call a OPT-Add-Ins.  Ce module tien une liste
;;;     interne qui est utilise a cette fin...  Chaque call a
;;;     la fct OPT-Add-Ins cree un vecteur qui contient :
;;;     (vector min max thunk list-reg prec next N)
;;;     ou :
;;;       min : peut pas etre schedule avant cycle "min".
;;;       max : Fait un sort sur max et ce sert de ceci pour scheduler.
;;;       thunk : Le code a exec lorsque opt..
;;;       list-reg : append de use-lst, read-lst, write-lst
;;;       prec : list d'arc. ie tout les arcs de ?X? vers N et ?X?>N
```

```
;;;       next : list d'arc. ie tout les arcs de N vers ?X? et ?X?<N.
;;;       N : N ieme ajout d'instruction.
;;;  2- Lorsque l'on call la fct "OPT-Basic-Block", l'optimiseur fait:
;;;      - ASAP(as soon as posible) pour trouver la duree du chemain
;;;        qui part de la 1re inst jusqu'a la derniere. (Sert aussi
;;;        a trouver le chemain critique).
;;;      - ALAP(as late as posible) Pour trouver le cycle maximum pour
;;;        l'exec d'une instruction.
;;;      - Sort avec le resultat du ALAP.
;;;      - Tant qu'il y a des instructions a scheduler:
;;;        - Prend une instruction dont les resultats
;;;          sont disponible au cycle X. (commence a chercher
;;;          par le ALAP le plus petit[plus pressant!]).
;;;        - La schedule.
;;;        - Arrenge la liste des Read/Write/Use... ex:
;;;          si le reg AX est ecrit et disponible dans 4 cycles,
;;;          ajuste la liste avec (AX (cycle courrant+4)).
;;;  3- Call tout les thunk dans l'ordre dicte par (2).
;;;
;;; NB: Le (2) change selon Block,NoBlock,Basic...
;;;
(define OPT-INFO-BASE 'Block) ; Block NoBlock Basic
(define OPT-INFO-N 2)
(define OPT-INFO-DELAY #f)

(define OPT-Ins '())
(define OPT-Link-thunk '())
(define OPT-Link-rlst '())
(define OPT-Link-wlst '())
(define OPT-Link-ulst '())
(define OPT-r-lst '())
(define OPT-w-lst '())
(define OPT-cant-Add #f)
(define OPT-N -1)

(define (OPT-Label l)
  (OPT-Add-Ins (lambda () (asm-label l))
               '()
               '((pc 0))
               '()
               #f)
  (OPT-Basic-Block))

(define (OPT-Set-Options Type-Block Block-Size isDelayBranch?)
  (if Type-Block
      (set! OPT-INFO-BASE Type-Block))
  (if Block-Size
      (set! OPT-INFO-N Block-Size))
  (set! OPT-INFO-DELAY isDelayBranch?))

(define (OPT-Basic-Block-Init)
  (set! OPT-Ins '())
  (set! OPT-Link-thunk '())
  (set! OPT-Link-rlst '())
  (set! OPT-Link-wlst '())
  (set! OPT-Link-ulst '())
  (set! OPT-r-lst '())
  (set! OPT-w-lst '())
  (set! OPT-N -1)
  (set! OPT-cant-Add #f))

;;;************************************************************
;;;************************************************************
;;;************************************************************
;;;************************************************************
```

```
;;; helper pour les fct sur la memoire... la 'region'
(define (OPT-Map-Region region duree)
  (define (f i)
       (list i duree))
  (if (pair? region)
         (map f region)
         (list (f region))))

;;;*************************************************************
;;;*************************************************************
;;;*************************************************************
;;;*************************************************************
;;; Creation du graph de dependance
;;;*************************************************************
;;;*************************************************************
;;;*************************************************************
;;;*************************************************************
;;; Add-Ins : Ajout d'une instruction
;;;             thunk    : lambda qui contient le code
;;;             cur-read : liste des regs lue
;;;             cur-write: liste des regs ecrit
;;;             use-list : liste des units utilises
;;;             link-with-next? : si #t, met le thunk/read/write/use dans des listes
;;;                       temporaire et fait le "concat" avec prochaine instruction
(define (OPT-Add-Ins thunk cur-read cur-write use-lst link-with-next?)
  (if OPT-cant-Add
      (begin
               (display "OPT-Add-Ins a ete caller mais OPT-Basic-Block n'a pas encore
terminer...") (newline)
               (display "thunk=")                    (pp thunk) (newline)
               (display "cur-read=")         (pp cur-read) (newline)
               (display "cur-write=")        (pp cur-write) (newline)
               (display "use-lst=")        (pp use-lst) (newline)
               (display "link-with-next?=")(pp link-with-next?) (newline)
               (error "find the bug in your prog!")))
  (if (not rtl-schedule?) (thunk))

  (set! cur-read (append cur-read '((pc 0))))

  (set! OPT-Link-thunk (append OPT-Link-thunk (list thunk)))
  (set! OPT-Link-rlst (append OPT-Link-rlst cur-read))
  (set! OPT-Link-wlst (append OPT-Link-wlst cur-write))
  (set! OPT-Link-ulst (append OPT-Link-ulst use-lst))

  (if (not link-with-next?)
      (begin
                              (set! OPT-N (+ OPT-N 1))
                              (let* ((a (append OPT-Link-ulst OPT-Link-rlst))
                                                (b (append a OPT-Link-wlst)))
                                     (set! OPT-Ins (append OPT-Ins
                                     (list (vector 0 32000
                                              OPT-Link-thunk b '() '() OPT-N)))))
                              (if (app OPT-Link-wlst OPT-r-lst)
                                              (set-it OPT-Link-wlst OPT-r-lst OPT-N))
                              (if (app OPT-Link-rlst OPT-w-lst)
                                              (set-it OPT-Link-rlst OPT-w-lst OPT-N))
                              (if (app OPT-Link-wlst OPT-w-lst)
                                              (set-it OPT-Link-wlst OPT-w-lst OPT-N))
                              (set! OPT-r-lst (diff (union OPT-r-lst
                                     OPT-Link-rlst OPT-N) OPT-Link-wlst))
                              (set! OPT-w-lst (union (diff OPT-w-lst
                                     OPT-Link-wlst) OPT-Link-wlst OPT-N))

                              (set! OPT-Link-thunk '())
                              (set! OPT-Link-rlst '())
```

```
                              (set! OPT-Link-wlst '())
                              (set! OPT-Link-ulst '())
                              (if (= OPT-N 40) (OPT-Basic-Block)))))

;;; set-it : pour toute la liste des regs(from), regarde pour trouver
;;;          son correspondant dans la a-list(into) et set les "ptr"
;;;          d'un(cdr into) a l'autre(OPT-N)
;;;          ex1:from='(r1 r2) into='((r2 1 T1) (r3 8 T2)) OPT-N=15
;;;               alors ptr de 1 a 15
;;;          ex2:from='((r1 2) (r2 3)) into='((r2 1 T1) (r3 8 T2)) OPT-N=15
;;;               alors ptr de 1 a 15
(define (set-it from into N)
  (define f 0)

  (let loop ((lfrm (car from))
             (next (cdr from)))
    (set! f (lambda (x)
              (cond ((and (pair? lfrm)
                          (equal? (car x) (car lfrm)))
                                  (setIns (cadr x) N (caddr x)))
                    ((equal? (car x) lfrm)
                                  (setIns (cadr x) N (caddr x))))))
    (map f into)
    (if (null? next)
        #t
        (loop (car next) (cdr next)))))

;;; cree les lien dans les 2 directions...
;;; NB : regarde si n'existe pas deja... (ex : cx=ax+ax ou qqch du genre!)
;;; N1 : 1er
;;; N2 : 2m
(define (setIns N1 N2 disp-time)
  (define (cdrN N L)
    (if (= 0 N)
        (car L)
        (cdrN (- N 1)
              (cdr L))))
  (let ((a (cdrN N1 OPT-Ins))
        (b (cdrN N2 OPT-Ins)))
    (let ((v1 (vector-ref a 4))
          (v2 (vector-ref b 5)))
      (let ((find1 (assq N2 v1))
            (find2 (assq N1 v2)))
        (if (not find1)
            (vector-set! a 4 (append v1 (list (cons N2 disp-time)))))
        (if find1
            (set-cdr! find1 (max (cdr find1) disp-time)))
        (if (not find2)
            (vector-set! b 5 (append v2 (list (cons N1 disp-time)))))
        (if find2
            (set-cdr! find2 (max (cdr find2) disp-time)))
        ))))

;;; Fait l'union d'une a-list avec une liste de regs
;;; ex : regpos = '((r1 0 1) (r2 3 4))
;;;      regs   = '(r1 r3)  ou '((r1 0) (r3 4))
;;;      N      = 8
;;;   Retourne : '((r1 0 1) (r2 3 3) (r1 8 0) (r3 8 4))
(define (union regpos regs N)
  (define (f x)
    (if (pair? x)
        (list (car x)
              N
              (cadr x))
        (list x N 1)))
```

```
  (let ((r (map f regs)))
    (if (null? regpos)
        r
        (append r regpos))))

;;; Remove les regs de la a-liste regpos
;;;  regs = '((r1 3) (r2 7))
;;; regpos= '((r1 1 3) (r5 8 4))
(define (diff regpos regs)
  (define (diff-k regpos regs rc)
    (cond ((null? regpos) rc)
          ((null? (car regpos))
                       (diff-k (cdr regpos) regs rc))
          ((assq (caar regpos) regs)
                        (diff-k (cdr regpos) regs rc))
          (else         (diff-k (cdr regpos) regs (cons (car regpos) rc)))))

  (cond ((null? regpos) '())
        ((null? regs)   regpos)
        (else (diff-k regpos regs '()))))

;;; retourne #t si un des 'regs'(a-list) appartient a la a-list regpos
(define (app regs regpos)
  (if (null? regs)
      #f
      (cond ((and (pair? (car regs))
                  (assq (caar regs) regpos))
                                    #t)
            ((assq (car regs) regpos)
                                    #t)
            (else
             (app (cdr regs) regpos)))))

;;;************************************************************
;;; ASAP et ALAP
;;;************************************************************

;;; ALAP.  veut en entre le resultat de ASAP.
(define (alap ins-vect)
  ;;; Partie 2 de ALAP. Loop principale
  (define (alap-2 ins-vect)
    (define l (vector-length ins-vect))
    (let loop ((pos (- l 1)))
      (let ((v (vector-ref ins-vect pos)))
       (asap-alap-set-value ins-vect
                            1
                            min
                            -
                            (vector-ref v 5)
                            (- (vector-ref v 1) 0))
       (if (> pos 0)
           (loop (- pos 1))
           ins-vect))))
  ;;; Set la value du ALAP a Value
  (define (set-lst value)
    (let loop ((pos 0)
               (max (vector-length ins-vect)))
    (let* ((vect (vector-ref ins-vect pos)))
      (if (= pos (- max 1))
          (vector-set! vect 1 value)
          (vector-set! vect 1 (- value 1)))
      (if (< pos (- max 1))
          (loop (+ 1 pos) max)
          #t))))
  (set-lst (max-lst (make-list ins-vect 0)))
```

```
  (alap-2 ins-vect))

;;; Fait le ASAP de la LIST d'instruction
(define (asap ins)
  ;;; partie 2 de ASAP... c'est la boucle. le tout est preparer avant
  (define (asap-2 ins-vect)
    (let loop ((pos 0)
               (top (vector-length ins-vect)))
      (let ((v (vector-ref ins-vect pos)))
       (asap-alap-set-value ins-vect
                            0
                            max
                            +
                            (vector-ref v 4)
                            (+ 0 (vector-ref v 0)))
       (if (< pos (- top 1))
           (loop (+ 1 pos) top)
           ins-vect))))
  (let* ((r (asap-2 (list->vector ins)))
         (l (vector-length r))
         (lst (make-list r 0))
         (m   (max-lst lst)))
    (if (equal? (list m) (member m lst))
        #t
        (vector-set! (vector-ref r (- l 1)) 0 (+ m 1)))
    r))

;;; met le poids correct pour la list des "suivant"(lst)
;;; ie : pout les element de 'lst', prend le min(ou max)
;;;      entre 'v' et le poids dans le vecteur de l'instruction
;;; ex : vecteurs = (#(3 0 thunk ...) ...)
;;;      V = 4 et f=max  alors 3 trois est remplace par 4...
(define (asap-alap-set-value ins-vect disp f1 f2 lst V)
  (if (null? lst)
      #t
      (let* ((vect (vector-ref ins-vect (caar lst)))
             (t-v  (vector-ref vect (+ 0 disp))))
        (vector-set! vect (+ 0 disp) (f1 t-v (f2 V (cdar lst))))
        (asap-alap-set-value ins-vect disp f1 f2 (cdr lst) V))))

;;; Prend la liste d'instruction et fait une liste avec tout les poids.
;;; DISP donne si des poids de ASAP ou ALAP
(define (make-list ins-vect disp)
  (let loop ((pos 0)
             (max (vector-length ins-vect))
             (lst '()))
    (let* ((vect (vector-ref ins-vect pos))
           (v    (vector-ref vect (+ 0 disp))))
      (if (< pos (- max 1))
          (loop (+ 1 pos) max (cons v lst))
          (reverse (cons v lst))))))

;;; MAX mais sur une liste
(define (max-lst l)
  (if (null? l)
      0
      (let loop ((v (car l))
                 (l (cdr l)))
        (if (null? l)
            v
            (loop (max v (car l)) (cdr l))))))

;;;*************************************************************
;;; Scheduler
;;;*************************************************************
```

```
;;; 1) (Scheduler Ins)
(define (Func-Comp v i1 i2)
  (let ((p1 (vector-ref v i1))
        (p2 (vector-ref v i2)))
    (let ((v1 (vector-ref OPT-Ins p1))
          (v2 (vector-ref OPT-Ins p2)))
      (let ((a (vector-ref v1 1))
            (b (vector-ref v2 1)))
        (- a b)))))

(define (init-Vect-Reorder v)
  (let loop ((i 0)
             (max (vector-length v)))
    (if (= i max)
        v
        (begin
          (vector-set! v i i)
          (loop (+ i 1) max)))))

; (assqor '(or (a 1) (b 2) (c 3)) '((m . 1) (n . 2)
;          (a . 3) (w . 3) (c . 3) (w . 3)) 1 #t)
(define (assqor wantUse lst CurCycle retPair)
  (define (helper1 l)
    (if (null? l)
        -1
        (let ((a (assq (caar l) lst)))
          (if (or (and a (>= CurCycle (cdr a)))
                  (not a))
              (if retPair
                  (cons a (car l))
                  a)
              (helper1 (cdr l))))))

  (define (helper2 l)
    (if (null? l)
        (if retPair
            (cons #f (cadr wantUse))
            #f)
        (let ((a (assq (caar l) lst)))
          (if a
              (if retPair
                  (cons a (cadar l))
                  a)
              (helper2 (cdr l))))))

  (if (equal? 'or (car wantUse))
      (let ((a (helper1 (cdr wantUse))))
        (if (not (equal? -1 a))
            a
            (helper2 (cdr wantUse))))
      (let ((i (assq (car wantUse) lst)))
           (if retPair
               (cons i wantUse)
               i))))

(define (Can-Exec-Ins-UseList cpuUseList insUseList CurCycle)
  (if (or (null? insUseList)
          (null? cpuUseList))
      #t
      (let ((tmp (assqor (car insUseList) cpuUseList CurCycle #f)))
        (if tmp
            (if (>= CurCycle (cdr tmp))
                (Can-Exec-Ins-UseList cpuUseList (cdr insUseList) CurCycle)
                #f)
            (Can-Exec-Ins-UseList cpuUseList (cdr insUseList) CurCycle)))))
```

```
(define (Can-Exec-Ins OneIns UseList CurCycle)
  (let ((pred (vector-ref OneIns 5))
        (Use  (vector-ref OneIns 3)))
    (if (and (null? pred)
             (Can-Exec-Ins-UseList UseList Use CurCycle))
        #t
        #f)))

(define (Get-One-Ins LatenceVect Ins UseList CurCycle)
  (let loop ((i 0)
             (NbIns (vector-length Ins)))
    (cond ((>= i NbIns)  #f)
          ((equal? #f (vector-ref LatenceVect i))
               (loop (+ i 1) NbIns))
          (else
           (let ((pos (vector-ref LatenceVect i)))
             (if (Can-Exec-Ins (vector-ref Ins pos) UseList CurCycle)
                 (begin
                   (vector-set! LatenceVect i #f)
                   (vector-ref Ins pos))
                 (loop (+ i 1) NbIns)))))))

(define (Update-UseList insUseList cpuUseList CurCycle)
  (if (null? insUseList)
      cpuUseList
      (let* ((rc (assqor (car insUseList) cpuUseList CurCycle #t))
             (UsedRes (car rc))
             (WantRes (cdr rc))
             (time (+ CurCycle (cadr WantRes))))
        (if UsedRes
            (begin
              (set-cdr! UsedRes time)
              (Update-UseList (cdr insUseList) cpuUseList CurCycle))
            (Update-UseList (cdr insUseList)
                            (cons (cons (car WantRes) time)
                                  cpuUseList)
                            CurCycle)))))

(define (Update-Prev-List Ins ptr-next MyNumber)
  (if (null? ptr-next)
      #t
      (let* ((p  (caar ptr-next))
             (v  (vector-ref Ins p))
             (l  (vector-ref v 5))
             (l2 (diff l (list (cons MyNumber 'X)))))
        (vector-set! v 5 l2)
        (Update-Prev-List Ins (cdr ptr-next) MyNumber))))

;;; Version qui schedule par Block de N instructions.
;;; par ex, si N=2 : trouve 2 inst ok et inc cycle.
;;;                  trouve 1 inst ok, inc cycle, trouve 1 inst ok, repart avec 2
;;;*************************************************************************
;;;*************************************************************************
;;; Version qui Schedule N instruction a la fois, par Block
;;; EX avec N=3
;;;   Temps 1 2 3 4 Block
;;; Inst 1  X         1
;;;      2  X         1
;;;      3  X         1
;;;      4    X       2
;;;      5    X       2
;;;      6      X     2
;;;      7        X   3
;;;      8        X   3
```

```
;;;      9        X   3
;;;  Donc, en resume... si trouve I instructions (I<N), les schedule et repart
;;;  pour N-I.
;;;*****************************************************************************
;;;*****************************************************************************
(define Schedule-Me-Block-N 3)
(define (Schedule-Me-Block Ins LatenceOrderVect ReorderVect)
  (set! Schedule-Me-Block-N OPT-INFO-N)
  (let loop ((i      0)
             (NbIns  (vector-length Ins))
             (UseList '())
             (Cycle   0)
             (Disp 0))
    (if (>= i NbIns)
       ReorderVect
       (let ((k (Get-One-Ins LatenceOrderVect Ins UseList Cycle)))
         (if (not k)
             (loop i NbIns UseList (+ 1 Cycle) Disp)
             (begin
              (set! UseList (Update-UseList (vector-ref k 3) UseList Cycle))
               (Update-Prev-List Ins (vector-ref k 4) (vector-ref k 6))
               (vector-set! ReorderVect i (vector-ref k 6))
               (if (< (remainder Disp Schedule-Me-Block-N)(- Schedule-Me-Block-N 1))
                  (loop (+ 1 i) NbIns UseList Cycle (+ 1 Disp))
                  (loop (+ 1 i) NbIns UseList (+ Cycle 1) (+ 1 Disp)))))))))

;;;*****************************************************************************
;;; Version qui Schedule N instruction a la fois s'il peut... (NoBlock)
;;; EX avec N=3
;;;   Temps 1 2 3 4 Block
;;; Inst 1  X         1
;;;      2  X         1
;;;      3  X         1
;;;      4    X       2
;;;      5    X       2
;;;      6      X     3
;;;      7      X     3
;;;      8      X     3
;;;      9        X   4
;;;  Donc, en resume... si trouve I instructions (I<N),
;;;    les schedule et repart pour N.
;;; ex: i386, M68k?, ...
;;;*****************************************************************************
(define Schedule-Me-NoBlock-N 2)
(define (Schedule-Me-NoBlock Ins LatenceOrderVect ReorderVect)
  (set! Schedule-Me-NoBlock-N OPT-INFO-N)
  (let loop ((i      0)
             (NbIns  (vector-length Ins))
             (UseList '())
             (Cycle   0)
             (NthisCycle 0))
    (if (>= i NbIns)
       ReorderVect
       (let ((k (Get-One-Ins LatenceOrderVect Ins UseList Cycle)))
         (if (not k)
             (loop i NbIns UseList (+ 1 Cycle) 0)
             (begin
              (set! UseList (Update-UseList (vector-ref k 3) UseList Cycle))
               (Update-Prev-List Ins (vector-ref k 4) (vector-ref k 6))
               (vector-set! ReorderVect i (vector-ref k 6))
               (if (< NthisCycle (- Schedule-Me-NoBlock-N 1))
                  (loop (+ 1 i) NbIns UseList Cycle (+ 1 NthisCycle))
                  (loop (+ 1 i) NbIns UseList (+ Cycle 1) 0))))))))

;;;*****************************************************************************
```

```
;;;*****************************************************************************
;;; Version qui Schedule 1 instruction a la fois ('Basic)
;;; EX:
;;;   Temps 1 2 3 4 Block
;;; Inst 1  X         1
;;;      2    X       2
;;;      3      X     3
;;;      4        X   4
;;;  Donc, en resume... Schedule une inst, incremente le cycle courant et restart...
;;;*****************************************************************************
;;;*****************************************************************************
(define (Schedule-Me Ins LatenceOrderVect ReorderVect)
  (let loop ((i      0)
             (NbIns  (vector-length Ins))
             (UseList '())
             (Cycle   0))
    (if (>= i NbIns)
        ReorderVect
        (let ((k (Get-One-Ins LatenceOrderVect Ins UseList Cycle)))
          (if (not k)
              (loop i NbIns UseList (+ 1 Cycle))
              (begin
                (set! UseList (Update-UseList (vector-ref k 3) UseList Cycle))
                (Update-Prev-List Ins (vector-ref k 4) (vector-ref k 6))
                (vector-set! ReorderVect i (vector-ref k 6))
                (loop (+ 1 i) NbIns UseList (+ Cycle 1))))))))))

;;;*****************************************************************************
;;; Call le bon ordonnanceur
;;;*****************************************************************************
(define (OPT-Basic-Block)
  (define LatenceOrderVect 1)
  (define ReorderVect '())
  (define (UpdateDelayBranch)
    (let ((len (vector-length OPT-Ins)))
      (if (and OPT-INFO-DELAY (> len 2))
          (let* ((p1 (vector-ref ReorderVect (- len 3)))
                 (i1 (vector-ref OPT-Ins p1))
                 (p2 (vector-ref ReorderVect (- len 2)))
                 (i2 (vector-ref OPT-Ins p2))
                 (p3 (vector-ref ReorderVect (- len 1)))
                 (i3 (vector-ref OPT-Ins p3))
                 (depen2 (diff (vector-ref i2 3) '((pc 0))))
                 (depen1 (vector-ref i1 3)))
            (if (not (app depen2 depen1))
                (begin
                  (vector-set! OPT-Ins p1 i2)
                  (vector-set! OPT-Ins p2 i1)
                  (vector-set! i3 2 (list (lambda () #t))))))))))

  (set! OPT-cant-Add #t)
  (if (not (null? OPT-Ins))
      (begin
                (set! OPT-Ins (alap (asap OPT-Ins)))
                (set! LatenceOrderVect (make-vector (vector-length OPT-Ins)))
                (set! ReorderVect (make-vector (vector-length OPT-Ins)))
                (set! LatenceOrderVect (init-Vect-Reorder LatenceOrderVect))
                (set! LatenceOrderVect (sort-Vector LatenceOrderVect Func-Comp))
                (case OPT-INFO-BASE
                  ((Block)    (set! ReorderVect
                                (Schedule-Me-Block   OPT-Ins LatenceOrderVect ReorderVect)))
                  ((NoBlock) (set! ReorderVect
                                (Schedule-Me-NoBlock OPT-Ins LatenceOrderVect ReorderVect)))
                  ((Basic)    (set! ReorderVect
                                (Schedule-Me OPT-Ins LatenceOrderVect ReorderVect)))
```

```
                (else (error "Bad scheduling type : OPT-INFO-BASE=" OPT-INFO-BASE)))
              (UpdateDelayBranch)
              (let loop ((l (vector-length ReorderVect))
                         (i 0))
                     (if (= i l)
                         (begin
                           (OPT-Basic-Block-Init)
                           ReorderVect)
                         (let* ((num (vector-ref ReorderVect i))
                                (ins (vector-ref OPT-Ins num))
                                (thk (vector-ref ins 2)))
                           (if rtl-schedule?
                               (for-each (lambda (thk) (thk)) thk))
                           (loop l (+ i 1))))))))
    #f)
  (set! OPT-cant-Add #f))

;;;*************************************************************
;;; Utilities
;;;*************************************************************
(define (sort-Vector v f)
  ;;; Echange 2 elements d'un vecteur
  (define (swap i1 i2)
    (let ((v1 (vector-ref v i1))
          (v2 (vector-ref v i2)))
      (vector-set! v i2 v1)
      (vector-set! v i1 v2)))
  ;;; Fonction standard du qsort
  (define (qsort debut fin)
    ;;; temp=v[debut]
    ;;; i=debut+1   j=fin
    ;;; tant que v[i]<=temp => i++
    ;;; tant que v[j]> temp => j--
    ;;; si j>i alors recurse sur les 2 sous ensemble (debut+1..i-1 et j+1..fin)
    ;;; sinon swap v[i],v[j] et goto "tant que"
    (cond ((>= debut fin)      #t)
          ((= (+ 1 debut) fin)
                              (if (>= (f v debut fin) 0)
                                (swap debut fin)))
          (else
           ;;; Loop pour les I... tant que v[i]<=v[debut] => i++
           (let loopi ((i (+ debut 1))
                       (j fin))
             (if (and (<= i fin)
                      (> (f v debut i) 0))
                 (loopi (+ i 1) j)
                 ;;; Loop pour les J... tant que v[j]>v[debut] => j--
                 (let loopj ((i i)
                             (j j))
                   (cond ((and (> j debut)
                               (<= (f v debut j) 0))
                                 (loopj i (- j 1)))
                         ;;; si I>J alors SWAP v[debut],v[j] et recurse...
                         ((> i j)
                                 (swap debut j)
                                 (qsort debut (- i 2))
                                 (qsort (+ j 1) fin))
                         ;;; sinon (v[j]<=v[debut] et v[i]>v[debut]), swap it and continue
                         (else       (swap i j)
                                     (loopi i j)))))))))
  (qsort 0 (- (vector-length v) 1))
  v)
```

Appendice B : Liste complète des instructions RTL

```
RTL-comnt
RTL-local
RTL-publi
RTL-align
RTL-data1
RTL-data2
RTL-data4
RTL-dataw
RTL-dataf
RTL-datad
RTL-datas
RTL-datak
RTL-datal
RTL-datap
RTL-jumpl
RTL-jumpp
RTL-jumpi
RTL-jeqwc
RTL-jeqwr
RTL-jeqfr
RTL-jnewc
RTL-jnewr
RTL-jnefr
RTL-jltwc
RTL-jltwr
RTL-jltfr
RTL-jlewc
RTL-jlewr
RTL-jlefr
RTL-jgtwc
RTL-jgtwr
RTL-jgtfr
RTL-jgewc
RTL-jgewr
RTL-jgefr
RTL-movwl
RTL-movwp
RTL-movwc
RTL-movws
RTL-movwk
RTL-movwr
RTL-movfr
RTL-conwr
RTL-confr
RTL-loa1i
RTL-sto1i
RTL-loa2i
RTL-sto2i
RTL-loa4i
RTL-sto4i
RTL-loawi
RTL-stowi
RTL-loawg
RTL-stowg
RTL-loawp
RTL-stowp
RTL-loafi
RTL-stofi
RTL-loadi
RTL-stodi
RTL-negwr
RTL-notwr
RTL-addwc
RTL-addwl
RTL-addwr
RTL-subwr
RTL-mulwc
RTL-mulwr
RTL-divwc
RTL-divwr
RTL-remwc
RTL-remwr
RTL-andwc
RTL-andwr
RTL-iorwc
RTL-iorwr
RTL-xorwc
RTL-xorwr
RTL-srawc
RTL-srawr
RTL-srlwc
RTL-srlwr
RTL-sllwc
RTL-sllwr
RTL-negfr
RTL-absfr
RTL-trufr
RTL-roufr
RTL-sqrfr
RTL-expfr
RTL-logfr
RTL-sinfr
RTL-cosfr
RTL-tanfr
RTL-asifr
RTL-acofr
RTL-atafr
RTL-addfr
RTL-subfr
RTL-mulfr
RTL-divfr
```

Appendice C : Résumé sur les processeurs

Le tableau qui suit nous donne un résumé des principaux processeurs étudiés.
Processeur : Le nom du processeur.
#I/Cycle : Le nombre d'instruction pouvant être exécuté par cycle machine.
Unités : Le nom des principales unités de traitements.
Dyn./Fixe : Type d'exécutions. Soit dynamique, fixe ou par instruction.
Réordo. : Indique la présence d'une unité de réordonancement du code.

Processeur	#I/Cycle	Unités	Dyn/Fixe	Réordo.
Intel 486	1	Entiers Point flottant	Par Ins.	Non
Pentium	2	U et V pipe (Entier) F pipe (Flottant)	Dynamique 2 ins.	Non
Pentium Pro et II	3 (Format 4-1-1)	Entier (2 unités) Flottant (2 unités) Mémoire MMX (Si disponible)	Dynamique 3 ins.	Oui
M68K	2	Poep (Entier) Soep (Entier) Flottant (avec Peop)	Dynamique 2 ins.	Non
Alpha 21064	2	Entier Flottant Mémoire Branchement Mul. (Entier) Division (Flottant)	Fixe 2 ins	Non
Alpha 21164	4	Entier (2 unités) Flottant (+ et /) Flottant (*) Branchement	Fixe 4 ins	Non
Alpha 21264	4	Entier (2 unités) 2 adresse ALU 2 flottant *,/,+,sqr	Fixe. 4 ins	Oui
MIPS R4400	1	Entier Entier *,/ Mémoire Adresse	Par Ins.	Non
MIPS R10000	5	Entier (2 unités) Flottant (2 unités) Adresse	Non disp.	Non disponible (Probablement non)
PowerPC 604	4	Entier (2 unités) Entier (Multi cycle) Mémoire Flottant Branchement	Fixe 4 ins	Non

Remerciements

Je tiens à remercier ma plus que belle Julie Trep pour m'avoir enduré tout au long de cette longue épreuve. Je la remercie très fortement d'avoir bien voulu corriger ce texte à de multiples reprises ainsi que d'avoir fait l'épicerie si souvent ☺…

Je me dois de remercier aussi ma P'tite moman pour avoir fait une correction plus approfondie de plusieurs chapitres en fin de parcours.

Il est de mon devoir de dire merci à mon grand directeur de recherches, Marc Feeley. Ces idées, critiques et commentaires étaient très formateurs.

Pour finir, je me remercie d'avoir fait tout ceci pour me rendre à ce que je suis aujourd'hui.