Ordonnancement des instructions pour un processeur ARM endochrone

par

Hamza HALLI

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION
DE LA MAÎTRISE EN GÉNIE
M.Sc.A.

MONTRÉAL, LE 11 AVRIL 2017

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. François Gagnon, directeur de la maîtrise, Directeur de Mémoire
Département de génie électrique à l'École de technologie supérieure

M. Claude Thibeault, codirecteur de la maîtrise, Co-directeur
Département de génie électrique à l'École de technologie supérieure

M. Yves Blaquière, Président du Jury
Département de génie électrique à l'École de technologie supérieure

M. Thomas Awad, membre du jury
Directeur à OCTASIC

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 21 FÉVRIER 2017

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

En premier lieu, je tiens à remercier mon directeur de maîtrise M. Gagnon et mon co-directeur de maîtrise M. Thibeault pour les conseils, l'appui et le soutien qu'ils m'ont procurés tout au long de la réalisation de mon travail de recherche. Je tiens également à remercier nos partenaires de projet Octasic, pour l'ensemble des ressources et des outils mis à ma disposition. Finalement, je remercie mes parents ainsi que ma conjointe, Arthémise, pour leur soutien financier et moral, qui m'a permis d'aller au terme de ma maîtrise.

ORDONNANCEMENT DES INSTRUCTIONS POUR UN PROCESSEUR ARM ENDOCHRONE

Hamza HALLI

RÉSUMÉ

Les processeurs endochrones, par définition, utilisent des mécanismes locaux de synchronisation leur permettant de s'affranchir du maintien d'un signal d'horloge globale. Cette spécificité les rend moins énergivores comparativement aux processeurs synchrones. Toutefois, les processeurs endochrones sont moins populaires en raison du manque d'outils de design et de vérification ainsi que l'évolution rapide des processeurs synchrones en terme de performance.

Ce mémoire s'inscrit dans le cadre du projet AnARM visant à développer un processeur à usage général ARM basé sur une architecture endochrone. Ce mémoire vise plus particulièrement l'exploration des méthodes d'ordonnancement des instructions pour développer une stratégie d'ordonnancement, basée sur les caractéristiques architecturales de l'AnARM, dans le but d'en améliorer les performances.

L'ordonnancement des instructions est une optimisation du compilateur ayant un grand impact sur la qualité du code généré. Cette optimisation consiste à résoudre un problème NP-complet en tenant compte des contraintes imposées par l'architecture du processeur cible. Tandis que l'ordonnancement des instructions pour les architectures synchrone bénéficie d'une large couverture littéraire, l'ordonnancement pour les architectures asynchrones a été moins abordé, en raison des nouvelles contraintes imposées par les mécanismes de synchronisation utilisées par ces architectures.

Ce mémoire présente l'élaboration, l'implémentation et l'évaluation d'une stratégie d'ordonnancement pour le processeur endochrone AnARM. La méthode d'ordonnancement présentée dans ce mémoire utilise un modèle d'ordonnancent dynamique basé sur le comportement spatio-temporel de l'AnARM. Cette méthode a été implémentée au sein d'un compilateur commercial moderne et évaluée comparativement à des méthodes d'ordonnancement usuelles. La méthode d'ordonnancement présentée dans ce mémoire engendre des améliorations de la performance allant de 6,22% à 17,48%, tout en préservant l'avantage énergétique de l'architecture endochrone à l'étude.

Mots clés: Ordonnancement des instructions, Processeurs endochrones, Optimisation logicielle, Compilation

INSTRUCTION SCHEDULING FOR A SELF-TIMED ARM

Hamza HALLI

ABSTRACT

Self-timed processors use local synchronization mechanisms in the absence of a global clock signal. This specificity makes them less energy-consuming compared to synchronous processors. However, self-timed processors are less popular due to lack of design and verification tools as well as the rapid evolution of synchronous processors in terms of performance.

This thesis is part of the AnARM project which aims to develop a general purpose ARM processor based on a self-timed architecture. This thesis's particular goal is the exploration of instruction scheduling methods in order to develop a scheduling strategy, based on the architectural features of the AnARM processor, with the aim of improving its performance.

Instruction scheduling is a compiler optimization that has a significant impact on the quality of the generated code. This optimization consists in solving an NP-complete problem while taking into account several constraints, imposed by the target processor's architecture. While instruction scheduling for synchronous architectures benefits from a wide literature coverage, scheduling for asynchronous architectures has been less addressed, due to the new constraints imposed by the synchronization mechanisms used by these architectures.

This paper presents the development, implementation and evaluation of a scheduling strategy for the AnARM processor. The scheduling method presented in this thesis uses a dynamic scheduling model based on the spatio-temporal behaviour of the AnARM. This method has been implemented within a modern commercial compiler and evaluated comparatively to usual scheduling methods. The scheduling method presented in this thesis yields performance improvements ranging between 6,22% and 17,48% while preserving the energy asset of the self-timed architecture under study.

Keywords: Instruction scheduling, Self-timed processors, Software optimization, Compilers

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuit
CMOS	Complementary Metal Oxyde Semiconductor
CPSR	Current Program Status Register
DSP	Digital Signal Processor
EU	Execution Unit
FAM	Fully Asynchronous Microprocessor
FIFO	First In First Out
FIQ	Fast Interruption Request
FPGA	Field-Programmable Gate Array
GFD	Graphe de flot de données
ILP	Instruction Level Parallelism
IR	Intermediate Representation
IRQ	Interruption Request
ISA	Instruction Set Architecture
LD	Latence Dynamique
MAP	Micronet Architecture Processor
MIMO	Multiple In Multiple Out
MIPS	Million Instructions Per Second
MMACS	Million Multiply and Accumulate per Second
NSR	Non Synchronous RISC
OFDM	Orthogonal Frequency-Division Multiplexing
PPC	Programmation Par Contraintes

PTD	Penalize True Dependencies
RISC	Reduced Instruction Set Computer
RAW	Read After Write
SIMD	Single Instruction Multiple Data
STRIP	Self Timed RISC Processor
VDD	Vecteur Dynamique des Délais
VECA	Vecteur d'État du Contrôle Asynchrone
VLIW	Very Large Instruction Word
VLSI	Very Large Scale Intergration
VORC	Vecteur d'Occupation des Ressources de Calcul
VSD	Vecteur Statique des Délais
WAR	Write After Read
WAW	Write After Write

INTRODUCTION

Depuis une vingtaine d'années, les applications de l'électronique embarquée se sont diversifiées et touchent désormais plusieurs aspects de notre vie quotidienne. Poussée par une demande accrue ainsi qu'un large flot d'innovations dans différents secteurs, cette industrie évolue selon une cadence très élevée. Les récents développements dans ce domaine suivent un modèle bidirectionnel, selon lequel, la complexité et la structure des applications logicielles (*Software*) déterminent l'orientation des innovations sur le plan matériel (*Hardware*) et réciproquement, les capacités matérielles encadrent les pratiques de programmation. Par exemple, les technologies SIMD (*Single Instruction Multiple Data*) introduites par les processeurs de traitement numérique du signal (DSP) récents permettent de traiter un grand nombre de données en une instruction, ce qui est particulièrement utile dans le cadre de l'implémentation des techniques avancées en communications numériques (Orthogonal Frequency Division Multiplexing, Multiple In Multiple Out, etc.), qui nécessitent des calculs vectoriels et matriciels. D'autre part, l'introduction sur le marché des processeurs à plusieurs unités de calcul (*Very Large Instruction Word*, SUPERSCALAR, etc.) a entrainé une réflexion académique et industrielle sur les techniques d'optimisation logicielles, dont le but est d'exploiter le parallélisme offert par ces architectures pour en améliorer les performances. À cette forte interdépendance s'ajoute l'influence des contraintes de l'environnement d'utilisation. Par exemple, les systèmes utilisés dans le secteur médical ont une obligation stricte de fiabilité. Les systèmes mobiles, quant à eux, opèrent sous une contrainte énergétique ; ils doivent fournir la performance nécessaire tout en ayant une autonomie raisonnable. Ces contraintes contribuent à définir le cahier des charges des composants logiciels et matériels.

Les microprocesseurs ont connu une évolution fulgurante en soutien à la complexité de plus en plus croissante des applications. Le principal moteur de cette évolution est le besoin de performance. Cependant, l'efficacité énergétique prend de plus en plus d'importance dans les

considérations de design. De ce fait, on constate, de nos jours, un regain d'intérêt envers les circuits asynchrones. Ces derniers, au moyen de synchronisation point à point, permettent de s'affranchir du maintien d'un système global de synchronisation (horloge), ce qui leur permet de réduire considérablement la consommation de puissance dynamique. Cependant, à cause de la maturité des processeurs synchrones et de leur forte pénétration dans le marché, la conception des processeurs asynchrones se confronte à une multitude d'obstacles, dont le manque d'outils et d'environnements de conception. De ce fait, la majorité des approches d'implémentation de tels circuits n'a pas franchi le cadre de la recherche académique.

Ce mémoire s'inscrit dans le cadre du projet global AnARM visant l'implémentation et l'exploration multi axiale d'une architecture asynchrone appliquée à un processeur à usage général de type ARM. Le type d'asynchronisme utilisé dans ce projet provient du design endochrone (*self-timed*) des processeurs DSP d'OCTASIC, partenaire majeur du projet. Ce projet de recherche vise plus particulièrement l'exploration des optimisations du compilateur permettant de mettre à profit les spécificités de cette architecture dans le but d'en améliorer la performance et l'efficacité énergétique. Le présent chapitre introduit ce mémoire en présentant son contexte, ses contributions et son plan.

Contexte du projet :

Le projet AnARM est une collaboration entre l'École de Technologie supérieure, l'École Polytechnique de Montréal et la compagnie montréalaise, de conception de processeurs de traitement numérique du signal, OCTASIC. Ce projet réunit une équipe, aux spécialités diversifiées, composée d'étudiants aux cycles supérieurs, de professionnels de recherche et d'ingénieurs travaillant au sein d'OCTASIC. L'objectif de ce projet est de concevoir et d'implémenter un processeur asynchrone supportant le jeu d'instructions ARMv7A. L'implémentation de l'AnARM en tant que telle constitue le premier volet de ce projet, qui devra démontrer les effets bé-

néfiques de l'utilisation de ce type de synchronisme sur la réduction de la consommation de puissance. De plus, cette implémentation devra offrir des performances de calcul compétitives par rapport aux processeurs synchrones de même catégorie. Le projet comprend trois autres volets d'une grande importance : la vérifiabilité, la fiabilité et l'optimisation énergétique. Le volet vérifiabilité se penche sur le développement d'une méthodologie permettant la simulation et la vérification des circuits endochrones. Ces étapes sont d'une grande importance dans le cycle de conception des systèmes VLSI. Cependant, peu d'outils conventionnels permettent la vérification des circuits asynchrones. Par conséquent, il est nécessaire d'adapter certains outils pour permettre la vérification fonctionnelle de l'architecture endochrone. Le volet fiabilité vise à développer une stratégie de test adaptée aux contraintes de l'architecture utilisée. Comme dans le cas de la vérifiabilité, ce volet se confronte à l'inadéquation des outils standard face aux spécificités topologiques de cette architecture. Finalement, le volet d'optimisation énergétique vise à mettre en place des stratégies additionnelles de réduction de la consommation énergétique sur différents niveaux d'abstraction. Le projet de recherche décrit dans ce mémoire, porte sur l'optimisation du compilateur dans le but d'améliorer les performances et l'efficacité énergétique de l'AnARM.

Contribution du mémoire :

Le présent mémoire se base sur des approches existantes d'optimisation du compilateur (optimisations dépendantes de la cible ou *target dependent optimisation*) pour proposer une nouvelle méthode d'ordonnancement des instructions. Cette méthode permet de mettre à profit les spécificités de l'architecture de l'AnARM pour en améliorer les performances et réduire sa consommation de puissance. L'approche proposée se base sur les caractéristiques temporelles de l'AnARM ainsi que son modèle de partage des ressources entre les unités de calcul. L'architecture endochrone, étudiée dans ce mémoire, repose sur une synchronisation point à point en utilisant des délais variables pour chaque opération. De plus, les mécanismes de synchroni-

sation utilisés par cette architecture permettent de définir un ordre d'exécution et d'accès aux ressources partagées. La présence de ces mécanismes offre certains avantages tout en imposant des contraintes au niveau du flot d'exécution. La stratégie d'ordonnancement développée prend en compte ces différentes caractéristiques architecturales de l'AnARM.

En addition à cela, ce mémoire définit un modèle d'estimation temporel et spatial permettant de décrire l'exécution d'un programme sur l'AnARM, en prenant en compte les mécanismes de synchronisation, l'analyse temporelle et le modèle de partage des ressources.

Finalement, une carte des itinéraires d'instructions a été implémentée sur le compilateur utilisé. Cet outil fournit les informations architecturales utiles à la phase d'ordonnancement. Ces informations comprennent les latences des différentes étapes du pipeline pour chaque instruction ainsi que les ressources de traitement disponibles à chaque étape. De plus, ces itinéraires permettent d'intégrer la description architecturale du processeur cible au compilateur utilisé dans le but d'évaluer les performances des différentes méthodes d'ordonnancement utilisées par ce dernier. Cette étape est fondamentale, car elle constitue une première avancée vers l'intégration d'une description complète de l'AnARM au compilateur utilisé (*back-end*).

Plan du mémoire :

Ce mémoire se divise en quatre chapitres. La présente section offre une description des chapitres restants et de la conclusion.

Chapitre 1. Ce chapitre porte sur la description des architectures asynchrones en se concentrant sur les circuits endochrones. Un historique des implémentations notables des processeurs asynchrones sera établi. Les processeurs ARM endochrones seront présentés plus en détail pour introduire l'AnARM. Finalement, une description détaillée du système à l'étude sera fournie. Le but de ce chapitre est de présenter les éléments du design de l'AnARM et leur influence sur

la stratégie d'optimisation adoptée.

Chapitre 2. Ce chapitre introduira les généralités relatives au processus de compilation. Différentes optimisations logicielles seront présentées, l'accent sera mis sur les optimisations d'arrière-plan (*back-end*) et plus particulièrement, l'ordonnancement des instructions. Le problème d'ordonnancement des instructions sera présenté formellement, de plus, l'interdépendance entre cette optimisation et l'allocation des registres sera étudiée. De plus, une revue des méthodes existantes de l'ordonnancement des instructions sera présentée. Finalement, une réflexion sur les contraintes de l'ordonnancement des instructions par rapport aux architectures asynchrones permettra d'introduire les différentes considérations de la stratégie adoptée.

Chapitre 3. Ce chapitre présentera la stratégie d'ordonnancement élaboré dans ce mémoire. Tout d'abord les modèles décrivant le comportement spatio-temporel de l'AnARM seront présentés. Par la suite, l'introduction des notions, relatives à l'ordonnancement basé sur les latences dynamiques, permettra de décrire la base de la stratégie d'ordonnancement proposée. Subséquemment, les différents volets qui composent cette stratégie seront discutés en détails relativement aux spécificités architecturales de l'AnARM. Finalement, une description, pas-à-pas, de l'algorithme d'ordonnancement proposé sera présentée.

Chapitre 4. Ce chapitre exposera l'évaluation expérimentale de l'approche proposée. Les contextes d'implémentation et d'expérimentation seront introduits. Des comparaisons avec les méthodes conventionnelles seront faites et une étude de sensibilité montrera l'influence des différents paramètres en jeu.

Conclusion. Ce chapitre conclura le mémoire en présentant la synthèse des travaux, les limites et les difficultés rencontrées dans le cadre du projet ainsi que les recommandations et les possibilités de travaux futurs.

Les premier et deuxième chapitres du mémoire établissent le contexte de ce travail de recherche en présentant des revues de la littérature portant sur les architectures asynchrones et l'ordonnancement des instructions. Le troisième chapitre expose le travail de développement de l'approche proposée dans ce mémoire. Finalement, le quatrième chapitre est consacré à l'évaluation de l'approche et la présentation du contexte expérimental.

CHAPITRE 1

ARCHITECTURE ASYNCHRONE ET ARM

Un ensemble d'inconvénients, présenté par les processeurs synchrones, a motivé les différentes recherches pourtant sur les architectures asynchrones (Garside & Furber, 2007). Tout d'abord, les systèmes synchrones maintiennent un signal d'horloge qui permet de coordonner les différentes opérations de traitement. Les transitions du signal d'horloge engendrent une dissipation de puissance (puissance dynamique) au niveau des charges capacitives du circuit. La puissance dynamique s'écrit comme telle (Horowitz *et al.*, 1994),

$$P_{dynamique} = C \times V_{DD}^2 \times f \tag{1.1}$$

où C est la capacité de la charge concernée par la transition (transistor, fil, etc.), V_{DD} est la tension d'opération et f représente la fréquence d'opération. La puissance dynamique est, par conséquent, proportionnelle à la fréquence d'opération. Dans le cas des processeurs récents qui opèrent à des hautes fréquences, la puissance dynamique dissipée est ample. De plus, la complexité des processeurs modernes nécessite l'implémentation d'un imposant circuit de distribution d'horloge. Les transitions sur ce signal puissant augmentent aussi l'effet des interférences électromagnétiques. Finalement, la présence de l'horloge contraint l'ensemble des opérations d'une unité d'exécution (ALU) à s'exécuter en une ou plusieurs périodes d'horloge. L'équation 1.2 (Rabaey *et al.*, 2002), exprimant les contraintes électroniques, montre que la période d'horloge est principalement déterminée par la longueur du chemin logique le plus lent (chemin critique). Dans l'équation 1.2, $t_{horloge}$ représente la période d'horloge, $max(t_{p-logique})$ représente le temps de propagation maximal dans le circuit logique, t_{setup} représente le temps pendant lequel les données doivent rester valides avant le déclenchement du registre, $t_{p-registre}$ représente le temps de propagation dans le circuit du registre et t_{skew} représente le décalage maximal entre les temps d'arrivée du signal d'horloge aux différentes composantes du circuit. Par conséquent, les instructions rapides sont pénalisées, car elles doivent attendre, au minimum, une période d'horloge pour produire leurs résultats.

$$t_{horloge} \geq max(t_{p-logique}) + t_{setup} + t_{p-registre} + t_{skew}. \quad (1.2)$$

Les processeurs asynchrones, quant à eux, utilisent des méthodes de synchronisation différentes. On dit qu'ils sont déclenchés par des événements (Garside & Furber, 2007). Ceci signifie qu'ils ne fournissent de la puissance que lorsqu'ils sont sollicités. Ces processeurs ne posent pas les mêmes contraintes que les processeurs synchrones quant aux décalage et gigue d'horloge. Ils ne nécessitent donc pas l'implémentation des circuits de balancement d'horloge, de plus en plus nécessaires dans les implémentations synchrones modernes. En plus de ceci, chaque instruction est exécutée à son rythme et n'est plus limitée par le temps de propagation dans le chemin critique. Les architectures asynchrones présentent, cependant, certains inconvénients. Parmi ceux-ci, on peut citer l'augmentation de la surface d'implémentation due aux ajouts architecturaux permettant ce type de synchronisme ou encore le manque d'outils de design dût à la prédominance des circuits synchrones.

Dans le but de présenter l'architecture de l'AnARM, ce chapitre offre un aperçu général des processeurs ARM. Par la suite, quelques implémentations notables de processeurs asynchrones sont décrites. Finalement, une description détaillée de l'AnARM et de ses spécificités est fournie.

1.1 Processeurs ARM

De nos jours, les processeurs ARM dominent largement le marché de l'électronique embarquée. En 2014, on a dénombré 12 milliards de puces en circulation qui utilisent la propriété intellectuelle ARM (Vijay & Bansode, 2015). L'adoption à grande échelle des processeurs ARM a été favorisée par la simplicité de leur architecture, leurs performances et leur grande efficacité énergétique. Les noyaux ARM se distinguent de la concurrence par leur modèle de mise en marché. Ces noyaux sont vendus sous la forme de propriété intellectuelle aux grands manufacturiers (Samsung, Texas Instrument, etc.). Ces derniers les intègrent avec d'autres périphériques, dans le cadre d'une implémentation spécifique, pour produire des microcontrôleurs,

des processeurs ou des *SoC (System On Chip)*. Ces circuits sont produits à grande échelle, par la suite, dans un laboratoire de semi-conducteurs.

Les noyaux ARM implémentent un jeu d'instructions RISC (*Reduced Instruction Set Computing*) où la plupart des instructions s'exécutent en un cycle d'horloge. Les instructions faisant partie du ARM ISA (*Instruction set architecture*) offrent, pour la plupart, des caractéristiques intéressantes comme les décalages intégrés ou l'exécution conditionnelle. L'architecture RISC impose la manipulation de la mémoire par le biais des registres. Les noyaux ARM permettent des accès multiples à la mémoire à travers les instructions *LDM (Load Multiple)*, *STM (Store Multiple)* et leurs variantes. Ces instructions permettent l'échange simultané de plusieurs valeurs avec la mémoire selon la largeur du bus reliant cette dernière au processeur. Les processeurs ARM possèdent différents modes d'opérations et ces modes se qualifient de "privilégiés" s'ils s'exécutent avec des mécanismes de protection. Les modes privilégiés, comme les modes superviseur et système, permettent l'exécution protégée des processus critiques du système d'exploitation, tandis que les modes *FIQ (Fast Interrupt Request)* et *IRQ (Interrupt Request)* permettent la gestion des interruptions. Le mode utilisateur, qui permet l'exécution de la majorité des applications, est un mode non privilégié. En mode utilisateur, un noyau ARM dispose de 16 registres dont 13, à usage général ainsi qu'un registre de contrôle et d'état CPSR (*Current Program Status Register*). Le tableau 1.1. montre les fonctions des registres. Ces registres de 32 bits peuvent contenir des données 8, 16, ou 32 bits selon la nature de la donnée. Dans le cas d'un registre contenant plusieurs données, ces dernières sont manipulées à l'aide d'un suffixe ajouté à l'instruction.

Le registre CPSR permet d'activer les différents modes d'opération ainsi que certaines extensions architecturales, telles qu'illustrées à la figure 1.1 (Vijay & Bansode, 2015). Ce registre contient surtout les bits d'état qui maintiennent les drapeaux conditionnels d'égalité, de signe, de dépassement et de retenue. Les instructions conditionnelles sont exécutées selon la valeur de ces bits.

Tableau 1.1 Les registres ARM

Registres	**Utilisation en description**
r0-r12	général
r13	pointeur de pile
r14	registre de retour (*link register*)
r15	compteur programme

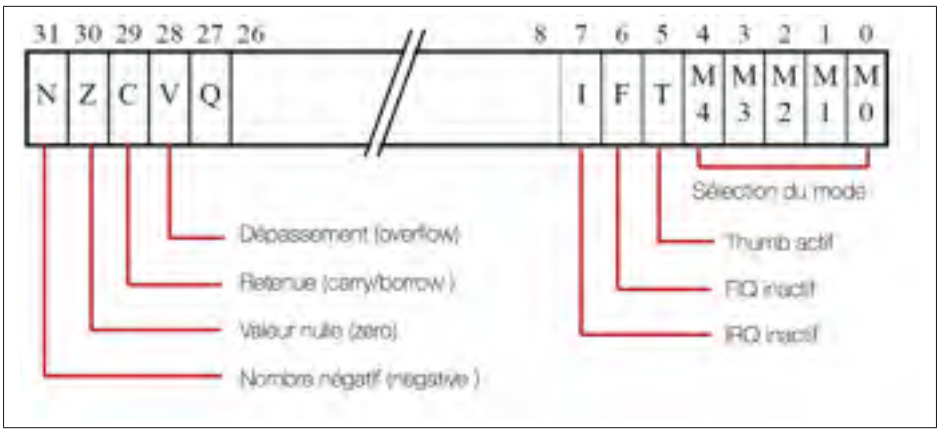


Figure 1.1 Registre CPSR
Tirée de Vijay & Bansode (2015, p. 2)

Finalement, certains processeurs ARM disposent d'extensions architecturales comme *Jazelle, Thumb, Neon* etc. Ces extensions sont résumées au tableau 1.2.

Tableau 1.2 Exemples d'extensions du ARM

Jazelle	Extension permettant l'exécution directe du JAVA Bytecode pour améliorer la performance des applications JAVA
Thumb et Thumb-2	Jeu d'instruction 16 bits permettant des gains en mémoire par rapport au jeu standard (32 bits)
VFP	Unité de calcul point flottant
NEON	Unité de calcul vectoriel (utile pour le traitement des signaux et le traitement graphique)

1.2 Historique des processeurs asynchrones

1.2.1 Premiers designs des processeurs asynchrones

La première implémentation notable d'un processeur asynchrone a été introduite par Martin (1989) dans un projet de l'Institut Caltech (*California Institute of Technology*). Ce processeur dispose d'un bus de données d'une largeur de 16 bits et effectue les différentes opérations de traitement d'un processeur RISC standard. Ces opérations sont les calculs arithmétiques et logiques sur des entiers, les accès à la mémoire (écriture et lecture) et les branchements. La synchronisation des différents éléments est basée sur un établissement de liaison (*Handshake*). Cette approche a démontré la possibilité d'implémenter un processeur fonctionnel ne disposant pas de distribution d'horloge et a permis d'atteindre une performance estimée à 15 MIPS en technologie 2μm.

L'approche, décrite par Cho *et al.* (1992), introduit un processeur 32 bits entièrement asynchrone. Le FAM (*Fully Asynchronous Microprocessor*) implémente une architecture endochrone à 18 instructions utilisant 32 registres. Cette architecture possède un pipeline à quatre étages : acquisition de l'instruction, communication avec la mémoire externe, décodage de l'instruction et finalement l'étage d'exécution (permettant aussi d'écrire le résultat dans le fichier des registres). Un établissement de liaison à 4 phases se fait entre les étages adjacents du pipeline, pour synchroniser le transfert des données entre ces derniers. Ce modèle de communication est illustré à la figure 1.2.

Dans ce modèle, le bloc 1 envoie une requête (R_{in}) au module de communication pour l'informer que les données sont prêtes. Le module de communication déclenche, dès lors, la bascule qui mémorise les données transmises par le bloc 1 à l'aide d'une communication bidirectionnelle (L_{in} et L_{ack}) et notifie le bloc 2 de cette requête avec le signal R_{out}. Ce dernier transmet sa confirmation au bloc 1 par l'entremise du module de communication (A_{in} et A_{out}). Le transfert des données se déclenche et le bloc 1 envoie une requête de terminaison. Dès lors que le bloc 2 reçoit l'ensemble des données, il confirme la terminaison. À la réception de cette dernière

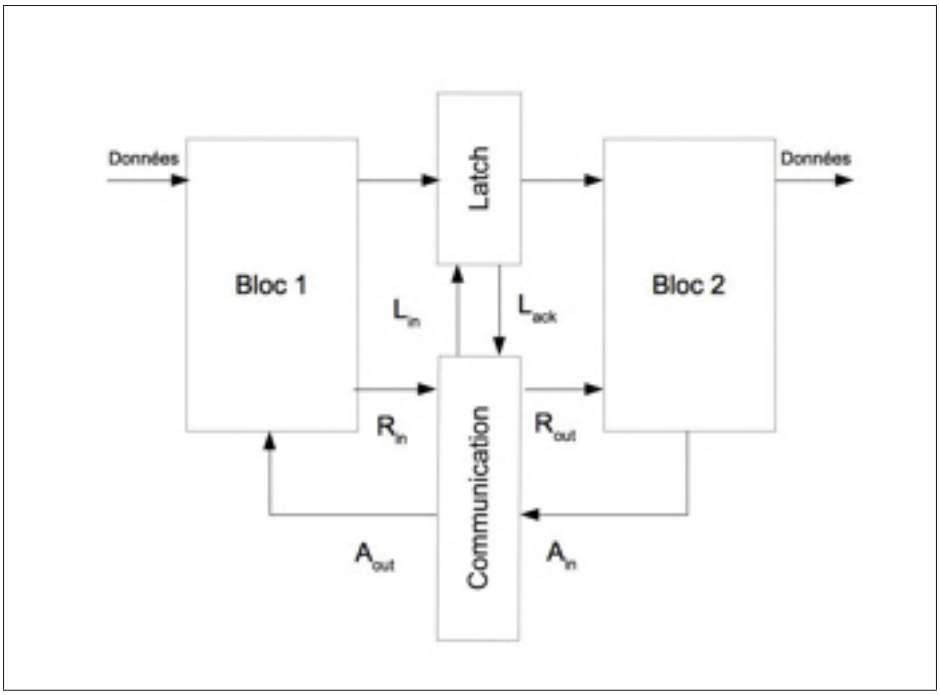


Figure 1.2 Modèle de synchronisation du FAM

confirmation, le bloc 1 met un terme au transfert et commence une nouvelle opération.

Brunvand (1993), chercheur à l'université de l'Utah, a développé un processeur asynchrone dénommé NSR (*Nonsynchronous Risc*). Le NSR est un processeur 16 bits qui implémente 16 instructions comprenant des opérations logiques et arithmétiques, des accès mémoire et des branchements. Ce processeur est un amalgame de plusieurs blocs endochrones qui sont synchronisés grâce à un établissement de liaison à deux phases. Chaque bloc représente une étape du pipeline. L'utilisation de ces blocs dépend de l'instruction, de manière à ce que chaque instruction utilise uniquement les blocs dont elle a besoin. La figure 1.3 (Werner & Akella, 1997), offre un aperçu, à haut niveau, de cette architecture. Le NSR a été implémenté sur FPGA et a permis d'obtenir une performance de 1.3 MIPS (million d'instructions par seconde). Du fait de son implémentation sur FPGA, les performances du NSR sont difficiles à comparer avec celles des autres processeurs, car les FPGA sont plus lents que les circuits ASIC (*Application Specific Integrated Circuit*) (Werner & Akella, 1997).

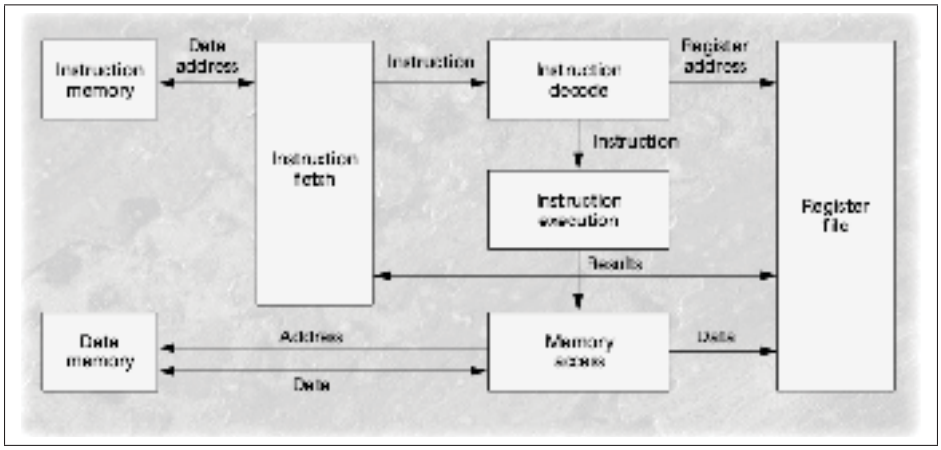


Figure 1.3 Architecture du NSR
Tirée de Werner & Akella (1997, p. 4)

L'architecture STRIP (*A self timed RISC processor*), publiée par Dean (1992), est un exemple de processeur endochrone possédant un système de synchronisation particulier. Alors que la majorité des processeurs asynchrones utilisent un système d'établissement de liaison pour coordonner les tâches de traitement, le STRIP utilise un système d'horloge dynamique. Ce système repose sur des cellules de repérage qui déterminent le temps de propagation critique du chemin logique utilisé pour générer un signal d'horloge ayant une période adéquate. Ce système est présenté à la figure 1.4 (Werner & Akella, 1997). En résumé, cette architecture est synchrone avec une horloge qui s'adapte ce qui permet d'imposer un temps d'exécution selon la charge du traitement requis. Ceci est particulièrement utile dans le cas où une étape de calcul ne nécessite pas d'accès à la mémoire. La période d'horloge dans ce cas-ci sera considérablement réduite, car elle ne prendra pas en compte le chemin logique de la transaction avec la mémoire. La performance moyenne de ce design est de 62.5 MIPS sur une technologie 2-μm CMOS.

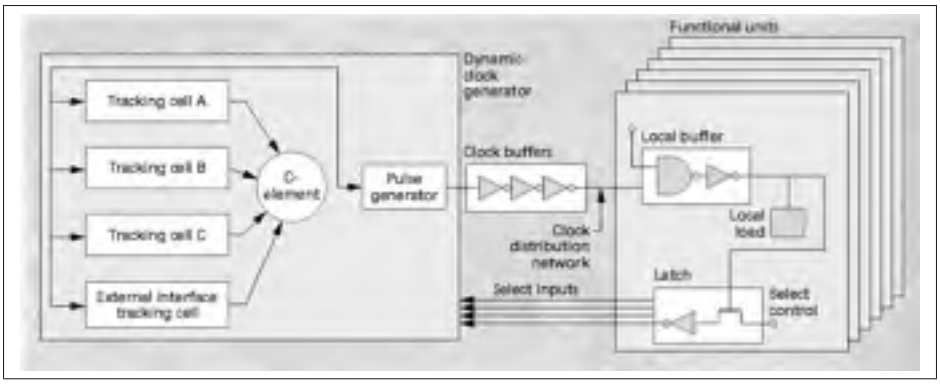


Figure 1.4 Horloge dynamique du STRIP
Tirée de Werner & Akella (1997, p. 6)

1.2.2 ARM asynchrone

Les recherches sur les processeurs asynchrones sont poussées par un certain nombre de motivations. Parmi lesquelles, le design d'un processeur asynchrone commercial capable de rivaliser avec les produits synchrones en termes de performances, consommation de puissance et surface d'implémentation. Étant donné que l'ARM ISA représente le jeu d'instructions le plus supporté par les SoC (*Systems on Chip*) embarqués, des chercheurs se sont, bien évidemment, penchés sur le design d'un processeur asynchrone supportant le ARM ISA.

Suivant cet élan, Furber *et al.* (1994), de l'université de Manchester, ont publié leur implémentation asynchrone d'un processeur ARM. L'AMULET1 est une version asynchrone du ARM6 visant à prouver la faisabilité d'un processeur commercial asynchrone. Ce processeur repose sur une synchronisation de type établissement de liaison à deux phases où les données sont échangées entre deux modules. Tel qu'illustré à la figure 1.5, le bloc émetteur déclenche une transition du signal "requête". Au moment où le receveur déclenche la transition du signal de "confirmation", la transmission se fait. Il est à noter que la transmission est déclenchée par les transitions montantes ou descendantes des signaux "requête" et "confirmation".

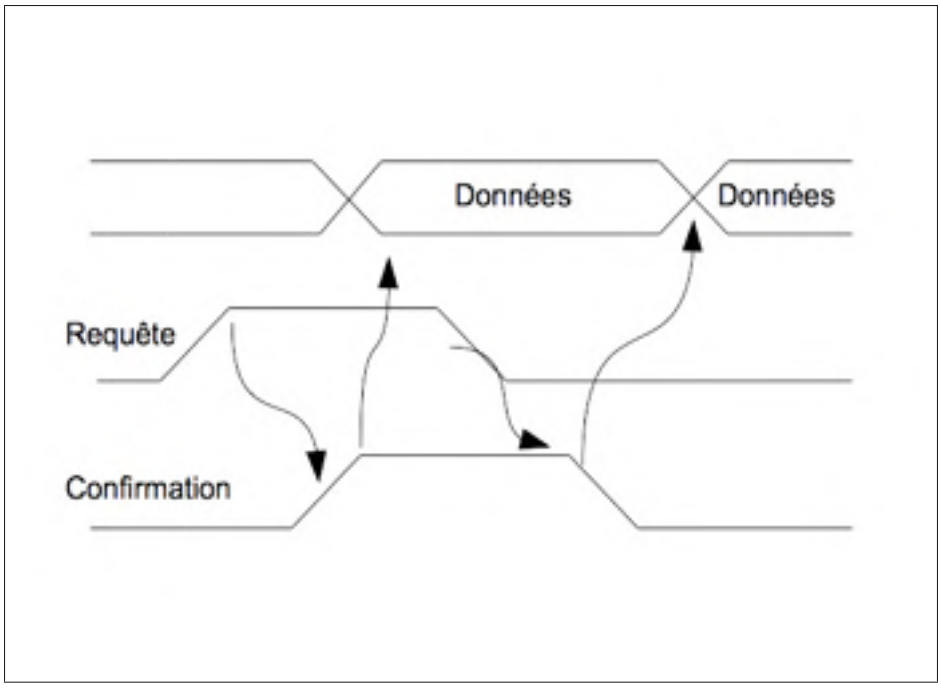


Figure 1.5 Établissement de liaison à deux phases

La figure 1.6 (Furber *et al.*, 1994), illustre l'architecture de l'AMULET1. Ce processeur dispose d'une banque de 30 registres de 32 bits, d'une unité d'exécution, ainsi que deux interfaces ; une pour les données et une pour les adresses. Une particularité intéressante de ce processeur réside dans son mécanisme de verrouillage des registres. À l'étape d'écriture du registre de destination, ce dernier est passé à un tampon *FIFO* (premier arrivé, premier servi). Ce tampon sert à maintenir l'ordre d'écriture des registres afin d'éviter les interférences de données pouvant biaiser l'exécution d'un programme.

L'AMULET1 a prouvé qu'un processeur commercial asynchrone était réalisable. Cependant, à technologie égale, la comparaison avec le ARM6 lui est défavorable sous tous les points de vue, comme le résume le tableau 1.3.

1. Dhrystone est un programme qui permet de tester les performances de calcul sur des entiers (Weicker, 1984)

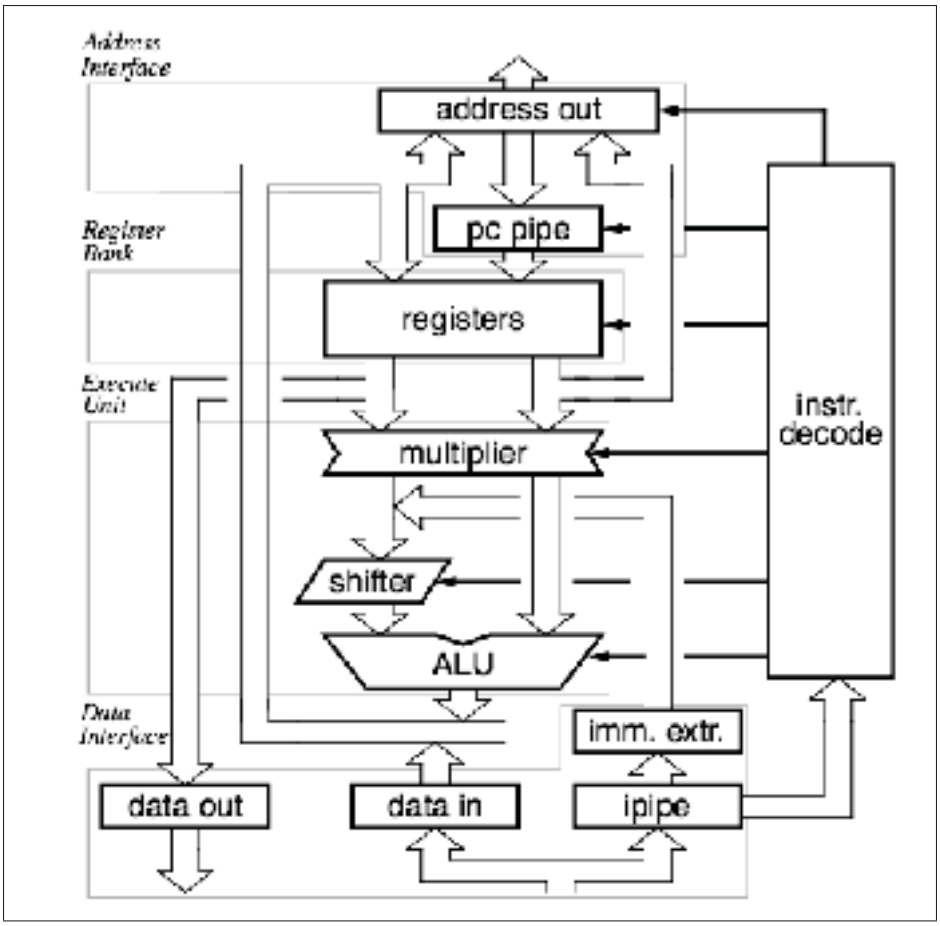


Figure 1.6 Architecture de l'AMULET1
Tirée de Furber *et al.* (1994, p. 2)

Tableau 1.3 AMULET1 vs ARM6

	AMULET1	**ARM6**
Aire	5.5mm x 4.1mm	4.1mm x 2.7mm
Transistors	58,374	33,494
Performance	9K Drystones [1]	15K Drystones
Puissance	83mW	75mW

En 1997, le design de l'AMULET2 apporte plusieurs améliorations au projet de base (Furber *et al.*, 1999). L'adoption d'un système d'établissement de liaison à 4 phases s'est avérée plus efficace que celui à 2 phases en termes de performances et de réduction de la consommation de puissance. De plus, la réduction de la profondeur du pipeline a permis de réduire le

temps d'exécution des instructions. Un système de transfert d'opérandes a permis de mitiger les désavantages du système de blocage des registres qui causait des blocages du pipeline. Ce système permet de transférer le résultat d'une instruction à l'instruction suivante à l'aide d'un multiplexeur. Cette caractéristique est très intéressante, car elle est similaire au système de partage des données de l'AnARM (processeur à l'étude) et elle permet de réduire l'affluence des accès à la banque de registres. Finalement, un dispositif d'analyse des branchements a réduit le nombre d'instructions (*prefetched*) passées de la mémoire programme au décodeur d'instruction après un branchement. Avec ces améliorations, l'AMULET2 parvient à rivaliser en performances avec les ARM synchrones similaires (ARM710 et ARM810). De plus, il montre une réduction significative de la consommation de puissance et des émissions électromagnétiques par rapport à ces derniers.

Une troisième itération du projet AMULET est publiée en 1998 (Furber *et al.*, 1998). Cette implémentation est basée sur la version ARMv4T. Elle intègre le décodage des instructions *Thumb* ainsi qu'un tampon de tri des résultats (voir figure 1.7). Ce processus de gestion des

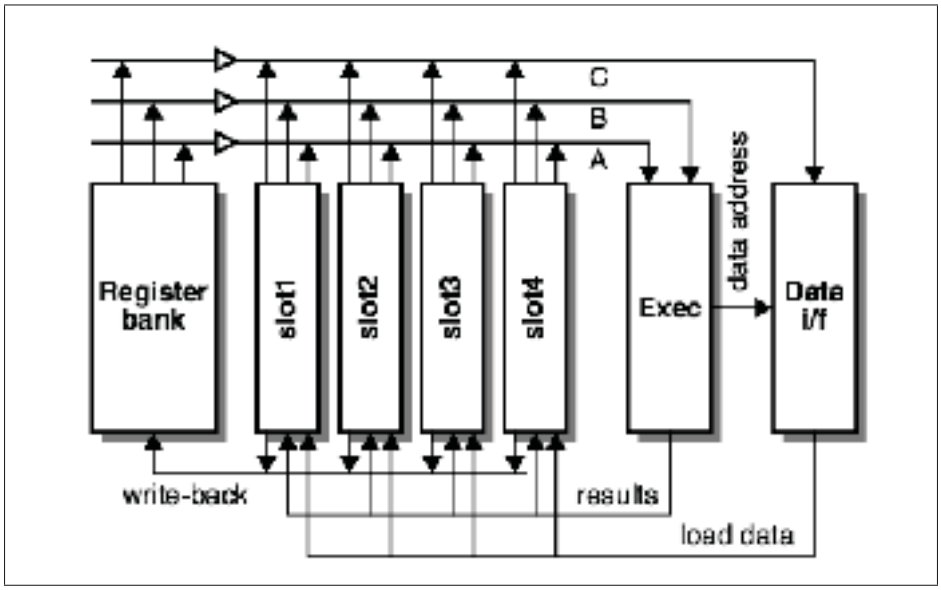


Figure 1.7 Tampon de tri des résultats de l'AMULET3
Tirée de Furber *et al.* (1998, p. 3)

résultats d'instruction diffère du mécanisme de blocage des registres de l'AMULET1 et du multiplexeur de transfert de l'AMULET2. Avec ce nouveau système, les résultats d'une instruction passent par le tampon de tri avant d'être fournis autant qu'opérande aux instructions suivantes ou d'être écrits dans la banque de registres. Ce mécanisme permet d'ordonner l'écriture des résultats dont l'ordre d'émission est imprévisible à cause de l'exécution asynchrone (*out of order*) et fournit, au besoin, les résultats récents à l'unité d'exécution. Les performances prévues pour l'implémentation de l'AMULET3 sont équivalentes à celles du processeur synchrone l'ARM9TDMI, pour une aire d'implémentation similaire (environ $4mm^2$).

Le plus récent design d'ARM endochrone est né d'une collaboration entre les compagnies ARM et *Handshake solutions* (Bink & York, 2007). L'ARM996HS est un processeur destiné pour des applications nécessitant une basse consommation de puissance ainsi que de faibles émissions électromagnétiques (implants médicaux, électronique embarquée, etc.). Ce processeur, basé sur la version ARMv5TE, implémente un système d'établissement de liaison à 4 phases pour synchroniser les différentes phases du pipeline. Ce processeur a présenté plusieurs attraits, comparé aux équivalents synchrones. Ces attraits comprennent, notamment, une moindre consommation de puissance, des émissions électromagnétiques plus faibles et une surface d'implémentation équivalente. Cependant, avec des performances largement en deçà de celles de ses équivalents synchrones, l'ARM996HS n'a pas connu le succès commercial escompté. Parmi les raisons de l'échec de ce processeur, on peut citer l'utilisation de l'établissement de liaison à 4 phase au lieu de 2 phases ce qui s'est traduit en une diminution des performances (Nowick & Singh, 2011).

1.3 Architecture de l'AnARM

Le processeur à l'étude, dans ce mémoire, est une adaptation de l'architecture ARMv7A à un modèle d'opération endochrone. Ce modèle provient de l'expertise d'OCTASIC, entreprise montréalaise spécialisée dans la conception des processeurs dédiés au traitement numérique du signal (DSP). Après avoir connu un bon succès commercial grâce aux produits OCT1010

et OCT2200, OCTASIC a cherché à consolider son avantage concurrentiel dans un secteur où les géants du semi-conducteur imposent un rythme d'innovation soutenu. OCTASIC a donc décidé de proposer les processeurs DSP les moins énergivores du marché. Dans cette optique, la compagnie montréalaise a entrepris une approche empirique dans le but de développer une architecture asynchrone dont l'impact sur l'efficacité énergétique serait considérable.

L'OPUS 2 (Laurence, 2012) est réalisé avec trois considérations principales. Tout d'abord, son design devait se faire avec une bibliothèque d'éléments standards. De plus, seuls les outils standards de design et d'analyse devaient être utilisés. Finalement, l'OPUS2 devait livrer une performance similaire aux DSP synchrones de même génération pour une consommation de puissance considérablement réduite.

L'OPUS2 a été conçu en utilisant un procédé 90nm et a été comparé au TEXAS INSTRUMENT C64xx. Pour une tension d'opération de 1V, l'OPUS2 exécute 2000 MMACS [2] tandis que le noyau du C64xx exécute 4000 MMACS. Cependant l'aire d'implémentation de l'OPUS2 est 3,5 fois moindre et sa consommation de puissance est considérablement plus basse ; l'OPUS2 livre 21 MMACS par mW, le C64xx ne livre que 7 MMACS par mW.

Ces résultats prometteurs sont la principale motivation du projet AnARM. Ce projet cherche à exploiter l'architecture endochrone de l'OPUS2 pour concevoir un processeur plus complexe supportant le jeu d'instruction ARMv7A. De ce fait, la coordination des opérations et des modules se fait à la manière de l'OPUS2. Cette section, par conséquent, présente les éléments spécifiques à ce type de synchronisme ainsi que l'ensemble des éléments de design pertinents permettant au lecteur de comprendre la démarche d'optimisation logicielle adoptée.

2. Millions de "multiplie puis accumule" par seconde (*multiply and accumulate*)

1.3.1 Présentation générale de l'AnARM

L'AnARM, est un processeur asynchrone doté de 16 unités d'exécution (ALU). Ces dernières effectuent l'ensemble des traitements requis par les instructions ARMv7A ce qui comprend les instructions arithmétiques et logiques sur des entiers, les accès mémoires, les sauts de programme, les interruptions, les décalages intégrés et l'exécution conditionnelle. La première implémentation de l'AnARM se présente comme une preuve de concept, par conséquent, les différentes extensions ARM ne sont pas supportées par le prototype. Il n'existe donc aucun support pour le calcul vectoriel, le calcul en point flottant et les instructions *Thumb*.

Les 16 ALUs sont indépendantes et permettent d'effectuer l'ensemble des étapes nécessaires à l'exécution d'une instruction. Ces étapes sont :

a. Acquisition de l'instruction ;
b. Lecture des registres ;
c. Exécution ;
d. Prédiction de branchement ;
e. Accès à la mémoire ;
f. Écriture des registres.

Les ALUs sont agencées selon le schéma montré à la figure 1.8. Ces dernières sont connectées entre elles et aux autres éléments du noyau par le biais d'un *Crossbar* qui joue un rôle central dans cette architecture. Ce dernier permet le partage des opérandes entre les ALUs. Par exemple, si le résultat d'une instruction est produit par l'ALU 1 et que ce résultat est sollicité par une instruction décodée par l'ALU 3, le *Crossbar* est commandé par une machine à états pour connecter la sortie de l'ALU 1 à l'entrée de l'ALU 3. Par ce procédé, l'ALU 3 fait l'acquisition d'un opérande sans passer par la banque des registres. De plus, le *Crossbar* fait office d'interface entre les ALUs et les autres éléments du noyau (banque des registres, unité de gestion de la mémoire, unité d'acquisition et de décodage d'instruction etc.).

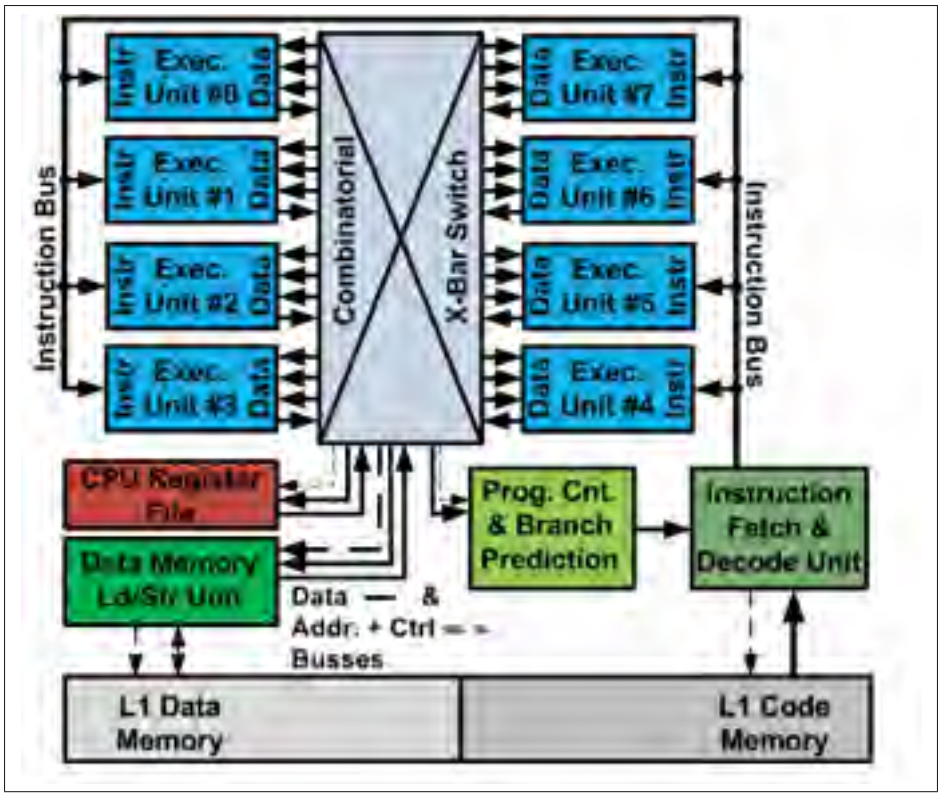


Figure 1.8 Noyau de l'AnARM
Tirée de Laurence (2013, p. 32)

Finalement, le fonctionnement de l'AnARM, repose sur une synchronisation point à point grâce à une ligne de délai variable ainsi qu'un système de jetons permettant la coordination des différentes ALUs et le maintien de la validité des données. Ces deux éléments seront présentés dans les deux sous-sections suivantes.

1.3.2 Synchronisation intra modulaire

Les unités d'exécution de l'AnARM, opèrent d'une façon asynchrone grâce à un système de délai "sur mesure". Ce système permet à chaque instruction de s'exécuter selon le temps qu'elle requiert, ce qui permet d'éviter les contraintes du chemin critique présentes dans les processeurs synchrones. L'ALU, illustrée à la figure 1.9, dispose d'un étage d'entrée composé de 4 registres d'état. Ces registres contiennent le code de l'instruction ainsi que les 3 opérandes.

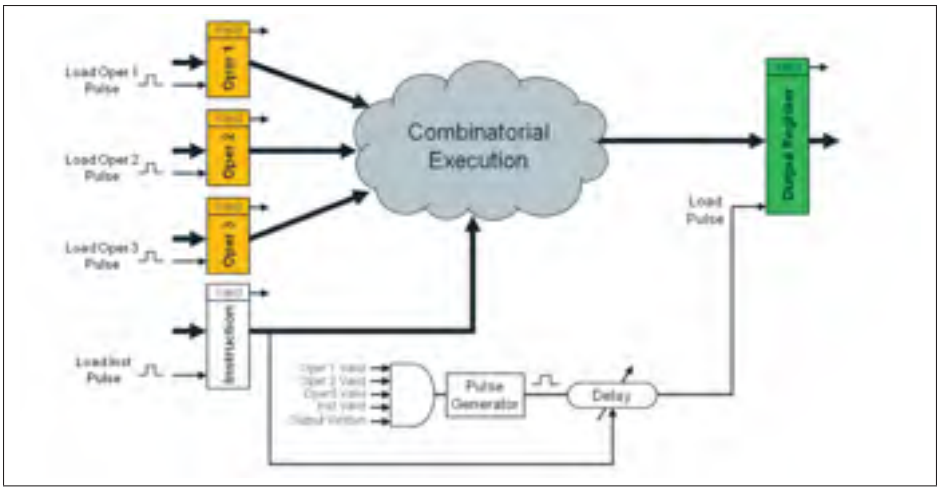


Figure 1.9 Aperçu d'une ALU de l'AnARM
Tirée de Laurence (2013, p. 21)

L'étage central est un nuage logique qui effectue une opération sélectionnée par le vecteur instruction (addition, multiplication, etc.). L'étage de sortie comprend le registre d'état qui contient le résultat de l'instruction. Le système de synchronisation intra modulaire est composé d'un générateur d'impulsion, de différents circuits de délai et d'un système de contrôle. À la réception des opérandes et de l'instruction par l'ALU, des signaux indiquant la validité de ces éléments sont envoyés au système de synchronisation. Ce dernier génère une impulsion et choisit le circuit de délai correspondant à l'instruction en cours de traitement. Passé le délai choisi, l'impulsion arrive au registre de sortie pour le déclencher. Il est à noter que pendant que l'impulsion traverse le circuit de délai, le nuage logique effectue l'opération demandée et transmet le résultat à entrée du registre de sortie. De cette manière, le résultat est toujours valide malgré l'acquisition asynchrone des opérandes. De plus les circuits de délai sont implémentés de manière à respecter l'ensemble des contraintes temporelles.

1.3.3 Synchronisation inter modulaire

Le mode de synchronisation intra modulaire de l'AnARM, est très avantageux, car il permet de s'affranchir de la distribution d'horloge. Cependant, l'exécution asynchrone nécessite une grande coordination entre les ALUs pour assurer la validité du flux d'exécution et des données.

Pour cela, l'AnARM, dispose d'un système de communication inter module qui permet d'ordonner les opérations des ALUs et d'éviter les conflits au niveau des accès aux ressources communes. La synchronisation inter modules repose sur un mécanisme de jetons correspondants aux différentes étapes d'exécution du pipeline. Ces jetons sont des ressources de synchronisation exclusives dont la possession permet à une ALU d'effectuer la tâche correspondante. Au départ ou après réinitialisation du pipeline, c'est la première ALU qui détient l'ensemble des jetons. Cette dernière effectue l'opération correspondante à un jeton puis le transmet à l'ALU suivante selon une topologie annulaire (figure 1.10) (Laurence, 2012). Si l'ALU qui détient un jeton ne requiert pas le traitement qui y est associé, elle le passe sans délai logique à l'ALU suivante. Grâce à ce modèle de jetons, l'intégrité des données est préservée. Par exemple, si deux instructions successives mettent leurs résultats dans le même registre, c'est toujours la première instruction qui le fera en premier. De cette manière, l'ordre d'opération reste valide malgré l'exécution asynchrone des instructions. La figure 1.11 donne un aperçu à haut niveau de l'agencement des ALUs au sein de la topologie annulaire.

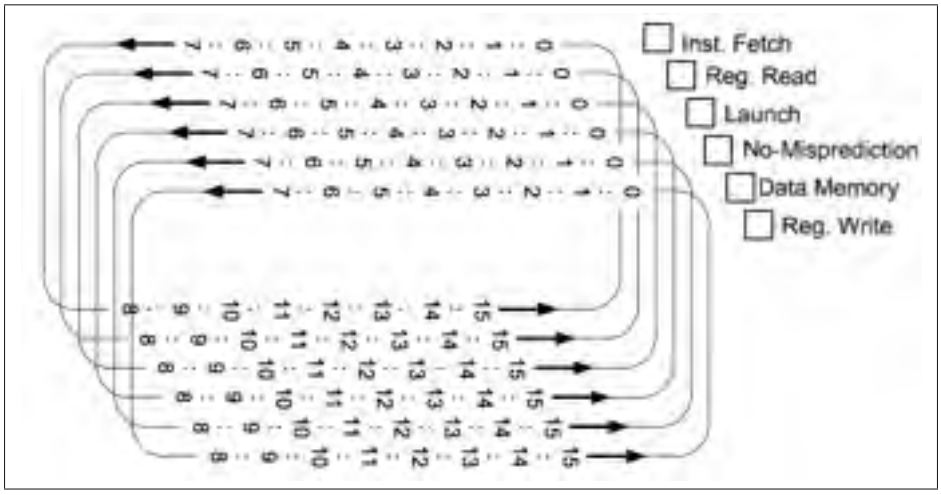


Figure 1.10 Transmission des jetons
Tirée de Laurence (2012, p. 3)

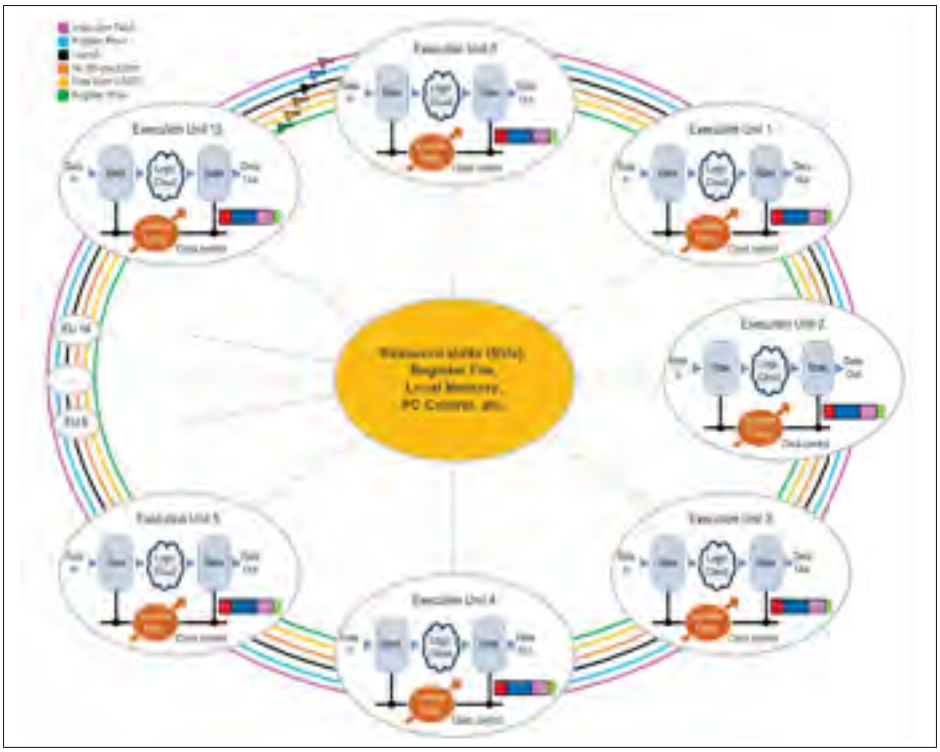


Figure 1.11 Aperçu haut niveau de la topologie annulaire de l'AnARM
Tirée de Laurence (2013, p. 34)

L'exclusivité des jetons entraine une dépendance au niveau des différentes étapes du pipeline à l'exception de l'étape de l'exécution. Cette étape est effectuée de manière indépendante par chaque ALU ce qui se traduit par un parallélisme partiel au niveau du flux de traitement. Ce comportement, illustré à la figure 1.12, diffère du comportement des architectures VLIW synchrones. Ces dernières intègrent des ALUs complètement parallèles et indépendantes où chaque étape de traitement requiert un délai prédéfini. À la figure 1.12, la largeur des colonnes est variable pour refléter les variations des délais du traitement pour chaque instruction.

1.4 Conclusion

Malgré leur domination du marché, les processeurs synchrones présentent des désavantages engendrés par l'imposante distribution d'horloge qu'ils intègrent. Les principaux désavantages

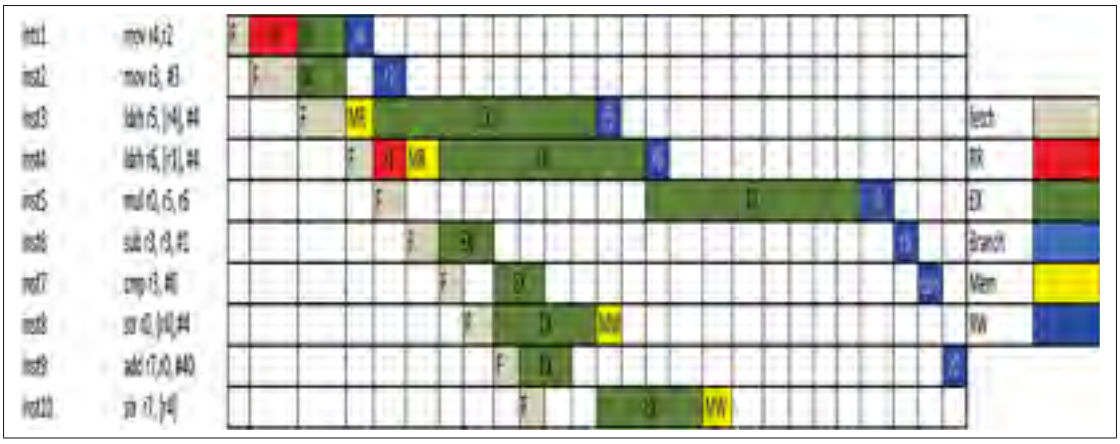


Figure 1.12 Exemple simplifié du pipeline pour une séquence d'instructions

sont la surconsommation de puissance en mode repos, les interférences électromagnétiques et les circuits additionnels de balancement. Ces désavantages ont motivé les recherches sur les circuits asynchrones et de nombreuses tentatives, plus ou moins fructueuses, d'implémentation de processeurs asynchrones ont vu le jour. Au fil du temps, les recherches sur les processeurs asynchrones ont su démontrer les attraits de ce type d'architecture, mais se sont souvent confrontées au manque d'outils de design, de test et de mesure de performance. Par conséquent, les premières approches ont eu vocation de prouver la faisabilité de tels designs. Par la suite, des approches visant des architectures plus complexes (ARM) se sont développées. Parmi celles-ci, l'architecture endochrone d'OCTASIC, a montré un gain substantiel en termes d'efficacité énergétique et de surface d'implémentation.

Cette architecture est basée sur un système de délai variable pour cadencer l'exécution des différentes instructions, ainsi qu'un système de jetons pour coordonner les différentes ressources de calcul. La présence du *Crossbar* offre un avantage considérable quant à l'accès aux opérandes.

L'approche d'optimisation logicielle décrite dans ce mémoire se base sur ces spécificités pour améliorer les performances du AnARM. Avant de présenter cette approche, il est nécessaire de

fournir au lecteur un contexte théorique et un recensement des travaux portants sur la compilation et plus particulièrement, sur l'ordonnancement des instructions.

CHAPITRE 2

ORDONNANCEMENT DES INSTRUCTIONS

La compilation logicielle est une série de transformations et d'optimisations permettant de produire, à partir d'un programme écrit en langage de haut niveau (code source), un fichier assembleur, objet ou exécutable pour une machine cible (code machine). Le processus de compilation se compose de trois étapes principales, l'avant-plan (*front-end*), le langage intermédiaire et l'arrière-plan (*back-end*).

Au niveau de l'avant-plan, le compilateur effectue, tout d'abord, une analyse lexicale du code source. Cette étape consiste à partitionner le code source pour générer des unités de langage sous la forme de jetons lexicaux (*tokens*). Les jetons lexicaux sont des unités atomiques qui représentent certains éléments du code comme les identifiants, les symboles et les opérateurs. Par la suite, le compilateur effectue une analyse syntaxique de la séquence de jetons résultante. Cette étape se base sur un ensemble de règles grammaticales pour transformer la séquence linéaire de jetons en arbre hiérarchique (*parse tree*) représentant la structure syntaxique du programme. L'analyse sémantique est la phase finale du *front-end*. Dans le cadre de celle-ci, le compilateur ajoute l'information sémantique à l'arbre syntaxique et effectue des vérifications de cohérence sur l'utilisation des types, identifiants, valeurs et opérateurs. À la suite de cette analyse, le compilateur rejette les codes sources contenant des erreurs et émet, éventuellement, des avertissements.

Suite à la phase d'avant-plan, le compilateur génère à partir du code source un code intermédiaire. Ce code intermédiaire est une représentation, interne du compilateur, indépendante des langages source et machine. Grâce à cette représentation, le compilateur peut générer du code machine pour une cible donnée à partir de différents langages sources (C, *java*, etc.) et de produire, à partir d'un langage source donné, des codes machines pour différents processeurs cibles (X86, ARM, etc.). La principale fonction de la représentation intermédiaire est d'offrir au compilateur un cadre simple et standardisé pour appliquer un certain nombre d'op-

timisations, dans le but d'améliorer la qualité du code généré. Les optimisations effectuées, à ce niveau, sont indépendantes du processeur cible (*Target-Independent*), car elles ne prennent pas en compte ses spécificités architecturales. Quelques exemples de ce type d'optimisations sont décrits au tableau 2.1.

Tableau 2.1 Exemples d'optimisations indépendantes de la cible

Optimisation	**Description**
Simplification algébrique	Utilisation (par exemple) des décalages binaires à la place des multiplications par les puissances de 2
Élimination du code mort	Suppression des instructions qui ne sont jamais atteintes durant le cycle d'exécution du programme
Pré-calcul des constantes	Calcul des résultats d'opération sur les constantes pendant la compilation pour alléger le programme
Propagation de copie	Élimination des redondances dans les définitions des variables

À la suite des optimisations sur le code intermédiaire, le compilateur entame la phase d'arrière-plan, aussi appelée *code generation*. Cette phase consiste à transformer la représentation intermédiaire en langage machine et se compose de trois étapes principales : la sélection des instructions, l'ordonnancement des instructions et l'allocation des registres. Durant cette phase, le compilateur se réfère à la description du processeur cible pour, dans un premier temps, traduire chaque instruction du code intermédiaire en une ou plusieurs instructions-machine. Par la suite, le compilateur assigne à chaque registre virtuel, utilisé dans le code intermédiaire, un registre physique de la banque des registres du processeur cible. Finalement, le code machine résultant est ordonnancé, selon une heuristique donnée, pour réduire le temps d'exécution du programme en tirant profit des spécificités architecturales (parallélisme, *forwarding*, etc.).

L'ordonnancement des instructions et l'allocation des registres sont d'une grande importance dans la génération du code, car ces deux optimisations permettent une gestion efficace des ressources (unités d'exécution et registres) et impactent, d'une manière significative, la qualité du code (vitesse d'exécution et consommation d'énergie).

Étant donné que l'objectif de ce mémoire est de développer une méthode d'ordonnancement permettant de tirer profit du comportement spatio-temporel de l'AnARM, l'accent sera mis sur l'étude des notions théoriques ainsi que les méthodes existantes portant sur l'ordonnancement des instructions. L'allocation des registres sera présentée d'une manière sommaire, relatant l'interdépendance entre cette optimisation et l'ordonnancement des instructions.

2.1 Cadre théorique

Dans cette section, un cadre théorique portant sur l'ordonnancement des instructions et l'allocation des registres sera établi. L'objectif principal sera de fournir au lecteur une définition formelle de ces deux problèmes d'optimisation, ainsi qu'une description des contraintes et des considérations en jeu.

2.1.1 Ordonnancement des instructions

L'ordonnancement des instructions est une étape clé du processus de compilation, car elle permet de réarranger le code machine dans le but d'en améliorer la qualité. Cependant, cette étape se confronte à certaines contraintes, dont les plus importantes sont les dépendances de données et la disponibilité des ressources. Les dépendances de données dictent la priorité de certaines instructions par rapport à d'autres, pour que l'exécution du programme aboutisse à un résultat valide. La disponibilité des ressources indique le nombre d'unités de calcul libres supportant le type des instructions à exécuter.

2.1.1.1 Dépendances de données

Les dépendances de données sont des relations entre les instructions du programme qui utilisent les mêmes opérandes. Ces dépendances doivent être respectées pour que le résultat du programme soit valide. Il existe trois types de dépendances de données :

Vraies dépendances : Une vraie dépendance, aussi dite RAW (*Read after write*), existe entre deux instructions si la deuxième instruction utilise (en lecture) une donnée résultante de la pre-

mière instruction. Ce type de dépendance doit être respecté en tout temps. Par conséquent si la deuxième instruction est placée avant la première, le résultat du programme sera invalide.

Fausses dépendances : Une fausse dépendance, aussi dite WAR (*Write after read*), existe entre deux instructions, si la deuxième instruction écrit un opérande (registre ou case mémoire) qui est lu par la première instruction. La deuxième instruction, dans ce cas-ci, doit attendre que la précédente lise la donnée qu'elle requiert à partir du registre ou de la case mémoire concernée, avant d'écrire son résultat dedans. Comme dans le cas des vraies dépendances, l'ordre des instructions doit être respecté. Cependant, cette dépendance peut être éliminée, sans biaiser le résultat du programme, en utilisant un opérande de sortie différent dans la deuxième instruction.

Dépendances de sortie : La dépendance de sortie, aussi appelée WAW (*Write after write*), existe entre deux instructions qui écrivent dans le même registre ou case mémoire. La dépendance de sortie et les fausses dépendances ont des effets similaires. Elles imposent l'ordre des instructions liées par la dépendance, mais peuvent être évitées par l'utilisation d'un opérande différent.

Le tableau 2.2 contient un exemple pour chaque type de dépendance :

Tableau 2.2 Exemples des dépendances de données

Dépendance	RAW	WAR	WAW
Exemple	i1 : ADD R3 R2 R1 i2 : MOV R4 R3	i1 : ADD R3 R2 R1 i2 : MOV R1 R5	i1 : ADD R3 R2 R1 i2 : MOV R3 R5

2.1.1.2 Disponibilité des ressources

Chaque processeur dispose d'une ou de plusieurs unités d'exécution qui effectuent le traitement dicté par les instructions. Les unités d'exécution peuvent supporter l'ensemble du jeu d'instruction du processeur ou uniquement un certain type d'instructions. L'AnARM, par exemple,

dispose de 16 unités d'exécution. Tandis que toutes les unités d'exécution de l'AnARM supportent l'arithmétique simple, seules les unités paires supportent les opérations sur la mémoire et seules les unités impaires supportent les multiplications. La disponibilité des unités d'exécution affecte la durée d'exécution d'un programme. Par exemple, si toutes les unités d'exécution supportant les multiplications sont occupées et que la prochaine instruction à s'exécuter est une multiplication, un blocage du pipeline surviendra et durera jusqu'à ce qu'une unité de ce type se libère. Il est donc très important de prendre en compte la disponibilité des ressources pendant la phase d'ordonnancement pour éviter, autant que possible, ces blocages.

2.1.1.3 Graphe de flot de données

Pendant la compilation, les programmes et les fonctions sont décomposés en blocs de base. Les blocs de base sont des séquences acycliques d'instructions possédant un seul point d'entrée (étiquette) et un seul point de sortie (étiquette ou branchement).

À la suite de cette décomposition, le compilateur effectue une analyse des dépendances pour modéliser chaque bloc de base sous la forme d'un graphe de flot de données (GFD). Le graphe résultant est un graphe orienté et acyclique qui se compose de noeuds et de branches. Les noeuds représentent les instructions du bloc et les branches relient les noeuds qui sont liés par une dépendance de données. La figure 2.1 (Srikant & Shankar, 2007), illustre un exemple de GFD pour un bloc de base. Dans cet exemple, une branche découle de l'instruction i1 et se dirige vers l'instruction i3. On dit que i3 est un successeur immédiat de i1 et qu'inversement, i1 est un prédécesseur immédiat de i3. Si un chemin relie indirectement deux instructions (en passant par un autre noeud), comme dans le cas de i2 et i9, le noeud supérieur est dit prédécesseur du noeud inférieur et le noeud inférieur est dit successeur du noeud inférieur.

Une valeur temporelle est assignée à chaque branche. Cette valeur représente la latence de l'instruction à partir de laquelle la branche découle. Par conséquent, les successeurs immédiats de ce noeud ne peuvent s'exécuter qu'à partir de l'achèvement de son exécution. Le chemin

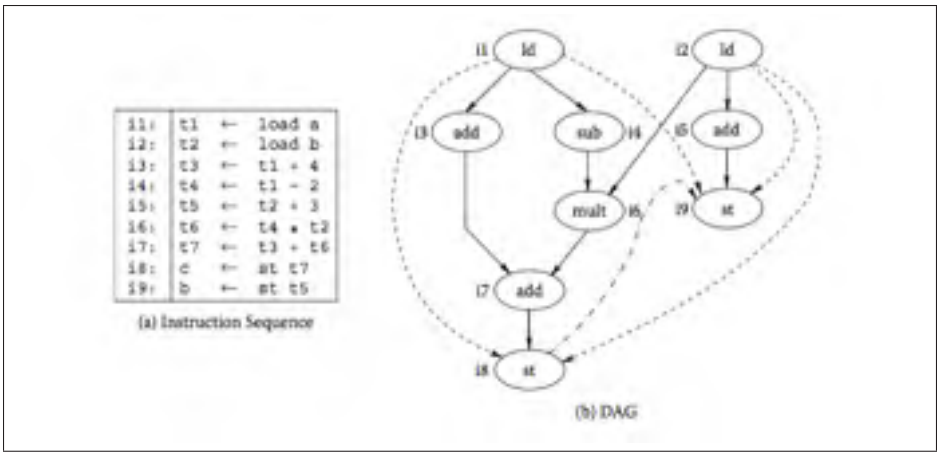


Figure 2.1 Exemple de GFD
Tirée de Srikant & Shankar (2007, p. 19-6)

critique est défini comme étant le chemin de latence le plus long entre un noeud ne possédant aucun prédécesseur et un noeud ne possédant aucun successeur.

2.1.1.4 Définition du problème d'ordonnancement des instructions

L'ordonnancement des instructions est un problème d'optimisation qui consiste à trouver un ordre d'instructions possédant un coût minimal. Ce problème fait l'objet de plusieurs contraintes qui restreignent le nombre de solutions possibles.

Considérons un graphe de flot de données représentant une séquence d'instructions $S = \{I_1, I_2, ..., I_k\}$ où k est le nombre d'instructions de la séquence. Chaque branche reliant une instruction I_n, et son successeur immédiat I_m (avec $n, m \leq k$) est représentée par $l\ (I_n, I_m)$. Cette valeur, représentant le poids de la branche, correspond à la latence de l'instruction I_n. Supposant que l'architecture cible possède un ensemble de p unités fonctionnelles. Chaque unité fonctionnelle est de type $t \in T = \{T_1, T_2, ..., T_l\}$ avec $l \leq p$. Soit $f\ (I_i)$ le type de l'instruction I_i, il existe un sous-ensemble $T' \subseteq T$ qui supporte le type d'instruction $f\ (I_i)$ quel que soit $i \leq k$. Soit $\prec$ une relation d'ordre irréflexive exprimant les contraintes des dépendances de données telles que $I_n \prec I_m$ signifie que I_n est un prédécesseur immédiat de I_m. Les contraintes de l'ordonnancement des instructions se définissent comme telles :

Contrainte de la largeur d'exécution : Le nombre d'instructions exécutées simultanément est inférieur ou égal au nombre d'unités d'exécution dont dispose le processeur.

Contrainte de la disponibilité des ressources : Le nombre d'instructions, supportées par $T' \subseteq T$ et exécutées simultanément, est inférieur ou égale à la taille du sous-ensemble T'.

Contrainte des dépendances : Soit I_n et I_m deux instructions de S telles que $I_n \prec I_m$. On note $t(I_n)$ et $t(I_m)$ pour designer, respectivement, les temps de fin d'exécution de I_n et I_m. La contrainte des dépendances s'écrit comme suit : $t(I_m) \geq t(I_n) + l\ (I_n, I_m)$

L'ordonnancement des instructions vise à minimiser une métrique de performance en respectant ces contraintes. Dans la majorité des cas, la fonction coût à minimiser est la durée d'exécution de la séquence. Cette métrique est exprimée par le temps de fin d'exécution de la dernière instruction de la séquence $t\ (I_k)$. Dans d'autres cas, l'ordonnancement vise à minimiser le nombre de blocages du pipeline d'exécution pour améliorer l'efficacité de calcul du processeur. D'autres méthodes d'ordonnancements cherchent à minimiser l'énergie consommée par l'exécution du programme. Finalement, il est important de préciser que le problème d'ordonnancement des instructions est un problème NP-complet (Division *et al.*, 1987).

2.1.2 Allocation des registres

Le langage intermédiaire, produit par le compilateur, utilise une banque de registres virtuels pour représenter les différentes variables du programme. Ces registres sont infinis, par conséquent, seules les vraies dépendances sont représentées à cette étape-ci de la compilation. Pour générer le code machine, le compilateur doit assigner les registres virtuels aux registres physiques de la banque des registres du processeur. L'allocation des registres consiste donc à remplacer les registres virtuels infinis par des registres physiques, en nombres finis, en respectant certaines contraintes et conventions. Cette sous-section présente les différentes notions pertinentes à l'allocation des registres ainsi la relation existante entre l'allocation des registres et l'ordonnancement des instructions.

2.1.2.1 Intervalle de vie d'une variable et graphe d'interférences

Chaque variable du programme possède une durée de vie. Cette durée de vie est la séquence d'instructions entre la définition de la variable et sa dernière utilisation. La figure 2.2 (Srikant & Shankar, 2007) illustre une séquence d'instructions et la durée de vie des différentes variables de la séquence.

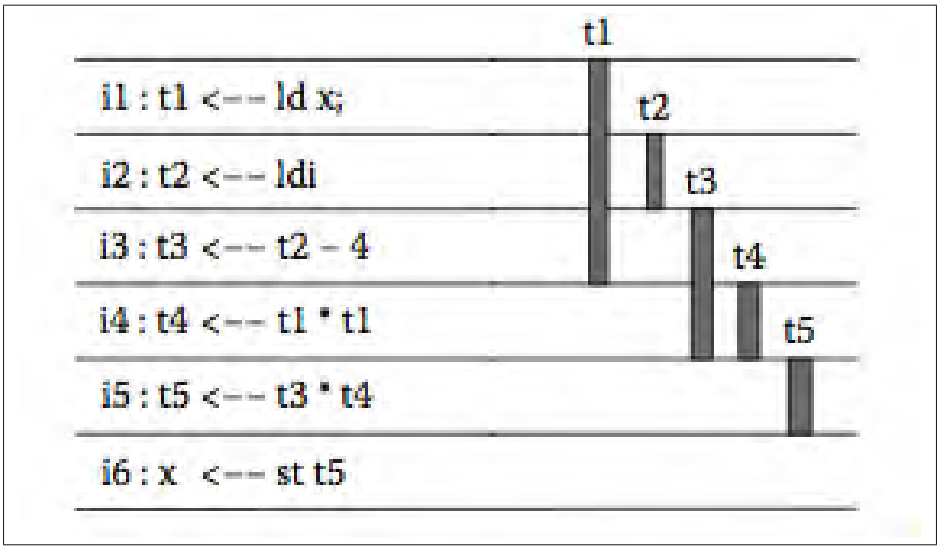


Figure 2.2 Intervalle de vie des variables
d'une séquence d'instruction
Tirée de Srikant & Shankar (2007, p. 19-4)

Dans cet exemple, la variable t_1 possède un intervalle de vie allant de l'instruction $i1$ à l'instruction $i3$. L'intervalle de vie de la variable t_2 est limité à l'instruction $i2$. Si deux variables coexistent à un moment donné du programme on dit qu'il y a interférence entre ces deux variables. Ceci est le cas des variables t_1 et t_2 de l'exemple illustré à la figure 2.2. À partir des intervalles de vie des variables, le compilateur construit un graphe d'interférences dans lequel les variables sont représentées par des noeuds. Ces noeuds sont reliés entre eux en cas d'interférence. Le graphe d'interférence pour l'exemple précédent est illustré à la figure 2.3.

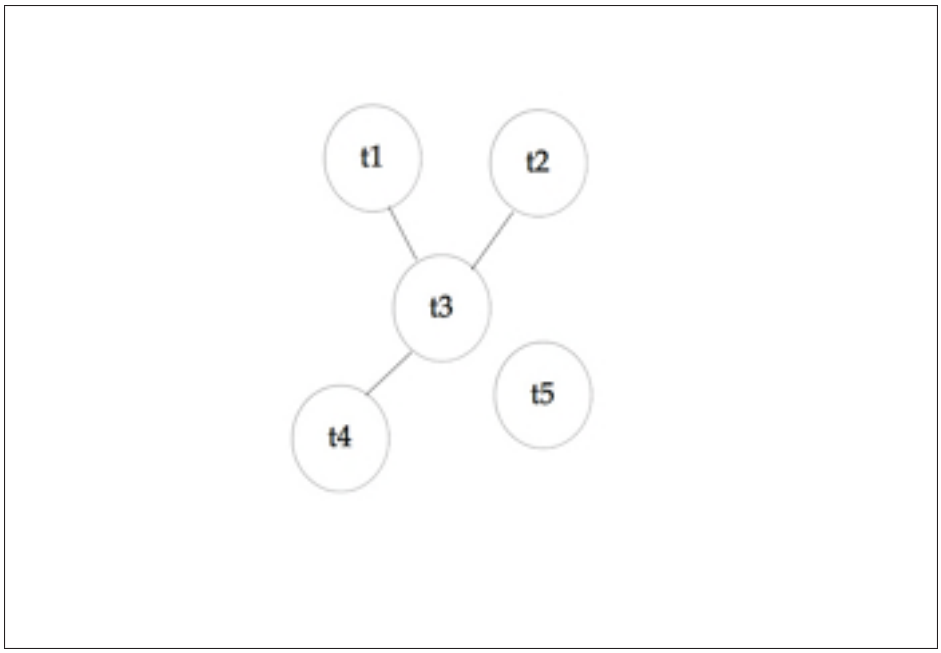


Figure 2.3 Graphe d'interférences

À partir du graphe d'interférence, le compilateur assigne les registres physiques aux variables à l'aide d'une méthode algorithmique. Les deux algorithmes les plus utilisés pour résoudre l'allocation des registres sont la coloration de graphe (Chaitin, 2004) et la programmation quadratique booléenne partitionnée (Scholz & Eckstein, 2002).

2.1.2.2 Relation entre l'ordonnancement des instructions et l'allocation des registres

Le problème d'allocation des registres est un problème NP-complet. Le nombre limité des registres physiques entraine des cas d'interférences des variables que l'algorithme d'allocation ne peut résoudre sans faire appel à des ressources externes. Dans le cas où le même registre doit être assigné à deux variables qui coexistent pendant une séquence du problème, l'algorithme d'allocation des registres procède au *spilling* d'une des variables. Ce procédé consiste à mettre en mémoire le contenu d'un registre le temps de l'utiliser pour achever le calcul sur une autre variable. Du point de vue de la performance, ce procédé est coûteux, car il entraîne l'ajout d'instructions mémoire (*LOAD, STORE*) coûteuses en temps d'exécution. L'ordonnancement des instructions peut subvenir avant (*pre-pass*) ou après (*post-pass*) l'allocation des registres.

Dans le cas du *pre-pass*, l'algorithme d'ordonnancement dispose de plus de flexibilité, car les registres virtuels sont infinis et seules les vraies dépendances contraignent le processus. Cependant, un ordonnancement agressif cause une plus grande pression sur les registres, car les variables qui interfèrent sont plus nombreuses. L'ordonnancement en *post-pass* est beaucoup plus limité, en raison de la présence des fausses dépendances et des dépendances de sortie. En contraste, ce type d'ordonnancement n'a aucun effet sur l'utilisation des registres. Ce problème est connu sous le nom d'ordonnancement de phase (*phase ordering*).

2.2 Techniques existantes d'ordonnancement des instructions

L'ordonnancement des instructions pour les architectures synchrones a été amplement couvert par la littérature scientifique. L'évolution des techniques d'ordonnancement proposées a été étroitement liée aux avancées dans le domaine des architectures des microprocesseurs. L'objectif principal des recherches, dans ce domaine, a été d'optimiser le flot d'exécution des instructions en tirant profit des spécificités architecturales des processeurs modernes.

La majorité des recherches sur l'ordonnancement des instructions s'est focalisée sur l'ordonnancement local. Cette optimisation consiste à ordonnancer chaque bloc de base séparément sans considérer le flot de contrôle du programme (branchements, boucles, etc.). L'ordonnancement global est une technique qui, à partir des probabilités des branchements, effectue des transferts d'instructions entre les blocs de base pour créer des séquences acycliques d'instructions plus amples, avant de les ordonnancer. Les techniques d'ordonnancement utilisées, lorsque les séquences globales d'instructions sont formées, sont similaires aux techniques d'ordonnancement local. La seule différence entre les deux modèles d'ordonnancement est la fonction coût qui, dans le cas de l'ordonnancement global, prend en compte les probabilités de branchement des blocs de base qui constituent la séquence (Srikant & Shankar, 2007). Par conséquent, les méthodes d'ordonnancement local s'appliquent tout autant à l'ordonnancement global. Étant donné que l'objectif du mémoire consiste à ordonnancer les instructions en tirant profit du comportement spatio-temporel de l'AnARM, seules les méthodes d'ordonnancement

local sont présentées. Il est à noter que l'impact de la technique développée dans ce mémoire est évalué aux niveaux local et global, grâce aux méthodes d'ordonnancement global disponibles dans le compilateur utilisé à un haut niveau d'optimisation (-O3).

L'ordonnancement des instructions pour les architectures asynchrones a suscité moins d'intérêt de la part la communauté scientifique. Les architectures asynchrones se caractérisent par des comportements temporels et spatiaux très différents des plateformes synchrones. Ces comportements rendent le processus d'ordonnancement plus difficile en raison des incertitudes liées aux latences d'instructions et de l'utilisation des ressources. Cette section offre un aperçu des techniques d'ordonnancement les plus utilisées dans les compilateurs modernes.

2.2.1 Ordonnancement de liste

L'ordonnancement de liste est une méthode heuristique qui cherche les instructions prêtes à ordonnancer et définit leur priorité selon certains critères. Cette méthode a été popularisée grâce à l'approche publiée par Gibbons & Muchnick (1986). Cette approche, de complexité $\theta(n^2)$, a été largement adoptée dans les compilateurs modernes. L'ordonnancement de liste consiste à dresser une liste d'instructions prêtes à chaque cycle d'horloge. Les instructions prêtes sont comparées entre elles selon un critère donné. La meilleure instruction selon ce critère est choisie pour être émise au cycle présent. L'algorithme met à jour, par la suite, la liste des instructions prêtes en libérant les instructions, dont les prédécesseurs ont été émis. L'algorithme 2.1 décrit en détail l'algorithme d'ordonnancement de liste.

Choisir la meilleure instruction parmi les instructions prêtes est l'étape clé de l'ordonnancement de liste. Il existe un grand nombre de critères proposés dans la littérature. Un recensement des critères existants est publié dans (Smotherman *et al.*, 1991). Une description des critères les plus utilisés est présentée ci-dessous.

L'approche de *Gibbons et al.* : Cette approche a été conçue dans l'objectif de réduire les blocages du pipeline. Dans le contexte des premiers processeurs RISC pipelinés, l'important délai

Algorithme 2.1 Ordonnancement de liste

Input : Bloc de base non ordonnancé, critère C, dépendances de données, ensemble d'unités fonctionnelles T, latences
Output : File d'exécution

```
1  Cycle = 0 ;
2  Liste prête = instructions initialement indépendantes;
3  while Liste prête est non vide do
4  |   Trier(Liste prête) selon C dans un ordre décroissant;
5  |   Mettre la première instruction de la liste I_C prête dans la file d'exécution;
6  |   Supprimer I_C de la liste des prédécesseurs de ses successeurs;
7  |   Supprimer I_C de la liste prête;
8  |   Ajouter à Liste prête les instructions dont tous les prédécesseurs ont été émis;
9  |   Cycle = Cycle +1;
10 end
11 return File d'exécution
```

nécessaire aux instructions (*LOAD*) pour charger les données à partir de la mémoire causait des interruptions du flot d'exécution. L'approche de *Gibbons et al.* ordonnance les instructions selon deux critères. Le premier critère consiste à prioriser l'instruction qui ne dépend pas de la dernière instruction émise. Le second critère consiste à prioriser l'instruction la plus susceptible d'engendrer des blocages du pipeline (nombre de successeurs, latence).

Chemin critique : Le critère du chemin critique est l'approche d'ordonnancement la plus utilisée dans les compilateurs modernes (GCC, LLVM), en raison des bons résultats qu'elle génère. L'approche consiste à calculer la marge d'ordonnancement de chaque instruction. Cette marge correspond à la différence entre le début tardif et hâtif de chaque instruction. Une instruction qui fait partie du chemin critique possède une marge nulle, par conséquent l'approche du chemin critique lui assigne une priorité maximale.

Hauteur des instructions : La hauteur d'une instruction est définie comme étant la distance maximale (en latence) entre l'instruction concernée et une instruction de sortie (une instruction ne possédant aucun successeur). Ce critère privilégie les instructions ayant la plus grande hauteur.

Nombre de successeurs : Ce critère priorise les instructions avec le plus grand nombre de successeurs. Privilégier de telles instructions permet de rendre disponible, le plutôt possible, leurs résultats à leurs nombreux successeurs.

Type d'instruction : Pour ce critère, le développeur de l'optimisation assigne des poids de priorité pour chaque catégorie d'instructions, selon l'architecture cible, les latences, et les ressources de calculs disponibles.

La recherche publiée par Beaty *et al.* (1996) utilise l'algorithme génétique pour choisir le meilleur critère d'ordonnancement de liste pour une situation donnée. Les résultats de cette recherche montrent que le critère du chemin critique est le meilleur dans la grande majorité des cas étudiés. L'étude publiée par Chase (2006) compare la qualité des échéanciers, produits par l'ordonnancement de liste, aux échéanciers optimaux. Les résultats de cette étude montrent que l'ordonnancement de liste produit des échéanciers optimaux dans 94% des cas, dans le cadre de l'ordonnancement local. Par conséquent, l'ordonnancement de liste est une méthode viable qui produit des bons résultats dans un délai polynomial.

2.2.2 Méthodes combinatoires

Les méthodes combinatoires sont des méthodes formelles et complètes. Ces méthodes expriment le problème d'ordonnancement sous une forme mathématique où les contraintes sont exprimées par des inégalités de latences. Des variables entières ou booléennes représentent les décisions qui forment une solution. Ces méthodes explorent la totalité de l'espace des solutions pour générer la solution optimale. Les méthodes combinatoires solutionnent le problème d'ordonnancement dans un délai exponentiel du fait de la NP-complétude du problème d'ordonnancement. Les deux méthodes combinatoires, les plus fréquemment appliquées au problème d'ordonnancement des instructions, sont la programmation linéaire et la programmation par contraintes.

Programmation linéaire : Cette méthode exprime le problème d'ordonnancement en problème de programmation linéaire. Soit σ_i le cycle d'horloge d'émission de l'instruction i. Soit

$d(i,j)$ la latence en nombre de cycles entre l'instruction i et son successeur j. La contrainte de dépendance s'exprime comme telle,

$$\sigma_j \geq \sigma_i + d(i,j). \tag{2.1}$$

Dans le but de simplifier ce modèle, on assume que le processeur cible dispose de T unités fonctionnelles capables d'exécuter l'ensemble des instructions. Soit D la longueur maximale de l'échéancier (somme des latences de toutes les instructions de la séquence). Finalement, on définit la variable booléenne $k_{i,t}$ telle que,

$$k_{i,t} = \begin{cases} 1, \text{ si i est émise au cycle t} \\ 0, \text{ sinon} \end{cases}. \tag{2.2}$$

Par conséquent, le cycle d'exécution de l'instruction i, σ_i, s'exprime en fonction de $k_{i,t}$,

$$\sigma_i = \sum_{t=0}^{D-1} k_{i,t} * t. \tag{2.3}$$

L'équation 3.4 généralise l'équation 3.3 pour une séquence de n instructions,

$$\begin{bmatrix} \sigma_0 \\ \sigma_1 \\ \vdots \\ \sigma_{n-1} \end{bmatrix} = \begin{bmatrix} k_{0,0} & k_{0,1} & \cdots & k_{0,D-1} \\ k_{1,0} & k_{1,1} & \cdots & k_{1,D-1} \\ \vdots & \vdots & \vdots & \vdots \\ k_{n-1,0} & k_{n-1,1} & \cdots & k_{n-1,D-1} \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ \vdots \\ D-1 \end{bmatrix}. \tag{2.4}$$

L'équation suivante exprime la contrainte de la largeur d'exécution,

$$\forall t < D, \sum_{i=0}^{n-1} k_{i,t} \leq T. \tag{2.5}$$

Cette équation signifie que le nombre d'instructions, émises simultanément, est inférieur au nombre d'unités d'exécution du processeur.

La recherche de la solution optimale se fait par séparation et évaluation (*branch and bound*). L'algorithme qui résout le problème évalue toutes les solutions respectant les contraintes du problème. Ces solutions sont des instances de la matrice des variables $k_{i,t}$. Les solutions sont évaluées selon une fonction objective déterminée. Dans la majorité des cas, cette fonction est la durée totale de l'échéancier représenté par le cycle d'exécution de la dernière instruction émise, auquel on additionne la latence de cette instruction. La solution, pour laquelle la fonction coût est minimale, est choisie. La première approche, appliquant la programmation linéaire à l'ordonnancement des instructions pour des processeurs VLIW, a été introduite par Leupers & Marwedel (1997). Cette approche utilise les inégalités exprimant les contraintes pour produire des échéanciers valides. Les auteurs appliquent leur approche pour ordonnancer optimalement des blocs de base d'une taille maximale de 45 instructions, sur les DSP TMS320C25 de *Texas Instruments* et DSP56k de *Motorola*. La méthode, publiée par Wilken *et al.* (2000), cible des problèmes plus larges (1000 instructions). Cette méthode utilise des transformations du GFD pour simplifier les contraintes des dépendances. Ces procédés permettent de diviser le problème d'ordonnancement en plusieurs sous problèmes sans compromettre l'optimalité de la solution. Additionnellement, les auteurs procèdent à éliminer les dépendances redondantes. L'approche de *Wilken et al.* est évaluée en simulant l'exécution du set de *benchmarks* SPEC95 sur un processeur HP PA-8500 à une unité d'exécution. Les auteurs concluent que la programmation linéaire peut ordonnancer optimalement des blocs de base de jusqu'à 1000 instructions dans un délai raisonnable.

Programmation par contraintes : La programmation par contraintes (PPC) est une généralisation de la programmation linéaire. La différence entre les deux méthodes réside dans la nature des variables de décision. Dans le cas de la programmation par contraintes, les variables utilisées sont des nombres entiers représentant les cycles d'émissions des instructions. La résolution du problème de la programmation par contraintes se fait aussi par séparation et évolution, de plus, les solveurs modernes propagent les contraintes dans le but de définir des contraintes globales permettant de réduire l'espace de recherche des solutions. Ertl & Krall (1991) introduisent la première approche PPC appliquée à l'ordonnancement local des instruc-

tions. Ils évaluent leur approche sur le processeur à une unité d'exécution *Motorolla 88100* en optimisant différents programmes (Dhrystone, FFT, etc.). Comparés aux échéanciers produits par GCC à un niveau au niveau d'optimisation -O0, leurs échéanciers sont plus courts pour 28% des blocs de base testés. L'accélération maximale est de 1,75 pour les blocs améliorés. Le temps de traitement requis de cette approche augmente significativement pour les blocs dépassant les 20 instructions. Par conséquent, leur approche est peu pratique pour des problèmes plus larges. Van Beek & Wilken (2001) publient une approche PPC utilisant l'élimination des contraintes redondantes et la subdivision des problèmes en sous-problèmes. Leur approche est 22 fois plus rapide que l'approche basée sur la programmation linéaire proposée par Wilken *et al.* (2000) quand elle est évaluée dans les mêmes conditions (processeur à une unité d'exécution, SPEC95). En 2008, Malik *et al.* (2008) proposent une méthode PPC pour des processeurs à plusieurs unités d'exécution. Cette méthode utilise les techniques introduites par les approches précédentes tout en y ajoutant les contraintes de disponibilité des ressources. Leur méthode est évaluée à l'aide du set de *benchmarks* SPEC2000 sur un processeur IBM PowerPC avec une largeur d'exécution allant de 1 à 6. Les résultats sont comparés à un ordonnancement de liste dont le critère principal est le chemin critique. Dans le cas d'une unité d'exécution, l'approche de Malik *et al.* (2008) génère une réduction moyenne de la longueur de l'échéancier de 4,4%. Cette réduction est de 4,7% pour 6 unités d'exécutions. Il est à noter que la réduction de la longueur de l'échéancier décroit considérablement avec l'augmentation de la taille des blocs de base. Par exemple, dans le cas d'une largeur d'exécution de 6, la réduction moyenne de la longueur de l'échéancier est de 12% pour le bloc de 6 à 10 instructions. Cette réduction décroit jusqu'à atteindre 0,6% pour les blocs de base de 251 à 2600 instructions. Pour une largeur d'exécution de 6 unités, le nombre de blocs pour lesquels leur méthode trouve un échéancier amélioré est de 2147 sur les 183112 blocs (1,2%) testés.

2.2.3 Ordonnancement pour architectures asynchrones

Les techniques d'ordonnancement pour les architectures synchrones bénéficient d'un simple modèle de latences découlant de l'utilisation de l'horloge. Les latences des instructions s'ex-

priment en cycles d'horloge et la fréquence des événements est précise. De plus, le flot d'exécution des instructions dans les pipelines synchrones est linéaire. Chaque instruction suit successivement les étapes de traitements et les interruptions dues aux dépendances de données et de ressources causent des blocages du pipeline pour la totalité de la durée d'exécution de l'instruction.

Le modèle d'ordonnancement des instructions pour les processeurs asynchrones est plus complexe du fait des mécanismes du contrôle asynchrone et des délais variables. Les mécanismes de contrôle induisent des dépendances au niveau des différentes étapes du pipeline. Subséquemment, le traitement des instructions est non linéaire, car chaque étape du pipeline est déclenchée par la disponibilité des données et des ressources de synchronisation. Par exemple, une unité d'exécution ayant lu l'instruction courante doit attendre le signal approprié avant de lire les opérandes et d'effectuer le calcul requis. L'ordre de propagation des signaux du contrôle asynchrone durant l'exécution d'un programme affecte le délai requis pour traiter chaque instruction. En conséquence, la latence de chaque instruction dépend de sa position dans le programme, des entrées et sorties et de l'état courant du contrôle asynchrone. Les figures 2.4 et 2.5 illustrent, respectivement, le modèle d'ordonnancement pour les architectures synchrones et le modèle d'ordonnancement pour les architectures asynchrones. Dans ces deux modèles, S représente la séquence d'instructions à ordonnancer, G représente le graphe de dépendances, T représente la quantité et la nature des ressources de calculs, L représente l'information de latence des instructions et E représente l'échéancier produit.

Dans le cas des architectures synchrones, l'ensemble des données d'ordonnancent est statique et peut être facilement évalué avant la phase d'ordonnancement. Les latences des instructions et par conséquent l'expression des contraintes de dépendances sont invariantes, ce qui permet de construire et d'évaluer statiquement et précisément les différentes solutions du problème. En contraste, l'information concernant les latences d'instructions ne peut être évaluée statiquement dans le cas des architectures asynchrones. Cette information dépend de la nature de l'instruction, de l'état de propagation de la signalisation asynchrone et des délais internes qui

ne sont pas connus au moment de la compilation. L'incertitude entourant ces informations rend la tâche d'ordonnancement plus complexe.

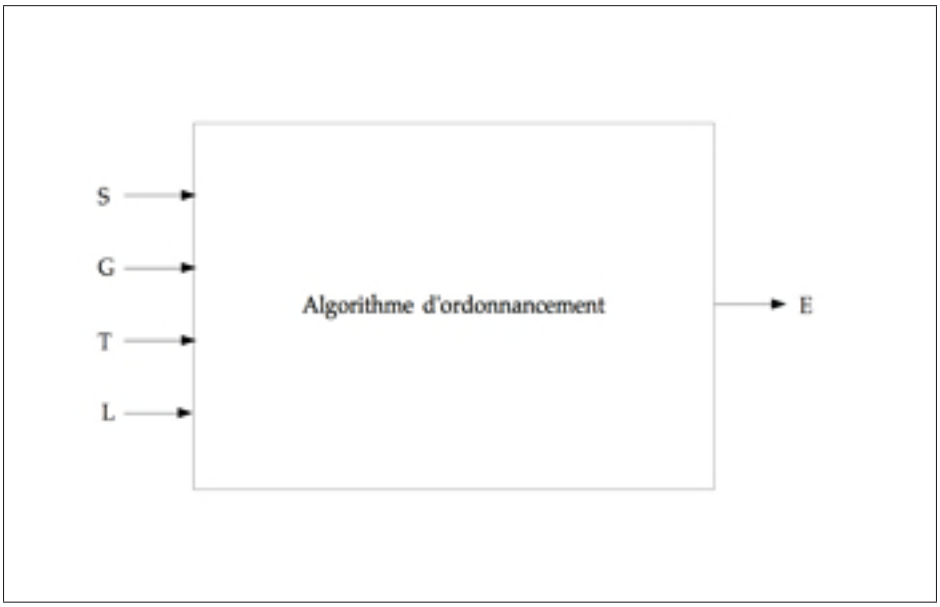


Figure 2.4 Modèle d'ordonnancement pour les architectures synchrones

Sotelo-Salazar (2003) publie son approche d'ordonnancement (*PTD Scheduler*) pour l'architecture MAP (*Micronet Asynchronous Processor*). L'architecture MAP est un amalgame de plusieurs unités d'exécution qui opèrent de manière asynchrone et concurrentielle sans contrôle centralisé (Arvind *et al.*, 1995). Leur approche vise le contournement des heuristiques d'ordonnancement basées sur les latences d'instruction à cause des incertitudes dont elles font l'objet. Les auteurs définissent une métrique d'ordonnancement sous forme de pénalité appliquée aux différentes dépendances de données et de ressources. Le tableau 2.3 donne les différentes pénalités associées aux dépendances. La somme de toutes les pénalités de la séquence d'instructions à ordonnancer est considérée comme une métrique globale, que leur algorithme cherche à minimiser. Les auteurs montrent la corrélation entre l'ampleur de la pénalité globale et la longueur de l'échéancier. L'algorithme de Sotelo-Salazar (2003) procède en deux étapes. Durant la

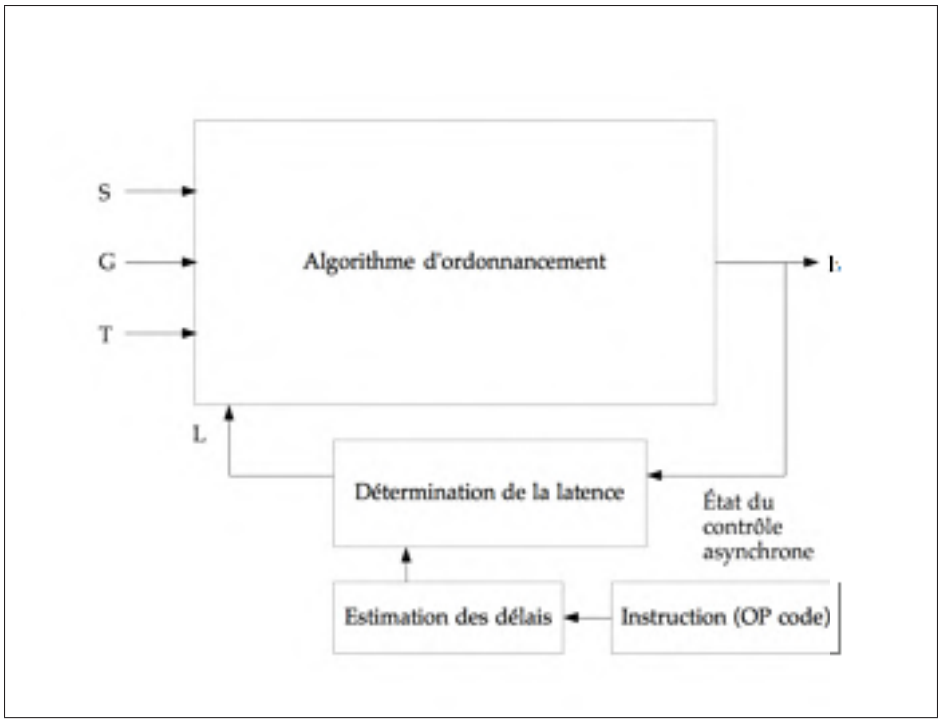


Figure 2.5 Modèle d'ordonnancement pour les architectures asynchrones

première étape, l'algorithme parcourt la séquence d'instructions en comparant les instructions successives. L'algorithme déplace les instructions, si possible, de manière à ce que le nombre d'instructions successives de même type ne dépasse pas le nombre d'unités fonctionnelles de ce type. À la deuxième phase, l'algorithme reparcourt la séquence en déplaçant, si possible, les instructions dans le but minimiser la pénalité globale. Les auteurs évaluent leur méthode à l'aide du compilateur SUIF et d'un simulateur de l'architecture MAP permettant différentes configurations des unités d'exécution. Pour une configuration à 4 unités d'exécution et en utilisant l'ordonnancement global, l'échéancier produit par le (*PTD Scheduler*) est, en moyenne, 2,6% plus court que celui produit par l'approche de Gibbons & Muchnick (1986). Pour la même configuration en ordonnancement local, la réduction moyenne du temps d'exécution est d'environ 2%. Le *PTD Scheduler* est d'une complexité égale à $\theta(n^2)$ dans le pire des cas.

Tableau 2.3 Pénalités associées aux différentes dépendances

Type de dépendance	Instructions consécutives	Instructions séparées de 1
RAW avec un LOAD	3	1
RAW avec un BRANCH	2	0
Dépendance de ressource LOAD	1	0
Autres RAW	1	0

2.3 Conclusion

Le processus de compilation permet la transformation et l'optimisation d'un code source dans le but de générer un code machine pouvant s'exécuter sur le processeur cible. La phase d'arrière-plan d'un compilateur utilise une description logicielle du processeur cible lui permettant de modeler le code selon les spécificités architecturales du processeur (nombre de registres, set d'instructions, unités de calculs, etc.). L'ordonnancement des instructions et l'allocation des registres constituent les étapes qui ont le plus d'impact sur la qualité du code machine généré. Ces deux optimisations visent la résolution de deux problèmes interdépendants pour lesquels, trouver une solution optimale est exponentiellement coûteux (NP-completude).

L'ordonnancement des instructions consiste à trier les instructions d'une séquence du programme pour réduire l'impact des dépendances de ressources et de données sur la vitesse d'exécution et d'augmenter le parallélisme au niveau des instructions (ILP). Les techniques d'ordonnancement pour les processeurs synchrones ont atteint une grande maturité grâce à une couverture généreuse de la littérature. Ces techniques bénéficient d'une modélisation statique et précise permettant la génération et l'évaluation de solutions performantes. Tandis que les approches combinatoires aboutissent aux solutions optimales, les approches heuristiques sont presque aussi performantes et ne nécessitent qu'un faible temps pour traiter les problèmes d'ordonnancement.

En conséquence, l'approche d'ordonnancement de liste est, à ce jour, la méthode la plus utilisée dans les compilateurs majeurs de l'industrie (GCC, LLVM, etc.).

L'ordonnancement des instructions pour les architectures asynchrones se confronte à plusieurs difficultés découlant du comportement de ces processeurs. Le modèle d'ordonnancement pour les processeurs asynchrones fait l'objet de plusieurs incertitudes liées à la signalisation asynchrone et aux délais variables. De plus, ce problème manque sévèrement de couverture de la part de la littérature en raison de l'impopularité des processeurs asynchrones.

Le chapitre suivant offrira une description du développement et de l'implémentation de la méthode d'ordonnancement proposée dans ce mémoire. Cette méthode vise l'optimisation des performances de l'AnARM en modélisant son comportement et en tirant profit de ses spécificités architecturales.

CHAPITRE 3

STRATÉGIE D'ORDONNANCEMENT POUR L'ANARM

L'objectif de ce chapitre est de procurer une description de la méthode d'ordonnancement développée dans ce mémoire. Cette méthode se base une description comportementale de l'architecture endochrone de l'AnARM pour estimer, dynamiquement, les délais des instructions. La modélisation des mécanismes de synchronisation des unités d'exécution de l'AnARM permet la prise en compte des dépendances dues à ces mécanismes. Contrairement à la méthode d'ordonnancement pour architecture asynchrone de Sotelo-Salazar (2003) décrite dans le chapitre précédent, notre méthode d'ordonnancement utilise des critères de décision basée sur les délais des instructions.

Le choix d'algorithme d'ordonnancement s'est porté sur l'ordonnancement de liste, en raison de son efficacité et de sa faible complexité. De plus, étant donnée la grande complexité des méthodes combinatoires pour un modèle d'ordonnancement simple (synchrone), il est encore plus difficile d'adapter ces méthodes à un comportement asynchrone en raison de la variabilité des latences d'instructions. En effet, les contraintes d'ordonnancement dans les méthodes combinatoires sont exprimées sous la forme d'inégalités temporelles. Ces inégalités sont variables dans le cas des architectures asynchrones, ce qui rend le processus de recherche des solutions au problème difficile.

L'algorithme d'ordonnancement de liste, établi dans ce mémoire, se distingue des algorithmes d'ordonnancement de liste pour architectures synchrones par l'évaluation dynamique des latences ainsi que la nature des critères. Notre algorithme d'ordonnancement ajuste, à chaque itération, les latences des instructions, tandis que les algorithmes classiques opèrent statiquement. Tandis que l'objectif principal des critères usuels (chemin critique, hauteur d'instruction, etc.) consiste à augmenter le parallélisme niveau instruction, notre algorithme d'ordonnancement cherche à trouver un équilibre entre le parallélisme et la transmission rapide des opérandes (*result forwarding*).

3.1 Modélisation

3.1.1 Modélisation temporelle

Dans le but d'ordonnancer les instructions d'une manière efficace, il est nécessaire d'estimer le coût temporel des différentes opérations supportées par l'AnARM. Dans cette optique, un modèle du comportement temporel de l'AnARM est établi. Ce modèle définit une échelle temporelle discrète et approximative permettant d'exprimer le temps requis pour chaque étape du pipeline en nombre d'unités temporelles. Un modèle similaire décrivant le comportement du DSP Vocallo d'OCTASIC est introduit par José-Phillipe Tremblay dans son mémoire (Tremblay, 2009). Pour ce modèle, l'unité temporelle considérée, qui constitue la base de l'échelle, représente la durée de l'étape de l'acquisition de l'instruction qui s'évalue à environ 800 picosecondes. Le coût temporel des autres étapes du pipeline est exprimé en multiples de cette unité temporelle. Le tableau 3.1 donne les délais discrets associés à chaque étape de l'itinéraire des instructions. À partir de ces valeurs temporelles, on associe à chaque instruction un vecteur statique (VSD) contenant les durées discrètes de chaque étape du pipeline qu'elle requiert. Le vecteur associé à l'instruction ADD R1 R2 R3, par exemple, est présenté dans le tableau 3.2. Le VSD d'une instruction permet d'évaluer sa latence statique. La latence statique est la somme des délais de chaque étape de l'itinéraire d'une instruction sans la prise en considéra-

Tableau 3.1 Délais associés aux différentes étapes du pipeline

Étage du pipeline	**Délai en unités temporelles**
Aquisition et décodage de l'instruction (F)	1
Lecture des registres (RR)	1 x Nombre des registres lus
Lecture des cases mémoire (MR)	1 x Nombre des cases mémoire lues
Exécution (EX)	1 pour MOV et logique 2 pour ADD, SUB et arithmétique 4 pour STR et MUL 8 pour LD
Écriture des registres (RW)	1 x Nombre de registres écrits
Écriture des cases mémoire (MW)	1 x Nombre de cases mémoire écrites

Tableau 3.2 Vecteur statique des délais (VSD) de l'instruction ADD R1 R2 R3

Fetch & decode	*Register read*	*Memory read*	*Execute*	*Register write*	*Memory write*
1	2	0	2	1	0

tion du flot d'exécution. Par conséquent, cette latence s'évalue comme si l'ensemble des étapes nécessaires à l'exécution d'une instruction s'enchaîne de manière séquentielle et qu'aucune interruption due à l'indisponibilité des ressources, de calcul ou de synchronisation, n'a lieu. Par exemple, la latence statique de l'instruction figurant dans le tableau 3.2 est évaluée à 6 unités temporelles.

3.1.2 Modélisation spatiale

La modélisation spatiale permet de décrire le comportement de l'AnARM au niveau des ressources de calcul et de synchronisation. Le modèle établi définit un étage d'exécution ayant une largeur de 16 instructions. Les autres étages du pipeline sont définis comme des ressources à usage exclusif, en raison de l'unicité des jetons de synchronisation. Comme décrit dans le chapitre 1, seul l'étage d'exécution permet le traitement parallèle de 16 instructions, les autres étapes du pipeline sont effectuées au tour par tour. La modélisation du comportement spatial de l'AnARM permet le calcul des latences dynamiques des instructions. La latence dynamique représente la durée réelle d'une instruction, évaluée en considérant les interruptions dues à l'indisponibilité des ressources de calcul et de synchronisation. Cette latence représente la durée effective de l'instruction au moment de son ordonnancement et dépend de l'état actuel de la propagation de la signalisation asynchrone et de la disponibilité des ressources de calculs. En considérant que les résultats des instructions sont disponibles (préalablement à l'étape d'écriture du résultat) à la fin de l'étape de l'exécution grâce au *crossbar*, la latence dynamique ne prend en compte que les étages d'entrée du pipeline ainsi que l'étage d'exécution. En résumé, la modélisation spatiale permet de documenter l'évolution de l'occupation des ressources et de la signalisation asynchrone, au fur et à mesure du processus d'ordonnancement, dans le but de calculer, en temps réel, les latences des instructions. Finalement la latence dynamique

d'une instruction se calcule en fonction de sa latence statique, du vecteur de l'état du contrôle asynchrone et du vecteur de l'occupation des ressources de calcul.

Vecteur de l'état du contrôle asynchrone (VECA) : Ce vecteur s'écrit comme suit,

$$VECA = \begin{bmatrix} tu_F \\ tu_{RR} \\ tu_{MR} \end{bmatrix}, \tag{3.1}$$

où tu_F, tu_{RR} et tu_{MR} représentent les instants à partir desquels, les étages d'acquisition de l'instruction, de lecture des registres et de lecture des cases mémoires seront, respectivement, disponibles. Initialement, ces étages du pipeline sont disponibles, par conséquent le vecteur d'état du contrôle asynchrone est nul. Ce vecteur est mis à jour, à chaque itération de l'algorithme d'ordonnancement, pour documenter la disponibilité des ressources de synchronisation.

Vecteur de l'occupation des ressources de calcul (VORC) : Ce vecteur indique, à chaque étape, la durée restante avant la prochaine disponibilité des unités d'exécution. Le VORC s'exprime comme suit,

$$VORC = \begin{bmatrix} tr_{EU0} \\ tr_{EU1} \\ \vdots \\ tr_{EU15} \end{bmatrix}, \tag{3.2}$$

où tr_{EUi} représente la durée restante (en unités temporelles) avant que l'unité d'exécution EUi ne soit disponible. Supposons, par exemple, que le flot d'exécution du programme est rendu à la 16e unité d'exécution et que $tr_{EU15} = 0$, ceci signifie que l'unité d'exécution EU15 est disponible et que l'instruction courante ne subira aucun retard au niveau de l'étage d'exécution.

Calcul de la latence dynamique : Le calcul de la latence dynamique d'une instruction se fait à partir du vecteur statique des délais de l'instruction, du vecteur de l'état du contrôle asynchrone et du vecteur de l'occupation des ressources de calcul. Soit l'instruction I, ayant le vecteur des

délais suivant,

$$VSD(I) = \begin{bmatrix} t_F \\ t_{RR} \\ t_{MR} \\ t_{EX} \\ t_{RW} \\ t_{MW} \end{bmatrix}. \tag{3.3}$$

Soient les vecteurs VECA et VORC au moment du calcul de la latence dynamique de l'instruction I,

$$VECA = \begin{bmatrix} tu_F \\ tu_{RR} \\ tu_{MR} \end{bmatrix}, VORC = \begin{bmatrix} tr_{EU0} \\ tr_{EU1} \\ \vdots \\ tr_{EU15} \end{bmatrix}. \tag{3.4}$$

Supposant que le flot d'exécution se situe au niveau de l'unité d'exécution EUi, le vecteur dynamique des délais (VDD) de l'instruction I s'écrit comme suit,

$$VDD(I) = \begin{bmatrix} td_F \\ td_{RR} \\ td_{MR} \\ td_{EX} \end{bmatrix} = \begin{bmatrix} t_F + tu_F \\ max(td_F, tu_{RR}) + t_{RR} \\ max(td_{RR}, tu_{MR}) + t_{MR} \\ td_{MR} + t_{EX} + tr_{EUi} \end{bmatrix}, \tag{3.5}$$

les valeurs de td_F, td_{RR}, td_{MR} et td_{EX} représentent les instants auxquels les étapes du pipeline, qui y sont associées, seront achevées.

La latence dynamique de l'instruction I se calcule à partir de VDD comme suit,

$$LD(I) = td_{EX} - td_F + t_F. \tag{3.6}$$

Il est à noter que les dépendances de données ne sont pas prises en compte dans le calcul de la latence dynamique. Les dépendances de données sont considérées pendant l'évaluation du critère d'ordonnancement comme dans le cas des méthodes d'ordonnancement usuelles.

Mise à jour des vecteurs VECA et VORC : Dans le cas où l'instruction I est ordonnancée à la position courante, il est nécessaire d'actualiser les vecteurs VECA et VORC en vue de l'itération d'ordonnancement suivante. Dans le cas du VECA, les ressources de synchronisation seront disponibles à la fin de leur utilisation par l'instruction I. En conséquence, le VECA actualisé s'exprime comme suit,

$$VECA(I) = \begin{bmatrix} td_F \\ td_{RR} \\ td_{MR} \end{bmatrix}. \tag{3.7}$$

L'actualisation du VORC consiste à décrémenter la durée restante de l'occupation des unités d'exécution (jusqu'à un minimum de 0) en vue de la prochaine itération. De plus, il est nécessaire d'ajouter, à la position de l'unité d'exécution EUi, le nombre d'unités temporelles requis pour l'exécution de l'instruction I,

$$VORC(I) = \begin{bmatrix} tr_{EU0} - 1 \\ tr_{EU1} - 1 \\ \vdots \\ tr_{EUi} + td_{EX} - 1 \\ \vdots \\ tr_{EU15} - 1 \end{bmatrix}. \tag{3.8}$$

Exemple : Considérons la séquence d'instructions de la figure 3.1. Initialement, les vecteurs VECA et VORC sont nuls et le flot d'exécution débute par l'unité d'exécution EU0. La première instruction ordonnancée est MUL R0 R6 R5. Le vecteur statique des délais de cette

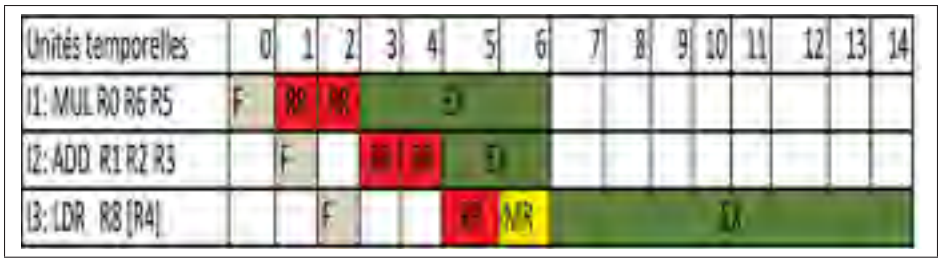


Figure 3.1 Diagramme de Gantt pour une séquence d'instructions donnée.

instruction, selon le tableau 3.1, est,

$$VSD(I1) = \begin{bmatrix} t_F \\ t_{RR} \\ t_{MR} \\ t_{EX} \\ t_{RW} \\ t_{MW} \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 4 \\ 1 \\ 0 \end{bmatrix}. \tag{3.9}$$

Le vecteur dynamique des délais de cette instruction est le suivant,

$$VDD(I1) = \begin{bmatrix} td_F \\ td_{RR} \\ td_{MR} \\ td_{EX} \end{bmatrix} = \begin{bmatrix} t_F + tu_F \\ max(td_F, tu_{RR}) + t_{RR} \\ max(td_{RR}, tu_{MR}) + t_{MR} \\ td_{MR} + t_{EX} + tr_{EU0} \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 3 \\ 7 \end{bmatrix}, \tag{3.10}$$

La latence dynamique de cette instruction est évaluée à 7 unités temporelles (7-1+1). L'actualisation des vecteurs VECA et VORC, suite à l'ordonnancement de l'instruction I1, est la suivante,

$$VECA(I1) = \begin{bmatrix} 1 \\ 3 \\ 3 \end{bmatrix}, VORC(I1) = \begin{bmatrix} 6 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{3.11}$$

La deuxième instruction ordonnancée (I2) est destinée à l'unité d'exécution EU1. I2 possède le vecteur statique des délais suivant,

$$VSD(I2) = \begin{bmatrix} t_F \\ t_{RR} \\ t_{MR} \\ t_{EX} \\ t_{RW} \\ t_{MW} \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 2 \\ 1 \\ 0 \end{bmatrix}, \tag{3.12}$$

VDD(I2) est calculé à partir de VECA(I1) et VORC(I1),

$$VDD(I2) = \begin{bmatrix} t_F + tu_F \\ max(td_F, tu_{RR}) + t_{RR} \\ max(td_{RR}, tu_{MR}) + t_{MR} \\ td_{MR} + t_{EX} + tr_{EU1} \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 5 \\ 7 \end{bmatrix}. \tag{3.13}$$

La latence dynamique de I2 est évaluée à 6 unités temporelles (7-2+1). L'actualisation des vecteurs d'état est la suivante,

$$VECA(I2) = \begin{bmatrix} 2 \\ 5 \\ 5 \end{bmatrix}, VORC(I2) = \begin{bmatrix} 5 \\ 6 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{3.14}$$

Le calcul de la latence dynamique de l'instruction I3 se fait de la même manière,

$$VSD(I3) = \begin{bmatrix} t_F \\ t_{RR} \\ t_{MR} \\ t_{EX} \\ t_{RW} \\ t_{MW} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 8 \\ 1 \\ 0 \end{bmatrix}, \tag{3.15}$$

$$VDD(I3) = \begin{bmatrix} t_F + tu_F \\ max(td_F, tu_{RR}) + t_{RR} \\ max(td_{RR}, tu_{MR}) + t_{MR} \\ td_{MR} + t_{EX} + tr_{EU2} \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 7 \\ 15 \end{bmatrix}. \tag{3.16}$$

La latence dynamique de l'instruction I3 est 13 unités temporelles (15-3+1) et l'actualisation des vecteurs d'états se fait de la même manière,

$$VECA(I3) = \begin{bmatrix} 3 \\ 6 \\ 7 \end{bmatrix}, VORC(I3) = \begin{bmatrix} 4 \\ 5 \\ 14 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{3.17}$$

3.2 Algorithme d'ordonnancement pour l'AnARM

3.2.1 Stratégie d'ordonnancement

La stratégie d'ordonnancement développée dans ce mémoire vise l'amélioration du temps d'exécution des programmes sur l'AnARM. Dans cette optique, le premier objectif consiste

à minimiser les blocages induits par les dépendances de données et l'augmentation du parallélisme au niveau des instructions. Le deuxième objectif vise l'exploitation et la prise en considération des spécificités architecturales de l'AnARM (latences variables, accès séquentiel aux ressources et partage des résultats). Ces objectifs sont fortement reliés dans la mesure où la réduction de l'impact des dépendances de données passe par la caractérisation du comportement architectural de l'AnARM. L'augmentation du parallélisme au niveau des instructions se fait par la séparation des instructions liées par des dépendances, ainsi que l'émission en parallèle des instructions indépendantes, pour profiter de largeur d'exécution du processeur. Soit I_m et I_n, deux instructions de la même séquence, telles que $I_n \prec I_m$. L'inégalité exprimant la contrainte des dépendances de données, $t(I_m) \geq t(I_n) + l\ (I_n, I_m)$, signifie que si l'instruction I_m est émise avant la fin du traitement de son prédécesseur I_n, un blocage du pipeline surviendra. Les méthodes d'ordonnancement axées sur le parallélisme au niveau des instructions émettent, autant que possible, des instructions indépendantes entre I_n et I_m pour couvrir la durée $l\ (I_n, I_m)$. Dans le cas de l'AnARM, le flot d'exécution n'est que partiellement parallèle à cause de la transmission séquentielle des jetons de synchronisation. Subséquemment, l'impact des dépendances de données est moindre, car la durée d'attente pour les signaux de synchronisation couvre une partie ou la totalité de $l\ (I_n, I_m)$. Pour cette raison, la méthode d'ordonnancement présentée dans ce mémoire utilise la latence dynamique actualisée du prédécesseur dans le calcul du critère de décision au moment de l'ordonnancement de son successeur. Dans le cas des instructions I_n et I_m, la latence dynamique de l'instruction I_n est évaluée lors de son émission. Cette latence est décrémentée de 1 à chaque itération. Au moment de l'ordonnancement de l'instruction I_m, la latence dynamique actualisée de I_n indique la durée restante avant la disponibilité de son résultat pour son successeur I_m. Considérons l'exemple de la figure 3.1. Supposons qu'on veut émettre une quatrième instruction I_4 qui dépend de I_1. La latence dynamique de l'instruction I_1 est de 7 unités temporelles. Au moment de l'ordonnancement de l'instruction I_4, la latence dynamique actualisée de I_1 s'évalue à 4, suite à l'émission des instructions I_2 et I_3. L'instruction I_4 aura donc accès à la donnée qu'elle requiert dans un délai de 4 unités temporelles à partir du début de son traitement.

L'évaluation de la latence dynamique actualisée du prédécesseur permet de pénaliser les instructions ayant des lents prédécesseurs au moment de leur ordonnancement, réduisant ainsi le nombre des blocages du pipeline dus aux dépendances de données. En addition à ceci, la méthode d'ordonnancement proposée attribue une plus haute priorité aux instructions lentes (selon leur latence statique), car ces instructions possèdent, en général, plus de successeurs et utilisent plus de ressources de synchronisation comparées aux instructions rapides. En effet, dans une architecture RISC, la manipulation de la mémoire est indirecte et passe par le chargement des valeurs stockées en mémoire dans les registres du processeur. De cette manière l'ensemble des données d'un programme ou d'une fonction est chargé à partir de la mémoire avant d'être traité dans le cadre des différentes opérations du programme. De plus, les instructions lentes, comme les instructions mémoire ou les multiplications, possèdent plus d'opérandes que les instructions binaires ou les instructions d'arithmétique simples. En conséquence, ces instructions occupent, plus longtemps, les étages du pipeline dont l'usage est exclusif (RR, MR).

Le deuxième objectif de la stratégie d'ordonnancement est de tirer profit du mécanisme avantageux de la transmission rapide des résultats de l'AnARM. Comme expliqué dans le cadre du chapitre 1, l'AnARM, dispose du *Crossbar*, ce qui lui permet de transmettre rapidement le résultat d'un calcul effectué par une unité d'exécution vers une autre unité d'exécution qui nécessite ce résultat. La disponibilité d'un résultat dans le *Crossbar* est limitée aux unités d'exécution suivantes jusqu'au retour à l'unité d'exécution qui émet le résultat. Par exemple, le résultat produit par l'unité d'exécution 3 sera disponible pour les unités d'exécution 4, 5, ..., 15 et 0, 1, 2. Par conséquent, le critère de décision, relatif à l'utilisation du *Crossbar* et utilisé dans la méthode d'ordonnancement proposée, évalue le nombre des opérandes lus par une instruction et produits par les 15 dernières instructions émises. Le but de ce critère est de donner la priorité aux instructions qui lisent leurs données à partir du *Crossbar* pour éviter des transactions coûteuses avec la banque des registres.

Le dernier élément de la stratégie d'ordonnancement concerne les unités d'exécution qui supportent les multiplications et les instructions mémoire. Dans le but d'éviter des interruptions du flot d'exécution, engendrées par l'émission d'une instruction vers une unité d'exécution ne supportant pas cette instruction, il est nécessaire de pénaliser cette instruction. Supposons qu'une instruction LOAD est disponible et que le flot d'exécution se situe au niveau d'une unité d'exécution impaire. Les instructions mémoire ne sont supportées que par les unités d'exécutions paires, en conséquence, l'instruction LOAD se verra attribuer une pénalité égale la durée de son étage d'exécution. Cette pénalité disparaîtra à la prochaine itération, car l'unité d'exécution suivante sera paire. La même pénalité est appliquée aux multiplications dans le cas d'une unité d'exécution paire.

Pour résumer, la stratégie d'ordonnancement proposée pour l'AnARM, vise deux objectifs principaux. Le premier objectif consiste à réduire l'effet des dépendances de données, tandis que le deuxième consiste à favoriser l'utilisation du *Crossbar*. Pour cela les instructions disponibles pour l'ordonnancement seront évaluées sur la latence dynamique actualisée de leurs prédécesseurs, leur latence statique et le nombre d'opérandes acquis grâce au *Crossbar*.

3.2.2 Description de l'algorithme d'ordonnancement proposé

L'algorithme d'ordonnancement proposé dans ce mémoire est un algorithme d'ordonnancement de liste modifié. À chaque itération, les instructions disponibles pour l'ordonnancement sont comparées entre elles selon un critère issu de la stratégie d'ordonnancement présentée au sein de la sous-section précédente. Le critère établi se compose de deux sous-critères et d'une pénalité. Le premier sous critère (C1) est calculé pour chaque instruction au moment de son ordonnancement en fonction de sa latence statique ainsi que la latence dynamique actualisée maximale de ses prédécesseurs. Ce critère s'exprime comme suit pour une instruction I,

$$C_1(I) = LS(I) - max(LDA(predecessors(I)). \tag{3.18}$$

Dans cette équation $LS(I)$ représente la latence statique de l'instruction I tandis que $max(LDA(predecessors(I))$ représente la latence dynamique actualisée maximale des prédécesseurs de I. Ce sous-critère favorise les instructions lentes et pénalise les instructions dont les prédécesseurs sont lents.

Le deuxième sous-critère (C2) favorise les instructions qui lisent des données préalablement écrites à une distance maximale de 16 instructions. Ce sous-critère permet de rapprocher les instructions de leurs prédécesseurs dans le but de profiter du *Crossbar*. C2 s'exprime comme suit,

$$C_2(I) = Nb(predecessors_{16}(I)). \tag{3.19}$$

$Nb(predecessors_{16}(I))$ est le nombre des prédécesseurs de I, qui ont été ordonnancés parmi les 16 dernières instructions émises.

Les deux sous critères C1 et C2 sont contradictoires, car C1 est utilisé pour séparer les instructions liées par des dépendances, tandis que C2 est utilisé pour les rapprocher suffisamment dans le but de favoriser l'utilisation du *Crossbar*. La pénalité P(I), associée au non support de l'instruction I, s'exprime comme telle,

$$P(I) = \begin{cases} LS(I), si \quad EU \notin EU(I) \\ 0, sinon \end{cases}. \tag{3.20}$$

La pénalité pour l'instruction I s'évalue à LS(I) dans le cas où l'unité d'exécution courante ne supporte pas le type de I.

Le critère principal C est une combinaison linéaire des deux sous critères C1 et C2 à laquelle on soustrait, éventuellement, la pénalité P. Le critère C(I) s'écrit comme suit,

$$C(I) = w_1 * C_1(I) + w_2 * C_2(I) - P(I). \tag{3.21}$$

Dans cette équation w_1 et w_2 sont deux facteurs complémentaires de poids tels que $w_1 + w_2 = 1$. Les valeurs de ces facteurs sont déterminées empiriquement, en évaluant la combinaison produisant les meilleurs résultats.

L'algorithme d'ordonnancement proposé est décrit par le pseudo-code de l'algorithme 3.1. Dans la première étape de l'algorithme, l'ensemble des paramètres est initialisé. L'algorithme crée une file d'exécution dans laquelle les instructions émises seront empilées. La liste des instructions prêtes est initialisée avec les instructions indépendantes (ne possédant aucun prédécesseur). Tant que la liste des instructions prêtes n'est pas vide, l'algorithme procède à évaluer chaque instruction de cette liste selon le critère d'ordonnancement C. Le sous-critère C1 est évalué en calculant la latence statique des instructions et la latence dynamique maximale de leurs prédécesseurs. Le sous-critère C2 est évalué en calculant le nombre de leurs prédécesseurs appartenant à la file d'exécution, entre les positions 0 et 15. La pénalité est calculée si l'unité d'exécution courante ne fait pas partie de l'itinéraire de l'instruction dont on évalue le critère. Par la suite, l'algorithme trie la liste des instructions prêtes, selon le critère C, dans un ordre décroissant. La meilleure instruction selon C est dépilée à partir de la liste triée des instructions prêtes. Cette instruction est empilée dans la file d'exécution. L'algorithme parcourt la liste des successeurs de l'instruction émise pour enlever cette instruction de la liste de leurs prédécesseurs. Par la suite, le vecteur dynamique des délais et la latence dynamique de l'instruction émise sont calculés à partir des vecteurs VECA et VORC. Ces derniers sont mis à jour suite à l'émission de la meilleure instruction selon la méthode présentée dans la section 3.1.2. La liste des instructions prêtes est mise à jour en y ajoutant les instructions dont tous les prédécesseurs ont été émis. Finalement, les latences dynamiques des instructions de la file d'exécution sont actualisées, tandis que l'instant courant et l'unité d'exécution courante sont incrémentés. L'algorithme d'ordonnancement produit une file d'exécution dont la première instruction est la dernière à s'exécuter.

Algorithme 3.1 Aperçu global de l'algorithme proposé

Input : Séquence d'instructions non ordonnancée (S), dépendances de données, latences statiques et itinéraires des instructions
Output : File d'exécution

```
1  t = 0 ;
2  EU = 0 ;
3  VECA = 0_3 ;
4  VORC = 0_16 ;
5  FileExecution = 0_sizeof(S) ;
6  ListePrete = instructions initialement indépendantes;
7  while ListePrete est non vide do
8      for i=0, i<sizeof(ListePrete),i++ do
9          CalculerC1(ListePrete[i]) ;
10         CalculerC2(ListePrete[i], FileExecution) ;
11         CalculerP(ListePrete[i]) ;
12         CalculerC(ListePrete[i]) ;
13     end
14     Sort(ListePrete) selon C dans un ordre décroissant;
15     TopInstruction = ListePrete.pop();
16     FileExecution.push(TopInstruction);
17     TopInstruction.releasePreds();
18     TopInstruction.calculerVDD(VECA,VORC);
19     updateVECA(TopInstruction);
20     updateVORC(TopInstruction);
21     updateListePrete();
22     for i=0, i<sizeof(FileExecution),i++ do
23         if FileExecution[i].LD > 0 then
24             FileExecution[i].LD- - ;
25         end
26     end
27     t++;
28     EU= (EU+1) mod 16;
29 end
30 return File d'exécution
```

3.3 Conclusion

L'algorithme d'ordonnancement, proposé dans ce mémoire, se base sur une description architecturale de l'AnARM, pour estimer les délais des instructions, selon leurs itinéraires dans le

pipeline d'exécution. Chaque étage du pipeline est représenté par une ressource à usage exclusif ou concurrentiel, selon le jeton de synchronisation qui y est associé. Les vecteurs d'état des ressources de calcul et de synchronisation documentent la progression du flot d'exécution et permettent d'évaluer le coût, temporel et spatial, des décisions à prendre au niveau de l'ordonnancement. Le critère d'ordonnancement établi permet d'appliquer la stratégie d'ordonnancement désirée. Cette stratégie favorise le parallélisme au niveau des instructions et le partage rapide des résultats des instructions, en prenant en considération les contraintes imposées par les mécanismes de synchronisation spécifiques à l'AnARM. L'algorithme d'ordonnancement proposé est un algorithme d'ordonnancement de liste qui se distingue par l'évaluation dynamique du critère de décision à travers la mise à jour de ses paramètres à chaque itération.

Le chapitre suivant présentera les résultats de l'approche adoptée. Une description de la plateforme et des spécificités de l'implémentation introduira le chapitre. Par la suite, une étude de sensibilité permettra de définir les valeurs des facteurs de poids aboutissant aux meilleurs résultats. Finalement, l'approche proposée sera évaluée sur différents programmes. Les résultats en termes de temps d'exécution et de consommation d'énergie seront comparés aux résultats des approches classiques.

CHAPITRE 4

ÉVALUATION DE L'APPROCHE PROPOSÉE

Le présent chapitre offre une description de la démarche expérimentale et des résultats obtenus suite à l'implémentation de l'approche proposée. Tout d'abord, le compilateur choisi pour l'implémentation sera présenté, pour offrir au lecteur un aperçu de l'environnement de l'implémentation. Par la suite, l'ensemble des éléments relatifs à l'environnement expérimental sera décrit (plateforme de simulation, structure des tests, etc.).

L'approche proposée a été évaluée en termes de réduction du temps d'exécution et de la consommation d'énergie d'une sélection de programmes d'évaluation (*benchmarks*) par rapport aux algorithmes d'ordonnancement utilisés par le compilateur au plus haut niveau d'optimisation.

De plus, une évaluation de l'effet des paramètres de l'algorithme d'ordonnancement présenté dans ce mémoire permettra de déceler la combinaison de paramètres produisant les meilleurs résultats.

4.1 Environnement d'implémentation

Afin d'implémenter et d'évaluer l'approche d'ordonnancement élaborée dans le cadre de ce travail de recherche, le choix de la plateforme de compilation s'est porté sur l'infrastructure de compilation LLVM (Lattner, 2006). LLVM est un compilateur moderne libre-source (*open-source*) basé sur la séparation des phases d'avant et d'arrière-plan. Ce compilateur est fréquemment utilisé dans les travaux de recherche en raison de sa nature extensible et modulaire. En effet, ce compilateur a été développé à travers plusieurs contributions visant à étendre ses fonctionnalités, au niveau du support des langages, des architectures et des optimisations. Pour cette raison, LLVM se distingue par le grand niveau d'organisation des différents outils qui le composent ainsi que la clarté de son implémentation (C++). Par conséquent, LLVM offre un

cadre simple et ergonomique pour les chercheurs et les entreprises désirant y contribuer ou le complémenter dans le but de l'adapter à leurs besoins.

En dépit du grand nombre d'outils et de fonctionnalités qui composent l'infrastructure LLVM, le processus de compilation s'effectue en trois étapes principales : l'avant-plan, le langage intermédiaire et l'arrière-plan.

La compilation d'avant plan de LLVM se fait à l'aide de l'outil *Clang*. Ce dernier permet de transformer le langage source en langage intermédiaire LLVM. *Clang* supporte la majorité des langages sources existants (C, C++, java, Objective-C, Fortran, Python, etc.) et il est utilisé par plusieurs logiciels de développement, dont X-Code d'Apple.

LLVM-IR est le langage intermédiaire de LLVM. Ce langage est une représentation compréhensible des programmes générés par *Clang*. LLVM-IR est un langage compact similaire à un langage assembleur d'une architecture RISC (*Load-Store*). Plusieurs accès à la mémoire peuvent se faire par le biais d'une seule instruction. De plus, les instructions arithmétiques peuvent opérer sur des vecteurs de données. LLVM-IR est un langage ayant une forme SSA (*Static Single Assignment*) ce qui signifie que les variables ne sont assignées qu'une seule fois et qu'elles sont définies avant leur utilisation. Ceci permet de faciliter certaines optimisations comme dans le cas de l'élimination du code mort où les instructions, qui définissent des variables non utilisées par la suite, sont éliminées. Subséquemment, le langage intermédiaire de LLVM utilise une banque infinie de registres. LLVM-IR joue un rôle central dans l'architecture de ce compilateur, car il offre un cadre de travail pour un certain nombre d'optimisations.

Finalement, la phase d'arrière-plan de LLVM s'effectue à l'aide de l'outil LLC. Cet outil transforme le code LLVM-IR en code assembleur de la machine cible spécifiée au moment de la compilation. La version actuelle de LLVM supporte un grand nombre d'architectures cibles dont ARM, X86, SPARC, etc. Le fonctionnement de la phase d'arrière-plan de LLVM s'appuie sur la communication entre une représentation générique indépendante de l'architecture cible

et une librairie contenant la description de chaque processeur supporté par LLVM à différents niveaux. La figure 4.1 (Lopes & Auler, 2014) illustre la structure de la phase d'arrière-plan de LLVM.

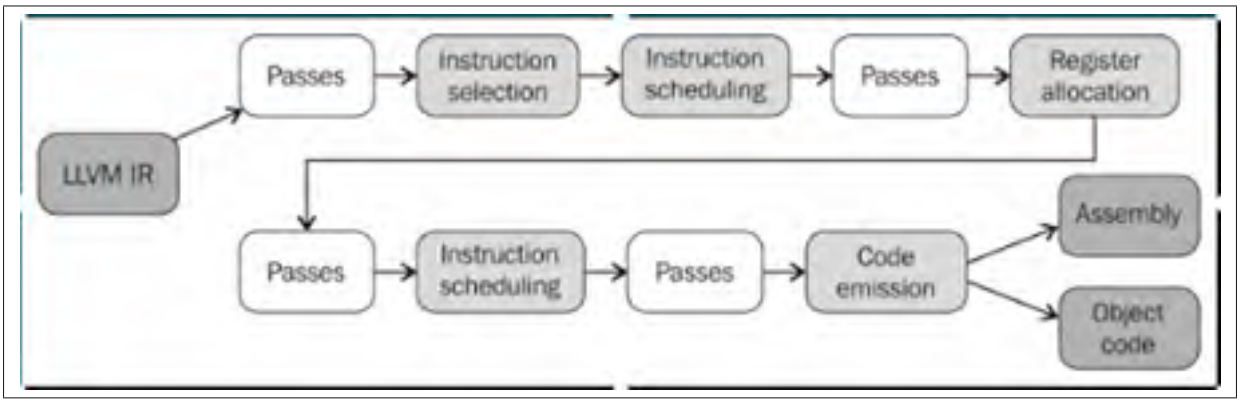


Figure 4.1 Structure de la phase d'arrière-plan de LLVM
Tirée de Lopes & Auler (2014, p. 248)

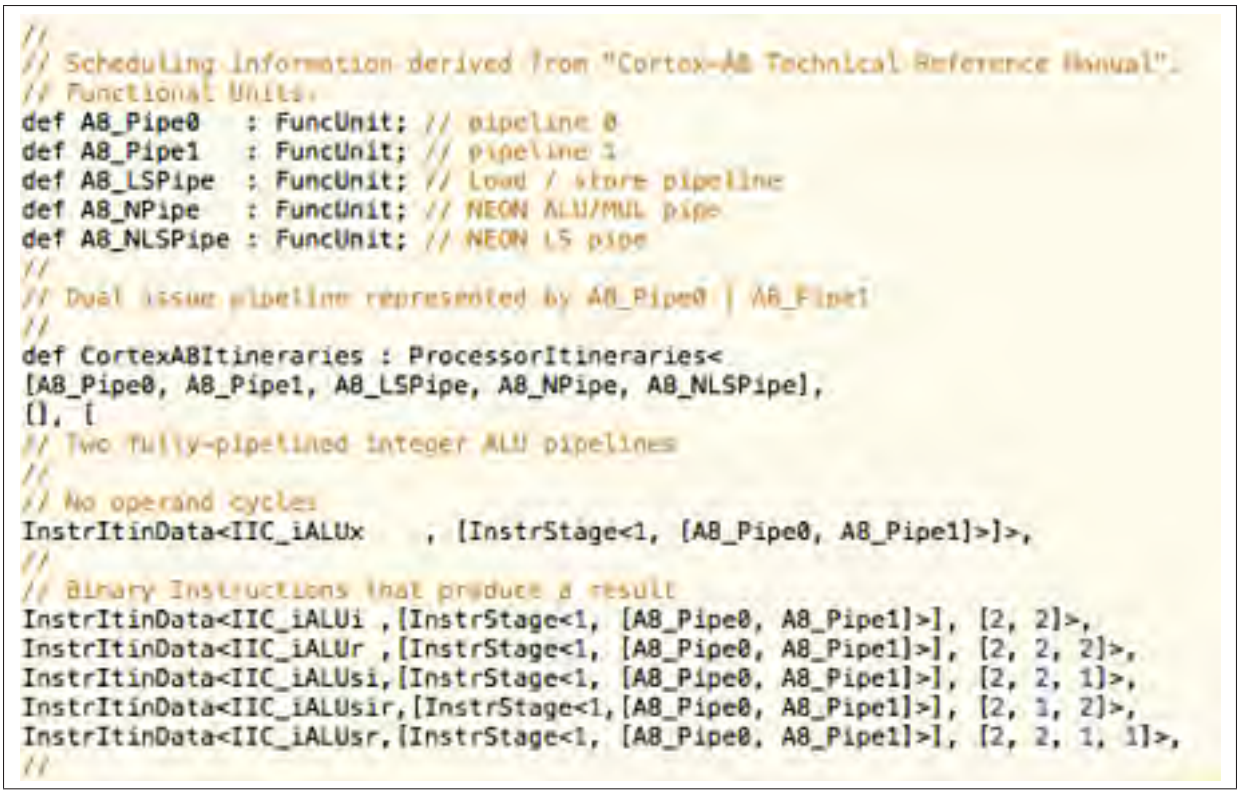
Tout d'abord, un algorithme, de sélection des instructions, choisit les meilleures instructions-machine qui correspondent au traitement décrit par le langage intermédiaire. Cette étape génère une représentation en graphe de flot de données, où chaque bloc de base est représenté par une hiérarchie de noeuds (instructions). La première phase d'ordonnancement des instructions survient juste après leur sélection. Il est à noter qu'à ce stade et jusqu'à l'allocation des registres, les instructions utilisent les registres virtuels infinis de la représentation intermédiaire. L'ordonnancement des instructions pré-allocation (*Pre-register Allocation*) est destiné à augmenter le parallélisme au niveau des instructions en l'absence des dépendances WAR et WAW. L'étape suivante est l'allocation des registres. Un algorithme spécifié, ou par défaut, se réfère à la description de la banque des registres du processeur cible pour convertir les registres virtuels en registres physiques tout en ajoutant des instructions de *spilling* (sous-section 2.1.2.2) si nécessaire. Une deuxième phase d'ordonnancement des instructions survient après l'allocation des registres. Cette phase est appelée *Post-register Allocation Scheduling*. Dorénavant, les registres utilisés sont connus du compilateur, par conséquent, l'ordonnancement analyse la présence des nouvelles contraintes de dépendance pour améliorer l'ordre des instructions émises. Finale-

ment, la phase d'émission du code émet les codes résultants en langage assembleur ou objet, selon les paramètres spécifiés à la compilation.

4.1.1 Ordonnancement des instructions dans LLVM

Représentation des instructions : À la suite de la sélection des instructions, le compilateur assigne à chaque instruction, du code intermédiaire, une classe générique d'instructions-machine qui encapsule un ensemble d'informations relatives aux dépendances. La formation du GFD se fait à ce niveau et les instructions sont représentées par des noeuds appelés *SDNodes* (*Selection DAG Nodes*). Après l'allocation des registres, ces noeuds sont transformés en instructions-machine spécifiques, dont l'information s'y reliant est acquise auprès de la description du processeur cible. La classe *SUnit* est une classe intermédiaire qui permet de faire le lien entre les noeuds du GFD et les instructions-machine. Cette classe encapsule l'ensemble des paramètres de l'ordonnancement comme les latences, les prédécesseurs, les successeurs, les opérandes virtuels (avant l'allocation des registres) et les opérandes physiques (après l'allocation des registres). L'ordonnancement des instructions se fait, en conséquence, à travers la manipulation de cette représentation des instructions.

Données d'ordonnancement : Les latences des instructions sont calculées à partir d'un fichier définissant la structure des ressources du processeur cible ainsi que les itinéraires des instructions. Ces itinéraires définissent le parcours de traitement des instructions dans les unités fonctionnelles. Par exemple, la description des itinéraires de quelques instructions pour les processeurs ARMV7 est montrée dans la figure 4.2, tirée du fichier ARMScheduleA8.td de l'implémentation de LLVM. Ce fichier descriptif définit, tout d'abord, les ressources de calcul du processeur. Le ARM Cortex A8 dispose de deux unités fonctionnelles pour le calcul sur les entiers, une unité pour les opérations-mémoire, une unité pour le calcul en point flottant (Neon) et une unité pour les opérations-mémoire sur les points flottants (Neon). Le modèle d'instruction **IIC_IALUr** englobe les instructions arithmétiques opérant sur des valeurs entières contenues dans les registres (suffixe "r"). L'itinéraire de cette instruction se compose d'un seul étage d'un cycle qui utilise selon la disponibilité l'unité **A8_Pipe0** ou **A8_Pipe1**. La



```
//
// Scheduling information derived from "Cortex-A8 Technical Reference Manual".
// Functional Units.
def A8_Pipe0   : FuncUnit; // pipeline 0
def A8_Pipe1   : FuncUnit; // pipeline 1
def A8_LSPipe  : FuncUnit; // Load / store pipeline
def A8_NPipe   : FuncUnit; // NEON ALU/MUL pipe
def A8_NLSPipe : FuncUnit; // NEON LS pipe
//
// Dual issue pipeline represented by A8_Pipe0 | A8_Pipe1
//
def CortexA8Itineraries : ProcessorItineraries<
[A8_Pipe0, A8_Pipe1, A8_LSPipe, A8_NPipe, A8_NLSPipe],
[], [
// Two fully-pipelined integer ALU pipelines
//
// No operand cycles
InstrItinData<IIC_iALUx    , [InstrStage<1, [A8_Pipe0, A8_Pipe1]>]>,
//
// Binary Instructions that produce a result
InstrItinData<IIC_iALUi ,[InstrStage<1, [A8_Pipe0, A8_Pipe1]>], [2, 2]>,
InstrItinData<IIC_iALUr ,[InstrStage<1, [A8_Pipe0, A8_Pipe1]>], [2, 2, 2]>,
InstrItinData<IIC_iALUsi,[InstrStage<1, [A8_Pipe0, A8_Pipe1]>], [2, 2, 1]>,
InstrItinData<IIC_iALUsir,[InstrStage<1,[A8_Pipe0, A8_Pipe1]>], [2, 1, 2]>,
InstrItinData<IIC_iALUsr,[InstrStage<1, [A8_Pipe0, A8_Pipe1]>], [2, 2, 1, 1]>,
//
```

Figure 4.2 Aperçu des itinéraires de quelques instructions du ARM CortexA8

latence d'opérande est un paramètre optionnel décrivant la durée entre l'émission de l'instruction et la libération des opérandes. Dans le cas de l'instruction **IIC_IALUr** le paramètre de la latence des opérandes est $[2,2,2]$ ce qui signifie que l'instruction commence son traitement au premier cycle et libère les opérandes au deuxième cycle.

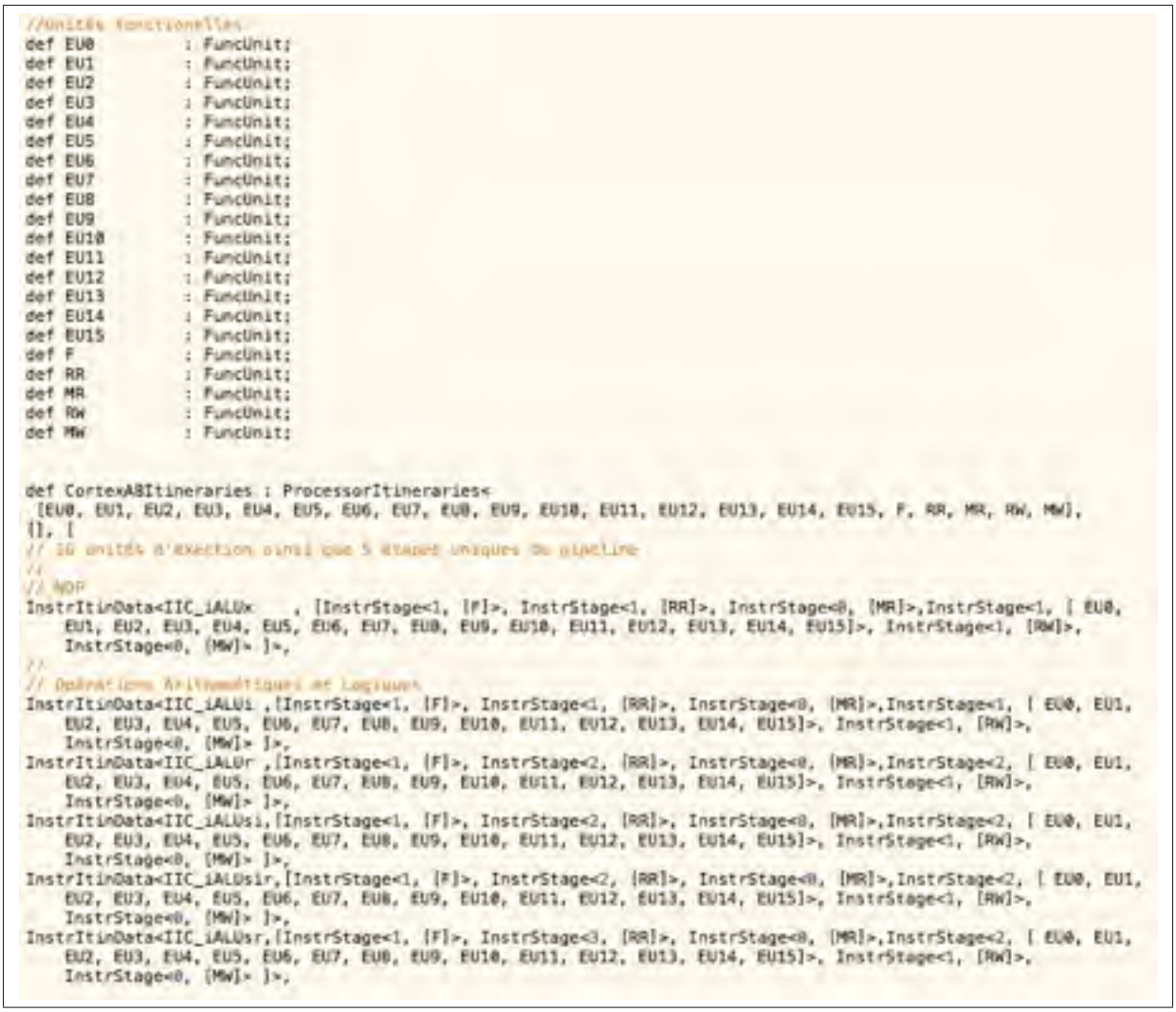
Les itinéraires permettent d'évaluer les latences des instructions selon la configuration du processeur. Cependant, ces itinéraires définissent les unités fonctionnelles à un haut niveau d'abstraction (entièrement pipelinées) et ne décrivent pas le comportement au niveau de chaque étage du pipeline, ce qui est essentiel dans le cas du AnARM en raison des mécanismes de synchronisation.

Algorithme et heuristiques d'ordonnancement : L'algorithme d'ordonnancement de LLVM est un algorithme d'ordonnancement de liste similaire à l'algorithme 2.1 (sous-section 2.2.1).

Plusieurs étapes préparatoires précédent le choix des instructions à ordonnancer en priorité. Tout d'abord la routine d'ordonnancement acquiert la composition du GFD et l'information des dépendances. La routine d'ordonnancement avant l'allocation des registres calcule l'évolution de la pression sur les registres (*register pressure*) qui représente le nombre de variables simultanément vivantes à chaque instant de l'ordonnancement. Les files contenant les instructions à ordonnancer et les instructions émises sont initialisées. Par la suite, la région à ordonnancer est parcourue dans le but de construire la liste des instructions prêtes pour l'ordonnancement. Les instructions prêtes sont comparées entre elles selon un certain nombre d'heuristiques définies dans l'implémentation de la stratégie d'ordonnancement. La stratégie d'ordonnancement de LLVM englobe plusieurs critères de comparaison placés dans un ordre de priorité défini. Tout d'abord des critères de cohérence précèdent les heuristiques d'ordonnancement. Parmi ces critères on peut citer la prévention des excès dans l'utilisation des registres, la validité des instructions à ordonnancer, etc. LLVM utilise deux heuristiques principales pour l'ordonnancement des instructions : la pression sur les registres (*register pressure*) et les heuristiques basées sur les latences. La première heuristique cherche à éviter une augmentation de la pression sur les registres en priorisant les instructions qui ne définissent pas de nouvelles variables. La deuxième heuristique est une combinaison des heuristiques de la hauteur d'instruction et du chemin critique telles que définies dans la section 2.2.1.

4.1.2 Implémentation de l'approche d'ordonnancement

Dans le cadre de l'implémentation de l'approche, proposée dans ce mémoire, la première étape consiste à définir le vecteur des délais statiques pour chaque instruction. Dans le but d'utiliser un modèle temporel qui définit des valeurs temporelles discrètes pour chaque étage du pipeline, une description des itinéraires des instructions dans l'AnARM, a été implémentée. Cette description définit 16 unités fonctionnelles pour l'étage d'exécution ainsi que 5 unités fonctionnelles uniques pour les autres étages du pipeline. Ces définitions sont illustrées dans la figure 4.3. La définition des itinéraires des instructions du AnARM est implémentée conformément au modèle des latences statiques défini au tableau 3.1. Chaque instruction est représentée



```
//Unités fonctionnelles
def EU0         : FuncUnit;
def EU1         : FuncUnit;
def EU2         : FuncUnit;
def EU3         : FuncUnit;
def EU4         : FuncUnit;
def EU5         : FuncUnit;
def EU6         : FuncUnit;
def EU7         : FuncUnit;
def EU8         : FuncUnit;
def EU9         : FuncUnit;
def EU10        : FuncUnit;
def EU11        : FuncUnit;
def EU12        : FuncUnit;
def EU13        : FuncUnit;
def EU14        : FuncUnit;
def EU15        : FuncUnit;
def F           : FuncUnit;
def RR          : FuncUnit;
def MR          : FuncUnit;
def RW          : FuncUnit;
def MW          : FuncUnit;


def CortexA8Itineraries : ProcessorItineraries<
 [EU0, EU1, EU2, EU3, EU4, EU5, EU6, EU7, EU8, EU9, EU10, EU11, EU12, EU13, EU14, EU15, F, RR, MR, RW, MW],
[], [
// 16 unités d'éxection ainsi que 5 étapes uniques du pipeline
//
// NOP
InstrItinData<IIC_iALUx    , [InstrStage<1, [F]>, InstrStage<1, [RR]>, InstrStage<0, [MR]>,InstrStage<1, [ EU0,
    EU1, EU2, EU3, EU4, EU5, EU6, EU7, EU8, EU9, EU10, EU11, EU12, EU13, EU14, EU15]>, InstrStage<1, [RW]>,
    InstrStage<0, [MW]> ]>,
//
// Opérations Arithmétiques et Logiques
InstrItinData<IIC_iALUi ,[InstrStage<1, [F]>, InstrStage<1, [RR]>, InstrStage<0, [MR]>,InstrStage<1, [ EU0, EU1,
    EU2, EU3, EU4, EU5, EU6, EU7, EU8, EU9, EU10, EU11, EU12, EU13, EU14, EU15]>, InstrStage<1, [RW]>,
    InstrStage<0, [MW]> ]>,
InstrItinData<IIC_iALUr ,[InstrStage<1, [F]>, InstrStage<2, [RR]>, InstrStage<0, [MR]>,InstrStage<2, [ EU0, EU1,
    EU2, EU3, EU4, EU5, EU6, EU7, EU8, EU9, EU10, EU11, EU12, EU13, EU14, EU15]>, InstrStage<1, [RW]>,
    InstrStage<0, [MW]> ]>,
InstrItinData<IIC_iALUsi,[InstrStage<1, [F]>, InstrStage<2, [RR]>, InstrStage<0, [MR]>,InstrStage<2, [ EU0, EU1,
    EU2, EU3, EU4, EU5, EU6, EU7, EU8, EU9, EU10, EU11, EU12, EU13, EU14, EU15]>, InstrStage<1, [RW]>,
    InstrStage<0, [MW]> ]>,
InstrItinData<IIC_iALUsir,[InstrStage<1, [F]>, InstrStage<2, [RR]>, InstrStage<0, [MR]>,InstrStage<2, [ EU0, EU1,
    EU2, EU3, EU4, EU5, EU6, EU7, EU8, EU9, EU10, EU11, EU12, EU13, EU14, EU15]>, InstrStage<1, [RW]>,
    InstrStage<0, [MW]> ]>,
InstrItinData<IIC_iALUsr,[InstrStage<1, [F]>, InstrStage<3, [RR]>, InstrStage<0, [MR]>,InstrStage<2, [ EU0, EU1,
    EU2, EU3, EU4, EU5, EU6, EU7, EU8, EU9, EU10, EU11, EU12, EU13, EU14, EU15]>, InstrStage<1, [RW]>,
    InstrStage<0, [MW]> ]>,
```

Figure 4.3 Aperçu des itinéraires des instructions de l'AnARM

par un chemin de traitement composé des différentes étapes qu'elle requiert selon sa nature ainsi que les opérandes qu'elle manipule. Par exemple, une instruction arithmétique qui utilise un décalage (*shift*) dont l'index est contenu dans un registre doit lire un registre de plus qu'une instruction similaire qui opère sans décalage. Ceci est le cas des instructions de type **iALUsr** (le suffixe "sr" signifie que l'instruction utilise un décalage dont l'index est contenu dans un registre). La latence de l'étage de lecture des registres de cette instruction est 3 unités temporelles.

Les multiplications et les instructions opérant sur la mémoire sont implémentées de la même manière. L'étage d'exécution de ces instructions utilise un sous-ensemble des unités fonctionnelles disponibles. La figure 4.4 illustre l'implémentation des itinéraires pour les multiplications et les instructions **LOAD**. La description des itinéraires de traitement des instructions

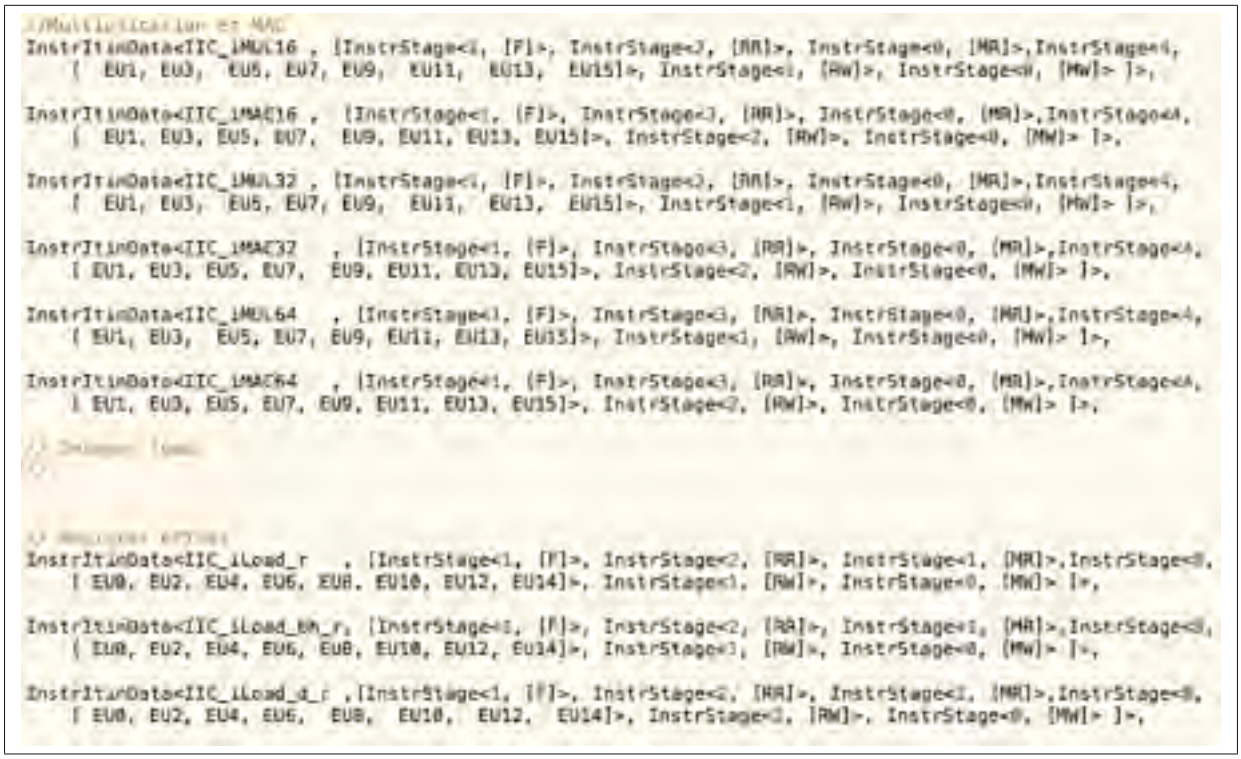


Figure 4.4 Aperçu des itinéraires des multiplications et des instructions LOAD

permet de renseigner l'algorithme d'ordonnancement sur les latences statiques des instructions ainsi que les unités fonctionnelles qu'elles occupent. À partir de cette description, des routines d'acquisition des délais permettent d'obtenir les valeurs temporelles associées à chaque étage du pipeline pour chaque instruction supportée par l'AnARM.

L'implémentation de l'approche d'ordonnancement basée sur les latences dynamiques a nécessité certaines modifications au niveau des routines d'ordonnancement de LLVM. Tandis que les heuristiques de LLVM utilisent des paramètres d'ordonnancement qui ne dépendent que de

l'instruction considérée, l'approche présentée dans ce mémoire requiert un ensemble d'informations concernant l'état du flot d'exécution et d'utilisation des ressources de calcul. Dans cette optique, une routine permettant l'acquisition des durées d'occupation de chaque étage du pipeline a été implémentée. Cette routine cherche la composition de l'itinéraire de l'instruction concernée et retourne un vecteur des latences qui correspond au vecteur des délais statiques. À partir du résultat de cette routine et des vecteurs VECA et VORC actuels, un vecteur des latences dynamiques est établi pour l'instruction ordonnancée. Une routine d'évaluation de la latence dynamique de l'instruction à partir de son VDD permet de modifier l'attribut de l'instruction qui correspond à la latence dynamique. Une fois qu'une instruction est ordonnancée, sa latence dynamique devient disponible pour l'évaluation du critère d'ordonnancement de ses successeurs. De plus, une routine de mise à jour de la latence dynamique permet de calculer la latence dynamique actualisée à chaque itération.

Dans le but de calculer le critère d'ordonnancement pour les instructions disponible, il est nécessaire d'évaluer chacun des sous-critères qui le composent. Tout d'abord, la latence statique de l'instruction considérée est calculée à partir de son vecteur des délais statiques. Par la suite, une routine parcourt la liste des prédécesseurs de l'instruction considérée et retourne la latence dynamique actualisée maximale des prédécesseurs. Ensuite, une autre routine parcourt les 16 dernières instructions émises et retourne le nombre des prédécesseurs de l'instruction considérée parmi celles-ci. Finalement, une routine évalue la pénalité associée au non-support de l'instruction par l'unité fonctionnelle courante. L'implémentation de cet ensemble de routines ainsi que la fonction d'ordonnancement principale sont fournies en annexe.

4.2 Évaluation de l'approche d'ordonnancement

4.2.1 Environnement expérimental

L'évaluation de l'approche d'ordonnancement s'est faite par le biais du simulateur *OCTFIRE* fourni par OCTASIC. Ce simulateur permet d'obtenir le temps d'exécution ainsi que l'énergie

consommée au niveau des fils et des portes logiques. Le simulateur supporte la majorité des instructions ARM ISA opérant sur les entiers, cependant, il ne procure aucun support pour les extensions architecturales ARM ainsi que les opérations sur les données en point flottant. Dans le but de tester une application donnée, il est impératif d'inclure un programme d'initialisation et de finalisation requis par le simulateur. Ce programme comprend l'initialisation des registres, des drapeaux CPSR et le nettoyage de la mémoire cache à la fin du test.

Les programmes compilés avec l'approche d'ordonnancement proposée sont comparés à ceux issus de LLVM à deux niveaux d'optimisation ; -O0 (minimal) et -O3(maximal). À -O0, LLVM ne performe aucune optimisation d'arrière-plan. L'ordonnancement se limite dans ce cas à la linéarisation du GFD. Au niveau -O3, deux méthodes d'ordonnancement sont utilisées pour comparer l'approche proposée. La première est la méthode par défaut présentée dans la sous-section 4.1.1. Cette méthode utilise plusieurs heuristiques, dont la hauteur, d'instruction, le chemin critique et, en priorité, la pression sur les registres. La deuxième méthode est une approche d'ordonnancement aléatoire (Kouveli *et al.*, 2011) qui favorise le parallélisme au niveau des instructions. Cette méthode génère aléatoirement plusieurs échéanciers et choisit l'échéancier final selon une métrique qui correspond au niveau de parallélisme dans les instructions. Cette métrique représente le nombre d'instructions émises en étant complètement prêtes (après la fin prévue de l'exécution de l'ensemble de leurs prédécesseurs).

La compilation, au niveau -O3, des applications choisies se fait avec deux compilateurs LLVM. Le premier intègre l'approche d'ordonnancement en intégralité, le second n'intègre que la description du processeur cible, ce qui permet aux méthodes d'ordonnancement de LLVM d'acquérir l'information des latences et des unités fonctionnelles du AnARM. Le processus de compilation se fait en deux temps. Tout d'abord, l'outil *Clang* est appelé pour convertir le programme testé en langage intermédiaire. Par la suite, l'outil LLC est invoqué avec les paramètres nécessaires relatifs à l'ordonnancement des instructions.

Il est à noter que l'ensemble des méthodes d'ordonnancement évaluées, y compris celles de LLVM, est appelé avant et après l'allocation des registres. Ceci permet d'ordonnancer librement en l'absence des fausses dépendances et des dépendances de sortie avant l'allocation des registres. De plus, l'ordonnancement après l'allocation des registres permet d'ordonnancer une seconde fois en prenant en compte les nouvelles contraintes imposées par le passage aux registres physiques.

Un ensemble varié d'applications (*benchmarks*) en langage C, a été utilisé pour évaluer l'approche d'ordonnancement proposée dans ce mémoire. La premiere application choisie est le programme-test *Dhrystone*. *Dhrystone* (Weicker, 1984) est un programme de test de performance fréquemment utilisé dans l'évaluation des performances des processeurs *RISC*. La deuxième application est l'implémentation en C de la multiplication matricielle *IntMM* prise à partir de la suite d'applications-test de l'université de Stanford. La troisième application est une implémentation de la transformée de Fourier rapide à 1024 points (et son inverse). La quatrième application est un algorithme de tri à bulle provenant de la suite des programmes-tests de l'université de Stanford. La dernière application est une implémentation de la solution au problème combinatoire des N-reines. Ce programme est également issu de la suite des applications-tests de l'université de Stanford. Une présentation des programmes-tests utilisés est fournie dans le tableau 4.1. Les codes sources de ces applications sont fournis en annexe.

Tableau 4.1 Présentation des programmes-tests utilisés

Indice	**Application**	**Description**
#1	Dhrystone	Programme-test varié avec des opérations arithmétiques, manipulation des chaines de caractères et des pointeurs
#2	IntMM	Multiplication matricielle 40x40
#3	FFT	Transformée de Fourier rapide et son inverse (1024 points)
#4	Bubble	Algorithme de tri à bulles
#5	Queens	Solution au problème combinatoire des N-reines

4.2.2 Choix des paramètres de priorité

La comparaison des instructions, pendant le processus d'ordonnancement, se fait selon deux sous-critères contradictoires auxquels deux facteurs de priorité complémentaires ont été assignés. Dans le but de fixer les valeurs de ces facteurs de priorité, le programme *Dhrystone* a été choisi comme référence. Ce programme-test est l'unique application choisie pour cette fin, car *Dhrystone* est une application variée au niveau des instructions qui la composent. De plus, *Dhrystone* est assez représentatif du traitement des programmes opérant sur des nombres entiers (opérations arithmétiques, opérations sur les chaînes de caractères, transaction avec la mémoire, etc.). Finalement, pour des raisons pratiques (integration permanente de l'approche au compilateur), les valeurs choisies à l'issue de l'évaluation de *Dhrystone* seront fixées pour l'ensemble des tests effectués.

Dans le but de déceler les valeurs des paramètres de priorité w_1 et w_2, aboutissant à la meilleure performance en terme de temps d'exécution, un ensemble de 10 tests a été effectué en variant les valeurs respectives de w_1 et w_2 de 100%/0% à 0%/100%. La figure 4.5 illustre la variation du temps d'exécution de *Dhrystone* en fonction des différentes combinaisons des paramètres w_1 et w_2. Les valeurs de w_1 et w_2 aboutissant au meilleur temps d'exécution sont, respectivement, 30% et 70%. Par conséquent, prioriser le sous-critère (C2), qui privilégie le partage rapide des opérandes, procure une meilleure performance. Il est tout de même nécessaire de maintenir le poids du sous-critère favorisant le parallélisme (C1) au-delà de 20%, car une valeur plus basse entraînerait un plus grand nombre de blocages du pipeline dus au rapprochement des instructions liées par des dépendances.

4.2.3 Évaluation de l'approche au niveau d'optimisation minimal

L'évaluation de l'approche proposée au niveau d'optimisation -O0 est effectuée à travers la comparaison des programmes compilés au niveau -O0 et optimisés uniquement par l'approche d'ordonnancement proposée, aux programmes issus de la compilation -O0 d'un compilateur LLVM (non modifié). L'objectif de cette comparaison est d'évaluer l'apport de la méthode

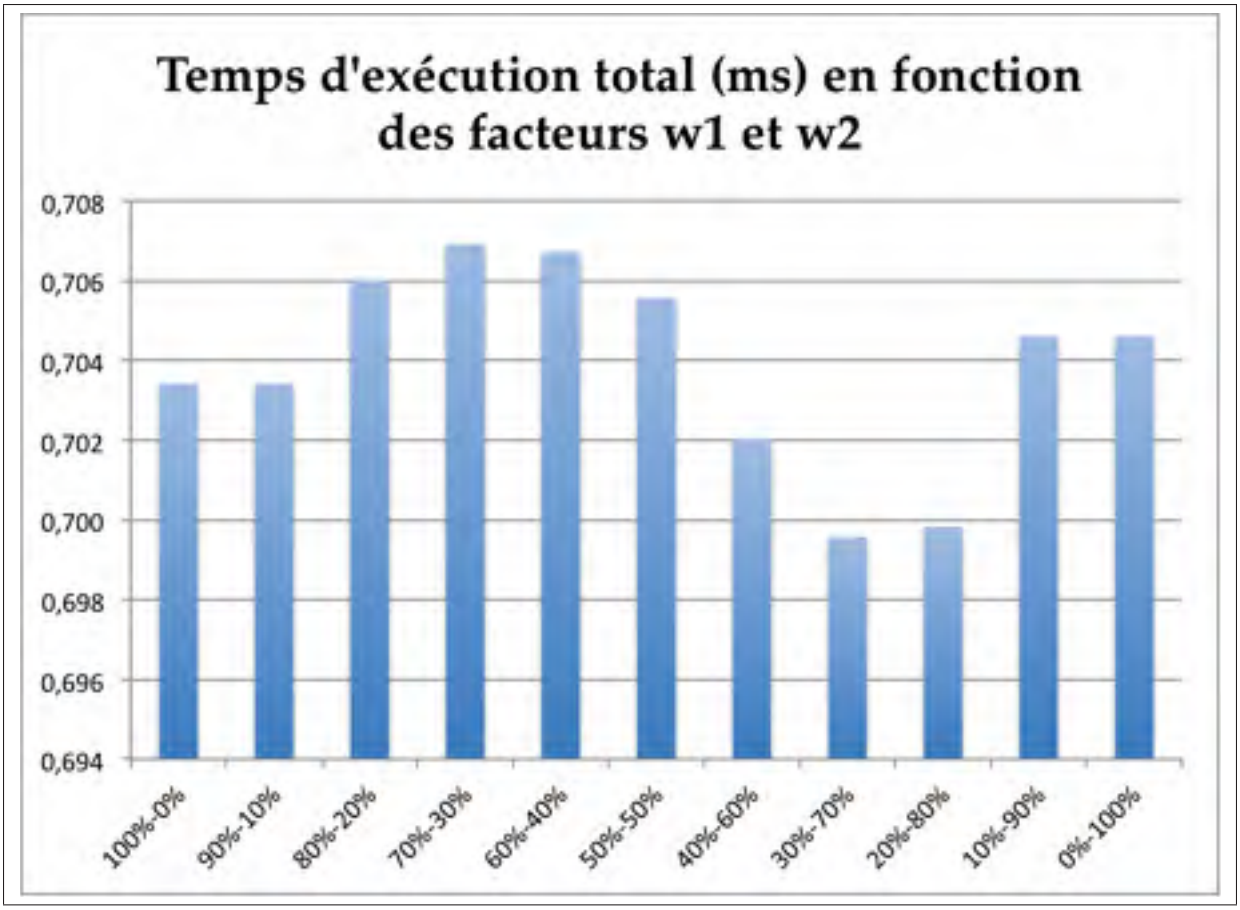


Figure 4.5 Temps d'exécution de *Dhrystone* en fonction des paramètres de priorité w_1 et w_2

d'ordonnancement développée dans ce mémoire en l'absence d'autres optimisations par rapport à un compilateur de base ("*vanilla*"). La figure 4.6 illustre la comparaison des temps d'exécution au niveau d'optimisation -O0. Les colonnes bleues représentent les temps d'exécution des exécutables issue de l'approche d'ordonnancement personnalisée (OP) tandis que les colonnes rouges représentent les temps d'exécution des exécutables issus de LLVM.

Le tableau 4.2 contient les pourcentages des améliorations, des temps d'exécution, induites par l'approche d'ordonnancement proposée, pour les différents programmes testés. L'approche d'ordonnancement proposée engendre une réduction moyenne du temps d'exécution de 17,48%

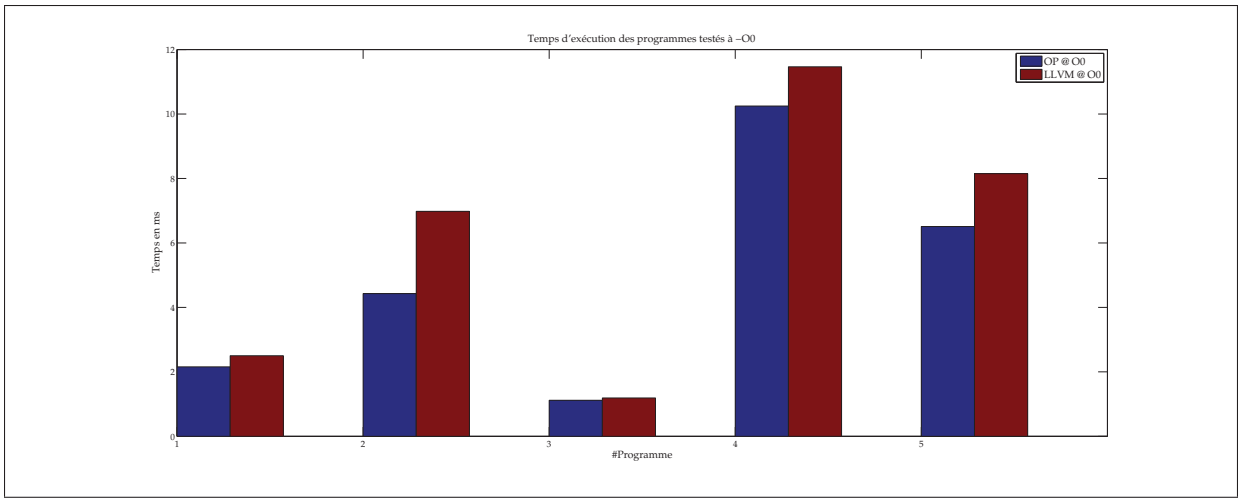


Figure 4.6 Comparaison du temps d'exécution à -O0

par rapport à LLVM, au niveau d'optimisation -O0.

Tableau 4.2 Amélioration du temps d'exécution à -O0

Indice	**Application**	**Réduction du temps d'exécution à -O0**
#1	Dhrystone	13,81%
#2	IntMM	36,54%
#3	FFT	6,12%
#4	Bubble	10,62%
#5	Queens	20,14%

La figure 4.7 illustre la comparaison en terme d'énergie consommée par les programmes testés. Les pourcentages des réductions de l'énergie consommée sont présentés dans le tableau 4.3.

Tableau 4.3 Réduction de la consommation d'énergie à -O0

Indice	**Application**	**Réduction de la consommation d'énergie à -O0**
#1	Dhrystone	16,29%
#2	IntMM	23,01%
#3	FFT	6,99%
#4	Bubble	17,84%
#5	Queens	20,34%

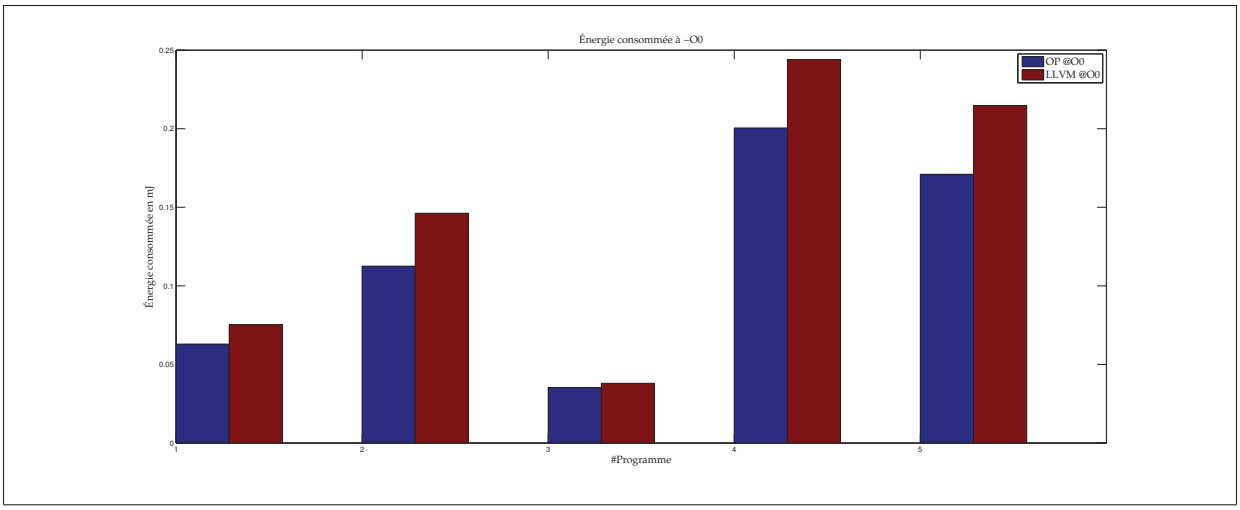


Figure 4.7 Comparaison de l'énergie consommée à -O0

L'approche d'ordonnancement proposée engendre une réduction moyenne de 16,89%. Ces résultats montrent que l'approche d'ordonnancement, proposée dans ce mémoire, induit des améliorations considérables en termes de temps d'exécution et de consommation d'énergie par rapport à un compilateur de base. Ces résultats démontrent qu'il est possible d'améliorer les performances ainsi que la consommation d'énergie du AnARM à travers un ordonnancement basé sur son comportement architectural. Ces améliorations permettent d'accroître l'atout principal du AnARM (son rendement énergétique) tout en le rendant plus compétitif au niveau des performances de calcul par rapport aux processeurs synchrones de la même catégorie.

4.2.4 Évaluation de l'approche au niveau d'optimisation maximal

L'évaluation de l'approche proposée, au niveau d'optimisation-O3, permet de la comparer aux méthodes existantes implémentées sur LLVM en présence de l'ensemble des optimisations activées à-O3. Dans le cadre de cette évaluation, les méthodes d'ordonnancement de LLVM utilisent la même description de l'AnARM que l'approche personnalisée. Ceci permet à ces méthodes d'ordonnancer les instructions en prenant en compte leurs latences ainsi que la configuration des unités fonctionnelles de l'AnARM. La comparaison entre les différentes approches en terme de temps d'exécution est illustrée dans la figure 4.8. Les temps d'exécutions des pro-

grammes issus de l'approche personnalisée (OP) sont représentés par les colonnes bleues, ceux issus de l'approche d'ordonnancement principale de LLVM (LLVM) sont représentés par les colonnes vertes et ceux issus de l'approche d'ordonnancement aléatoire favorisant le parallélisme (ILP) sont représentés par les colonnes rouges.

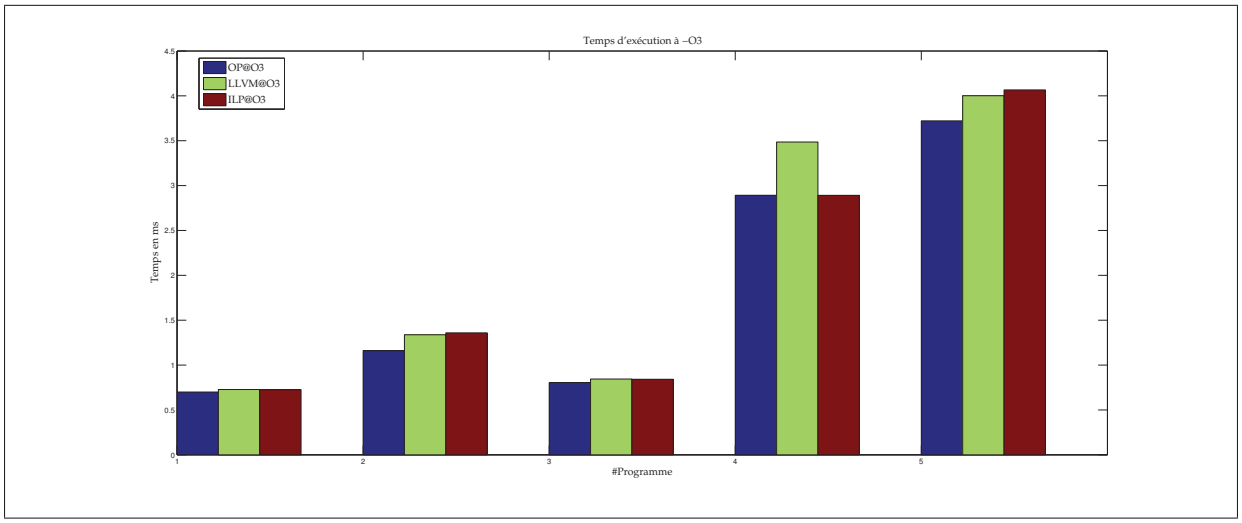


Figure 4.8 Comparaison des temps d'exécution à -O3

Les pourcentages de l'amélioration des temps d'exécution par rapport aux méthodes d'ordonnancement de LLVM sont présentés dans le tableau 4.4. La comparaison de la consommation

Tableau 4.4 Amélioration du temps d'exécution à -O3

Indice	**Application**	**Amélioration VS LLVM**	**Amélioration VS ILP**
#1	Dhrystone	3,85%	3,65%
#2	IntMM	13,21%	14,59%
#3	FFT	4,7%	4,42%
#4	Bubble	17,02%	-0,02%
#5	Queens	7%	8,42%
	Moyenne	9,15%	6,22%

d'énergie est illustrée dans la figure 4.9. Les pourcentages des réductions de la consommation d'énergie par rapport aux méthodes d'ordonnancement de LLVM sont présentés au tableau 4.5.

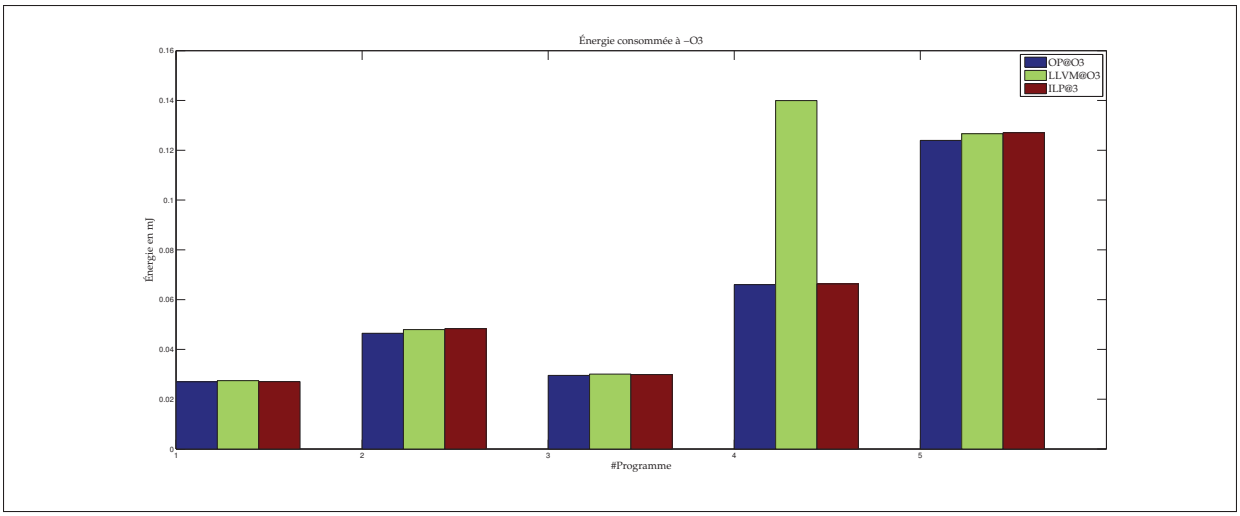


Figure 4.9 Comparaison de l'énergie consommée à -O3

Tableau 4.5 Réduction de la consommation d'énergie à -O3

Indice	Application	Réduction VS LLVM	Réduction VS ILP
#1	Dhrystone	1,62%	1,6%
#2	IntMM	3,09%	3,6%
#3	FFT	1,79%	1,13%
#4	Bubble	52,82%	0,42%
#5	Queens	2,11%	2,43%
	Moyenne	12,29%	1,59%

L'approche d'ordonnancement proposée performe mieux que les deux méthodes d'ordonnancement de LLVM. En terme de temps d'exécution, l'approche d'ordonnancement personnalisée engendre des améliorations plus amples comparées à l'approche de LLVM basée sur la pression sur les registres, le chemin critique et la hauteur d'instruction (9,15% en moyenne). Les performances issues de l'approche d'ordonnancement personnalisée sont en moyenne 6,22% meilleures que celles issues de l'approche d'ordonnancement ILP. En terme de consommation d'énergie, les gains induits par l'approche proposée sont faibles sauf dans le cas du programme *Bubble* issue de l'ordonnancement principal de LLVM.

Les résultats obtenus indiquent qu'une approche favorisant exclusivement le parallélisme au niveau des instructions génère des échéanciers assez performants, comparativement à une approche dont le principal critère est la réduction de la pression sur les registres. Les résultats obtenus lors de la détermination des paramètres de priorité pour le critère d'ordonnancement personnalisé confirment cette assertion. En effet, l'exécution de *Dhrystone* est plus rapide quand w1 est à 100% ou 90% que quand il est à 10% ou 0%. Cependant, l'approche personnalisée génère des échéanciers plus courts pour la majorité des programmes testés. Par conséquent, la recherche d'un équilibre entre le parallélisme et le partage rapide des résultats permet d'aboutir aux meilleures performances. En conclusion, l'approche proposée dans ce mémoire induit une accélération modérée de l'exécution des programmes testés par rapport aux méthodes d'ordonnancement de LLVM, opérant avec la même description du processeur cible, à un haut niveau d'optimisation.

4.3 Conclusion

Dans ce chapitre, l'environnement d'implémentation ainsi que la démarche expérimentale ont été présentés. Le choix de la plateforme d'implémentation s'est porté sur LLVM ; un compilateur moderne, performant et modulable. L'ordonnancement des instructions dans LLVM repose sur des descriptions architecturales des processeurs cibles permettant l'acquisition des données d'ordonnancement. LLVM implémente des heuristiques d'ordonnancement puissantes et rapides, qui prennent en considération les latences d'instructions, l'occupation des ressources et les contraintes imposées par les registres. La première étape de l'implémentation de l'approche proposée a été d'introduire la description architecturale de l'AnARM au niveau des latences des instructions et des ressources de calculs. Par la suite, l'ensemble des outils permettant l'ordonnancement, proposé dans mémoire, a été intégré à LLVM.

L'évaluation de l'effet de la variation des paramètres d'ordonnancement sur la qualité des échéanciers produits a permis de déceler la combinaison des paramètres offrant la meilleure performance pour une application-référence. Par la suite, l'approche a été évaluée compara-

tivement aux approches utilisées par le compilateur dans le même contexte architectural. Les résultats obtenus ont permis de constater les apports de l'approche proposée et de comparer les méthodes utilisées par le compilateur en rapport avec l'architecture à l'étude. La méthode d'ordonnancement proposée a engendré des améliorations moyennes des performances de 9,15% par rapport à l'ordonnancement de LLVM et de 6,22% par rapport à l'approche ILP.

Le prochain chapitre conclura ce mémoire en présentant une synthèse des travaux, les limitations de l'approche proposée, les recommandations et les possibilités de travaux futurs.

CONCLUSION ET RECOMMANDATIONS

Dans ce chapitre, une récapitulation des travaux présentés dans ce mémoire est fournie. Les limites et les difficultés rencontrées dans le cadre du projet seront exposées. Finalement, une réflexion sur les possibilités de travaux futurs sera présentée dans le but de conclure ce mémoire.

Synthèse des travaux :

Ce projet de recherche s'inscrit dans le cadre du projet AnARM visant le développement d'un processeur à usage général ARM basé sur une architecture endochrone, inspirée des processeurs de traitement numérique du signal d'OCTASIC. Ce projet fait partie du volet portant sur l'optimisation énergétique de l'AnARM. L'objectif principal du projet décrit dans ce mémoire consiste à élaborer et évaluer une stratégie d'ordonnancement des instructions permettant de tirer profit de certaines spécificités architecturales de l'AnARM dans le but d'en améliorer les performances.

Les processeurs endochrones se distinguent par un ensemble de mécanismes de synchronisation permettant une exécution valide des applications, en dépit de l'absence d'un signal global de synchronisation (horloge). L'AnARM utilise un système de synchronisation point à point utilisant des signaux locaux, permettant de respecter les contraintes temporelles et de maintenir la validité des données. Ces signaux parcourent des circuits de délai correspondant aux différentes opérations de traitement du processeur, ce qui permet un délai de traitement sur mesure pour chacune de ces opérations. Additionnellement, l'AnARM est un processeur à plusieurs unités d'exécution ce qui permet le traitement en parallèle de plusieurs instructions. Cependant, l'absence d'un signal de synchronisation globale est compensée par un mécanisme de jetons permettant de définir un ordre d'accès aux ressources partagées et d'éviter les conflits entre les unités d'exécution. Le mécanisme des jetons engendre des dépendances au niveau de

plusieurs étapes du pipeline. Finalement, l'AnARM dispose d'un mécanisme de transmission des résultats de calcul entre les différentes unités d'exécution ce qui leur permet, sous certaines conditions, d'acquérir rapidement les opérandes qu'elles requièrent.

La stratégie d'ordonnancement élaborée dans ce mémoire se base sur la dynamique du comportement spatio-temporel du AnARM, dans le but d'estimer le coût relié à l'ordonnancement d'une instruction donnée à un instant donné. Cette stratégie s'étale sur deux axes opposés d'optimisation. Le premier axe concerne la réduction des blocages du pipeline dus aux dépendances de données et l'augmentation du parallélisme au niveau des instructions. Le second axe porte sur l'augmentation du taux d'utilisation du système de transmission des résultats de calcul.

L'implémentation de la stratégie d'ordonnancement proposée a été effectuée à travers l'intégration d'une description architecturale de l'AnARM au compilateur LLVM ainsi que l'adaptation de l'algorithme d'ordonnancement principal de LLVM au modèle d'ordonnancement dynamique proposé.

Une évaluation de l'influence des paramètres du critère d'ordonnancement a permis de trouver un équilibre au niveau des deux axes d'optimisation. Cet équilibre s'est traduit par une amélioration moyenne de 17,48% des performances par rapport à une compilation standard et une amélioration des performances supérieure à 6% comparativement à deux méthodes d'ordonnancement utilisées par le compilateur, à un haut niveau d'optimisation. De plus, ces améliorations des performances n'engendrent aucun coût énergétique supplémentaire. Dans la majorité des cas évalués, l'approche d'ordonnancement proposée induit une réduction de la consommation énergétique.

La principale contribution de ce mémoire est le développement ainsi que l'implémentation d'une approche d'ordonnancement dynamique permettant l'amélioration des performances de l'AnARM. L'approche développée prouve qu'il est possible d'améliorer les performances d'une architecture endochrone en utilisant des optimisations logicielles. Ces améliorations permettent de réduire l'écart de performance entre ce type d'architectures et les architectures synchrones sans compromettre l'avantage essentiel des processeurs endochrones : leur faible consommation énergétique.

Limites des travaux :

Les travaux présentés dans ce mémoire présentent certaines limitations. Tout d'abord, le modèle d'estimation des délais statiques est d'une précision limitée, à cause de la démarche visant à discretiser les valeurs temporelles des délais de traitement. Ce modèle a été défini à partir du modèle développé par Tremblay (2009) et d'une analyse temporelle fournie par Feriel Ben Abdellah, une stagiaire post doctorale travaillant sur un projet connexe. Ce modèle est assez simple et ne prend pas en considération certaines variations pouvant être causées par l'échec des accès à la mémoire cache ou l'influence de paramètres physiques, tels que la variation de vitesse d'opération due à la température. Ce modèle pourrait être affiné, à l'aide d'une analyse temporelle plus élaborée d'un prototype AnARM opérant sous des conditions réelles. Des valeurs moyennes précises pourraient être assignées à chaque étape de traitement.

Additionnellement, le modèle d'estimation proposé ne traite pas les instructions opérant sur le flot de contrôle (branchement, saut de programme, etc.). Le comportement du processeur au niveau des sauts de programme est difficile à prévoir au moment de la compilation. Pour cette raison, aucune approche d'ordonnancement global utilisant des informations dynamiques n'a été implémentée. À un haut niveau d'optimisation, l'ordonnancement global est exclusivement géré par les algorithmes de LLVM, qui définissent des régions d'ordonnancement global à par-

tir de la fréquence estimée des blocs de base. Une meilleure analyse des sauts de programme en rapport avec les mécanismes de synchronisation permettrait de définir une méthode d'ordonnancement global basée sur les caractéristiques de l'AnARM.

La principale difficulté rencontrée pendant le projet concerne la phase de l'évaluation de l'approche. Le simulateur fourni par OCTASIC ne couvre pas l'ensemble du jeu d'instruction des processeurs ARM. Les simulations de certains programmes ont échoué, à cause de la présence de pseudo-instructions ou d'instructions non supportées. Ceci a limité le processus d'évaluation à un nombre restreint d'applications-test.

La détermination des paramètres de priorité du critère d'ordonnancement a été basée uniquement sur l'application *Dhrystone*. Il serait plus judicieux de déceler cette combinaison de paramètres pour chaque application dans le but d'obtenir les meilleures performances. Cependant, ce procédé nécessiterait la reconstruction du compilateur pour chaque application, ce qui ne serait pas pratique.

Travaux futurs :

La première possibilité de travaux futurs consiste à analyser le comportement de l'AnArm au niveau du flot de contrôle dans le but de développer une optimisation globale permettant d'améliorer les performances. Un second horizon d'optimisation consiste à évaluer les méthodes d'ordonnancement optimales. Ce processus peut s'avérer complexe en raison des contraintes additionnelles qui engendrent un flot d'exécution fragmenté, dans le cas de l'AnARM. Cette évaluation permettrait d'analyser la praticabilité ainsi que l'apport des méthodes combinatoires par rapport aux méthodes heuristiques. En troisième lieu, une méthode d'ordonnancement visant l'amélioration des performances énergétiques de l'AnARM pourrait être développée, à travers une analyse énergétique de plusieurs heuristiques d'ordonnancement. Finalement, l'écri-

ture d'une description complète de l'AnARM au sein d'un compilateur commercial permettrait l'adaptation de l'ensemble des optimisations d'arrière-plan à cette architecture.

ANNEXE I

CODE SOURCE DE L'APPROCHE D'ORDONNANCEMENT

Extrait I.1 Algorithme d'ordonnacement personalisé

```
//
//  anArmPass.cpp
//
// Algorithme d'ordonnacement implemente a partir de PostRASched.cpp de LLVM
//
//  Created by Hamza Halli on 2015-03-07.
//
//

#define DEBUG_TYPE "AnArmSched"

#include "llvm/CodeGen/MachineScheduler.h"
#include "llvm/ADT/PriorityQueue.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/CodeGen/LiveIntervalAnalysis.h"
#include "llvm/CodeGen/MachineDominators.h"
#include "llvm/CodeGen/MachineLoopInfo.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/CodeGen/RegisterClassInfo.h"
#include "llvm/CodeGen/ScheduleDFS.h"
#include "llvm/CodeGen/ScheduleHazardRecognizer.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/GraphWriter.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetInstrInfo.h"
#include "llvm/CodeGen/Passes.h"
#include "/Users/hamzahalli/Desktop/projet/llvm/lib/CodeGen/AggressiveAntiDepBreaker.h"
#include "/Users/hamzahalli/Desktop/projet/llvm/lib/CodeGen/AntiDepBreaker.h"
#include "/Users/hamzahalli/Desktop/projet/llvm/lib/CodeGen/CriticalAntiDepBreaker.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/CodeGen/LatencyPriorityQueue.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/ScheduleDAG.h"
#include "llvm/CodeGen/ScheduleDAGInstrs.h"
#include "llvm/CodeGen/SchedulerRegistry.h"
#include "llvm/Target/TargetLowering.h"
#include "llvm/Target/TargetRegisterInfo.h"
#include "llvm/Target/TargetSubtargetInfo.h"

#include "ARM.h"
#include <queue>
using namespace llvm;
STATISTIC(NumNoops, "Number of noops inserted");
STATISTIC(NumStalls, "Number of pipeline stalls");
STATISTIC(NumFixedAnti, "Number of fixed anti-dependencies");
```

```
namespace {
    class anArmPass : public MachineFunctionPass
    {       const TargetInstrInfo *TII;
        RegisterClassInfo RegClassInfo;
        public :

        static char ID ;
        anArmPass(): MachineFunctionPass(ID){}
        bool enablePostRAScheduler(
                                const TargetSubtargetInfo &ST, CodeGenOpt::Level OptLevel,
                                TargetSubtargetInfo::AntiDepBreakMode &Mode,
                                TargetSubtargetInfo::RegClassVector &CriticalPathRCs) const;
        bool runOnMachineFunction(MachineFunction &Fn) override;
//aquisition de l'environnement de la passe
        void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
            AU.addRequired<AliasAnalysis>();
            AU.addRequired<TargetPassConfig>();
            AU.addRequired<MachineDominatorTree>();
            AU.addPreserved<MachineDominatorTree>();
            AU.addRequired<MachineLoopInfo>();
            AU.addPreserved<MachineLoopInfo>();
            MachineFunctionPass::getAnalysisUsage(AU);
        }


    };

    class anArmSched : public ScheduleDAGInstrs {
        /// AvailableQueue - The priority queue to use for the available SUnits.
        ///
        LatencyPriorityQueue AvailableQueue;

        /// PendingQueue - This contains all of the instructions whose operands have
        /// been issued, but their results are not ready yet (due to the latency of
        /// the operation).  Once the operands becomes available, the instruction is
        /// added to the AvailableQueue.
        std::vector<SUnit*> PendingQueue;

        /// HazardRec - The hazard recognizer to use.
        ScheduleHazardRecognizer *HazardRec;

        /// AntiDepBreak - Anti-dependence breaking object, or NULL if none
        AntiDepBreaker *AntiDepBreak;

        /// AA - AliasAnalysis for making memory reference queries.
        AliasAnalysis *AA;

        /// The schedule. Null SUnit*'s represent noop instructions.
        std::vector<SUnit*> Sequence;

        /// The index in BB of RegionEnd.
        ///
        /// This is the instruction number from the top of the current block, not
        /// the SlotIndex. It is only used by the AntiDepBreaker.
        unsigned EndIndex;

    public:
        anArmSched(
```

```
                            MachineFunction &MF, MachineLoopInfo &MLI, AliasAnalysis *AA,
                            const RegisterClassInfo &,
                            TargetSubtargetInfo::AntiDepBreakMode AntiDepMode,
                            SmallVectorImpl<const TargetRegisterClass *> &CriticalPathRCs);

    ~anArmSched();

    /// startBlock - Initialize register live-range state for scheduling in
    /// this block.
    ///
    void startBlock(MachineBasicBlock *BB) override;

    // Set the index of RegionEnd within the current BB.
    void setEndIndex(unsigned EndIdx) { EndIndex = EndIdx; }

    /// Initialize the scheduler state for the next scheduling region.
    void enterRegion(MachineBasicBlock *bb,
                     MachineBasicBlock::iterator begin,
                     MachineBasicBlock::iterator end,
                     unsigned regioninstrs) override;

    /// Notify that the scheduler has finished scheduling the current region.
    void exitRegion() override;
    int computeC (SUnit *su, std::vector<SUnit*> sequence, unsigned EU, unsigned iter, const InstrItineraryData* itin);
   unsigned * VDS( SUnit* SU, const InstrItineraryData* itin);
    unsigned* VDD( SUnit* SU, unsigned* VECA, unsigned* VORC, unsigned EU,const InstrItineraryData* itin);
    unsigned DL (unsigned * VDD);
    bool isSupported( SUnit* SU, unsigned EU);
    void updateVECA (unsigned* VECA , unsigned * VDD);
    void updateVORC(unsigned* VECA , unsigned * VDD, unsigned EU);
    unsigned updatedDL (unsigned DL, unsigned iter);
    unsigned penalty (SUnit* SU, unsigned EU,const InstrItineraryData* itin);
    void updateEU ( unsigned EU);
    unsigned SL( SUnit* SU,const InstrItineraryData* itin);
    unsigned C2(SUnit* SU, std::vector<SUnit*> sequence);
    unsigned maxPredDL(SUnit* SU, unsigned iter);
    void swap(std::vector<SUnit*>  data, int i, int j)
    {
        SUnit tmp = *data[i];
        data[i] = data[j];
        data[j] = &tmp;
    }
void SortAvailable(std::vector<SUnit*>  data, std::vector<SUnit*> sequence
                  ,unsigned EU, unsigned iter, const InstrItineraryData* itin)
    {
        int length = data.size();

        for (int i = 0; i < length; ++i)
        {
            int min = i;
            for (int j = i+1; j < length; ++j)
            {
                if (computeC(data[j], sequence, EU, iter, itin) < computeC (data[min], sequence, EU, iter, itin))
                {
                    min = j;
                }
            }

            if (min != i)
            {
```

```
                    swap(data, i, min);
                }
            }
        }

        /// Schedule - Schedule the instruction range using list scheduling.
        ///
        void schedule() override;

        void EmitSchedule();

        /// Observe - Update liveness information to account for the current
        /// instruction, which will not be scheduled.
        ///
        void Observe(MachineInstr *MI, unsigned Count);

        /// finishBlock - Clean up register live-range state.
        ///
        void finishBlock() override;

    private:
        void ReleaseSucc(SUnit *SU, SDep *SuccEdge);
        void ReleaseSuccessors(SUnit *SU);
        void ScheduleNodeTopDown(SUnit *SU, unsigned CurCycle);
        void ListScheduleTopDown();

        void dumpSchedule() const;
        void emitNoop(unsigned CurCycle);
    };


}


int anArmSched :: computeC (SUnit *su, std::vector<SUnit*> sequence, unsigned EU, unsigned iter, const InstrItineraryData* itin)

{ int score;

    score= 30*(SL(su, itin)-maxPredDL(su, iter))+70*C2(su, sequence) - 100* penalty(su, EU, itin);
    return score;}


anArmSched::anArmSched(
                                        MachineFunction &MF, MachineLoopInfo &MLI, AliasAnalysis *AA,
                                        const RegisterClassInfo &RCI,
                                        TargetSubtargetInfo::AntiDepBreakMode AntiDepMode,
                                        SmallVectorImpl<const TargetRegisterClass *> &CriticalPathRCs)
: ScheduleDAGInstrs(MF, &MLI, /*IsPostRA=*/true), AA(AA), EndIndex(0) {

    const InstrItineraryData *InstrItins =
    MF.getSubtarget().getInstrItineraryData();
    HazardRec =
    MF.getSubtarget().getInstrInfo()->CreateTargetPostRAHazardRecognizer(
                                                                    InstrItins, this);

    assert((AntiDepMode == TargetSubtargetInfo::ANTIDEP_NONE ||
            MRI.tracksLiveness()) &&
           "Live-ins must be accurate for anti-dependency breaking");
    AntiDepBreak =
    ((AntiDepMode == TargetSubtargetInfo::ANTIDEP_ALL) ?
```

```
      (AntiDepBreaker *)new AggressiveAntiDepBreaker(MF, RCI, CriticalPathRCs) :
      ((AntiDepMode == TargetSubtargetInfo::ANTIDEP_CRITICAL) ?
       (AntiDepBreaker *)new CriticalAntiDepBreaker(MF, RCI) : nullptr));
}

anArmSched::~anArmSched() {
    delete HazardRec;
    delete AntiDepBreak;
}

void anArmSched::enterRegion(MachineBasicBlock *bb,
                                    MachineBasicBlock::iterator begin,
                                    MachineBasicBlock::iterator end,
                                    unsigned regioninstrs) {
    ScheduleDAGInstrs::enterRegion(bb, begin, end, regioninstrs);
    Sequence.clear();
}

void anArmSched::exitRegion() {
    DEBUG({
        dbgs() << "***␣Final␣schedule␣***\n";
        dumpSchedule();
        dbgs() << '\n';
    });
    ScheduleDAGInstrs::exitRegion();
}

#if !defined(NDEBUG) || defined(LLVM_ENABLE_DUMP)
/// dumpSchedule - dump the scheduled Sequence.
void anArmSched::dumpSchedule() const {
    for (unsigned i = 0, e = Sequence.size(); i != e; i++) {
        if (SUnit *SU = Sequence[i])
            SU->dump(this);
        else
            dbgs() << "****␣NOOP␣****\n";
    }
}
#endif

bool anArmPass::enablePostRAScheduler(
                                          const TargetSubtargetInfo &ST,
                                          CodeGenOpt::Level OptLevel,
                                          TargetSubtargetInfo::AntiDepBreakMode &Mode,
                                          TargetSubtargetInfo::RegClassVector &CriticalPathRCs) const {
        Mode = ST.getAntiDepBreakMode();
        ST.getCriticalPathRCs(CriticalPathRCs);
        return ST.enablePostMachineScheduler() &&
        OptLevel >= ST.getOptLevelToEnablePostRAScheduler();
    }


/// StartBlock - Initialize register live-range state for scheduling in
/// this block.
///
void anArmSched::startBlock(MachineBasicBlock *BB) {
    // Call the superclass.
    ScheduleDAGInstrs::startBlock(BB);

    // Reset the hazard recognizer and anti-dep breaker.
    HazardRec->Reset();
```

```
    if (AntiDepBreak)
        AntiDepBreak->StartBlock(BB);
}


/// Schedule - Schedule the instruction range using list scheduling.
///
void anArmSched::schedule() {
    // Build the scheduling graph.
    buildSchedGraph(AA);

    if (AntiDepBreak) {
        unsigned Broken =
        AntiDepBreak->BreakAntiDependencies(SUnits, RegionBegin, RegionEnd,
                                            EndIndex, DbgValues);

        if (Broken != 0) {
            // We made changes. Update the dependency graph.
            // Theoretically we could update the graph in place:
            // When a live range is changed to use a different register, remove
            // the def's anti-dependence *and* output-dependence edges due to
            // that register, and add new anti-dependence and output-dependence
            // edges based on the next live range of the register.
            ScheduleDAG::clearDAG();
            buildSchedGraph(AA);

            NumFixedAnti += Broken;
        }
    }

    DEBUG(dbgs() << "**********␣anArm␣Scheduling␣**********\n");
    DEBUG(for (unsigned su = 0, e = SUnits.size(); su != e; ++su)
          SUnits[su].dumpAll(this));

    AvailableQueue.initNodes(SUnits);
    ListScheduleTopDown();
    AvailableQueue.releaseState();
}

/// Observe - Update liveness information to account for the current
/// instruction, which will not be scheduled.
///
void anArmSched::Observe(MachineInstr *MI, unsigned Count) {
    if (AntiDepBreak)
        AntiDepBreak->Observe(MI, Count, EndIndex);
}

/// FinishBlock - Clean up register live-range state.
///
void anArmSched::finishBlock() {
    if (AntiDepBreak)
        AntiDepBreak->FinishBlock();

    // Call the superclass.
    ScheduleDAGInstrs::finishBlock();
}
/// ReleaseSucc - Decrement the NumPredsLeft count of a successor. Add it to
/// the PendingQueue if the count reaches zero.
void anArmSched::ReleaseSucc(SUnit *SU, SDep *SuccEdge) {
    SUnit *SuccSU = SuccEdge->getSUnit();
```

```
    if (SuccEdge->isWeak()) {
      --SuccSU->WeakPredsLeft;
      return;
    }
#ifndef NDEBUG
    if (SuccSU->NumPredsLeft == 0) {
      dbgs() << "*** Scheduling failed! ***\n";
      SuccSU->dump(this);
      dbgs() << " has been released too many times!\n";
      llvm_unreachable(nullptr);
    }
#endif
    --SuccSU->NumPredsLeft;

    // Standard scheduler algorithms will recompute the depth of the successor
    // here as such:
    //   SuccSU->setDepthToAtLeast(SU->getDepth() + SuccEdge->getLatency());
    //
    // However, we lazily compute node depth instead. Note that
    // ScheduleNodeTopDown has already updated the depth of this node which causes
    // all descendents to be marked dirty. Setting the successor depth explicitly
    // here would cause depth to be recomputed for all its ancestors. If the
    // successor is not yet ready (because of a transitively redundant edge) then
    // this causes depth computation to be quadratic in the size of the DAG.

    // If all the node's predecessors are scheduled, this node is ready
    // to be scheduled. Ignore the special ExitSU node.
    if (SuccSU->NumPredsLeft == 0 && SuccSU != &ExitSU)
      PendingQueue.push_back(SuccSU);
}
/// ReleaseSuccessors - Call ReleaseSucc on each of SU's successors.
void anArmSched::ReleaseSuccessors(SUnit *SU) {
    for (SUnit::succ_iterator I = SU->Succs.begin(), E = SU->Succs.end();
         I != E; ++I) {
      ReleaseSucc(SU, &*I);
    }
}
/// ScheduleNodeTopDown - Add the node to the schedule. Decrement the pending
/// count of its successors. If a successor pending count is zero, add it to
/// the Available queue.
void anArmSched::ScheduleNodeTopDown(SUnit *SU, unsigned CurCycle) {
    DEBUG(dbgs() << "*** Scheduling [" << CurCycle << "]: ");
    DEBUG(SU->dump(this));

    Sequence.push_back(SU);
    assert(CurCycle >= SU->getDepth() &&
           "Node scheduled above its depth!");
    SU->setDepthToAtLeast(CurCycle);

    ReleaseSuccessors(SU);
    SU->isScheduled = true;
    AvailableQueue.scheduledNode(SU);
}
void anArmSched::emitNoop(unsigned CurCycle) {
    DEBUG(dbgs() << "*** Emitting noop in cycle " << CurCycle << '\n');
    HazardRec->EmitNoop();
    Sequence.push_back(nullptr);   // NULL here means noop
    ++NumNoops;
}
```

```
bool anArmSched::isSupported( SUnit* SU, unsigned EU)
{MachineInstr * MI = SU->getInstr();
    bool t = true;

    if(EU%2 ==0){

      switch (MI->getOpcode())

    { case 209://ARM::MLA:
        case 210://ARM::MLAv5:
        case 211://ARM::MLS:
        case 244://ARM::MUL:
        case 245:// ARM::MULv5:
        case 317://ARM::SDIV:

        case 337://ARM::SMLABB:
        case 338:// ARM::SMLABT:
        case 339:// ARM::SMLAD:
        case 340:// ARM::SMLADX:
        case 341:// ARM::SMLAL:
        case 342:// ARM::SMLALBB:
        case 343:// ARM::SMLALBT:
        case 344:// ARM::SMLALD:
        case 345:// ARM::SMLALDX:
        case 346:// ARM::SMLALTB:
        case 347:// ARM::SMLALTT:
        case 348:// ARM::SMLALv5:
        case 349:// ARM::SMLATB:
        case 350:// ARM::SMLATT:
        case 351:// ARM::SMLAWB:
        case 352:// ARM::SMLAWT:
        case 353:// ARM::SMLSD:
        case 354:// ARM::SMLSDX:
        case 355:// ARM::SMLSLD:
        case 356:// ARM::SMLSLDX:
        case 357:// ARM::SMMLA:
        case 358:// ARM::SMMLAR:
        case 359:// ARM::SMMLS:
        case 360:// ARM::SMMLSR:
        case 361:// ARM::SMMUL:
        case 362:// ARM::SMMULR:
        case 363:// ARM::SMUAD:
        case 364:// ARM::SMUADX:
        case 365:// ARM::SMULBB:
        case 366:// ARM::SMULBT:
        case 367:// ARM::SMULL:
        case 368:// ARM::SMULLv5:
        case 369:// ARM::SMULTB:
        case 370:// ARM::SMULTT:
        case 371:// ARM::SMULWB:
        case 372:// ARM::SMULWT:
        case 373:// ARM::SMUSD:
        case 374:// ARM::SMUSDX:
        case 500://ARM::UMAAL:
        case 501:// ARM::UMLAL:
        case 502:// ARM::UMLALv5:
        case 503:// ARM::UMULL:
        case 504:// ARM::UMULLv5:
        case 493:// ARM::UDIV:
            t = false;
```

```
            break;
}}
else{

    switch (MI->getOpcode())

    {
    case 389://ARM::STC2_OFFSET:
    case 390:// ARM::STC2_OPTION:
    case 391:// ARM::STC2_POST:
    case 392:// ARM::STC2_PRE:
    case 393:// ARM::STC2L_OFFSET:
    case 394:// ARM::STC2L_OPTION:
    case 395:// ARM::STC2L_PRE:
    case 396:// ARM::STC2L_POST:
    case 397:// ARM::STC_OFFSET:
    case 398:// ARM::STC_OPTION:
    case 399:// ARM::STC_POST:
    case 400:// ARM::STC_PRE:
    case 401:// ARM::STCL_OFFSET:
    case 402:// ARM::STCL_OPTION:
    case 403:// ARM::STCL_POST:
    case 404:// ARM::STCL_PRE:
    case 405:// ARM::STL:
    case 406:// ARM::STLB:
    case 407:// ARM::STLEX:
    case 408:// ARM::STLEXB:
    case 409:// ARM::STLEXD:
    case 410:// ARM::STLEXH:
    case 411:// ARM::STLH:
    case 420:// ARM::STR_POST_IMM:
    case 421:// ARM::STR_POST_REG:
    case 422:// ARM::STR_PRE_IMM:
    case 423:// ARM::STR_PRE_REG:
    case 424:// ARM::STRB_POST_IMM:
    case 425:// ARM::STRB_POST_REG:
    case 426:// ARM::STRB_PRE_IMM:
    case 427:// ARM::STRB_PRE_REG:
    case 428:// ARM::STRBi12:
    case 429:// ARM::STRBi_preidx:
    case 430:// ARM::STRBr_preidx:
    case 431:// ARM::STRBrs:
    case 432:// ARM::STRBT_POST:
    case 433:// ARM::STRBT_POST_IMM:
    case 434:// ARM::STRBT_POST_REG:
    case 435:// ARM::STRD:
    case 436:// ARM::STRD_POST:
    case 437:// ARM::STRD_PRE:
    case 438:// ARM::STREX:
    case 439:// ARM::STREXB:
    case 440:// ARM::STREXD:
    case 441:// ARM::STREXH:
    case 442:// ARM::STRH:
    case 443:// ARM::STRH_POST:
    case 444:// ARM::STRH_PRE:
    case 445:// ARM::STRH_preidx:
    case 446:// ARM::STRHTi:
    case 447:// ARM::STRHTr:
    case 448:// ARM::STRi12:
    case 449:// ARM::STRi_preidx:
```

```
case 450: // ARM::STRr_preidx:
case 451: // ARM::STRrs:
case 452: // ARM::STRT_POST:
case 453: // ARM::STRT_POST_IMM:
case 454: // ARM::STRT_POST_REG:

case 412: // ARM::STMDA:
case 413: // ARM::STMDA_UPD:
case 414: // ARM::STMDB:
case 415: // ARM::STMDB_UPD:
case 416: // ARM::STMIA:
case 417: // ARM::STMIA_UPD:
case 418: // ARM::STMIB:
case 419: // ARM::STMIB_UPD:


    ///LD

case 123: //ARM::LDA:
case 124: // ARM::LDAB:
case 125: // ARM::LDAEX:
case 126: // ARM::LDAEXB:
case 127: // ARM::LDAEXD:
case 128: // ARM::LDAEXH:
case 129: // ARM::LDAH:
case 130: // ARM::LDC2_OFFSET:
case 131: // ARM::LDC2_OPTION:
case 132: // ARM::LDC2_POST:
case 133: // ARM::LDC2_PRE:
case 134: // ARM::LDC2L_OFFSET:
case 135: // ARM::LDC2L_OPTION:
case 136: // ARM::LDC2L_POST:
case 137: // ARM::LDC2L_PRE:
case 138: // ARM::LDC_OFFSET:
case 139: // ARM::LDC_OPTION:
case 140: // ARM::LDC_POST:
case 141: // ARM::LDC_PRE:
case 142: // ARM::LDCL_OFFSET:
case 143: // ARM::LDCL_OPTION:
case 144: // ARM::LDCL_POST:
case 145: // ARM::LDCL_PRE:
case 155: // ARM::LDR_POST_IMM:
case 156: // ARM::LDR_POST_REG:
case 157: // ARM::LDR_PRE_IMM:
case 158: // ARM::LDR_PRE_REG:
case 159: // ARM::LDRB_POST_IMM:
case 160: // ARM::LDRB_POST_REG:
case 161: // ARM::LDRB_PRE_IMM:
case 162: // ARM::LDRB_PRE_REG:
case 163: // ARM::LDRBi12:
case 164: // ARM::LDRBrs:
case 165: // ARM::LDRBT_POST:
case 166: // ARM::LDRBT_POST_IMM:
case 167: // ARM::LDRBT_POST_REG:
case 168: // ARM::LDRcp:
case 169: // ARM::LDRD:
case 170: // ARM::LDRD_POST:
case 171: // ARM::LDRD_PRE:
case 172: //ARM::LDREX:
case 173: // ARM::LDREXB:
```

```
        case 174:// ARM::LDREXD:
        case 175:// ARM::LDREXH:
        case 176:// ARM::LDRH:
        case 177:// ARM::LDRH_POST:
        case 178:// ARM::LDRH_PRE:
        case 179:// ARM::LDRHTi:
        case 180:// ARM::LDRHTr:
        case 181:// ARM::LDRi12:
        case 182:// ARM::LDRLIT_ga_abs:
        case 183:// ARM::LDRLIT_ga_pcrel:
        case 184:// ARM::LDRLIT_ga_pcrel_ldr:
        case 185:// ARM::LDRrs:
        case 186:// ARM::LDRSB:
        case 187:// ARM::LDRSB_POST:
        case 188:// ARM::LDRSB_PRE:
        case 189:// ARM::LDRSBTi:
        case 190:// ARM::LDRSBTr:
        case 191:// ARM::LDRSH:
        case 192:// ARM::LDRSH_POST:
        case 193:// ARM::LDRSH_PRE:
        case 194:// ARM::LDRSHTi:
        case 195:// ARM::LDRSHTr:
        case 196:// ARM::LDRT_POST:
        case 197:// ARM::LDRT_POST_IMM:
        case 198:// ARM::LDRT_POST_REG:


        case 146:// ARM::LDMDA:
        case 147:// ARM::LDMDA_UPD:
        case 148:// ARM::LDMDB:

        case 149:// ARM::LDMDB_UPD:
        case 150:// ARM::LDMIA:
        case 151:// ARM::LDMIA_RET:
        case 152:// ARM::LDMIA_UPD:
        case 153:// ARM::LDMIB:
        case 154:// ARM::LDMIB_UPD:
                t=false;
                break;
        }
    }

    return t;


}
//
unsigned * anArmSched::VDS( SUnit* SU, const InstrItineraryData* itin){

    unsigned statLat[6]  ;
    MachineInstr *I = SU->getInstr();
    unsigned i =0 ;

    const InstrStage *E= itin->endStage(I->getDesc().getSchedClass());
    for(const InstrStage *S= itin->beginStage(I->getDesc().getSchedClass());S!=E ; ++S)
    {statLat[i]= S->getNextCycles();
        i++;

    }
```

```
    return  statLat;
}

//
unsigned  anArmSched::SL( SUnit* SU,const InstrItineraryData* itin)

{ unsigned SL=0;
    for(int i=0; i<6 ;i++)
    {SL+=VDS(SU, itin)[i];}
    return SL;}
//
unsigned* anArmSched::VDD( SUnit* SU, unsigned* VECA, unsigned* VORC, unsigned EU, const InstrItineraryData* itin)
{ unsigned VDD [4] ;
    VDD[0] = VDS(SU, itin)[0] + VECA[0];
    VDD[1] = std::max(VDD[0], VECA[1]) + VDS(SU, itin)[1];
    VDD[2] = std::max(VDD[1], VECA[2]) + VDS(SU, itin)[2];
    VDD[3] = VDD[2]+ VDS(SU, itin)[3] + VORC[EU];
    return VDD;
    }
//
unsigned anArmSched::DL (unsigned * VDD)
{ unsigned DL = VDD[3]-VDD[0]+1;
    return DL;}
//
void anArmSched::updateVECA (unsigned* VECA , unsigned * VDD)
{ for(int i=0 ;i< 3; i++){VECA[i]=VDD[i]; return;}
}
//
void anArmSched::updateVORC(unsigned* VORC , unsigned * VDD, unsigned EU)
{ for (int i=0 ;i<16; i++){
    if (VORC[i]>0)
    {VORC[i]-=1;}
    else {VORC[i]=0;}
}
    VORC[EU]+= VDD[3];
    return;
}
//
unsigned anArmSched::updatedDL (unsigned DL, unsigned iter)
{ if (DL-iter > 0) {return DL-iter;} else {return 0;} }

unsigned anArmSched::penalty (SUnit* SU, unsigned EU, const InstrItineraryData* itin)
{ if ( !isSupported(SU, EU)) {return SL(SU,itin);} else {return 0;}}
void anArmSched::updateEU ( unsigned EU)
{EU = (EU+1)%16;}
unsigned anArmSched::C2(SUnit* SU, std::vector<SUnit*> sequence)
{ unsigned C2 = 0;
    for (unsigned i = 0 ; (i < 16)&&(i< sequence.size()) ; ++i)
    { if (SU->isPred(sequence[i])){C2++;}
    }
    return C2;
}
unsigned anArmSched::maxPredDL(SUnit* SU, unsigned iter)
{ std::vector <unsigned int> PredDL(SU->Preds.size());
    int i=0;
    for (SUnit::const_pred_iterator I = SU->Preds.begin(), E = SU->Preds.end();
         I != E; ++I)
    {   SUnit* pred= I->getSUnit();
        PredDL[i]= updatedDL(pred->getDynaLat(), iter);
        i++;}
```

```
    unsigned int out =0;
    for( unsigned j=0; j< PredDL.size(); j++)
    { if (PredDL[j]>out)
    { out=PredDL[j];}}
    return out;
}

void anArmSched::ListScheduleTopDown() {
    const InstrItineraryData *itin = MF.getSubtarget().getInstrItineraryData();
    unsigned CurCycle = 0;
    unsigned iter =0;
    unsigned EU =0;
    unsigned VECA[3]={0};
    unsigned VORC[16]={0};
    std::vector<SUnit* > listPrete  ;
    // We're scheduling top-down but we're visiting the regions in
    // bottom-up order, so we don't know the hazards at the start of a
    // region. So assume no hazards (this should usually be ok as most
    // blocks are a single region).
    HazardRec->Reset();

    // Release any successors of the special Entry node.
    ReleaseSuccessors(&EntrySU);


    // Add all leaves to Available queue.
    for (unsigned i = 0, e = SUnits.size(); i != e; ++i) {
        // It is available if it has no predecessors.
        if (!SUnits[i].NumPredsLeft && !SUnits[i].isAvailable) {
            AvailableQueue.push(&SUnits[i]);
            listPrete.push_back(&SUnits[i]);
            SUnits[i].isAvailable = true;
        }
    }

    // In any cycle where we can't schedule any instructions, we must
    // stall or emit a noop, depending on the target.
    bool CycleHasInsts = false;

    // While Available queue is not empty, grab the node with the highest
    // priority. If it is not ready put it back.  Schedule the node.
    std::vector<SUnit*> NotReady;
    Sequence.reserve(SUnits.size());
    while (!AvailableQueue.empty() || !PendingQueue.empty()||!listPrete.empty()) {
        // Check to see if any of the pending instructions are ready to issue.  If
        // so, add them to the available queue.
        unsigned MinDepth = ~0u;

        for (unsigned i = 0, e = PendingQueue.size(); i != e; ++i) {
            if (PendingQueue[i]->getDepth() <= CurCycle) {
                AvailableQueue.push(PendingQueue[i]);
                listPrete.push_back(PendingQueue[i]);
                PendingQueue[i]->isAvailable = true;
                PendingQueue[i] = PendingQueue.back();
                PendingQueue.pop_back();
                --i; --e;
            } else if (PendingQueue[i]->getDepth() < MinDepth)
                MinDepth = PendingQueue[i]->getDepth();
        }
```

```
DEBUG(dbgs() << "\n***␣Examining␣Available\n"; AvailableQueue.dump(this));

SUnit *FoundSUnit = nullptr, *NotPreferredSUnit = nullptr;
bool HasNoopHazards = false;

while (!AvailableQueue.empty()&& !listPrete.empty()) {
    SortAvailable(listPrete, Sequence, EU, iter, itin);

    SUnit *CurSUnit = AvailableQueue.pop();
    SUnit* bestSU = listPrete.back();
    listPrete.pop_back();

    ScheduleHazardRecognizer::HazardType HT =
    HazardRec->getHazardType(CurSUnit, 0/*no stalls*/);
    if (HT == ScheduleHazardRecognizer::NoHazard) {
        if (HazardRec->ShouldPreferAnother(bestSU)) {
            if (!NotPreferredSUnit) {
                // If this is the first non-preferred node for this cycle, then
                // record it and continue searching for a preferred node. If this
                // is not the first non-preferred node, then treat it as though
                // there had been a hazard.
                NotPreferredSUnit = bestSU;
                continue;
            }
        } else {
            FoundSUnit = bestSU;
            break;
        }
    }

    // Remember if this is a noop hazard.
    HasNoopHazards |= HT == ScheduleHazardRecognizer::NoopHazard;

    NotReady.push_back(bestSU);
}

// If we have a non-preferred node, push it back onto the available list.
// If we did not find a preferred node, then schedule this first
// non-preferred node.
if (NotPreferredSUnit) {
    if (!FoundSUnit) {
        DEBUG(dbgs() << "***␣Will␣schedule␣a␣non-preferred␣instruction...\n");
        FoundSUnit = NotPreferredSUnit;
    } else {
        AvailableQueue.push(NotPreferredSUnit);
        listPrete.push_back(NotPreferredSUnit);
    }

    NotPreferredSUnit = nullptr;
}

// Add the nodes that aren't ready back onto the available list.
if (!NotReady.empty()) {
    AvailableQueue.push_all(NotReady);
    for( unsigned i = 0; i< NotReady.size();++i)
    {listPrete.push_back(NotReady[i]);}
    NotReady.clear();
}

// If we found a node to schedule...
```

```
        if (FoundSUnit) {
            // If we need to emit noops prior to this instruction, then do so.
            unsigned NumPreNoops = HazardRec->PreEmitNoops(FoundSUnit);
            for (unsigned i = 0; i != NumPreNoops; ++i)
                emitNoop(CurCycle);

            // ... schedule the node...
            ScheduleNodeTopDown(FoundSUnit, CurCycle);
            unsigned *vdd  =VDD(FoundSUnit, VECA, VORC, EU, itin);
            FoundSUnit->setDynaLat(DL(vdd));


          updateVECA(VECA, vdd);
            updateVORC(VORC, vdd, EU);
            updateEU(EU);
            iter++;
                        HazardRec->EmitInstruction(FoundSUnit);
            CycleHasInsts = true;
            if (HazardRec->atIssueLimit()) {
                DEBUG(dbgs() << "***␣Max␣instructions␣per␣cycle␣" << CurCycle << '\n');
                HazardRec->AdvanceCycle();
                ++CurCycle;
                CycleHasInsts = false;
            }
        } else {
            if (CycleHasInsts) {
                DEBUG(dbgs() << "***␣Finished␣cycle␣" << CurCycle << '\n');
                HazardRec->AdvanceCycle();
            } else if (!HasNoopHazards) {
                // Otherwise, we have a pipeline stall, but no other problem,
                // just advance the current cycle and try again.
                DEBUG(dbgs() << "***␣Stall␣in␣cycle␣" << CurCycle << '\n');
                HazardRec->AdvanceCycle();
                ++NumStalls;
            } else {
                // Otherwise, we have no instructions to issue and we have instructions
                // that will fault if we don't do this right.  This is the case for
                // processors without pipeline interlocks and other cases.
                emitNoop(CurCycle);
            }

            ++CurCycle;
            CycleHasInsts = false;
        }
    }

#ifndef NDEBUG
    unsigned ScheduledNodes = VerifyScheduledDAG(/*isBottomUp=*/false);
    unsigned Noops = 0;
    for (unsigned i = 0, e = Sequence.size(); i != e; ++i)
        if (!Sequence[i])
            ++Noops;
    assert(Sequence.size() - Noops == ScheduledNodes &&
           "The␣number␣of␣nodes␣scheduled␣doesn't␣match␣the␣expected␣number!");
#endif // NDEBUG
}
void anArmSched::EmitSchedule() {
    RegionBegin = RegionEnd;

    // If first instruction was a DBG_VALUE then put it back.
```

```
    if (FirstDbgValue)
        BB->splice(RegionEnd, BB, FirstDbgValue);

    // Then re-insert them according to the given schedule.
    for (unsigned i = 0, e = Sequence.size(); i != e; i++) {
        if (SUnit *SU = Sequence[i])
            BB->splice(RegionEnd, BB, SU->getInstr());
        else
            // Null SUnit* is a noop.
            TII->insertNoop(*BB, RegionEnd);

        // Update the Begin iterator, as the first instruction in the block
        // may have been scheduled later.
        if (i == 0)
            RegionBegin = std::prev(RegionEnd);
    }

    // Reinsert any remaining debug_values.
    for (std::vector<std::pair<MachineInstr *, MachineInstr *> >::iterator
         DI = DbgValues.end(), DE = DbgValues.begin(); DI != DE; --DI) {
        std::pair<MachineInstr *, MachineInstr *> P = *std::prev(DI);
        MachineInstr *DbgValue = P.first;
        MachineBasicBlock::iterator OrigPrivMI = P.second;
        BB->splice(++OrigPrivMI, BB, DbgValue);
    }
    DbgValues.clear();
    FirstDbgValue = nullptr;
}


 bool anArmPass::runOnMachineFunction(MachineFunction &mf)



{       if (skipOptnoneFunction(*mf.getFunction()))
    return false;

    TII = mf.getSubtarget().getInstrInfo();
    MachineLoopInfo &MLI = getAnalysis<MachineLoopInfo>();
    AliasAnalysis *AA = &getAnalysis<AliasAnalysis>();
    TargetPassConfig *PassConfig = &getAnalysis<TargetPassConfig>();

    RegClassInfo.runOnMachineFunction(mf);

    // Check for explicit enable/disable of post-ra scheduling.
    TargetSubtargetInfo::AntiDepBreakMode AntiDepMode =
    TargetSubtargetInfo::ANTIDEP_NONE;
    SmallVector<const TargetRegisterClass*, 4> CriticalPathRCs;


   anArmSched Scheduler(mf, MLI, AA, RegClassInfo, AntiDepMode,
                                    CriticalPathRCs);

    // Loop over all of the basic blocks
    for (MachineFunction::iterator MBB = mf.begin(), MBBe = mf.end();
         MBB != MBBe; ++MBB) {


        // Initialize register live-range state for scheduling in this block.
        Scheduler.startBlock(MBB);
```

```
    // Schedule each sequence of instructions not interrupted by a label
    // or anything else that effectively needs to shut down scheduling.
    MachineBasicBlock::iterator Current = MBB->end();
    unsigned Count = MBB->size(), CurrentCount = Count;
    for (MachineBasicBlock::iterator I = Current; I != MBB->begin(); ) {
        MachineInstr *MI = std::prev(I);
        --Count;
        // Calls are not scheduling boundaries before register allocation, but
        // post-ra we don't gain anything by scheduling across calls since we
        // don't need to worry about register pressure.
        if (MI->isCall() || TII->isSchedulingBoundary(MI, MBB, mf)) {
            Scheduler.enterRegion(MBB, I, Current, CurrentCount - Count);
            Scheduler.setEndIndex(CurrentCount);
            Scheduler.schedule();
            Scheduler.exitRegion();
            Scheduler.EmitSchedule();
            Current = MI;
            CurrentCount = Count;
            Scheduler.Observe(MI, CurrentCount);
        }
        I = MI;
        if (MI->isBundle())
            Count -= MI->getBundleSize();
    }
    assert(Count == 0 && "Instruction count mismatch!");
    assert((MBB->begin() == Current || CurrentCount != 0) &&
           "Instruction count mismatch!");
    Scheduler.enterRegion(MBB, MBB->begin(), Current, CurrentCount);
    Scheduler.setEndIndex(CurrentCount);
    Scheduler.schedule();
    Scheduler.exitRegion();
    Scheduler.EmitSchedule();

    // Clean up register live-range state.
    Scheduler.finishBlock();

    // Update register kills
    Scheduler.fixupKills(MBB);
}




    return true;
}
```

```
FunctionPass
*llvm::createMyCustomMachinePass(){ return new anArmPass();}
char anArmPass::ID =0;

static RegisterPass<anArmPass>
X("anarmpass","AnARmPass");
```

ANNEXE II

CODE SOURCE DES PROGRAMMES TÉSTÉS

1. Dhrystone

Extrait II.1 dhry1.c

```
/*
 ****************************************************************************
 *
 *                   "DHRYSTONE" Benchmark Program
 *                   -----------------------------
 *
 *  Version:    C, Version 2.1
 *
 *  File:       dhry_1.c (part 2 of 3)
 *
 *  Date:       May 25, 1988
 *
 *  Author:     Reinhold P. Weicker
 *
 ****************************************************************************
 */

#include "dhry.h"

/* Global Variables: */
Rec_Pointer     Ptr_Glob,
                Next_Ptr_Glob;
int             Int_Glob;
Boolean         Bool_Glob;
char            Ch_1_Glob,
                Ch_2_Glob;
int             Arr_1_Glob [50];
int             Arr_2_Glob [50] [50];
Rec_Type        Glob;
Rec_Type        Next_Glob;

#ifndef ITERATIONS
    #define ITERATIONS (1000)
#endif

#ifndef REG
        Boolean Reg = false;
#define REG
        /* REG becomes defined as empty */
        /* i.e. no register variables   */
#else
        Boolean Reg = true;
#endif

// Prototypes
Enumeration Func_1 (Capital_Letter Ch_1_Par_Val, Capital_Letter Ch_2_Par_Val);
Boolean     Func_2(Str_30 Str_1_Par_Ref, Str_30 Str_2_Par_Ref);
Boolean     Func_3(Enumeration Enum_Par_Val);
```

```
void        Proc_1(REG Rec_Pointer Ptr_Val_Par);
void        Proc_2(One_Fifty *Int_Par_Ref);
void        Proc_3(Rec_Pointer *Ptr_Ref_Par);
void        Proc_4(void);
void        Proc_5(void);
void        Proc_6(Enumeration  Enum_Val_Par, Enumeration *Enum_Ref_Par);
void        Proc_7(One_Fifty Int_1_Par_Val, One_Fifty Int_2_Par_Val, One_Fifty *Int_Par_Ref);
void        Proc_8(Arr_1_Dim Arr_1_Par_Ref, Arr_2_Dim Arr_2_Par_Ref, int Int_1_Par_Val, int Int_2_Par_Val);
char        *strcpy(char *dest, const char *src);
/* variables for time measurement: */
//
//#ifdef TIMES
//struct tms      time_info;
//extern  int     times ();
                /* see library function "times" */
//#define Too_Small_Time 120
                /* Measurements should last at least about 2 seconds */
//#endif
//#ifdef TIME
//extern long     time();
                /* see library function "time"  */
//#define Too_Small_Time 2
                /* Measurements should last at least 2 seconds */
//#endif

long            Begin_Time,
                End_Time,
                User_Time;
float           Microseconds,
                Dhrystones_Per_Second;

/* end of variables for time measurement */


int main(void)
/*****/

  /* main program, corresponds to procedures        */
  /* Main and Proc_0 in the Ada version             */
{
        One_Fifty       Int_1_Loc;
  REG   One_Fifty       Int_2_Loc;
        One_Fifty       Int_3_Loc;
  REG   char            Ch_Index;
        Enumeration     Enum_Loc;
        Str_30          Str_1_Loc;
        Str_30          Str_2_Loc;
  REG   int             Run_Index;
  REG   int             Number_Of_Runs = ITERATIONS;

  /* Initializations */

  //Next_Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));
    Next_Ptr_Glob = &Next_Glob;
  //Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));
    Ptr_Glob = &Glob;

  Ptr_Glob->Ptr_Comp                    = Next_Ptr_Glob;
  Ptr_Glob->Discr                       = Ident_1;
  Ptr_Glob->variant.var_1.Enum_Comp     = Ident_3;
```

```
  Ptr_Glob->variant.var_1.Int_Comp        = 40;
  strcpy (Ptr_Glob->variant.var_1.Str_Comp,
          "DHRYSTONE PROGRAM, SOME STRING");
  strcpy (Str_1_Loc, "DHRYSTONE PROGRAM, 1'ST STRING");

  Arr_2_Glob [8][7] = 10;
        /* Was missing in published program. Without this statement,    */
        /* Arr_2_Glob [8][7] would have an undefined value.             */
        /* Warning: With 16-Bit processors and Number_Of_Runs > 32000,  */
        /* overflow may occur for this array element.                   */
/*
  printf ("\n");
  printf ("Dhrystone Benchmark, Version 2.1 (Language: C)\n");
  printf ("\n");
  if (Reg)
  {
    printf ("Program compiled with 'register' attribute\n");
    printf ("\n");
  }
  else
  {
    printf ("Program compiled without 'register' attribute\n");
    printf ("\n");
  }
  printf ("Please give the number of runs through the benchmark: ");
  {
    int n;
    scanf ("%d", &n);
    Number_Of_Runs = n;
  }
  printf ("\n");

  printf ("Execution starts, %d runs through Dhrystone\n", Number_Of_Runs);
*/
  /***************/
  /* Start timer */
  /***************/
/*
    TIMER
    from sim time
#ifdef TIMES
  times (&time_info);
  Begin_Time = (long) time_info.tms_utime;
#endif
#ifdef TIME
  Begin_Time = time ( (long *) 0);
#endif
*/
  for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
  {

    Proc_5();
    Proc_4();
      /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
    Int_1_Loc = 2;
    Int_2_Loc = 3;
    strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
    Enum_Loc = Ident_2;
    Bool_Glob = ! Func_2 (Str_1_Loc, Str_2_Loc);
      /* Bool_Glob == 1 */
```

```
    while (Int_1_Loc < Int_2_Loc)  /* loop body executed once */
    {
      Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
        /* Int_3_Loc == 7 */
      Proc_7 (Int_1_Loc, Int_2_Loc, &Int_3_Loc);
        /* Int_3_Loc == 7 */
      Int_1_Loc += 1;
    } /* while */
      /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
    Proc_8 (Arr_1_Glob, Arr_2_Glob, Int_1_Loc, Int_3_Loc);
      /* Int_Glob == 5 */
    Proc_1 (Ptr_Glob);
    for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
                            /* loop body executed twice */
    {
      if (Enum_Loc == Func_1 (Ch_Index, 'C'))
          /* then, not executed */
        {
        Proc_6 (Ident_1, &Enum_Loc);
        strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 3'RD STRING");
        Int_2_Loc = Run_Index;
        Int_Glob = Run_Index;
        }
    }
      /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
    Int_2_Loc = Int_2_Loc * Int_1_Loc;
    Int_1_Loc = Int_2_Loc / Int_3_Loc;
    Int_2_Loc = 7 * (Int_2_Loc - Int_3_Loc) - Int_1_Loc;
      /* Int_1_Loc == 1, Int_2_Loc == 13, Int_3_Loc == 7 */
    Proc_2 (&Int_1_Loc);
      /* Int_1_Loc == 5 */

  } /* loop "for Run_Index" */
  /*
  __asm
  {
    MOVW r0, 0xFFFF
    MOVT r0, 0xFFFF
    MCR p15, 0, r0, c9, c12, 2
  }
  */
  /***************/
  /* Stop timer */
  /***************/
/* TIMER
    from sim time
#ifdef TIMES
  times (&time_info);
  End_Time = (long) time_info.tms_utime;
#endif
#ifdef TIME
  End_Time = time ( (long *) 0);
#endif
*/
/*
  printf ("Execution ends\n");
  printf ("\n");
  printf ("Final values of the variables used in the benchmark:\n");
  printf ("\n");
  printf ("Int_Glob:            %d\n", Int_Glob);
```

```
  printf ("          should be:   %d\n", 5);
  printf ("Bool_Glob:           %d\n", Bool_Glob);
  printf ("          should be:   %d\n", 1);
  printf ("Ch_1_Glob:           %c\n", Ch_1_Glob);
  printf ("          should be:   %c\n", 'A');
  printf ("Ch_2_Glob:           %c\n", Ch_2_Glob);
  printf ("          should be:   %c\n", 'B');
  printf ("Arr_1_Glob[8]:       %d\n", Arr_1_Glob[8]);
  printf ("          should be:   %d\n", 7);
  printf ("Arr_2_Glob[8][7]:    %d\n", Arr_2_Glob[8][7]);
  printf ("          should be:   Number_Of_Runs + 10\n");
  printf ("Ptr_Glob->\n");
  printf ("  Ptr_Comp:          %d\n", (int) Ptr_Glob->Ptr_Comp);
  printf ("          should be:   (implementation-dependent)\n");
  printf ("  Discr:             %d\n", Ptr_Glob->Discr);
  printf ("          should be:   %d\n", 0);
  printf ("  Enum_Comp:         %d\n", Ptr_Glob->variant.var_1.Enum_Comp);
  printf ("          should be:   %d\n", 2);
  printf ("  Int_Comp:          %d\n", Ptr_Glob->variant.var_1.Int_Comp);
  printf ("          should be:   %d\n", 17);
  printf ("  Str_Comp:          %s\n", Ptr_Glob->variant.var_1.Str_Comp);
  printf ("          should be:   DHRYSTONE PROGRAM, SOME STRING\n");
  printf ("Next_Ptr_Glob->\n");
  printf ("  Ptr_Comp:          %d\n", (int) Next_Ptr_Glob->Ptr_Comp);
  printf ("          should be:   (implementation-dependent), same as above\n");
  printf ("  Discr:             %d\n", Next_Ptr_Glob->Discr);
  printf ("          should be:   %d\n", 0);
  printf ("  Enum_Comp:         %d\n", Next_Ptr_Glob->variant.var_1.Enum_Comp);
  printf ("          should be:   %d\n", 1);
  printf ("  Int_Comp:          %d\n", Next_Ptr_Glob->variant.var_1.Int_Comp);
  printf ("          should be:   %d\n", 18);
  printf ("  Str_Comp:          %s\n",
                                Next_Ptr_Glob->variant.var_1.Str_Comp);
  printf ("          should be:   DHRYSTONE PROGRAM, SOME STRING\n");
  printf ("Int_1_Loc:           %d\n", Int_1_Loc);
  printf ("          should be:   %d\n", 5);
  printf ("Int_2_Loc:           %d\n", Int_2_Loc);
  printf ("          should be:   %d\n", 13);
  printf ("Int_3_Loc:           %d\n", Int_3_Loc);
  printf ("          should be:   %d\n", 7);
  printf ("Enum_Loc:            %d\n", Enum_Loc);
  printf ("          should be:   %d\n", 1);
  printf ("Str_1_Loc:           %s\n", Str_1_Loc);
  printf ("          should be:   DHRYSTONE PROGRAM, 1'ST STRING\n");
  printf ("Str_2_Loc:           %s\n", Str_2_Loc);
  printf ("          should be:   DHRYSTONE PROGRAM, 2'ND STRING\n");
  printf ("\n");

  User_Time = End_Time - Begin_Time;

  if (User_Time < Too_Small_Time)
  {
    printf ("Measured time too small to obtain meaningful results\n");
    printf ("Please increase number of runs\n");
    printf ("\n");
  }
  else
  {
#ifdef TIME
    Microseconds = (float) User_Time * Mic_secs_Per_Second
```

```
                        / (float) Number_Of_Runs;
    Dhrystones_Per_Second = (float) Number_Of_Runs / (float) User_Time;
#else
    Microseconds = (float) User_Time * Mic_secs_Per_Second
                        / ((float) HZ * ((float) Number_Of_Runs));
    Dhrystones_Per_Second = ((float) HZ * (float) Number_Of_Runs)
                        / (float) User_Time;
#endif
    printf ("Microseconds for one run through Dhrystone: ");
    printf ("%6.1f \n", Microseconds);
    printf ("Dhrystones per Second:                      ");
    printf ("%6.1f \n", Dhrystones_Per_Second);
    printf ("\n");
  }
  */
     /*
  __asm
  {
    mcr p15, 0, r0, c9, c12, 2
  }
  */
}


void Proc_1(Rec_Pointer Ptr_Val_Par)
/******************/
    /* executed once  */
{
  REG Rec_Pointer Next_Record = Ptr_Val_Par->Ptr_Comp;
                                        /* == Ptr_Glob_Next */
  /* Local variable, initialized with Ptr_Val_Par->Ptr_Comp,     */
  /* corresponds to "rename" in Ada, "with" in Pascal            */

  structassign (*Ptr_Val_Par->Ptr_Comp, *Ptr_Glob);
  Ptr_Val_Par->variant.var_1.Int_Comp = 5;
  Next_Record->variant.var_1.Int_Comp
        = Ptr_Val_Par->variant.var_1.Int_Comp;
  Next_Record->Ptr_Comp = Ptr_Val_Par->Ptr_Comp;
  Proc_3 (&Next_Record->Ptr_Comp);
    /* Ptr_Val_Par->Ptr_Comp->Ptr_Comp
                        == Ptr_Glob->Ptr_Comp */
  if (Next_Record->Discr == Ident_1)
    /* then, executed */
  {
    Next_Record->variant.var_1.Int_Comp = 6;
    Proc_6 (Ptr_Val_Par->variant.var_1.Enum_Comp,
           &Next_Record->variant.var_1.Enum_Comp);
    Next_Record->Ptr_Comp = Ptr_Glob->Ptr_Comp;
    Proc_7 (Next_Record->variant.var_1.Int_Comp, 10,
           &Next_Record->variant.var_1.Int_Comp);
  }
  else /* not executed */
    structassign (*Ptr_Val_Par, *Ptr_Val_Par->Ptr_Comp);
} /* Proc_1 */


void Proc_2(One_Fifty *Int_Par_Ref)
/******************/
    /* executed once */
    /* *Int_Par_Ref == 1, becomes 4 */
```

```
{
  One_Fifty   Int_Loc;
  Enumeration    Enum_Loc = Ident_1;

  Int_Loc = *Int_Par_Ref + 10;
  do /* executed once */
    if (Ch_1_Glob == 'A')
      /* then, executed */
    {
      Int_Loc -= 1;
      *Int_Par_Ref = Int_Loc - Int_Glob;
      Enum_Loc = Ident_1;
    } /* if */
  while (Enum_Loc != Ident_1); /* true */
} /* Proc_2 */


void Proc_3(Rec_Pointer *Ptr_Ref_Par)
/******************/
    /* executed once */
    /* Ptr_Ref_Par becomes Ptr_Glob */
{
  if (Ptr_Glob != Null)
    /* then, executed */
    *Ptr_Ref_Par = Ptr_Glob->Ptr_Comp;
  Proc_7 (10, Int_Glob, &Ptr_Glob->variant.var_1.Int_Comp);
} /* Proc_3 */


void Proc_4()
/******************/
    /* executed once */
{
  Boolean Bool_Loc;

  Bool_Loc = Ch_1_Glob == 'A';
  Bool_Glob = Bool_Loc | Bool_Glob;
  Ch_2_Glob = 'B';
} /* Proc_4 */


void Proc_5()
/******************/
    /* executed once */
{
  Ch_1_Glob = 'A';
  Bool_Glob = false;
} /* Proc_5 */


/* Procedure for the assignment of structures,          */
/* if the C compiler doesn't support this feature       */
#ifdef  NOSTRUCTASSIGN
    void memcpy (register char *d, register char *s, register int l)
    {
        while (l--) *d++ = *s++;
    }
#endif
```

Extrait II.2 dhry2.c

```
/*
 ****************************************************************************
 *
 *                   "DHRYSTONE" Benchmark Program
 *                   -----------------------------
 *
 *  Version:    C, Version 2.1
 *
 *  File:       dhry_2.c (part 3 of 3)
 *
 *  Date:       May 25, 1988
 *
 *  Author:     Reinhold P. Weicker
 *
 ****************************************************************************
 */

#include "dhry.h"

#ifndef REG
#define REG
    /* REG becomes defined as empty */
    /* i.e. no register variables   */
#endif

extern int  Int_Glob;
extern char Ch_1_Glob;

// Prototypes
Enumeration Func_1 (Capital_Letter Ch_1_Par_Val, Capital_Letter Ch_2_Par_Val);
Boolean     Func_2(Str_30 Str_1_Par_Ref, Str_30 Str_2_Par_Ref);
Boolean     Func_3(Enumeration Enum_Par_Val);
void        Proc_1(REG Rec_Pointer Ptr_Val_Par);
void        Proc_2(One_Fifty *Int_Par_Ref);
void        Proc_3(Rec_Pointer *Ptr_Ref_Par);
void        Proc_4(void);
void        Proc_5(void);
void        Proc_6(Enumeration  Enum_Val_Par, Enumeration *Enum_Ref_Par);
void        Proc_7(One_Fifty Int_1_Par_Val, One_Fifty Int_2_Par_Val, One_Fifty *Int_Par_Ref);
void        Proc_8(Arr_1_Dim Arr_1_Par_Ref, Arr_2_Dim Arr_2_Par_Ref, int Int_1_Par_Val, int Int_2_Par_Val);

void Proc_6(Enumeration Enum_Val_Par, Enumeration *Enum_Ref_Par)
/*********************************/
    /* executed once */
    /* Enum_Val_Par == Ident_3, Enum_Ref_Par becomes Ident_2 */
{
  *Enum_Ref_Par = Enum_Val_Par;
  if (! Func_3 (Enum_Val_Par))
    /* then, not executed */
    *Enum_Ref_Par = Ident_4;
  switch (Enum_Val_Par)
  {
    case Ident_1:
      *Enum_Ref_Par = Ident_1;
      break;
    case Ident_2:
      if (Int_Glob > 100)
        /* then */
```

```
      *Enum_Ref_Par = Ident_1;
      else *Enum_Ref_Par = Ident_4;
      break;
    case Ident_3: /* executed */
      *Enum_Ref_Par = Ident_2;
      break;
    case Ident_4: break;
    case Ident_5:
      *Enum_Ref_Par = Ident_3;
      break;
  } /* switch */
} /* Proc_6 */


void Proc_7(One_Fifty Int_1_Par_Val, One_Fifty Int_2_Par_Val, One_Fifty *Int_Par_Ref)
/**********************************************/
    /* executed three times                                      */
    /* first call:      Int_1_Par_Val == 2, Int_2_Par_Val == 3,  */
    /*                  Int_Par_Ref becomes 7                    */
    /* second call:     Int_1_Par_Val == 10, Int_2_Par_Val == 5, */
    /*                  Int_Par_Ref becomes 17                   */
    /* third call:      Int_1_Par_Val == 6, Int_2_Par_Val == 10, */
    /*                  Int_Par_Ref becomes 18                   */
{
  One_Fifty Int_Loc;

  Int_Loc = Int_1_Par_Val + 2;
  *Int_Par_Ref = Int_2_Par_Val + Int_Loc;
} /* Proc_7 */


void Proc_8(Arr_1_Dim Arr_1_Par_Ref, Arr_2_Dim Arr_2_Par_Ref, int Int_1_Par_Val, int Int_2_Par_Val)
/*********************************************************************/
    /* executed once      */
    /* Int_Par_Val_1 == 3 */
    /* Int_Par_Val_2 == 7 */
{
  REG One_Fifty Int_Index;
  REG One_Fifty Int_Loc;

  Int_Loc = Int_1_Par_Val + 5;
  Arr_1_Par_Ref [Int_Loc] = Int_2_Par_Val;
  Arr_1_Par_Ref [Int_Loc+1] = Arr_1_Par_Ref [Int_Loc];
  Arr_1_Par_Ref [Int_Loc+30] = Int_Loc;
  for (Int_Index = Int_Loc; Int_Index <= Int_Loc+1; ++Int_Index)
    Arr_2_Par_Ref [Int_Loc] [Int_Index] = Int_Loc;
  Arr_2_Par_Ref [Int_Loc] [Int_Loc-1] += 1;
  Arr_2_Par_Ref [Int_Loc+20] [Int_Loc] = Arr_1_Par_Ref [Int_Loc];
  Int_Glob = 5;
} /* Proc_8 */


Enumeration Func_1(Capital_Letter Ch_1_Par_Val, Capital_Letter Ch_2_Par_Val)
/*************************************************/
    /* executed three times                                         */
    /* first call:      Ch_1_Par_Val == 'H', Ch_2_Par_Val == 'R'    */
    /* second call:     Ch_1_Par_Val == 'A', Ch_2_Par_Val == 'C'    */
    /* third call:      Ch_1_Par_Val == 'B', Ch_2_Par_Val == 'C'    */
{
  Capital_Letter        Ch_1_Loc;
```

```
  Capital_Letter        Ch_2_Loc;

  Ch_1_Loc = Ch_1_Par_Val;
  Ch_2_Loc = Ch_1_Loc;
  if (Ch_2_Loc != Ch_2_Par_Val)
    /* then, executed */
    return (Ident_1);
  else  /* not executed */
  {
    Ch_1_Glob = Ch_1_Loc;
    return (Ident_2);
   }
} /* Func_1 */


Boolean Func_2(Str_30 Str_1_Par_Ref, Str_30 Str_2_Par_Ref)
/*************************************************/
    /* executed once */
    /* Str_1_Par_Ref == "DHRYSTONE PROGRAM, 1'ST STRING" */
    /* Str_2_Par_Ref == "DHRYSTONE PROGRAM, 2'ND STRING" */
{
  REG One_Thirty        Int_Loc;
      Capital_Letter    Ch_Loc = 'A';

  Int_Loc = 2;
  while (Int_Loc <= 2) /* loop body executed once */
    if (Func_1 (Str_1_Par_Ref[Int_Loc],
                Str_2_Par_Ref[Int_Loc+1]) == Ident_1)
      /* then, executed */
    {
      Ch_Loc = 'A';
      Int_Loc += 1;
    } /* if, while */
  if (Ch_Loc >= 'W' && Ch_Loc < 'Z')
    /* then, not executed */
    Int_Loc = 7;
  if (Ch_Loc == 'R')
    /* then, not executed */
    return (true);
  else /* executed */
  {
    if (strcmp (Str_1_Par_Ref, Str_2_Par_Ref) > 0)
      /* then, not executed */
    {
      Int_Loc += 7;
      Int_Glob = Int_Loc;
      return (true);
    }
    else /* executed */
      return (false);
  } /* if Ch_Loc */
} /* Func_2 */


Boolean Func_3(Enumeration Enum_Par_Val)
/***************************/
    /* executed once        */
    /* Enum_Par_Val == Ident_3 */
{
  Enumeration Enum_Loc;
```

```
  Enum_Loc = Enum_Par_Val;
  if (Enum_Loc == Ident_3)
    /* then, executed */
    return (true);
  else /* not executed */
    return (false);
} /* Func_3 */
```

2. Multiplication matricielle

Extrait II.3 IntMM.c

```
#include <stdio.h>
#include <stdlib.h>

#define   nil             0
#define   false           0
#define   true            1
#define   bubblebase      1.61f
#define   dnfbase         3.5f
#define   permbase        1.75f
#define   queensbase 1.83f
#define   towersbase 2.39f
#define   quickbase       1.92f
#define   intmmbase       1.46f
#define   treebase        2.5f
#define   mmbase          0.0f
#define   fpmmbase        2.92f
#define   puzzlebase      0.5f
#define   fftbase         0.0f
#define   fpfftbase       4.44f
    /* Towers */
#define maxcells          18

    /* Intmm, Mm */
#define rowsize           40

    /* Puzzle */
#define size              511
#define classmax          3
#define typemax           12
#define d                     8

    /* Bubble, Quick */
#define sortelements 5000
#define srtelements  500

    /* fft */
#define fftsize           256
#define fftsize2          129
/*
type */
    /* Perm */
#define permrange     10

   /* tree */
struct node {
        struct node *left,*right;
        int val;
};

    /* Towers */ /*
    discsizrange = 1..maxcells; */
#define     stackrange    3
/*    cellcursor = 0..maxcells; */
struct    element {
        int discsize;
```

```
        int next;
};
/*    emsgtype = packed array[1..15] of char;
*/
    /* Intmm, Mm */ /*
    index = 1 .. rowsize;
    intmatrix = array [index,index] of integer;
    realmatrix = array [index,index] of real;
*/
    /* Puzzle */ /*
    piececlass = 0..classmax;
    piecetype = 0..typemax;
    position = 0..size;
*/
    /* Bubble, Quick */ /*
    listsize = 0..sortelements;
    sortarray = array [listsize] of integer;
*/
    /* FFT */
struct    complex { float rp, ip; } ;
/*
    carray = array [1..fftsize] of complex ;
    c2array = array [1..fftsize2] of complex ;
*/

float value, fixed, floated;

    /* global */
long    seed;  /* converted to long for 16 bit WR*/

    /* Perm */
int    permarray[permrange+1];
/* converted pctr to unsigned int for 16 bit WR*/
unsigned int    pctr;

    /* tree */
struct node *tree;

    /* Towers */
int         stack[stackrange+1];
struct element    cellspace[maxcells+1];
int    freelist,  movesdone;

    /* Intmm, Mm */

int   ima[rowsize+1][rowsize+1], imb[rowsize+1][rowsize+1], imr[rowsize+1][rowsize+1];
float rma[rowsize+1][rowsize+1], rmb[rowsize+1][rowsize+1], rmr[rowsize+1][rowsize+1];

    /* Puzzle */
int     piececount[classmax+1], class[typemax+1], piecemax[typemax+1];
int     puzzl[size+1], p[typemax+1][size+1], n, kount;

    /* Bubble, Quick */
int sortlist[sortelements+1], biggest, littlest, top;

    /* FFT */
struct complex    z[fftsize+1], w[fftsize+1], e[fftsize2+1];
float    zr, zi;

void Initrand () {
```

```
    seed = 74755L;   /* constant to long WR*/
}

int Rand () {
    seed = (seed * 1309L + 13849L) & 65535L;  /* constants to long WR*/
    return( (int)seed );      /* typecast back to int WR*/
}


    /* Multiplies two integer matrices. */

void Initmatrix (int m[rowsize+1][rowsize+1]) {
        int temp, i, j;
        for ( i = 1; i <= rowsize; i++ )
            for ( j = 1; j <= rowsize; j++ ) {
            temp = Rand();
                m[i][j] = temp - (temp/120)*120 - 60;
        }
}

void Innerproduct( int *result, int a[rowsize+1][rowsize+1], int b[rowsize+1][rowsize+1], int row, int column) {
        /* computes the inner product of A[row,*] and B[*,column] */
        int i;
        *result = 0;
        for(i = 1; i <= rowsize; i++ )*result = *result+a[row][i]*b[i][column];
}

void Intmm (int run) {
    int i, j;
    Initrand();
    Initmatrix (ima);
    Initmatrix (imb);
    for ( i = 1; i <= rowsize; i++ )
                for ( j = 1; j <= rowsize; j++ )
                        Innerproduct(&imr[i][j],ima,imb,i,j);
        printf("%d\n", imr[run + 1][run + 1]);
}

int main()
{
        int i;
        for (i = 0; i < 10; i++) Intmm(i);
        return 0;
}
```

3. FFT

Extrait II.4 fft.c

```
/*      fix_fft.c - Fixed-point Fast Fourier Transform  */
/*
        fix_fft()       perform FFT or inverse FFT
        window()        applies a Hanning window to the (time) input
        fix_loud()      calculates the loudness of the signal, for
                        each freq point. Result is an integer array,
                        units are dB (values will be negative).
        iscale()        scale an integer value by (numer/denom).
        fix_mpy()       perform fixed-point multiplication.
        Sinewave[1024]  sinewave normalized to 32767 (= 1.0).
        Loudampl[100]   Amplitudes for lopudnesses from 0 to -99 dB.
        Low_pass        Low-pass filter, cutoff at sample_freq / 4.


        All data are fixed-point short integers, in which
        -32768 to +32768 represent -1.0 to +1.0. Integer arithmetic
        is used for speed, instead of the more natural floating-point.

        For the forward FFT (time -> freq), fixed scaling is
        performed to prevent arithmetic overflow, and to map a 0dB
        sine/cosine wave (i.e. amplitude = 32767) to two -6dB freq
        coefficients; the one in the lower half is reported as 0dB
        by fix_loud(). The return value is always 0.

        For the inverse FFT (freq -> time), fixed scaling cannot be
        done, as two 0dB coefficients would sum to a peak amplitude of
        64K, overflowing the 32k range of the fixed-point integers.
        Thus, the fix_fft() routine performs variable scaling, and
        returns a value which is the number of bits LEFT by which
        the output must be shifted to get the actual amplitude
        (i.e. if fix_fft() returns 3, each value of fr[] and fi[]
        must be multiplied by 8 (2**3) for proper scaling.
        Clearly, this cannot be done within the fixed-point short
        integers. In practice, if the result is to be used as a
        filter, the scale_shift can usually be ignored, as the
        result will be approximately correctly normalized as is.


        TURBO C, any memory model; uses inline assembly for speed
        and for carefully-scaled arithmetic.

        Written by:  Tom Roberts  11/8/89
        Made portable:  Malcolm Slaney 12/15/94 malcolm@interval.com

                Timing on a Macintosh PowerBook 180.... (using Symantec C6.0)
                        fix_fft (1024 points)            8 ticks
                        fft (1024 points - Using SANE)  112 Ticks
                        fft (1024 points - Using FPU)    11

*/

/* FIX_MPY() - fixed-point multiplication macro.
   This macro is a statement, not an expression (uses asm).
   BEWARE: make sure _DX is not clobbered by evaluating (A) or DEST.
   args are all of type fixed.
```

```
   Scaling ensures that 32767*32767 = 32767. */
#define dosFIX_MPY(DEST,A,B)        {       \
        _DX = (B);                          \
        _AX = (A);                          \
        asm imul dx;                        \
        asm add ax,ax;                      \
        asm adc dx,dx;                      \
        DEST = _DX;             }

#define FIX_MPY(DEST,A,B)       DEST = ((long)(A) * (long)(B))>>15

#define N_WAVE          1024    /* dimension of Sinewave[] */
#define LOG2_N_WAVE     10      /* log2(N_WAVE) */
#define N_LOUD          100     /* dimension of Loudampl[] */
#ifndef fixed
#define fixed short
#endif

extern fixed Sinewave[N_WAVE]; /* placed at end of this file for clarity */
extern fixed Loudampl[N_LOUD];
int db_from_ampl(fixed re, fixed im);
fixed fix_mpy(fixed a, fixed b);

/*
        fix_fft() - perform fast Fourier transform.

        if n>0 FFT is done, if n<0 inverse FFT is done
        fr[n],fi[n] are real,imaginary arrays, INPUT AND RESULT.
        size of data = 2**m
        set inverse to 0=dft, 1=idft
*/
int fix_fft(fixed fr[], fixed fi[], int m, int inverse)
{
        int mr,nn,i,j,l,k,istep, n, scale, shift;
        fixed qr,qi,tr,ti,wr,wi,t;

                n = 1<<m;

        if(n > N_WAVE)
                return -1;

        mr = 0;
        nn = n - 1;
        scale = 0;

        /* decimation in time - re-order data */
        for(m=1; m<=nn; ++m) {
                l = n;
                do {
                        l >>= 1;
                } while(mr+1 > nn);
                mr = (mr & (l-1)) + 1;

                if(mr <= m) continue;
                tr = fr[m];
                fr[m] = fr[mr];
                fr[mr] = tr;
                ti = fi[m];
                fi[m] = fi[mr];
                fi[mr] = ti;
```

```
        }

        l = 1;
        k = LOG2_N_WAVE-1;
        while(l < n) {
                if(inverse) {
                        /* variable scaling, depending upon data */
                        shift = 0;
                        for(i=0; i<n; ++i) {
                                j = fr[i];
                                if(j < 0)
                                        j = -j;
                                m = fi[i];
                                if(m < 0)
                                        m = -m;
                                if(j > 16383 || m > 16383) {
                                        shift = 1;
                                        break;
                                }
                        }
                        if(shift)
                                ++scale;
                } else {
                        /* fixed scaling, for proper normalization -
                           there will be log2(n) passes, so this
                           results in an overall factor of 1/n,
                           distributed to maximize arithmetic accuracy. */
                        shift = 1;
                }
                /* it may not be obvious, but the shift will be performed
                   on each data point exactly once, during this pass. */
                istep = l << 1;
                for(m=0; m<l; ++m) {
                        j = m << k;
                        /* 0 <= j < N_WAVE/2 */
                        wr =  Sinewave[j+N_WAVE/4];
                        wi = -Sinewave[j];
                        if(inverse)
                                wi = -wi;
                        if(shift) {
                                wr >>= 1;
                                wi >>= 1;
                        }
                        for(i=m; i<n; i+=istep) {
                                j = i + l;
                                        tr = fix_mpy(wr,fr[j]) -
fix_mpy(wi,fi[j]);
                                        ti = fix_mpy(wr,fi[j]) +
fix_mpy(wi,fr[j]);
                                qr = fr[i];
                                qi = fi[i];
                                if(shift) {
                                        qr >>= 1;
                                        qi >>= 1;
                                }
                                fr[j] = qr - tr;
                                fi[j] = qi - ti;
                                fr[i] = qr + tr;
                                fi[i] = qi + ti;
                        }
```

```
                }
                --k;
                l = istep;
        }

        return scale;
}


/*      window() - apply a Hanning window       */
void window(fixed fr[], int n)
{
        int i,j,k;

        j = N_WAVE/n;
        n >>= 1;
        for(i=0,k=N_WAVE/4; i<n; ++i,k+=j)
                FIX_MPY(fr[i],fr[i],16384-(Sinewave[k]>>1));
        n <<= 1;
        for(k-=j; i<n; ++i,k-=j)
                FIX_MPY(fr[i],fr[i],16384-(Sinewave[k]>>1));
}

/*      fix_loud() - compute loudness of freq-spectrum components.
        n should be ntot/2, where ntot was passed to fix_fft();
        6 dB is added to account for the omitted alias components.
        scale_shift should be the result of fix_fft(), if the time-series
        was obtained from an inverse FFT, 0 otherwise.
        loud[] is the loudness, in dB wrt 32767; will be +10 to -N_LOUD.
*/
void fix_loud(fixed loud[], fixed fr[], fixed fi[], int n, int scale_shift)
{
        int i, max;

        max = 0;
        if(scale_shift > 0)
                max = 10;
        scale_shift = (scale_shift+1) * 6;

        for(i=0; i<n; ++i) {
                loud[i] = db_from_ampl(fr[i],fi[i]) + scale_shift;
                if(loud[i] > max)
                        loud[i] = max;
        }
}

/*      db_from_ampl() - find loudness (in dB) from
        the complex amplitude.
*/
int db_from_ampl(fixed re, fixed im)
{
        static long loud2[N_LOUD] = {0};
        long v;
        int i;

        if(loud2[0] == 0) {
                loud2[0] = (long)Loudampl[0] * (long)Loudampl[0];
                for(i=1; i<N_LOUD; ++i) {
                        v = (long)Loudampl[i] * (long)Loudampl[i];
                        loud2[i] = v;
```

```
                        loud2[i-1] = (loud2[i-1]+v) / 2;
                }
        }

        v = (long)re * (long)re + (long)im * (long)im;

        for(i=0; i<N_LOUD; ++i)
                if(loud2[i] <= v)
                        break;

        return (-i);
}

/*
        fix_mpy() - fixed-point multiplication
*/
fixed fix_mpy(fixed a, fixed b)
{
        FIX_MPY(a,a,b);
        return a;
}

/*
        iscale() - scale an integer value by (numer/denom)
*/
int iscale(int value, int numer, int denom)
{
#ifdef  DOS
        asm     mov ax,value
        asm     imul WORD PTR numer
        asm     idiv WORD PTR denom

        return _AX;
#else
                return (long) value * (long)numer/(long)denom;
#endif
}

/*
        fix_dot() - dot product of two fixed arrays
*/
fixed fix_dot(fixed *hpa, fixed *pb, int n)
{
        fixed *pa;
        long sum;
        register fixed a,b;
        unsigned int seg,off;

/*      seg = FP_SEG(hpa);
        off = FP_OFF(hpa);
        seg += off>>4;
        off &= 0x000F;
        pa = MK_FP(seg,off);
 */
        sum = 0L;
        while(n--) {
                a = *pa++;
                b = *pb++;
                FIX_MPY(a,a,b);
                sum += a;
```

```
        }

        if(sum > 0x7FFF)
                sum = 0x7FFF;
        else if(sum < -0x7FFF)
                sum = -0x7FFF;

        return (fixed)sum;
#ifdef  DOS
        /* ASSUMES hpa is already normalized so FP_OFF(hpa) < 16 */
        asm     push    ds
        asm     lds     si,hpa
        asm     les     di,pb
        asm     xor     bx,bx

        asm     xor     cx,cx

loop:   /* intermediate values can overflow by a factor of 2 without
           causing an error; the final value must not overflow! */
        asm     lodsw
.
        asm     imul    word ptr es:[di]
        asm     add     bx,ax
        asm     adc     cx,dx
        asm     jo      overflow
        asm     add     di,2
        asm     dec     word ptr n
        asm     jg      loop

        asm     add     bx,bx
        asm     adc     cx,cx
        asm     jo      overflow

        asm     pop     ds
        return _CX;

overflow:
        asm     mov     cx,7FFFH
        asm     adc     cx,0

        asm     pop     ds
        return _CX;
#endif

}


#if N_WAVE != 1024
        ERROR: N_WAVE != 1024
#endif
fixed Sinewave[1024] = {
      0,    201,    402,    603,    804,   1005,   1206,   1406,
   1607,   1808,   2009,   2209,   2410,   2610,   2811,   3011,
   3211,   3411,   3611,   3811,   4011,   4210,   4409,   4608,
   4807,   5006,   5205,   5403,   5601,   5799,   5997,   6195,
   6392,   6589,   6786,   6982,   7179,   7375,   7571,   7766,
   7961,   8156,   8351,   8545,   8739,   8932,   9126,   9319,
   9511,   9703,   9895,  10087,  10278,  10469,  10659,  10849,
  11038,  11227,  11416,  11604,  11792,  11980,  12166,  12353,
  12539,  12724,  12909,  13094,  13278,  13462,  13645,  13827,
```

```
14009,  14191,  14372,  14552,  14732,  14911,  15090,  15268,
15446,  15623,  15799,  15975,  16150,  16325,  16499,  16672,
16845,  17017,  17189,  17360,  17530,  17699,  17868,  18036,
18204,  18371,  18537,  18702,  18867,  19031,  19194,  19357,
19519,  19680,  19840,  20000,  20159,  20317,  20474,  20631,
20787,  20942,  21096,  21249,  21402,  21554,  21705,  21855,
22004,  22153,  22301,  22448,  22594,  22739,  22883,  23027,
23169,  23311,  23452,  23592,  23731,  23869,  24006,  24143,
24278,  24413,  24546,  24679,  24811,  24942,  25072,  25201,
25329,  25456,  25582,  25707,  25831,  25954,  26077,  26198,
26318,  26437,  26556,  26673,  26789,  26905,  27019,  27132,
27244,  27355,  27466,  27575,  27683,  27790,  27896,  28001,
28105,  28208,  28309,  28410,  28510,  28608,  28706,  28802,
28897,  28992,  29085,  29177,  29268,  29358,  29446,  29534,
29621,  29706,  29790,  29873,  29955,  30036,  30116,  30195,
30272,  30349,  30424,  30498,  30571,  30643,  30713,  30783,
30851,  30918,  30984,  31049,
31113,  31175,  31236,  31297,
31356,  31413,  31470,  31525,  31580,  31633,  31684,  31735,
31785,  31833,  31880,  31926,  31970,  32014,  32056,  32097,
32137,  32176,  32213,  32249,  32284,  32318,  32350,  32382,
32412,  32441,  32468,  32495,  32520,  32544,  32567,  32588,
32609,  32628,  32646,  32662,  32678,  32692,  32705,  32717,
32727,  32736,  32744,  32751,  32757,  32761,  32764,  32766,
32767,  32766,  32764,  32761,  32757,  32751,  32744,  32736,
32727,  32717,  32705,  32692,  32678,  32662,  32646,  32628,
32609,  32588,  32567,  32544,  32520,  32495,  32468,  32441,
32412,  32382,  32350,  32318,  32284,  32249,  32213,  32176,
32137,  32097,  32056,  32014,  31970,  31926,  31880,  31833,
31785,  31735,  31684,  31633,  31580,  31525,  31470,  31413,
31356,  31297,  31236,  31175,  31113,  31049,  30984,  30918,
30851,  30783,  30713,  30643,  30571,  30498,  30424,  30349,
30272,  30195,  30116,  30036,  29955,  29873,  29790,  29706,
29621,  29534,  29446,  29358,  29268,  29177,  29085,  28992,
28897,  28802,  28706,  28608,  28510,  28410,  28309,  28208,
28105,  28001,  27896,  27790,  27683,  27575,  27466,  27355,
27244,  27132,  27019,  26905,  26789,  26673,  26556,  26437,
26318,  26198,  26077,  25954,  25831,  25707,  25582,  25456,
25329,  25201,  25072,  24942,  24811,  24679,  24546,  24413,
24278,  24143,  24006,  23869,  23731,  23592,  23452,  23311,
23169,  23027,  22883,  22739,  22594,  22448,  22301,  22153,
22004,  21855,  21705,  21554,  21402,  21249,  21096,  20942,
20787,  20631,  20474,  20317,  20159,  20000,  19840,  19680,
19519,  19357,  19194,  19031,  18867,  18702,  18537,  18371,
18204,  18036,  17868,  17699,  17530,  17360,  17189,  17017,
16845,  16672,  16499,  16325,  16150,  15975,  15799,  15623,
15446,  15268,  15090,  14911,  14732,  14552,  14372,  14191,
14009,  13827,  13645,  13462,  13278,  13094,  12909,  12724,
12539,  12353,  12166,  11980,  11792,  11604,  11416,  11227,
11038,  10849,  10659,  10469,  10278,  10087,   9895,   9703,
 9511,   9319,   9126,   8932,   8739,   8545,   8351,   8156,
 7961,   7766,   7571,   7375,   7179,   6982,   6786,   6589,
 6392,   6195,   5997,   5799,   5601,   5403,   5205,   5006,
 4807,   4608,   4409,   4210,   4011,   3811,   3611,   3411,
 3211,   3011,   2811,   2610,   2410,   2209,   2009,   1808,
 1607,   1406,   1206,   1005,    804,    603,    402,    201,
    0,   -201,   -402,   -603,   -804,  -1005,  -1206,  -1406,
-1607,  -1808,  -2009,  -2209,  -2410,  -2610,  -2811,  -3011,
-3211,  -3411,  -3611,  -3811,  -4011,  -4210,  -4409,  -4608,
-4807,  -5006,  -5205,  -5403,  -5601,  -5799,  -5997,  -6195,
```

−6392, −6589, −6786, −6982, −7179, −7375, −7571, −7766,
−7961, −8156, −8351, −8545, −8739, −8932, −9126, −9319,
−9511, −9703, −9895, −10087, −10278, −10469, −10659, −10849,
−11038, −11227, −11416, −11604, −11792, −11980, −12166, −12353,
−12539, −12724, −12909, −13094, −13278, −13462, −13645, −13827,
−14009, −14191, −14372, −14552, −14732, −14911, −15090, −15268,
−15446, −15623, −15799, −15975, −16150, −16325, −16499, −16672,
−16845, −17017, −17189, −17360, −17530, −17699, −17868, −18036,
−18204, −18371, −18537, −18702, −18867, −19031, −19194, −19357,
−19519, −19680, −19840, −20000, −20159, −20317, −20474, −20631,
−20787, −20942, −21096, −21249, −21402, −21554, −21705, −21855,
−22004, −22153, −22301, −22448, −22594, −22739, −22883, −23027,
−23169, −23311, −23452, −23592, −23731, −23869, −24006, −24143,
−24278, −24413, −24546, −24679, −24811, −24942, −25072, −25201,
−25329, −25456, −25582, −25707, −25831, −25954, −26077, −26198,
−26318, −26437, −26556, −26673, −26789, −26905, −27019, −27132,
−27244, −27355, −27466, −27575, −27683, −27790, −27896, −28001,
−28105, −28208, −28309, −28410, −28510, −28608, −28706, −28802,
−28897, −28992, −29085, −29177, −29268, −29358, −29446, −29534,
−29621, −29706, −29790, −29873, −29955, −30036, −30116, −30195,
−30272, −30349, −30424, −30498, −30571, −30643, −30713, −30783,
−30851, −30918, −30984, −31049, −31113, −31175, −31236, −31297,
−31356, −31413, −31470, −31525, −31580, −31633, −31684, −31735,
−31785, −31833, −31880, −31926, −31970, −32014, −32056, −32097,
−32137, −32176, −32213, −32249, −32284, −32318, −32350, −32382,
−32412, −32441, −32468, −32495, −32520, −32544, −32567, −32588,
−32609, −32628, −32646, −32662, −32678, −32692, −32705, −32717,
−32727, −32736, −32744, −32751, −32757, −32761, −32764, −32766,
−32767, −32766, −32764, −32761, −32757, −32751, −32744, −32736,
−32727, −32717, −32705, −32692, −32678, −32662, −32646, −32628,
−32609, −32588, −32567, −32544, −32520, −32495, −32468, −32441,
−32412, −32382, −32350, −32318, −32284, −32249, −32213, −32176,
−32137, −32097, −32056, −32014, −31970, −31926, −31880, −31833,
−31785, −31735, −31684, −31633, −31580, −31525, −31470, −31413,
−31356, −31297, −31236, −31175, −31113, −31049, −30984, −30918,
−30851, −30783, −30713, −30643, −30571, −30498, −30424, −30349,
−30272, −30195, −30116, −30036, −29955, −29873, −29790, −29706,
−29621, −29534, −29446, −29358, −29268, −29177, −29085, −28992,
−28897, −28802, −28706, −28608, −28510, −28410, −28309, −28208,
−28105, −28001, −27896, −27790, −27683, −27575, −27466, −27355,
−27244, −27132, −27019, −26905, −26789, −26673, −26556, −26437,
−26318, −26198, −26077, −25954, −25831, −25707, −25582, −25456,
−25329, −25201, −25072, −24942, −24811, −24679, −24546, −24413,
−24278, −24143, −24006, −23869, −23731, −23592, −23452, −23311,
−23169, −23027, −22883, −22739, −22594, −22448, −22301, −22153,
−22004, −21855, −21705, −21554, −21402, −21249, −21096, −20942,
−20787, −20631, −20474, −20317, −20159, −20000, −19840, −19680,
−19519, −19357, −19194, −19031, −18867, −18702, −18537, −18371,
−18204, −18036, −17868, −17699, −17530, −17360, −17189, −17017,
−16845, −16672, −16499, −16325, −16150, −15975, −15799, −15623,
−15446, −15268, −15090, −14911, −14732, −14552, −14372, −14191,
−14009, −13827, −13645, −13462, −13278, −13094, −12909, −12724,
−12539, −12353, −12166, −11980, −11792, −11604, −11416, −11227,
−11038, −10849, −10659, −10469, −10278, −10087, −9895, −9703,
−9511, −9319, −9126, −8932, −8739, −8545, −8351, −8156,
−7961, −7766, −7571, −7375, −7179, −6982, −6786, −6589,
−6392, −6195, −5997, −5799, −5601, −5403, −5205, −5006,
−4807, −4608, −4409, −4210, −4011, −3811, −3611, −3411,
−3211, −3011, −2811, −2610, −2410, −2209, −2009, −1808,
−1607, −1406, −1206, −1005, −804, −603, −402, −201,

```
};

#if N_LOUD != 100
        ERROR: N_LOUD != 100
#endif
fixed Loudampl[100] = {
  32767,  29203,  26027,  23197,  20674,  18426,  16422,  14636,
  13044,  11626,  10361,   9234,   8230,   7335,   6537,   5826,
   5193,   4628,   4125,   3676,   3276,   2920,   2602,   2319,
   2067,   1842,   1642,   1463,   1304,   1162,   1036,    923,
    823,    733,    653,    582,    519,    462,    412,    367,
    327,    292,    260,    231,    206,    184,    164,    146,
    130,    116,    103,     92,     82,     73,     65,     58,
     51,     46,     41,     36,     32,     29,     26,     23,
     20,     18,     16,     14,     13,     11,     10,      9,
      8,      7,      6,      5,      5,      4,      4,      3,
      3,      2,      2,      2,      2,      1,      1,      1,
      1,      1,      1,      0,      0,      0,      0,      0,
      0,      0,      0,      0,
};

#ifdef  MAIN

#include        <stdio.h>
#include        <math.h>

#define M       4
#define N       (1<<M)

main(){
        fixed real[N], imag[N];
        int     i;

        for (i=0; i<N; i++){
                real[i] = 1000*cos(i*2*3.1415926535/N);
                imag[i] = 0;
        }

        fix_fft(real, imag, M, 0;

        for (i=0; i<N; i++){
                printf("%d: %d, %d\n", i, real[i], imag[i]);
        }

        fix_fft(real, imag, M, 1);

        for (i=0; i<N; i++){
                printf("%d: %d, %d\n", i, real[i], imag[i]);
        }
}
#endif  /* MAIN */
```

4. Tri à bulles

Extrait II.5 bubble.c

```
#include <stdio.h>
#include <stdlib.h>

#define   nil             0
#define   false           0
#define   true            1
#define   bubblebase      1.61f
#define   dnfbase         3.5f
#define   permbase        1.75f
#define   queensbase 1.83f
#define   towersbase 2.39f
#define   quickbase       1.92f
#define   intmmbase       1.46f
#define   treebase        2.5f
#define   mmbase          0.0f
#define   fpmmbase        2.92f
#define   puzzlebase      0.5f
#define   fftbase         0.0f
#define   fpfftbase       4.44f
    /* Towers */
#define maxcells          18

    /* Intmm, Mm */
#define rowsize           40

    /* Puzzle */
#define size              511
#define classmax          3
#define typemax           12
#define d                     8

    /* Bubble, Quick */
#define sortelements 5000
#define srtelements  500

    /* fft */
#define fftsize           256
#define fftsize2          129
/*
type */
    /* Perm */
#define permrange     10

   /* tree */
struct node {
        struct node *left,*right;
        int val;
};

    /* Towers */ /*
    discsizrange = 1..maxcells; */
#define     stackrange    3
/*    cellcursor = 0..maxcells; */
struct    element {
        int discsize;
```

```
        int next;
};
/*    emsgtype = packed array[1..15] of char;
*/
    /* Intmm, Mm */ /*
    index = 1 .. rowsize;
    intmatrix = array [index,index] of integer;
    realmatrix = array [index,index] of real;
*/
    /* Puzzle */ /*
    piececlass = 0..classmax;
    piecetype = 0..typemax;
    position = 0..size;
*/
    /* Bubble, Quick */ /*
    listsize = 0..sortelements;
    sortarray = array [listsize] of integer;
*/
    /* FFT */
struct    complex { float rp, ip; } ;
/*
    carray = array [1..fftsize] of complex ;
    c2array = array [1..fftsize2] of complex ;
*/

float value, fixed, floated;

    /* global */
long    seed;  /* converted to long for 16 bit WR*/

    /* Perm */
int    permarray[permrange+1];
/* converted pctr to unsigned int for 16 bit WR*/
unsigned int    pctr;

    /* tree */
struct node *tree;

    /* Towers */
int         stack[stackrange+1];
struct element    cellspace[maxcells+1];
int    freelist,  movesdone;

    /* Intmm, Mm */

int   ima[rowsize+1][rowsize+1], imb[rowsize+1][rowsize+1], imr[rowsize+1][rowsize+1];
float rma[rowsize+1][rowsize+1], rmb[rowsize+1][rowsize+1], rmr[rowsize+1][rowsize+1];

    /* Puzzle */
int     piececount[classmax+1], class[typemax+1], piecemax[typemax+1];
int     puzzl[size+1], p[typemax+1][size+1], n, kount;

    /* Bubble, Quick */
int sortlist[sortelements+1], biggest, littlest, top;

    /* FFT */
struct complex    z[fftsize+1], w[fftsize+1], e[fftsize2+1];
float    zr, zi;

void Initrand () {
```

```
    seed = 74755L;   /* constant to long WR*/
}

int Rand () {
    seed = (seed * 1309L + 13849L) & 65535L;  /* constants to long WR*/
    return( (int)seed );      /* typecast back to int WR*/
}


    /* Sorts an array using bubblesort */

void bInitarr() {
        int i;
        long temp; /* converted temp to long for 16 bit WR*/
        Initrand();
        biggest = 0; littlest = 0;
        for ( i = 1; i <= srtelements; i++ ) {
            temp = Rand();
            /* converted constants to long in next stmt, typecast back to int WR*/
            sortlist[i] = (int)(temp - (temp/100000L)*100000L - 50000L);
            if ( sortlist[i] > biggest ) biggest = sortlist[i];
            else if ( sortlist[i] < littlest ) littlest = sortlist[i];
        }
}

void Bubble(int run) {
        int i, j;
        bInitarr();
        top=srtelements;

        while ( top>1 ) {

                i=1;
                while ( i<top ) {

                        if ( sortlist[i] > sortlist[i+1] ) {
                                j = sortlist[i];
                                sortlist[i] = sortlist[i+1];
                                sortlist[i+1] = j;
                        }
                        i=i+1;
                }

                top=top-1;
        }
        if ( (sortlist[1] != littlest) || (sortlist[srtelements] != biggest) )
        printf ( "Error3 in Bubble.\n");
        printf("%d\n", sortlist[run + 1]);
}

int main()
{
        int i;
        for (i = 0; i < 100; i++) Bubble(i);
        return 0;
}
```

5. N-reines

Extrait II.6 queens.c

```
#include <stdio.h>
#include <stdlib.h>

#define   nil             0
#define   false           0
#define   true            1
#define   bubblebase      1.61f
#define   dnfbase         3.5f
#define   permbase        1.75f
#define   queensbase 1.83f
#define   towersbase 2.39f
#define   quickbase       1.92f
#define   intmmbase       1.46f
#define   treebase        2.5f
#define   mmbase          0.0f
#define   fpmmbase        2.92f
#define   puzzlebase      0.5f
#define   fftbase         0.0f
#define   fpfftbase       4.44f
    /* Towers */
#define maxcells          18

    /* Intmm, Mm */
#define rowsize           40

    /* Puzzle */
#define size              511
#define classmax          3
#define typemax           12
#define d                     8

    /* Bubble, Quick */
#define sortelements 5000
#define srtelements  500

    /* fft */
#define fftsize           256
#define fftsize2          129
/*
type */
    /* Perm */
#define permrange     10

   /* tree */
struct node {
        struct node *left,*right;
        int val;
};

    /* Towers */ /*
    discsizrange = 1..maxcells; */
#define     stackrange    3
/*    cellcursor = 0..maxcells; */
struct    element {
        int discsize;
```

```
        int next;
};
/*    emsgtype = packed array[1..15] of char;
*/
    /* Intmm, Mm */ /*
    index = 1 .. rowsize;
    intmatrix = array [index,index] of integer;
    realmatrix = array [index,index] of real;
*/
    /* Puzzle */ /*
    piececlass = 0..classmax;
    piecetype = 0..typemax;
    position = 0..size;
*/
    /* Bubble, Quick */ /*
    listsize = 0..sortelements;
    sortarray = array [listsize] of integer;
*/
    /* FFT */
struct    complex { float rp, ip; } ;
/*
    carray = array [1..fftsize] of complex ;
    c2array = array [1..fftsize2] of complex ;
*/

float value, fixed, floated;

    /* global */
long    seed;  /* converted to long for 16 bit WR*/

    /* Perm */
int    permarray[permrange+1];
/* converted pctr to unsigned int for 16 bit WR*/
unsigned int    pctr;

    /* tree */
struct node *tree;

    /* Towers */
int        stack[stackrange+1];
struct element    cellspace[maxcells+1];
int    freelist, movesdone;

    /* Intmm, Mm */

int   ima[rowsize+1][rowsize+1], imb[rowsize+1][rowsize+1], imr[rowsize+1][rowsize+1];
float rma[rowsize+1][rowsize+1], rmb[rowsize+1][rowsize+1], rmr[rowsize+1][rowsize+1];

    /* Puzzle */
int     piececount[classmax+1], class[typemax+1], piecemax[typemax+1];
int     puzzl[size+1], p[typemax+1][size+1], n, kount;

    /* Bubble, Quick */
int sortlist[sortelements+1], biggest, littlest, top;

    /* FFT */
struct complex    z[fftsize+1], w[fftsize+1], e[fftsize2+1];
float    zr, zi;

void Initrand () {
```

```
    seed = 74755L;   /* constant to long WR*/
}

int Rand () {
    seed = (seed * 1309L + 13849L) & 65535L;  /* constants to long WR*/
    return( (int)seed );      /* typecast back to int WR*/
}


    /* The eight queens problem, solved 50 times. */
/*
        type
            doubleboard =   2..16;
            doublenorm  =   -7..7;
            boardrange  =   1..8;
            aarray      =   array [boardrange] of boolean;
            barray      =   array [doubleboard] of boolean;
            carray      =   array [doublenorm] of boolean;
            xarray      =   array [boardrange] of boardrange;
*/

void Try(int i, int *q, int a[], int b[], int c[], int x[]) {
        int     j;
        j = 0;
        *q = false;
        while ( (! *q) && (j != 8) ) {
                j = j + 1;
                *q = false;
                if ( b[j] && a[i+j] && c[i-j+7] ) {
                        x[i] = j;
                    b[j] = false;
                    a[i+j] = false;
                    c[i-j+7] = false;
                    if ( i < 8 ) {
                        Try(i+1,q,a,b,c,x);
                                if ( ! *q ) {
                                        b[j] = true;
                                    a[i+j] = true;
                                    c[i-j+7] = true;
                                }
                        }
                    else *q = true;
            }
        }
}

void Doit () {
        int i,q;
        int a[9], b[17], c[15], x[9];
        i = 0 - 7;
        while ( i <= 16 ) {
                if ( (i >= 1) && (i <= 8) ) a[i] = true;
            if ( i >= 2 ) b[i] = true;
            if ( i <= 7 ) c[i+7] = true;
            i = i + 1;
        }

        Try(1, &q, b, a, c, x);
        if ( !q ) printf ("␣Error␣in␣Queens.\n");
}
```

```
void Queens (int run) {
    int i;
    for ( i = 1; i <= 50; i++ ) Doit();
         printf("%d\n", run + 1);
}

int main()
{
        int i;
        for (i = 0; i < 100; i++) Queens(i);
        return 0;
}
```

BIBLIOGRAPHIE

Arvind, D., Mullins, R. D. & Rebello, V. E. (1995). Micronets : A model for decentralising control in asynchronous processor architectures. *Asynchronous design methodologies, 1995. proceedings., second working conference on*, pp. 190–199.

Beaty, S. J., Colcord, S. & Sweany, P. H. (1996). Using genetic algorithms to fine-tune instruction-scheduling heuristics. *Proceedings of the international conference on massively parallel computer systems*.

Bink, A. & York, R. (2007). Arm996hs : the first licensable, clockless 32-bit processor core. *Ieee micro*, 27(2), 58–68.

Brunvand, E. (1993). The nsr processor. *System sciences, 1993, proceeding of the twenty-sixth hawaii international conference on*, 1, 428–435.

Chaitin, G. (2004). Register allocation and spilling via graph coloring. *Acm sigplan notices*, 39(4), 66–74.

Chase, M. (2006). On the near-optimality of list scheduling heuristics for local and global instruction scheduling.

Cho, K.-R., Okura, K. & Asada, K. (1992). Design of a 32-bit fully asynchronous microprocessor (fam). *Circuits and systems, 1992., proceedings of the 35th midwest symposium on*, pp. 1500–1503.

Dean, M. E. (1992). *Strip : a self-timed risc processor*. (Thèse de doctorat, Stanford University).

Division, T. J. W. I. R. C. R., Lawler, E., Lenstra, J., Martel, C., Simons, B. & Stockmeyer, L. (1987). *Pipeline scheduling : A survey*.

Ertl, M. A. & Krall, A. (1991). Optimal instruction scheduling using constraint logic programming. *International symposium on programming language implementation and logic programming*, pp. 75–86.

Furber, S. B., Day, P., Garside, J. D., Paver, N. C., Temple, S. & Woods, J. V. (1994). The design and evaluation of an asynchronous microprocessor. *Computer design : Vlsi in computers and processors, 1994. iccd'94. proceedings., ieee international conference on*, pp. 217–220.

Furber, S. B., Garside, J. D. & Gilbert, D. A. (1998). Amulet3 : A high-performance self-timed arm microprocessor. *Computer design : Vlsi in computers and processors, 1998. iccd'98. proceedings. international conference on*, pp. 247–252.

Furber, S. B., Garside, J. D., Riocreux, P., Temple, S., Day, P., Liu, J. & Paver, N. C. (1999). Amulet2e : An asynchronous embedded controller. *Proceedings of the ieee*, 87(2), 243–256.

Garside, J. & Furber, S. (2007). Asynchronous and self-timed processor design. Dans *Processor Design* (pp. 367–389). Springer.

Gibbons, P. B. & Muchnick, S. S. (1986). Efficient instruction scheduling for a pipelined architecture. *Acm sigplan notices*, 21(7), 11–16.

Horowitz, M., Indermaur, T. & Gonzalez, R. (1994, Oct). Low-power digital design. *Proceedings of 1994 ieee symposium on low power electronics*, pp. 8-11. doi : 10.1109/LPE.1994.573184.

Kerns, D. R. & Eggers, S. J. (1993). Balanced scheduling : Instruction scheduling when memory latency is uncertain. *Acm sigplan notices*, 28(6), 278–289.

Kouveli, G., Kourtis, K., Goumas, G. & Koziris, N. (2011). Exploring the Benefits of Randomized Instruction Scheduling. GROW.

Lattner, C. (2006). Introduction to the llvm compiler infrastructure. *Itanium conference and expo*.

Laurence, M. (2012). Introduction to octasic asynchronous processor technology. *Asynchronous circuits and systems (async), 2012 18th ieee international symposium on*, pp. 113–117.

Laurence, M. (2013). Low-power high-performance asynchronous general purpose armv7 processor for multi-core applications. *13th international forum on embedded mpsoc and multicore*, 304-314.

Leupers, R. & Marwedel, P. (1997). Time-constrained code compaction for dsps. *Ieee transactions on very large scale integration (vlsi) systems*, 5(1), 112–122.

Lopes, B. C. & Auler, R. (2014). *Getting started with llvm core libraries*. Packt Publishing Ltd.

Malik, A. M., McInnes, J. & Van Beek, P. (2008). Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. *International journal on artificial intelligence tools*, 17(01), 37–54.

Martin, A. J. (1989). The design of an asynchronous microprocessor.

Nowick, S. M. & Singh, M. (2011). High-performance asynchronous pipelines : an overview. *Ieee design & test of computers*, 28(5), 8–22.

Rabaey, J. M., Chandrakasan, A. P. & Nikolic, B. (2002). *Digital integrated circuits*. Prentice hall Englewood Cliffs.

Scholz, B. & Eckstein, E. (2002). *Register allocation for irregular architectures*. ACM.

Smotherman, M., Krishnamurthy, S., Aravind, P. & Hunnicutt, D. (1991). Efficient dag construction and heuristic calculation for instruction scheduling. *Proceedings of the 24th annual international symposium on microarchitecture*, pp. 93–102.

Sotelo-Salazar, S. (2003). Instruction scheduling in micronet-based asynchronous ilp processors.

Srikant, Y. & Shankar, P. (2007). *The compiler design handbook : optimizations and machine code generation*. CRC Press.

Tremblay, J.-P. (2009). *Analyse de performance multi-niveau et partionnement d'application radio sur une plateforme multiprocesseur*. (Thèse de doctorat, École Polytechnique de Montréal).

Van Beek, P. & Wilken, K. (2001). Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. *International conference on principles and practice of constraint programming*, pp. 625–639.

Vijay, J. V. & Bansode, B. (2015). Arm processor architecture. *International journal of science, engineering and technology research (ijsetr), volume 4, issue 10*.

Weicker, R. P. (1984). Dhrystone : a synthetic systems programming benchmark. *Communications of the acm*, 27(10), 1013–1030.

Werner, T. & Akella, V. (1997). Asynchronous processor survey. *Computer*, 30(11), 67–76.

Wilken, K., Liu, J. & Heffernan, M. (2000). Optimal instruction scheduling using integer programming. *Acm sigplan notices*, 35(5), 121–133.