THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

SPÉCIALITÉ INFORMATIQUE

Présentée par Maxime PALUS
Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PARIS VI

ÉTUDE ET VALIDATION DE L'ARCHITECTURE D'UNE MACHINE JAVA DE HAUTES PERFORMANCES

Présentée le 14 novembre 2006, devant le jury composé de

Jean-Pierre SCHOELLKOPF	Rapporteur
Bernard LÉCUSSAN	Rapporteur
Alain GREINER	Examinateur
Patrice QUINTON	Examinateur
Nathalie DRACH	Examinateur
François ANCEAU	Directeur

Avant-propos

Cette thèse a été effectuée au sein du laboratoire d'informatique de Paris VI (LIP6) de l'université Pierre et Marie Curie, dans le département d'Architecture des Systèmes Intégrés et Micro-électronique (ASIM), sous la direction du Professeur F. Anceau (CNAM : Chaire des Techniques Fondamentales de l'Informatique). Je tiens à le remercier pour m'avoir fait confiance au cours de ces années, d'avoir su me motiver dans les périodes de doute, et pour le soin avec lequel il a suivi la rédaction de ce manuscrit.

J'exprime aussi toute ma reconnaissance au Professeur A. Greiner (LIP6 ASIM) pour m'avoir accueilli au sein du département ASIM, puis d'avoir accepté d'examiner ce travail et de participer au jury.

Je remercie Messieurs J.P. Schoellkopf (Directeur "Advanced design" ST Microelectronics) et le Professeur B. Lécussan (CNAM : Chaire Informatique et Programmation) pour l'intérêt qu'ils ont montré envers mon travail, en acceptant d'être les rapporteurs de cette thèse.

Je remercie également Les Professeurs N. Drach, (LIP6 ASIM) et M P. Quinton (ENS Cachan) d'avoir bien voulu examiner ce travail et participer au jury.

Je remercie Jérôme Dumesnil et Frédéric Arzel Pour avoir développé respectivement l'unité de prédiction de branchement et le cache instructions des modèles de simulation.

Je remercie tous mes amis du laboratoire, thésards ou permanents, avec qui j'ai travaillé dans la bonne humeur. Plus particulièrement, Laurent De Lamarre, Nicolas Beilleau, Etienne Faure, Richard Buchmann, Christophe Alexandre, Vincent Bourguet, Hugo Clément, Jean-Paul Chaput, Franck Wajsburt et François Pecheux.

Je remercie mes amis pour avoir fait semblant de comprendre en quoi consistaient mes travaux même après des explications pas toujours très claires.

Je remercie ma famille pour le soutien qu'elle m'a apporté tout au long de mes études.

Enfin, je remercie mon amie Aurélie Bédoucha qui a su me motiver en toutes circonstances.

Résumé

Le langage Java est très utilisé par les concepteurs d'applications mobiles et sans fil. Son exécution sur les processeurs embarqués se heurte à la lenteur de son interpréteur (JVM). Un certain nombre de fabricants se sont donc tournés vers l'exécution directe du code intermédiaire Java (Bytecode) par des processeurs spécialisés. Ceux-ci sont basés sur des machines à pile et intègrent certains mécanismes destinés à accélérer l'exécution du Bytecode. Cette thèse présente une nouvelle architecture appelée JMQ (Java Machine on Queue) dont la pile d'exécution Java est remplacée par une file. Cette approche permet d'augmenter le parallélisme de l'exécution de Java. La JMQ est comparée à un modèle de machine à pile appelé JMS (Java Machine on Stack) basé sur le PicoJava-II développé par Sun Microsystems. Par rapport à la machine sur pile, les résultats montrent un facteur d'accélération de l'exécution des applications Java de 1,05 à 1,69 selon les programmes testés. Ceci est obtenu au prix d'une augmentation de la complexité matérielle de l'ordre de 13%.

Mots clefs

Java, Processeur, Exécution post-fixé.

Abstract

The Java language is used by many embedded and wireless applications developers. Its execution on the embedded processors runs up against the slowness of its interpretor (JVM). Some manufacturers thus turned to direct execution of the intermediate Java code (Bytecode) by dedicated processors. These processors are based on stack architectures and have some integrated mechanisms to speed up the Bytecode execution. This thesis is about a new architecture called JMQ (Java Machine on Queue) which uses a queue to implement the Java execution stack. This approach allows increasing the Java execution parallelism. The JMQ is compared to a Java stack processor called JMS (Java Machine on Stack) based on the PicoJava-II architecture developed by Sun Microsystems. The JMQ allows speeding the execution of Java program up from 1.05 to 1.69 (depending on the programs). The hardware complexity is increased by 13% compared to that of the JMS.

Keywords

Java, Processor, Post-fixed execution.

Table des matières

Table des figures

Liste des tableaux

Introduction

Java est un langage de programmation orienté objet très connu. Une des clefs de son succès est sa grande portabilité due au fait que les programmes Java sont compilés dans un code intermédiaire appelé Bytecode. Celui-ci est interprété ou exécuté sur un grand nombre de systèmes, il est compact (1,8 octets en moyenne par instruction) et il peut être sécurisé. Java est aujourd'hui l'un des langages les plus utilisés pour développer des applications destinées aux systèmes embarqués.

La complexité croissante des applications Java embarquées demande de plus en plus de puissance de calcul. Notamment pour les jeux. Ceci entraîne, à l'instar des processeurs des machines de bureau, la recherche de nouvelles architectures matérielles capables d'améliorer les performances des processeurs.

Les techniques logicielles d'accélération de l'exécution de Java ne sont pas adaptées aux systèmes embarqués.

Etant donné que l'exécution directe du Bytecode Java, en tant que jeu d'instruction, permet d'augmenter la vitesse d'exécution des applications indépendamment de la technologie utilisée, cette thèse va s'attacher à étudier les moyens d'améliorer les mécanismes d'exécutions matérielles sur des processeurs spécialisés.

Ce travail doit tout d'abord permettre de répondre aux questions suivantes :

- Quelles sont les techniques d'accélérations de l'exécution utilisées sur les processeurs Java existants ?
- Quels nouveaux mécanismes peuvent être proposés ?

L'idée développée dans cette thèse est d'appliquer la technique d'exécution sur FILE, déjà utilisée sur une machine Pascal, pour l'exécution du Bytecode Java. Pour cela, il s'agit de développer un modèle de processeur Java fondé sur cette technique afin de répondre aux questions suivantes :

- La technique d'exécution sur FILE est-elle applicable à Java ?
- Quels sont les mécanismes à mettre en place ?
- Quel est le gain de performance obtenu ?
- Quel est le coût matériel ?

Le Chapitre 1 vise à fournir au lecteur les rappels essentiels concernant le langage Java et sa prééminence dans certaines applications afin de favoriser la compréhension de la suite de la démarche entreprise. Nous y présenterons le langage Java, l'intérêt de son utilisation dans les systèmes embarqués, ainsi que les techniques liées à son exécution.

Nous décrirons ensuite, dans le Chapitre 2, la spécification de la machine virtuelle Java (JVM), et les sous-ensembles de ce langage. Nous présenterons également un sous-ensemble des processeurs Java étudiés ou réalisés ainsi que les techniques utilisées pour augmenter leurs performances. Nous introduirons aussi les bases des techniques d'exécution sur Pile et sur File.

Dans le Chapitre 3, nous présenterons l'architecture de notre processeur à exécution sur file appelé JMQ (Java Machine on Queue) développé au cours de cette thèse. Nous

présenterons également l'architecture du modèle de comparaison appelé JMS (Java Machine on Stack) semblable au Picojava-II de Sun Microsystems.

Le Chapitre 4 décrira de façon détaillée les différents modules présents dans ces processeurs.

Le Chapitre 5 présentera les résultats de la simulation de ces machines sur plusieurs programmes Java. Nous y étudierons l'impact des mécanismes mis en place sur leurs performances. Nous évaluerons également le coût matériel de ces architectures.

Nous conclurons dans le Chapitre 6 où nous aborderons les perspectives de développement liées à la poursuite de notre projet.

Chapitre 1 Le Java et les systèmes embarqués

1.1 - Introduction

Dans ce chapitre, nous présenterons les systèmes embarqués, les évolutions du matériel utilisé et des applications associées. Nous verrons l'intérêt d'utiliser le langage Java pour de tels systèmes. Nous présenterons ensuite le langage Java ainsi que les techniques liées à son exécution. Enfin, nous discuterons du besoin d'accélération de l'exécution des applications Java embarquées et de la recherche de nouvelles architectures matérielles permettant cette accélération.

1.2 - Les systèmes embarqués

Les systèmes embarqués sont des systèmes mixtes contenant du matériel et du logiciel intégrés sur des circuits monolithiques (SOC : System On Chip). Ils sont généralement autonomes et dédiés à des applications spécifiques.

En 1999, 95% des processeurs vendus étaient destinés à l'embarqué, une grande partie de ceux-ci étaient des processeurs 8 bits. Les progrès des techniques de fabrication des circuits intégrés et des outils CAO (Conception Assistée par Ordinateur) ont permis de réduire la taille des systèmes embarqués tout en augmentant leurs fréquences de fonctionnement ainsi que d'étendre leurs domaines d'application. Aujourd'hui, un cœur de CPU de 16 ou 32 bits ne représente qu'une augmentation de quelques pour cent de la surface donc du coût par rapport à un cœur 8 bits. Le marché évolue d'ailleurs vers la généralisation de l'utilisation des processeurs 32 bits dans les systèmes l'embarqués (Figure 1-1 [1]).

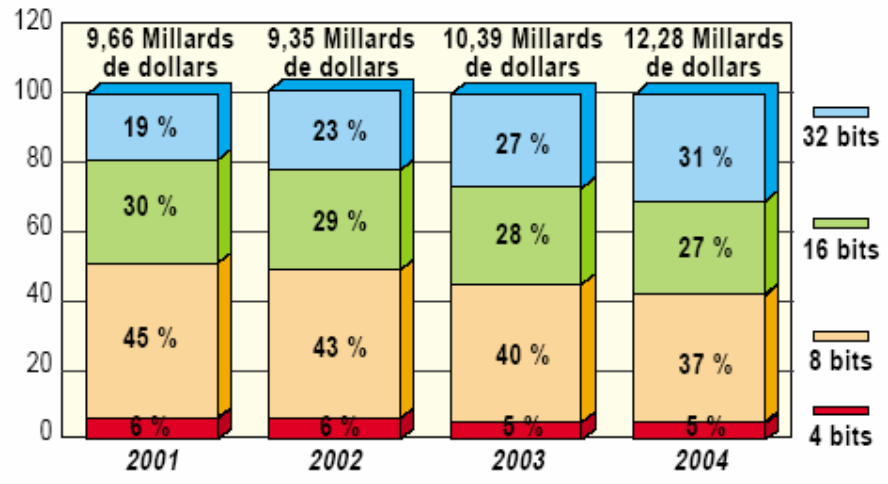


Figure 1-1 - Répartition des ventes de microcontrôleur.

Les systèmes embarqués connaissent une progression importante dans tous les domaines d'applications, des télécommunications (et en particulier la téléphonie mobile) à la robotique en passant par l'automobile et la défense. La société d'étude Frost & Sullivan estime que la part des composants électroniques dans les automobiles (comme ceux utilisés pour la sécurité et la navigation) devrait croître de 25 à 40% au cours des six prochaines années. Les marchés des téléphones portables et des PDAs (Personnal Digital Assistant) sont également en constante progression.

L'explosion de la demande de systèmes incluant des microprocesseurs et la diversité des tâches et des applications concernées ont accru le besoin "d'intelligence" embarquée. Cette intelligence doit permettre aux systèmes d'être adaptatifs et évolutifs. Elle doit également permettre de communiquer avec d'autres dispositifs ou d'autres applications fonctionnant sur la même plate-forme.

Les fabricants de systèmes embarqués se tournent aujourd'hui vers le langage Java pour programmer l'intelligence embarquée. En effet, le langage Java est un langage de programmation orienté-objet et indépendant de la plate-forme sur lequel il s'exécute. Il est particulièrement adapté à l'écriture d'applications et de jeux qui deviennent ainsi indépendants de la plate-forme. L'orientation objet est une manière puissante de structurer les applications. Le langage C++ est également orienté-objet mais il dérive du langage C qui est plus "adhérent" aux plates-formes et laisse la possibilité d'écrire des programmes peu sûrs. En revanche, la plupart des programmeurs constatent qu'il est plus facile de programmer en Java, qui est plus purement orienté-objet. Une étude montre qu'un programmeur est approximativement 40% plus productif lorsqu'il écrit des programmes en Java plutôt qu'en C++ [2]. De plus, depuis décembre 2005, le Java est devenu le langage dominant sur SourceForge [3], un site dédié aux projets "open source". Selon une étude récente [4], 56% des développeurs d'applications sans fil utilisent le Java J2ME (Java 2 Mobile Edition) (2.2.c -), ce chiffre monte à 66% pour l'utilisation de Java sous toutes ses formes.

1.3 - JAVA

Java est un langage de programmation orienté-objet très populaire qui doit son succès à certaines de ses caractéristiques. En effet, c'est un langage simple, qui peut être sécurisé et dont la philosophie est "Write Onces, Run Anywhere" (WORA) : "compilé une fois, exécuté partout". Ce langage est particulièrement adapté à son utilisation sur des réseaux et sur Internet car il est portable. La portabilité de Java lui a permis d'être disséminé dans tous les types d'applications et sur un grand nombre de support. Ainsi, de nos jours un grand nombre de téléphones portables ou de PDA se targuent de permettre l'utilisation de Java.

1.3.a - Historique

L'histoire du Java commence en décembre 1990 par la création d'un groupe de programmeurs de Sun Microsystems, "The Green Project", chargé d'anticiper la convergence entre les appareils électroniques et les ordinateurs [5]. Dans le cadre de ce projet, James Gosling a créé le langage de programmation "Oak" orienté-objet, robuste et sûr et spécifiquement conçu pour être indépendant des plates-formes. Oak est alors destiné aux applications embarquées.

En août 1992, le président de Sun Microsystems assiste à une démonstration des travaux de l'équipe "Green Project" sur le Star7, un PDA basé sur un processeur SPARC.

L'industrie de la télévision par câble s'intéresse au projet, ce qui lui fait prendre de l'ampleur. L'équipe est réorganisée en tant que filiale distincte de Sun Microsystems, sous le nom de "FirstPerson". L'objectif étant de vendre le langage Oak et des produits qui l'utilisent, aux principaux fabricants de produits électroniques grand public.

FirstPerson entame des négociations avec plusieurs sociétés et se propose de développer des boîtiers de programmation à la demande - sensés représenter la future TV interactive - pour TimeWarner. Ce marché n'est pas conclu car le marché de l'électronique grand public n'était pas réellement prêt pour cette offre. En 1994, le monde est pris par la fièvre d'Internet, et les technologies réseau prennent un tournant décisif, s'éloignant de la recherche, du monde universitaire et de l'informatique, pour devenir des biens de consommation.

Sous l'impulsion de Bill Joy, co-fondateur de Sun, qui voyait le potentiel d'un environnement de programmation sûr et indépendant des plates-formes sur Internet, Sun décide de sauver le projet Oak. Sa mission était simple : faire de Oak le langage de programmation du réseau Internet.

En janvier 1995, Oak, étendu et amélioré, est rebaptisé "Java" et distribué gratuitement à la communauté Internet. Ce nom aurait été trouvé dans un café fréquenté par plusieurs membres de l'équipe [6], "java" désignant le café en argot américain. Le langage est renforcé et d'autres niveaux de sécurité sont ajoutés. Les utilisateurs et les diffuseurs du Web, frustrés par l'interactivité limitée de HTML (Hypertext Markup Language) et CGI (Common Gateway Interface), accueillent cette nouvelle philosophie avec enthousiasme. Sun adopte une attitude "ouverte" en donnant le compilateur, les spécifications, les bibliothèques et la documentation. La disponibilité de Java sur Internet permet à tous les développeurs de travailler dans un environnement de qualité totalement non-propriétaire.

Aujourd'hui, le langage Java retrouve le domaine pour lequel il avait été conçu à l'origine en étant embarqué dans les appareils mobiles.

1.3.b - Caractéristiques

Pour Sun, Java est un langage simple, interprété, orienté objet, distribué, robuste, sûr, indépendant de l'architecture, portable, efficace, multi-thread, et dynamique [7] [8].

Simple

Le langage Java est conçu pour être appris facilement. Sa syntaxe est largement inspirée de celle de C/C++ et son écriture est simplifiée. En effet, un certain nombre de possibilités offertes par C/C++ sont interdites en Java (telles que : goto, structure, union, héritage multiple, surcharge d'opérateur, arithmétique sur les pointeurs, etc.).

De plus, contrairement à d'autres langages, un programme Java ne nécessite pas de fichier d'en-tête et sa compilation n'utilise pas de pré-processeur.

La gestion de la mémoire est elle-même simplifiée par un système d'allocation / désallocation automatique.

Interprété

Le compilateur Java génère un code intermédiaire post-fixé appelé *Bytecode* indépendant de toute architecture qui est ensuite interprété. Le langage Java est traduit en Bytecode par un compilateur (Figure 1-2). Il n'y a pas à proprement parler de phase d'édition des liens, celle-ci est remplacée par un mécanisme de liaison dynamique réalisé grâce à une table symbolique appelée "constant pool" propre à chaque classe. Le Bytecode est ensuite directement exécuté sur une machine virtuelle appelée JVM (Java Virtual Machine) qui peut être installée sur une grande variété de machines d'exécution.

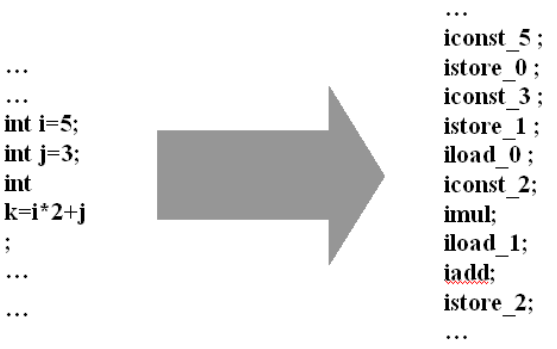


Figure 1-2 - Transformation du java en Bytecode.

Cette technique n'est pas originale. Elle a été largement utilisée dans les années soixante-dix pour le langage Pascal à partir duquel un code intermédiaire, post-fixé, indépendant des machines d'exécution, appelé P-Code était généré. Ce P-Code était ensuite exécuté sur toute machine munie d'un interpréteur simple, lui-même écrit initialement en Pascal. Comme pour Java, des implémentations matérielles de l'interpréteur Pascal ont été réalisées, y compris en exécution sur file (2.4.b - L'exécution sur FILE).

Orienté objet

La programmation orientée objet est un mode de programmation dans lequel les données et les procédures qui les manipulent sont regroupées en entités appelées "objets". On entend par "objet" une entité constituée d'un ensemble d'informations et de lois de comportement. Un objet est une "boîte noire" qui reçoit et transmet des *messages*. Le langage orienté-objet est fondé sur la communication entre objets. Un objet est défini via sa classe, qui détermine tout à propos de l'objet. Les objets sont des *instances* individuelles d'une classe.

Pour être réellement orienté-objet, un langage doit respecter au minimum 4 caractéristiques [5] :

- *L'encapsulation* : permet de séparer l'interface de l'implémentation. L'implémentation peut alors être modifiée sans modifier l'interface ou le comportement extérieur.
- *Le polymorphisme* : le même message envoyé vers différents objets retourne un résultat dépendant de la nature de l'objet recevant le message.
- *L'héritage* : on peut définir de nouvelles classes basées sur des classes existantes, pour obtenir du code réutilisable et de l'organisation.
- *Liens dynamiques* : les objets peuvent venir de n'importe où sur le réseau. Il faut être capable d'envoyer des messages à des objets sans connaître leur type spécifique au moment du codage. Les liens dynamiques permettent une grande flexibilité au moment de l'exécution.

La définition précédente de la programmation objet comporte une ambiguïté sur la façon dont les objets vont s'exécuter par rapport à l'émetteur des messages qu'ils reçoivent. L'interprétation directe de cette définition laisse penser que les objets fonctionnent concurremment aux émetteurs des messages (comme des processus). En fait, comme les principaux langages orientés objets dérivent de langages séquentiels, les envois de messages ne sont que des appels de procédure et les messages des blocs de données, ce qui correspond à un fonctionnement séquentiel de l'objet par rapport à son appelant, réduisant beaucoup la portée du concept.

Java est un langage plus purement orienté objet que le C++ qui dérive du C et permet d'écrire des programmes dans lesquels la notion d'objet est totalement absente. L'objet Java appartient à une classe. Cette classe est une collection de variables, de méthodes encapsulées dans un objet réutilisable et chargé dynamiquement en mémoire à l'exécution. Une classe n'est donc qu'une collection de code qui modélise sous forme logicielle le comportement d'objets. Le concept d'héritage de classe est respecté puisqu'il est possible de dériver une nouvelle classe d'objets à partir d'une classe originale dont on conserve les fonctionnalités en les étendant. Le chargement automatique de classes Java est essentiel pour son utilisation sur Internet. Cela permet, lors du chargement d'un type de fichier ou document inconnu, de télécharger le code qui permettra de traiter ou visualiser ce fichier.

Robuste

Un langage ne peut, à lui seul, garantir la fiabilité des applications, mais il peut plus ou moins faciliter le développement d'applications fiables. En Java, la gestion des exceptions est beaucoup plus stricte qu'en C++.

Java se veut robuste dans la mesure où le langage, fortement typé, permet moins de "bidouillages" que le langage C. Beaucoup d'erreurs classiques de programmation en C comme le dépassement de tableau sont impossibles. C et C++ sont plus difficilement portables d'une plate-forme à l'autre puisqu'ils permettent l'usage de pointeurs pour directement adresser des portions de mémoire et ne gèrent pas automatiquement la désallocation de la mémoire. Cela débouche sur des programmes syntaxiquement corrects mais qui peuvent néanmoins provoquer des erreurs système. Pour garantir la robustesse de Java, celui-ci a été débarrassé des pointeurs : il est donc impossible de référencer une adresse mémoire particulière ou d'utiliser l'arithmétique de pointeurs pour se déplacer dans des tableaux. De plus, Java vérifie toujours, lors de l'utilisation de tableaux, que le programmeur ne cherche pas à utiliser des portions non allouées de la mémoire. Néanmoins, le Java est un langage orienté-objet : tous les objets sont adressés via leurs références, celles-ci sont constantes, par conséquent, elles ne peuvent ni être modifiées, ni être utilisées comme base de déplacement dans un tableau.

La gestion de la mémoire et sa désallocalion automatique, via un "ramasse-miettes" appelé "garbage collector", sont les garants d'une meilleure stabilité des applications. Ce garbage collector fonctionne comme un processus séparé tournant en tâche de fond avec une basse priorité. Il désalloue des pans de mémoire lorsque l'utilisation du CPU est faible ou lorsqu'un besoin de mémoire se fait sentir.

Distribué

La plate-forme Java est conçue pour supporter des applications réparties sur le réseau, elle fût l'un des premiers systèmes permettant l'exécution de programmes à partir d'une source distante. Une applet peut fonctionner sur un navigateur web local et exécuter le code téléchargé depuis un serveur distant.

Sûr

La première tâche de l'interpréteur Java est de vérifier la conformité du programme, analyser si le code n'a pas été transformé entre la compilation et l'exécution du code. Un "Bytecode verifier" se charge d'observer si une classe n'a pas accès aux registres, ne manipule pas la pile ou n'accède pas au système de fichier de manière anormale. De plus, Java utilise pour ses transferts une technique de cryptage basée sur un système de clé publique. Le chargement d'une classe Java via le réseau s'opère par le biais d'un "Class-Loader" qui gère tous les mécanismes de sécurité. Les applets chargées via le réseau sont restreintes au niveau :

- des accès en lecture et écriture des fichiers locaux.
- de la création et de la copie de répertoires ou de fichiers.
- de l'établissement des connexions réseau à l'exception de la machine hôte.
- de l'exécution de programmes externes via fork (création d'un programme fils) ou exec (recouvrement par un autre programme).

Le gestionnaire de sécurité (la classe *SecurityManager*) permet de définir les droits d'accès aux ressources (lecture, écriture, accès réseau…) d'un programme.

Les éditeurs d'applet peuvent obtenir des certificats (signature numérique) de sûreté de leurs applets, permettant à celles-ci d'accéder aux ressources du système local.

Une API très riche

Le langage Java possède une très grande bibliothèque de classes réutilisables appelées API (Application Programming Interface) permettant de faciliter la programmation, allant de la gestion du graphique aux protocoles réseaux en passant par les bases de données. Cette bibliothèque est très bien documentée. En effet, Java permet de générer une documentation sous forme html en utilisant les commentaires présents dans les fichiers contenant les classes java (fichier *.java)*. Ce qui rend l'utilisation de nouvelles classes plus facile, diminuant le temps de développement de nouvelles applications.

Portable et indépendant de l'architecture

Les applications Java peuvent fonctionner, sans recompilation, sur toute machine disposant de la machine virtuelle Java (JVM). Cette machine virtuelle exécute le Bytecode Java. L'interpréteur est écrit en C ANSI et le compilateur en Java. Il est donc relativement facile de les modifier pour les adapter aux différents environnements, permettant ainsi, de s'adapter à différentes architectures.

Les accès aux ressources du système se font de façon homogène à travers différentes APIs standard. Ainsi, l'utilisation de l'AWT (API graphique de Java) permet une manipulation homogène de l'interface graphique (GUI, Graphical User Interface) de la machine hôte locale.

L'indépendance de Java vis à vis de l'architecture est intéressante pour les applications réseau car le même programme va pouvoir fonctionner sur toutes les machines d'un réseau et ce quelles que soient les plates-formes cibles.

De plus, les types Java sont constants quelle que soit l'architecture, et sont vérifiés à la compilation Les types Java sont les suivant :

- boolean : true (vrai) ou false (faux).
- byte : entier 8 bits signé en complément à 2.
- short : entier 16 bits signé en complément à 2.
- int : entier 32 bits signé en complément à 2.
- long : entier 64 bits signé en complément à 2.
- char : entier 16 bit au format unicode (non-signé).
- float : flottant 32 bits au format IEEE 754.
- double : flottant 64 bits au format IEEE 754.
- référence : référence d'objet ou de tableau (adresse sur 32 bits).

L'API standard de Java est assez riche pour développer des applications complètes. Néanmoins, la portabilité du langage implique que les spécificités du système d'exploitation ou celles du matériel ne peuvent pas être prises en charge. Pour pouvoir utiliser ces spécificités, le langage Java fourni l'interface JNI (Java Native Interface) qui permet de charger du code C ou C++. Cette interface permet également de réutiliser du code non-Java que l'on ne souhaite pas redévelopper ou pour implémenter des parties de code ayant des contraintes temps réel fortes. Les méthodes dites natives ne sont évidemment pas portables.

Multi-threading

La notion de thread (processus léger) est très intégrée en Java. Contrairement à un grand nombre de langages (C, C++, Python, Perl …) qui ne peuvent s'appuyer que sur les threads du système d'exploitation, la JVM incorpore sa propre notion de thread. Elle est donc capable d'utiliser le multitraitement offert par le système d'exploitation mais également de simuler ce comportement dans le cas d'un système d'exploitation mono-tâche. L'API Java fournit une classe *Thread* et une interface *Runnable* incluant des méthodes de gestion des threads. En Java, le multi-threading est supporté au niveau du langage (primitives de

synchronisation, exclusion mutuelle). Les méthodes d'une classe déclarées *synchronized* ne peuvent pas s'exécuter en parallèle sur un même objet. Ces méthodes sont exécutées sous le contrôle du moniteur. Chaque classe et chaque objet possèdent un moniteur permettant de protéger les variables partagées. Les moniteurs Java sont ré-entrant pour pouvoir utiliser des méthodes récursives.

Dynamique

Les classes sont chargées en cours d'exécution et uniquement lorsque c'est nécessaire : il s'agit donc d'un chargement "paresseux". Bien que Java soit typé statiquement, les informations de typage sont disponibles à l'exécution, une grande partie des instructions du Bytecode Java étant elles mêmes typées. Lors de la modification d'une classe, il n'est pas nécessaire de recompiler tout le programme, seules les classes modifiées sont à recompiler.

1.4 - L'exécution de JAVA

Lors de l'apparition du langage Java, il n'y avait pas de processeur capable d'exécuter directement le Bytecode. L'unique moyen d'utiliser ce langage était alors de l'interpréter par du logiciel. Cet interpréteur est appelé "machine virtuelle" ou JVM (Java Virtual Machine) car il émule un processeur spécialisé.

1.4.a - La machine virtuelle JAVA (introduction)

La machine virtuelle Java est un programme d'émulation qui exécute le Bytecode Java sur le processeur physique de la machine hôte (Figure 1-3 [9]).

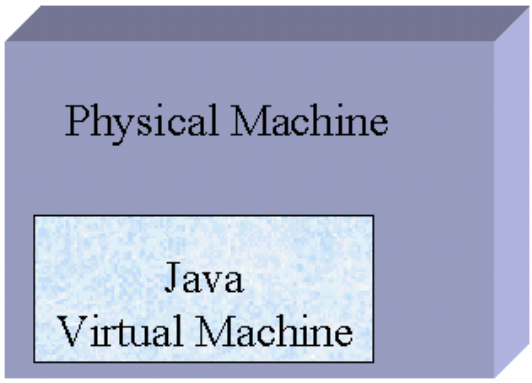


Figure 1-3 - Emulation de la JVM sur un processeur physique.

Cette machine virtuelle est responsable de l'interprétation du code Java pour qu'il soit exécuté sur la machine hôte : elle gère également tous des appels système permettant au programmeur de ne tenir compte ni de l'architecture, ni des aspects systèmes [9].

La JVM est partie intégrante du JRE (Java Runtime Environment) qui contient également les classes de base permettant à Java d'être exécuté dans un environnement spécifique (Figure 1-4 [9]). Ces classes de base contiennent des méthodes natives.

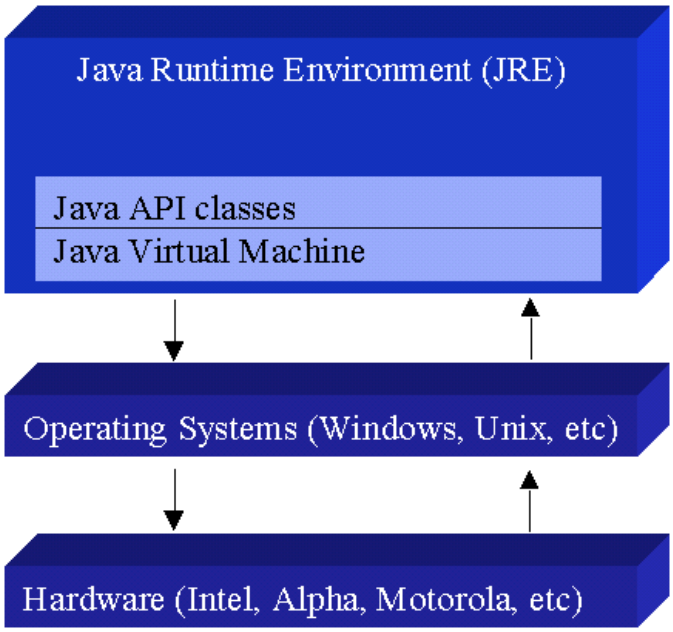


Figure 1-4 - Communication entre le JRE et le système hôte.

Le JRE est au cœur de la portabilité de Java, puisqu'il faut l'adapter pour chaque couple (architecture, système). Sans JRE disponible pour un environnement, il est impossible d'exécuter des programmes Java.

1.4.b - Les techniques d'accélération de l'exécution

La portabilité est un sérieux atout du langage Java, mais elle a néanmoins un effet non négligeable sur ses performances car l'interprétation du Bytecode engendre une lenteur d'exécution. Cette lenteur est due au fait que plusieurs instructions du processeur hôte sont nécessaires pour émuler chaque instruction du Bytecode Java. L'interprétation est une méthode peu adaptée aux processeurs embarqués car, en plus de sa lenteur, l'interpréteur demande beaucoup de bande passante pour le chargement des instructions depuis la mémoire. Pour remédier à cela, différentes techniques d'accélération de l'exécution de Java ont été développées.

Compilation Just-In-Time (JIT)

La première solution apportée à la lenteur d'exécution est le JIT (Just In Time) [10]. Cette technologie, présente dans la machine virtuelle Java, consiste en une compilation paresseuse du Bytecode dans le code de la machine hôte pendant son exécution. Comme Java est un langage dynamique, le JIT ne compile une classe que lors de son appel, "juste à temps". Lors des appels ultérieurs à cette classe, le code déjà compilé peut être directement réutilisé.

Le JIT permet donc une seconde exécution plus rapide que lors d'une simple interprétation. Toutefois, cette opération est très gourmande en mémoire : en effet, la place occupée est multipliée par un facteur d'environ 3, ce qui pose des problèmes pour les applications embarquées qui ne disposent pas de beaucoup de mémoire.

Compilation pour un système cible

Le gain apporté par le JIT est limité car certaines optimisations complexes ne peuvent pas être effectuées à la volée [11]. Il existe donc d'autres techniques plus performantes :

- Compilation "off-line" ou "a way ahead of time compilation"[12] : Le Bytecode est recompilé pour une machine cible. Une telle compilation est possible car le Bytecode contient presque toutes les informations disponibles dans le code source. Pour cela on doit disposer de toutes les classes et superclasses d'un programme.
- Compilation "native" : le programme Java est compilé directement pour une machine cible. Le compilateur peut alors exploiter toutes les informations présentes dans le code source afin d'optimiser le code généré. Pour cela, il faut disposer du programme en Java.

Ces deux techniques permettent un gain en performance important mais sont très loin de la philosophie de Java (WORA). Le chargement dynamique de classes pendant l'exécution n'est supporté que par très peu de compilateurs et nécessite l'ajout d'un interpréteur capable de réaliser cette fonction [13]. De plus, il faut développer des compilateurs pour chaque architecture et pour chaque système que l'on veut cibler.

Exécution directe

Le Bytecode généré par la compilation du Java peut être utilisé comme jeu d'instructions d'un processeur physique. L'exécution directe permet d'augmenter la vitesse d'exécution sans sacrifier la compacité ou la portabilité du code. De nombreux fabricants de processeurs destinés aux applications embarquées ont donc choisi cette solution pour l'exécution rapide du langage Java. Il existe de nombreuses implémentations de processeur Java utilisant différentes techniques d'exécution du Bytecode allant de la traduction pour un processeur généraliste à l'exécution par un processeur spécialisé. Ces techniques et une partie de ces processeurs Java seront détaillées dans le prochain chapitre.

1.5 - Conclusion

Dans ce chapitre nous avons présenté le langage Java. Nous avons rappelé l'historique de ce langage et sa popularisation par son utilisation sur Internet. Nous avons détaillé ses caractéristiques (simplicité d'écriture, portabilité, réutilisabilité, compacité du code, etc.), qui ont permis son utilisation dans tous les types d'applications et sur un grand nombre de plates-formes. Nous avons vu que ces mêmes caractéristiques conduisent les fabricants de systèmes embarqués à se tourner vers le langage Java pour programmer l'intelligence embarquée dans leurs systèmes. Ce langage est aujourd'hui très utilisé pour le développement d'applications embarquées pour lesquelles il avait initialement été développé. Cependant, les performances de Java sont bridées par la lenteur de son interprétation. Nous avons décrit les différentes techniques développées pour accélérer son exécution. Ces techniques sont aussi bien logicielles (JIT, compilation ""off-line et "native") que matérielles (traducteurs ou processeurs spécialisés). Elles apportent chacune différents avantages et inconvénients. L'exécution du Bytecode Java sur un processeur spécialisé est la technique la plus performante. De plus, celle-ci permet de ne sacrifier ni la portabilité ni la compacité du code Java contrairement aux techniques d'accélérations logicielles.

Ce chapitre montre l'intérêt d'utiliser un processeur spécialisé pour l'exécution performante des programmes Java.

Chapitre 2 Etat de l'art

2.1 - Introduction

Dans ce chapitre nous verrons plus en détail la machine virtuelle Java (appelé JVM), son jeu d'instructions (appelé Bytecode Java) et ses zones mémoires. Nous verrons ensuite les différents sous-ensembles de Java. Nous étudierons un sous-ensemble de processeurs exécutant le Bytecode Java existant ainsi que les techniques matérielles associées. Nous présenterons les aspects que l'on souhaite améliorer. Enfin nous verrons les techniques d'accélération que l'on souhaite mettre en place.

2.2 - La machine virtuelle Java (JVM)

La JVM est définie comme une machine abstraite qui exécute le Bytecode Java. La spécification de cette machine [14] ne définit pas une implémentation particulière mais deux éléments qui doivent être supportés par l'implémentation :

- Le jeu d'instruction Java : le Bytecode.
- Le format de fichier *.class* : fichier binaire contenant le Bytecode et toutes les informations relatives aux classes qu'il contient.

La JVM doit également permettre de charger, vérifier et lier des classes dynamiquement. Dans nos modèles de simulation, les classes seront chargées statiquement en mémoire via une représentation interne et liées dynamiquement à l'exécution. Dans notre projet, le chargeur de classes utilisé sera le ClassLoader développé par Sun pour le PicoJava-II. Nous avons modifié ce chargeur de classes afin de prendre en charge des instructions supplémentaires. Contrairement à sa spécification, ce chargeur ne fait aucune vérification sur les classes.

La mémoire de la JVM est divisée en plusieurs parties assimilables aux segments dans les processeurs "classiques". Nous nous intéresserons tout d'abord à l'organisation mémoire de la JMV puis à son jeu d'instructions.

2.2.a - Les zones mémoire de la JVM

Pendant l'exécution de la JVM, plusieurs zones mémoire sont définies. Certaines de ces zones existent pendant toute l'exécution d'un programme et d'autres sont attribuées à des threads. Ces dernières sont créées en même temps que les threads auxquelles elles sont attribuées et sont libérées lors de leurs arrêts.

Pile Java : "Java Stack"

Chaque thread dispose d'une "Java Stack" contenant les éléments suivants :

- La zone d'environnement appelée Frame contenant les informations permettant le retour à la méthode appelante. Ces informations correspondent aux registres liés à l'environnement et à l'exécution de la méthode appelante ainsi qu'à l'adresse de retour de la méthode courante.
- Une zone de variables locales qui contient les variables locales à une méthode.
- La pile d'exécution sur laquelle toutes les opérations sont effectuées.

Ces trois éléments sont généralement contenus dans une seule et même pile, mais elles ne doivent pas nécessairement être regroupées au même endroit en mémoire et peuvent donc être séparées lors de l'implémentation d'une JVM.

Le "tas" : Heap

Cette zone mémoire est partagée entre tous les threads, elle est créée à l'initialisation de la JVM. Elle contient les instances des objets et des tableaux (array). Le Heap doit être doté d'un mécanisme d'allocation mémoire et de désallocation automatique appelé "Garbage Collector". Le heap peut être étendu ou réduit pendant l'exécution en fonction des besoins. Il n'a pas l'obligation d'être une zone contiguë de mémoire.

Zone des méthodes : "Method Area"

Cette zone mémoire est partagée entre tous les threads. Elle est assimilable au segment "text" UNIX. Elle contient le code des méthodes des classes et un Constant Pool par classe. Le Constant Pool contient des informations de liaison symbolique permettant la résolution dynamique des références ainsi que les constantes liées aux méthodes.

Les registres liés à l'environnement d'une méthode sont les suivant :

- PC : Compteur ordinal de la méthode courante.
- FRAME : Pointe sur la zone d'environnement.
- VARS : Pointe sur la zone des variables locales à la méthode.
- OPTOP : Pointe le sommet de la pile d'exécution.
- CONST_POOL : Pointe sur le Constant pool de la méthode courante.

2.2.b - Le jeu d'instruction de la JVM

Le Bytecode Java contient 201 instructions. Pour une grande partie d'entre elles, le mnémonique qui leur est associé est préfixée par une lettre représentant le type de la donnée sur laquelle ces instructions travaillent. Ces préfixes sont les suivants :

- i : instruction agissant sur donnée de type "integer" (entier 32 bits signé).
- l : instruction agissant sur donnée de type "long" (entier 64 bits signé).
- f : instruction agissant sur donnée de type "float" (flottant 32 bits).
- d : instruction agissant sur donnée de type "double" (flottant 64 bits).
- b : instruction agissant sur donnée de type "byte" (entier 8 bits signé).
- s : instruction agissant sur donnée de type "short" (entier 16 bits signé).
- c : instruction agissant sur donnée de type "char" (entier 16 bits non-signé).
- a : instruction agissant sur donnée de type "array" (référence 32 bits).

Ainsi, deux instructions ayant des mnémoniques différents, tel que *iload* et *fload* peuvent avoir la même implémentation matérielle. Le typage des instructions n'intervient que pour la vérification des classes. Les mnémoniques des instructions sont sur un octet. La taille des instructions varie de un à cinq octets. Les instructions peuvent être classées en sept catégories décrites et commentées dans ce qui suit :

- chargement et rangement,
- arithmétiques,
- conversion de type,
- création ou manipulation d'objet,
- manipulation de la pile d'exécution,
- transfert de contrôle,
- appel et retour de méthode.

Instructions de chargement (load) et de rangement (store)

Pour les instructions de chargement et de mise à jour des variables locales, <n> correspond à un index dans la zone de variable locale de la méthode courante. Pour les instructions de chargement de constante, <i>, <l>, <f> et <d> correspondent à la constante à charger.

Les instructions appartenant à ce groupe sont :

- Les instructions chargeant dans la pile d'exécution une variable locale à la méthode courante : iload, iload_<n>, lload, lload_<n>, fload, fload_<n>, dload, dload_<n>, aload, aload_<n> (n ε [0,3]).
- Les instructions qui mettent à jour une variable locale à la méthode courante : istore, istore_<n>, lstore, lstore_<n>, fstore, fstore_<n>, dstore, dstore_<n>, astore, astore_<n> (n ε [0,3]).
- Les instructions qui chargent une constante dans la pile d'exécution : bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_<i> (i ε [0,5]), lconst_<l> (l ε [0,1]), fconst_<f> (f ε [0,2], dconst_<d> (d ε [0,2]).

Instructions arithmétiques

Les instructions arithmétiques agissent sur les opérandes contenus au sommet de la pile d'exécution. Le résultat obtenu est remis au sommet de cette pile. Ces instructions intègrent le type de données sur lequel elles s'effectuent. Cependant, il n'existe pas d'instruction agissant sur les types byte, short, char. Ces types sont donc traités avec les instructions arithmétiques destinées aux entiers. Le signe des données de type short et byte est étendu sur 32 bits lorsqu'elles sont insérées dans la pile d'exécution.

Les instructions arithmétiques sont :

- Addition : iadd, ladd, fadd, dadd.
- Soustraction : isub, lsub, fsub, dsub.
- Multiplication : imul, lmul, fmul, dmul.
- Division : idiv, ldiv, fdiv, ddiv.
- Modulo : irem, lrem, frem, drem.
- Négation : ineg, lneg, fneg, dneg.
- Décalage : ishl, ishr, iushr, lshl, lshr, iushr.
- Opération bit à bit : ior, lor, iand, land, ixor, lxor.
- Incrémentation de variable : iinc.
- Comparaison : dcmpg, dcmpl, fcmpg, fcmpl, lcmp.

Instructions de conversion de type

Ces instructions convertissent les données entre les différents types utilisés par Java.

On peut distinguer trois catégories suivantes :

- i2l, i2d, f2d : Conversion sans perte de précision.
- i2f, l2f, l2d : Conversion avec une perte de précision possible.
- i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, d2f : Conversion pouvant donner un résultat de signe différent ou de magnitude plus faible, voire une perte de précision.

Instructions de création ou de manipulation d'objet

La machine virtuelle crée et manipule les objets, via un ensemble distinct d'instructions.

- Création d'instance : new.
- Création de tableau : newarray, anewarray, multianewarray.
- Accès aux champs statiques des classes et aux champs non-statique des instances : putstatic, getstatic, putfield, getfield.
- Chargement d'une donnée à partir d'un tableau dans la pile d'exécution : baload, caload, saload, iaload, laload, faload, daload, aaload.
- Rangement d'une donnée dans un tableau à partir de la pile d'exécution : bastore, castore, sastore, iastore lastore, fastore, dastore, aastore.
- Chargement de la taille d'un tableau : arraylength.
- Vérification des propriétés d'une instance ou d'un tableau : instanceof, checkcast.

L'instruction *new* est remplacée pendant l'exécution par l'instruction *new_quick* (cf : Modification dynamique du Bytecode) après la résolution des liens via le constant pool. De même, les instructions d'accès aux champs des objets étant non-typés, une résolution pendant l'exécution du constant pool est nécessaire pour connaître le type du champ sur lequel elles s'appliquent. Ainsi, *putstatic* est remplacé par *putstatic_quick* pour un champ 32 bits et par *putstatic2_quick* pour un champ 64 bits. Les instructions *instanceof* et *checkcast* sont, elles aussi, remplacées à l'exécution (*instanceof_quick* et *checkcast_quick*).

Instructions de manipulation de la pile d'exécution

Ces instructions permettent de manipuler le sommet de la pile : *pop*, *pop2*, *dup*, *dup2*, *dup_x1*, *dup2_x1*, *dup_x2*, *dup2_x2*, *swap*.

Instruction de transfert de contrôle

Les instructions de transfert de contrôle conditionnelles ou inconditionnelles permettent de continuer l'exécution avec une instruction non-consécutive à l'instruction de transfert de contrôle.

Ces instructions sont :

- Les branchements conditionnels: ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne.
- Les branchements conditionnels composés : tableswitch, lookupswitch.
- Les branchements inconditionnels : goto, goto_w, jsr, jsr_w, ret.

Les branchements conditionnels de la JVM agissent sur les entiers et les références. Ainsi, les types byte, chart, short et boolean utilisent les branchements sur les entiers. Pour les autres types (float, double et long), les branchements sont également effectués par les instructions sur les entiers mais ils sont en plus précédés d'une instruction arithmétique de comparaison. Le résultat de ces dernières étant un entier. Toutes les instructions conditionnelles sur les entiers sont signées.

Instruction d'invocation et de retour de méthode

Les instructions d'invocations de méthode sont les suivantes :

- invokevirtual : invoque une méthode sur l'instance d'un objet en se basant sur son type (virtuel). C'est l'instruction d'invocation la plus courante du langage Java.
- invokeinterface : invoque une méthode implémentée dans une interface, le JVM cherche la méthode la plus appropriée pour l'objet sur lequel la méthode est invoquée.
- invokespecial : invoque le constructeur d'un objet, une méthode privée ou une une méthode d'une superclasse.
- invokestatic : invoque une méthode statique (méthode de classes).

Les instructions d'invocation sont aussi remplacées par leurs versions quick après résolution du constant pool. *invokespecial* est remplacé par *invokenonvirtual_quick* pour un constructeur ou une méthode privée et par *invokesuper_quick* pour une méthode d'une superclasse.

Les instructions de retour de méthode sont typées par le type de données à retourner. *return*, *ireturn*, *lreturn*, *freturn*, *dreturn* et *areturn*. Les types byte, char, short et boolean utilise l'instruction *ireturn*.

Modification dynamique du Bytecode

Un certain nombre d'instructions non typées de création d'objet et de tableau ainsi que les instructions de manipulation d'objet nécessitent la résolution du constant pool lors de leur exécution pour connaître les propriétés propres à l'objet ou au tableau. Une fois cette résolution effectuée, l'instruction peut être remplacée par sa version "quick". Lors de l'exécution suivante de l'instruction remplacée, cette dernière s'effectue plus vite, les liens étant déjà résolus.

2.2.c - Les déclinaisons de Java

Depuis 1998 et l'apparition de la version 1.2 de Java (ou Java 2), le Java se décline en trois éditions distinctes regroupant des APIs destinées à des domaines d'application différents.

Java 2 Standard Edition (J2SE)

Cette édition de Java reprend toutes les API des versions précédentes et contient tout le nécessaire au développement d'applications et d'applets.

Java 2 Enterprise Edition (J2EE)

Cette édition a besoin du J2SE pour fonctionner. Elle contient un ensemble d'APIs permettant de développer des applications destinées aux entreprises. Ainsi, elle permet la gestion de bases de données, de servlets et de pages HTML dynamiques.

Java 2 Micro Edition (J2ME)

Cette édition est destinée aux applications embarquées. Les APIs qui la composent ont été allégées pour définir un sous-ensemble minimum de fonctionnalités.

Plusieurs sous-ensembles sont définis :

- Connected Limited Device Configuration (CLDC) : Ce sous-ensemble a été défini pour des processeurs 16 ou 32 bits ayant les caractéristiques suivantes :
  - Une fréquence de fonctionnement de 16 Mhz ou supérieur.
  - Un minimum de 160 Ko de mémoire non-volatile contenant les bibliothèques CLDC et la machine virtuelle.
  - Un minimum de 192 Ko de mémoire pour la plateforme Java
  - Une faible consommation d'énergie.
  - Des connexions réseaux avec une faible bande-passante.

  La JVM est remplacée par la KVM (Kilobyte Virtual Machine). La première version ne permettait pas l'utilisation de nombres flottants ce qui n'est plus le cas aujourd'hui.
- Mobile Information Device Profile (MIDP) : Ce sous-ensemble est une extension de CLDC qui définit la façon dont les applications logicielles se connectent à l'interface des téléphones cellulaires. Cette extension contient également des API destinées à améliorer certains aspects tels que la lecture de sons ou de fichiers multimédias. MIDP contient aussi une bibliothèque destinée aux jeux et le support de protocoles de communications supplémentaires.
- Connected Device Configuration (CDC) : Ce sous-ensemble de J2ME est le plus proche du J2SE. Il en existe trois profiles :
  - Foundation Profile : API proche du J2SE mais sans interface graphique. Compatibilité avec la bibliothèque CLDC 1.0.
  - Personal Basis Profile : Extension au Foundation Profile avec une API graphique légère.
  - Personal Profile : Extension au Personal Basis Profile avec le support complet de la bibliothèque AWT et des applets.

  Le profil matériel requit est le suivant :
  - Processeur 32 bits.
  - 2 Mo de RAM.
  - 2,5 Mo de ROM.

Tous ces sous-ensembles possèdent des extensions optionnelles.

JavaCard

Le JavaCard est destiné aux cartes à puce et doit donc être exécuté sur un système très réduit, la taille de la JVM et du système ne devant pas dépasser 16Ko. Le JavaCard définit donc un sous-ensemble restreint de Java. De plus, la JVM est découpée en deux parties, une partie est dans la carte, l'autre est dans le lecteur. La configuration minimum pour faire tourner un programme JavaCard est un processeur 8 bits avec 24Ko de ROM, 1Ko de RAM et 16Ko EEPROM.

Beaucoup d'aspects de Java ne peuvent pas être utilisés. Ainsi, le chargement dynamique de classes est impossible, une nouvelle applet ne peut utiliser que des classes "connues". Il n'existe pas de Garbage Collector, la mémoire peut donc être mal gérée et ne plus être disponible pour de nouvelles classes ou objets. Seuls les types byte, short et boolean sont utilisables (les entiers sont optionnels).

2.3 - Les exécutions matérielles

Les exécutions matérielles de Java existantes découlent de deux approches différentes. L'une s'appuie sur des processeurs généralistes auxquels est ajouté un coprocesseur spécialisé Java. Ce coprocesseur est chargé de transformer le langage Java en séquence d'instructions du code natif du processeur généraliste ou d'exécuter directement le code Java. Les instructions complexes sont émulées par le processeur principal. Le code natif du processeur peut toujours être utilisé permettant de faire coexister des applications de types différents. La seconde approche est la réalisation d'un processeur exécutant directement le Bytecode Java. Cette solution nécessite que toutes les applications, ainsi que le système d'exploitation, soient écrits en Java.

2.3.a - Les unités de transformations d'instructions matérielles et les coprocesseurs

L'utilisation d'un coprocesseur spécialisé en coordination avec un processeur généraliste est la méthode la plus simple pour accélérer l'exécution du Bytecode Java de façon matériel. Les coprocesseurs remplacent généralement le switch de la boucle de l'interpréteur Java.

Hard-Int

Le Hard-Int a été étudié et développé par R. Radhakrishnan à l'université du Texas à Austin en 2000 à l'occasion de sa thèse. Le Hard-Int (Hardware Interpreter) consiste en un module matériel de transformation du Bytecode Java en suite d'instructions RISC pour un processeur généraliste [15].Ce module est placé entres les étages de fetch et de décodage des instructions (Figure 2-1 [15]) et peut être contourné afin d'exécuter le jeu d'instruction standard du processeur généraliste utilisé.

Les instructions Java sont traduites par le module Hard-Int, qui contient le microcode correspondant à la traduction du Bytecode Java. Les instructions correspondantes sont envoyées à l'étage de décodage du processeur. Les instructions complexes ne font pas partie du microcode mais demandent l'appel à des routines logicielles pour leurs exécutions. Les instructions traduites sont également envoyées dans un buffer d'instructions afin d'être utilisées plus rapidement lors de la réutilisation d'une séquence de programme (boucle).

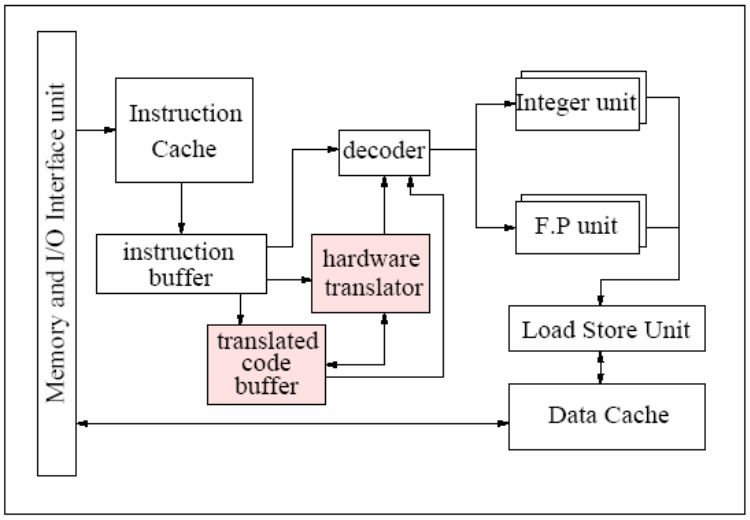


Figure 2-1 - Module Hard-Int.

Une simulation de cette architecture a été effectuée sur un processeur SPARC avec 4 unités d'exécutions, elle montre une augmentation des performances d'un facteur de 2,6 par rapport au JIT en Java 1.2 sur le même processeur. Aucune évaluation, ni du coût matériel, ni de l'impact sur le cycle d'horloge n'a été effectuée.

DELFT-JAVA

Le DELFT-JAVA est le produit de la thèse de J. Glossner à l'Université de technologie Delft, Hollande, en 2001. Le DELFT-JAVA [16] est basé sur un processeur RISC optimisé pour les applications multimédia en Java. Ce processeur possède une extension DSP et son jeu d'instructions intègre des instructions spécifiques à Java. Les instructions sont traduites dynamiquement à partir du Bytecode dans le jeu d'instructions du DELFT-JAVA. Malheureusement, aucune explication n'est donnée sur la façon dont cette traduction fonctionne. L'auteur de cette thèse soutient également que le coût de cette transformation n'est pas excessif mais ne donne pas d'information sur la taille d'un tel dispositif. Le sommet de la pile d'exécution est stocké dans une file de registres adressée indirectement. Trois autres files de 32 registres 32 bits sont utilisé pour stocker les informations liées à l'exécution. La transformation du Bytecode Java effectuée produit un grand nombre de dépendances de données. Ces dépendances sont minimisées en utilisant le renommage de registre ou l'exécution dans le désordre. Ce processeur intègre également un buffer de traduction de lien dynamique (Link Translation Buffer) qui permet d'accélérer les liens dynamiques lors de l'invocation de méthodes (lien déjà effectué lors d'un précédant appel).

Un modèle C++ du DELFT-JAVA a été développé et expérimenté à l'aide d'un programme de multiplication de vecteurs. Il utilise le renomage de registre et l'exécution dans le désordre sur plusieurs unités d'exécutions. Le DELFT-JAVA "idéal" est comparé à une exécution sur pile "idéale". Le gain obtenu est d'un facteur 2,7.

JIFFY

Cette thèse de G. Archer à l'université technique de Munich [17] présente un compilateur Just-in-Time réalisé dans un FPGA. L'auteur affirme qu'un compilateur JIT logiciel demande des ressources trop importantes pour la plupart des processeurs embarqués, il propose donc de faire cette opération matériellement dans un FPGA. Le processus se fait en plusieurs étapes. Le Bytecode est traduit dans un langage intermédiaire avec trois registres et une pile. Les trois registres correspondent aux trois opérandes accédés dans la pile d'exécution. Le code est alors optimisé (Figure 2-2 [17]) et les opérations sur la pile deviennent des opérations sur des registres. Enfin, le code ainsi obtenu est traduit dans le langage du processeur cible.

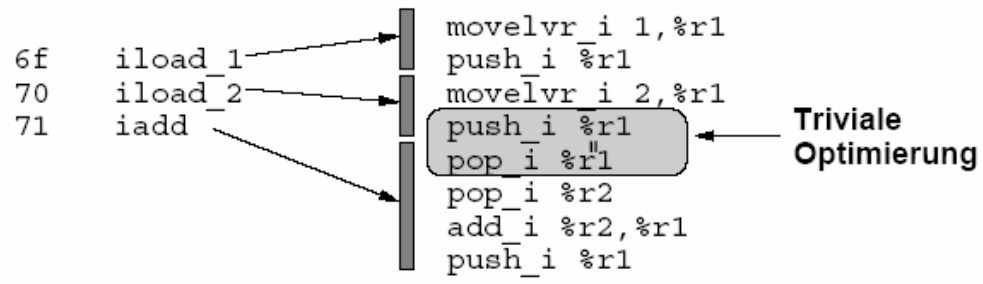


Figure 2-2 - Transformation dans le code intermédiaire JIFFY et optimisation de ce code.

Cette technique a été testée sur une version logicielle de JIFFY pour deux architectures, CISC (80586) et RISC (Alpha 21164). Les résultats obtenus sont de 1,1 à 7,5 fois plus rapides

qu'une interprétation sur ces mêmes machines. L'exécution est estimée entre 50 et 70 cycles par instruction Bytecode. Une implémentation sur un FPGA Xilinx XC2S200 utilise 3800 LC et 8 Kbits de RAM.

Jazelle

Jazelle [18] est une extension aux processeurs RISC 32-bit ARM. Ce coprocesseur est intégré sur la même puce que le processeur auquel il est associé. Il est disponible depuis 2001. Historiquement, les processeurs ARM sont capables d'exécuter deux jeux d'instructions, ARM et "thumb", "thumb", étant un jeu d'instructions compact (16-bit). Ces deux jeux d'instructions sont utilisés par les applications en fonction du besoin de performance et de compacité du code. Jazelle autorise le Bytecode Java comme troisième jeu d'instructions (Figure 2-3 [19]). Le décodeur Jazelle est implémenté en moins de 12K portes. 140 instructions du Bytecode sont exécutées directement en matériel, les 94 instructions restantes sont émulées par des séquences d'instructions ARM. Certaines instructions sont remplacées par leurs versions "quick" après résolution des liens.

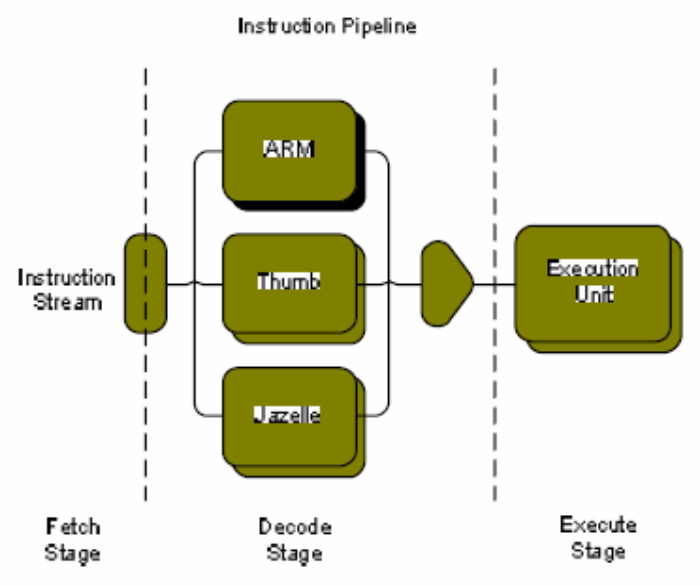


Figure 2-3 - Décodage du Java avec ARM Jazelle.

Une nouvelle instruction ARM permet de commuter le processeur en mode Java. Le chargement et le décodage des instructions en mode Java entraîne une pénalité d'un cycle supplémentaire par rapport au mode ARM.
En mode Java, quatre registres représentent les quatre entrées du sommet de la pile d'exécution. Le chargement et le rangement de leurs valeurs en mémoire sont effectués automatiquement par le matériel. D'autres registres contiennent l'environnement courant (VARS, CONST_POOL, OPTOP, PC) et la variable locale 0 (le pointeur *this*).
ARM propose également un optimiseur Bytecode vers Bytecode pour améliorer les performances de Jazelle. Le code ainsi obtenu reste utilisable pour les autres techniques d'exécution de Java [19]. Le processeur ARM926EJ-S incorporant Jazelle est implémenté sur un ASIC en 0,13µ, sa fréquence de fonctionnement est 266 Mhz. Le sous-ensemble de Java ciblé est le J2ME CLDC MIDP 2.0.

JSTAR, JA108

Le JSTAR à été développé par JEDI technologie devenu depuis Nazomi communications [20] [21]. Il est maintenant connu comme JA108. C'est un coprocesseur qui charge les instructions depuis la mémoire. S'il s'agit d'instructions du Bytecode, il effectue la transformation dans le jeu d'instructions du processeur auquel il est associé. Ce coprocesseur traducteur est destiné à être utilisé dans les téléphones portables afin d'optimiser les applications multimédias en Java. L'implémentation matérielle demande 30K portes et 45 Kbits de microcode pour un MISP R3000. Le JA108 possède également un cache instructions (Bytecode) associatif 2 voies de 4Ko et un cache de données associatif 2 voies de 2Ko. La première implémentation sur un FPGA avec un MIPS R3000 à 12Mhz permettait une augmentation des performances d'un facteur 5,5. Aujourd'hui la fréquence de fonctionnement est de 104Mhz. Il est implémenté sur un ASIC en 0 ,18µ. Le sous-ensemble de Java ciblé est le J2ME CLDC.

2.3.b - Les processeurs

Les processeurs Java sont destinés aux applications embarquées. Dans ce contexte, le système d'exploitation et les drivers doivent être écrits en Java.

PicoJava

Le PicoJava est un processeur 32-bit développé par Sun Microsystems. Mais qui n'a jamais été réalisé. En effet, ne souhaitant pas le produire, Sun avait décidé de vendre la licence à plusieurs sociétés (LG, NEC Corporation, Fujitsu Limited, Rockwell Collins Inc et IBM Corporation) qui ne produirent en définitive, ni le PicoJava, ni aucun SOC utilisant le PicoJava.

La première version du processeur [22] [23] à été développée en 1997. Le PicoJava-I est un processeur pipeline à 4 étages. La seconde version du processeur est apparue en 1999 (Figure 2-4 [24]).

Les instructions simples sont exécutées directement en matériel en un à trois cycles. Les instructions plus complexes (ex : changement de contexte) sont exécutées en microcode, tandis que les instructions très complexes (instructions demandant la résolution de constant pool) sont exécutées par du logiciel grâce à des routines du gestionnaire d'interruption (trap handler) écrites en Java. Pour accéder à la mémoire ou aux registres internes du processeur, 115 instructions supplémentaires sur deux octets (le premier octet étant 0xFF) ont été ajoutées. Ces instructions supplémentaires permettent d'écrire le système d'exploitation.

Le PicoJava-II comporte plusieurs mécanismes qui permettent d'augmenter la vitesse d'exécution des programmes Java.

Ainsi, le PicoJava-II possède un buffer d'instructions de 16 octets qui permet de découpler le chargement des instructions depuis le cache instruction (8 octets par accès au cache) et leur consommation par le processeur.

Le sommet de la pile Java est contenu dans une file circulaire de 64 registres 32 bits. Cette file possède deux accès en lecture et un accès en écriture permettant l'accès à toutes les données qu'elle contient. Cette file se remplie et se vide automatiquement, depuis et vers la mémoire, par un mécanisme appelé "dribbling". L'exécution de ce mécanisme est possible grâce à un accès supplémentaire en lecture et en écriture de la file.

Le buffer d'instruction et la file circulaire permettent le décodage multiple d'instructions et le groupement de celles-ci. En effet, le buffer d'instruction permet de rendre visible sept octets pour l'étage de décodage du pipeline. Il est alors possible de décoder jusqu'a quatre instructions de types différents. Un groupement d'instruction typique est :

iload_1 iload_2 iadd istore_3, ce qui permet d'effectuer ces instructions en un cycle au lieu de quatre. Le groupement d'instruction du PicoJava-II sera développé par la suite.

Le picoJava-II possède également une unité de calcul entier et une unité de calcul flottant. L'unité flottante peut être retirée pour utiliser moins de matériel, les instructions correspondantes sont alors émulées par le gestionnaire d'interruption.

Le cache instruction est un cache à correspondance directe tandis que le cache de données est un cache associatif deux voies, chacun des caches est constitué de lignes de 16 octets. Les deux caches sont configurables pour une taille allant de 0 à 16 Kilooctets.

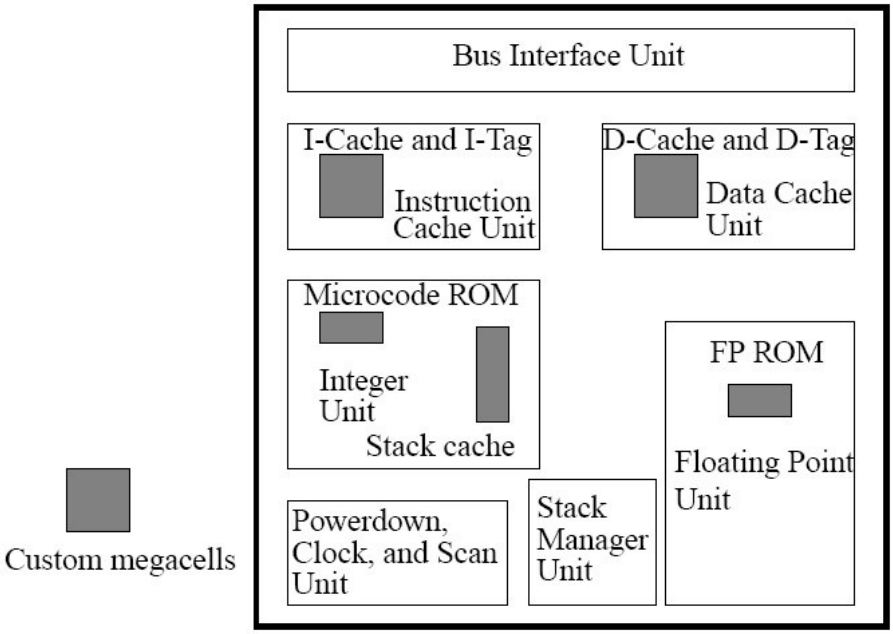


Figure 2-4 - Diagramme du coeur du PicoJava-II.

Le PicoJava-II est doté d'un pipeline dont les 6 étages [24] sont constitués par :

- Fetch : Charge 8 octets depuis le cache instruction ou 4 octets depuis le bus mémoire vers un buffer 16 octets.
- Decode : Décode et groupe jusqu'à 4 instructions d'une longueur totale de 7 octets.
- Register : Charge les opérandes dans le pipeline depuis la pile, détermine les conditions de bypass. Charge les constantes.
- Execution : Exécute les instructions arithmétiques, calcule les adresses d'accès à la mémoire, les adresses de saut des branchements.
- Cache : Accès à la mémoire.
- Write Back : Range les données dans la pile.

Le PicoJava-II est le processeur Java le plus complexe mais également le plus puissant. Une implémentation matérielle nécessite 440K portes (128K pour la logique et 314K pour les composants mémoires : une ROM 284x80 bits de microcodes, 2x192x64 bits pour la ROM de la FPU et 2x 16Ko pour les caches). La fréquence de fonctionnement annoncée pour le PicoJava-I était de 100Mhz. Le PicoJava-II n'est pas limité à un sous-ensemble de Java.

aJile JEMcore

Le JEMcore [25] [26] existe comme IP et SOC, il est disponible comme processeur et coprocesseur. Il est basé sur le processeur 32 bits JEM2 développé par Rockwell-Collins. Le JEM2 est une version améliorée du JEM1 créée en 1997, qui fut développé sur la base d'un processeur sur pile destiné aux applications embarquées. Rockwell-collins décida de ne pas vendre le JEM mais de donner une licence exclusive à la société aJile qui fut fondée en 1999 par des ingénieurs de Rockwell-Collins, de Centaur Technologies, de Sun Microsystems et de IDT.

Le JEM2 possède 24 registres 32 bits dont six sont utilisés pour stocker les éléments du sommet de la pile. Le chemin de données est constitué d'une unité arithmétique et logique (ALU) 32 bits, d'un décaleur 32 bits et d'une unité flottante (FPU). Une ROM contient le microcode effectuant les opérations du Bytecode Java. Une RAM permet d'ajouter des instructions spécifiques. Le JEM2 possède un mécanisme optionnel permettant d'exécuter deux programmes Bytecode indépendants en temps partagé sur le processeur appelé MJM (Multiple JVM Manager). Ce mécanisme simule deux processeurs JEM2 virtuels (hyper-threading). Ces deux processeurs virtuels peuvent être assimilés à deux JVMs indépendantes qui possèdent chacune leur propre mode d'exécution. Les interruptions restent en attente lors du changement de JVM active. Un système de protection mémoire permet d'allouer à chaque processeur virtuel une partie de la mémoire. Selon aJile le JEM2 peut être implémenté en 25K portes auxquelles il faut ajouter la ROM contenant le microcode. Le modèle aJ-100 possède 48Ko de RAM (mémoire microcode = 16Ko, mémoire dédiée aux données = 32Ko). Le MJM demande 10K portes supplémentaires.

Ce processeur est destiné aux applications embarquées. La fréquence de fonctionnement de l'aJ-100 est de 100Mhz. Il est implémenté sur un ASIC en 0,25µ. Le sous-ensemble de Java ciblé est le J2ME CLDC.

Cjip

Le Cjip, conçu par Imsys technologies [27] [28], est un processeur micro-programmé qui supporte plusieurs jeux d'instructions permettant d'exécuter de nombreux langages (C, C++ Java, assembleur). Il en résulte une représentation interne de son microcode sur 72 bits. Ce processeur est basé sur une architecture CISC. Il possède une ROM de 36Kbits et une RAM de 18Kbits pour pouvoir supporter des instructions additionnelles configurables. Trois piles indépendantes représentent les trois zones de la « pile java » (2.2.a -) [29]. Les entrées des sommets de la pile des variables locales et de la pile d'exécution sont stockées dans une mémoire rapide pour un accès plus optimisé. Ce processeur possède un FPU et un multiplieur 8x8. Les instructions du microcode s'exécutent en deux à trois cycles. Les instructions du Bytecode nécessitent plusieurs instructions du microcode. Ainsi un simple *nop* s'exécute en six cycles, tandis qu'une addition sur des entiers *iadd* demande douze cycles. Les instructions orientées-objet comme *putfield*, *getfield* ou *invovevirtua*l ne font pas partie du jeu d'instructions. Elles sont réalisées par pur logiciel. Le Cjip est implémenté sur un ASIC en 0,35µ. La fréquence de fonctionnement est de 80Mhz. Le sous-ensemble de Java ciblé est le J2ME CLDC.

Ignite, PSC1000

Le PSC1000 [30] est un processeur 32 bits basé sur le processeur Shboom, destiné à l'exécution du langage Forth. La compagnie Scientific Patriot décide d'en faire un processeur Java et de le renommer Ignite. Le jeu d'instructions de l'Ignite n'est pas réellement le Bytecode Java mais le ROSC (Removed Operand Set Computer). Les instructions sont

représentées sur un octet. Un prétraitement logiciel permet de transformer le Bytecode Java en ROSC.

Ce processeur contient deux piles, une de 16 registres pour la partie Framework et une de 18 registres pour la pile d'exécution, ainsi que 16 registres globaux [31]. La pile d'exécution est automatiquement remplie et vidée depuis et vers la mémoire. La fréquence de fonctionnement du Ignite est de 80Mhz en tant qu'ASIC en 0,13µ. Il est également disponible en FPGA sur Xilinx et Altera (9700 LC).

Moon 2

Le Moon2 [32] [33] est un processeur 32 bits développé par VULCAN machines. Les instructions simples sont directement exécutées, les plus complexes sont micro-programmées voire nécessitent de faire appel à un gestionnaire d'interruption (trap handler) pour les émuler avec des instructions simples. Un buffer d'instructions ainsi qu'un mécanisme simple de groupement d'instructions sont utilisés pour réduire le nombre de cycle. Ce processeur utilise une RAM à un port pour implémenter une pile de 256 entrées 32bits. La première version du Moon utilisait 3840 LC d'un FPGA Altera. Ce processeur est actuellement disponible sous forme de HDL pour FPGA Altera (3660 LC), ainsi qu'en VHLD et Verilog. Une implémentation minimum demande 27K portes + 3K ROM + 1K RAM simple port. La fréquence de fonctionnement du Moon 2 est de 100Mhz. Le sous-ensemble de Java ciblé est le J2ME CLDC.

Ligthfoot

Le Ligthfoot [34] est un processeur hybride 8/32 bits développé par Digital Communication Technologies sur les bases d'une architecture Harvard à pile. L'accès à la mémoire des programmes est sur 8 bits tandis que la mémoire des données est sur 32 bits, les adresses sont quand à elles sur 24 bits. Ce processeur pipeline à trois étages possède une ALU 32 bits, un décaleur 32 bits ainsi qu'un multiplieur par pas de 2 bits. Deux piles matérielles sont étendues en mémoire. La pile de données est utilisée pour stoker uniquement la pile d'exécution. Elle est constituée, dans sa partie matérielle, de 8 registres 32 bits. La pile de retour contient les adresses de retour de sous-programme, son sommet est utilisé comme registre d'index pour accéder à la mémoire des programmes, sa partie matérielle est constituée de 4 registres 32 bits. Un banc de registres contient les quatre premières variables locales de la méthode courante. Trois formats d'instructions sont spécifiés : "soft Bytecode" IF0 (128 instructions); les instructions sans retour IF1 (64 instructions); les instructions simples sur un octet qui peuvent être groupées avec une instruction de retour IF2 (32 instructions). Les 128 instructions IF0 ne sont pas exécutées directement mais émulées par des sous programmes. Le chargement de ces sous programmes nécessite un cycle supplémentaire. L'adresse de la prochaine instruction est placée au sommet de la pile de retour. Une implémentation demande 30K portes sous sa forme VHDL. Le processeur est disponible également sous forme d'IP comme netlist pour un FPGA Xilinx et occupe 1710 CLB (=3400 LC) et deux blocks de RAM. Sa fréquence de fonctionnement dans un Vertex-II est de 40Mhz. Les sous-ensembles de Java ciblés sont les J2ME CLDC et CDC.

LavaCORE

Le LavaCORE [35] est un processeur 32 bits développé par Derivation Systems Inc. Il possède une ALU 32 bits et une RAM double accès 32x32 implémentant le sommet de la pile Java. Le registre instruction est composé de trois registres 8-bits. Une unité de pré-chargement des instructions permet de réduire le nombre d'accès mémoire. Les instructions simples sont implémentées en matériel, tandis que les plus complexes sont effectuées par des

routines logicielles. Les instructions flottantes ne font pas partie du jeu d'instruction sur la version standard du processeur, ainsi, pour pouvoir les utiliser, un module optionnel doit être ajouté. Un outil logiciel permet de profiler les programmes destinés à des applications embarquées spécialisées afin de n'implémenter que les instructions réellement utilisées. La version standard utilise 1926 CLB (= 3800 LC) dans un FPGA Altera Virtex-II pour une fréquence de fonctionnement de 20Mhz.

Komodo

Ce processeur [36] [37] est le résultat de la thèse de R. Zulauf à l'université de Karlsruhe en 2000. Il est aujourd'hui utilisé comme base de recherche pour des systèmes temps réel multi-thread. Le Komodo est un processeur pipeline à 4 étages : Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch (OF) et execute / memory access / IO acces (EXE). L'originalité de ce processeur réside dans le fait que le processeur Komodo permet, grâce à ces quatre compteurs ordinaux et ces quatre piles, de supporter quatre thread simultanés (Figure 2-5 [38]). Un mécanisme gère les priorités des threads et permet de changer de thread actif sans cycle de gel. Les instructions simples sont directement exécutées, tandis que celles plus complexes sont exécutées par des séquences de microcode allant jusqu'à 50 opcodes. Les instructions trop complexes pour être exécutées en microcode sont effectuées en logiciel grâce à des routines du gestionnaire d'interruption (trap handler).

Le Komodo est doté d'un chargeur de classe dynamique, permettant également la mise à jour de classes déjà présentes en mémoire, et d'un mécanisme de Garbage Collector. Les auteurs déclarent que ces mécanismes ne sont pas antagonistes avec son utilisation pour le temps réel.

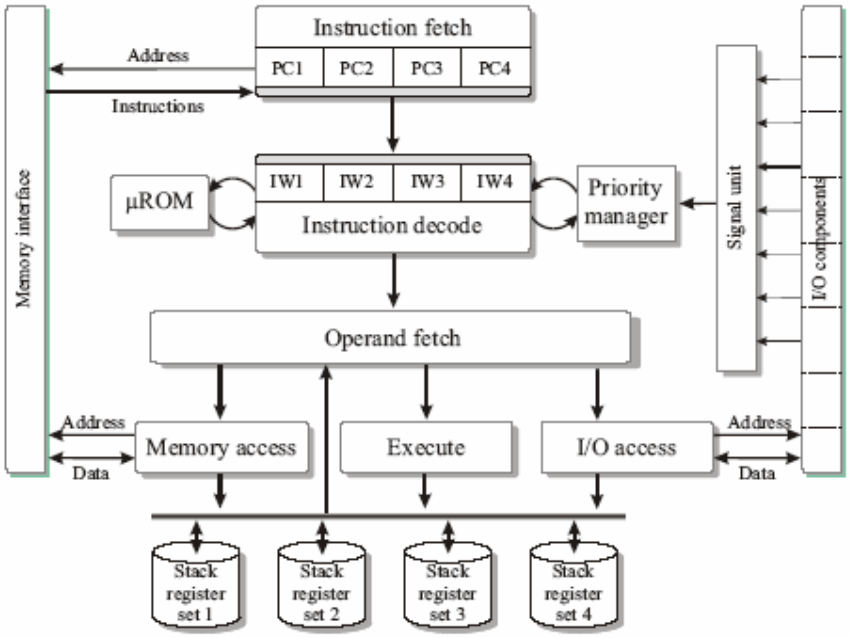


Figure 2-5 - Architecture du Komodo.

La première version du Komodo était implémentée dans un FPGA cadencé à 20Mhz et supportait un faible nombre de Bytecode (50 Bytecodes). Cette implémentation nécessitait 1300 CLB (=2600LC) sur un Xilinx XC 4036 XL.

FemtoJava

Le FemtoJava [39] est un projet de recherche sur un processeur Java destiné à des applications spécifiques. Les applications destinées à être exécutées sur le FemtoJava sont profilées afin de n'implémenter que les instructions du Bytecode nécessaires. Seulement 69 instructions du Bytecode sont implémentables dans le FemtoJava. Les auteurs montrent à l'aide de petites applications (de 50 à 280 Bytecode) que seulement 22 à 69 instructions distinctes sont utilisées. Pour respecter cette contrainte, un certain nombre de règles sont à respecter : la création d'objet est interdite, les méthodes doivent être statiques et sans récursivités, les interfaces sont interdites, pas de données flottantes et pas de programmation multithread. Le chemin de données peut être 8 bits ou 16 bits ce qui n'est pas conforme à la spécification de Java. Chaque instruction prend plusieurs cycles pour s'exécuter. Une implémentation matérielle varie entre 1000 et 2000 LC sur un FPGA pour des fréquences de fonctionnement allant de 3 Mhz à 8 Mhz.

JOP

Le processeur JOP est le résultat de la thèse de M. Schöberl à l'université de technologie de Vienne achevée en janvier 2005 [40]. Ce processeur est destiné aux applications embarquées temps-réel. Son pipeline de 4 étages est le suivant : Bytecode Fetch, microcode Fetch, Decode et Execute. Les instructions Java ne sont pas exécutées directement mais remplacées par une instruction ou une séquence d'instruction du microcode JOP (Figure 2-6 [40]).

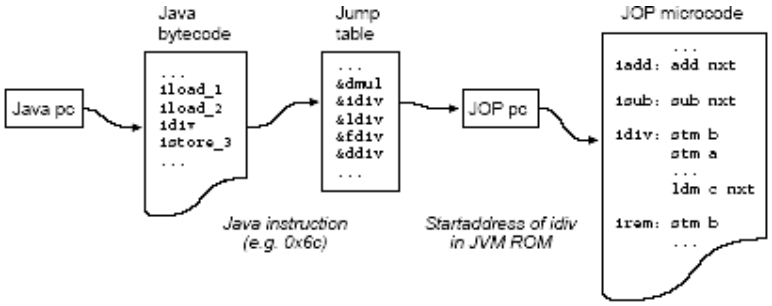


Figure 2-6 - Traduction en microcode JOP.

Une implémentation nécessite 1830 LC et 3Ko de RAM sur un FPGA cadencé à 100Mhz. Son auteur estime l'équivalence en porte à 11K portes et 40K portes pour la mémoire. Le sous-ensemble de Java ciblé est le J2ME CLDC.

JAP

Le JAP est un processeur 32 bits développé par Advanced Elesctronique Design (AED), sous architecture CISC Harvard dotée d'un pipe-line à 4 étages. Ce processeur, capable d'exécuter les langages C et Java, possède une pile matérielle de 32 registres 32-bits pour minimiser le nombre de transferts avec la mémoire ainsi que les échanges de données entre le mode natif et le mode Java. Une FPU est disponible optionnellement. Une grande partie des instructions est exécutée en un cycle. Les instructions complexes sont exécutées par du microcode ou par des routines du trap-handler écrites dans le jeu d'instructions natif du processeur JAP. Le prototype réalisé en 2002 était implémenté sur un ASIC en 0,35µ avec une fréquence de 80 Mhz. Aujourd'hui, une implémentation matérielle demande 25K portes pour une fréquence de fonctionnement de 100Mhz. Le système d'exploitation, appelé JOS, gère les interruptions, l'ordonnancement des threads, la mémoire et le Garbage Collector. Il représente 45Ko. Le sous-ensemble de Java ciblé est le J2ME CLDC MIDP.

2.3.c - Récapitulatif

Les caractéristiques des matériels précédents sont récapitulées dans le Tableau 2-1.

nom	type	technologie	taille	fréquence (Mhz)	sous ensemble Java
Hard-int	traducteur	simulation			
DELFT	traducteur	simulation			
JIFFY	traducteur	FPGA Xilinx	3800 LC 1Ko RAM		
Jazelle	coprocesseur	ASIC 0,13µ	12K portes	266	J2ME CDLC MIDP 2.0
JA108	coprocesseur	ASIC 0,18µ	30K portes 7Ko (mémoire)	104	J2ME CDLC
PicoJava-II	processeur		128K portes 314K portes (mémoire)		tous
JEM2	processeur	ASIC 0,25µ	25K portes 48Ko RAM ROM	100	J2ME CLDC
Cjip	processeur	ASIC 0,35µ	70K portes RAM ROM	80	J2ME CLDC
Ignite	processeur	ASIC 0,13µ FPGA Altera Xilinx	- 9700 LC	80	
Moon 2	processeur	ASIC 0,18µ FPGA Altera Xilinx	27K portes ou 3600 LC + 4Ko (mémoire)	100	J2ME CLDC
Lightfoot	processeur	FPGA Xilinx	3400 LC	40	J2ME CLDC CDC
LavaCORE	processeur	FPGA Altera	3800 LC	20	
Komodo	processeur	FPGA Xilinx	2600 LC	20	50 bytecodes
FemtoJava	processeur	FPGA Altera	2000 LC	8	69 Bytecodes
JOP	processeur	FPGA	1830 LC 3Ko RAM	100	J2ME CLDC
JAP	processeur	ASIC 0,35µ	25K portes 3Ko	100	J2ME CLDC MIDP

Tableau 2-1 - Caractéristiques des processeurs Java.

Les mécanismes d'accélérations de l'exécution du Bytecode Java peuvent être subdivisés en plusieurs catégories comme le montre la Figure 2-7.

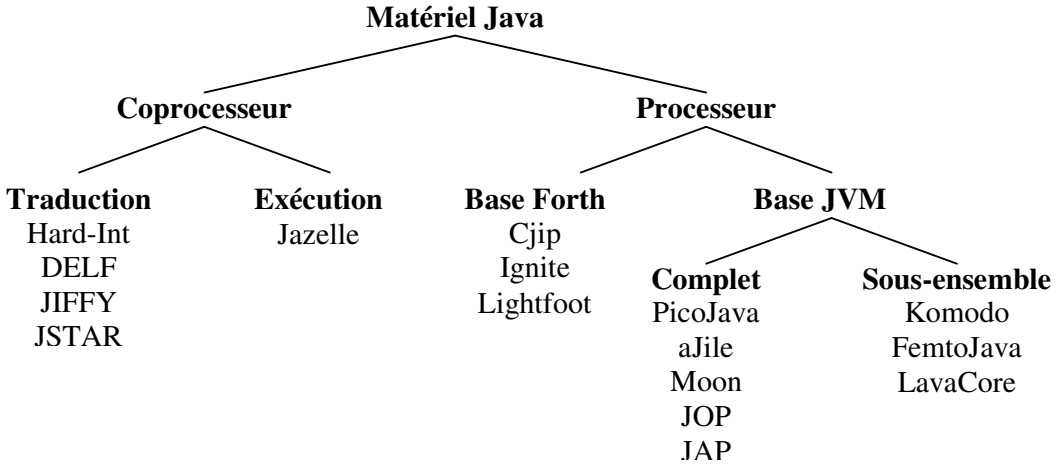


Figure 2-7 - Matériel d'accélération de l'exécution de Java.

L'emploi de coprocesseurs permet une intégration "facile" dans un système d'exploitation existant mais ne permet qu'une légère amélioration des performances de l'exécution du Bytecode Java par les processeurs généralistes et ceci malgré les optimisations faite sur le code généré.

Les processeurs basés sur des architectures Forth permettent une gestion séparée de la pile de contexte (FRAME) mais n'autorisent qu'un seul accès au sommet de la pile d'exécution et demandent donc plusieurs cycles d'exécution par instruction.

Les processeurs basés sur la JVM sont capables d'exécuter un grand nombre d'instructions du Bytecode en un cycle et permettent donc de réduire le nombre de cycles nécessaires à l'exécution des programmes Java. Ces processeurs utilisent différentes techniques d'accélération du Bytecode.

Stockage du sommet de la pile Java en matériel

Le fait de stocker le sommet de la pile Java dans un organe matériel permet de réduire le nombre d'accès au cache de données. La plupart des processeurs utilisent une pile matérielle pour stocker le sommet de la pile Java. Ces piles se vident et se remplissent dans et depuis la mémoire, grâce à des mécanismes automatique (sauf Moon 2 : pile 256 entrées fixes). La taille de cette pile détermine le taux d'échange avec la mémoire (plus la pile est grande, plus le taux d'échange diminue).

Buffer d'instructions

L'utilisation d'un buffer d'instructions permet de minimiser le nombre d'accès au cache instructions. En effet, les instructions Java sont de taille variable (1 à 5 octets) et ne sont pas alignées sur 32-bits en mémoire. Un buffer d'instructions permet de n'effectuer que des accès alignés (32 ou 64 bits) au cache d'instructions.

Groupement d'instructions

Le groupement d'instructions permet de regrouper des instructions utilisant des parties différentes du matériel afin de les exécuter en parallèle. Cette technique sera développée dans le prochain chapitre (3.2.c -).

Le processeur PicoJava-II utilise tous ces mécanismes, La plupart des autres processeurs spécialisés en sont inspiré. Il est le processeur Java existant le plus gros et également le plus performant. De plus, la description de ce processeur est disponible sous forme Verilog, nous avons donc naturellement décidé de l'utiliser comme base de comparaison pour le processeur à exécution sur file dont nous développerons l'architecture dans le prochain chapitre.

2.4 - Objectif

Nous cherchons à accélérer l'exécution des programmes Java, pour se faire nous nous intéressons à deux aspects de l'exécution matérielle de Java.

2.4.a - La gestion de l'environnement d'exécution

La gestion de l'environnement apparaît comme l'un des aspects pouvant être amélioré dans l'exécution des programmes Java par le processeur Picojava-II. En effet, les instructions liées aux changements de contextes (appels et retours de méthodes) représentent 11% des instructions des programmes Java [41]. Les branchements inconditionnels représentent 1% des instructions exécutées. Ces instructions ne peuvent pas être regroupées dans le Picojava-II. Celui-ci regroupe en moyenne entre 1,4 et 1,7 instructions par groupe [15], ce qui donne une instruction liée à l'environnement tous les 5,5 groupes. De plus, les instructions d'appels et de retours de méthode sont des instructions complexes qui nécessitent plusieurs cycles pour être exécutées [56].

Pour accélérer l'exécution de la gestion de l'environnement nous séparerons les piles d'exécution et d'environnement Java. L'utilisation de deux mécanismes distincts nous permettera d'exécuter les instructions liées à la gestion de l'environnement en parallèle avec les autres types d'instructions. Ces instructions pourront alors être regroupées avec les autres types d'instructions.

2.4.b - L'exécution d'un code post-fixé

Le Bytecode Java est un langage post-fixé, son exécution « classique » est effectuée sur une pile. Nous remplacerons cette pile par l'utilisation de la technique d'exécution sur file.

L'exécution sur PILE

Le principe de l'exécution d'un langage post fixé sur pile est très connu. Les opérandes sont empilés au sommet de la pile. A l'occurrence d'une instruction de calcul, les opérandes sont sortis de la pile par son sommet, l'opération est effectuée, puis le résultat est empilé sur le nouveau sommet (Figure 2-8).

Ainsi l'expression "(16 – 11) + 12 * 2" dont la représentation post-fixée est 16 11 – 12 2 * + est traduite en Bytecode par la séquence d'instruction bipush 16, bipush 11, isub, bipush 12, iconst_2, imul, iadd. Cette séquence est exécutée sur une pile de la façon suivante :

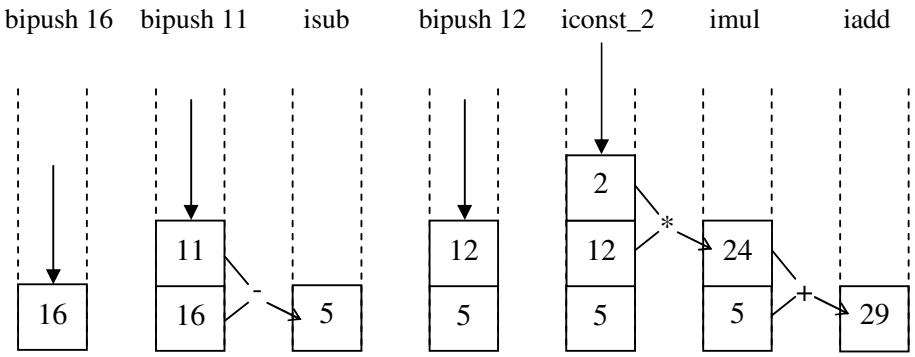


Figure 2-8 - Exécution sur pile.

L'utilisation de cette technique force l'exécution séquentielle des instructions et interdit tout parallélisme entre ces instructions. L'exécution d'un langage post-fixé à l'aide d'une pile impose donc le traitement séquentiel des instructions. Dans une pile, l'entrée des opérandes suivants ne peut être réalisée qu'après que l'opération précédente ait été réalisée. Il existe donc un goulot d'étranglement au sommet de la pile.

L'exécution sur FILE

L'utilisation d'une file pour exécuter le même langage permet de briser cette contrainte en permettant le chargement anticipé des opérandes. Le processeur devient alors « pipe-line ».

Cette technique a été développé en 1974 dans le groupe d'architecture des ordinateurs du laboratoire de l'IMAG à Grenoble [42] [43] sur une idée de François Anceau et mise en forme par Jean-Pierre Schoellkopf [44]. Elle donna lieu à plusieurs thèses et fut utilisée pour un prototype de machine Pascal appelé PASC-HLL.

Le code post-fixé peut être exécuté sur une file en utilisant des pointeurs d'extraction [42] [43] (Figure 2-9). Les opérandes sont chargés dans la file d'un coté et sont extraits de l'autre pour effectuer les instructions associées. Un pointeur P1 indique le premier opérande (sommet de la pile d'exécution virtuelle), un second P2 indique la position du deuxième opérande (qui est effacée après l'opération). Le résultat est replacé dans la file via P1, effaçant le fond de la file. Avant chaque opération d'exécution, le pointeur P1 est déplacé vers le nouveau sommet de la pile virtuelle. Le déplacement de P1 correspond au nombre d'opérandes chargés dans la file depuis la dernière opération. P2 est déplacé sur le premier opérande valide suivant P1. Il ne s'agit pas d'une simulation d'une pile dans une file car il n'est pas possible de réaliser des fonctions uniques "push" et "pull" sur la file. Dans l'exécution sur file le résultat est rangé à la place du premier opérande et non à la place du second, laissant ainsi des espaces libres. Ces espaces doivent être réduits par un mécanisme déplaçant les données vers le sommet de la file afin de ne pas laisser les données stagner, ce qui provoquerait une "fausse" occupation de celle-ci et empêcherait l'insertion de nouvelles données.

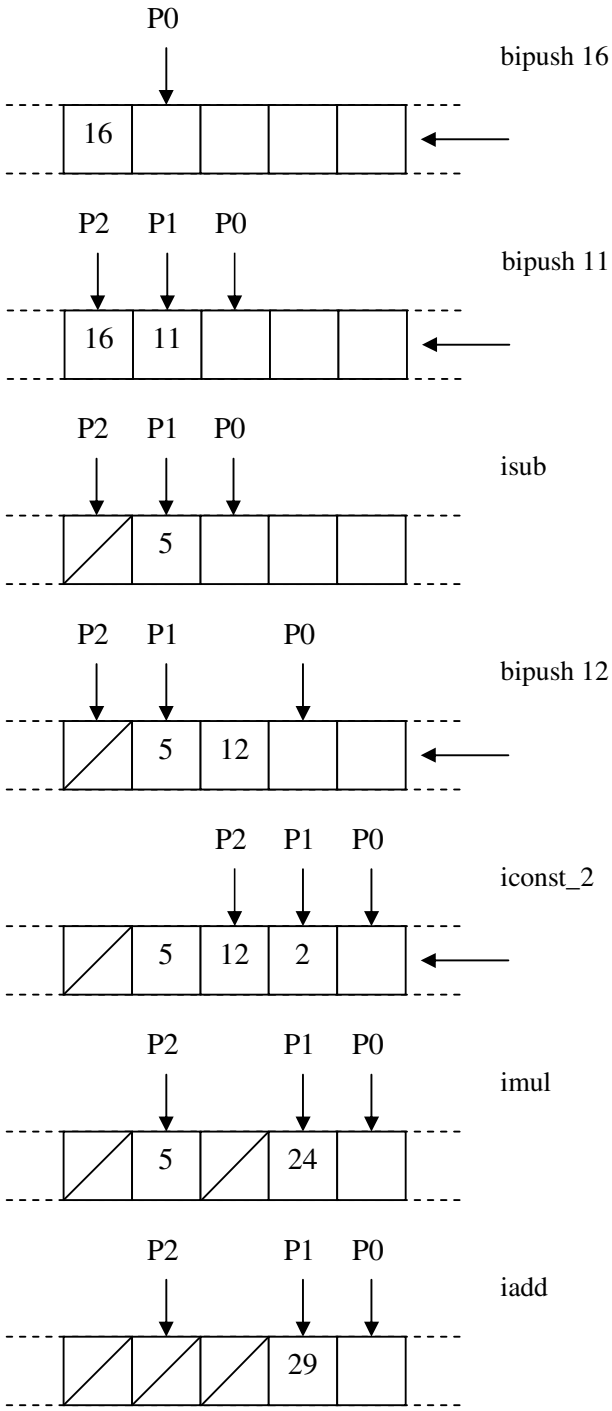


Figure 2-9 - Exécution sur FILE.

Contrairement à la pile, la file peut être « alimentée » par une extrémité tandis que l'exécution des opérations s'effectue sur l'autre. Les pointeurs (P1 et P2) indiquent les deux cases opérandes, et un troisième (P0) indique la case où s'effectue le chargement de la prochaine donnée. Il devient donc possible d'effectuer simultanément des opérations de chargement et de calcul. Cette technique permet d'effectuer en parallèle les opérations de chargement des opérandes et d'exécution des instructions. Cette File d'exécution ne concerne

que la partie exécution du Bytecode. Les zones de variables locales et de sauvegarde de l'environnement Java (FRAME) doivent toujours être contenues dans une pile.

2.5 - Conclusion

Dans ce chapitre, nous avons décrit le jeu d'instructions, les zones mémoires et les déclinaisons du langage Java.

Nous avons étudié différents processeurs utilisant des techniques diverses d'accélération matérielle. Ces processeurs se divisent en deux catégories.

La première est basée sur des processeurs généralistes auxquels on ajoute une unité de traduction du Bytecode dans le langage du processeur cible. Dans cette technique, chaque instruction du Bytecode est remplacée par une suite d'instructions du processeur cible et nécessite plusieurs cycles pour être exécutée, ceci malgré les optimisations effectuées sur le code généré par certains des mécanismes de traduction.

La seconde catégorie regroupe les processeurs spécialisés exécutant le Bytecode Java directement. Une partie de ces processeurs n'autorise qu'un seul accès à la pile d'exécution et nécessitent donc plusieurs cycles pour exécuter chaque instruction du Bytecode Java. Les processeurs plus performants sont capables d'exécuter une grande partie des instructions en un cycle. Ces processeurs utilisent plusieurs techniques d'accélération de l'exécution que nous avons décrites dans ce chapitre. Nous cherchons à développer un processeur de hautes performances, nous nous tournerons donc vers la conception d'un processeur spécialisé.

Nous avons choisi d'utiliser le processeur Picojava-II de Sun Microsystems comme base de comparaison car ce processeur est l'un des plus performants et nous disposons de sa description Verilog.

Nous avons ensuite exposé notre objectif qui consiste à accélérer l'exécution des programmes Java. Pour ce faire, nous cherchons tout d'abord à optimiser l'exécution des instructions liées à la gestion de l'environnement en utilisant deux mécanismes différents pour stocker la pile d'exécution et le reste de la pile Java. Cette séparation devrait permettre l'exécution en parallèle des instructions de gestion de l'environnement et des autres instructions. De plus, certains mécanismes seront mis en place pour réduire le nombre de cycles nécessaires à l'exécution de ces instructions. Nous chercherons également à augmenter le parallélisme d'exécution des autres instructions en utilisant la technique d'exécution sur file que nous avons décrite dans ce chapitre.

Chapitre 3 Architecture Générale

3.1 - Introduction

Pour évaluer les performances de l'exécution sur file appliquée au langage Java, un modèle de processeur Java à exécution sur pile a été développé afin de servir de modèle de comparaison. Ce modèle est inspiré du processeur PicoJava-II de Sun Microsystems. Il intègre les mêmes techniques d'accélération de l'exécution des programmes Java.

Dans ce chapitre, nous décrirons tout d'abord l'architecture des modèles de comparaison à exécution sur pile nommés JMS (Java Machine on Stack) et JMS_PRED. Le modèle JMS_PRED se différencie du modèle JMS par l'addition d'un prédicteur de branchement. Puis, nous verrons les principes de fonctionnement et l'architecture générale de la machine Java à exécution sur file appelée JMQ (Java Machine on Queue). Enfin nous examinerons le déroulement sur les deux architectures d'un petit exemple de programme Java.

3.2 - Le modèle JMS (Java Machine on Stack)

Le modèle JMS est un processeur Java à exécution sur pile qui hérite des techniques d'accélération d'exécution du Bytecode présentes dans le processeur PicoJava-II de Sun Microsystems. Ces techniques d'accélérations sont les suivantes :

- Utilisation d'un buffer de 16 octets de pré-chargement des instructions.
- L'utilisation d'une pile matérielle de 64 registres de 32-bits permettant de stocker le sommet de la "pile Java".
- Le groupement d'instructions qui permet le décodage de plusieurs instructions simultanément et le groupement de celles-ci.
- L'utilisation d'un pipeline constitué par la duplication d'éléments du sommet de la pile.

3.2.a - Le pipeline de la JMS

La machine JMS est dotée d'un pipeline hérité du PicoJava-II dont les 6 étages (Figure 3-1) sont les suivants :

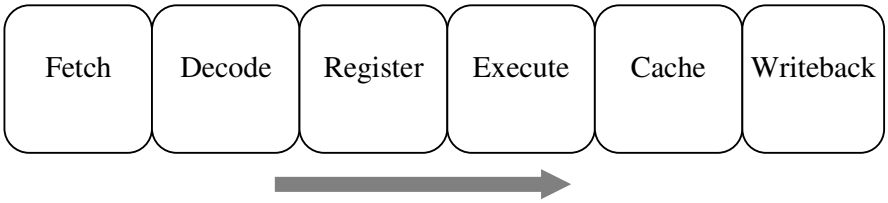


Figure 3-1 - Le pipeline des modèles JMS.

- Fetch : Charge 8 octets depuis le cache instruction vers un buffer de 16 octets.
- Decode : Décode et groupe jusqu'à 4 instructions d'une longueur maximale de 7 octets.
- Register : Charge les opérandes dans le pipeline depuis la pile, détermine les conditions de by-pass. Charge les constantes.
- Execution : Exécute les instructions arithmétiques, calcule les adresses d'accès à la mémoire ainsi que les adresses de saut des branchements.
- Cache : Accès à la mémoire.
- Write Back : Range les données dans la pile.

3.2.b - La pile de la JMS

Dans cette architecture, la pile d'exécution et la pile d'environnement sont regroupées dans une seule et même pile de mots de 32-bits (Figure 3-2). La partie supérieure de cette pile est stockée dans une file circulaire constituée de 64 registres 32-bits. Cette file possède deux accès en lecture et un accès en écriture qui permettent chacun l'accès à toutes les données qu'elle contient. Elle peut se vider dans le cache de données et peut se remplir à partir des données contenues dans le cache de données grâce à un mécanisme appelé "dribbling". Celui-ci est effectué par un accès supplémentaire en lecture et en écriture. La partie matérielle de la pile permet de réduire les accès au cache de données, les contenus du cache de données et de la pile matérielle ne sont donc pas cohérents.

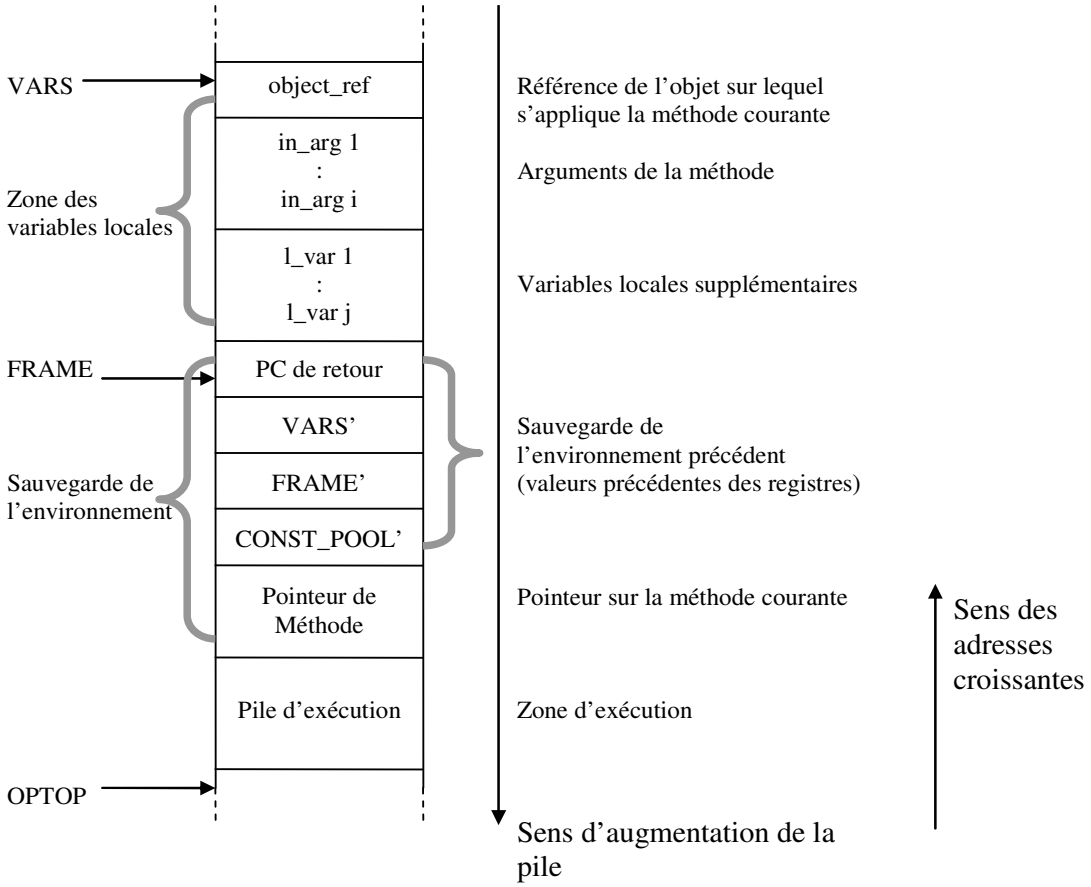


Figure 3-2 - La pile des modèles JMS.

Les registres de l'environnement sont les suivants :

- PC : pointe sur l'instruction courante.
- VARS : pointe la zone de variables locales de la méthode courante.
- FRAME : pointe la zone de sauvegarde de l'environnement précédent.
- CONST_POOL : pointe sur le constant pool de la classe de la méthode courante.
- OPTOP : pointe le sommet de la pile.

L'appel de méthode

Lors de l'appel d'une méthode, le processeur sauvegarde l'environnement courant et met en place l'environnement de la méthode appelée.

La première étape consiste à chercher l'adresse de la zone mémoire décrivant la méthode appelée. Cette zone contient toutes les informations concernant la méthode :

- Adresse du début de la méthode.
- Le nombre d'arguments.
- Le nombre de variables locales
- Adresse du constant pool.
- Adresse de la zone mémoire décrivant la classe à laquelle la méthode appartient (descripteur de classe).

Le descripteur de méthode est obtenu de façon différente selon le type de la méthode à appeler :

- Pour une méthode virtuelle, le choix de la méthode spécifique à appeler est résolu à l'exécution, le descripteur de méthode est donc obtenu à partir de la référence de l'objet sur lequel cette méthode s'applique. Le nombre d'arguments de la méthode est un champ de l'instruction d'appel de méthode virtuelle.
- Pour l'invocation d'une méthode d'une superclasse, le descripteur de méthode de la méthode à appeler est obtenu à partir du descripteur de méthode courant, celui-ci permet d'avoir accès à la zone mémoire qui décrit la classe (A.b).
- Pour les méthodes statiques (méthodes de classes) ou les méthodes non-virtuelles, le descripteur de méthode est obtenu à partir du constant pool de la méthode courante.

Une fois le descripteur de méthode obtenu, il est utilisé pour obtenir les informations relatives à la méthode appelée. Celles-ci permettent de mettre en place le nouvel environnement. Ainsi le nombre d'arguments permet d'obtenir la nouvelle zone de variables locales :

$$newVARS <= OPTOP + nbArgs$$

Le nombre de variables locales permet de calculer l'adresse de la nouvelle zone de sauvegarde de l'environnement et le nouveau sommet de la pile :

$$newFRAME <= OPTOP - nbVars$$

$$OPTOP <= newFRAME - 20$$

Les registres de l'environnement courant (PC de retour, VARS, FRAME, CONST_POOL) sont alors sauvegardés dans la pile à partir de l'adresse pointée par le registre newFRAME. Il en est de même pour le descripteur de méthode (un registre sauvegardé dans la pile par cycle) (Figure 3-3).

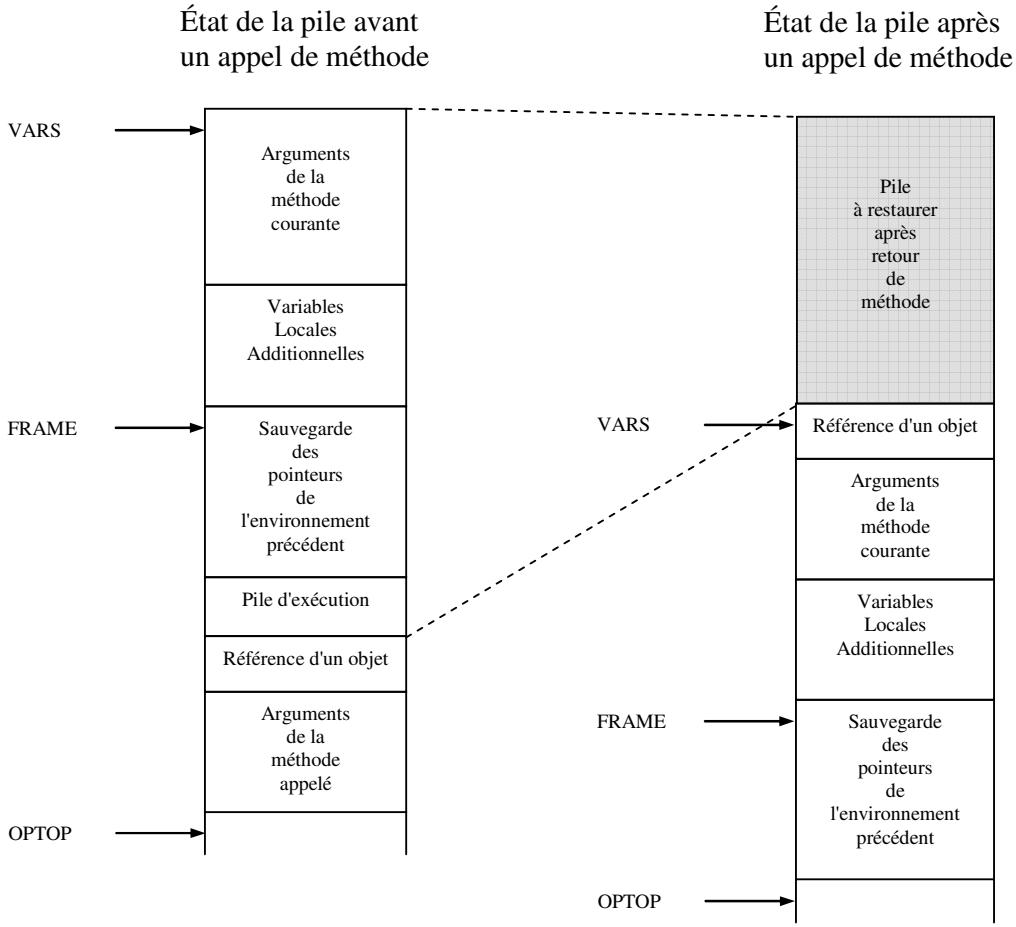


Figure 3-3 - Appel d'une méthode pour le modèle JMS.

Chacun des registres est alors mis à jour pendant le cycle qui suit la sauvegarde de son ancienne valeur dans la pile.

Le processeur continue l'exécution à partir de l'adresse de la méthode appelée.

Pour les méthodes d'instance (méthodes agissantes sur un objet), une exception est générée si la référence de l'objet est nulle.

Le retour de méthode

Lors d'un retour de méthode, la première opération à effectuer est de sauvegarder, si elle existe, la donnée à retourner.

L'environnement précédent est restauré en utilisant les sauvegardes des registres présentes dans la pile (un registre restauré par cycle).

La nouvelle valeur du registre OPTOP est obtenue à partir du registre VARS de la méthode courante et du type de la donnée à retourner (présence et taille de la donnée).

OPTOP <= VARS – taille de la donnée à retourner

La donnée de retour est alors recopiée au sommet de la pile (Figure 3-4).

L'exécution reprend alors depuis l'adresse pointée par le PC de retour.

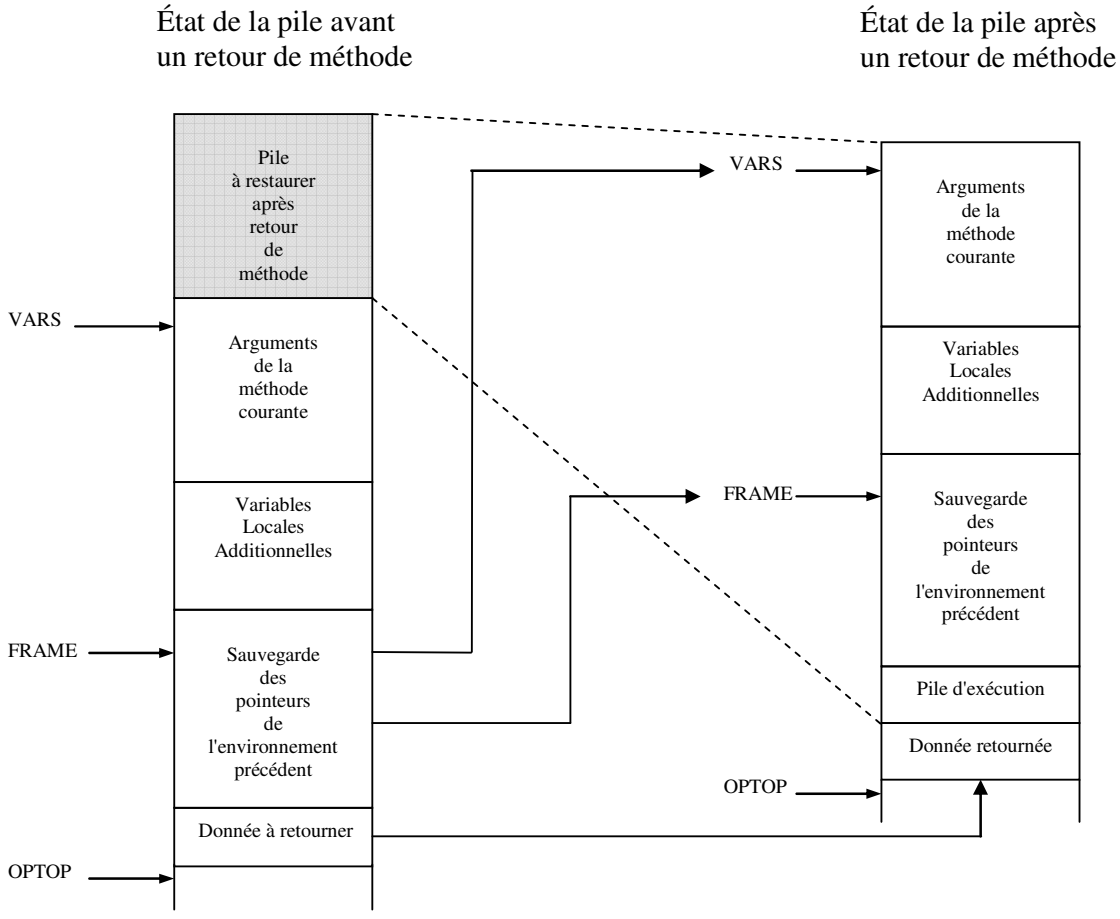


Figure 3-4 - Retour de méthode pour le modèle JMS.

3.2.c - Les regroupements d'instructions des modèles JMS

Le regroupement d'instructions consiste à décoder et exécuter simultanément plusieurs instructions qui utilisent des parties différentes du matériel. Ces instructions sont exécutées simultanément dans le pipeline comme s'il s'agissait d'une seule macro-instruction. Pour le Bytecode Java, ce mécanisme fut implémenté sur les processeurs PicoJava-I et II [45].

Ainsi, l'expression "z = x + y" dont la suite d'instructions du Bytecode Java équivalentes est "iload_1, iload_2, iadd, istore_3" est exécutée en quatre cycles sans groupement d'instructions (Figure 3-5 [46]). L'utilisation du groupement d'instructions permet d'exécuter cette même suite d'instructions en un seul cycle (Figure 3-6 [46]).

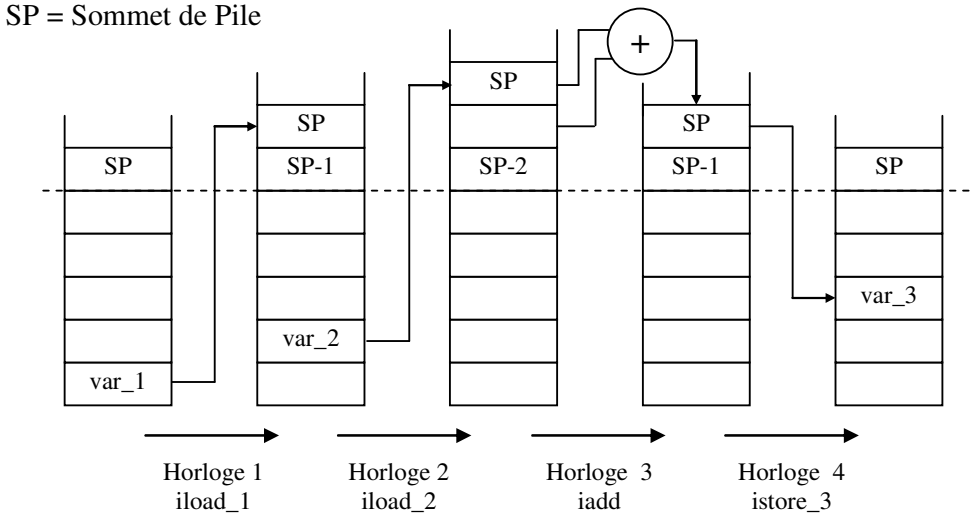


Figure 3-5 - Exécution sans groupement d'instructions.

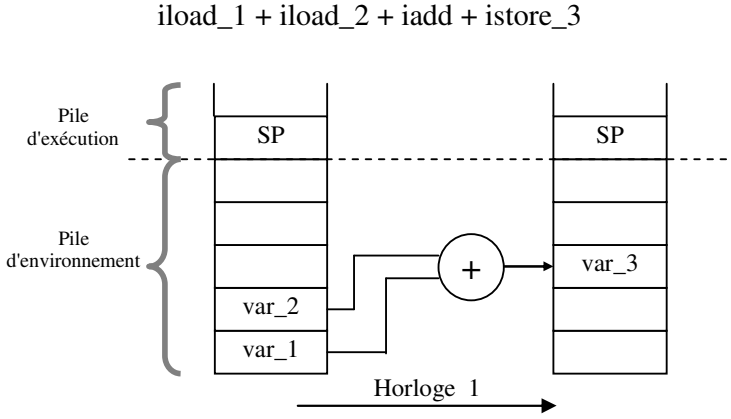


Figure 3-6 - Exécution avec groupement d'instructions.

Cette technique réduit le temps d'exécution des programmes puisqu'elle permet d'exécuter plusieurs instructions simultanément mais réduit également le nombre d'accès à la pile [47].

Les types d'instructions des machines JMS sont les suivants :

- NF : Non classable.
- LV : Chargement d'une variable locale 32-bits ou d'une constante 32-bits au sommet de la pile d'exécution.
- LV2 : Chargement d'une variable locale 64 bits au sommet de la pile d'exécution.
- OP : Opération sur les deux données 32 bits du sommet de la pile.
- OP2 : Opération sur les deux données 64 bits du sommet de la pile.
- OP12 : Opération sur la donnée 32 bits du sommet de la pile, renvoie une donnée 64 bits.
- OP21 : Opération sur la donnée 64 bits du sommet de la pile, renvoie une donnée 32 bits.
- BG1 : Opération sur la donnée 32 bits du sommet de la pile qui ne peut pas être suivie d'un rangement mémoire de variable locale.
- BG2 : Opération sur la donnée 64 bits ou les deux données 32 bits du sommet de la pile qui ne peut pas être suivie d'un rangement mémoire de variable locale.
- MEM : Opération de rangement en mémoire d'une variable locale de 32-bits.
- MEM2 : Opération de rangement en mémoire d'une variable locale de 64-bits.

Les machines JMS permettent de grouper un maximum de quatre instructions, l'ensemble du groupement n'excédant pas 7 octets consécutifs. Les groupements d'instructions qui en découlent sont les suivants :

- 1 : LV LV OP MEM
- 2 : LV LV OP
- 3 : LV LV BG2
- 4 : LV OP12 MEM2
- 5 : LV OP12
- 6 : LV OP MEM
- 7 : LV OP
- 8 : LV BG1
- 9 : LV BG2
- 10 : LV MEM
- 11 : LV2 LV2 OP2 MEM2
- 12 : LV2 LV2 OP2
- 13 : LV2 OP21 MEM
- 14 : LV2 OP21
- 15 : LV2 OP2 MEM2
- 16 : LV2 OP2
- 17 : LV2 BG2
- 18 : LV2 MEM2
- 19 : OP MEM
- 20 : OP21 MEM
- 21 : OP2 MEM2
- 22 : OP12 MEM2

3.2.d - Le chemin de données des modèles JMS

Le chemin de données du modèle JMS est celui d'un pipeline "classique" (Figure 3-7), les données sont chargées dans le pipeline à l'étage REG, les instructions arithmétiques et logiques ainsi que les instructions de saut sont effectuées à l'étage EX, les échanges avec la mémoire sont effectués à l'étage CACHE, enfin les données sont rangées dans le banc de registres à l'étage WriteBack.

Comme le langage Java est post-fixé, le banc de registre du processeur est remplacé par une pile matérielle. Celle-ci n'étant que le sommet de la pile Java, les adresses des données à charger dans le pipeline, aussi bien que celles des données à ranger dans la pile, peuvent ne pas appartenir à la pile matérielle. Ces données doivent donc être chargées dans le pipeline depuis la mémoire ou rangées en mémoire depuis celui-ci.

Un mécanisme de by-pass permet de tenir compte des dépendances de données et de ne pas geler le pipeline (voir 3.6.a -). Ce mécanisme est particulièrement sollicité par le langage Java dont la plupart des instructions agissent sur le sommet de la pile. Cependant, le groupement des instructions permet de réduire ce type de dépendances. En effet, un groupe peut contenir des chargements de données au sommet de la pile, une instruction qui s'exécute au sommet de la pile et une instruction de rangement de la donnée du sommet de la pile. Les registres présentés sur la Figure 3-7 représentent le sommet de la pile d'exécution correspondant à l'instruction en cours pour chaque étage du pipeline. Un registre supplémentaire, après l'étage WB permet de garder les données un cycle de plus car le mécanisme de by-pass s'effectue à l'étage EX et non à l'étage REG. Le même mécanisme de choix du "producteur" est présent pour le second opérande (flèche en pointillée Figure 3-7).

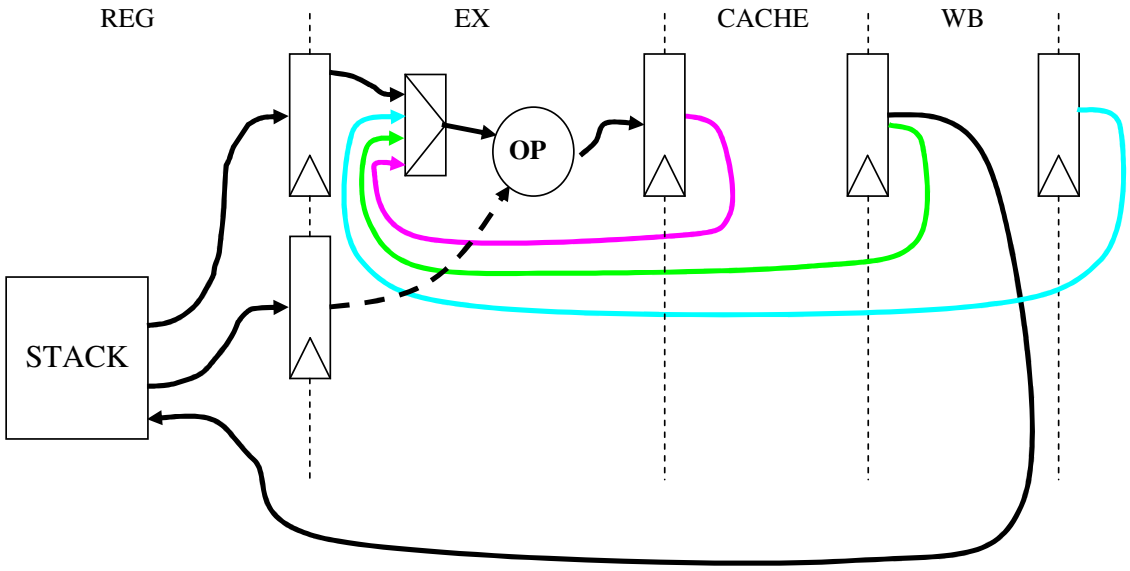


Figure 3-7 - Chemin de données des modèles JMS.

Le chemin de données de la machine JMS est similaire à celui du PicoJava-II à ceci près que celui du PicoJava-II traite des données 32-bits alors que celui de la JMS traite des données 64-bits.

3.3 - L'organisation matérielle de la JMS

Le matériel de la machine JMS s'organise autour de la pile matérielle. La Figure 3-8 montre l'organisation matérielle de la machine JMS.

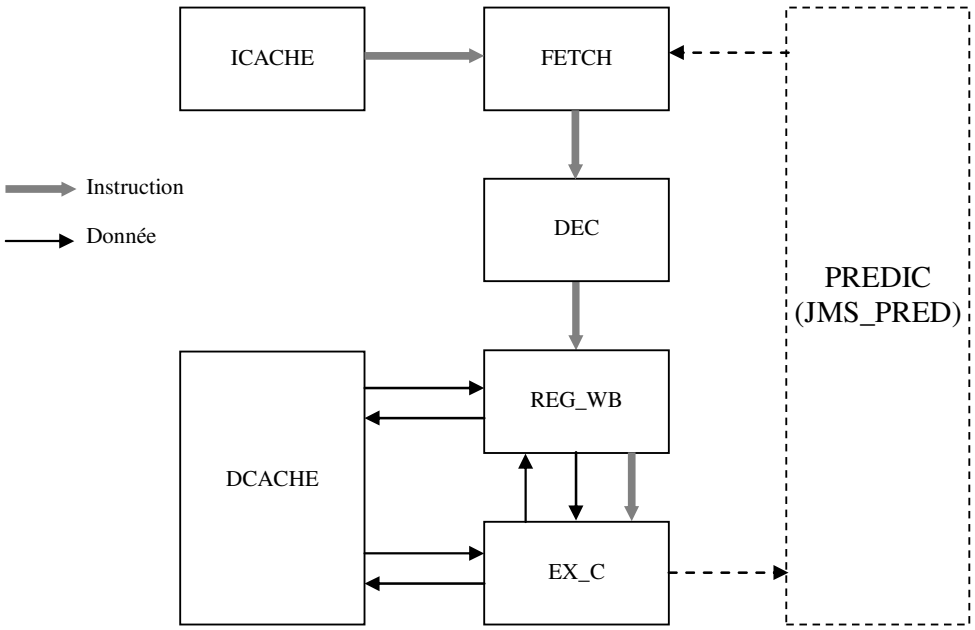


Figure 3-8 - Organisation matérielle de la JMS (JMS_PRED).

La machine JMS est constituée des modules principaux suivant :

- ICACHE : Contient le cache instruction.
- FETCH : Charge les instructions dans le pipeline (étage FETCH).
- DEC : Décode plusieurs instructions et les regroupe (étage DECODE).
- REG_WB : Contient la pile d'environnement, et en gère tous les accès (étages REG et WriteBack).
- EX_C : Effectue les opérations arithmétiques et logiques ainsi que les accès au cache de données (étage EX et CACHE).
- DCACHE : Contient le cache de données.
- PREDIC : Contient le prédicteur de branchement (uniquement pour le modèle JMS_PRED).

La fonction de chacun de ses modules sera développée dans la suite de ce chapitre, leurs architectures matérielles feront l'objet du prochain chapitre. Une partie des modules étant commune aux modèles JMS et JMQ, seul les modules spécifiques à la JMS seront développés ici. Les modules communs seront détaillés avec ceux spécifiques à la JMQ (3.5 - L'organisation matérielle de la JMQ).

3.3.a - Le module REG_WB

Le module REG_WR contient la pile matérielle, il gère le chargement des données dans le pipeline (étage REG), le rangement des données dans la pile (étage WriteBack), ainsi que les dépendances de données dans le pipeline. Ce module est également responsable de la gestion du dribbling de la pile. Il contient les étages suivants :

REG

L'étage REG charge deux données dans le pipeline à chaque cycle. Chacune de ces données peut être soit une variable locale, soit une constante, soit le sommet de la pile. En cas de MISS sur la pile, une requête de lecture du cache de données est effectuée, l'amont et l'étage REG du pipeline sont alors gelés en attente de cette donnée. L'adresse d'écriture dans la pile de l'instruction ou du groupe d'instructions effectuées est sauvegardée par le mécanisme de by-pass, qui vérifie également que les données à insérer dans le pipeline ne s'y trouvent pas déjà. Le cas échéant, le module REG_WB informe le module EX_C d'utiliser pour un des opérandes (ou pour les deux) une donnée déjà présente dans le pipe-line comme on peut le voir sur la Figure 3-7.

WriteBack

L'étage WriteBack reçoit la donnée à écrire dans la pile du module EX_C. L'adresse de rangement de cette donnée est fournie par le mécanisme de by-pass qui fait correspondre à chaque étage de pipeline l'adresse de rangement de la donnée qu'il contient. En cas de MISS (l'adresse de destination n'est pas contenue dans la pile matérielle), une requête d'écriture de la donnée vers le cache de données est effectuée. Tous les étages du pipeline sont alors gelés.

Le "Dribbling"

Le module REG_WB, en tant que gestionnaire de la pile matérielle, doit veiller à un bon taux de remplissage de la pile. La pile matérielle est l'extension physique de la pile d'environnement. Elle ne contient que son sommet. Le nombre d'entrées dans la pile, ainsi que l'adresse de son sommet sont constamment modifiés. La pile matérielle doit donc compenser ces modifications en se remplissant depuis le cache de données ou en se vidant dans celui-ci afin de toujours avoir un nombre suffisant de données dans la pile matérielle. Ce phénomène est appelé "dribbling". Le dribbling ne se déclenche qu'à partir du franchissement des limites du nombre de données dans la pile, défini en mode superviseur. Il se fait de manière non prioritaire aussi longtemps que le nombre d'entrées dans la pile matérielle, n'excède pas le nombre d'entrées maximum (60) ou n'est pas inférieur au nombre d'entrées minimum (6) devant se trouver dans la pile. Au-delà de ces limites, les étages du pipeline du processeur sont gelés. Le processus de dribbling devient alors prioritaire. L'adresse du sommet de la pile est contenue dans le registre OPTOP, tandis que l'adresse du fond de la pile matérielle est contenue dans le registre SC_BOTTOM.

3.3.b - Le module EX_C

Le module EX_C effectue toutes les instructions autres que celles correspondant aux manipulations de la pile. Il en existe deux versions, l'une destinée au modèle JMS et l'autre destinée au modèle JMS_PRED. Ces deux versions ne diffèrent que dans la gestion des sauts. Ce module se compose des étages suivants :

EX

L'étage EX effectue les calculs sur les données chargées dans le pipeline. Il contient l'unité de calcul flottant (FPU) et les registres internes du processeur. Cet étage exécute également les instructions d'appel ou de retour de méthode. Les instructions ayant besoin de plusieurs cycles pour s'exécuter gèlent les étages précédents du pipeline. Certaines de ces instructions commandent la lecture ou l'écriture, en tenant compte des dépendances, de données dans la pile par le module REG_WB. Ces instructions sont exécutées par du microcode stocké dans une ROM pour le PicoJava-II et par un automate dans le cas de la JMS.

L'étage EX calcule également les adresses d'accès au cache de données lors des instructions de lecture et d'écriture mémoire.

L'étage EX exécute toutes les instructions de saut. Il fait lui-même les sauts dans le cas de la JMS. Pour le modèle JMS_PRED, l'étage EX donne toutes les informations relatives aux sauts. Dans les deux modèles, il résout les branchements conditionnels.

CACHE

L'étage CACHE effectue les requêtes de lecture et d'écriture du cache de données. En cas de MISS du cache, tous les étages précédents du pipeline sont gelés.

3.4 - Le modèle JMQ (Java Machine on Queue)

Le modèle JMQ est un processeur Java à exécution sur file. La pile d'exécution est remplacée par une file circulaire. Ce modèle reprend les techniques utilisées pour les modèles JMS, à savoir :

- L'utilisation d'un buffer de pré-chargement des instructions de 16 octets.
- L'utilisation d'une pile matérielle de 64 registres 32-bits permettant de stocker le sommet de la "pile d'environnement".
- Le groupement d'instructions qui permet le décodage de plusieurs instructions simultanément et le groupement de celles-ci.

L'utilisation de deux organes différents pour stocker la pile d'environnement et la pile d'exécution entraîne un profond changement dans l'architecture et dans les mécanismes de groupement d'instructions ou de changement d'environnement. Cette modification permet également de mettre en place d'autres mécanismes permettant d'accélérer encore l'exécution du Bytecode Java.

Le modèle JMQ possède une unité de prédiction de branchement.

3.4.a - Le pipeline de la JMQ

L'utilisation d'une file pour simuler le fonctionnement d'une pile entraîne une désynchronisation de l'exécution des instructions de chargement de données et de calculs. Le pipeline de la JMQ peut alors être vu de deux manières différentes, soit comme un pipeline instruction (Figure 3-9), soit comme un pipeline de données (Figure 3-10).

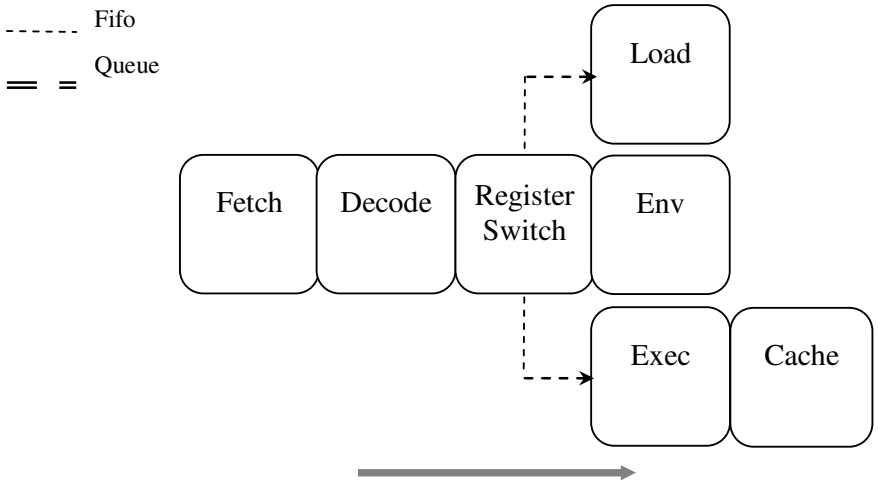


Figure 3-9 - Vision "pipeline instruction" de la JMQ.

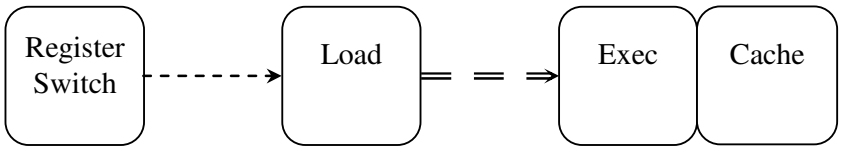


Figure 3-10 - Vision "pipeline de données" de la JMQ.

- Fetch : Charge 8 octets depuis le cache instruction vers un buffer 16 octets.
- Decode : Décode et regroupe jusqu'à 4 instructions d'une longueur maximale de 7 octets.
- Register Switch :
  - Charge les opérandes depuis la pile dans le pipeline, détermine les conditions de by-pass.
  - Oriente les instructions vers chacun des modules spécifiques.
- Env : Exécute les instructions d'environnement tel que les appels et les retours de méthode.
- Load : Charge les données, depuis la pile d'environnement, dans la file d'exécution.
- Exec : Effectue les opérations arithmétiques et logiques, calcule les adresses d'accès au cache de donnée, ainsi que la résolution des branchements.
- Cache : Effectue les échanges avec le cache de donnée.

Remarque : L'organisation de la machine JMQ rend optionnelle l'utilisation de l'étage cache par les instructions. Le rangement des données, soit dans l'espace des variables locales, soit dans la file d'exécution pourra donc être effectué en fonction de l'instruction exécutée dans les étages EXEC ou CACHE du pipe-line. On considérera par la suite le mécanisme d'écriture des variables locales dans la pile d'environnement comme un étage de pipe-line concurrent des étages EXEC ou CACHE et appelé WriteBack.

3.4.b - Les structures de la JMQ

Comme indiqué précédemment, le modèle JMQ utilise deux organes différents. La pile d'environnement qui est en charge de la gestion de l'environnement et la pile d'exécution qui est en charge de l'exécution.

La pile d'environnement

La pile d'environnement, comme pour les JMS, est stockée dans une file circulaire constituée de 64 registres 32 bits (Figure 3-11). Cette file possède 2 accès en lecture et 1 accès en écriture permettant l'accès à toutes les données quelle contient, elle est également pourvue d'un mécanisme de "dribbling" lui permettant de se vider et de se remplir dans, et à partir, du cache de données. Comme pour les modèles JMS, la partie matérielle de la pile permet de réduire les accès au cache de donnée. De même, les contenus du cache de données et de la pile matérielle ne sont pas cohérents.

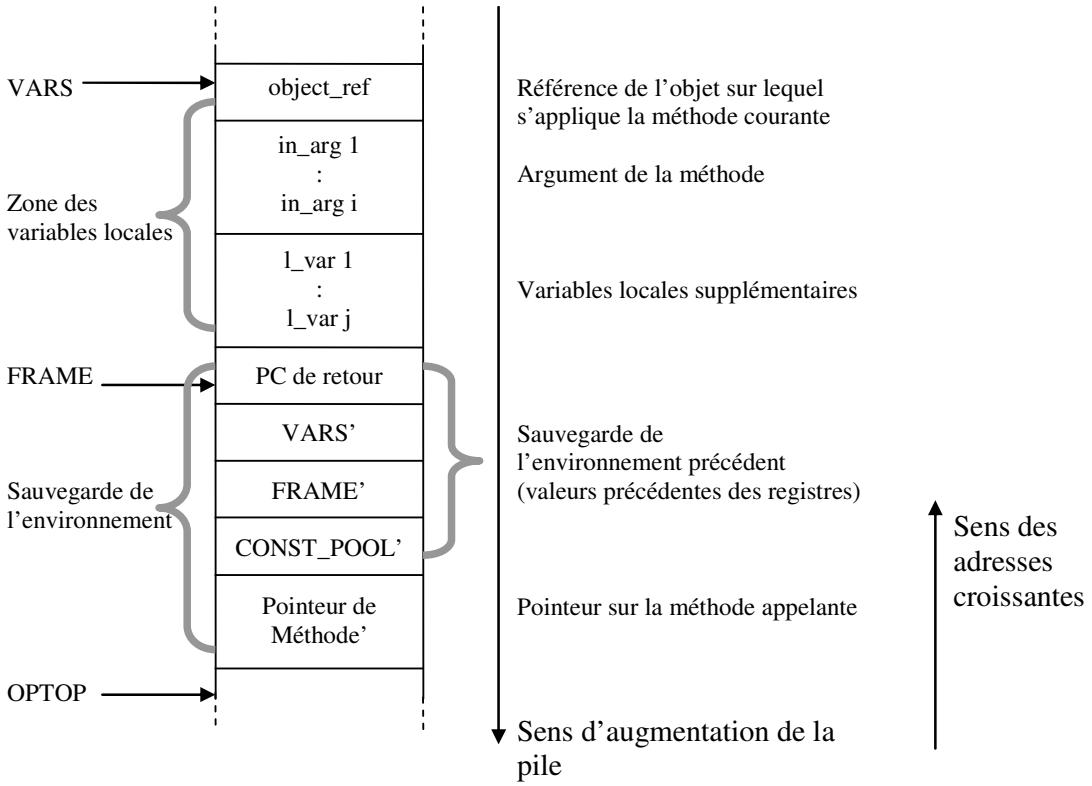


Figure 3-11 - La pile d'environnement de la JMQ.

Les registres de l'environnement de la JMQ reprennent ceux des JMS et sont les suivants :

- PC : pointe sur l'instruction courante.
- VARS : pointe sur la zone des variables locales de la méthode courante.
- FRAME : pointe la zone de sauvegarde de l'environnement précédent.
- CONST_POOL : pointe sur le constant pool de la classe de la méthode courante.
- Method_desc : pointe sur la zone de description de la méthode courante.
- OPTOP : pointe sur le sommet de la pile.

Le registre Method_desc a été ajouté pour avoir un accès plus facile à l'adresse de la méthode courante. En effet, dans le modèle JMS, le pointeur vers la méthode courante se trouve dans la pile d'environnement (Figure 3-2).

Un banc de 5 registres appelé "old" contient une copie des valeurs de sauvegarde des registres liés à la pile. Il a été ajouté afin d'accélérer le changement de contexte lors des appels et retours de méthode. Ces registres sont les suivants :

- oldPC : pointe sur l'instruction à effectuer après le retour de la méthode courante.
- oldVARS : pointe sur la zone des variables locales de la méthode appelante.
- oldFRAME : pointe sur la zone de sauvegarde de l'environnement précédant de la méthode appelante.
- oldCONST_POOL : pointe sur le constant pool de la classe de la méthode appelante.
- oldMethod_desc : pointe sur la zone de description de la méthode appelante.

La file d'exécution

L'exécution des opérations est réalisée dans une file circulaire ayant 66-bits de largeur. Chaque mot de cette file contient 64 bits de données, 1 bit de présence et 1 bits indiquant la taille de la donnée. La file d'exécution permet de charger des opérandes d'un coté pendant qu'on effectue des opérations de l'autre (Figure 3-12). Cette file émule le fonctionnement d'une pile du point de vue de l'exécution. En effet, deux pointeurs appelés P1 et P2 permettent d'accéder au sommet et au sous-sommet de la pile simulée en lecture et en écriture. Les positions des pointeurs P1 et P2 sont obtenues à partir de la position d'un pointeur appelé P dont le déplacement correspond au nombre de chargement d'opérande dans la file entre deux opérations. Les valeurs pointées par P1 et P2 sont alors les deux premières données valides à partir de P. Le résultat d'une opération sur deux opérandes est rangé au sommet (P1) au lieu du sous-sommet (P2) de manière à "déplacer" la zone de calcul vers l'amont. Deux pointeurs appelés P0_a et P0_b donnent chacun un accès en écriture, ce qui permet de charger dans la file jusqu'à deux données à chaque cycle.

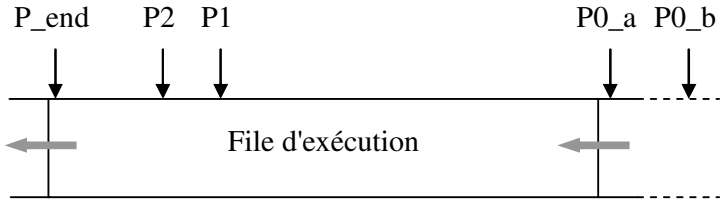


Figure 3-12 - File d'exécution de la JMQ.

L'appel de méthode

La séparation des piles d'environnement et d'exécution ainsi que l'ajout des registres précédemment cités entraîne des modifications de comportement lors de l'appel d'une méthode par rapport au modèle JMS. En effet, les arguments de la méthode appelée et la référence de l'objet se trouvent dans la file d'exécution et doivent être recopiés dans la pile d'environnement (Figure 3-13).

Le changement d'environnement commence par la sauvegarde des registres courants dans le banc de registres supplémentaires appelé old<reg>:

oldPC <= PC + taille de l'instruction d'appel

oldVARS <= VARS

oldFRAME <= FRAME

oldCONSTANT_POOL <= CONSTANT_POOL

oldmedhod_desc <= method_desc

Ceci permet de mettre en place les nouvelles valeurs des registres sans attendre leur recopie dans la pile d'environnement.

L'absence des arguments de la méthode dans la pile d'environnement permet de connaître immédiatement le nouvel emplacement de la zone de variable locale :

VARS <= OPTOP

Dès que le nombre d'arguments de la méthode est connu (3.2.b -), un nombre équivalent d'instructions de déplacement (32bits) est généré (pour une méthode ayant trois arguments, trois ordres de rangement du sommet de la pile d'exécution dans la pile

d'environnement seront générés) (Figure 3-13), les arguments ne sont alors plus présents dans la file d'exécution mais dans la pile d'environnement.

Les valeurs des registres sont alors mises en place :

FRAME <= OPTOP + nbargs + nbVars

OPTOP <= FRAME – 20

Dès que la zone de sauvegarde de l'environnement est connue, un mécanisme en tâche de fond (actif pendant les gels de l'étage de pipeline ou lorsqu'il n'y pas de variable locale à mettre à jour) recopie le banc de registres "old" dans la pile d'environnement (5 registres à recopier entre l'adresse FRAME et FRAME - 16). Cette recopie n'a pas besoin d'être terminée pour continuer l'exécution sur la méthode appelée. L'appel d'une nouvelle méthode ne pourra commencer que si ce mécanisme a terminé la recopie. Une demande de retour de méthode interrompt la recopie du banc "old".

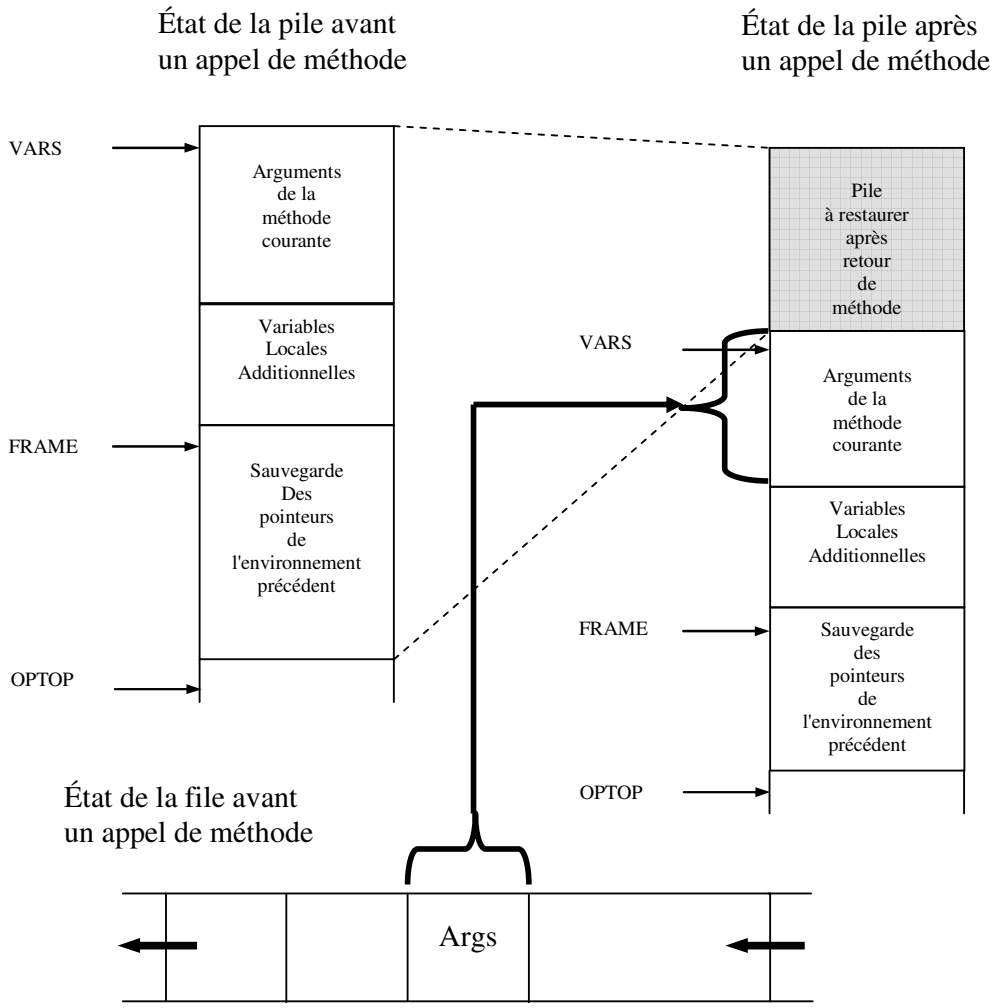


Figure 3-13 - Appel d'une méthode pour le modèle JMQ.

Une fois la mise à jour des registres terminée, le processeur saute à l'adresse de la méthode et l'exécution reprend.

Pour les méthodes d'instances (méthodes agissantes sur un objet), une exception est générée si la référence de l'objet est nulle.

Le retour de méthode

Comme pour l'appel de méthode, la séparation des piles ainsi que le banc de registres supplémentaire influent sur le déroulement du retour de méthode.

La donnée à retourner se trouve dans la file d'exécution et ne nécessite aucun déplacement puisqu'elle se trouve déjà au sommet de la pile d'exécution simulée (plus d'enchevêtrement pile d'exécution - pile d'environnement).

Le banc de registre "old" permet de commuter vers l'environnement de la méthode appelante en un cycle (Figure 3-14).

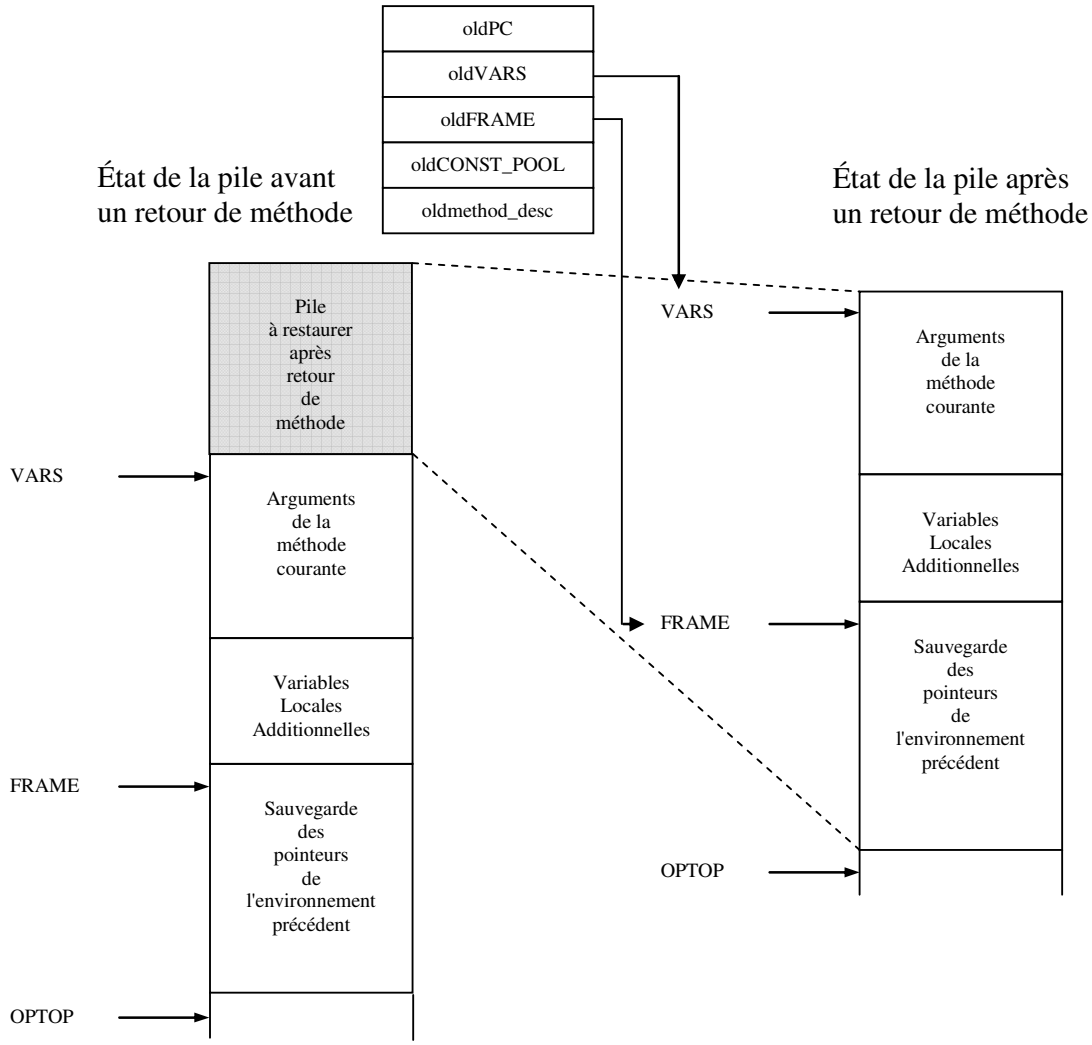


Figure 3-14 - Retour de méthode avec le modèle JMQ.

Toutes les valeurs des registres sont remplacées par celles contenues dans les registres de sauvegarde.

PC <= oldPC

VARS <= oldVARS

FRAME <= oldFRAME

CONSTANT_POOL <= oldCONSTANT_POOL

medhod_desc <= oldmethod_desc

L'exécution reprend alors depuis l'adresse pointée par le PC de retour.

La nouvelle valeur du sommet de la pile d'environnement doit alors devenir l'adresse de la zone de variables locales de la méthode retournée :

OPTOP <= VARS

Mais cette valeur ne peut pas être effective immédiatement car la partie exécution du processeur travaille encore sur la méthode dont on effectue le retour. La valeur du registre est donc placée dans une FIFO, la valeur du registre OPTOP ne sera mise à jour que lorsque la partie exécution aura effectué l'instruction de retour de méthode.

Un mécanisme, agissant en tâche de fond, remplit le banc de registre "old" avec les valeurs de l'environnement précédent de la nouvelle méthode courante. Ce mécanisme lit ces valeurs dans la pile d'environnement à raison d'un registre par cycle si l'étage de pipeline est en gel ou si moins de deux lectures de variables locales sont demandées. Si un nouveau retour de méthode est demandé, le remplissage du banc de registre "old" doit être terminé, sinon l'étage de pipeline se met en gel en attente de la terminaison du processus. L'appel d'une nouvelle méthode interrompt le processus de remplissage qui ne devient plus nécessaire, les registres du banc "old" recevant alors les nouvelles valeurs courantes des registres.

Les files de dépendances

Comme dans toutes les machines pipeline, il existe des dépendances de données entre les différents étages du pipeline de la JMQ. L'utilisation d'une file pour anticiper le chargement des données ainsi que de FIFOs pour découpler le décodage des instructions de leurs exécutions, amplifie ce phénomène. En effet, on peut vouloir charger une donnée dans la file alors que le calcul de cette donnée n'a pas encore été effectué. Il faut donc prévoir un mécanisme permettant de gérer ces dépendances.

Le mécanisme de by-pass destiné à gérer ces dépendances de données est appelé files de dépendances. Il est constitué de deux files circulaires : l'une contenant les données à charger dans la file d'exécution et appelé file de dépendance de lecture ; l'autre contenant les adresses des données à ranger dans la pile d'environnement et appelé file de dépendance d'écriture.

Lors d'une demande de lecture d'une variable locale, la donnée correspondante, présente dans la pile d'environnement, est recopiée dans la file de dépendance de lecture. La file de dépendance d'écriture est interrogée simultanément pour savoir s'il n'existe pas une écriture pendante à la même adresse. En cas d'écriture pendante, l'adresse du registre de la file de dépendance d'écriture ayant la même adresse d'écriture est écrite dans la file de dépendance de lecture et, la donnée est alors marquée "à recopier". S'il n'y a pas d'écriture pendante, la donnée est marquée "valide" ou "invalide" en fonction de la présence de la donnée dans la pile matérielle. La donnée est lue ultérieurement lors de son chargement dans la file d'exécution.

Lors d'une demande d'écriture de variable locale dans la pile d'exécution, l'adresse de la donnée est écrite dans la file de dépendance d'écriture. Lors de l'écriture de la variable

locale, la donnée est écrite dans la pile d'environnement à l'adresse contenue dans la file de dépendance d'écriture (si l'adresse se trouve dans la pile), la file de dépendance de lecture est interrogée simultanément pour savoir si la donnée doit être recopiée. Le cas échéant, la donnée est mise à jour dans la file de dépendance de lecture et marquée "valide".

3.4.c - Les regroupements d'instructions du modèle JMQ

Le regroupement d'instructions par le modèle JMQ utilise les mêmes principes que celui de la JMS, à savoir le décodage multiple d'instructions qui seront exécutées en parallèle par des matériels indépendants.

L'architecture spécifique de la JMQ entraîne la création d'un nouveau type d'instruction grâce à la séparation de la gestion de l'environnement et de l'exécution. En effet, les instructions liées à l'environnement utilisent désormais du matériel différent de celui utilisé par l'exécution, ce qui permet un certain parallélisme.

L'utilisation de FIFOs pour découpler le décodage des instructions et leur exécution par les modules concernés permet également une indépendance entre le chargement des opérandes et l'exécution des opérateurs aux deux extrémités de la file d'exécution. Cela entraîne les opportunités suivantes :

- Groupement des instructions agissant sur des données 32-bits et 64-bits indifféremment (plus de différentiation LV/LV2 ou OP/OP2).
- Un opérateur unaire peut être groupé avec deux lectures de variables locales.
- Un opérateur peut être suivi d'une ou deux lectures de variables locales.

Il en résulte un regroupement plus "agressif" des instructions (Figure 3-15).

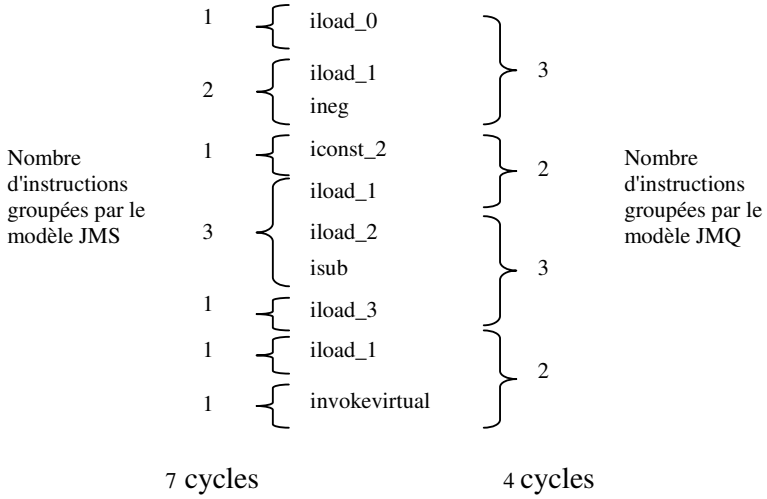


Figure 3-15 - Comparaison des groupements d'instructions des modèles JMS et JMQ.

Les types d'instructions de la machine JMQ sont les suivants :

- NF : Non classable.
- LV : Chargement d'une variable locale 32-bits ou 64bits, d'une constante 32-bits au sommet de la pile d'exécution ou d'un registre.
- OP : Opération sur le sommet de la pile d'exécution.
- BG : Opération sur le sommet de la pile d'exécution qui ne peut pas être suivie d'un rangement mémoire de variable locale.
- MEM : Opération de rangement en mémoire d'une variable locale.
- ENV : Instructions liées à l'environnement.
- LVOP : Instructions appartenant aux types LV et OP.
- LVBG : Instructions appartenant aux types LV et BG.
- BGENV : Instructions appartenant aux types BG et ENV.

La machine JMQ permet de grouper un maximum de 4 instructions, l'ensemble du groupement n'excédant pas 8 octets consécutifs. Les groupements d'instructions qui en découlent sont les suivants :

1 :	LV	LV	OP	MEM	26 :	LV	MEM		
2 :	LV	LV	OP	ENV	27 :	LV	ENV		
3 :	LV	LV	OP		28 :	LVOP	MEM	ENV	
4 :	LV	LV	BG	ENV	29 :	LVOP	MEM		
5 :	LV	LV	BG		30 :	LVOP	ENV		
6 :	LV	LV	BGENV		31 :	LVOP	LV	ENV	
7 :	LV	LV			32 :	LVOP	LV		
8 :	LV	LVOP	MEM	ENV	33 :	LVBG	ENV		
9 :	LV	LVOP	MEM		34 :	LVBG	LV	ENV	
10 :	LV	LVOP	ENV		35 :	LVBG	LV		
11 :	LV	LVOP			36 :	OP	MEM	ENV	
12 :	LV	LVBG	ENV		37 :	OP	MEM		
13 :	LV	LVBG			38 :	OP	ENV		
14 :	LV	OP	MEM	ENV	39 :	OP	LV	LV	ENV
15 :	LV	OP	MEM		40 :	OP	LV	LV	
16 :	LV	OP	ENV		41 :	OP	LV	ENV	
17 :	LV	OP	LV	ENV	42 :	OP	LV		
18 :	LV	OP	LV		43 :	BG	LV	LV	ENV
19 :	LV	OP			44 :	BG	LV	LV	
20 :	LV	BG	ENV		45 :	BG	LV	ENV	
21 :	LV	BG	LV	ENV	46 :	BG	LV		
22 :	LV	BG	LV		47 :	BG	ENV		
23 :	LV	BG			48 :	MEM	ENV		
24 :	LV	BGENV			49 :	ENV			
25 :	LV	MEM	ENV		50 :	BGENV			

3.5 - L'organisation matérielle de la JMQ

Le matériel de la machine JMQ s'organise autour de la pile d'environnement et de la file d'exécution. La Figure 3-16 montre l'organisation de la machine JMQ.

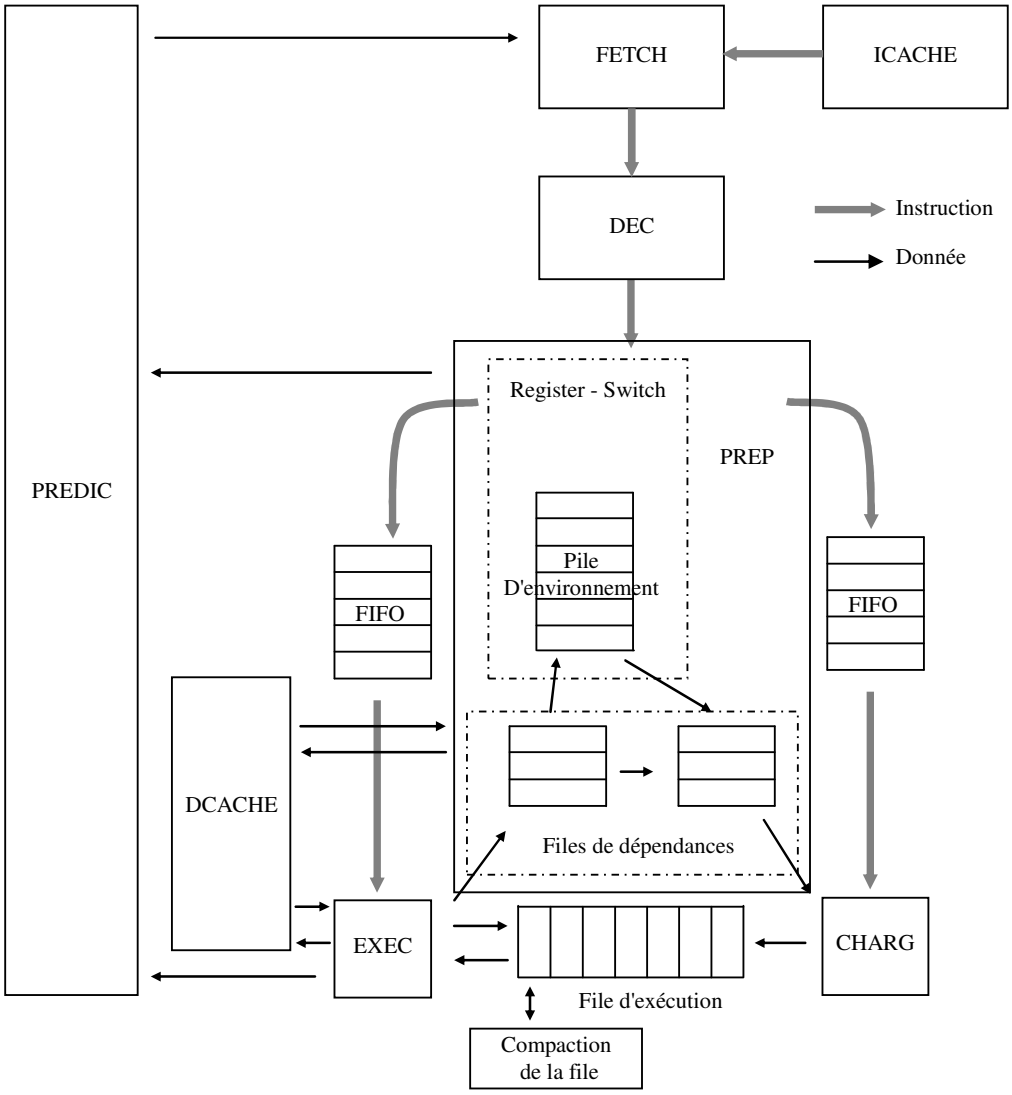


Figure 3-16 - Organisation matérielle de la JMQ.

Les flèches représentant le trajet des instructions dans le pipeline permettent de visualiser le pipeline instruction (Figure 3-9). Les flèches représentant le trajet des données permettent de visualiser le pipeline de donnée (Figure 3-10).

La machine JMQ est constituée de 9 modules principaux :

- ICACHE : Contient le cache instruction.
- FETCH : Charge les instructions dans le pipeline (étage FETCH).
- DEC : Décode plusieurs instructions et les regroupe (étage DECODE).
- PREP : Contient la pile d'environnement, il en gère tous les accès (étages REGISTER-SWITCH, ENV et WriteBack).
- CHARG : Charge les données dans la file d'exécution (étage LOAD).

- EXEC : Effectue les opérations arithmétiques et logiques ainsi que les accès au cache de données (étage EXEC et CACHE).
- FILE_EX : Contient la file d'exécution et les mécanismes de gestion qui lui sont associés.
- DCACHE : Contient le cache de données.
- PREDIC : Contient le prédicteur de branchement.

La fonction de chacun de ces modules va être vue plus en détails dans la suite de ce chapitre, leurs architectures matérielles feront l'objet du prochain chapitre.

3.5.a - Le module ICACHE

Le module ICACHE est un cache instruction développé par Frédéric Arzel au département SOC du LIP6 [48]. Ce cache est non bloquant, il permet donc au processeur d'effectuer une requête différente après un MISS du cache. La requête ayant effectué un MISS du cache continue à être traitée par le cache et la réponse à cette requête est positionnée sur l'interface du cache dès que celle-ci est présente dans le cache. Pour pouvoir traiter plusieurs requêtes du processeur, les requêtes et les réponses sont découplées et n'interviennent pas dans le même cycle. Le découplage dans le temps des requêtes et de leurs réponses nous entraîne à considérer le cache instruction comme un étage de pipeline supplémentaire. Ce cache accepte de nouvelles requêtes tant que le nombre de requêtes pendantes n'a pas atteint le nombre maximum pouvant être traité simultanément. Les réponses ne sont pas obligatoirement satisfaites dans l'ordre dans lequel le processeur les a effectuées, il devient donc nécessaire de donner un identifiant à chaque requête afin de pouvoir leur associer les réponses du cache instruction correspondantes.

Le cache instruction implémenté pour les modèles JMS et JMQ permet de transférer, à chaque cycle, 64-bits d'instructions alignées sur 64-bits. Il est capable d'absorber deux requêtes au maximum. Il permet de traiter des requêtes d'invalidation d'instruction dues au remplacement de celle-ci dynamiquement par leurs versions "quick" (2.2.b - Modification dynamique du Bytecode).

Ce type de cache est particulièrement intéressant dans le cas de plusieurs threads exécutés en temps partagé sur un processeur, le mécanisme de gestion des threads peut alors basculer d'un processus à l'autre en fonction de la disponibilité des instructions de chaque thread.

3.5.b - Le module FETCH

Le module FETCH est dédié au chargement des instructions depuis le cache instructions dans le pipeline. Ce module effectue un premier décodage des instructions en affectant le nombre d'octets nécessaire au décodage de l'instruction potentielle à chaque octet chargé. En effet, la taille des instructions Java étant variable, chaque octet ne représente pas obligatoirement un mnémonique d'instructions. De plus, les tailles associées permettent de savoir si une instruction est complète lors de son décodage.

Un buffer d'instructions permet de réduire le nombre d'accès au cache instructions. Ce buffer contient, pour chaque octet d'instruction, la taille de l'instruction potentielle correspondante et sa validité. Une nouvelle requête d'instructions est générée à chaque cycle à concurrence de deux requêtes pendantes simultanées. L'ordre des réponses n'étant pas forcément l'ordre des requêtes, un mécanisme permet de réordonnancer les réponses avant de les insérer dans le buffer d'instructions. Le buffer d'instructions ne peut recevoir de nouvelles instructions que s'il a 8 octets libres.

Le même module FETCH est implémenté sur les modèles JMQ et JMS_PRED, Il permet d'invalider les instructions se trouvant derrière un branchement prédit "pris" lors de la

réponse du cache instruction. Cette fonctionnalité n'est pas implémentée sur le module FETCH du modèle JMS car il ne possède pas d'unité de prédiction de branchement. Toutes les requêtes étant alignées 64-bits, les instructions chargées qui ne correspondent pas à l'adresse réelle de la demande sont invalidées.

3.5.c - Le module DECODE

Le module DECODE lit huit octets présents dans le buffer d'instructions, puis les décode et les regroupe. Chaque octet décodé correspond potentiellement à une instruction et se voit affecter un type (3.2.c -) (3.4.c -). Le premier octet contenu dans le buffer d'instructions correspond forcément au mnémonique d'une instruction et sa taille permet de connaître l'emplacement de la deuxième instruction, de même la taille de la seconde instruction permet de connaître l'emplacement de la suivante et ainsi de suite. L'emplacement de chaque instruction permet de grouper un maximum de quatre instructions correspondant au groupe décrit précédemment. La longueur du groupement d'instructions ne peut pas excéder 7 octets et ne doit contenir que des instructions complètes. Le module DECODE informe le module FETCH du nombre d'instructions consommées, pour permettre leur invalidation dans le buffer d'instructions. Les modules DECODE destinés aux modèles JMS et au modèle JMQ diffèrent uniquement dans les types associés aux instructions et les groupes formés avec celles-ci.

3.5.d - Le module PREP

Le module PREP contient la pile d'environnement et regroupe donc plusieurs étages du pipeline de la machine JMQ. En effet, les étages REGISTER-SWITCH, ENV et WriteBack doivent accéder à la pile d'environnement.

REGISTER-SWITCH

L'étage REGISTER-SWITCH traite en parallèle les instructions destinées aux différents étages du pipeline.

Les instructions de type LV sont transmises au module CHARG par l'intermédiaire d'une FIFO. Deux instructions de type LV peuvent être traitées à chaque cycle. Si l'une de ces instructions (ou les deux) correspond(ent) au chargement d'une variable locale, la pile d'environnement est accédée afin de charger la(es) donnée(s) dans le pipeline. Les données sont alors écrites dans une file, en attente d'être consommées par le module CHARG lors de l'exécution de l'instruction correspondante. Cette file d'attente est appelée file de dépendance de lecture et fait partie du mécanisme de by-pass entre les modules EXEC et CHARG. Les instructions de lecture des registres internes sont traitées de la même façon et les valeurs des registres sont insérées dans la file de dépendance de lecture. Chaque instruction de type LV comprise entre deux instructions transmises au module EXEC est comptabilisée afin de pouvoir générer les déplacements ultérieurs du sommet de la pile d'exécution simulée. En cas de MISS sur la pile, l'étage REGISTER-SWITCH doit effectuer une requête de lecture du cache de données. La réponse, une fois obtenue, sera écrite dans la file de dépendance de lecture.

Les instructions de type ENV sont transmises à la partie gestion de l'environnement du module PREP.

Les instructions de type OP, BG et MEM sont transmises au module EXEC par l'intermédiaire d'une FIFO. Une instruction de type MEM peut être transmise parallèlement à une instruction de type OP. Chaque transmission d'instruction au module EXEC est enrichie par deux informations supplémentaires. Ces informations sont, d'une part, la valeur du déplacement du pointeur de sommet de pile simulée correspondant, comme indiqué précédemment, au nombre de chargements d'opérandes entre deux opérations et, d'autre part,

la présence d'une instruction de retour de méthode dans le groupement d'instructions. Les instructions de modification des variables locales (type MEM) et les instructions de modification des registres internes du processeur sont également partiellement exécutées. Les adresses des données à ranger dans la pile d'environnement sont insérées dans une file d'attente appelée file de dépendance d'écriture. Les écritures de variables locales, et de registres internes, restent pendantes jusqu'à leurs exécutions par l'étage WriteBack du pipeline.

ENV

L'étage ENV du pipeline traite les instructions liées à l'environnement d'exécution des méthodes Java. Il est donc chargé d'exécuter les instructions de modifications des registres internes du processeur, de la sauvegarde de ces registres lors de l'appel d'une méthode et de leur remise en place lors d'un retour de méthode. Il est également chargé de la prise du "trap-handler" lors des instructions devant être émulées par des routines logicielles. Il communique avec l'unité de prédiction de branchement afin de mettre à jour les informations sur les branchements. Il n'effectue pas la résolution des branchements conditionnels.

WriteBack

L'étage WriteBack est chargé d'effectuer les écritures de variables locales dans la pile d'environnement. Il utilise pour cela les adresses contenues dans la file de dépendance d'écriture. En cas de MISS sur la pile, il est chargé d'effectuer une requête d'écriture sur le cache de données.

Le "Dribbling"

Le module PREP, en tant que gestionnaire de la pile d'environnement, doit veiller à un bon taux de remplissage de la pile. Comme pour le modèle JMS, la pile matérielle est l'extension physique de la pile (d'environnement) et n'en contient que le sommet. Nous sommes donc confrontés au même besoin de compensation du nombre d'entrées dans la pile suite à la modification de l'adresse du sommet de celle-ci. Le dribbling de la pile de la JMQ obéit au même impératif que celui de la JMS : il ne se déclenche qu'à partir du franchissement des limites du nombre de données dans la pile, défini en mode superviseur. Il se fait de manière non prioritaire aussi longtemps que le nombre d'entrées dans la pile matérielle, n'excède pas le nombre d'entrées maximum (60) ou n'est pas inférieur au nombre d'entrées minimum (6) devant se trouver dans la pile. En dehors de ces limites, le processus devient prioritaire. Contrairement au modèle JMS, seuls les étages du pipeline du processeur se trouvant en amont de la pile sont gelés. L'adresse du sommet de la pile est contenue dans le registre OPTOP, tandis que l'adresse du fond de la pile matérielle est contenue dans le registre SC_BOTTOM.

3.5.e - Le module CHARG

Le module CHARG charge les données dans la file d'exécution. Il reçoit les instructions le concernant depuis une FIFO lui permettant de ne pas geler l'amont du pipeline lors des cycles d'attentes d'une donnée. Il est capable d'insérer deux données 64-bits dans la file d'exécution à chaque cycle. Chacune de ces données peut être soit chargée depuis la file de dépendance de lecture, soit une constante générée par le module lui-même.

3.5.f - Le module EXEC

Le module EXEC traite toutes les instructions opérant sur le sommet de la pile d'exécution simulée. Il reçoit les instructions le concernant depuis une FIFO lui permettant de ne pas geler l'amont du pipeline lors des cycles d'attentes d'une donnée ou lors d'instructions complexes nécessitant plusieurs cycles. Il lit les opérandes fournis par le module FILE_EX correspondant au sommet de la pile d'exécution simulée et renvoit le résultat, soit dans la file au sommet de la pile simulée, soit vers le module PREP pour écriture dans la pile d'environnement s'il s'agit de l'écriture d'une variable locale. Il calcule les adresses de lecture et d'écriture dans le cache de données et effectue les accès au cache dans un second cycle. Il résout les branchements conditionnels et transmet le résultat au module PREDICT.

3.5.g - Le module FILE_EX

Le module FILE_EX contient la file d'exécution ainsi que le mécanisme de compaction des vides présents dans la file.

Le remplissage de la file d'exécution est effectué par le module CHARG via deux pointeurs d'écritures appelés P0_a et P0_b. Ces deux pointeurs permettent d'écrire dans la file deux données dans des positions consécutives de la file, à chaque cycle.

L'exécution des instructions sur la file d'exécution se fait en deux phases, tout d'abord, un pointeur P est déplacé au "sommet" de la file et les pointeurs P1 et P2 en sont déduit (ils sont recalculés à chaque fois que l'on autorise le déplacement de P), puis on effectue le calcul sur les données pointées par P1 et P2 au cycle suivant. Pour connaître le nouveau sommet (P) de la file, il faut connaître le nombre de données insérées dans la file depuis la dernière opération. Ce nombre est fourni par le module EXEC qui donne également les ordres de déplacement de P. Le calcul de la nouvelle position de P peut se faire en parallèle du calcul sur les données de la file, P ne sera déplacé que s'il ne dépasse pas les données chargées à l'autre bout de la file. Les pointeurs P1 et P2 donnent accès en lecture et en écriture aux données qu'ils pointent.

Contrairement à une pile, les opérandes lus pour effectuer un calcul ne disparaissent pas de la file. Il est donc nécessaire de prévoir un bit supplémentaire d'état associé à chaque mot de la file pour indiquer si la donnée présente dans cette case serait également présente dans une pile. La file étant circulaire, les données qui ne sont pas consommées rapidement risquent d'empêcher l'introduction de nouvelles données. Un mécanisme de compaction des données non-valides présentes dans la file doit donc être implémenté. Ce mécanisme est un processus non prioritaire, il cherche à déplacer les données valides de l'extrémité de la file vers le sommet P de la pile simulée. Les schémas de la Figure 3-17 à la Figure 3-21 montrent le fonctionnement du mécanisme de compaction de la file d'exécution.

Un pointeur P_end représente l'extrémité de la file. Le pointeur P1_end pointe sur la première donnée valide précédant une case vide de la file en partant de P_end. Le pointeur P2_end pointe sur la première case vide précédant une donnée valide à partir de P1_end (Figure 3-17).

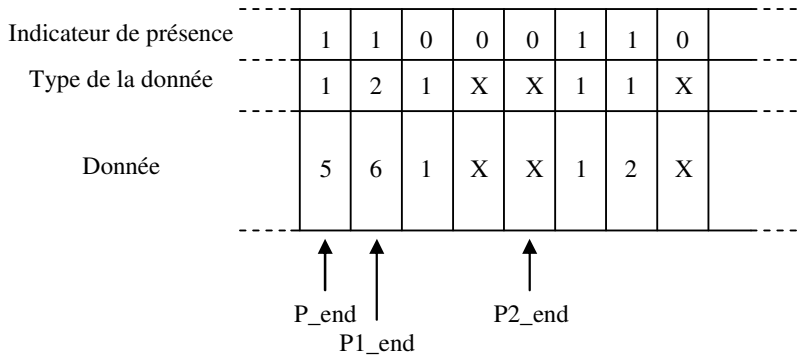


Figure 3-17 - Compaction de la file d'exécution au temps T.

Pour ne pas utiliser d'accès en lecture ou en écriture supplémentaire, le déplacement de la donnée pointée par P1_end, vers le registre pointé par P2_end, se fait par l'intermédiaire des ports de lectures et d'écritures dédiés au module EXEC. Ce dernier donne l'ordre de déplacement d'une donnée à chaque cycle dans lequel il n'effectue ni lecture ni écriture (Figure 3-18 et Figure 3-20). La recherche des nouvelles positions de P1_end et de P2_end ne se fait pas en parallèle du déplacement de la donnée (Figure 3-19).

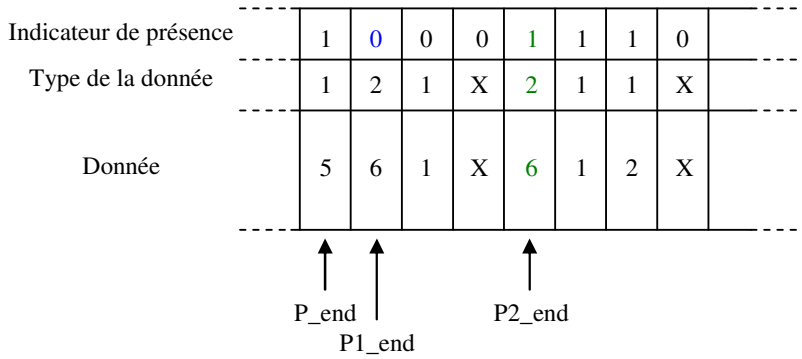


Figure 3-18 - Compaction de la file d'exécution (T+1).

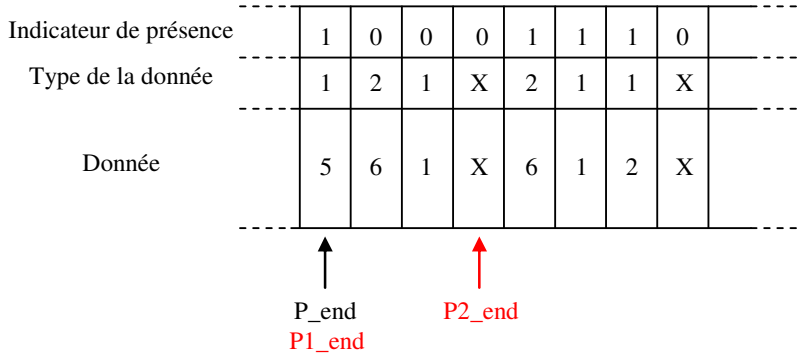


Figure 3-19 - Compaction de la file d'exécution (T+2).

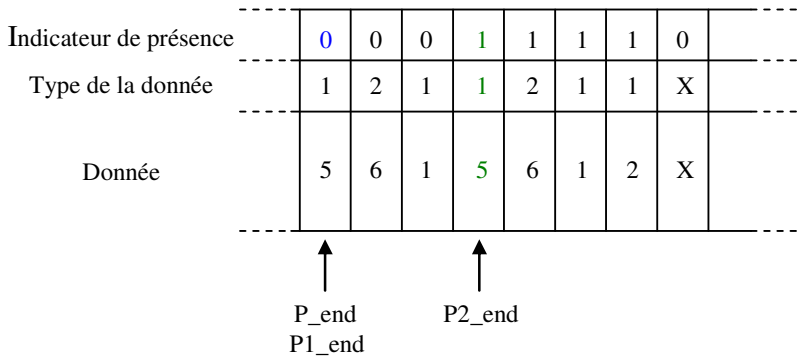


Figure 3-20 - Compaction de la file d'exécution (T+3).

En cas de déplacement ou d'invalidation de la donnée pointée par P_end, la nouvelle position de l'extrémité de la file est recherchée, le pointeur P_end est alors mis à jour (Figure 3-21). Aucun déplacement de donnée n'est possible au cycle suivant, ce cycle étant dédié à la recherche des nouvelles positions de P1_end et de P2_end.

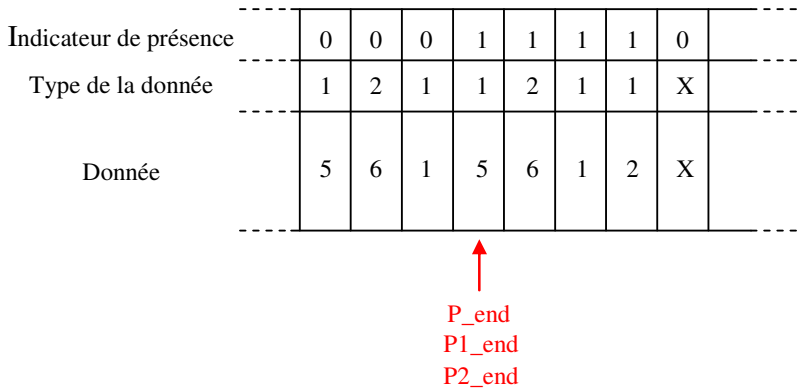


Figure 3-21 - Compaction de la file d'exécution (T+4).

3.5.h - Le module DCACHE

Le module DCACHE est un cache de donnée simulé par de simples tirages aléatoires. Il possède deux ports de lecture de donnée et deux ports d'écriture de donnée. Lors des lectures, un tirage aléatoire est effectué pour générer des MISS du cache. Le taux de HIT est fixé à 90%, le temps de latence lors des MISS est de 8 cycles.

3.5.i - Le module PREDIC

Le module PREDIC est une unité de prédiction de branchement développée par Jérôme Dumesnil au département SOC du LIP6 [49]. Ce mécanisme de prédiction dynamique exploite les informations recueillies pendant l'exécution afin de prédire les prochaines occurrences des branchements ainsi que leur direction dès le chargement des instructions de saut dans le pipeline. Les branchements se divisent en quatre catégories communes à tous les processeurs plus une spécifique au Bytecode :

- Conditionnels : Branchements effectués en fonction d'une condition (l'adresse de saut est déterminée à partir de l'adresse courante plus un index compris dans l'instruction).

- Inconditionnels immédiats : Branchements dont l'adresse de saut est déterminée à partir de l'adresse courante plus un index compris dans l'instruction.
- Inconditionnels indirects : Branchement dont l'adresse de saut est contenue dans un registre.
- Retour de sous-programme : Equivalent aux branchements inconditionnels indirects mais toujours précédés d'un appel de fonction.
- Non-sauvegardés : Branchements ne devant pas être sauvegardé par le prédicteur.

Les branchements non-sauvegardés correspondent aux instructions remplacées après résolution du constant pool (pas de sauvegarde de la prise du gestionnaire d'interruption et à l'instruction de retour de la routine de modification de l'instruction).

Les informations recueillies par le prédicteur sont :

- L'adresse du branchement.
- L'adresse de saut (dernière occurrence pour les branchements inconditionnels indirects).
- Type du saut.

Il existe plusieurs méthodes de prédiction de branchement pour les branchements conditionnels, l'approche retenue est celle dite du compteur à saturation [50]. Cette méthode de prédiction utilise un Branch Target Buffer (BTB) qui sauvegarde les informations sur les branchements rencontrés et un compteur 2-bits pour enregistrer l'historique de chaque branchement conditionnel. Si le branchement est pris, le compteur correspondant est incrémenté, si le branchement n'est pas pris, le compteur est décrémenté. Lorsque qu'une instruction de branchement conditionnel correspondant à une adresse contenue dans le BTB est insérée dans le pipeline, le prédicteur de branchement est interrogé (par le module FETCH) pour connaître l'adresse de la prochaine instruction à charger. Si le compteur correspondant indique –1 ou 0 le branchement n'est pas pris et l'exécution continue à l'adresse consécutive, si le compteur indique 1 ou 2 le branchement est pris et le prédicteur donne l'adresse de saut au module FETCH. Le prédicteur de branchement permet également de prédire les autres types de saut dès la requête de lecture du cache instructions correspondante. Ces autres types de saut n'utilisent pas le compteur 2-bits. Les instructions de type "retour de sous-programme" accèdent à une pile sauvegardant les adresses de retour au moment de l'appel de la méthode ou du sous-programme.

Les modèles JMS_PRED et JMQ utilisent des versions différentes de ce module car l'ajout de branchement et la résolution des branchements conditionnels est faite à deux étages différents du pipeline de la JMQ, alors que ces deux opérations sont simultanées dans le pipeline que la JMS_PRED.

3.6 - Exemple de fonctionnement

Cette partie montre le fonctionnement des modèles JMS et JMQ sur un exemple très simple. Cet exemple vise à dérouler l'exécution des instructions agissant sur la pile (file JMQ) d'exécution Java. La Figure 3-22 montre la transformation en instructions du ByteCode du programme Java utilisé pour l'exemple. La Figure 3-23 montre les groupements effectués par les modules DECODE des modèle JMS et JMQ sur cette suite d'instructions.

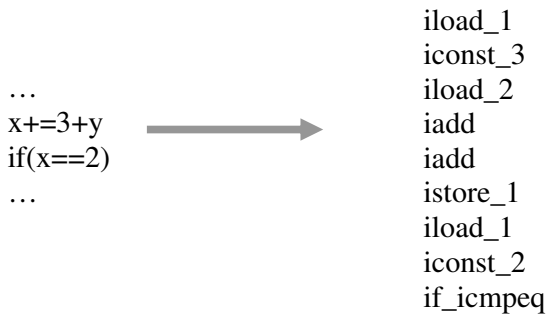


Figure 3-22 - Exemple de transformation de programme Java en ByteCode.

Regroupements JMS	**Regroupements JMQ**
iload_1	iload_1 + iconst_3
iconst_3 + iload_2 + iadd	iload_2 + iadd
iadd + istore_1	iadd + istore_1
iload_1 + iconst_2 + if_icmpeq	iload_1 + iconst_2 + if_icmpeq

Figure 3-23 - Regroupements JMS et JMQ.

3.6.a - Exécution sur le modèle JMS

Les schémas de la Figure 3-24 à la Figure 3-29 montrent l'exécution des regroupements d'instructions JMS présenté par la Figure 3-23.

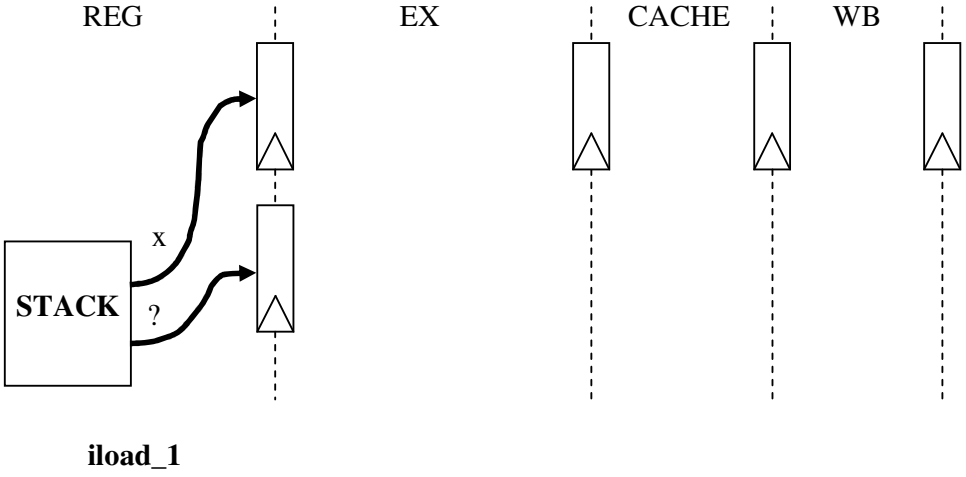


Figure 3-24 - Exemple JMS au temps T.

La suite d'instructions commence par le chargement dans le pipeline de données de la variable locale 1 (x) par l'étage REG (Figure 3-24), cette variable devient le nouveau sommet de la pile. Le sommet actuel de la pile est également chargé dans le pipeline comme second opérande bien que celui-ci ne soit pas utilisé par la suite.

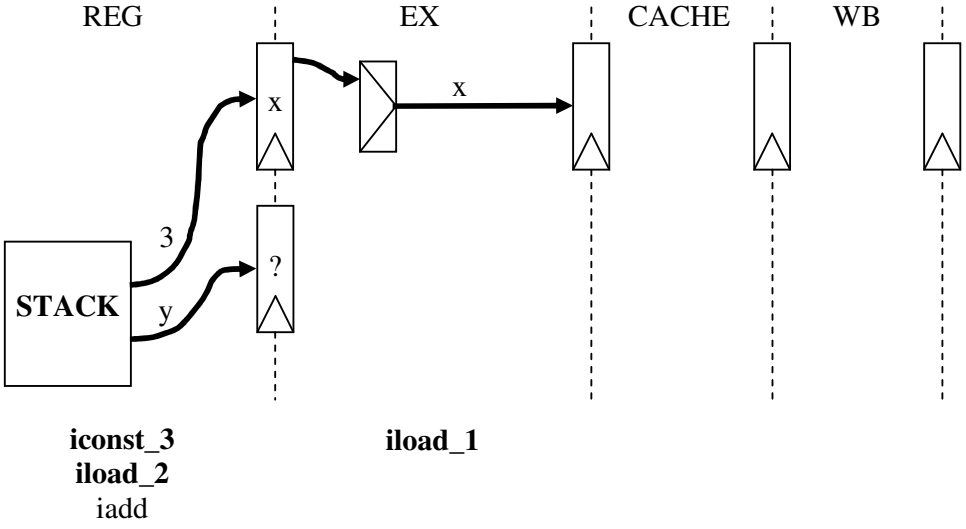


Figure 3-25 - Exemple JMS (T+1) .

Le second groupe d'instructions est partiellement exécuté au temps T+1 par le module REG, la variable locale 2 (y) et la constante 3 sont alors insérés dans le pipeline (Figure 3-25). Parallèlement, l'instruction "iload_1" est exécutée par l'étage EX et la donnée "x" progresse dans le pipeline.

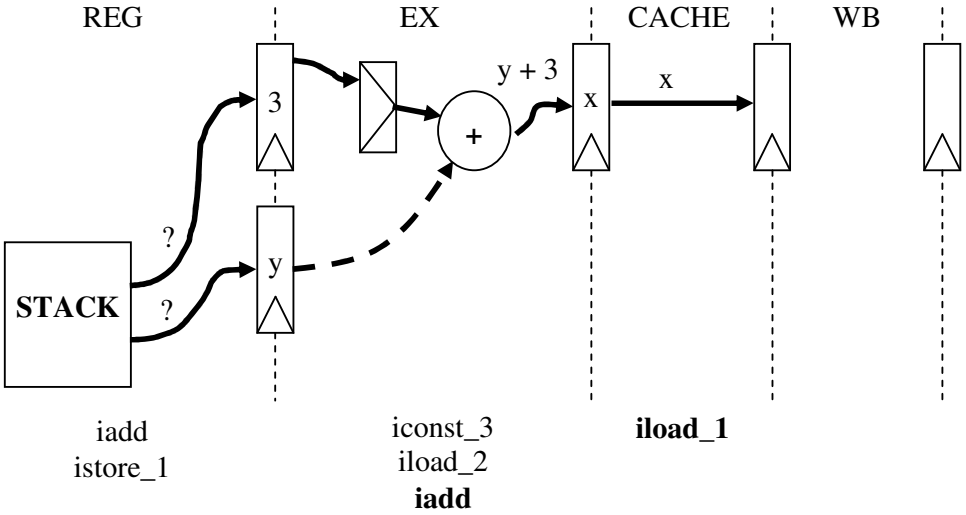


Figure 3-26 - Exemple JMS (T+2).

Le troisième groupe d'instructions est partiellement exécuté au temps T+2 par le module REG (Figure 3-26). Aucune instruction de chargement ne faisant partie de ce groupe, le sommet et le sous-sommet de la pile sont alors insérés dans le pipeline, les instructions de calcul agissant par défaut sur le sommet de la pile. Les valeurs insérées dans le pipeline sont fausses car elles sont le fruit d'instructions présentes dans le pipeline, elles seront remplacées ultérieurement par le mécanisme de by-pass. Parallèlement, l'instruction "iadd" du second groupe d'instructions est exécuté par l'étage EX. L'instruction "iload_1" ainsi que la donnée "x" continuent leurs progressions à l'étage CACHE.

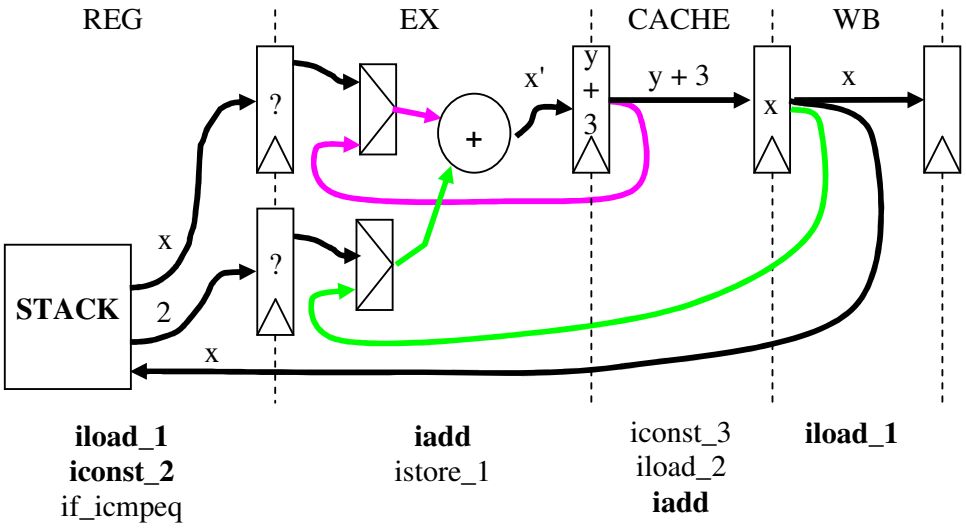


Figure 3-27 - Exemple JMS (T+3).

Les instructions de chargement de la variable locale 1 (x) et de la constante 2 du quatrième groupe d'instructions sont exécutées par le module REG au temps T+3 (Figure 3-27). La valeur de la variable locale 1 correspond à une ancienne valeur de x, le troisième groupe d'instruction contenant une instruction d'écriture de la variable x. Comme au cycle précédent, la valeur insérée dans le pipeline est fausse et devra être remplacée par le mécanisme de by-pass. L'étage EX traite l'instruction "iadd" appartenant au troisième groupe d'instructions et produit la nouvelle valeur de la variable locale 1 (x'). Pour ce calcul, le mécanisme de by-pass remplace le sommet et le sous-sommet de la pile par leurs valeurs, respectivement contenues aux étages CACHE et WB du pipeline. Ces mêmes données continuent leurs progressions dans le pipeline. L'étage WB écrit l'ancienne valeur de la variable locale 1 au sommet de la pile.

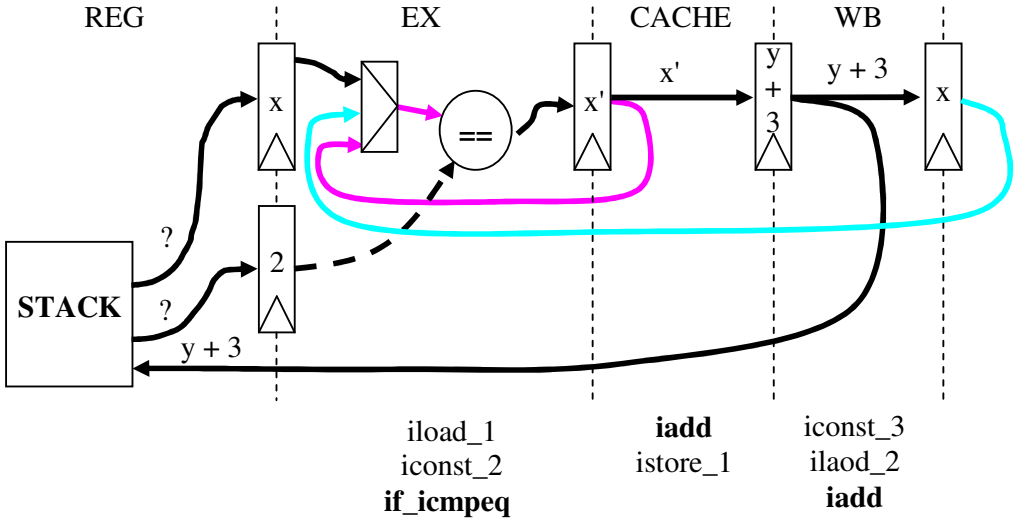


Figure 3-28 - Exemple JMS (T+4).

La Figure 3-28 montre l'exécution de l'instruction de saut conditionnel "if_icmpeq". Pour le modèle JMS, l'étage EX commande le saut. Pour le modèle JMS_PRED, l'étage EX transmet au prédicteur de branchement toutes les informations relatives au saut. Deux valeurs de la variable locale 1 sont présentes dans le pipeline, le mécanisme de by-pass remplace la valeur insérée au cycle précédent par la valeur la plus récente de x (notée x') présente à l'étage CACHE. Le modèle JMS ne possède pas de prédicteur de branchement, il peut donc être assimilé à un modèle prédisant, à chaque fois, la non-prise des branchements. En cas d'erreur de prédiction, les instructions présentes dans les étages FETCH, DECODE et REG sont invalidées et l'exécution reprend à la bonne adresse de branchement. Dans le cas contraire, l'exécution continue. L'étage WB écrit au sommet de la pile la valeur correspondant au résultat du second groupement d'instructions.

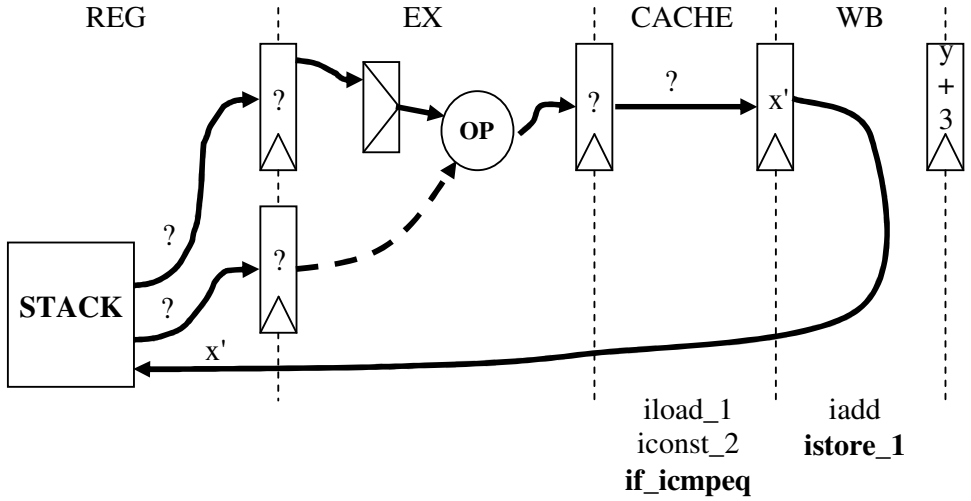


Figure 3-29 - Exemple JMS (T+5).

Au temps T+5, la nouvelle valeur de la variable locale 1 est écrite dans la pile par l'étage WB (Figure 3-29). Au temps T+6, aucune écriture ne sera faite dans la pile, l'instruction "if_icmpeq" ne renvoyant pas de résultat à stoker dans la pile.

3.6.b - Exécution sur le modèle JMQ

L'exécution de la même suite d'instructions par le modèle JMQ est soumise à des mécanismes différents de ceux utilisés par le modèle JMS. Cette exécution peut être considérée comme le produit de deux phases distinctes. Une première phase d'exécution suit le flot d'instructions et est effectuée par le module PREP. Cette première phase est désynchronisée de la seconde par un ensemble de FIFOs. La seconde phase est l'exécution des instructions par les modules CHARG et EXEC sur la file d'exécution, cette exécution est synchronisée par les données.

La Figure 3-30 montre le traitement effectué par le module PREP sur le flot d'instructions ainsi qu'une partie du mécanisme de by-pass permettant de prendre en compte les dépendances de données entre les modules CHARG et EXEC. Les instructions sont acheminées par le module PREP, vers les modules en charge de les exécuter, par l'intermédiaire de FIFOs. Ces instructions y sont également partiellement exécutées.

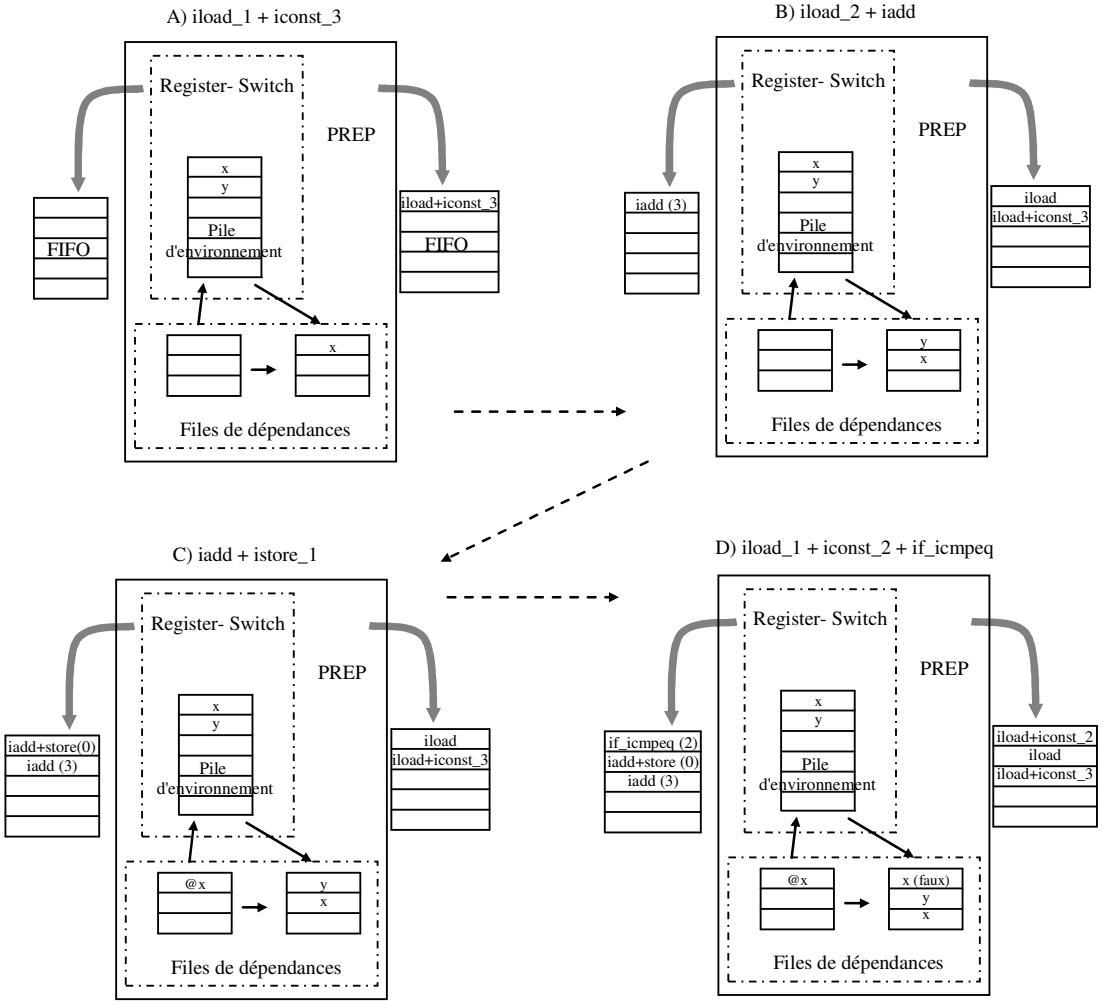


Figure 3-30 - Exemple d'exécution du module PREP.

- A) Ce groupe d'instructions est uniquement destiné au module CHARG, il est donc acheminé vers la FIFO correspondante par le module PREP. L'instruction "iload_1" est remplacée par l'instruction "iload" car l'index sert uniquement à connaître la position de la variable par rapport au pointeur VARS. Il en sera de même pour toutes les instructions de lecture d'une variable locale ou d'un registre interne du processeur. Les lectures de données 64-bit seront remplacées par l'instruction "lload". Par ailleurs, la donnée "x" correspondant au "iload_1" est insérée dans la file de dépendance de lecture par le module PREP.
- B) Le second groupe d'instructions contient une instruction de lecture de variable locale (iload_2) à acheminer au module CHARG et une instruction à acheminer au module EXEC (iadd). La donnée "y" correspondant à la lecture est insérée directement dans la file de dépendance de lecture. L'instruction destinée au module EXEC est enrichie du nombre d'opérandes chargés (3) depuis la dernière instruction d'opération au sommet de la pile Java.

- C) Ce groupe d'instructions (iadd + istore_1) est uniquement destiné au module EXEC, il est donc acheminé vers la FIFO correspondante par le module PREP. Comme pour l'instruction précédente, ce groupe d'instructions est enrichi par le nombre de chargement d'opérandes entre deux opérations. Comme aucun chargement n'a été effectué, il est égal à zéro. L'instruction "istore_1" est remplacée par un simple bit qui indique que le résultat de l'instruction doit être fourni au module PREP et non rangé dans la pile d'exécution. Il en serait de même lors de l'écriture d'un registre interne de processeur. L'adresse de rangement "@x" (dans la pile d'environnement) de la donnée correspondant à l'instruction "istore_1" est insérée dans la file de dépendance d'écriture.
- D) Dans ce groupe d'instructions, "iload_1" et "iconst_2" sont acheminées vers le module CHARG. La donnée "x" chargée dans la file de dépendance de lecture par l'instruction 'iload_1" est marquée "à recopier" car son adresse est présente dans la file de dépendance de lecture. L'instruction "if_icmpeq" est acheminée vers le module EXEC (avec le déplacement du pointeur de sommet de pile virtuelle (2)) et l'étage ENV du module PREP. Au cycle suivant, l'étage ENV transmettra les informations relatives à ce branchement au prédicteur de branchement.

Les FIFOs entre le module PREP et les modules CHARG et EXEC, ainsi que les files de dépendances permettent le parcours des instructions et l'échange de données entre ceux-ci ; elles rendent indépendantes l'exécution des flots d'instructions destinés à chaque module.

Les schémas de la Figure 3-31 à la Figure 3-35 montrent la deuxième phase d'exécution qui correspond à l'exécution des instructions par les modules CHARG et EXEC sur la file d'exécution.

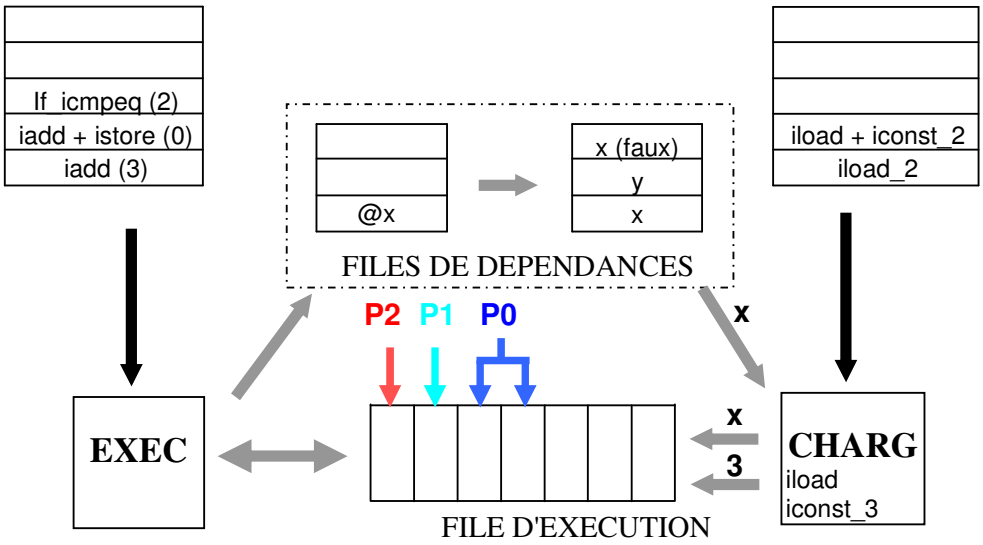


Figure 3-31 - Exemple d'exécution sur FILE au temps T.

La Figure 3-31 illustre le chargement dans la file d'exécution de la variable x (iload) et de la constante 3 (iconst_3) par le module de CHARG dans les registres pointés par P0_a et P0_b. Ces pointeurs sont ensuite déplacés pour pointer sur de nouvelles cases libres. La variable "x" est lue dans la file de dépendance de lecture. Le pointeur de lecture de la file de lecture est déplacé. Le déplacement des pointeurs P1 et P2 (sommets de la pile d'exécution

simulée) n'est pas effectué car seulement deux nouvelles données sont insérées, alors qu'un déplacement de trois est requis.

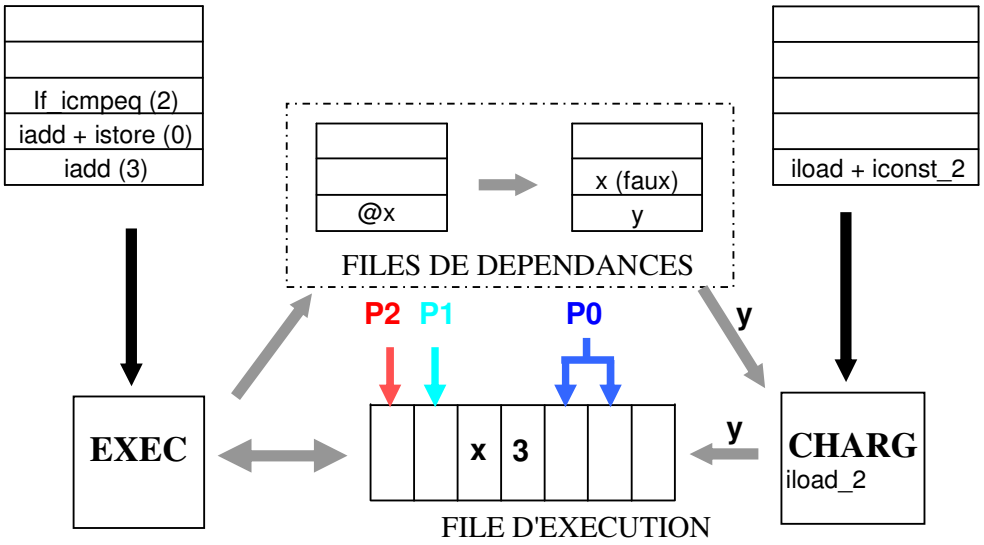


Figure 3-32 - Exemple d'exécution sur FILE (T+1).

Au temps T+1, le module CHARG insère la variable y dans la file d'exécution, lue à partir de la file de dépendance de lecture (Figure 3-32). Les pointeurs P0 sont déplacés. Le pointeur P (pointeur de sommet matériel de la pile virtuelle) peut être déplacé car le nombre de nouvelles entrées dans la file est suffisant. La nouvelle position des pointeurs P1 et P2 est ensuite calculée.

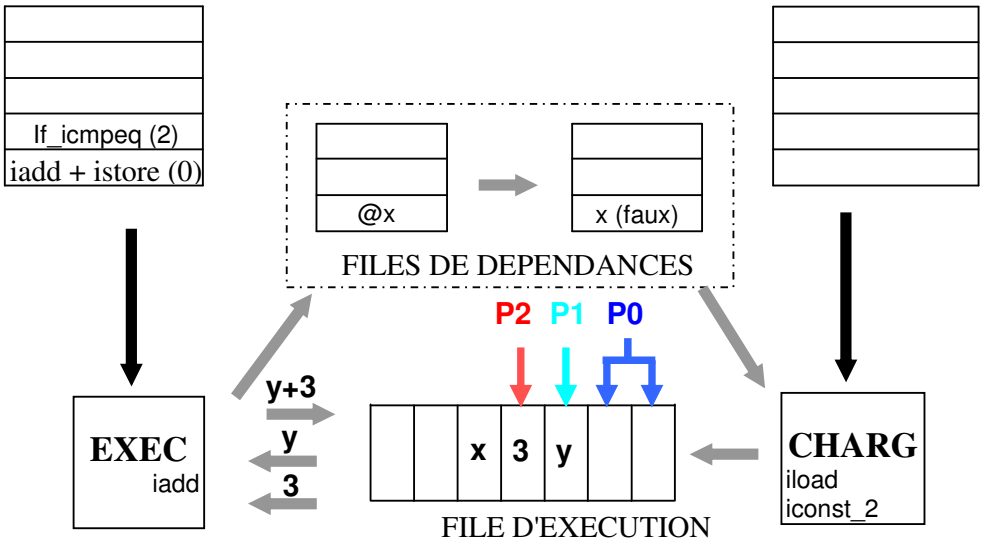


Figure 3-33 - Exemple d'exécution sur FILE (T+2).

Au temps T+2 (Figure 3-33), Le module EXEC effectue l'addition sur les deux données extraites de la file d'exécution via les pointeurs P1 et P2 et renvoie le résultat dans le registre pointé par P1. Le registre pointé par P2 est invalidé. Le pointeur P2 est déplacé sur le nouveau sous-sommet de la pile simulée. La valeur de x contenu dans la file de dépendance

de lecture étant toujours invalide, l'instruction iload ne peut pas être exécutée par le module CHARG. De ce fait, aucune donnée n'est chargée dans la file.

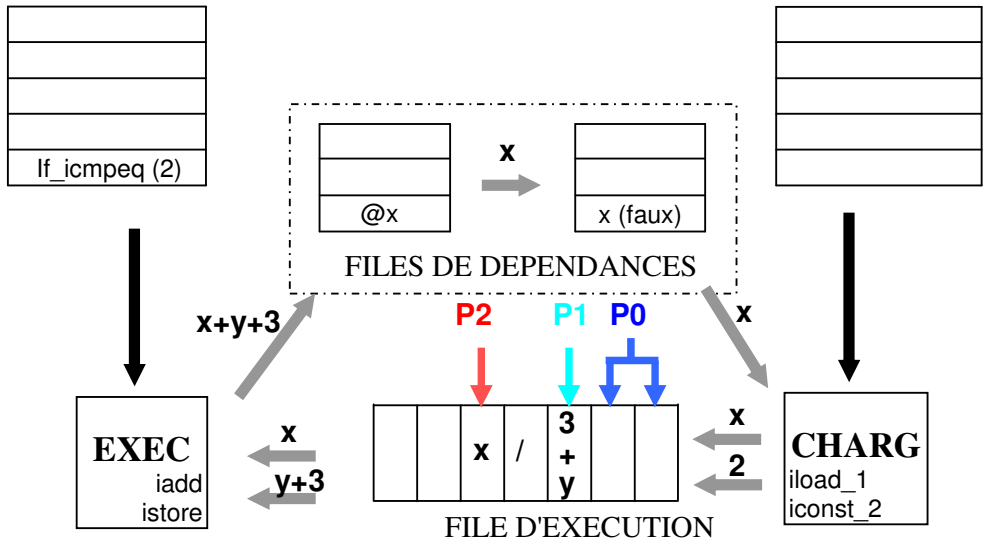


Figure 3-34 - Exemple d'exécution sur FILE (T+3).

Au temps T+3 (Figure 3-34), Le module EXEC effectue l'addition sur les deux données extraites de la file d'exécution et renvoie le résultat vers la file de dépendance d'écriture (module PREP). La donnée est écrite dans la pile d'environnement et mise à jour dans la file de dépendance de lecture. Elle est simultanément fournie par le mécanisme de by-pass au module CHARG qui l'insère dans la file d'exécution. La constante 2 est insérée dans la file d'exécution. Les pointeurs P0 sont déplacés. Le pointeur P est déplacé et les positions des pointeurs P1 et P2 en sont déduites.

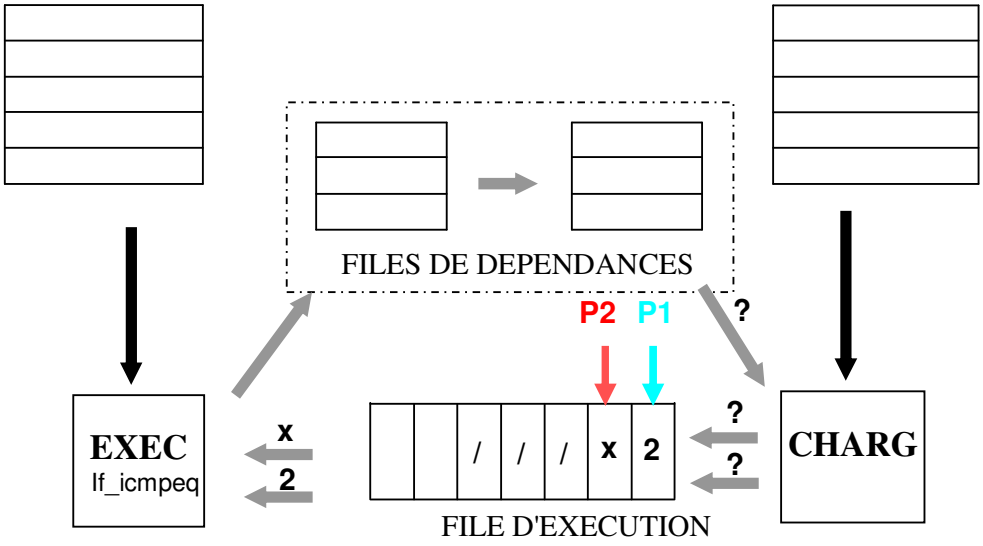


Figure 3-35 - Exemple d'exécution sur FILE (T+4).

Au temps T+4 (Figure 3-35), le module EXEC effectue la comparaison sur les données extraites de la file d'exécution. Le résultat de la comparaison est fourni au module

PREDIC. Si la résolution du branchement correspond à la prédiction effectuée, l'exécution continue. Si la prédiction est mauvaise, le pointeur P0_a prend la position P+1 (P0_b <= P+2), les files de dépendances et les FIFOs sont vidées, tous les étages du pipeline sont invalidés et l'exécution reprend à la bonne adresse de branchement.

3.7 - Conclusion

Dans ce chapitre, nous avons décrit l'architecture des trois modèles de processeurs développés au cours de cette thèse. L'architecture du modèle JMS est basé sur celle du Picojava-II et a été amélioré afin de grouper les instructions qui agissent sur les données 64 bits. Ce modèle sert de point de comparaison. Le modèle JMS_PRED est un modèle JMS auquel ont a ajouté une unité de prédiction de branchement. Le modèle JMQ est le processeur Java à exécution sur file, il reprend les techniques utilisées par le modèle JMS pour l'exécution du Bytecode et intègre des mécanismes supplémentaires afin d'améliorer la vitesse d'exécution des programmes Java.

Ce chapitre décrit les mécanismes d'accélérations utilisés dans les différents modèles présentés.

Enfin, nous avons déroulé un exemple de simulation du Bytecode Java sur les modèles JMS et JMQ.

Chapitre 4 Architecture Matérielle

Ce chapitre présente l'architecture matérielle des différents modules qui constituent les modèles JMS et JMQ. Les symboles utilisés sont décrits en annexe (A.a).

4.1 - Le cache instruction ICACHE

Le cache instruction est constitué de 5 modules (Figure 4-1 [48]) :

- REQ gère la réception des requêtes venant du processeur.
- DATA contient les instructions.
- MSHR (Miss Status-info Holding Register) est une file d'attente qui gère l'envoi et la réception de requêtes avec la mémoire.
- RSP gère les réponses vers le processeur.
- VCI permet de transformer les requêtes du MSHR en requêtes VCI et réciproquement (en cas de hiérarchie de cache mémoire, ce module n'est présent que sur le niveau de cache le plus proche de la mémoire).

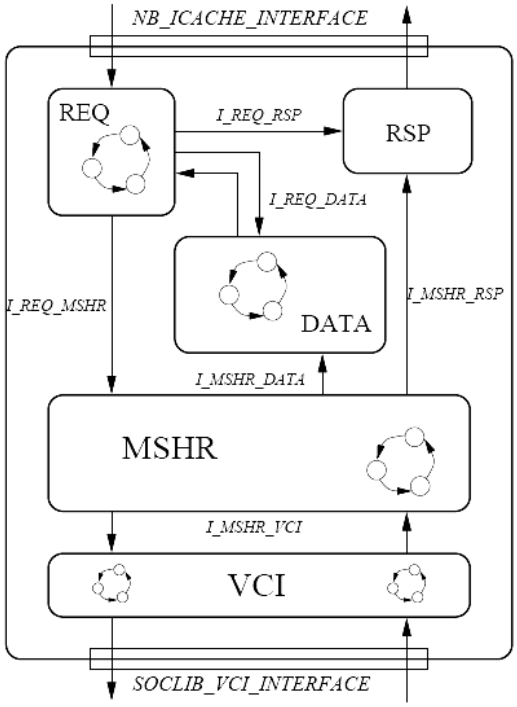


Figure 4-1 - Diagramme du cache instruction.

4.1.a - REQ

Le module REQ reçoit les requêtes du processeur. Si le nombre maximum de requêtes n'est pas atteint, il interroge les modules DATA et MSHR simultanément pour les satisfaire. Le cache permet de traiter trois types de requête : lectures, flush (invalidation de donnée) et prefetch (préchargement d'instruction dans le cache sans demande de réponse vers le processeur).

En cas de requête de lecture, si le MSHR contient la donnée, il la transmet au module RSP au cycle suivant. Dans le cas contraire, Si le module DATA contient la donnée, il la

transmet à REQ qui la transmettra à RSP au cycle suivant. Enfin, si il y a un double MISS, une demande d'allocation pour cette requête est envoyée au module MSHR.

En cas de requête de flush, il invalide la donnée si elle se trouve dans le cache et transmet le flush au niveau supérieur de cache.

En cas de requête de prefetch, si le MSHR ou le module DATA contiennent la donnée rien ne se passe. Dans le cas contraire, une demande d'allocation pour cette requête est envoyée au module MSHR.

4.1.b - DATA

Le module DATA contient les instructions présentes dans le cache. L'automate, DATA_FSM répond aux requêtes provenant des deux interfaces *I_MSHR_DATA* et *I_REQ_DATA*. Lors de requêtes de lecture faites par REQ, les instructions lui sont transmises en cas de HIT.

Le module MSHR positionne sur l'interface *I_MSHR_DATA* les blocs nécessitant d'être écrits dans le cache, suite à une requête de lecture ayant provoqué un MISS. Une ligne de cache est transmise en un cycle entre le MHSR et le module DATA, l'interface *I_MSHR_DATA* est donc dimensionnée en conséquence.

L'interface avec l'automate REQ n'est disponible que lorsqu'aucune mise à jour de ligne de cache n'est en cours. Réciproquement, quand une invalidation est effectuée par l'automate REQ, l'écriture d'un bloc transmis par le MSHR doit être mise en attente.

4.1.c - MSHR

Le MSHR est responsable de la gestion des requêtes vers la mémoire (ou le niveau supérieur de cache) et de la réception des réponses. Il est constitué de deux automates et de deux files (Figure 4-2 [48]) :

- ALLOC : insère les requêtes dans la Request Queue et la MSHR Queue.
- Request Queue : contient les informations sur les requêtes en cours.
- MSHR Queue : contient les requêtes à effectuer.
- RESP_FSM : gère les réponses aux requêtes effectuées.

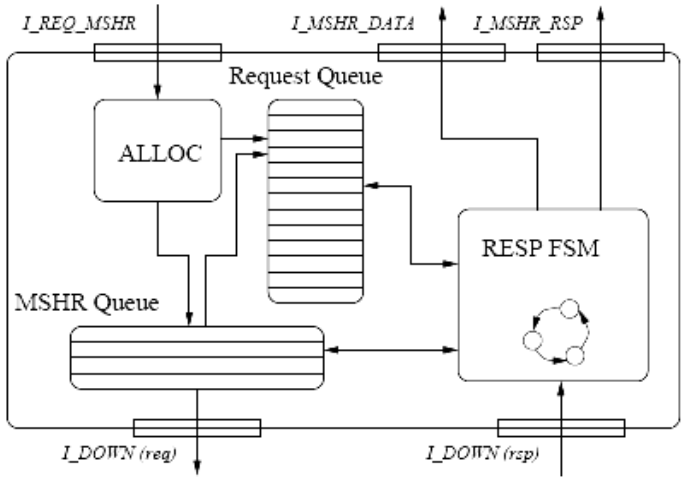


Figure 4-2 - Diagramme du MSHR.

Chaque entrée de la file MSHR est constituée des champs suivants (Figure 4-3 [48]) :

- V : Ce bit indique la validité d'une entrée de la file. Sa mise à zéro entraîne une désallocation de l'entrée.
- S : Ce bit indique que les requêtes d'écriture éventuellement nécessaires ont été émises vers le niveau inférieur (toute nouvelle écriture concernant le bloc associé à l'entrée du MSHR nécessitera donc l'allocation d'une nouvelle entrée).
- Type : de même taille que le champ utilisé par les requêtes (voir "Interfaces du modèle"), il s'agit de la recopie du type de la requête ayant provoqué l'allocation du MSHR. Ce champ est rendu nécessaire, d'une part car il convient de propager l'information sur la nature d'une requête dans le cas d'une hiérarchie à plusieurs niveaux, et d'autre part pour le cas du préchargement, puisque dans ce cas le bloc ne sera pas mis à jour dans la mémoire "classique" du cache, mais dans le *prefetch buffer*.
- Block address : adresse du bloc associé.
- Owner : champ identifiant le propriétaire futur de la ligne de cache quand elle aura été mise à jour.
- Flush : il s'agit d'un bit, qui sera mis à 1 si le MSHR est déjà alloué pour une ligne et qu'une requête d'invalidation pour cette même ligne arrive ; ceci permet, au moment ou le bloc mémoire correspondant parvient au cache, de savoir que la mise à jour du module DATA ne doit pas être effectuée. Le MSHR doit en effet rester alloué puisqu'une requête mémoire a éventuellement été effectuée; en effet, on a choisi que toute requête du niveau inférieur soit suivie de la réponse associée, même si celle-ci a été suivie d'une requête d'invalidation.

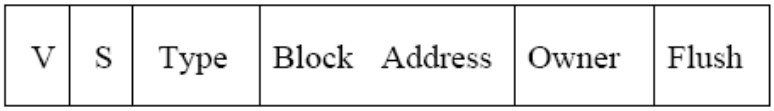


Figure 4-3 - Entrée de la file MSHR.

La file des requêtes est constituée des champs suivants (Figure 4-4 [48]) :

- V : Ce bit indique la validité d'une entrée de la file. Il est mis à zéro pour invalider une requête satisfaite.
- MSHRID : indique le numéro de l'entrée de la file MSHR associé à la requête. Ce numéro est utilisé par RESP_FSM à la réception d'une réponse depuis la mémoire (ou le niveau supérieur de cache) pour identifier immédiatement une requête en attente dans la *Request Queue*.
- REQID : Ce champ permet d'identifier une requête de façon unique, il représente la concaténation du numéro de port initial, de l'identifiant du demandeur (owner) et de l'identifiant de requête.
- OFFSET : adresse de la donnée au sein du bloc.

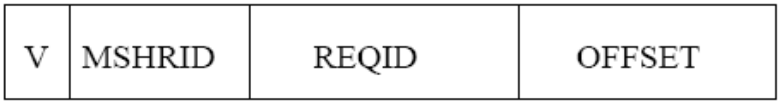


Figure 4-4 - Entrée de la file de requêtes.

4.1.d - RSP

Le module RSP est constitué essentiellement par un multiplexeur. Il gère l'envoi des réponses au processeur (ou au niveau inférieur de cache en cas de cache multi-niveau) qui proviennent soit du module REQ soit du MSHR. Les réponses provenant du MHSR sont prioritaires sur celles du module REQ en cas de double HIT car elles correspondent à des requêtes antérieures.

4.2 - Le module de chargement des instructions FETCH

Le module de chargement est piloté par un automate chargé de communiquer avec les interfaces de requête et de réponse du cache instructions. Il est donc responsable de l'émission de requêtes vers le cache instructions et de la réception des instructions associées. Cet automate permet la gestion de deux requêtes de lecture pendantes. En effet, le cache instructions est non-bloquant et permet donc l'envoi de plusieurs requêtes de lecture. Une seule requête est effectuée à chaque cycle à condition que le nombre de requête maximum ne soit pas atteint et que le cache instruction puisse la recevoir. L'automate permet de réordonnancer les réponses du cache instructions qui sont susceptibles d'être reçues dans le désordre. Cet automate est détaillé en ANNEXE (A.e).

4.2.a - L'émission de requêtes

Les requêtes effectuées sont de deux types, soit une demande de lecture d'instructions (alignées sur 64-bits), soit une demande d'invalidation d'instructions (invalidation d'une ligne du cache instructions). La Figure 4-5 montre le mécanisme de requête et de gestion de la prochaine adresse de demande d'instructions. Elle présente également la partie matérielle supplémentaire en cas d'utilisation d'un prédicteur de branchement.

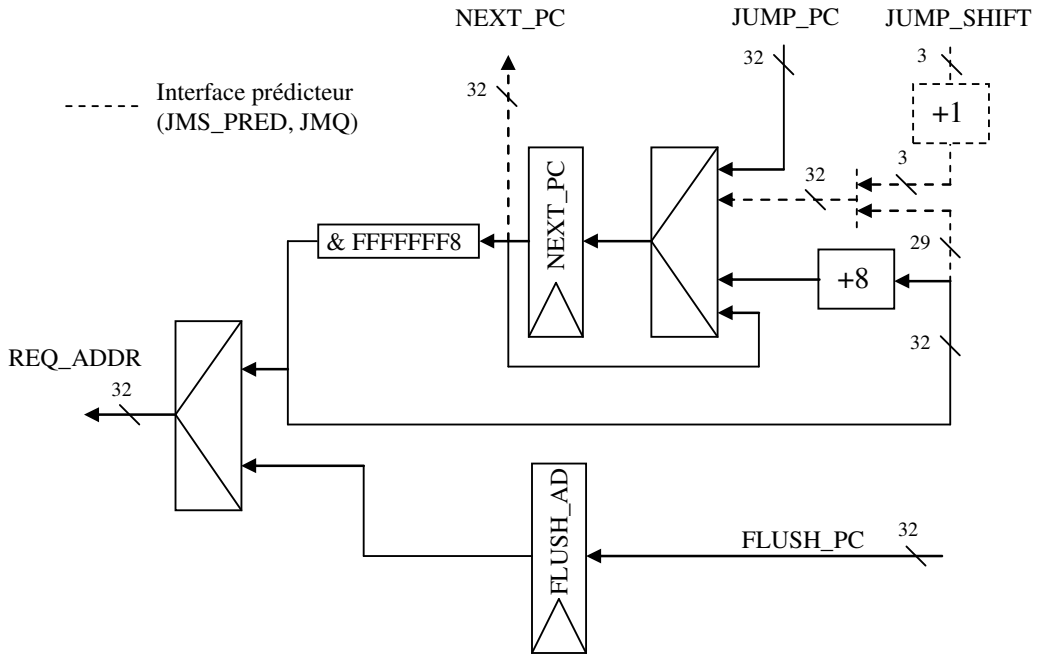


Figure 4-5 - Mécanisme de Requête du module Fetch.

La prochaine adresse de lecture d'instruction est, par ordre de priorité :

- JUMP_PC en cas de saut.
- NEXT_PC si aucune requête n'est effectuée (ou requête d'invalidation).
- NEXT_PC + 8 (aligné 64-bit) si une requête est effectuée.
- NEXT_PC (29 bits de poids fort) et JUMP_SHIFT+1 (3 bits poids faible) en cas de non-prise de branchement et présence d'un second branchement dans les mêmes 64 bits d'instructions (JMP_SHIFT représente la place du dernier octet de l'instruction de saut non-pris).

L'adresse NEXT_PC est présentée au prédicteur de branchement.

Les requêtes d'invalidation sont uniquement relayées par le module FETCH et sont prioritaires sur les requêtes de lecture.

4.2.b - Mécanisme de réception et de stockage des instructions

Les instructions présentes dans le module FETCH sont stockées dans un buffer d'instructions de 16 entrées qui permet de découpler le transfert des instructions et le décodage de celles-ci. Ce buffer contient les instructions, ainsi que les tailles associées à chaque octet représentant potentiellement une instruction Java et un bit de validité pour chaque octet (Figure 4-6). Pour pouvoir insérer de nouvelles instructions, ce buffer doit impérativement avoir un minimum de 8 entrées vides. S'il s'agit d'instructions consécutives à un saut, le buffer d'instruction doit être, soit complètement vide, soit le nombre d'instructions consommées doit être égal au nombre d'entrées valides.

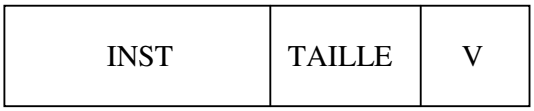


Figure 4-6 - Entrée du buffer d'instruction.

Le mécanisme de stockage des instructions, ainsi que celui de gestion du PC correspondant à l'adresse du premier octet présent dans le buffer d'instructions sont décrit par la Figure 4-7.

Chaque réponse du cache instructions transfère 64 bits d'instructions vers le module FETCH. En cas de réponse désordonnée, le registre INS_TEMP permet de stocker ses 64 bits d'instructions en attendant la réponse à la requête antérieure.

Les instructions reçues sont décalées en utilisant les poids faibles de l'adresse de requête appropriée (RSP_AD0 et RSP_AD1) afin d'éliminer les octets ne correspondant pas à la requête. Le bit de validité des instructions reçues est obtenu de la même façon avec, pour les modèles dotés de prédicteur de branchement, une invalidation des octets consécutive à un branchement pris (RSP_SHIFT0 et RSP_SHIFT1 sauvegarde JUMP_SHIFT).

Un décodeur détermine une taille associée à chaque octet correspondant à la taille de l'instruction potentielle.

Un second décaleur permet de décaler les entrées contenues dans le buffer d'instructions du nombre d'octets consommés par le module DECODE et d'y insérer les nouveaux octets chargés. Le premier octet contenu dans le buffer correspond toujours à une instruction. Le nombre maximum d'octets consommés par cycle est de 7.

Huit octets sont présentés au module DECODE, ainsi que leurs tailles et leurs bits de validité.

Le registre INS_PC contient l'adresse du premier octet contenu dans le buffer instruction au cycle précédent. Il est obtenu en additionnant les registres INS_SHIFT (nombre d'instructions consommées par le module DECODE au cycle précédent) et le registre TEMP_PC qui contient, soit la même valeur qu'INS_PC, soit l'adresse de saut lors d'un branchement.

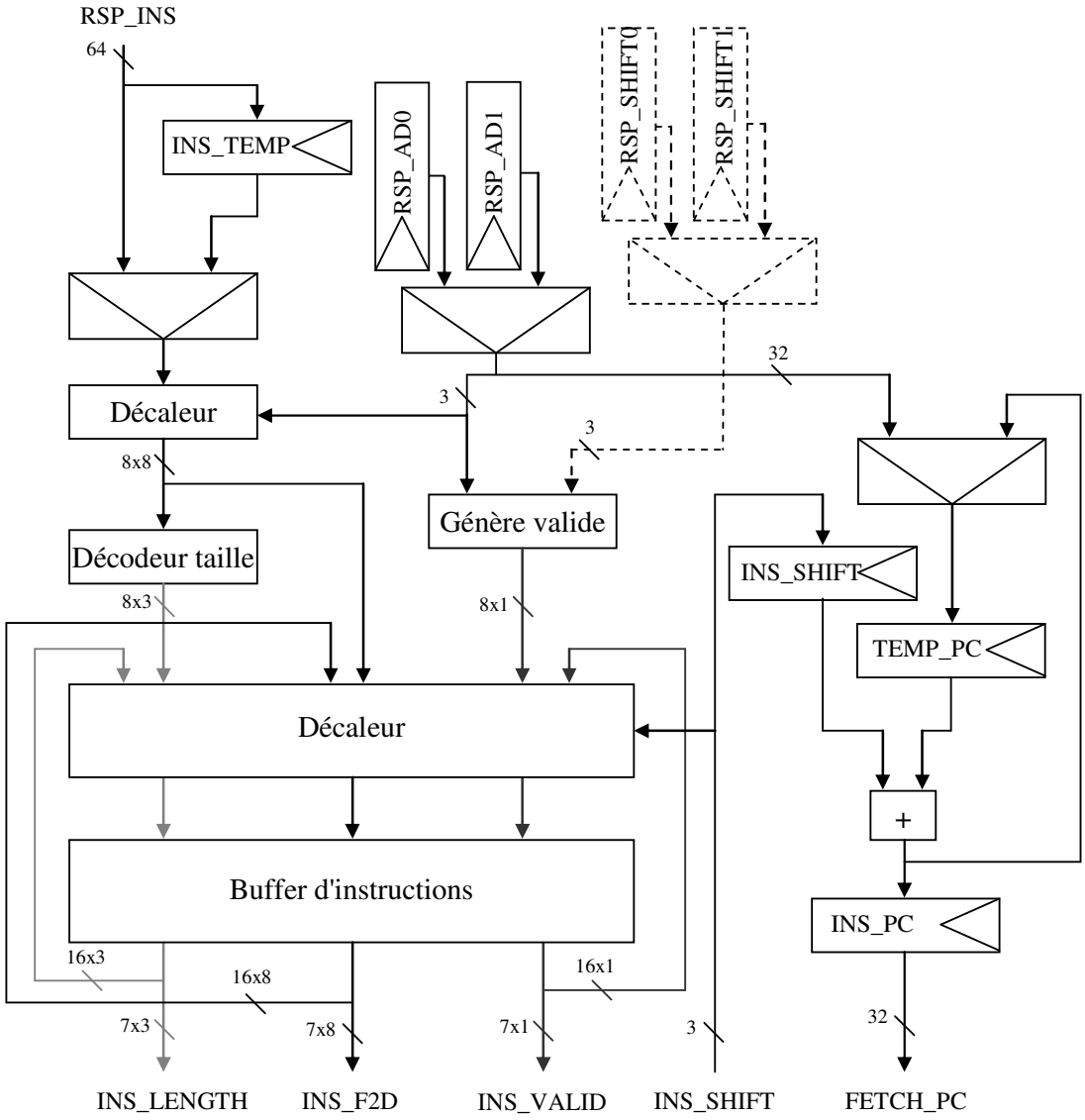


Figure 4-7 - Mécanisme de gestion de réception et de stockage des instructions du module Fetch.

4.3 - Le module de décodage des instructions DECODE

Le module de décodage décode et regroupe les instructions présentées par le module FETCH. Il commence par déterminer le nombre d'octets consommés par les regroupements potentiels d'une à quatre instructions. Pour ce faire, il additionne, pour chaque octet, la taille de l'instruction potentielle avec sa position dans le buffer d'instructions à l'aide d'additionneur 3-bits. Les tailles accumulées des regroupements d'instructions sont obtenues grâce à des multiplexeurs utilisant le résultat du groupement d'instructions inférieur (Figure 4-8). Le

nombre total d'octets consommés est obtenu grâce à un multiplexeur utilisant le nombre d'instructions regroupées.

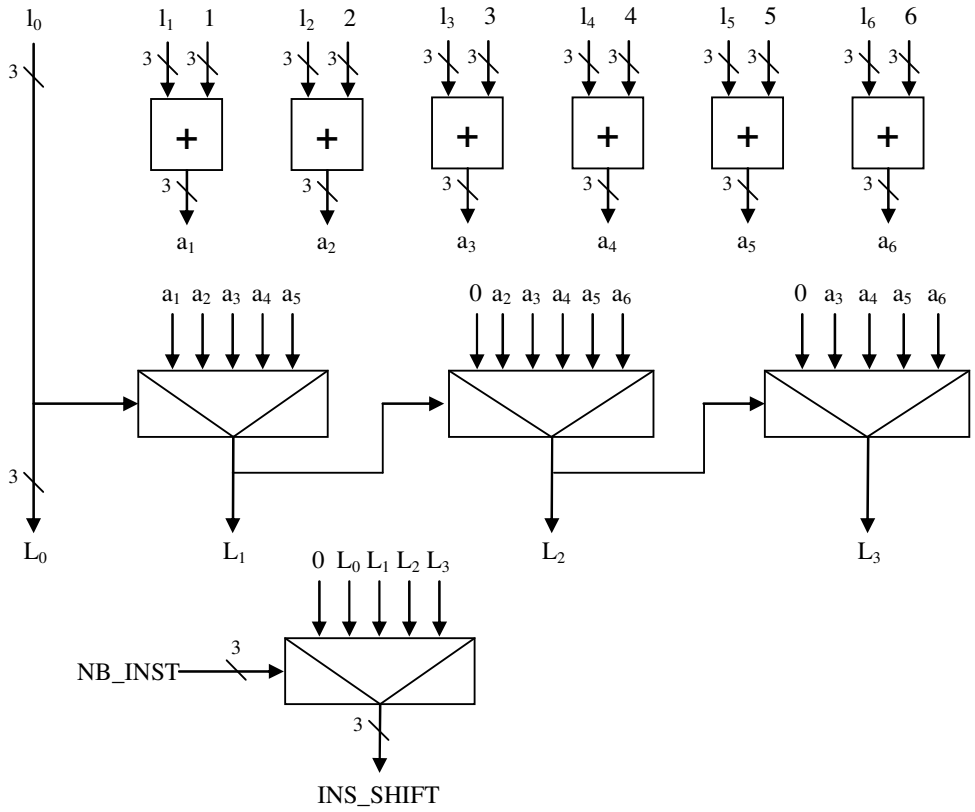


Figure 4-8 - Décodage de la taille des instructions et du nombre d'octets consommés.

Le type des instructions est obtenu de façon similaire, chaque couple d'octets d'instructions potentielles (le mnémonique d'une instruction pouvant être sur un ou deux octets) est typé et des multiplexeurs utilisant le résultat du mécanisme de décodage de tailles accumulées des instructions permet de connaître les types associés à chaque instruction (Figure 4-9).

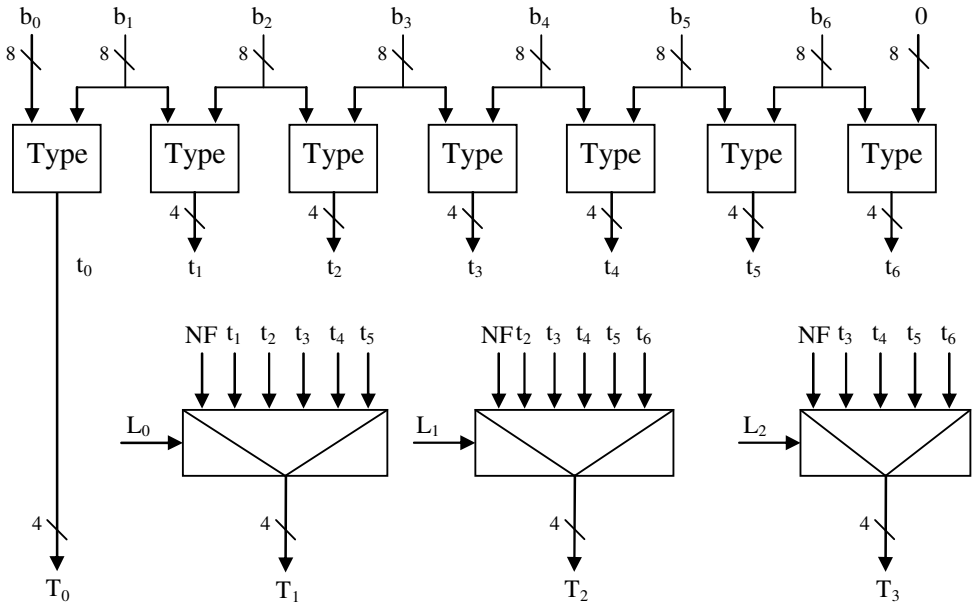


Figure 4-9 - Typage des instructions.

La validité de chaque instruction est également obtenue grâce à des multiplexeurs utilisant la taille accumulée des instructions. Le bit de validité correspondant permet de savoir si une instruction est entière, donc valide (Figure 4-10).

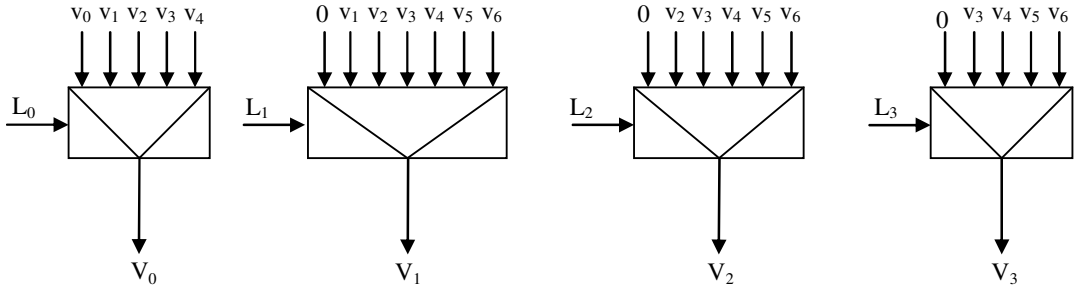


Figure 4-10 - Décodage de la validité des instructions.

Les types des instructions, ainsi que leurs validités, sont utilisés pour regrouper les instructions (Figure 4-11). Les types des instructions et les groupements associés des modèles JMS et JMQ sont différents et ont été défini dans le chapitre précédent (3.2.c - , 3.4.c -). Les instructions ne sont pas modifiées par l'unité de regroupement, celle-ci associe un numéro de groupe aux instructions et un nombre d'instructions consommées par ce groupe.

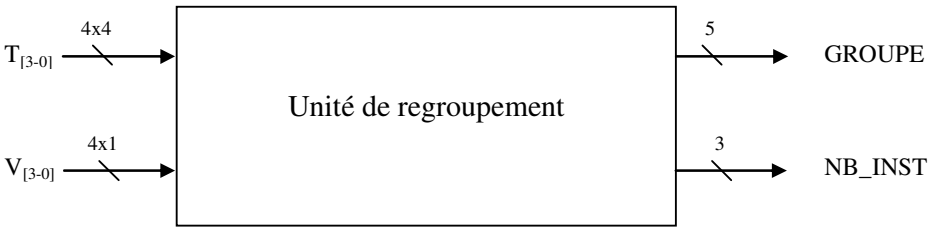


Figure 4-11 - Décodage du groupe d'instructions.

Les instructions de types LV et MEM (LV2 et MEM2 également pour la JMS) doivent être décodé.

Le décodeur RS1, décode le premier octet qui peut être de tous types. Si l'instruction est de type LV ou MEM, le décodeur détermine s'il s'agit d'une constante, d'une lecture ou d'une écriture de variable locale (détermine l'index par rapport au pointeur VARS). Pour toutes les instructions, il détermine la taille de la donnée sur laquelle l'instruction travaille.

Les décodeurs RS2 décodent les lectures de variable locale (LV). Le modèle JMS en possède trois car la seconde instruction de type LV (ou LV2) ne peut se trouver qu'entre le deuxième et le quatrième octet compris. Le modèle JMQ en possède six car ces mêmes instructions peuvent être entre le premier octet (déjà décodé par RS1) et le septième.

Les décodeurs RSD décodent les écritures de variable locale. Ils sont au nombre de sept car chaque octet est potentiellement de type MEM.

Les groupes effectués par l'unité de regroupement et la taille accumulée des instructions permettent d'écrire, dans les registres correspondants, les instructions destinées aux différentes parties matérielles du pipeline, ainsi que les informations ajoutées par les décodeurs RS1, RS2 et RSD (Figure 4-12).

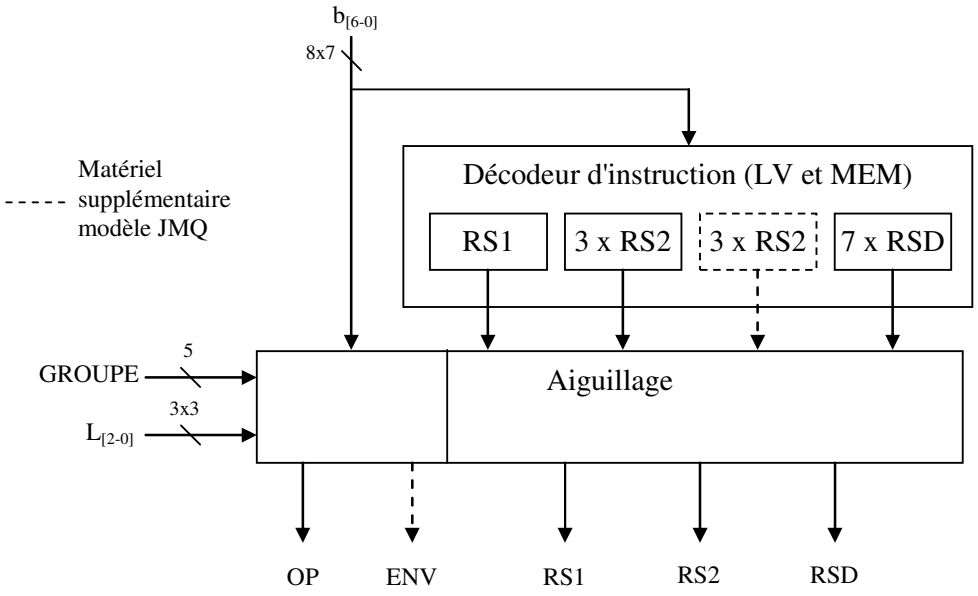


Figure 4-12 - Décodage des instructions et aiguillage en fonction du regroupement.

4.4 - Le module REGISTER (JMS)

Comme indiqué au chapitre précédent le module Register contient le sommet de la pile contenue dans une file matérielle de 64 registres 32-bit. Il est responsable de tous les accès à la pile.

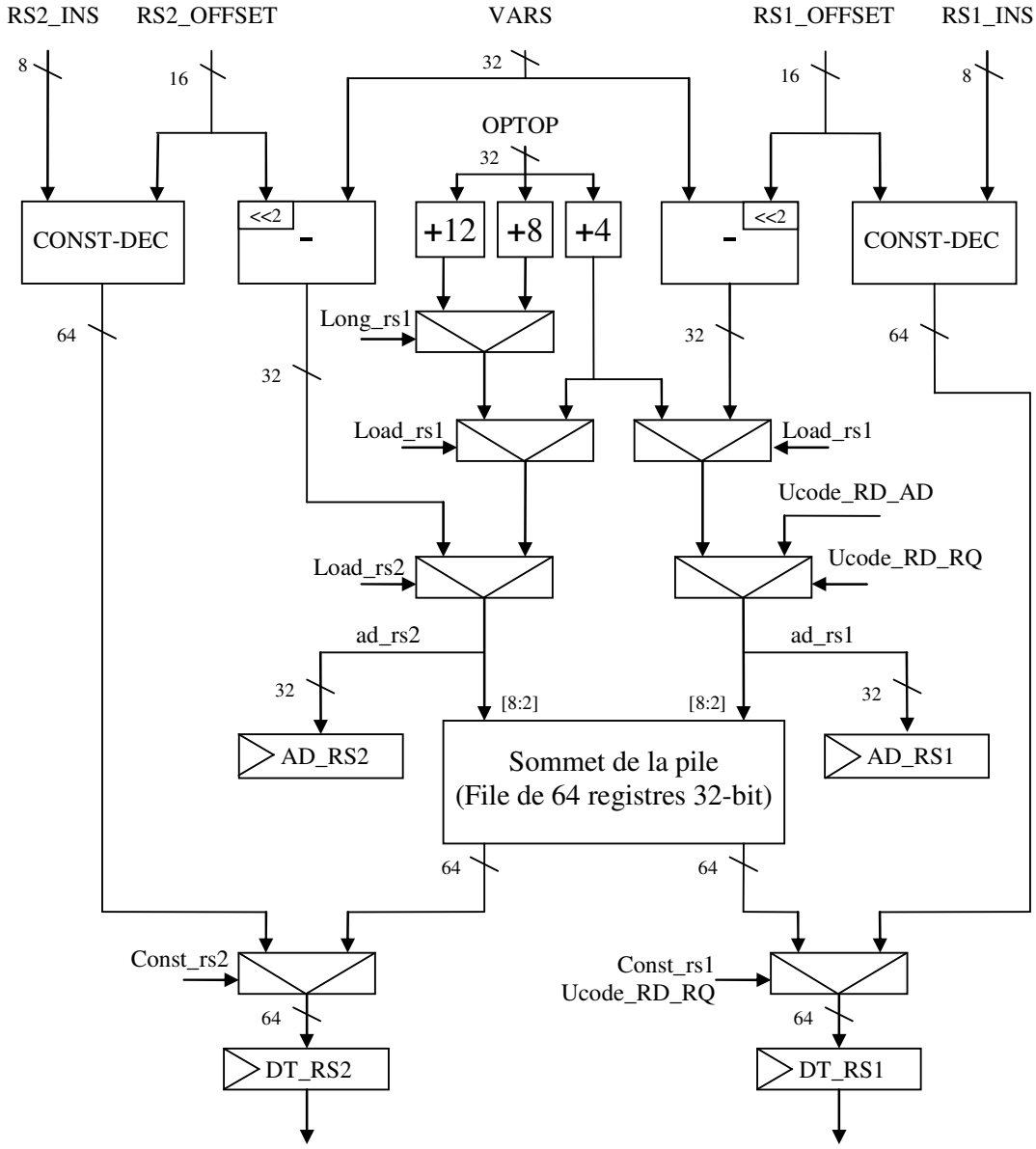


Figure 4-13 - Chargement des données dans le pipeline (JMS).

La première fonction de ce module est d'exécuter les instructions de chargement de données dans le pipeline. Il utilise les informations présentées par le module de décodage afin de présenter les données correspondantes au module EX_C. La Figure 4-13 montre le fonctionnement de ce mécanisme. Pour chaque donnée à insérer dans le pipeline, le module REGISTER décode simultanément les adresses de sommet de la pile (à partir du registre OPTOP), les adresses d'accès aux variables locales (à partir du registre VARS auquel on soustrait l'indexe de la variable locale), ainsi que la constante potentielle à insérer. Une série

de multiplexeurs permet de choisir la donnée à accéder dans la pile pour l'insérer dans le pipeline. L'adresse de la donnée RS1 à extraire de la pile peut être également fournie par le module EX_C lors d'instructions exécutées en plusieurs cycles et ayant besoin d'éléments contenus dans la pile. Le module REGISTER vérifie également que l'adresse de lecture de la donnée ne correspond, ni à une donnée à écrire présente dans le pipeline, ni à une adresse absente de la pile matérielle (Figure 4-16).

La seconde fonction de ce module est d'exécuter les instructions de rangement des données dans la pile. Le module REGISTER décode l'adresse de rangement du résultat du groupe d'instruction qui peut être, soit le nouveau sommet de la pile (newOPTOP), soit l'adresse d'une variable locale (à partir du registre VARS). Cette adresse est propagée dans le pipe-line par une série de registres constituant le mécanisme de by-pass (Figure 4-16). Un mécanisme propage également l'autorisation d'écriture de la pile et vérifie que l'adresse d'écriture appartient à l'espace adressable contenu dans la file matérielle (Figure 4-14, Figure 4-15). Le module EX_C peut générer des écritures sur la pile, celles-ci sont propagées de la même façon.

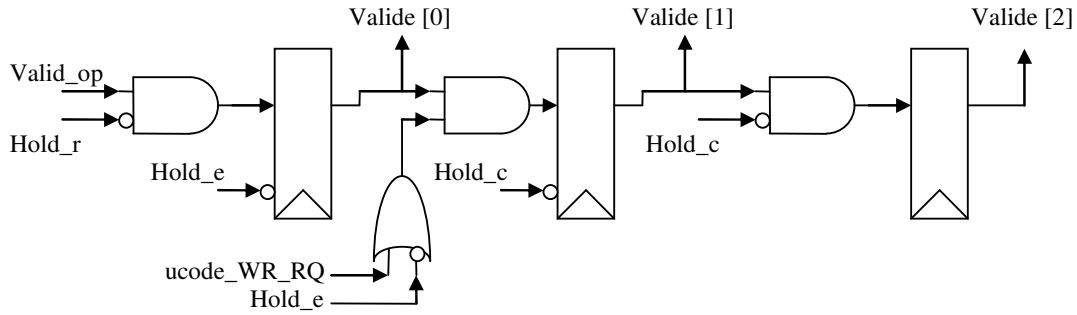


Figure 4-14 - Mécanisme de validité de l'instruction de chaque étage du pipeline.

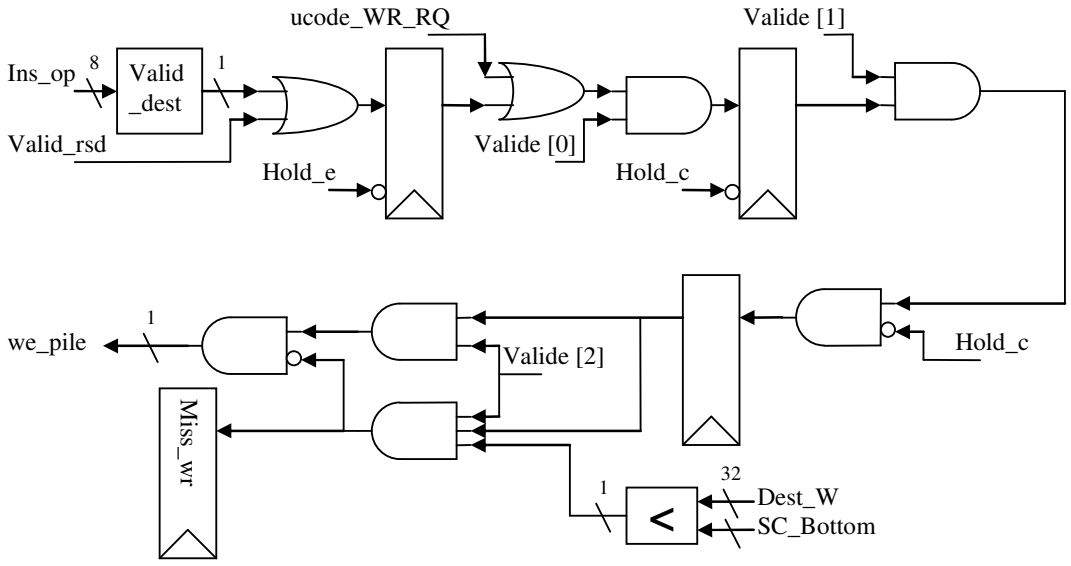


Figure 4-15 - Mécanisme d'autorisation d'écriture de la pile (JMS).

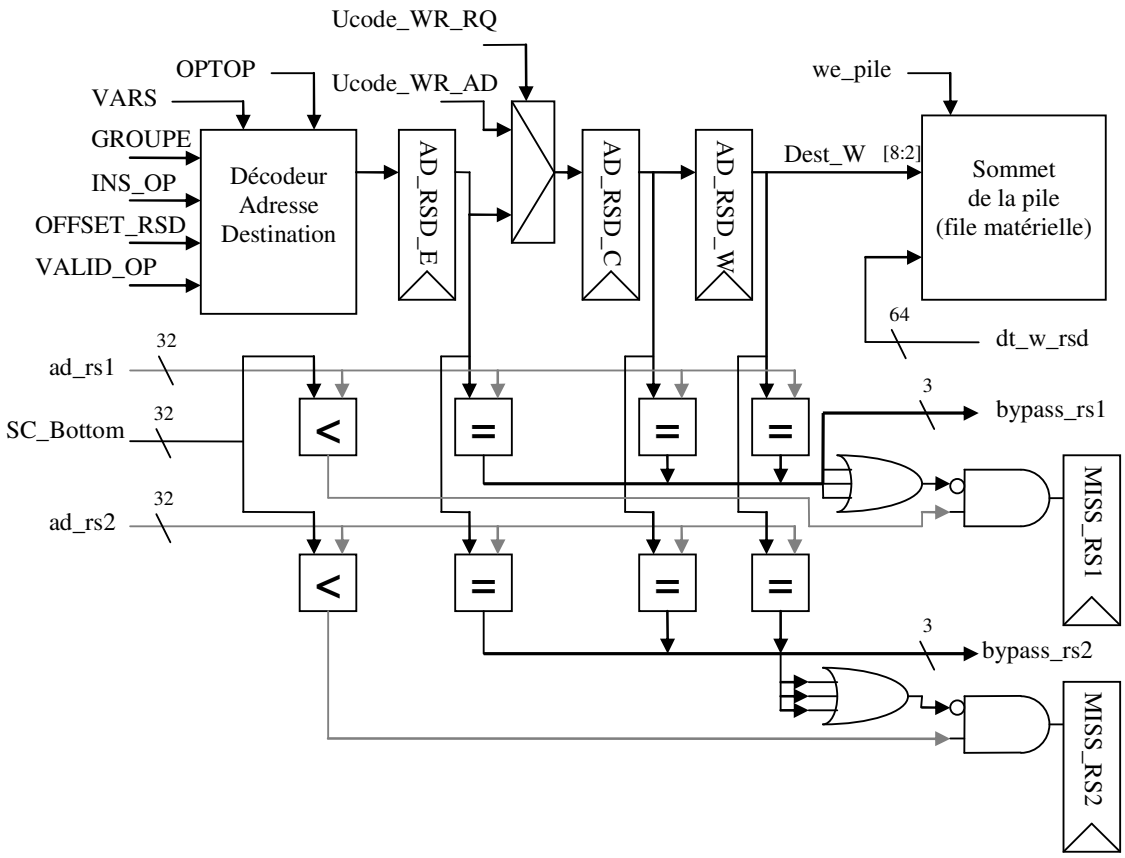


Figure 4-16 - Mécanismes d'écriture de la pile, de by-pass et de miss de lecture (JMS).

Le module REGISTER effectue le dribbling de la pile. Un automate vérifie le niveau de remplissage de la pile matérielle et ajuste ce niveau en générant des lectures et des écritures mémoire (cf : 4.6.d -). Cet automate modifie l'adresse de fond de pile matérielle (SC_BOTTOM).

4.5 - Le module EX_C (JMS)

Le module EX_C effectue toutes les instructions n'appartenant ni au type LV (LV2), ni au type MEM (MEM2). L'automate EX_FSM reçoit les instructions du module DECODE et pilote le matériel pour les exécuter. Il pilote également l'automate C_FSM qui est chargé des accès au cache de données. Le choix des opérandes à utiliser est effectué grâce aux informations sur les by-pass donnés par le module REGISTER. En cas de MISS de lecture de la pile, le cache de données renvoie la donnée qui est enregistrée dans le registre "Hold" correspondant à l'opérande manquant. Ces registres gardent également les valeurs des opérandes lors des cycles de gel de l'étage EXECUTE (Figure 3-1).

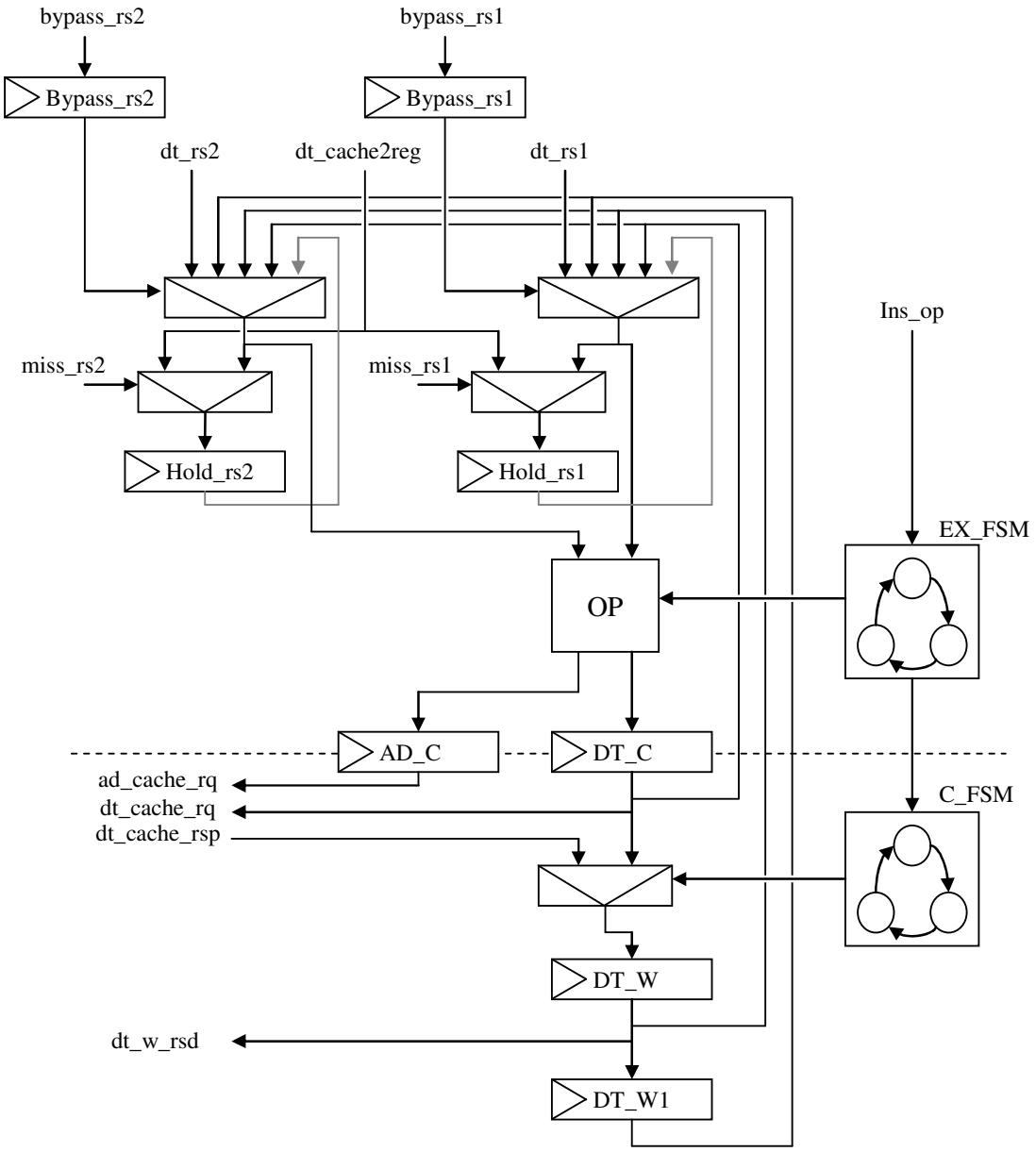


Figure 4-17 - Architecture du module EX_C.

Le bloc OP est constitué des éléments suivants :

- Une Unité Arithmétique et Logique 32-bit (ALU).
- Une Unité de calcul Flottant 32-bit (FPU).
- Un additionneur/soustracteur 32-bit supplémentaire.
- Un décaleur32-bit.

Lors des cycles de gel de l'étage EXECUTE, La FPU n'est pas gelée pour les opérations nécessitant plusieurs cycles. L'étage EXECUTE contient les registres internes du processeur (PC, VARS, FRAME, OPTOP, etc.), les instructions d'accès aux registres ne peuvent pas être groupées.

Le registre DT_C reçoit les données, tandis que le registre AD_C reçoit les adresses d'accès au cache, ces deux registres marquent la frontière entre les étages EXECUTE et CACHE du pipeline. L'automate C_FSM gère les accès au cache de données et pilote le multiplexeur permettant de choisir entre une donnée émise, soit par l'étage précédent, soit par le cache de données. Cette donnée est écrite dans le registre DT_W. Si les requêtes effectuées par l'automate C_FSM ne sont pas satisfaites, les étages en amont sont gelés.

La donnée contenue dans le registre DT_W est envoyée au module REGISTER afin d'être enregistrée dans la pile. Elle est également écrite dans le registre DT_W1 qui permet de sauvegarder cette valeur au cas où elle serait accédée en lecture au même cycle. En effet, le module REGISTER ne peut pas enregistrer dans les registres DT_ RS1 et DT_RS2 une donnée provenant du module EX_C (Figure 4-13).

Les modèles JMS et JMS_PRED implémentent des versions différentes de ce module car ils gèrent différemment les sauts.

Pour le modèle JMS, l'automate EX_FSM gère les branchements, il génère l'adresse de saut et demande au module FETCH de sauter, le cas échéant, après résolution des branchements (pour les branchements conditionnels).

Pour le modèle JMS_PRED, l'automate EX_FSM envoie toutes les informations relatives aux sauts au module de prédiction de branchement (type de branchement, adresse de saut, adresse instruction suivante et résultat de la résolution (branchement conditionnel)) qui effectue le saut s'il n'a pas été prédit ou mal prédit. De plus, une pile de huit entrées 32-bit garde les adresses de retour de méthode.

4.6 - Le module PREP (JMQ)

Comme indiqué au chapitre précédent, le module PREP contient plusieurs étages du pipe-line et est responsable de beaucoup de mécanismes essentiels au fonctionnement de la machine JMQ. Ces mécanismes sont, l'aiguillage des instructions vers les étages suivants du pipe-line, la gestion des accès à la pile d'environnement (extraction et écriture des variables locales, dribbling) et la gestion de l'environnement.

4.6.a - Aiguillage des instructions

L'étage REGISTER_SWITCH est responsable de l'aiguillage des instructions vers les autres étages du pipe-line. Il effectue cette opération grâce aux informations décodées par le module DECODE qui présente les instructions, ainsi que des informations complémentaires, sur des interfaces différentes en fonction de leurs types (Figure 4-12).

Les instructions de type LV sont aiguillées vers la FIFO de chargement. Toutes les instructions qui ne corresponde pas à l'insertion d'une constante dans la pile sont remplacées par l'instruction iload et un bit donnant la longueur de la donnée à charger (Figure 4-18). La donnée (variable locale ou registre) sera transmise au module CHARG par l'intermédiaire de la file de dépendance de lecture. S'il n'y a pas de seconde instruction de chargement valide (RS2), celle-ci est remplacée par le mnémonique nop (pas d'opération).

L'écriture d'une nouvelle entrée dans la FIFO d'instruction de chargement est commandée par le bit de validité de l'instruction de chargement RS1 et le signal Hold_r (gel de l'étage REGISTER-SWITCH).

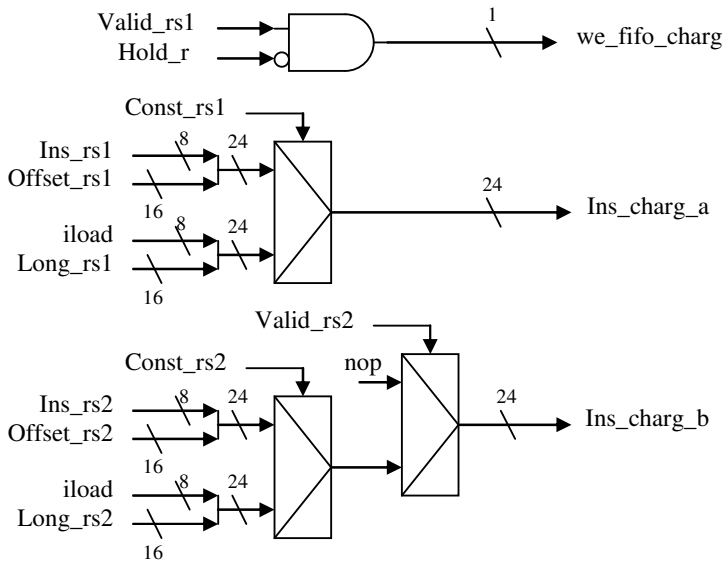


Figure 4-18 - Aiguillage des instructions destinées au module de chargement.

Une entrée de la FIFO d'instruction de chargement est constituée de deux instructions de 24-bits chacune (Figure 4-19).

Figure 4-19 - Format d'une entrée de la FIFO d'instruction de chargement.

Les instructions de type OP, BG et MEM sont aiguillées vers la FIFO d'instruction de calcul (Figure 4-20). Ce mécanisme est plus compliqué que celui des instructions de chargement car les instructions de type ENV influent sur ce qui est envoyé au module EXEC. En effet, les instructions d'appel de méthode demandent le déplacement de leurs arguments de la file d'exécution vers la pile d'environnement (3.4.b - L'appel de méthode), ce déplacement est effectué grâce à la génération d'écriture par l'automate Write_FSM. Le module EXEC doit également indiquer au module PREP lorsqu'il rencontre une instruction de retour de méthode ou de sous-programme (3.4.b - Le retour de méthode), ces instructions lui sont donc transmises sous la forme d'un bit (R).

L'instruction transmise à la FIFO d'instruction de calcul est remplacée par rstore (instruction interne ne faisant pas partie du Bytecode) en cas de génération d'écriture ou par nop s'il n'y a pas d'instruction de type OP ou BG valide.

L'écriture d'une nouvelle entrée dans la FIFO d'instruction de calcul est commandée par le signal we_fifo_calc.

Le calcul du nombre de chargement effectué entre deux opérations est calculé, le résultat est transmis au module EXEC par la FIFO d'instructions de calcul.

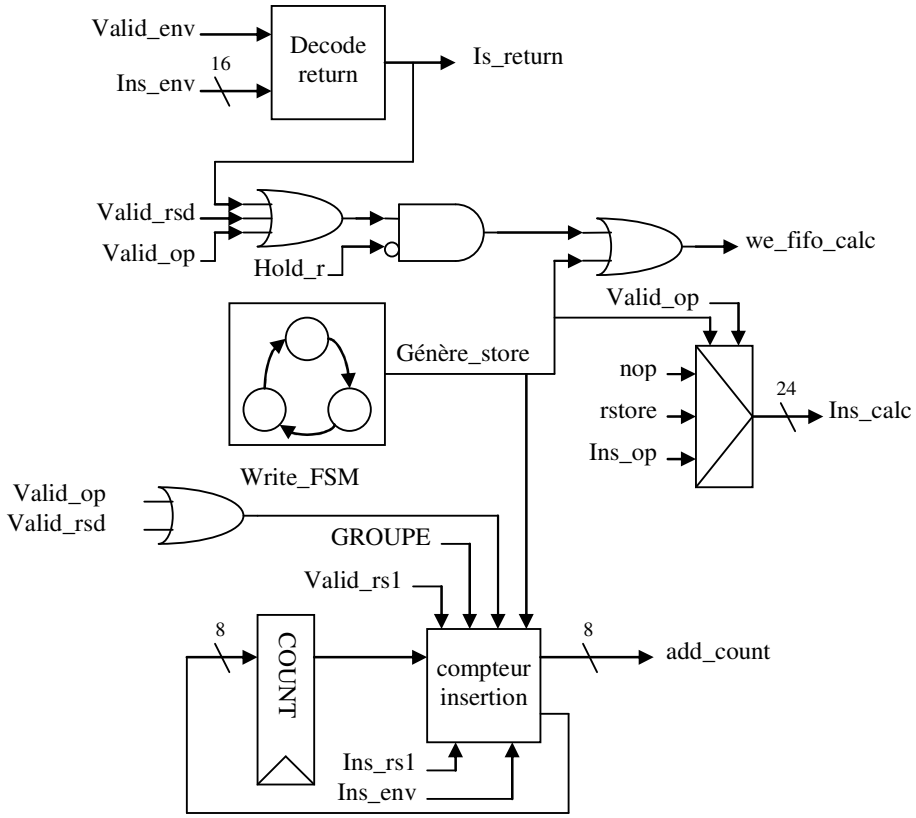


Figure 4-20 - Aiguillage et construction de l'instruction destinée au module EXEC.

La Figure 4-21 montre le format d'une entrée de la FIFO d'instruction de calcul, elle est constituée des éléments suivants :

- R : Ce bit indique une instruction de retour (Is_return).
- M : Ce bit indique une instruction de type MEM (Valid_rsd).
- Ins_calc : Instruction à exécuter (24 bits).
- Add_count : Indique le nombre de chargement entre deux opérandes (8 bits).

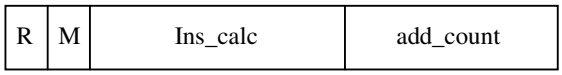


Figure 4-21 - Format d'une entrée de la FIFO d'instruction de calcul.

Les instructions destinées à l'étage ENV du pipe-line lui sont transmises sans modification.

4.6.b - Extraction et écriture de données dans la pile d'environnement

Le module PREP effectue les accès à la pile d'environnement et contient le mécanisme de by-pass.

Le mécanisme d'extraction des données de la pile (Figure 4-22) est simplifié par rapport à celui de la JMS car la pile matérielle ne contient plus la pile d'exécution, il n'y a donc plus d'accès au sommet de la pile mais uniquement des accès aux variables locales à partir du registre VARS. En cas de gel de l'étage REGISTER-SWITCH ou si l'instruction RS2 ne correspond pas à une lecture, l'automate *Posetd_pile_FSM* peut utiliser le second accès en

lecture pour lire un registre d'environnement contenu dans la pile (recopie, en tache de fond, d'un ancien environnement dans le banc de registres *Old*).

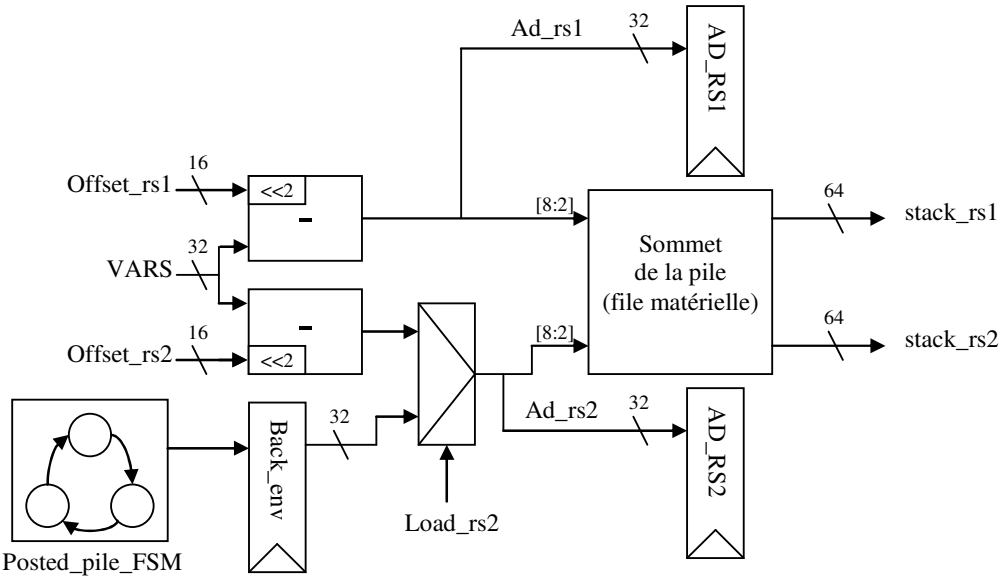


Figure 4-22 - Extraction des variables locales (JMQ).

La Figure 4-23 montre le mécanisme de choix des données à insérer dans la file de dépendance de lecture..

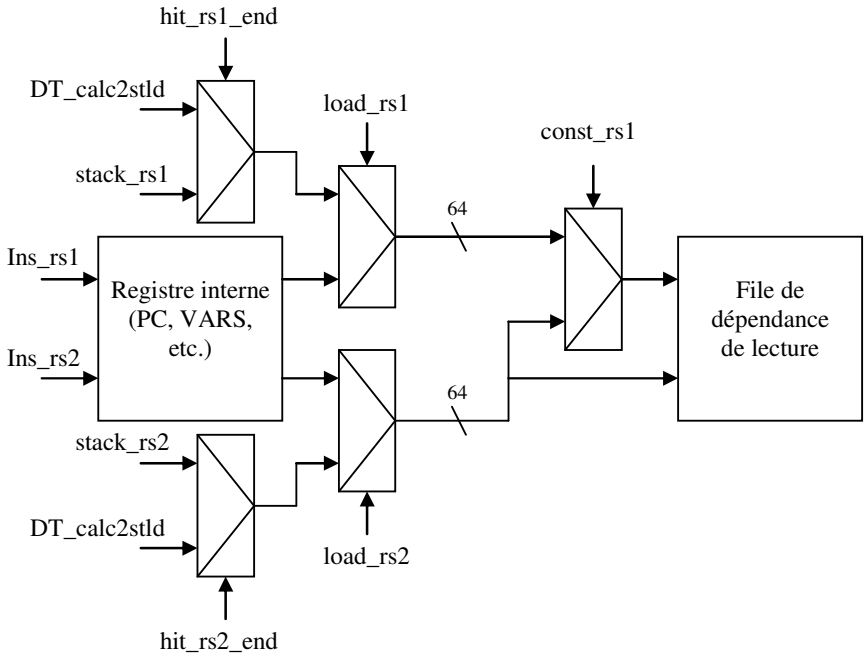


Figure 4-23 - Insertion des données dans la file de dépendance de lecture.

Ces données peuvent être, soit extraites de la pile d'environnement, soit la valeur de l'un des registres internes du processeur. Pour chaque opérande, un multiplexeur permet de substituer la donnée écrite par le module EXEC à la donnée extraite de la pile si l'adresse de

lecture correspond à l'écriture effectuée (signal hit_rsx_end). La file de dépendance de lecture accepte jusqu'à deux nouvelles entrées par cycleLes données insérées dans la file de dépendance de lecture peuvent ne pas être valide. En effet, l'adresse de la donnée insérée peut se trouver dans la file de dépendance d'écriture ou peut ne pas être contenue dans la partie matérielle de la pile d'environnement. Une entrée de la file de dépendance de lecture est donc composée des champs suivants (Figure 4-24) :

- FLAG : Indique l'état de la donnée (2 bits).
- AD_W : Adresse de la donnée dans la file de dépendance d'écriture (8 bits).
- Donnée : Contient la donnée (64 bits).

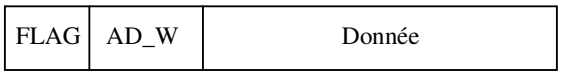


Figure 4-24 - Format d'une entrée de la file de dépendance de lecture.

L'état de la donnée est, soit 'miss' lorsque l'adresse de la données n'est pas contenue ni dans la pile matérielle ni dans la file de dépendance d'écriture, soit 'valide' (1), soit 'à recopier' (2) lorsque l'adresse se trouve dans la file de dépendance d'écriture. Le signal *hit_rsx* indique que l'adresse de lecture est présente dans la file de dépendance d'écriture tandis que hit_AD_rsx donne l'adresse de l'entrée correspondante (Figure 4-25).

De même que pour les adresses de lecture de donnée dans la pile, le décodage des adresses d'écriture est simplifié car il ne gère que les écritures de variables locales (à partir du registre VARS) (Figure 4-25). Une nouvelle adresse d'écriture peut être insérée dans la file de dépendance d'écriture à chaque cycle. Cette file vérifie si celle-ci est déjà présente.

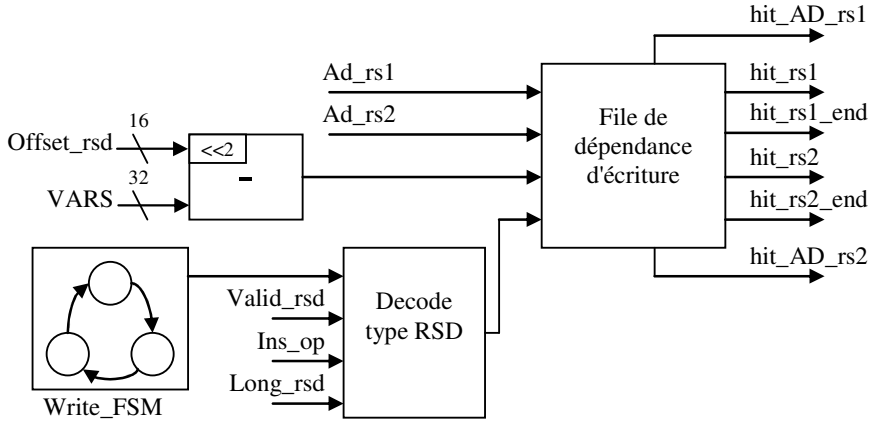


Figure 4-25 - Insertion dans la file de dépendance d'écriture et vérification des lectures.

Outre les écritures de variables locales provoquées par les instructions de types MEM, la file de dépendance d'écriture gère également les d'écritures générées, lors de l'appel d'une méthode (déplacement des arguments de la file d'exécution vers la pile d'environnement), par l'automate *Write_FSM* et les écritures de registre. La file de dépendance d'écriture garde donc les informations suivantes (Figure 4-26) :

- V : Indique que l'adresse peut être utilisée pour les by-pass (1 bit).
- FLAG : Indique le type de l'écriture (2 bits).
- Adresse : Adresse d'écriture pour une variable locale (32 bits).

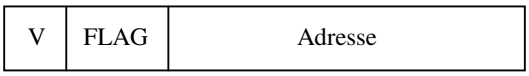


Figure 4-26 - Format d'un entrée de la file de dépendance d'écriture.

Le champ FLAG représente les types d'écriture suivants :

- Data32 : écriture d'une variable locale 32-bits.
- Data64 : écriture d'une variable locale 64-bits.
- DataREG : écriture d'une donnée 32-bits attendu par le module PREP (ex : registre interne).
- DataACC : écriture d'une variable locale 32-bits attendu par le module PREP (ex : référence d'un objet).

Le bit 'V' n'est mis à '1' que pour les écritures de variables locales. Si une adresse à insérer dans la file de dépendance d'écriture y est déjà présente, Le bit 'V' de l'entrée correspondante est mis à '0' et celui de la nouvelle entrée est mis à '1'. Ainsi lors d'une lecture à la même adresse, seule la dernière écriture est considérée. La file de dépendance d'écriture envoie alors l'adresse de l'entrée correspondante à la file de dépendance de lecture qui l'enregistre dans le champ AD_W de la nouvelle entrée.

La Figure 4-27 montre le mécanisme de by-pass entre les files de dépendances lors des lectures par le module CHARG et les écritures par le module EXEC.

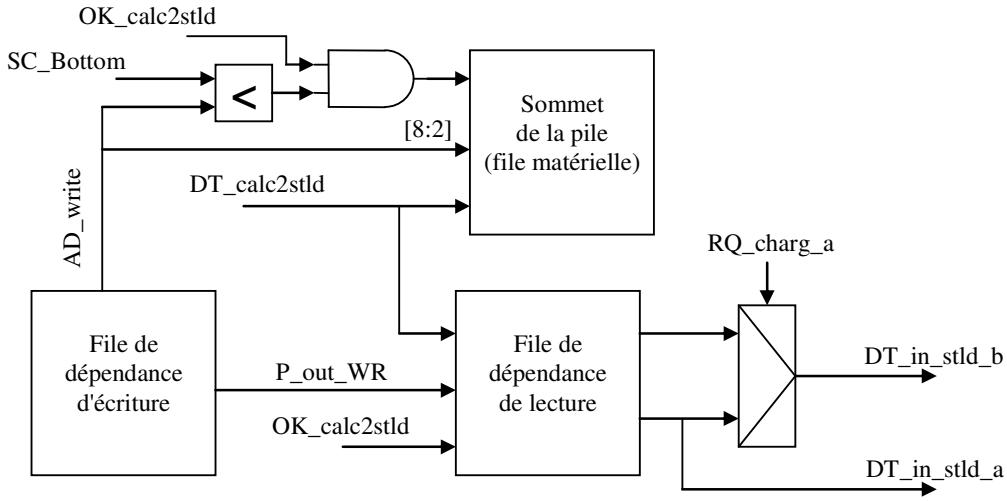


Figure 4-27 - Mécanisme d'écriture de la pile et by-pass (JMQ).

Lors d'une écriture par le module EXEC, si la donnée correspond à une variable locale, la file de dépendance d'écriture donne l'adresse d'écriture dans la pile d'environnement qui doit recevoir la donnée. En cas de 'miss' de la pile matérielle, la donnée est enregistrée dans un registre pour être écrite en mémoire au cycle suivant, se qui permet de ne pas geler le module EXEC. Cependant, celui-ci sera tout de même gelé s'il essaie de faire une nouvelle écriture de variable locale avant l'écriture en mémoire de la donnée incriminée. Parallèlement à l'écriture dans la pile d'environnement, l'adresse de l'entrée correspondante de la file de dépendance de lecture est transmise à la file de dépendance de lecture qui écrit la donnée dans

toutes les entrées dont le champ AD_WR correspond à l'entrée dans la file de dépendance d'écriture et dont le champ FLAG indique 'à recopier' (l'entrée devient 'valide' FLAG=1). Si la donnée correspond à une entrée lue par le module CHARG, celle-ci lui est transférée.La file de dépendance de lecture présente les deux plus anciennes entrées au module CHARG. Ces données sont transmises au module CHARG par les ports DT_in_stld_a et DT_in_stld_b. En fonction d'une demande de lecture par l'automate *A_FSM* du module CHARG, le port DT_in_stld_b présente, soit la donnée la plus ancienne, soit la seconde donnée la plus ancienne.

4.6.c - Gestion de l'environnement (étage ENV)

Le module PREP est responsable de la gestion de l'environnement effectuée dans l'étage ENV du pipeline. L'automate *Env_FSM* reçoit les instructions liées à l'environnement et effectue toutes les opérations nécessaires à leur exécution (appel et retour de méthode, prise du trap-handler, modification des registres internes, etc.). Il pilote les automates *Write_FSM*, *Posted_pile_FSM* et *Cache_FSM* (Figure 4-28). Il possède un additionneur 32-bits.

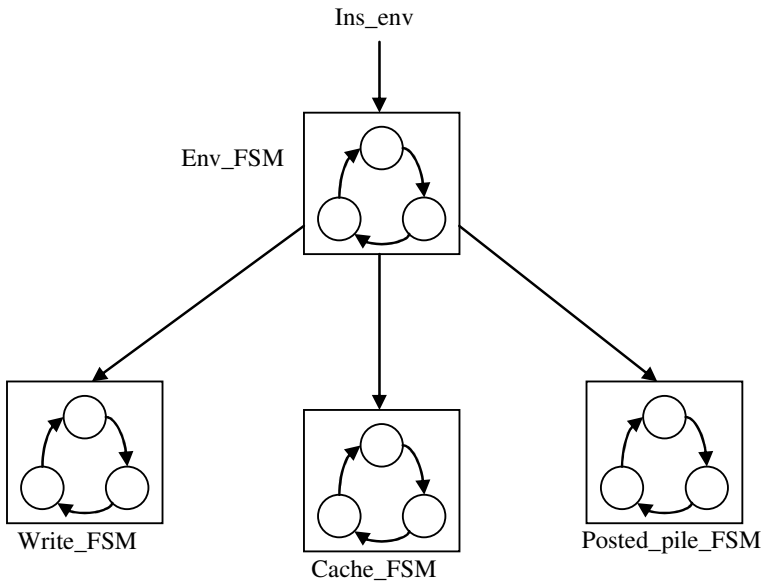


Figure 4-28 - Automates de gestion de l'environnement.

L'automate *Cache_FSM* effectue les accès au cache de données. L'automate *Write_FSM* gère le déplacement des arguments de la file d'exécution vers la pile d'environnement lors des appels de méthode. L'automate *Posted_pile_FSM* effectue la recopie des registre "old" dans la pile d'environnement après l'appel d'une méthode et recopie un environnement précédent depuis la pile vers les registre "old" pour préparer un éventuel second retour de méthode après un retour de méthode.

4.6.d - Le "dribbling"

Pour les trois processeurs (JMS, JMS_PRED et JMQ), le dribbling est assuré par l'automate *Dribbling_FSM* qui gère le nombre d'entrées dans la partie matérielle de la pile d'environnement. Cet automate effectue les accès au cache de données pour remplir et vider la file de 64 registres 32-bits.

Les états de l'automate sont les suivant :

- Idle : pas de changement.
- Fill : remplissage de la pile non-prioritaire.
- Spill : déversement de la pile en mémoire non-prioritaire.
- Underflow : remplissage de la pile prioritaire.
- Overflow : déversement de la pile en mémoire prioritaire.

L'état de l'automate est conditionné par le nombre d'entrées présentes dans pile d'environnement. Ce nombre est obtenu par la soustraction de l'adresse du fond de la pile (registre SC_Bottom) par celle de son sommet (registre OPTOP). Les limites de passage dans les états Fill et Spill sont définies dans le registre PSR (Tableau A-2). L'état Underflow est déclenché lorsque la pile contient moins de 6 entrées. L'état Overflow est déclenché lorsque la pile contient plus de 60 entrées.

4.6.e - Les accès au cache et à la pile d'environnement

Tous les mécanismes du module PREP se partagent un accès au cache de donnée, l'ordre de priorité d'accès au cache est le suivant :

- Le Dribbling si l'état est Underflow ou Overflow.
- Ecriture d'une variable locale.
- Lecture d'une variable locale.
- Lecture par l'étage ENV.
- Dribbling si l'état est Fill ou Spill.

La pile d'environnement possède deux ports de lecture destinés à la lecture des variables locales et à la lecture des environnements précédents par l'automate *Posted_pile_FSM* de façon non-prioritaire. Un port d'écriture est utilisé pour l'écriture des variables locales (incluant les écritures générées par l'automate *Write_FSM*), ainsi que l'écriture des anciens environnements par l'automate *Posted_pile_FSM* de façon non-prioritaire.

Le dribbling utilise un port en lecture et un port en écriture dédié à son fonctionnement.

4.7 - Le module de chargement CHARG (JMQ)

Ce module très simple est constitué de deux automates qui permettent chacun d'insérer une donnée à chaque cycle dans la file d'exécution. Deux générateurs de constantes et deux multiplexeurs sont pilotés par ces automates (Figure 4-29). En effet, chaque donnée à insérer est, soit une constante, soit une donnée présente dans la file de dépendance de lecture. L'automate B ne peut insérer une donnée dans la file que lorsque l'automate A est également capable de le faire. Si l'automate A insère une donné dans la file et que l'automate B ne le peut pas, l'instruction de l'automate B est transférée à l'automate A qui l'effectuera au cycle suivant.

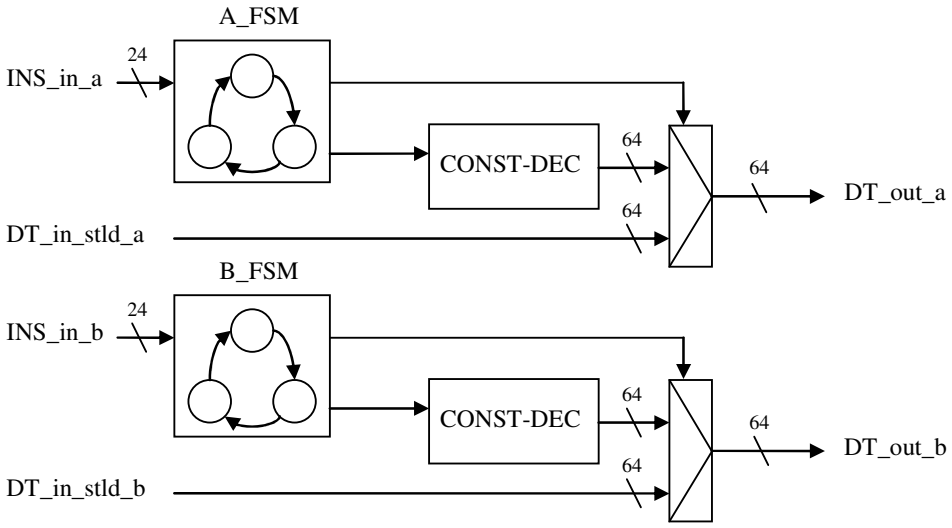


Figure 4-29 - module de chargement.

4.8 - L'unité de calcul EXEC (JMQ)

Le module EXEC effectue toutes les instructions mises à part les chargements au sommet de la pile (type LV) et les instructions uniquement liées à l'environnement (type ENV). Il reçoit les instructions et le nombre de variables chargées entre deux opérations par l'intermédiaire de la FIFO d'instructions de calcul. Il effectue les instructions sur les opérandes présentées par la file d'exécution.

Ce module est doté du même matériel de calcul que le module EX_C de la machine JMS, le bloc OP est constitué des éléments suivants :

- Une Unité Arithmétique et Logique 32-bit (ALU).
- Une Unité de calcul Flottant 32-bit (FPU).
- Un additionneur/soustracteur 32-bit supplémentaire.
- Un décaleur32-bit.

Le module EXEC demande au module FILE_EX, contenant la file d'exécution, de rechercher les opérandes correspondant à la prochaine opération à effectuer. Ce déplacement est directement lu dans la FIFO d'instruction de calcul ou bien généré par l'automate EX_FSM lors des instructions exécutées en plusieurs cycles (Figure 4-30).

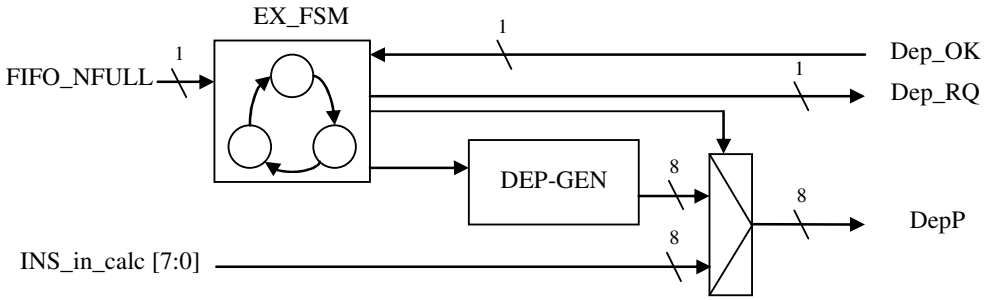


Figure 4-30 - Requête de recherche des opérandes.

Les bits 'R' et 'M' contenus dans la FIFO d'instruction de calcul sont sauvegardés par le module EXEC (Figure 4-31) afin d'informer le module PREP de l'exécution d'un retour de méthode (ou de sous-programme) (bit 'R') et/ou de l'exécution d'une écriture dans la pile d'environnement (bit 'M'). Le bit 'M' permet également d'invalider le bit de présence de la donnée pointée par P1 dans la file d'exécution.

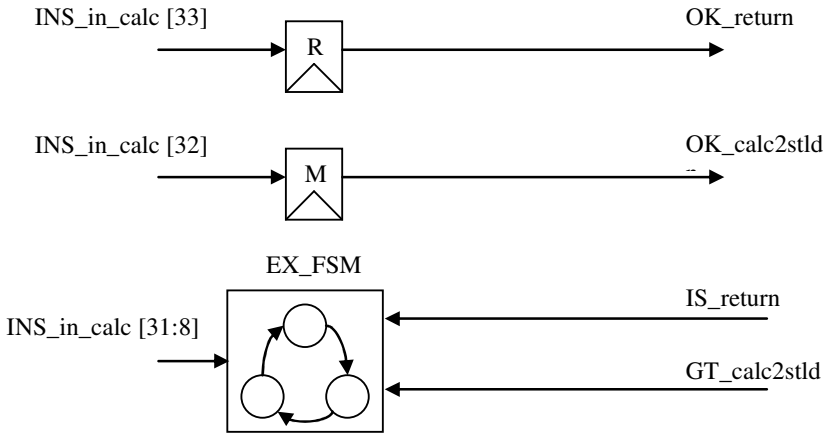


Figure 4-31 - Gestion des communication EXEC-PREP (instruction return et istore).

La Figure 4-32 montre le chemin de données du module EXEC. L'automate EX_FSM reçoit les instructions et pilote le matériel pour les exécuter. Il pilote également l'automate C_FSM qui est chargé des accès au cache de données.

Les données DT_in_P1 et DT_in_P2 proviennent des registres de la file d'exécution, les données DT_out_P1 et DT_out_P2 sont les résultats, de l'exécution de l'instruction courante, à écrire dans la file d'exécution. La donnée DT_calc2stld est destinée au module PREP pour être écrite dans la pile d'environnement (si M=1).

Le registre DT_C reçoit les données, tandis que le registre AD_C reçoit les adresses d'accès au cache, ces deux registres marquent la frontière entre les étages EXEC et CACHE du pipe-line. L'automate C_FSM gère les accès au cache de données et pilote les multiplexeurs choisissant les données à retourner à la file d'exécution et à la pile d'environnement. Lors d'une lecture du cache de données par l'étage CACHE qui nécessite une écriture, soit dans la file d'exécution, soit dans la pile d'environnement, l'étage EXEC est gelé. Lors d'une écriture non-satisfaite, l'étage EXEC n'est pas gelé tant qu'il n'y a pas d'autre instruction d'accès au cache.

La gestion des branchements est très simple dans ce module car il ne gère que deux types de saut. Les sauts conditionnels sont résolus, et le résultat est envoyé au module PREDIC. Pour les sauts inconditionnels, le module EXEC envoie le contenu du registre au module PREP par l'intermédiaire du signal DT_out_STLD.

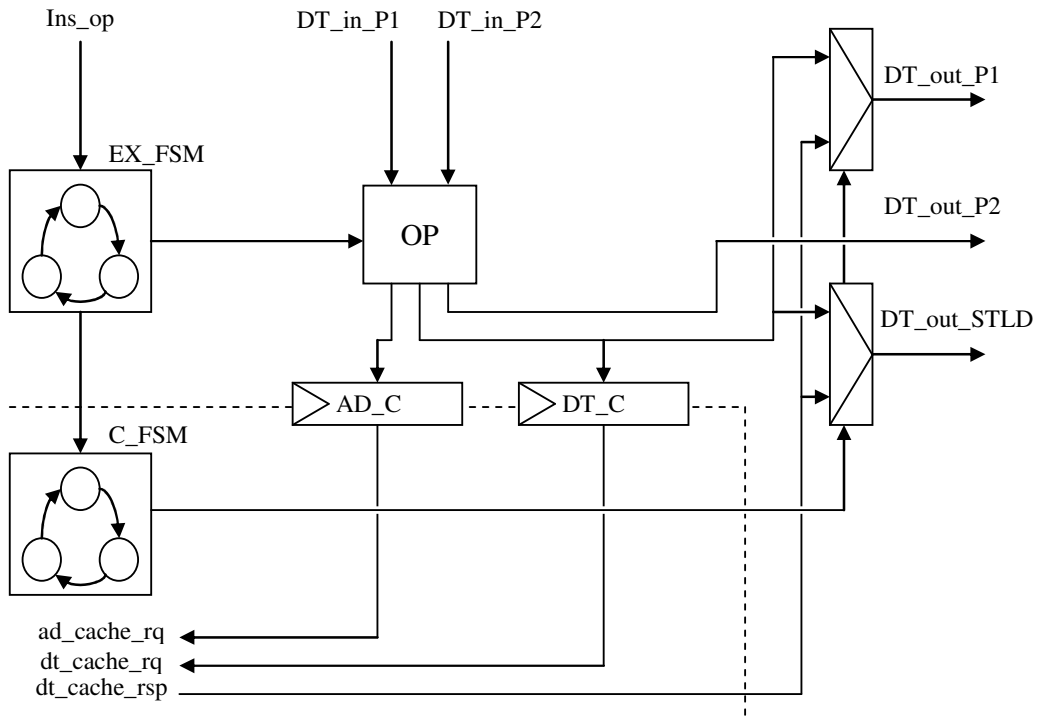


Figure 4-32 - Architecture du module EXEC.

4.9 - L'unité de prédiction de branchement PREDIC

Ce module est présent sur les modèle JMS_PRED et JMQ. Il permet de prédire dynamiquement la direction des sauts et ceci dès la requête de nouvelles instructions effectué par le module FETCH au cache instructions.

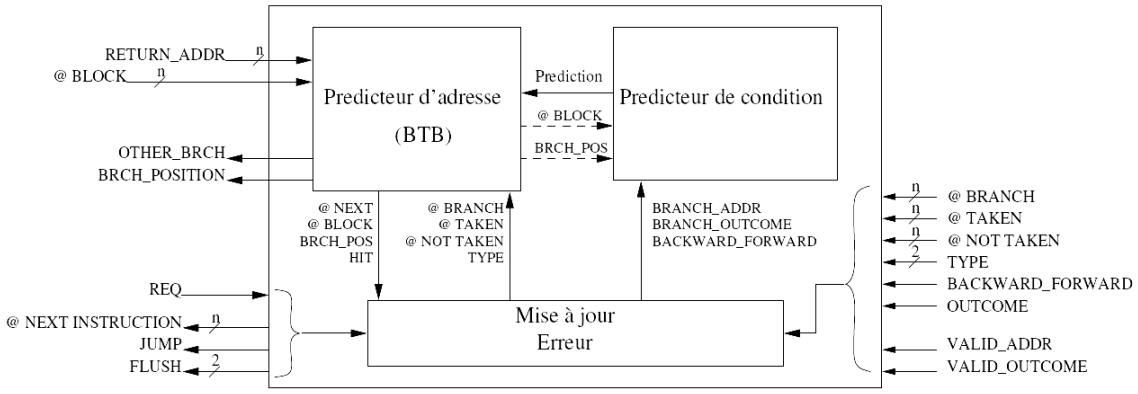


Figure 4-33 - Vue interne du préditeur de branchement.

L'unité de prédiction de branchement est constituée de trois modules (Figure 4-33 [49]) :

- Le prédicteur d'adresse : Contient les informations relatives aux branchements connus.
- Le prédicteur de condition : Contient le mécanisme de prédiction dynamique des branchements conditionnels.

Le contrôle de branchements : Effectue les mises à jours des autres modules, le suivit des branchements effectués mais non-résolus et le contrôle d'erreurs.

4.9.a - Le prédicteur d'adresses

Le prédicteur d'adresses est un cache associatif n-voies appelé Branch Target Buffer (BTB). Il contient les informations sur les branchements sauvegardées. Ces informations sont les suivantes (Figure 4-34) :

- V : Bit de validité d'une ligne du cache (1 bit).
- MRU : Bit pseudo-LRU (1 bit).
- TAG : Bit de poids fort de l'adresse du branchement (m bit).
- @TARGET : Adresse de saut du branchement (32 bit).
- T : Type du branchement (2 bits).
- O : Indique qu'il existe un second branchement dans la même requête de cache (1 bit).

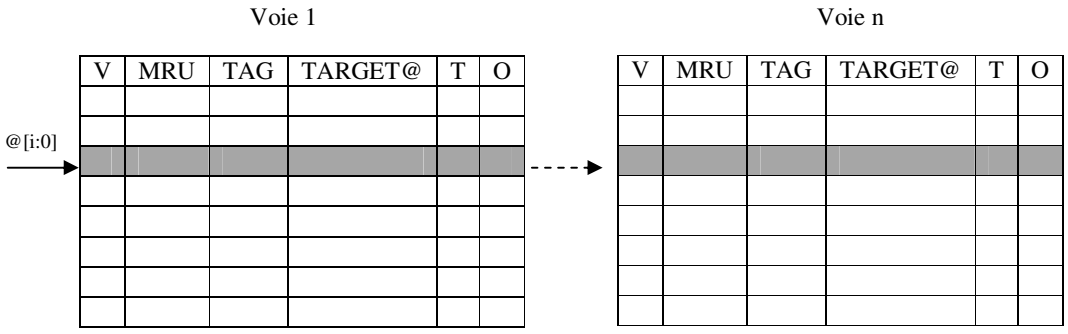


Figure 4-34 - Branch Target Buffer implémenté dans un cache associatif n-voies (n = 2).

Contrairement aux instructions des processeurs RISC "classiques", les instructions Java sont sur un octet, les adresses des sauts ne sont donc pas alignées sur 32-bits mais sur 8-bits. De plus, les requêtes effectuées par le module FETCH demandent huit octets d'instructions alignés sur 64-bits. Ces deux spécificités entraînent les choix suivants :

- On considère que l'adresse d'un branchement est l'adresse du dernier octet de l'instruction de saut.
- Plusieurs branchements peuvent être présents dans les instructions retournées par une requête faite au cache instruction (trois maximum) (ajout du bit O).

Le bit MRU est utilisé par la stratégie de remplacement des lignes de cache. L'algorithme utilisé est le P-LRUm (Pseudo-Least Recently Used base on Most recently used bit). Lorsque une nouvelle ligne est insérée dans le cache, elle se substitue à une ligne non-valide (V=0) ou ayant le bit MRU à '0' si toutes les lignes pouvant être utilisé sont valide. Le bit MRU de la nouvelle ligne est mis à '1', si tous les bits MRU des autres voies de l'adresse correspondantes sont à '1', elles sont mises à '0'.

Lors d'une requête de prédiction d'adresse, le BTB renvoie les informations suivantes :

- HIT : Indique si un branchement existe (1 bit).
- AD_JUMP : Adresse de saut (32 bits).
- BR_pos : Position de l'instruction de saut dans la requête de lecture du cache instruction (3 bits).
- Other_branch : Indique q'un second branchement existe dans la requête de lecture du cache instruction (1 bit).

4.9.b - Le prédicteur de conditions

Le prédicteur de conditions contient une table de compteurs à saturation 2-bits. Chaque compteur 2-bits correspond à une ligne du BTB et permet de prédire dynamiquement la direction des branchements contenus dans le BTB.

4.9.c - Le contrôle de branchements

Les parties effectuant le contrôle des branchements des modèles JMS_PRED et JMQ diffèrent pour s'adapter aux spécificités des deux architectures. En effet, le modèle JMQ ne résout pas tous les types de branchements dans le même module.

Pour les deux modèles, lors d'un 'hit" du BTB (avec prédiction 'pris' par le prédicteur de condition en cas de branchement conditionnel), l'adresse de saut, ainsi que la position du branchement, sont envoyés au module FETCH. Cette adresse est, soit celle contenue dans le BTB, soit le sommet de la pile de retour de méthode, en fonction du type du branchement. Deux FIFOs reçoivent l'adresse du branchement et l'adresse de saut.

Pour le modèle JMS_PRED, tous les branchements sont effectués par le module EX_C. Ce module présente sur l'interface du prédicteur les informations suivantes :

- Adresse du saut.
- Adresse suivante si le branchement est pris.
- Adresse suivante si le branchement n'est pas pris.
- Résolution du branchement (branchement pris/non-pris).
- Type du branchement.
- Indique le traitement d'un branchement.

Ces informations sont comparées avec celles présentes dans le prédicteur (contenues dans le BTB et les FIFOs d'adresse de branchement et de saut). Le BTB et le préditeur de condition (pour les sauts conditionnels) sont mis à jour. En cas d'erreur de prédiction (l'absence de prédiction est considérée comme une prédiction de saut 'non-pris'), les étages du pipe-line en amont de la résolution sont vidés et la bonne adresse de saut (présenté par l'interface du module EX_C) est utilisée par le prédicteur pour demander au module FETCH d'effectuer un saut.

Pour le modèle JMQ (Figure 4-35 [49]), les branchements conditionnels sont résolus par le module EXEC. Le module PREP (étage ENV) présente sur l'interface du prédicteur les mêmes informations que le module EX_C du modèle JMS_PRED à l'exception de la résolution des branchements conditionnels. Ces informations sont comparées également à celles contenues dans le prédicteur. Le BTB est mis à jour. En cas de mauvaise prédiction les étages du pipe-line en amont de l'étage ENV sont vidés et la bonne adresse de saut (présenté par l'interface du module PREP est utilisée par le prédicteur pour demander au module FETCH d'effectuer un saut.

Pour les branchements conditionnels, quatre FIFOs comprise dans le prédicteur gardent les informations suivantes :

- Adresse de branchement (permet de mettre à jour le prédicteur de condition).
- Adresse si le saut est pris.
- Adresse si le saut n'est pas pris.
- Prédiction du saut.

Lors de l'exécution d'un branchement conditionnel par le module EXEC, ce dernier indique au prédicteur la résolution d'un saut et le résultat de cette résolution. Le prédicteur de condition est mis à jour. En cas d'erreur de prédiction les étages du pipe-line en amont de l'étage EXEC sont vidés et la bonne adresse de saut (présenté soit dans la FIFO des adresses 'saut pris', soit dans celle des adresses 'saut non-pris') est utilisée par le prédicteur pour demander au module FETCH d'effectuer un saut.

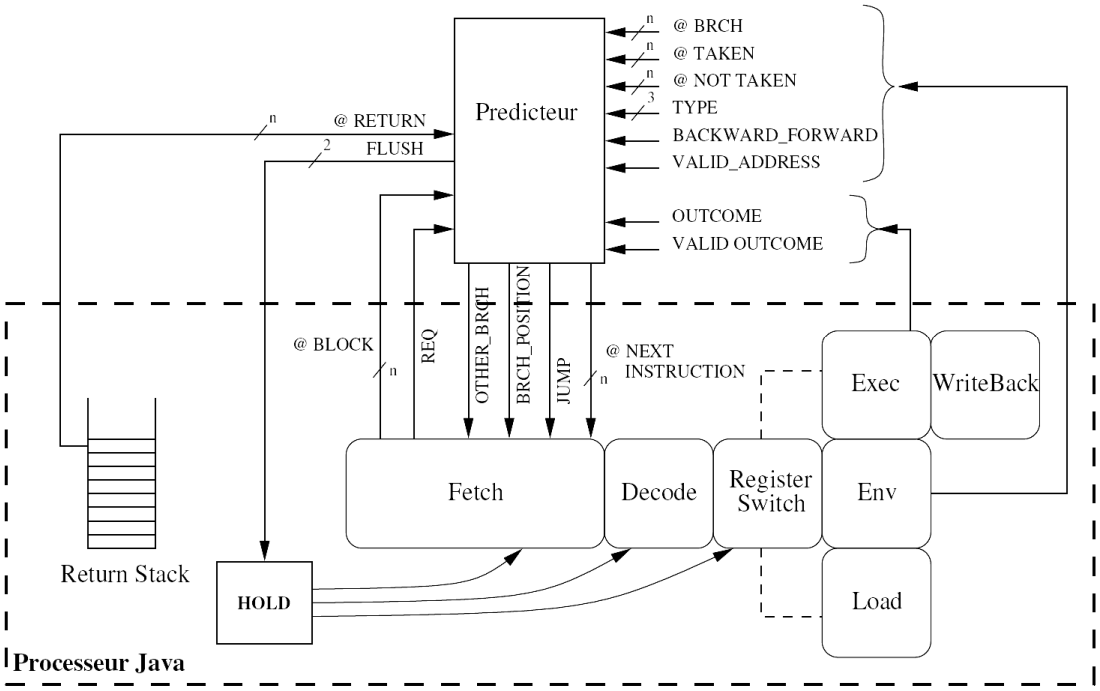


Figure 4-35 - Interface entre le prédicteur de branchement et modèle JMQ.

4.10 - La File d'exécution (FILE_EX)

La File d'exécution est le mécanisme contenant les données de la pile d'exécution Java. Elle doit permettre l'insertion de données par le module CHARG (jusqu'à deux données simultanément), ainsi que fournir un accès aux données situées au sommet de la pile d'exécution virtuelle au module EXEC.

Une entrée de la File d'exécution est constituée des éléments suivants (Figure 4-36) :

- Donnée : Contient les données (64 bits).
- T : Indique si la donnée est sur 32 ou 64 bits (1 bit).
- V : Indique la présence d'une donnée (1 bit).
- P1 : Indique le pointeur P1, sommet de la pile virtuelle de la prochaine opération (1 bit).
- P2 : Indique le pointeur P2, sous-sommet de la pile virtuelle de la prochaine opération (1 bit).

- P1_end : Indique le pointeur P1_end, entrée à déplacer par le mécanisme de compaction de la File (1 bit).
- P2_end : Indique le pointeur P2_end, emplacement de recopie du mécanisme de compaction de la File (1 bit).
- P0_a : Indique le pointeur P0_a, emplacement d'écriture dans la File par le module CHARG (1 bit).
- P0_b : Indique le pointeur P0_b, second emplacement d'écriture dans la File par le module CHARG (1 bit).

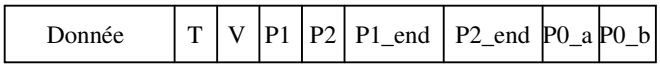


Figure 4-36 - Format d'une entrée de la file d'exécution.

La Figure 4-37 montre le mécanisme d'autorisations d'écritures du module CHARG via les pointeur P0_a et P0_b. Ce mécanisme vérifie qu'il existe des places libres dans la file d'exécution (présence des bits P0_a ou P0_b à l'adresse pointée par P_end). En cas d'écriture (avec autorisation) dans la file d'exécution, les bits représentant les pointeurs P0_a et P0_b sont décalées (de 1 par écriture).

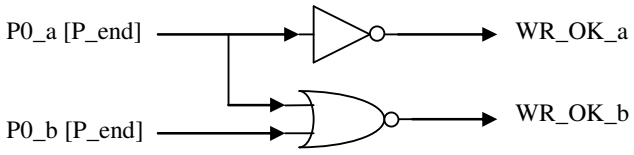


Figure 4-37 - Autorisation d'écriture dans la file d'exécution par le module CHARG.

Les pointeurs P et P_end sont conservés dans des registres de N+1 bits (N=log2(m), m étant le nombre d'entrée de la file d'exécution) pour pouvoir vérifier que les déplacements du pointeur P sont possible. En effet, la file d'exécution est circulaire, une simple soustraction ne permet donc pas de vérifier que le pointeur P ne dépasse pas le pointeur P_end. L'utilisation d'un adressage avec un bit supplémentaire rend cela réalisable (Figure 4-38). Ainsi, un déplacement est possible dans les cas suivants :

- P et P_end ont le même bit supplémentaire et P > P_end (sur N bit);
- P et P_end ont des bits supplémentaires différents et P < P_end (sur N bits).

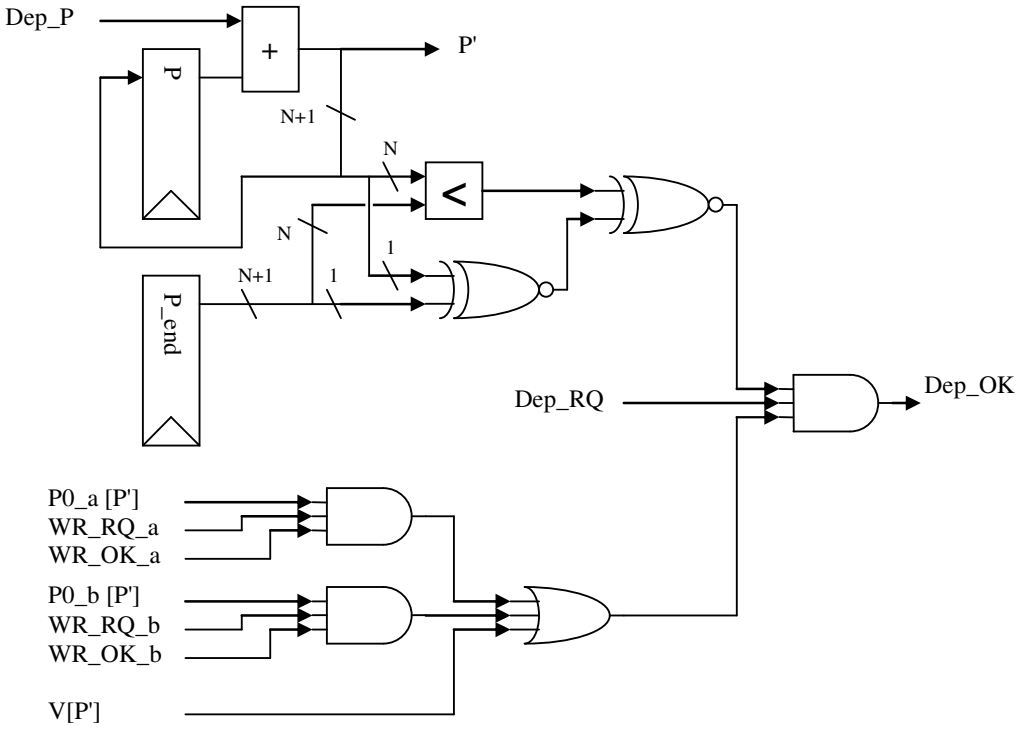


Figure 4-38 - Autorisation de déplacement du pointeur P.

Pour que le déplacement de P soit possible, il faut également qu'une requête de déplacement soit effectuée par le module EXEC et que l'entrée pointée par la nouvelle valeur de P (P') soit valide. L'autorisation de déplacement est envoyée au module EXEC et autorise également l'écriture des registres P, P1 et P2.

Les pointeurs P1 et P2 sont déduits de la nouvelle position de P (P'). La valeur de P' (sur N bits) est décodée (Figure 4-39) afin "d'exciter" le mécanisme de recherche des prochains opérandes à partir de la nouvelle position de P.

Figure 4-39 - Décodage de la nouvelle position du pointeur P.

La Figure 4-40 montre le mécanisme de recherche du sommet et du sous-sommet de la pile virtuelle contenue dans la file d'exécution. Une fois excité, le mécanisme de recherche de la prochaine position du pointeur P1 (P1'), parcourt la file par propagation jusqu'à ce qu'il trouve une entrée valide (au prochain cycle). Lorsque la position P1' est trouvée, le mécanisme de recherche de la prochaine position de P2 (P2') est excité à partir de l'entrée suivantes de la file d'exécution. Cette recherche parcourt de la même façon la file d'exécution, pour trouver, par propagation, la première entrée valide (au prochain cycle).

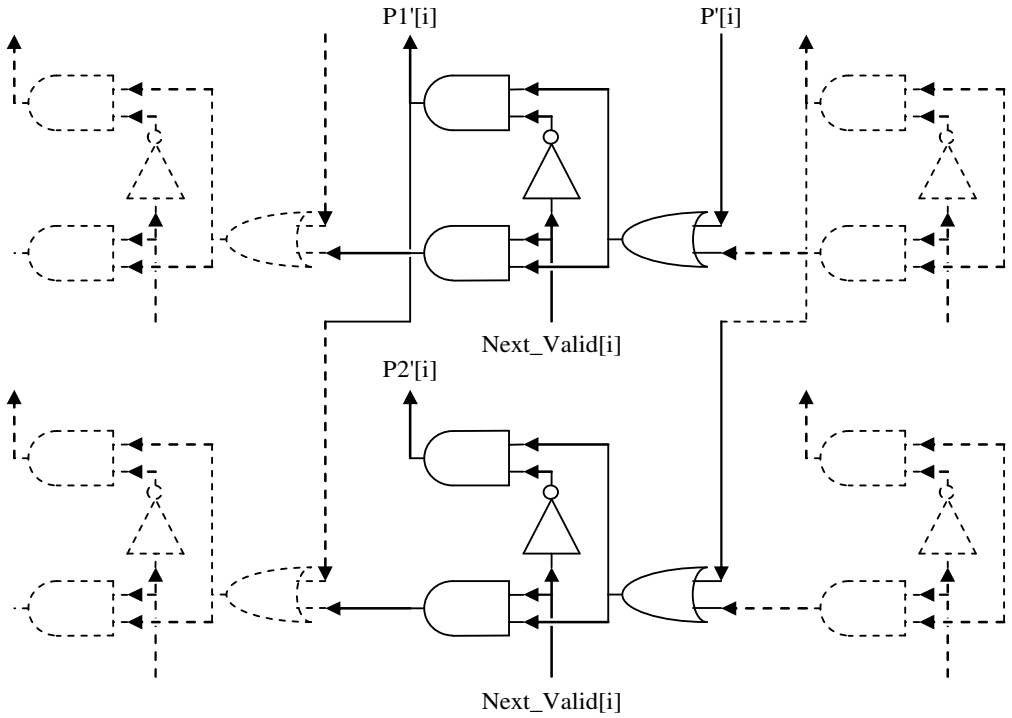


Figure 4-40 - Mécanisme de recherche des opérandes dans la file d'exécution.

Pour chaque entrée de la file, la valeur du bit de validité au prochain cycle est déterminée grâce à la valeur de son bit de validité au cycle courant, la position courante des pointeurs P1 et P2, et grâce aux informations d'écriture envoyée par le module EXEC. Ce mécanisme est représenté par la Figure 4-41

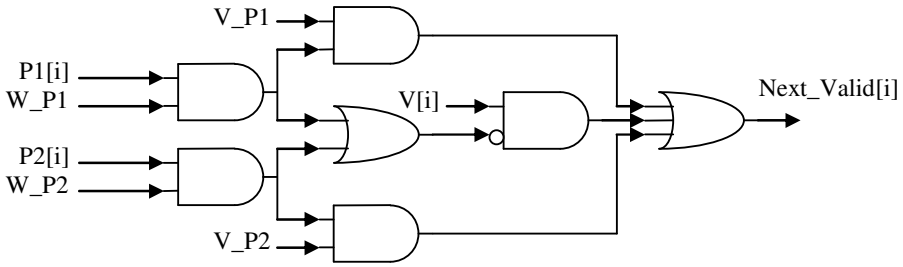


Figure 4-41 - Mécanisme de vérification de la validité d'une entrée au prochain cycle.

Le mécanisme de recherche des pointeurs P1_end et P2_end à partir du pointeur P_end (compaction de la file) est similaire au mécanisme destiné à la recherche des positions de P1 et P2.

4.11 - Conclusion

Dans ce chapitre, nous avons décrit en détails les modules présents dans les modèles de simulation ainsi que les différentes techniques mises en place pour accélérer l'exécution des programmes Java.

L'architecture des modules du modèle JMS, à l'exception des modules Fetch et de cache instructions et de données, reprend celle du processeur Picojava-II. L'architecture de ces modules est décrite au niveau porte.

Une proposition d'architecture est décrite pour les autres modules de la JMS, ainsi que ceux de la JMQ. Ces propositions d'architectures sont décrites au niveau porte.

Chapitre 5 Résultats

5.1 - Introduction

Ce chapitre présente les résultats de simulation des modèles JMS, JMS_PRED et JMQ sur des programmes Java. Il présente également l'influence de certains paramètres sur les différents modèles. Enfin, une évaluation du coût matériel est effectuée.

5.2 - Simulation

5.2.a - Les programmes de test

Pour comparer les performances des processeurs JMS, JMS_PRED et JMQ, leurs modèles sont utilisés pour exécuter les programmes suivants (définition d'après Wikipédia) :

BubbleSort

BubbleSort est un Programme de tri utilisant l'algorithme de tri à bulles. Il s'agit d'un tri itératif constitué de deux boucles imbriquées. Il consiste à faire remonter le plus grand élément du tableau (comme une bulle d'air remonte à la surface) en comparant les éléments successifs. C'est-à-dire qu'on va comparer le 1er et le 2e élément du tableau, repérer le plus grand et puis les échanger s'ils sont désordonnés l'un par rapport à l'autre. On recommence cette opération jusqu'à la fin du tableau. Ensuite, il ne reste plus qu'à renouveler cela jusqu'à l'avant-dernière place et ainsi de suite… Le tri s'arrête quand le tableau à trier est de taille 1 [51]. Ce programme simple est réparti sur une classe et n'effectue pas d'appel de méthode. Ce tri est effectué sur un tableau de mille entiers.

QuickSort

QuickSort est un programme de tri utilisant un algorithme de tri rapide inventé par C. A. R. Hoare en 1962. Il consiste à placer le premier élément d'un tableau d'éléments à trier (appelé pivot) à sa place définitive en permutant tous les éléments de telle sorte que tous ceux qui lui sont inférieurs soient à sa gauche et que tous ceux qui lui sont supérieurs soient à sa droite. Cette opération s'appelle partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète récursivement l'opération de partitionnement, jusqu'à ce que l'ensemble des éléments soient triés [52].Ce programme simple est réparti sur une classe. Ce tri est effectué sur un tableau de mille entiers.

Raytracer

Raytracer est un programme de synthèse d'image simulant le parcours inverse de la lumière de la scène vers l'œil. Il consiste, pour chaque pixel de l'image générée, à lancer un rayon depuis le point de vue (la caméra) dans la scène 3D. Le premier point d'impact du rayon sur un objet définit l'objet concerné par le pixel correspondant. Des rayons sont ensuite lancés depuis le point d'impact en direction de chaque source de lumière pour déterminer sa luminosité (est-il éclairé ou à l'ombre d'autres objets ?). Cette luminosité combinée avec la couleur de l'objet ainsi que d'autres informations éventuelles (angles entre la normale à l'objet et les sources de lumières, réflexions, transparence, etc.) déterminent la couleur finale du pixel [53]. Ce programme fait partie de la suite JavaGrande regroupant des programmes de test destinés à la comparaison des techniques d'exécution de Java, il est réparti sur 79 classes. L'image générée est de 8x8 pixels.

FFT

FFT (Fast Fourier Transform) est un programme de calcul de la transformée de Fourier basé sur l'algorithme de calcul de la transformée de Fourier discrète. Cet algorithme est couramment utilisé en traitement numérique du signal pour transformer des données du domaine temporel dans le domaine fréquentiel, en particulier dans les analyseurs de spectre. Son efficacité permet de réaliser des filtrages en passant dans le domaine transformé [54].Ce programme fait partie de la suite JavaGrande regroupant des programmes de test destinés à la comparaison des techniques d'exécution de Java, il est réparti sur 9 classes. Ce programme est appliqué sur 32768 échantillons.

Crypt

Crypt est un programme de cryptage et décryptage basé sur un algorithme de chiffrement symétrique appelé IDEA (International Data Encryption Algorithm) conçu par Xuejia Lai et James Massey. IDEA est un algorithme de chiffrement symétrique par blocs utilisé pour chiffrer et déchiffrer des données. Il manipule des blocs de texte en clair de 64 bits. Une clé de chiffrement longue de 128 bits (qui doit être choisie aléatoirement) est utilisée pour le chiffrement des données, cette même clé est nécessaire pour les déchiffrer [55].Ce programme fait partie de la suite JavaGrande regroupant des programmes de test destinés à la comparaison des techniques d'exécution de Java, il est réparti sur 8 classes. Ce programme est appliqué sur 16384 octets.

5.2.b - Paramètres de simulation

Dans les différents modèles de processeurs que nous avons étudié, un certain nombre de paramètres peuvent être modifiés. L'influence de ces paramètres sera étudiée en prenant comme point de référence les paramètres suivants.

Les paramètres généraux de simulation utilisés sont les suivants :

- Cache instructions : La taille du cache instructions utilisé est de 16Ko réparti en une voie de 512 lignes de 256 bits (4 mots de 64 bits correspondant à la taille d'une requête).
- Fetch : Le buffer d'instructions contient jusqu'à 16 octets.
- Decode : Le module de décodage est capable de grouper jusqu'à 7 octets.

Les paramètres de l'unité de prédiction des modèles JMS_PRED et JMQ sont les suivants :

- Prédicteur d'adresses : Cache 2-voies de 128 entrées chacune.
- Prédicteur de conditions : Cache de 128 compteurs 2 bits (les deux voies du prédicteur d'adresses partage le même compteur).

Les paramètres des FIFOs de la JMQ sont les suivants :

- FIFO_charg : 16 entrées.
- FIFO_calc : 32 entrées.
- File de dépendance de lecture (FDL) : 16 entrées.
- File de dépendance d'écriture (FDE) : 8 entrées.
- File d'exécution 256 entrées.

5.2.c - Comparaison

On compare les modèles JMS_PRED et JMQ au modèle JMS. On compare également les performances de la JMS à un modèle JMS possédant les groupes d'instructions du processeur Picojava-II (A.d) et appelé JMS_PICO. Le Tableau 5-1 présente le nombre de cycles nécessaires à l'exécution des programmes de test pour les différents processeurs. Le Tableau 5-2 et la Figure 5-1 présentent le gain de performance de chaque processeur comparé au modèle JMS.

Le ratio entre l'exécution JMS et l'exécution JMS_PICO permet de connaître le gain de performance apporté par les groupements d'instructions supplémentaires du modèle JMS.

Le ratio entre l'exécution JMS et l'exécution JMS_PRED permet de connaître le gain de performance apporté par l'utilisation d'une unité de prédiction de branchement, les pourcentages de réussite des différents types de branchement seront donnés ultérieurement pour chaque programme.

Le ratio entre l'exécution JMS et l'exécution JMQ permet de connaître le gain de performance apporté par l'ensemble des techniques utilisées par le modèle JMQ.

Programmes		JMS_PICO	JMS	JMS_PRED	JMQ
Tri	BubbleSort	20 344 103	20 344 065	18 150 485	19 405 962
	QuickSort	325 372	325 426	301 941	302 960
Raytracer	init	436 915	435 297	364 747	315 381
	exec	142 077 863	132 370 714	122 297 065	112 664 887
	total	142 521 447	132 812 678	122 667 915	112 985 579
FFT	transformée	78 800 601	78 276 273	76 774 159	54 991 392
	inverse	81 079 964	80 555 043	78 798 377	56 818 440
Crypt		59 766 370	56 847 445	49 841 599	33 596 241

Tableau 5-1 - Nombre de cycles des programmes de test pour les différents processeurs.

Programmes		JMS_PICO	JMS	JMS_PRED	JMQ
Tri	BubbleSort	1,00	1,00	1,12	1,05
	QuickSort	1,00	1,00	1,08	1,07
Raytracer	init	1,00	1,00	1,19	1,38
	exec	0,93	1,00	1,08	1,17
	total	0,93	1,00	1,08	1,18
FFT	transformée	0,99	1,00	1,02	1,42
	inverse	0,99	1,00	1,02	1,42
Crypt		0,95	1,00	1,14	1,69

Tableau 5-2 - Ratio JMS/processeurs.

Le gain apporté par les groupements supplémentaires n'est significatif que pour les programmes qui manipulent des variables locales 64 bits. En effet, les programmes de tri sont sur des données 32 bits, tandis que le programme FFT charge et range les données 64 bits depuis et vers des tableaux en mémoire, il n'y a donc pas de gain de performance apporté par l'ajout de groupement des instructions 64 bits.

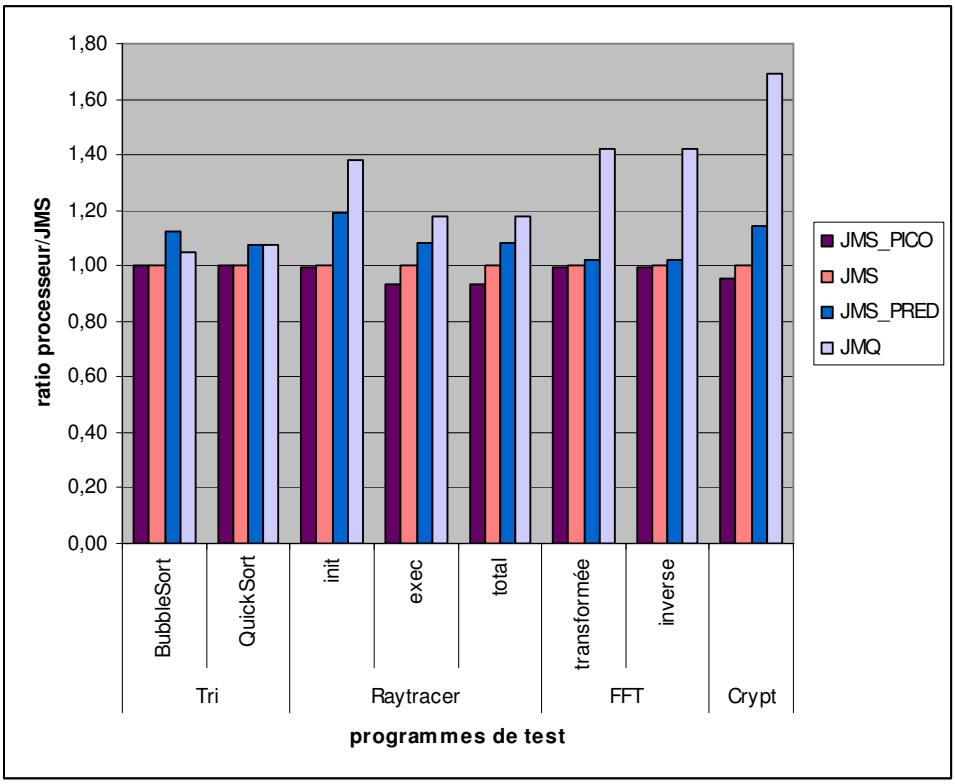


Figure 5-1 - Comparaison des performance sur les programmes de test pour chaque processeurs.

Tri

Le programme BubbleSort est un programme itératif qui ne comporte aucun appel de méthode. De plus, l'étude de l'algorithme montre qu'il n'y a pas de gain apporté par les groupements d'instructions de la JMQ par rapport à la JMS. Les performances de la JMS_PRED et de la JMQ devrait donc être les mêmes, or le gain apporté par la JMS_PRED est de 1,12 alors que celui de la JMQ est de 1,05. Cette différence s'explique de la façon suivante : lors d'une lecture mémoire par le modèle JMS, si la donnée chargée depuis la mémoire est utilisée par l'instruction suivante, le mécanisme de by-pass réclame un cycle supplémentaire. Dans le cas de la JMQ, ce cycle supplémentaire est toujours nécessaire. Cette perte de cycle représente 2 cycles par itération de la boucle de déplacement des données (avec ou sans déplacement) ce qui correspond à 1 000 000 cycles perdus sur le programme. Si donc une donnée est déplacée, la JMQ perd 4 cycles supplémentaires. La machine JMQ pourrait être utilement modifiée afin de ne plus perdre ce cycle sauf lors d'une instruction consécutive utilisant la donnée chargée.

Le programme QuickSort comporte moins de lecture mémoire mais perd également des cycles pour une partie des lectures mémoires. Cependant, une partie de ces pertes est compensée par le gain de cycles lors des appels et des retours de méthode. Le gain de la JMS_PRED est de 1,08, celui de la JMQ est de 1,07.

Pour ce test, les pourcentages de réussite aux branchements sont les suivants :

- Branchement inconditionnel direct : JMS_PRED = 98,8% JMQ = 98,5%.
- Branchement conditionnel : JMS_PRED = 85% JMQ = 85%.
- Branchement retour de méthode : JMS_PRED = 94,7% JMQ = 94,8%.
- Total : JMS_PRED = 85% JMQ = 85%.

Raytracer

Le gain de performance apporté par le prédicteur de branchement est de 1,19 pendant l'initialisation, de 1,08 pendant l'exécution et de 1,08 sur l'exécution du programme complet. Le gain de performance apporté par le modèle JMQ est de 1,38 pendant l'initialisation, de 1,17 pendant l'exécution et de 1,18 sur l'exécution du programme complet. Le programme Raytracer effectue un grand nombre d'appels et de retours de méthode permettant un gain de performance important avec le modèle JMQ. De plus, les groupements d'instructions du modèle JMQ permettent également un gain de performance. Comme pour les programmes précédents, des cycles sont perdus par le modèle JMQ lors des lectures mémoire dont la donnée n'est pas utilisée au cycle suivant.

Pour ce test, les pourcentages de réussite aux branchements sont les suivants :

- Branchement inconditionnel direct : JMS_PRED = 98% JMQ = 97,5%.
- Branchement conditionnel : JMS_PRED = 83,1% JMQ = 83,6%.
- Branchement retour de méthode : JMS_PRED = 92,5% JMQ = 92,2%.
- Total : JMS_PRED = 89% JMQ = 89%.

FFT

Le gain de performance apporté par le prédicteur de branchement est de 1,02 pour la transformée de Fourier et la transformée inverse. Ce gain est 1,42 pour le modèle JMQ. . Comme pour les programmes précédents, des cycles sont perdus par le modèle JMQ lors des lectures mémoire dont la donnée n'est pas utilisée au cycle suivant.

Pour ce test, les pourcentages de réussite aux branchements sont les suivants :

- Branchement inconditionnel direct : JMS_PRED = 99,9% JMQ = 99,9%.
- Branchement inconditionnel direct : JMS_PRED = 33% JMQ = 33,5%.
- Branchement conditionnel : JMS_PRED = 87,2% JMQ = 87,2%.
- Branchement retour de méthode : JMS_PRED = 99,9% JMQ = 99,9%.
- Total : JMS_PRED = 92% JMQ = 92%.

Crypt

Le gain de performance apporté par le prédicteur de branchement est de 1,14 pour le cryptage puis le décryptage. Ce gain est 1,69 pour le modèle JMQ. Ce programme n'appelle pas de méthode mais possède des instructions à émuler en logiciel par la prise du gestionnaire d'interruption (trap handler). Le modèle JMQ permet de paralléliser l'appel de la routine logiciel ainsi que son retour avec les autres instructions et ainsi d'en gagner tous les cycles. De plus, toutes les lectures mémoire de l'algorithme sont suivies d'une instruction les utilisant, le modèle JMQ n'est donc pas pénalisé. Enfin, les groupements d'instructions du modèle JMQ permettent de paralléliser toutes les lectures de variables locales qui ne peuvent pas l'être par la JMS.

Pour ce test, les pourcentages de réussite aux branchements sont les suivants :

- Branchement inconditionnel direct : JMS_PRED = 99,9% JMQ = 99,9%.
- Branchement conditionnel : JMS_PRED = 96,7% JMQ = 95,4%.
- Branchement retour de méthode : JMS_PRED = 99,9% JMQ = 99,9%.
- Total : JMS_PRED = 97,4% JMQ = 96,4%.

Le cache du prédicteur de branchement de la JMS_PRED a été porté à 2 voies de 256 entrées car le cache 128 entrées donnait un pourcentage de réussite moins bon que la JMQ à cause de deux branchements antagonistes partageant le même compteur à saturation.

5.2.d - Influences des paramètres généraux

Nous cherchons ici à déterminer l'influence d'une augmentation du nombre maximum d'octet décodé à chaque cycle et celle d'une augmentation de la taille du buffer d'instructions pour chaque architecture. Nous étudierons également l'influence de la taille du cache instructions.

Nombre d'instructions décodées

Nous avons augmenté le nombre maximum d'octets décodé par cycle de 7 à 8 octets. Comme les résultats de simulation (A.e Automate d'état du module Fetch) ne montrent aucune diminution du nombre de cycles pour les modèles JMS et JMS_PRED, nous ne l'avons pas retenue. Ces mêmes résultats montrent que le gain apporté par cette modification est inférieur à 0,2% pour les programmes Raytracer et FFT, et inexistant pour les autres programmes. Cette augmentation n'a donc aucune influence sur le nombre de cycles de simulation nécessaires à l'exécution des programmes de test sur les modèles JMS, JMS_PRED et JMQ. Elle ne permet pas de grouper plus d'instructions par cycle.

Taille du cache instructions et taille du buffer d'instructions

La taille des mémoires peut également être un facteur important. Dans les systèmes embarqués, celle-ci est généralement réduite pour des raisons de coût. Nous avons donc cherché à déterminer l'influence de la taille du cache instruction sur les différents modèles que nous avons étudiés, ainsi que l'influence de la taille du buffer d'instruction en fonction de la taille du cache. Les programmes de tri et de transformée de Fourier sont trop petits pour que la taille du cache ait une réelle influence sur les performances, néanmoins des résultats concernant ces programmes sont disponibles en Annexe (A.f Résultats complémentaires). Nous nous bornerons donc à commenter ici l'impact des tailles mémoires sur les programmes Raytracer et Crypt en faisant varier la taille du cache de 8 à 512 lignes (de 256 bits) représentant de 256 octets à 16 Ko de cache. Nous étudierons également l'influence de la taille du buffer d'instructions en fonction de la taille du cache instructions. Les deux tailles de buffer d'instructions étudiées sont 16 et 24 octets. Le Tableau 5-3, le Tableau 5-4, la Figure 5-2 et la Figure 5-3 montrent l'évolution du nombre de cycles nécessaires à l'exécution du programme Raytracer (init et exec) en fonction des tailles du buffer et du cache. Le Tableau 5-5 et la Figure 5-4 montrent l'influence de ces paramètres sur l'exécution du programme Crypt.

Taille du cache instructions	Nombre de cycles (Raytracer : init)			Nombre de cycles (Raytracer : exec)		
	JMS	JMS_PRED	JMQ	JMS	JMS_PRED	JMQ
256	635 927	484 739	441 147	163 669 161	128 131 962	118 503 351
512	589 930	454 157	400 680	150 225 410	126 808 064	117 253 499
1024	521 919	422 332	360 496	139 073 894	125 866 701	115 544 491
2048	483 903	395 818	342 706	136 467 159	124 070 115	114 466 179
4096	450 930	375 510	329 081	134 454 003	123 103 219	113 544 130
8192	440 203	368 365	320 324	132 794 025	122 516 675	112 783 429
16384	435 297	364 747	315 381	132 370 714	122 297 065	112 664 887

Tableau 5-3 - Influence de la taille du cache instructions (B=16 : Raytracer).

Taille du cache instructions	Nombre de cycles (Raytracer : init)			Nombre de cycles (Raytracer : exec)		
	JMS	JMS_PRED	JMQ	JMS	JMS_PRED	JMQ
256	649 478	468 373	427 995	165 735 002	131 523 874	116 723 426
512	598 808	440 701	390 942	149 485 675	130 633 126	115 864 189
1024	532 960	414 430	356 035	138 489 222	125 980 624	113 936 711
2048	489 797	391 015	340 040	135 862 958	124 343 575	112 890 695
4096	454 096	373 509	327 264	133 822 917	123 482 588	112 002 151
8192	442 000	367 402	319 637	132 268 464	123 019 957	111 326 916
16384	436 163	364 323	314 781	131 863 198	122 854 229	111 209 020

Tableau 5-4 - Influence de la taille du cache instructions (B=24 : Raytracer).

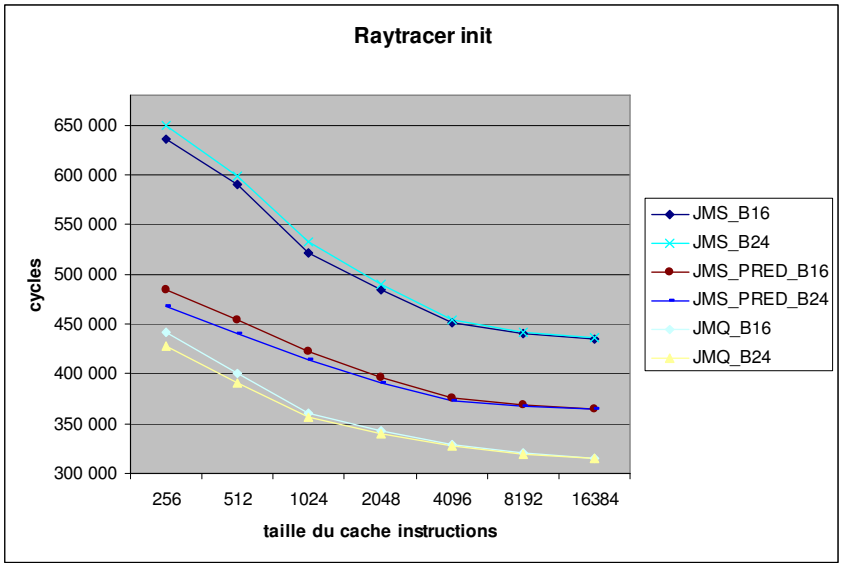


Figure 5-2 - Influence de la taille du cache instructions (Raytrace : init).

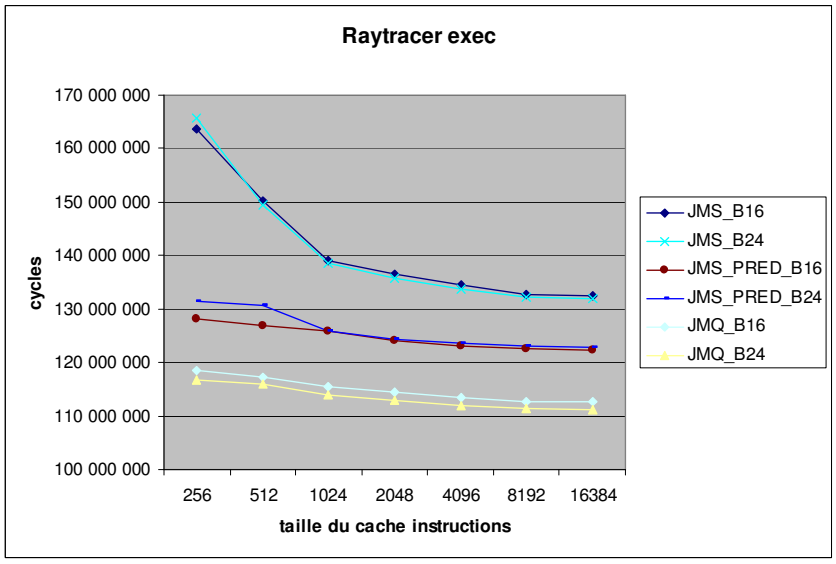


Figure 5-3 - Influence de la taille du cache instructions (Raytrace : exec).

Taille du cache instructions	Nombre de cycles (Crypt : B = 16)			Nombre de cycles (Crypt : B = 24)		
	JMS	JMS_PRED	JMQ	JMS	JMS_PRED	JMQ
256	94 328 154	69 225 988	53 611 152	88 359 611	61 846 201	49 975 915
512	85 832 396	62 970 091	42 036 759	82 724 693	58 380 099	40 408 808
1024	63 272 023	52 574 670	35 732 169	64 176 630	51 243 164	35 386 051
2048	61 982 365	52 078 647	35 699 912	62 934 850	51 221 925	35 364 827
4096	61 981 972	52 078 367	35 698 920	62 934 540	51 221 648	35 363 872
8192	56 847 633	49 841 719	33 596 703	56 847 668	49 861 660	33 415 644
16384	56 847 445	49 841 599	33 596 241	56 847 539	49 861 583	33 415 244

Tableau 5-5 - Influence de la taille du cache instructions (Crypt).

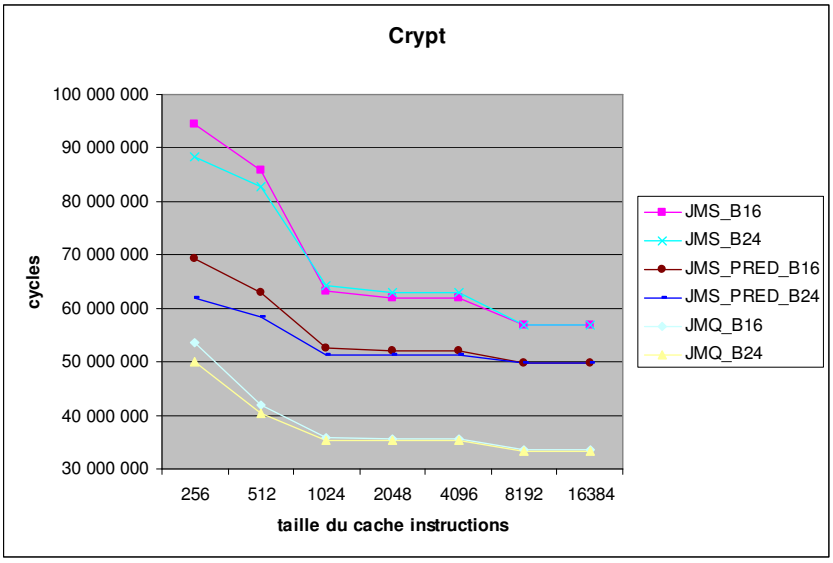


Figure 5-4 - Influence de la taille du cache instructions (Crypt).

Pour chacun des modèles, l'augmentation de la taille du cache instructions se traduit par la diminution du nombre de cycles nécessaires à l'exécution des programmes Raytracer et Crypt. Cette diminution est due à l'augmentation de la part d'application stockée dans le cache instructions qui entraîne une diminution des accès à la mémoire. Les modèles JMS_PRED et JMQ sont moins sensibles à la diminution de la taille du cache instructions grâce aux unités de prédiction de branchement dont ils sont équipés. En effet, la prédiction de branchement permet d'anticiper les sauts et donc de masquer une partie de la latence d'obtention des instructions depuis la mémoire lorsque celles-ci ne sont pas présentes dans le cache.

L'augmentation de la taille du buffer d'instruction permet de charger plus d'instructions en avance et donc de recouvrir une partie de la latence d'accès aux instructions lorsque celles-ci ne se trouvent pas dans le cache. Elle doit également permettre de grouper un maximum d'instructions à chaque cycle. Elle produit des effets différents en fonction des processeurs et de la taille du cache.

Pour le modèle JMS, cela se traduit, tantôt par la diminution, tantôt par l'augmentation du nombre de cycles de simulation. En effet, en l'absence d'unité de prédiction de branchement, les instructions chargées à l'avance ont plus de chance d'être invalidées. Comme le type de cache utilisé ne permet pas l'invalidation des requêtes en attente, le module doit

donc attendre le retour de ces requêtes pour les invalider avant de demander de nouvelles instructions.

Pour le modèle JMS_PRED, l'augmentation de la taille du buffer d'instructions permet de diminuer le nombre de cycles d'exécution pour les programmes Raytracer : init (Figure 5-2) et Crypt (Figure 5-4). Ces programmes ont de très bons taux de réussite aux branchements. Pour les autres programmes, le nombre de cycles augmente.

Pour le modèle JMQ, cette augmentation permet de réduire le nombre de cycles nécessaires pour exécuter les programmes Raytracer et Crypt. Néanmoins, pour les autres programmes de test ce nombre augmente.

Pour tous les processeurs, et pour tous les programmes, l'influence de la taille du buffer d'instructions diminue en fonction de l'augmentation de la taille du cache. Ainsi, pour un cache instructions de 16 Ko, l'influence de la taille du buffer d'instructions est minime et rarement bénéfique.

Le gain de performance obtenu par l'utilisation de la JMQ peut permettre de réduire la taille du cache instructions afin de réduire la taille globale du processeur tout en obtenant des performances supérieures. En effet, un processeur JMQ équipé d'un cache instructions de 4 Ko est plus rapide qu'une JMS avec 16 Ko de cache instructions.

5.2.e - Influence de la taille des files.

La taille des files concerne à la fois les files d'attente et la file d'exécution.

La taille des files d'attentes a été déterminée de façon expérimentale en cours de mise au point du modèle JMQ. La question s'est posée ensuite, après optimisation fonctionnelle de ce modèle, de s'assurer de la pertinence des tailles adoptées au départ.

Pour optimiser la taille des files d'attentes du modèle JMQ nous avons procédé à des simulations des programmes de tests en utilisant des combinaisons de tailles réduites des files.

L'hypothèse a été prise que la réduction ne devait pas altérer la performance de plus de 0,75%. Nous n'avons cependant pas exploré la totalité de combinaisons possibles mais seulement quelques unes qui paraissaient intéressantes à tester.

Le Tableau 5-6 en page suivante présente les résultats obtenus en termes de nombre de cycles obtenus pour chaque programme avec l'indication du pourcentage d'augmentation entraîné par la réduction des files relativement à la configuration initiale présentée en colonne de droite du tableau.

Il apparaît que, pour les combinaisons de tailles des files d'attentes testées, la combinaison qui semble optimale est la suivante :

- FIFO_CHARG = 8 entrées.
- File de dépendance de lectures = 8 entrées.
- File de dépendance d'écriture = 4 entrées.
- FIFO_CALC = 8 entrées.

Par ailleurs, la taille optimale de la file d'exécution n'a pas pu être recherchée car elle ne comporte, pour l'instant, aucun mécanisme de dribbling permettant de la prolonger en mémoire. Les programmes de test nécessitent une file d'exécution minimum de 256 entrées, celle-ci n'a donc pas pu être réduite à ce stade du développement.

Dans une phase ultérieure, il pourrait être utile de prévoir la mise en place d'un tel mécanisme et d'optimiser la taille de la file d'exécution.

Tailles des files	**charg**	**4**	**8**	**8**	**16**	**16**	**16**
	read	**4**	**8**	**16**	**16**	**16**	**16**
	write	**2**	**4**	**4**	**4**	**4**	**8**
	calc	**4**	**8**	**16**	**16**	**32**	**32**
Tri	**QuickSort**	308 998	305 020	302 953	302 953	302 953	**302 960**
		101,99%	**100,68%**	100,00%	100,00%	100,00%	**100,00%**
	BubbleSort	19 370 933	19 445 721	19 382 246	19 380 497	19 405 962	**19 405 962**
		99,82%	**100,20%**	99,88%	99,87%	100,00%	**100,00%**
Raytracer	**init**	325 929	316 007	315 978	315 937	315 875	**315 381**
		103,34%	**100,20%**	100,19%	100,18%	100,16%	**100,00%**
	exec	113 806 014	112 407 776	112 417 910	112 418 679	112 424 874	**112 664 887**
		101,01%	**99,77%**	99,78%	99,78%	99,79%	**100,00%**
	total	114 137 515	112 729 126	112 739 164	112 739 911	112 745 994	**112 985 579**
		101,02%	**99,77%**	99,78%	99,78%	99,79%	**100,00%**
FFT	**transformée**	56 308 546	55 175 071	55 072 566	55 072 566	55 075 897	**54 991 392**
		102,40%	**100,33%**	100,15%	100,15%	100,15%	**100,00%**
	inverse	58 141 497	57 004 882	56 916 936	56 916 936	56 912 583	**56 818 440**
		102,33%	**100,33%**	100,17%	100,17%	100,17%	**100,00%**
Crypt		35 580 590	33 618 965	33 612 912	33 612 912	33 612 912	**33 596 241**
		105,91%	**100,07%**	100,05%	100,05%	100,05%	**100,00%**

Tableau 5-6 - Influence de la taille des files.

5.3 - Évaluation du coût matériel

L'évaluation du coût matériel des différents modèles de processeurs implémentés (JMS, JMS_PRED, JMQ) n'est pas aisée car ils sont décrits dans un langage de description précis au niveau cycle (SystemC) et non en VHDL ou Verilog qui sont plus proches de l'implémentation matérielle. Cependant, ce niveau de description permet de connaître avec précision le nombre de points mémorisants contenus dans chacun de ces modèles. Nous utiliserons donc ce nombre pour comparer le coût de nos modèles, une bonne approximation peut être effectuée en utilisant la complexité du processeur Picojava-II puis en utilisant les trois hypothèses suivantes :

- Picojava-II : 442K portes dont 128K portes de logique et 314K portes de mémoire (16 Ko de cache de données + 16 ko de cache instructions + ROM).
- Un bit de registre = 5 portes.
- Un bit de RAM = 1,5 portes.

Le nombre de points mémorisants contenus dans chacun des modules des modèles développés est présenté des les tableaux suivants :

Le Tableau 5-7 montre le nombre de points mémorisants (registres et RAM) contenus dans les modules qui constitue le modèle JMS.

	Fetch	**Decode**	**Reg_wb**	**Ex_c**	**Total**
Registre	441	146	311	1188	2086
RAM	0	0	2048	0	2048

Tableau 5-7 - Répartition des points mémorisants du modèle JMS.

Le Tableau 5-8 montre le nombre de points mémorisants (registres et RAM) contenus dans les modules du modèle JMS_PRED.

	Fetch	Decode	Reg_wb	Ex_c	Pred	Total
Registre	451	153	311	1516	0	2431
RAM	0	0	2048	0	16640	18688

Tableau 5-8 - Répartition des points mémorisants du modèle JMS_PRED.

Le Tableau 5-9 montre le nombre de registres contenus dans les modules du modèle JMQ.

	Fetch	Decode	Prep	Charg	Exec	Total
Registre	451	181	1484	64	350	2530

Tableau 5-9 - Répartition des registres dans le modèle JMQ.

Le Tableau 5-10 montre le nombre de points de mémoire RAM contenus dans les modules du modèle JMQ.

	Prep	Fifo_charg	Fifo_calc	FDL	FDE	File_ex	Pred	Total
RAM	2516	384	272	550	138	17920	17028	38808

Tableau 5-10 - Répartition des mémoires RAM dans le modèle JMQ.

Nous utilisons la taille du Picojava-II comme base de calcul pour estimer le coût matériel des processeurs simulés. Ainsi, nous estimons la taille du modèle JMS comme équivalente à celle du Picojava-II (442 Kportes). L'estimation des autres processeurs est faite en ajoutant la taille correspondant aux points mémorisants supplémentaires.

Le Tableau 5-11 présente une évaluation du coût matériel des processeurs JMS, JMS_PRED et JMQ, ainsi que l'accroissement relatif des modèles JMS_PRED et JMQ par rapport au modèle JMS.

	JMS (portes)	JMS_PRED (portes)	accroissement (JMS/JMS_PRED)	JMQ (portes)	accroissement (JMS/JMQ)
Logique	128	155	21,1%	185	44,5%
Logique + mémoire	442	469	6,1%	499	12,9%

Tableau 5-11 - Evaluation du coût matériel des processeurs JMS, JMS_PRED et JMQ.

Cette évaluation montre une augmentation de 6,1% du nombre de portes utilisée par le modèle JMS_PRED par rapport au modèle JMS. Cette augmentation est de 12,9% pour le modèle JMQ.

Cette évaluation est à tempérer par deux remarques :

- Le grand nombre de ports de lecture et d'écriture de la file d'exécution augmente le nombre de portes à prendre en compte par point mémorisant. Cet organe est donc sous-évalué.
- Le nombre d'entrées de la file d'exécution est surévalué car aucun mécanisme de dribbling n'est actuellement implémenté pour cet organe. La taille optimum de la file d'exécution devrait être comprise entre 16 et 128 entrées.

5.4 - Conclusion

Dans ce chapitre, nous avons présenté les résultats de simulation des programmes de test sur les architectures JMS, JMS_PRED et JMQ. Nous avons vu que l'architecture du modèle JMQ permet une exécution de 1,05 à 1,69 fois plus rapide de ces programmes par rapport à leurs exécutions sur le modèle JMS. Nous avons également dégagé, grâce au modèle JMS_PRED, la part de ce gain imputable à l'ajout d'une unité de prédiction de branchement.

Les résultats obtenus, pour les différentes valeurs affectées au nombre d'octets décodés par cycle et d'octets présents dans le buffer d'instructions, nous montrent que l'augmentation de ces valeurs n'a pas d'influence sur les performances des différentes architectures.

Nous avons défini les tailles optimums des files présentes dans le modèle JMQ à l'exception de la file d'exécution. En effet, l'ajout d'un mécanisme de dribbling est nécessaire pour pouvoir utiliser une file d'exécution plus petite et définir la taille optimum de celle-ci.

Une évaluation du coût matériel à été effectuée, rapportant une augmentation du nombre de portes nécessaires au modèle JMQ de 12,9% par rapport au modèle JMS.

Les programmes de test nous ont permis de mettre en évidence une perte de cycles lors des lectures mémoire dans le modèle JMQ. En effet, lors des lectures mémoire, le modèle JMQ a toujours besoin d'un cycle supplémentaire alors que celui-ci n'est nécessaire que lors de l'utilisation de cette donnée par l'instruction suivant immédiatement cette lecture mémoire. Le modèle JMQ devra donc être modifié en conséquence. Ces performances en seront améliorées.

Chapitre 6 Conclusion

Dans cette thèse nous avons présenté une nouvelle architecture de processeurs Java. Cette architecture utilise deux organes différents pour stocker, d'une part, la pile d'environnement Java, et d'autre part, le mécanisme d'exécution. Ce dernier utilise la technique d'exécution sur file. Trois modèles de processeurs ont été décrits en SystemC afin de pouvoir comparer notre nouvelle architecture appelée JMQ avec deux autres modèles de processeur à pile : un modèle simple (JMS) et un autre doté d'une unité de prédiction de branchement (JMS_PRED).

Nous avons vu que l'architecture JMQ permet d'accélérer l'exécution des programmes Java d'un facteur 1,05 à 1,69 au prix d'une augmentation de la complexité de l'ordre de 13%.

La démonstration a été faite que l'avantage est d'autant plus grand que les programmes testés utilisent les mécanismes d'accélération mis en place.

Cette thèse est le résultat de quatre années de travail consacrées en grande partie au développement et à la mise au point des modèles de simulation. Les résultats obtenus permettent d'attester la validité des hypothèses de départ, à savoir que l'exécution des programmes Java peut être accélérée :

- en utilisant deux organes différents pour stocker la pile d'environnement Java et le mécanisme d'exécution afin d'accélérer les changements de contexte Java.
- en utilisant la technique d'exécution sur file pour accélérer l'exécution des autres instructions.

Les temps consacrés aux développements et au débogage n'ont certes permis qu'une mise au point partielle du modèle JMQ, mais celui-ci est d'ores et déjà capable d'exécuter 90% des instructions du Bytecode Java. Une phase de développement ultérieure devrait permettre d'implémenter les instructions manquantes, d'améliorer ou de résorber certaines faiblesses de comportement relevées lors des tests et d'approcher ainsi un facteur 2 d'accélération par rapport à la machine JMS.

Perspectives

Les perspectives de développement de ce projet sont multiples, aussi bien du coté matériel que logiciel.

Du coté matériel, Il reste quelques instructions à implémenter, mais surtout des possibilités d'accélérer encore l'exécution des programmes Java. Plusieurs pistes peuvent être suivies.

La première d'entre elles consisterait à ajouter un bit dans la file d'exécution pour indiquer qu'une donnée est prête. Ceci permettrait de réaffecter l'un des deux ports en écriture du module EXEC à la partie accès mémoire de ce module. Ce port est, dans la configuration actuelle, utilisé par un faible nombre d'instructions qui nécessiteront après modification un cycle supplémentaire. Ce couple de modification permettra, lors des instructions de lecture mémoire, de continuer l'exécution des instructions sans geler le pipeline. Si l'instruction suivante nécessite l'utilisation de la donnée à charger, celle-ci attend que la donnée soit prête sinon l'exécution continue. Le gain apporté est double car il permettra non seulement de ne pas perdre un cycle par rapport au modèle JMS lors des lectures mémoire mais également de continuer l'exécution sans geler le pipeline en cas d'absence de la donnée (miss) dans le cache de données et tant que la donnée à charger n'est pas utilisée.

Une modification à apporter à la machine JMQ serait l'adjonction d'un troisième port en écriture sur la file d'exécution par le module EXEC. Ce port supplémentaire serait dédié à l'écriture dans la file d'exécution des résultats provenant de la FPU (Floating Point Unit). En effet, les instructions de calcul sur les nombres flottants nécessitent plusieurs cycles pour être exécutées. Cette modification permettrait d'exécuter des instructions non flottantes pendant l'exécution de celles-ci. Les différentes parties du module EXEC deviendraient alors indépendantes les unes des autres (opérateurs d'exécution multiples) :

- Accès mémoire.
- Unité flottante (FPU).
- Extraction des opérandes et instructions restantes.

Il existe encore de nombreuses pistes à explorer (ajout d'opérateurs d'exécution, exécution multi-thread en temps partagé, utilisation pour des applications temps réel, etc.).

L'étude des techniques actuelles de Garbage collector devrait être effectuée afin d'implémenter ce mécanisme sur le modèle JMQ et de développer le matériel nécessaire pour l'accélérer.

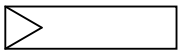
Du coté logiciel, le chargeur de classes actuel est un chargeur statique de classe. Il est aujourd'hui capable de charger des classes au format Java 1.3. Une modification de ce chargeur nous permettrait de charger des classes au format actuel (Java 1.5). De plus, l'écriture d'un chargeur dynamique de classe écrit en Java permettrait de charger des classes "à la volée" en mémoire.

Chapitre A ANNEXES

A.a Lexique des symboles

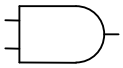
Les symboles utilisés dans la description de l'architecture sont les suivants :

- registre :

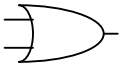


- multiplexeur :

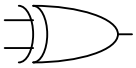
- porte "ET" :



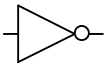
- porte "OU" :



- porte "XOR"



- porte "NON"



- complémente une entrée ou une sortie logique :

- incrémenteur :

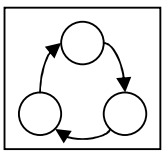
- masque logique :

- comparateur (de supériorité) :

- comparateur (d'égalité) :

- décaleur :

- automate :



A.b Les registres internes des modèles JMS et JMQ

A.b.i Le compteur ordinal (PC)

Ce registre contient l'adresse du premier octet de l'instruction exécuté :

- par l'étage EXEC du pipeline du modèle JMS.
- Par l'étage ENV du pipeline du modèle JMQ

Le registre PC est représenté par la Figure A-1.

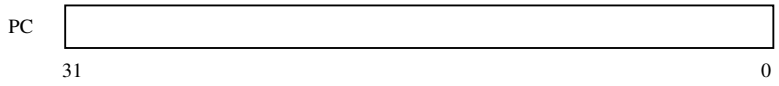


Figure A-1 - Compteur ordinal (PC).

A.b.ii Pointeur de zone de variables locales (VARS)

Ce registre contient l'adresse de la zone de variables locales de la pile Java de la méthode courante. Cette adresse, alignée sur 32-bits, indique la variable locale zéro. Les autres variables locales sont accédées à partir de ce registre moins un index. Ainsi, la variable locale 1 est accédée à partir de l'adresse contenue dans le registre VARS moins 4 (taille en octet d'un mot).

Le registre VARS est représenté par la Figure A-2.

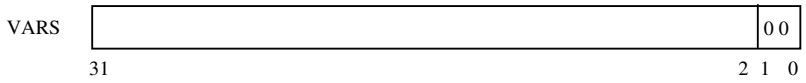


Figure A-2 - Pointeur de zone de variables locales (VARS).

Les bits 1 et 0 du registre VARS sont en lecture seul et sont égaux à zéro.

A.b.iii Pointeur de zone d'environnement (FRAME)

Ce registre contient l'adresse de la zone de sauvegarde de l'environnement à rétablir lors d'un retour de méthode Java. Cette adresse, alignée sur 32-bits, indique la sauvegarde du registre PC à restaurer lors d'un retour de la méthode courante. Les autres informations liées à l'environnement précédent sont accessible à partir de l'adresse contenue dans le registre FRAME moins un index.

Le registre FRAME est représenté par la Figure A-3.

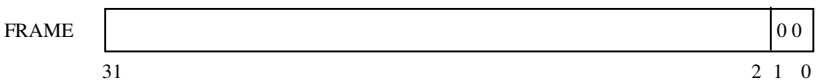


Figure A-3 - Pointeur de zone d'environnement (FRAME).

Les bits 1 et 0 du registre FRAME sont en lecture seul et sont égaux à zéro.

A.b.iv Pointeur de sommet de pile (OPTOP)

Ce registre contient l'adresse du sommet de pile d'exécution pour le modèle JMS et de la pile d'environnement pour le modèle JMQ. Cette adresse, alignée sur 32-bits, indique l'adresse d'une entrée libre au sommet des piles précédemment citées. La première donnée valide se trouve donc à l'adresse OPTOP+4.

Le registre OPTOP est représenté par la Figure A-4.

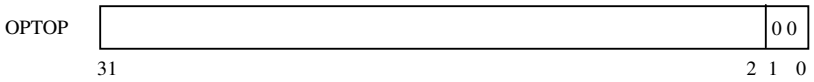


Figure A-4 - Pointeur de sommet de pile (OPTOP).

Les bits 1 et 0 du registre OPTOP sont en lecture seul et sont égaux à zéro.

A.b.v Valeur minimum de sommet de pile (OPLIM)

Ce registre contient la valeur minimale pouvant être contenu dans le registre de sommet de pile (OPTOP). L'adresse contenue dans le registre OPLIM marque la limite de croissance en mémoire de la pile Java. En cas de dépassement de la valeur OPLIM par le registre OPTOP, une exception est provoquée.

Le registre OPLIM est représenté par la Figure A-5.

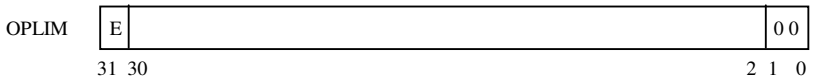


Figure A-5 - Valeur minimal de sommet de pile (OPLIM).

Les bits 1 et 0 du registre OPLIM sont en lecture seule et sont égaux à zéro.

Le bit 31 du registre OPLIM (E) indique si la vérification (OPTOP>OPLIM) doit être effectuée. Cette vérification est effectuée sur les bits 30 à 2 des registres OPTOP et OPLIM.

A.b.vi Adresse de fond de la pile matérielle (SC_BOTTOM)

Ce registre contient l'adresse de fond de la pile matérielle Java pour le modèle JMS et de la pile matérielle d'environnement pour le modèle JMQ. Cette adresse est alignée sur 32-bits et correspond à l'entrée la plus profondément contenue dans la pile matérielle.

Le registre SC_BOTTOM est représenté par la Figure A-6.

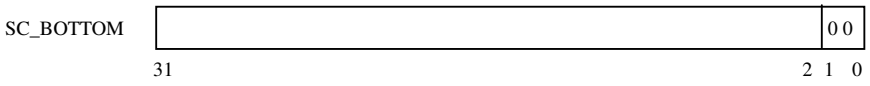


Figure A-6 - Adresse de fond de la pile (SC_BOTTOM).

Les bits 1 et 0 du registre SC_BOTTOM sont en lecture seule et sont égaux à zéro.

A.b.vii Pointeur sur le constant_pool (CONST_POOL)

Ce registre contient l'adresse de la zone de constant pool de la méthode courante. Cette adresse est alignée sur 32-bits et indique l'élément zéro de la zone de constant pool. Les autres éléments du constant_pool sont accessibles à partir de l'adresse contenue dans le registre CONST_POOL plus un index positif.

Le registre CONST_POOL est représenté par la Figure A-7.

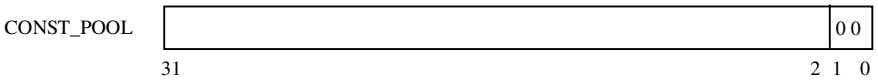


Figure A-7 - Pointeur sur le constant pool (CONST_POOL).

Les bits 1 et 0 du registre CONST_POOL sont en lecture seule et sont égaux à zéro.

A.b.viii Le pointeur de méthode (METHOD_DESC)

Ce registre contient l'adresse de la zone de description de la méthode courante. Ce registre est présent uniquement sur le modèle JMQ. Ce registre se trouve dans la pile d'environnement pour le modèle JMS. Cette adresse est alignée sur 32-bits et pointe sur l'adresse de début de la méthode. Les autres informations relatives à la méthode sont contenues aux adresses consécutives.

Le registre METHOD_DESC est représenté par la Figure A-8.

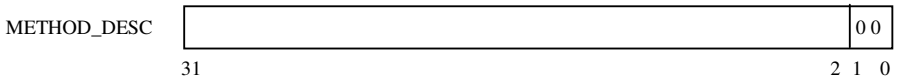


Figure A-8 - Le pointeur de méthode (METHOD_DESC).

Les bits 1 et 0 du registre METHOD_DESC sont en lecture seule et sont égaux à zéro.

A.b.ix Registre d'état du processeur (PSR)

Ce registre contient l'état de processeur.

Le registre PSR est représenté par la Figure A-9, les champs qui le composent sont décrits par le Tableau 2-1.

Champs	Descriptions
BDH	Indique la limite supérieure qui déclanche le dribbling
DBL	Indique la limite inférieure qui déclanche le dribbling
DE	Indique le fonctionnement du mécanisme de dribbling
FE	Indique le fonctionnement de l'unité de groupement
SU	Indique le mode superviseur
IE	Autorise les interruptions
PIL	Niveau d'interruption

Tableau A-1 - Champs du registre PSR.

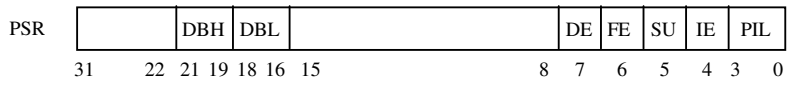


Figure A-9 - Registre d'état du processeur (PSR).

Les limites de fonctionnement en mode non-prioritaire du mécanisme de dribbling sont fournies par le Tableau A-2.

DBH ou DBL	Limites
000	Non asigné
001	8
010	16
011	24
100	32
101	40
110	48
111	56

Tableau A-2 - Limites du dribbing.

Les valeurs de DBH et DBL doivent suivre les règles suivantes :

- DBH != DBL.
- DBH > DBL.

De plus, les valeurs de DBH et DBL ne doivent pas être modifiées lorsque le mécanisme de dribbling est autorisé.

A.b.x Pointeur de routines du trap handler (TRAPBASE)

Ce registre contient l'adresse du pointeur de la routine courante dans la table des routines du gestionnaire d'interruptions (trap-handler). La table des routines contient l'adresse de début des routines de gestion des exceptions.

Le registre TRAPBASE est représenté par la Figure A-10, les champs qui le composent sont décrits par le.

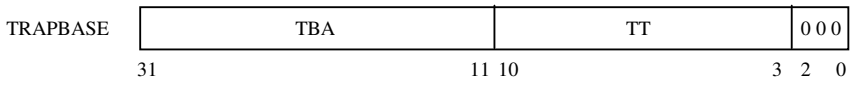


Figure A-10 - Pointeur de routines du trap-handler (TRAPBASE).

Champs	Descriptions
TBA	Base de la table de routines du trap-handler
TT	Index dans la table des routines du trap-handler

Tableau A-3 - Champs du registre TRAPBASE.

Les bits 2 à 0 du registre TRABASE sont en lecture seul et sont égaux à zéro.

A.c Les structures Java en mémoire

La structure utilisée pour stocker les programmes Java en mémoire est héritée du processeur Picojava-II de Sun. Toute cette partie fait donc référence à la documentation de Sun Microsystems [56].

A.c.i Les types primitifs

Les modèles JMS et JMQ supportent tous les types primitifs de la JMV plus le type "octet non-signé". Ces types sont décrits par le Tableau A-4.

Types	Descriptions
Unsigned byte	Entier 8-bits non-signé
byte	Entier 8-bits signé en complément à 2
Char	Entier 16-bits non-signé
Short	Entier 16-bits signé en complément à 2
Integer	Entier 32-bits signé en complément à 2
Long	Entier 64-bits signé en complément à 2
Float	Flottant 32-bits simple précisions IEEE 754
double	Flottant 32-bits double précisions IEEE 754

Tableau A-4 - Types primitifs

A.c.ii Références et entêtes

Une référence est un pointeur (non-modifiable) qui représente un objet ou un tableau. Une référence nulle prend la valeur 0x00000000. Le format d'une référence est représenté par la Figure A-11. Ce format réserve quatre bits dont la fonctionnalité est décrite par le Tableau A-5, ces bits sont masqués lors de l'utilisation de la référence en tant qu'adresse.

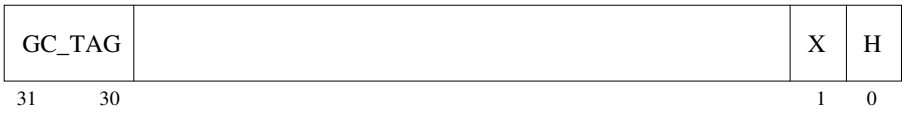


Figure A-11 - Format d'une référence.

Bit	Description
GC_TAG	2 bits utilisés par le Garbage colector
H	Bit (appelé "handle") qui indique si l'objet est référencé directement (0) ou indirectement (1).
X	Bit qui peut être utilisé par le logiciel de diverses façons.

Tableau A-5 - bits résevé d'une référence.

Une référence pointe sur l'entête d'un objet ou d'un tableau. Cet entête est un mot 32-bits qui contient l'adresse du vecteur de méthode (cf Annexe A.c.iv) de la classe correspondant à l'objet. Quatre bits de la référence sont réservés pour contenir des informations pour le Garbage Colector ou la synchronisation. Le bit L (LOCK) est utilisé comme verrou (Figure A-12). Les bits réservés sont masqués pour obtenir l'adresse du vecteur de méthode.

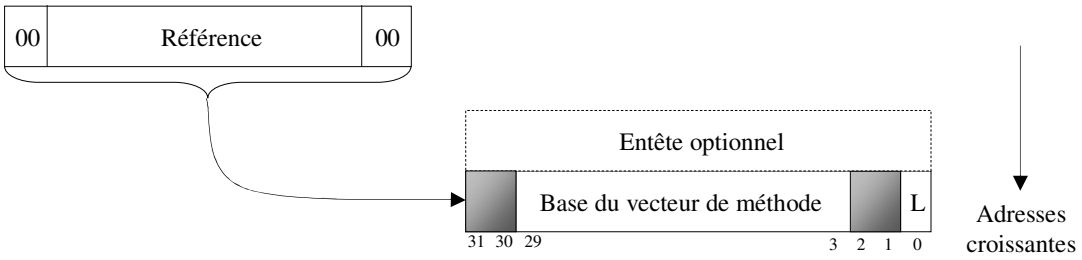


Figure A-12 - Format d'une entête d'objet ou de tableau.

A.c.iii Stockage des objets

Le stockage des objets peut être effectué de deux façons, soit directement, soit indirectement en fonction du bit handle (H) de la référence de l'objet.

Si H = 0, les objets sont stockés directement comme indiqué sur la Figure A-13.

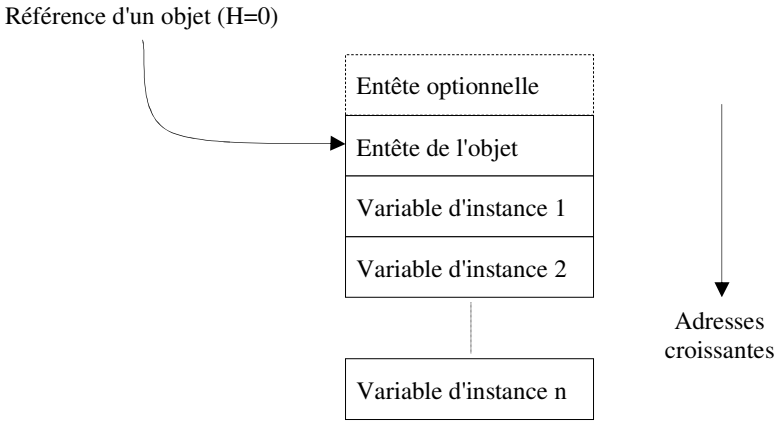


Figure A-13 - Structure d'un objet (H=0).

Si H = 1, les objets sont stockés indirectement comme indiqué sur la Figure A-14.

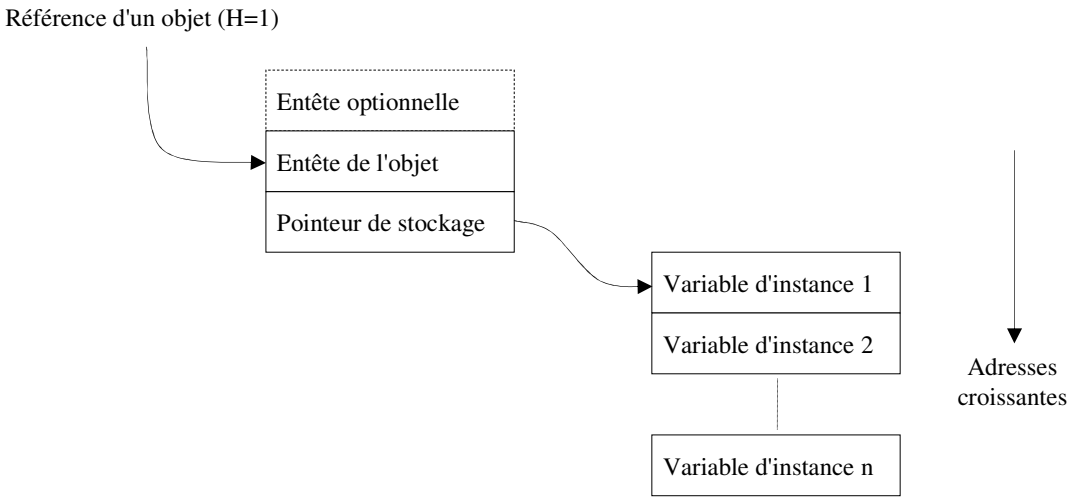


Figure A-14 - Structure d'un objet (H=1).

Les variables de type Long ou double sont stockées sur deux mots de 32-bits, les 32-bits de poids faible sont alors stockés à l'adresse basse. Les autres types sont stockés sur un mot de 32-bits.

A.c.iv Stockage des tableaux

Le stockage des tableaux est basé sur le même principe que celui des objets, les tableaux peuvent être stockés, soit directement, soit indirectement en fonction du bit handle (H) de la référence du tableau.

Si H = 0, les tableaux sont stockés directement comme indiqué sur la Figure A-15.

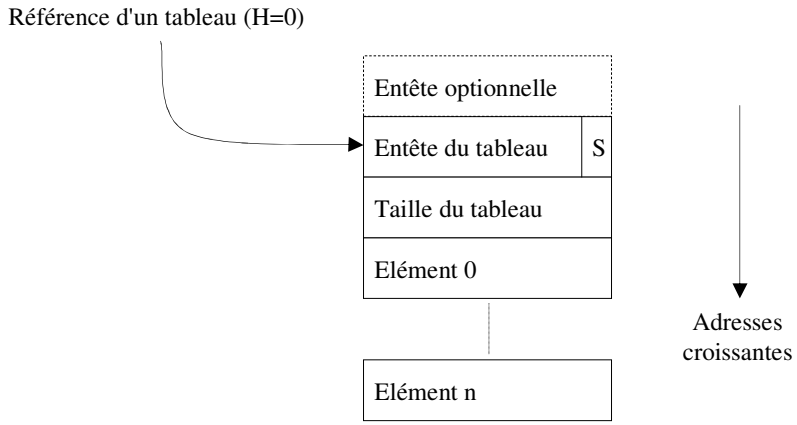


Figure A-15 - Structure d'un tableau (H=0).

Si H = 1, les tableaux sont stockés indirectement comme indiqué sur la Figure A-16.

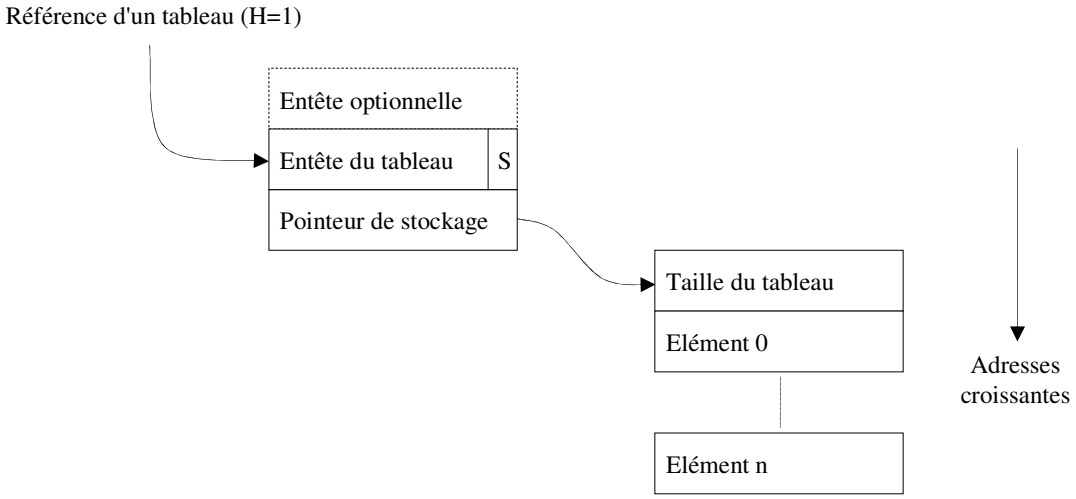


Figure A-16 - Structure d'un tableau (H=1).

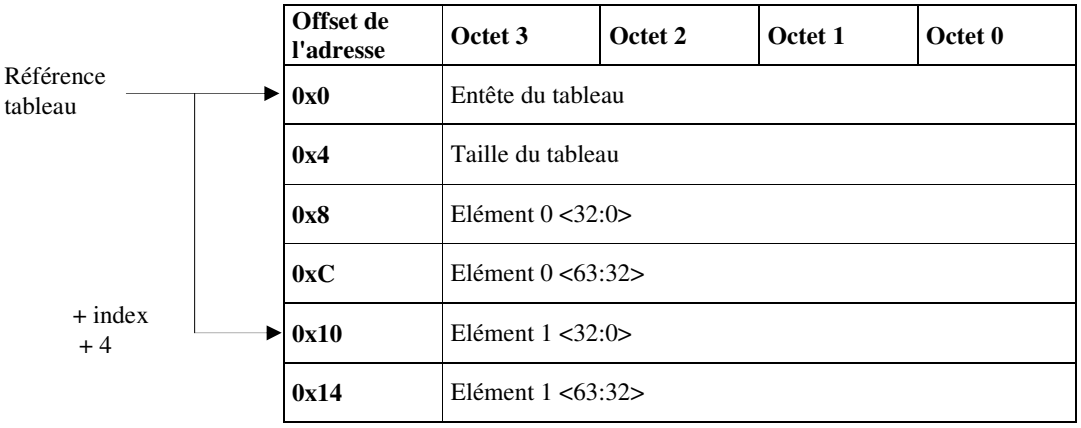
Les deux bits de poids faible de l'entête du tableau (S) contiennent la taille en octets des éléments compris dans le tableau (Tableau A-6).

S	Taille de la donnée
0	8 bits
1	16 bits
2	32 bits
3	64 bits

Tableau A-6 - Taille des éléments d'un tableau.

La structure des tableaux est différente selon le type de données présentes dans le tableau. Les structures de tableaux sont alignées sur 32-bits, les données plus petites sont compactées. Les exemples suivants sont basés sur des références dont le handle est à '0'.

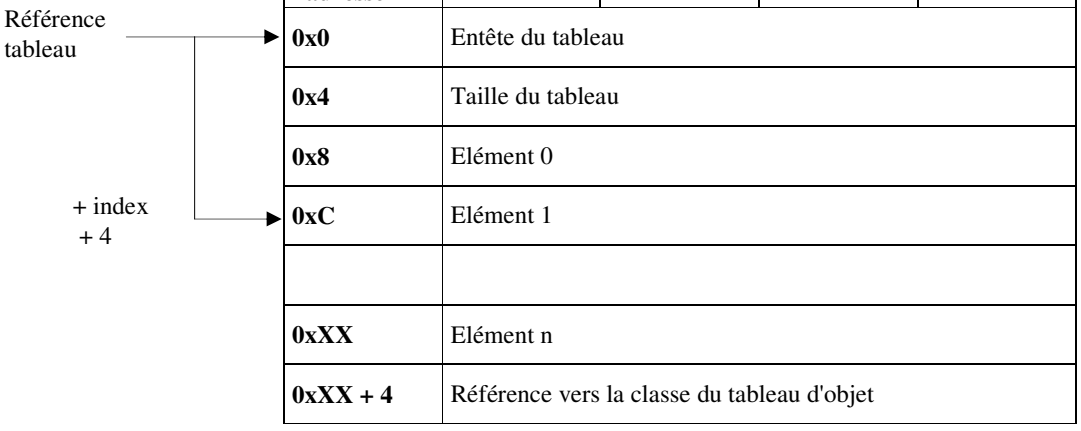
Tableau de long et de double



Offset de l'adresse	Octet 3	Octet 2	Octet 1	Octet 0
0x0	Entête du tableau			
0x4	Taille du tableau			
0x8	Elément 0 <32:0>			
0xC	Elément 0 <63:32>			
0x10	Elément 1 <32:0>			
0x14	Elément 1 <63:32>			

Figure A-17 - Structure des tableaux de long et de double.

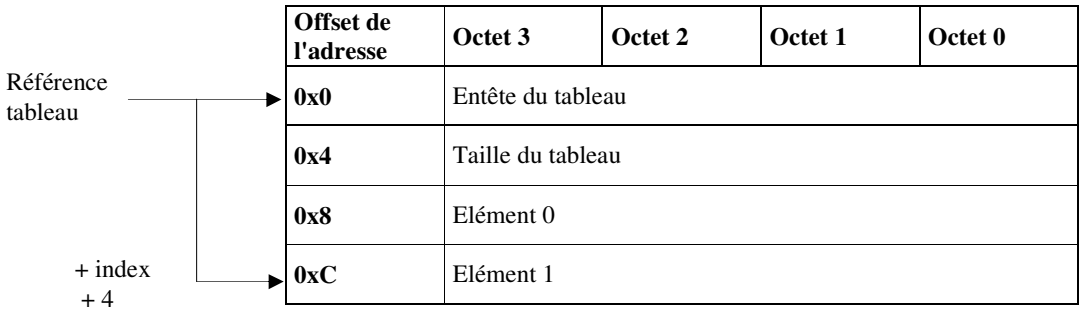
Tableau d'objet et de tableau



Offset de l'adresse	Octet 3	Octet 2	Octet 1	Octet 0
0x0	Entête du tableau			
0x4	Taille du tableau			
0x8	Elément 0			
0xC	Elément 1			
0xXX	Elément n			
0xXX + 4	Référence vers la classe du tableau d'objet			

Figure A-18 - Structure des tableaux d'objet et de tableau.

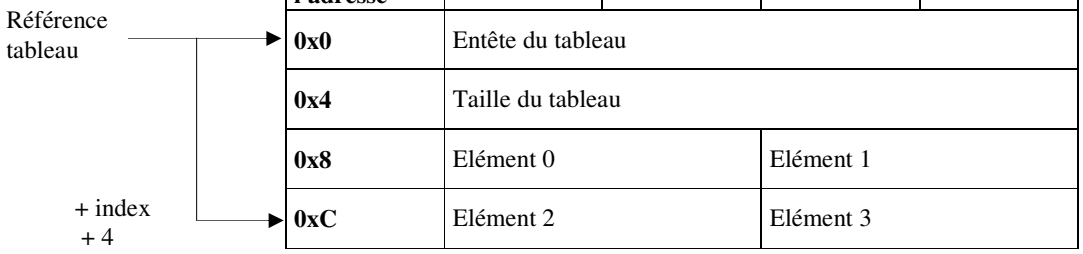
Tableau d'entier et de flottant



Offset de l'adresse	Octet 3	Octet 2	Octet 1	Octet 0
0x0	Entête du tableau			
0x4	Taille du tableau			
0x8	Elément 0			
0xC	Elément 1			

Figure A-19 - Structure de tableau d'entier et de flottant.

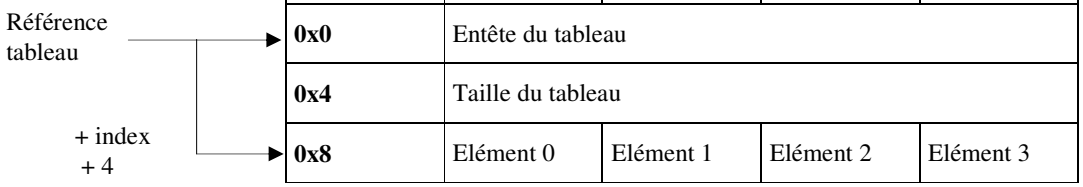
Tableau de caractère et d'entier court



Offset de l'adresse	Octet 3	Octet 2	Octet 1	Octet 0
0x0	Entête du tableau			
0x4	Taille du tableau			
0x8	Elément 0		Elément 1	
0xC	Elément 2		Elément 3	

Figure A-20 - Structure de tableau de caractère et d'entier court.

Tableau d'octet et de booléen



Offset de l'adresse	Octet 3	Octet 2	Octet 1	Octet 0
0x0	Entête du tableau			
0x4	Taille du tableau			
0x8	Elément 0	Elément 1	Elément 2	Elément 3

Figure A-21 - Structure de tableau d'octet et de booléen.

Le type de donnée contenu dans les tableaux de type primitif est inscrit à la fin de la structure à l'adresse du dernier élément +8, ce type est aligné sur 32-bit.

A.c.v Les vecteurs de méthodes

La base du vecteur de méthode est une entête d'objet ou de tableau pointant sur le début de la table des pointeurs de structures des méthodes, qui peut être invoquée par une référence, afin d'accéder à la méthode la plus spécialisée (en cas de surcharge) pour l'objet sur lequel elle s'applique.

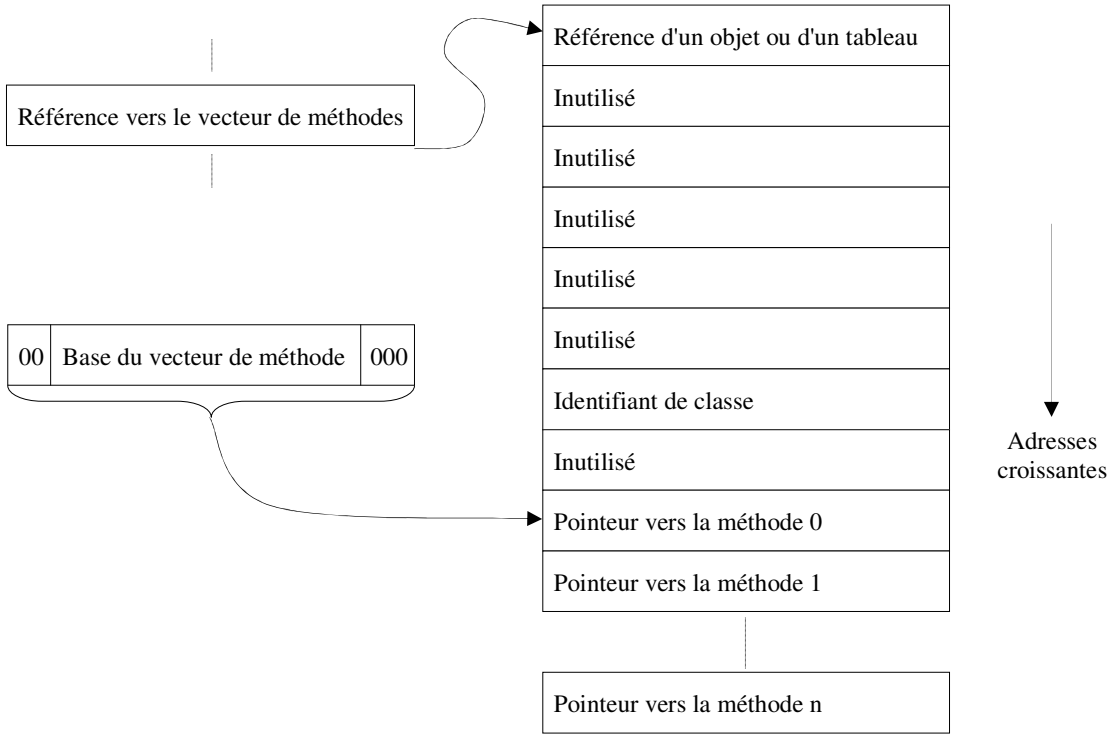


Figure A-22 - Structure du vecteur de méthode.

Aucune référence vers cette structure ne peut avoir le bit handle à '1'. L'identifiant de classe est sur 32-bit et est unique pour chaque classe.

A.c.vi Les méthodes

Chaque pointeur de méthode pointe sur la structure d'une méthode (Figure A-23). Les champs de cette structure sont sur 32-bits (sauf exceptions) et sont définit par le Tableau A-7.

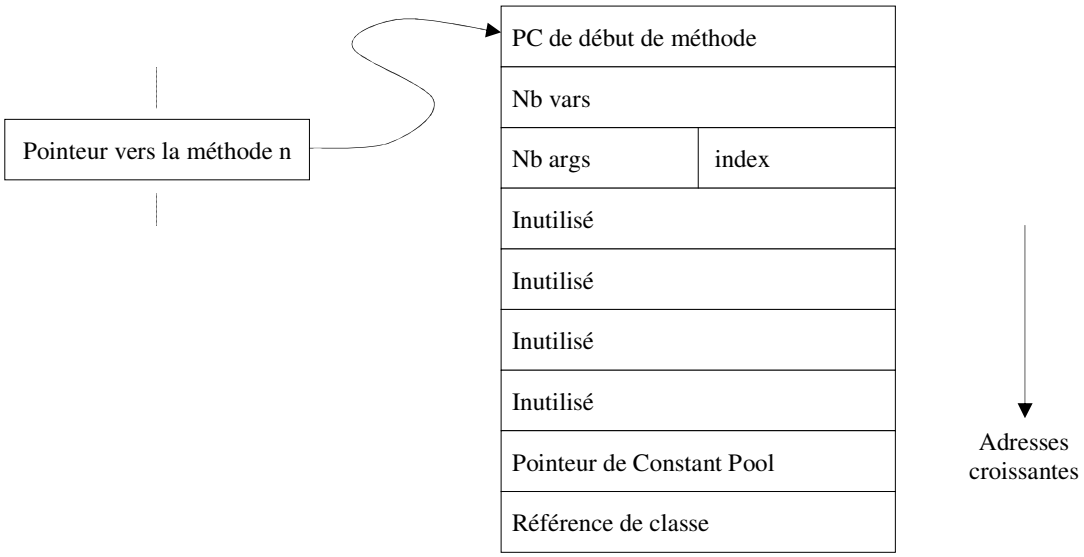


Figure A-23 - Structure de méthode.

Champs	**Descriptions**
PC de début de méthode	Adresse de la première instruction de la méthode
Nb vars	Nombre de variables locales (en octets), excluant les arguments
Nb args	Nombre d'arguments (en octets) sur 16-bits.
Index	Index 16-bits dans la table des entêtes de tableau
Pointeur de Constant Pool	Pointeur sur la table du Constant Pool
Référence de classe	Référence sur la structure de la classe à laquelle la méthode appartient. Le bit handle doit être à '0'.

Tableau A-7 - Champs de la struture d'une méthode.

A.c.vii Les classes

La Figure A-24 montre la structure d'une classe chargée en mémoire. Aucune référence vers cette structure ne peut avoir le bit Handle à '1'.

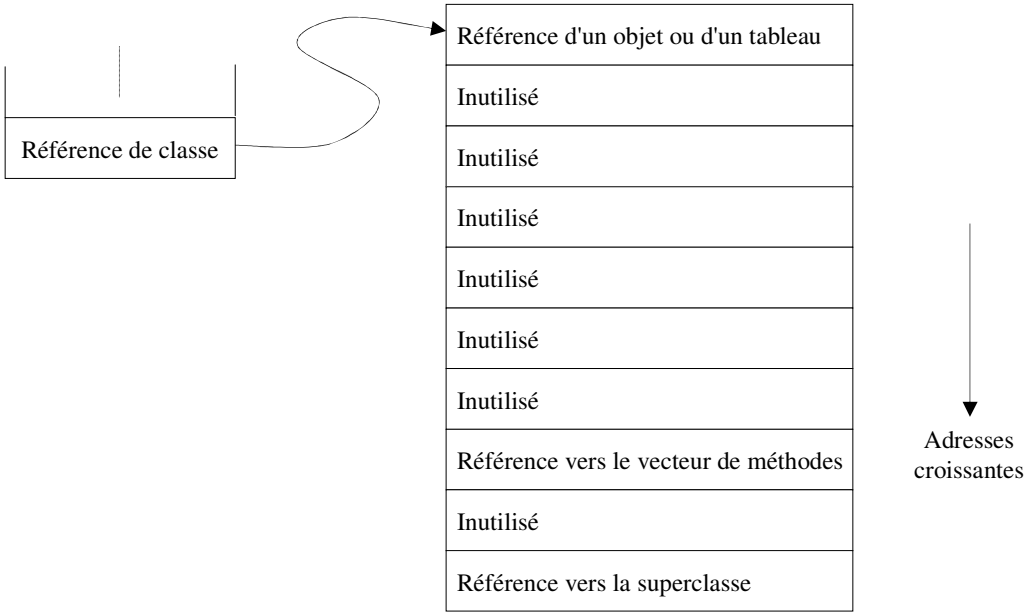


Figure A-24 - Structure d'une classe.

Cette structure contient une référence vers le vecteur des méthodes de la classe et une référence vers la classe mère (superclasse). Ces deux références ne peuvent pas avoir le bit handle à '1'.

A.c.viii Le CONSTANT POOL.

Chaque classe possède un Constant Pool qui contient des informations relatives à cette classe (type des champs ou des méthodes, référence de la classe, etc.). Chaque élément du Constant Pool est sur 32 bits. Le Constant pool est accédé via un pointeur et un index, sa structure est représentée par la Figure A-25.

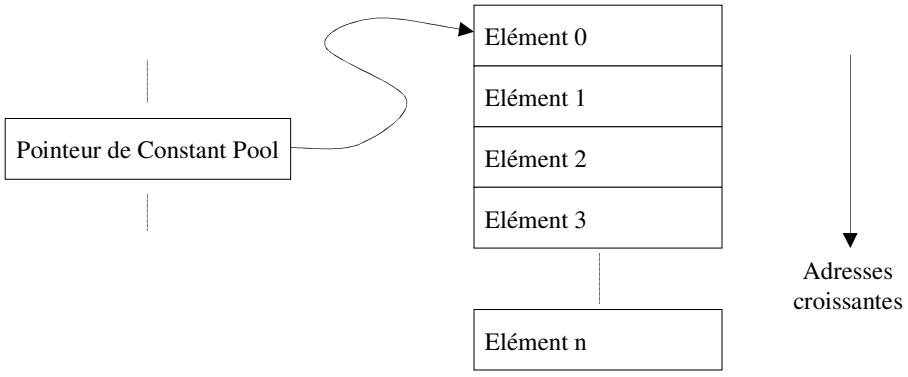


Figure A-25 - Structure du CONSTANT POOL.

A.c.ix Les outils.

Les structures décrites précédemment sont implanté dans la mémoire des modèles de test par le chargeur de classes développé par Sun Microsystems. Ce chargeur de classes permet de charger en mémoire des classes en s'appuyant sur la définition du format ".class" Java 1.3 [14] [57]. Nous avons modifié la mémoire virtuelle et ce chargeur afin de les rendrent compatibles avec le compilateur G++ et également pour que le chargeur de classes soit capable de traiter les instructions supplémentaires spécifiques à nos modèles.

Les routines logicielles du gestionnaire d'interruption (trap handler) sont écrites en assembleur Java, il est donc nécessaire de pouvoir assembler (jasm → class) et désassembler (class → jasm) ces routines. Nous utilisons pour assembler et désassembler les routines logicielles du trap handler respectivement les outils JASM et JDIS développer par Sun Microsystems pour le Picojava-II. Ces outils ont également été modifiés pour être compatibles avec G++ et intégrer les instructions spécifiques à nos modèles de simulations.

A.d Opcodes

Cette partie décrit les instructions des processeurs Picojava-II, JMS et JMQ. Le Tableau A-8 présente les instructions du Bytecode Java, tandis que le Tableau A-9 présente les instructions étendues des processeurs.

Les tableaux indiquent pour chaque instruction le groupe auquel elle appartient pour chaque processeur.

Les groupes pour chaque processeur sont les suivants :

- Picajava-II :
  - NF Non classable
  - LV Chargement d'une variable locale 32-bits ou d'une constante 32-bits au sommet de la pile d'exécution
  - OP Opération sur les deux données 32 bits du sommet de la pile
  - BG1 Opération sur la donnée 32 bits du sommet de la pile qui ne peut pas être suivie d'un rangement mémoire de variable locale
  - BG2 Opération sur les deux données 32 bits du sommet de la pile qui ne peut pas être suivie d'un rangement mémoire de variable locale
  - MEM Opération de rangement en mémoire d'une variable locale de 32-bits

- JMS :
  - NF Non classable
  - LV Chargement d'une variable locale 32-bits ou d'une constante 32-bits au sommet de la pile d'exécution
  - LV2 Chargement d'une variable locale 64 bits au sommet de la pile d'exécution
  - OP Opération sur les deux données 32 bits du sommet de la pile
  - OP2 Opération sur les deux données 64 bits du sommet de la pile
  - OP12 Opération sur la donnée 32 bits du sommet de la pile, renvoie ne donnée 64 bits
  - OP21 Opération sur la donnée 64 bits du sommet de la pile, renvoie une donnée 32 bits
  - BG1 Opération sur la donnée 32 bits du sommet de la pile qui ne peut pas être suivie d'un rangement mémoire de variable locale
  - BG2 Opération sur la donnée 64 bits ou les deux données 32 bits sommet de la pile qui ne peut pas être suivie d'un rangement mémoire de variable locale
  - MEM Opération de rangement en mémoire d'une variable locale de 32-bits
  - MEM2 Opération de rangement en mémoire d'une variable locale de 64-bits

- JMQ :
  - NF Non classable
  - LV Chargement d'une variable locale 32-bits ou 64bits, d'une constante 32-bits au sommet de la pile d'exécution ou d'un registre
  - OP Opération sur le sommet de la pile d'exécution
  - BG Opération sur le sommet de la pile d'exécution qui ne peut pas être suivie d'un rangement mémoire de variable locale
  - MEM Opération de rangement en mémoire d'une variable locale
  - ENV Instructions liées à l'environnement
  - LVOP Instructions appartement aux types LV et OP
  - LVBG Instructions appartement aux types LV et BG
  - BGENV Instructions appartement aux types BG et ENV

- Pour chaque processeur :
  - X non prévu
  - * non implémenté

Les cycles indiqués sont ceux du Picojava-II, toutes les modifications et commentaires pour les autres processeurs sont indiquées par ce qui suit :

1 Référence directe (H=0)
2 Référence indirecte (H=1)
3 Exécuté en logiciel (trap handler) optionnellement
4 Branchement non pris (sans prédiction) ou bien prédis
5 Branchement pris (sans prédiction) ou mal prédis (5 cycles pour JMS et JMQ à cause de l'étage supplémentaire du au cache non bloquant)
6 Dépend de l'index et des bornes du tableswitch. Si l'index est inférieur à la borne inférieure, tableswitch prend 10 cycles. Si l'index est supérieur à la borne supérieure, tableswitch prend 11 cycles. (11,12 et 16 cycles pour JMS et JMQ à cause de l'étage supplémentaire du au cache non bloquant)
7 Peut prendre le trap handler si la référence de l'objet n'est pas contenue dans le registre LOCKADDR
8 Peut prendre le trap handler si le matérielle demande le type de la superclasse de l'objet vérifié
9 Prend 1 cycle de moins pour les modèle JMS et JMQ car le chemin de données est 64 bits et les échanges avec la mémoire sont 64 bits
10 Demande un cycle de plus si la donnée chargée depuis la mémoire doit être utilisée par l'instruction suivante (prend toujours un cycle supplémentaire pour la JMQ, modification à effectuer)
11 Prend un cycle de moins pour la JMQ
12 Prend 2 cycles pour les modèles JMS et JMQ
13 Prend 5 cycles si prédis avec JMS; prend 1 cycle si prédis, sinon 5 (JMQ)

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
0 (0x00)	nop	1	Sans action	NF	NF	NF	1
1 (0x01)	aconst_null	1	Insère un objet nul	LV	LV	LV	1
2 (0x02)	iconst_m1	1	Insère la constante entière -1	LV	LV	LV	1
3 (0x03)	iconst_0	1	Insère la constante entière 0	LV	LV	LV	1
4 (0x04)	iconst_1	1	Insère la constante entière 1	LV	LV	LV	1
5 (0x05)	iconst_2	1	Insère la constante entière 2	LV	LV	LV	1
6 (0x06)	iconst_3	1	Insère la constante entière 3	LV	LV	LV	1
7 (0x07)	iconst_4	1	Insère la constante entière 4	LV	LV	LV	1
8 (0x08)	iconst_5	1	Insère la constante entière 5	LV	LV	LV	1
9 (0x09)	lconst_0	1	Insère la constante longue 0	NF	LV2	LV	2^9
10 (0x0a)	lconst_1	1	Insère la constante longue 1	NF	LV2	LV	2^9
11 (0x0b)	fconst_0	1	Insère la constante flottante 0	LV	LV	LV	1
12 (0x0c)	fconst_1	1	Insère la constante flottante 1	LV	LV	LV	1
13 (0x0d)	fconst_2	1	Insère la constante flottante 2	LV	LV	LV	1
14 (0x0e)	dconst_0	1	Insère la constante double 0	NF	LV2	LV	2^9
15 (0x0f)	dconst_1	1	Insère la constante double 1	NF	LV2	LV	2^9
16 (0x10)	bipush	2	Insère 1 octet d'entier	LV	LV	LV	1
17 (0x11)	sipush	3	Insère 2 octets d'entier	LV	LV	LV	1
18 (0x12)	ldc	2	Charge 1 constante depuis le constant pool	NF	NF	ENV	Trap
19 (0x13)	ldc_w	3	Charge 1 constante depuis le constant pool (index 16 bits)	NF	NF	ENV	Trap
20 (0x14)	ldc2_w	3	Charge 1 longue ou 1 double depuis le constant pool (index 16 bits)	NF	NF	ENV	Trap
21 (0x15)	iload	2	Charge 1 variable locale entière	LV	LV	LV	1
22 (0x16)	lload	2	Charge 1 variable locale longue	NF	LV2	LV	1
23 (0x17)	fload	2	Charge 1 variable locale flottante	LV	LV	LV	1
24 (0x18)	dload	2	Charge 1 variable locale double	NF	LV2	LV	1
25 (0x19)	aload	2	Charge 1 variable locale objet	LV	LV	LV	1
26 (0x1a)	iload_0	1	Charge la variable locale entière 0	LV	LV	LV	1
27 (0x1b)	iload_1	1	Charge la variable locale entière 1	LV	LV	LV	1

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
28 (0x1c)	iload_2	1	Charge la variable locale entière 2	LV	LV	LV	1
29 (0x1d)	iload_3	1	Charge la variable locale entière 3	LV	LV	LV	1
30 (0x1e)	lload_0	1	Charge la variable locale longue 0	NF	LV2	LV	2^{9}
31 (0x1f)	lload_1	1	Charge la variable locale longue 1	NF	LV2	LV	2^{9}
32 (0x20)	lload_2	1	Charge la variable locale longue 2	NF	LV2	LV	2^{9}
33 (0x21)	lload_3	1	Charge la variable locale longue 3	NF	LV2	LV	2^{9}
34 (0x22)	fload_0	1	Charge la variable locale flottante 0	LV	LV	LV	1
35 (0x23)	fload_1	1	Charge la variable locale flottante 1	LV	LV	LV	1
36 (0x24)	fload_2	1	Charge la variable locale flottante 2	LV	LV	LV	1
37 (0x25)	fload_3	1	Charge la variable locale flottante 3	LV	LV	LV	1
38 (0x26)	dload_0	1	Charge la variable locale double 0	NF	LV2	LV	2^{9}
39 (0x27)	dload_1	1	Charge la variable locale double 1	NF	LV2	LV	2^{9}
40 (0x28)	dload_2	1	Charge la variable locale double 2	NF	LV2	LV	2^{9}
41 (0x29)	dload_3	1	Charge la variable locale double 3	NF	LV2	LV	2^{9}
42 (0x2a)	aload_0	1	Charge la variable locale objet 0	LV	LV	LV	1
43 (0x2b)	aload_1	1	Charge la variable locale objet 1	LV	LV	LV	1
44 (0x2c)	aload_2	1	Charge la variable locale objet 2	LV	LV	LV	1
45 (0x2d)	aload_3	1	Charge la variable locale objet 3	LV	LV	LV	1
46 (0x2e)	iaload	1	Charge 1 entier depuis 1 tableau	BG2	BG2	OP	$3^{1,10}$ /$5^{2,10}$
47 (0x2f)	laload	1	Charge 1 long depuis 1 tableau	BG2	BG2	OP	$4^{1,9,10}$ /$6^{2,9,10}$
48 (0x30)	faload	1	Charge 1 flottant depuis 1 tableau	BG2	BG2	OP	$3^{1,10}$ /$5^{2,10}$
49 (0x31)	daload	1	Charge 1 double depuis 1 tableau	BG2	BG2	OP	$4^{1,9,10}$ /$6^{2,9,10}$
50 (0x32)	aaload	1	Charge 1 objet depuis 1 tableau	BG2	BG2	OP	$3^{1,10}$ /$5^{2,10}$
51 (0x33)	baload	1	Charge 1 byte depuis 1 tableau	BG2	BG2	OP	$3^{1,10}$ /$5^{2,10}$
52 (0x34)	caload	1	Charge 1 char depuis 1 tableau	BG2	BG2	OP	$3^{1,10}$ /$5^{2,10}$
53 (0x35)	saload	1	Charge 1 short depuis 1 tableau	BG2	BG2	OP	$3^{1,10}$ /$5^{2,10}$
54 (0x36)	istore	2	Range 1 entier dans 1 variable locale	MEM	MEM	MEM	1
55 (0x37)	lstore	2	Range 1 long dans 1 variable locale	NF	MEM2	MEM	2^{9}

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
56 (0x38)	fstore	2	Range 1 flottant dans 1 variable locale	MEM	MEM	MEM	1
57 (0x39)	dstore	2	Range 1 double dans 1 variable locale	NF	MEM2	MEM	2^{9}
58 (0x3a)	astore	2	Range 1 objet dans 1 variable locale	MEM	MEM	MEM	1
59 (0x3b)	istore_0	1	Range 1 entier dans la variable locale 0	MEM	MEM	MEM	1
60 (0x3c)	istore_1	1	Range 1 entier dans la variable locale 1	MEM	MEM	MEM	1
61 (0x3d)	istore_2	1	Range 1 entier dans la variable locale 2	MEM	MEM	MEM	1
62 (0x3e)	istore_3	1	Range 1 entier dans la variable locale 3	MEM	MEM	MEM	1
63 (0x3f)	lstore_0	1	Range 1 double dans la variable locale 0	NF	MEM2	MEM	2^{9}
64 (0x40)	lstore_1	1	Range 1 double dans la variable locale 1	NF	MEM2	MEM	2^{9}
65 (0x41)	lstore_2	1	Range 1 double dans la variable locale 2	NF	MEM2	MEM	2^{9}
66 (0x42)	lstore_3	1	Range 1 double dans la variable locale 3	NF	MEM2	MEM	2^{9}
67 (0x43)	fstore_0	1	Range 1 flottant dans la variable locale 0	MEM	MEM	MEM	1
68 (0x44)	fstore_1	1	Range 1 flottant dans la variable locale 1	MEM	MEM	MEM	1
69 (0x45)	fstore_2	1	Range 1 flottant dans la variable locale 2	MEM	MEM	MEM	1
70 (0x46)	fstore_3	1	Range 1 flottant dans la variable locale 3	MEM	MEM	MEM	1
71 (0x47)	dstore_0	1	Range 1 double dans la variable locale 0	NF	MEM2	MEM	2^{9}
72 (0x48)	dstore_1	1	Range 1 double dans la variable locale 1	NF	MEM2	MEM	2^{9}
73 (0x49)	dstore_2	1	Range 1 double dans la variable locale 2	NF	MEM2	MEM	2^{9}
74 (0x4a)	dstore_3	1	Range 1 double dans la variable locale 3	NF	MEM2	MEM	2^{9}
75 (0x4b)	astore_0	1	Range 1 objet dans la variable locale 0	MEM	MEM	MEM	1
76 (0x4c)	astore_1	1	Range 1 objet dans la variable locale 1	MEM	MEM	MEM	1
77 (0x4d)	astore_2	1	Range 1 objet dans la variable locale 2	MEM	MEM	MEM	1
78 (0x4e)	astore_3	1	Range 1 objet dans la variable locale 3	MEM	MEM	MEM	1
79 (0x4f)	iastore	1	Range dans 1 tableau d'entier	BG2	BG2	BG	5^{1} /7^{2}
80 (0x50)	lastore	1	Range dans 1 tableau de long	BG2	BG2	BG	6^{1} /$8^{2,9}$
81 (0x51)	fastore	1	Range dans 1 tableau de flottant	BG2	BG2	BG	5^{1} /7^{2}
82 (0x52)	dastore	1	Range dans 1 tableau de double	BG2	BG2	BG	6^{1} /$8^{2,9}$
83 (0x53)	aastore	1	Range dans 1 tableau d'objet	BG2	BG2	BG	5^{1}/ 7^{2}

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
84 (0x54)	bastore	1	Range dans 1 tableau de byte	BG2	BG2	BG	5[1] /7[2]
85 (0x55)	castore	1	Range dans 1 tableau de char	BG2	BG2	BG	5[1] /7[2]
86 (0x56)	sastore	1	Range dans 1 tableau de short	BG2	BG2	BG	5[1] /7[2]
87 (0x57)	pop	1	Enlève le sommet	NF	NF	NF	1
88 (0x58)	pop2	1	Enlève 2 mots du sommet	NF	NF	NF	1
89 (0x59)	dup	1	Duplique le sommet	NF	NF	NF	1
90 (0x5a)	dup_x1	1	Duplique le sommet et le place 2 mots en dessous	BG2	BG2	NF	3
91 (0x5b)	dup_x2	1	Duplique le sommet et le place 3 mots en dessous	BG2	BG2*	NF	4
92 (0x5c)	dup2	1	Duplique 2 mots du sommet	NF	NF	NF	2
93 (0x5d)	dup2_x1	1	Duplique 2 mots du sommet et les place 3 mots en dessous	BG2	BG2*	NF	5
94 (0x5e)	dup2_x2	1	Duplique 2 mots du sommet et les place 4 mots en dessous	BG2	BG2*	NF	6
95 (0x5f)	swap	1	Intervertie le sommet et le sous sommet	BG2	BG2	OP	2[11]
96 (0x60)	iadd	1	Addition entière	OP	OP	OP	1
97 (0x61)	ladd	1	Addition longue	NF	OP2	OP	2[9]
98 (0x62)	fadd	1	Addition flottante	OP	OP	OP	3
99 (0x63)	dadd	1	Addition double	NF	OP2	OP	11
100 (0x64)	isub	1	Soustraction entière	OP	OP	OP	1
101 (0x65)	lsub	1	Soustraction longue	NF	OP2	OP	2[9]
102 (0x66)	fsub	1	Soustraction flottante	OP	OP	OP	3
103 (0x67)	dsub	1	Soustraction double	NF	OP2	OP	14
104 (0x68)	imul	1	Multiplication entière	OP	OP	OP	2-18[12]
105 (0x69)	lmul	1	Multiplication longue	NF	NF	ENV	Trap
106 (0x6a)	fmul	1	Multiplication flottante	OP	OP	OP	3
107 (0x6b)	dmul	1	Multiplication double	NF	OP2	OP	14
108 (0x6c)	idiv	1	Division entière	OP	OP	OP	32
109 (0x6d)	ldiv	1	Division longue	NF	NF	ENV	Trap
110 (0x6e)	fdiv	1	Division flottante	OP	OP	OP	30
111 (0x6f)	ddiv	1	Division double	NF	OP2	OP	60
112 (0x70)	irem	1	Reste division entière	OP	OP	OP	32
113 (0x71)	lrem	1	Reste division longue	NF	NF	ENV	Trap
114 (0x72)	frem	1	Reste division flottante	OP	OP*	OP*	<200
115 (0x73)	drem	1	Reste division double	NF	OP2*	OP*	< 2000[3]

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
116 (0x74)	ineg	1	Négation entière	BG1	BG1	OP	1
117 (0x75)	lneg	1	Négation longue	NF	BG2	OP	2^9
118 (0x76)	fneg	1	Négation flottante	BG1	BG1	OP	1
119 (0x77)	dneg	1	Négation double	NF	BG2	OP	2^9
120 (0x78)	ishl	1	Décalage à droite entier	OP	OP	OP	1
121 (0x79)	lshl	1	Décalage à droite long	NF	OP2	OP	2^9
122 (0x7a)	ishr	1	Décalage arithmétique à gauche entier	OP	OP	OP	1
123 (0x7b)	lshr	1	Décalage arithmétique à gauche long	NF	OP2	OP	2^9
124 (0x7c)	iushr	1	Décalage logique à gauche entier	OP	OP	OP	1
125 (0x7d)	lushr	1	Décalage logique à gauche long	NF	OP2	OP	2^9
126 (0x7e)	iand	1	ET bit à bit entier	OP	OP	OP	1
127 (0x7f)	land	1	ET bit à bit long	NF	OP2	OP	2^9
128 (0x80)	ior	1	OU bit à bit entier	OP	OP	OP	1
129 (0x81)	lor	1	OU bit à bit long	NF	OP2	OP	2^9
130 (0x82)	ixor	1	OU exclusif bit à bit entier	OP	OP	OP	1
131 (0x83)	lxor	1	OU exclusif bit à bit long	OP	OP	OP	2^9
132 (0x84)	iinc	3	Incrémente une variable locale entière	NF	NF	NF	1
133 (0x85)	i2l	1	Conversion : entier vers long	NF	OP12	OP	1
134 (0x86)	i2f	1	Conversion : entier vers flottant	NF	BG1	OP	6
135 (0x87)	i2d	1	Conversion : entier vers double	NF	OP12	OP	3
136 (0x88)	l2i	1	Conversion : long vers entier	NF	OP21	OP	1
137 (0x89)	l2f	1	Conversion : long vers flottant	NF	OP21	OP	8
138 (0x8a)	l2d	1	Conversion : long vers double	NF	BG2	OP	6
139 (0x8b)	f2i	1	Conversion : flottant vers entier	NF	BG1	OP	4
140 (0x8c)	f2l	1	Conversion : flottant vers long	NF	OP12	OP	5
141 (0x8d)	f2d	1	Conversion : flottant vers double	NF	OP12	OP	3
142 (0x8e)	d2i	1	Conversion : double vers entier	NF	OP21	OP	5
143 (0x8f)	d2l	1	Conversion : double vers long	NF	BG2	OP	5
144 (0x90)	d2f	1	Conversion : double vers flottant	NF	OP21	OP	7
145 (0x91)	i2b	1	Conversion : entier vers byte	BG1	BG1	OP	1
146 (0x92)	i2c	1	Conversion : entier vers char	BG1	BG1	OP	1

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
147 (0x93)	i2s	1	Conversion : entier vers short	BG1	BG1	OP	1
148 (0x94)	lcmp	1	Compare un long à 0	NF	BG2	OP	2^9
149 (0x95)	fcmpl	1	Compare un flottant à 0 (-1 si NAN)	OP	OP	OP	7
150 (0x96)	fcmpg	1	Compare un flottant à 0 (1 si NAN)	OP	OP	OP	7
151 (0x97)	dcmpl	1	Compare un double à 0 (-1 si NAN)	NF	OP2	OP	7
152 (0x98)	dcmpg	1	Compare un double à 0 (1 si NAN)	NF	OP2	OP	7
153 (0x99)	ifeq	3	Saut si = 0	BG1	BG1	BGENV	1^4 /4^5
154 (0x9a)	ifne	3	Saut si ≠ 0	BG1	BG1	BGENV	1^4 /4^5
155 (0x9b)	iflt	3	Saut si < 0	BG1	BG1	BGENV	1^4 /4^5
156 (0x9c)	ifge	3	Saut si ≥ 0	BG1	BG1	BGENV	1^4 /4^5
157 (0x9d)	ifgt	3	Saut si > 0	BG1	BG1	BGENV	1^4 /4^5
158 (0x9e)	ifle	3	Saut si ≤ 0	BG1	BG1	BGENV	1^4 /4^5
159 (0x9f)	if_icmpeq	3	Compare les deux entrées du sommet de la pile : Saut si =	BG2	BG2	BGENV	1^4 /4^5
160 (0xa0)	if_icmpne	3	Compare les deux entrées du sommet de la pile : Saut si ≠	BG2	BG2	BGENV	1^4 /4^5
161 (0xa1)	if_icmplt	3	Compare les deux entrées du sommet de la pile : Saut si <	BG2	BG2	BGENV	1^4 /4^5
162 (0xa2)	if_icmpge	3	Compare les deux entrées du sommet de la pile : Saut si ≥	BG2	BG2	BGENV	1^4 /4^5
163 (0xa3)	if_icmpgt	3	Compare les deux entrées du sommet de la pile : Saut si >	BG2	BG2	BGENV	1^4 /4^5
164 (0xa4)	if_icmple	3	Compare les deux entrées du sommet de la pile : Saut si ≤	BG2	BG2	BGENV	1^4 /4^5
165 (0xa5)	if_acmpeq	3	Compare les deux objets du sommet de la pile : Saut si =	BG2	BG2	BGENV	1^4 /4^5
166 (0xa6)	if_acmpne	3	Compare les deux objets du sommet de la pile : Saut si ≠	BG2	BG2	BGENV	1^4 /4^5
167 (0xa7)	goto	3	Saut	NF	NF	ENV	$4^5(1^4)$
168 (0xa8)	jsr	3	Saut vers sous programme	NF	NF	ENV	$4^5(1^4)$
169 (0xa9)	ret	2	Retour de sous programme	NF	NF*	ENV*	$4^5(1^4)$
170 (0xaa)	tableswitch	--	Accès à la table de saut par index et saute	NF	NF	BGENV	15^6

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
171 (0xab)	lookupswitch	--	Accès à la table de saut par correspondance et saute	NF	NF	ENV*	Trap
172 (0xac)	ireturn	1	Retourne un entier depuis une méthode	BG1	BG1	ENV	8^{13}
173 (0xad)	lreturn	1	Retourne un long depuis une méthode	NF	BG2	ENV	8^{13}
174 (0xae)	freturn	1	Retourne un flottant depuis une méthode	BG1	BG1	ENV	8^{13}
175 (0xaf)	dreturn	1	Retourne un double depuis une méthode	NF	BG2	ENV	8^{13}
176 (0xb0)	areturn	1	Retourne un objet depuis une méthode	BG1	BG1	ENV	8^{13}
177 (0xb1)	return	1	Retour de méthode sans variable	NF	NF	ENV	8^{13}
178 (0xb2)	getstatic	3	Lit 1 champ statique	NF	NF	ENV	Trap
179 (0xb3)	putstatic	3	Ecrit 1 champ statique	NF	NF	ENV	Trap
180 (0xb4)	getfield	3	Lit 1 champ	NF	NF	ENV	Trap
181 (0xb5)	putfield	3	Ecrit 1 champ	NF	NF	ENV	Trap
182 (0xb6)	invokevirtual	3	Invoque une méthode basée sur un objet	NF	NF	ENV	Trap
183 (0xb7)	invokespecial	3	Invoque une méthode non basée sur un objet	NF	NF	ENV	Trap
184 (0xb8)	invokestatic	3	Invoque une méthode statique	NF	NF	ENV	Trap
185 (0xb9)	invokeinterface	5	Invoque une interface	NF	NF	ENV	Trap
186 (0xba)	Non défini						
187 (0xbb)	new	3	Crée un nouvel objet	NF	NF	ENV	Trap
188 (0xbc)	newarray	2	Alloue un nouveau tableau	NF	NF	ENV	Trap
189 (0xbd)	anewarray	3	Alloue un nouveau tableau d'objet	NF	NF	ENV	Trap
190 (0xbe)	arraylenth	1	Donne la longueur du tableau	BG1	BG1	OP	$1^{1,10}$ /$3^{2,10}$
191 (0xbf)	athrow	1	Génère une exception	NF	NF*	ENV*	Trap
192 (0xc0)	checkcast	3	Vérifie si l'objet est d'un type donné	NF	NF	ENV	Trap
193 (0xc1)	instanceof	3	Détermine si l'objet est d'un type donné	NF	NF	ENV*	Trap
194 (0xc2)	moniterenter	1	Entre dans une région de code monitorée	NF	NF*	NF*	3^{7}
195 (0xc3)	moniterexit	1	Sort d'une région de code monitorée	NF	NF*	NF*	2^{7}
196 (0xc4)	wide	4/6	Etant l'index des variables locales par des octets supplémentaires	NF	NF	ENV*	Trap
197 (0xc5)	multianewarray	4	Alloue un nouveau tableau multidimensionnel	NF	NF	ENV	Trap
198 (0xc6)	ifnull	3	Test si nul	BG1	BG1	BGENV	1^{4} /4^{5}

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
199 (0xc7)	ifnonnull	3	Test si non nul	BG1	BG1	BGENV	1^{4} /4^{5}
200 (0xc8)	goto_w	5	Saut (index long)	NF	NF	ENV	$4^{5}(1^{4})$
201 (0xc9)	jsr_w	5	Saut vers sous programme (index long)	NF	NF	ENV	$4^{5}(1^{4})$
202 (0xca)	breakpoint	1	Appel le gestionnaire de point d'arrêt	NF	NF*	ENV*	Trap
203 (0xcb)	ldc_quick	2	Charge 1 constante depuis le constant pool	NF	NF	LVOP	1^{10}
204 (0xcc)	ldc_w_quick	3	Charge 1 constante depuis le constant pool (index 16 bits)	NF	NF	LVOP	1^{10}
205 (0xcd)	ldc2_w_quick	3	Charge 1 longue ou 1 double depuis le constant pool (index 16 bits)	NF	NF	LVOP	$2^{9,10}$
206 (0xce)	getfield_quick	3	Lit 1 champ 32 bits	BG1	BG1	OP	$1^{1,10}$ /$4^{2,10}$
207 (0xcf)	putfield_quick	3	Ecrit 1 champ 32 bits	BG2	BG2	BG	$1^{1,10}$ /$4^{2,10}$
208 (0xd0)	getfield2_quick	3	Lit 1 champ 64 bits	BG1	BG1	OP	$2^{1,9,10}$ /$5^{2,9,10}$
209 (0xd1)	putfield2_quick	3	Ecrit 1 champ 64 bits	NF	BG2	BG	$2^{1,9,10}$ /$5^{2,9,10}$
210 (0xd2)	getstatic_quick	3	Lit 1 champ statique 32 bits	NF	NF	LVOP	3^{10}
211 (0xd3)	putstatic_quick	3	Ecrit 1 champ statique 32 bits	BG1	BG1	LVBG	3
212 (0xd4)	getstatic2_quick	3	Lit 1 champ statique 64 bits	NF	NF	LVOP	$4^{9,10}$
213 (0xd5)	putstatic2_quick	3	Ecrit 1 champ statique 64 bits	NF	BG2	LVBG	4^{9}
214 (0xd6)	invokevirtual_quick	3	Invoque une méthode basée sur un objet	NF	NF	ENV	15
215 (0xd7)	invokenonvirtual_quick	3	Invoque une méthode non basée sur un objet	NF	NF	ENV	13
216 (0xd8)	invokesuper_quick	3	Invoque une méthode (constructeur)	NF	NF	ENV	21
217 (0xd9)	invokestatic_quick	3	Invoque une méthode statique	NF	NF	ENV	11
218 (0xda)	invokeinterface_quick	3	Invoque une interface	NF	NF	ENV	Trap
219 (0xdb)	Non défini						
220 (0xdc)	aastore_quick	1	Range dans 1 tableau d'objet (sans vérification de type)	BG2	BG2*	NF*	6^{1} /8^{2}
221 (0xdd)	new_quick	3	Crée un nouvel objet	NF	NF	ENV	Trap
222 (0xde)	anewarray_quick	3	Alloue un nouveau tableau d'objet	NF	NF	ENV	Trap
223 (0xdf)	multianewarray_quick	3	Alloue un nouveau tableau multidimensionnel	NF	NF	ENV	Trap
224 (0xe0)	checkcast_quick	3	Vérifie si l'objet est d'un type donné	NF	NF	NF	6^{8}

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
225 (0xe1)	instanceof_quick	3	Détermine si l'objet est d'un type donné	NF	NF*	NF*	7^{8}
226 (0xe2)	invokevirtual_quick_w	3	Invoque une méthode basée sur un objet	NF	NF*	ENV*	19
227 (0xe3)	getfield_quick_w	3	Lit 1 champ 32 bits (index long)	NF	NF	ENV*	Trap
228 (0xe4)	putfield_quick_w	3	Ecrit 1 champ 32 bits (index long)	NF	NF	ENV*	Trap
229 (0xe5)	nonnull_quick	1	Enlève l'objet du sommet et prend le trap handler si l'objet est nul	BG1	BG1*	BGENV*	1
230 (0xe6)	agetfield_quick	3	Lit la référence d'un objet	BG1	BG1*	OP*	$1^{1,10}$ /$4^{2,10}$
231 (0xe7)	aputfield_quick	3	Ecrit la référence d'un objet avec vérification du garbage collector	BG2	BG2*	BG*	$1^{1,10}$ /$4^{2,10}$
232 (0xe8)	agetstatic_quick	3	Lit une référence statique d'une classe	NF	NF	LVOP	3^{10}
233 (0xe9)	aputstatic_quick	3	Ecrit une référence statique d'une classe avec vérification du garbage collector	BG1	BG1*	LVBG*	3
234 (0xea)	aldc_quick	2	Charge 1 référence depuis le constant pool	NF	NF	LVOP	1^{10}
235 (0xeb)	aldc_w_quick	3	Charge 1 référence depuis le constant pool	NF	NF	LVOP	1^{10}
236 (0xec)	exit_sync_method	1	Saut à l'instruction de retour d'une méthode synchronisée	NF	NF*	NF*	6
237 (0xed)	sethi	3	Ecrit les 16 bits de poids fort de l'entrée du sommet de la pile	BG1	BG1	OP	1
238 (0xee)	load_word_index	3	Charge un mot 32 bits à partir d'une adresse et d'un index	NF	NF	LVOP	1^{10}
239 (0xef)	load_short_index	3	Charge un short à partir d'une adresse et d'un index	NF	NF	LVOP	1^{10}
240 (0xf0)	load_char_index	3	Charge un char à partir d'une adresse et d'un index	NF	NF	LVOP	1^{10}
241 (0xf1)	load_byte_index	3	Charge un byte à partir d'une adresse et d'un index	NF	Nf	LVOP	1^{10}
242 (0xf2)	load_ubyte_index	3	Charge un ubyte à partir d'une adresse et d'un index	NF	NF	LVOP	1^{10}
243 (0xf3)	store_word_index	3	Range un mot 32 bits à partir d'une adresse et d'un index	NF	NF	LVOP	1
244 (0xf4)	nastore_word_index	3	Range un mot 32 bits à partir d'une adresse et d'un index (non caché)	NF	NF	LVBG	1
245 (0xf5)	store_short_index	3	Range un short à partir d'une adresse et d'un index	NF	NF	LVBG	1

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
246 (0xf6)	store_byte_index	3	Range un octet à partir d'une adresse et d'un index	NF	NF	LVBG	1
247 (0xf7) à 254 (0xfe)	Non défini						

Tableau A-8 - Liste et description des mnémoniques du Bytecode Java.

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
255 (0xff)	Le premier opcode signifie qu'il s'agit d'une instruction étendue, le second contient l'instruction.						
0 (0x00)	load_ubyte	2	Charge 1 octet non signé depuis la mémoire	BG1	BG1	OP	1^{10}
1 (0x01)	load_byte	2	Charge 1 octet depuis la mémoire	BG1	BG1	OP	1^{10}
2 (0x02)	load_char	2	Charge 2 octets non signés depuis la mémoire	BG1	BG1	OP	1^{10}
3 (0x03)	load_short	2	Charge 2 octets depuis la mémoire	BG1	BG1	OP	1^{10}
4 (0x04)	load_word	2	Charge 4 octets depuis la mémoire	BG1	BG1	OP	1^{10}
5 (0x05)	priv_ret_from_trap	2	Retour de sous programme en mode privilège	NF	NF	ENV	8
6 (0x06)	priv_read_dcache_tag	2	Lit un tag du cache de données en mode privilège	NF	X	X	1
7 (0x07)	priv_read_dcache_data	2	Lit une donnée du cache de données en mode privilège	NF	X	X	1
8 (0x08)	Non défini						
9 (0x09)	Non défini						
10 (0x0a)	load_char_oe	2	Charge 2 octets non signés depuis la mémoire (endian swap)	BG1	BG1	OP	1^{10}
11 (0x0b)	load_short_oe	2	Charge 2 octets depuis la mémoire (endian swap)	BG1	BG1	OP	1^{10}
12 (0x0c)	load_word_oe	2	Charge 4 octets depuis la mémoire (endian swap)	BG1	BG1	OP	1^{10}
13 (0x0d)	return0	2	Retour sans valeur d'une fonction appelé par l'instruction call	BG2	BG2*	BGENV*	6
14 (0x0e)	priv_read_icache_tag	2	Lit un tag du cache instructions en mode privilège	NF	X	X	2
15 (0x0f)	priv_read_icache_data	2	Lit une donnée du cache instructions en mode privilège	NF	X	X	2
16 (0x10)	ncload_ubyte	2	Charge 1 octet non signé depuis la mémoire (non caché)	BG1	BG1	OP	1^{10}

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
17 (0x11)	ncload_byte	2	Charge 1 octet depuis la mémoire (non caché)	BG1	BG1	OP	1^{10}
18 (0x12)	ncload_char	2	Charge 2 octets non signés depuis la mémoire (non caché)	BG1	BG1	OP	1^{10}
19 (0x13)	ncload_short	2	Charge 2 octets depuis la mémoire (non caché)	BG1	BG1	OP	1^{10}
20 (0x14)	ncload_word	2	Charge 4 octets depuis la mémoire (non caché)	BG1	BG1	OP	1^{10}
21 (0x15)	iucmp	2	Compare deux entiers non signés	OP	OP*	OP*	1
22 (0x16)	priv_powerdown	2	Entre en mode standby en mode privilège	NF	X	X	1
23 (0x17)	cache_invalidate	2	Invalide une ligne du cache	BG1	X	X	1
24 (0x18)	Non défini						
25 (0x19)	Non défini						
26 (0x1a)	ncload_char_oe	2	Charge 2 octets non signés depuis la mémoire (non caché endian swap)	BG1	BG1	OP	1^{10}
27 (0x1b)	ncload_short_oe	2	Charge 2 octets depuis la mémoire (non caché endian swap)	BG1	BG1	OP	1^{10}
28 (0x1c)	ncload_word_oe	2	Charge 4 octets depuis la mémoire (non caché endian swap)	BG1	BG1	OP	1^{10}
29 (0x1d)	return1	2	Retour une donnée 32 bits d'une fonction appelé par l'instruction call	BG2	BG2*	BGENV *	7
30 (0x1e)	cache_flush	2	Invalide une ligne du cache de données	BG1	BG1	BGENV	1
31 (0x1f)	cache_flush_index	2	Invalide une ligne du cache de données (pas de vérification du tag)	BG1	X	X	1
32 (0x20)	store_byte	2	Ecrit 1 octet en mémoire	BG2	BG2	BG	1
33 (0x21)	Non défini						
34 (0x22)	store_short	2	Ecrit 2 octets en mémoire	BG2	BG2	BG	1
35 (0x23)	Non défini						
36 (0x24)	store_word	2	Ecrit 4 octets en mémoire	BG2	BG2	BG	1
37 (0x25)	soft_trap	2	Invoque une routine logicielle	NF	NF	ENV*	Trap
38 (0x26)	priv_write_dcache_tag	2	Ecrit un tag dans le cache de données en mode privilège	NF	X	X	1
39 (0x27)	priv_write_dcache_data	2	Ecrit une donnée dans le cache de données en mode privilège	NF	X	X	1
40 (0x28)	Non défini						
41 (0x29)	Non défini						

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
42 (0x2a)	store_short_oe	2	Ecrit 2 octets en mémoire (endian swap)	BG2	BG2	BG	1
43 (0x2b)	Non défini						
44 (0x2c)	store_whord_oe	2	Ecrit 4 octets en mémoire (endian swap	BG2	BG2	BG	1
45 (0x2d)	return2	2	Retour une donnée 64 bits d'une fonction appelé par l'instruction call	BG2	BG2	BG	7
46 (0x2e)	priv_write_dcache_tag	2	Ecrit un tag dans le cache instrution en mode privilège	NF	X	X	1
47 (0x2f)	priv_write_dcache_data	2	Ecrit une donnée dans le cache instruction en mode privilège	NF	X	X	1
48 (0x30)	ncstore_byte	2	Ecrit 1 octet en mémoire (non caché)	BG2	BG2	BG	1
49 (0x31)	Non défini						
50 (0x32)	ncstore_short	2	Ecrit 2 octets en mémoire (non caché)	BG2	BG2	BG	1
51 (0x33)	Non défini						
52 (0x34)	ncstore_word	2	Ecrit 4 octets en mémoire (non caché)	BG2	BG2	BG	1
53 (0x35)	Non défini						
54 (0x36)	priv_reset	2	Génère une réinitialisation logicielle (mode privilège)	NF	NF*	ENV*	1
55 (0x37)	get_current_class	2	Insère un pointeur sur classe de la méthode courante	NF	NF	NF	3
56 (0x38)	Non défini						
57 (0x39)	Non défini						
58 (0x3a)	ncstore_short_oe	2	Ecrit 2 octets en mémoire (non caché endian swap)	BG2	BG2	BG	1
59 (0x3b)	Non défini						
60 (0x3c)	ncstore_word_oe	2	Ecrit 4 octets en mémoire (non caché endian swap)	BG2	BG2	BG	1
61 (0x3d)	call	2	Appel un sous programme avec un nombre d'arguments spécifiés	BG2	BG2*	BGENV*	6
62 (0x3e)	zero_line	2	Configure une ligne de cache	BG1	X	X	5
63 (0x3f)	priv_update_optop	2	Mise à jour atomique des registres OPTOP et OPLIM	NF	NF*	NF*	2
64 (0x40)	read_pc	2	Lit le registre PC courant	NF	NF	LV	1
65 (0x41)	read_vars	2	Lit le registre VARS courant	NF	NF	LV	1

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
66 (0x42)	read_frame	2	Lit le registre FRAME courant	NF	NF	LV	1
67 (0x43)	read_optop	2	Lit le registre OPTOP courant	NF	NF	LV	1
68 (0x44)	priv_read_oplim	2	Lit le registre OPLIM courant (mode privilège)	NF	NF	LV	1
69 (0x45)	read_cons_pool	2	Lit le registre CONST_POOL courant	NF	NF	LV	1
70 (0x46)	priv_read_psr	2	Lit le registre PSR courant (mode privilège)	NF	NF	LV	1
71 (0x47)	priv_read_trapbase	2	Lit le registre TRABASE courant	NF	NF	LV	1
72 (0x48)	priv_read_lockcount0	2	Lit le registre LOCKCOUNT0 courant (mode privilège)	NF	NF*	LV*	1
73 (0x49)	priv_read_lockcount1	2	Lit le registre LOCKCOUNT1 courant (mode privilège)	NF	NF*	LV*	1
74 (0x4a)	priv_read_cycle		Lit le registre CYCLE (mode privilège)	X	NF	LV	1
75 (0x4b)	Non défini						
76 (0x4c)	priv_read_lockaddr0	2	Lit le registre LOCKADDR0 courant (mode privilège)	NF	NF*	LV*	1
77 (0x4d)	priv_read_lockaddr1	2	Lit le registre LOCKADDR1 courant (mode privilège)	NF	NF*	LV*	1
78(0x4e)	Non défini						
79 (0x4f)	Non défini						
80 (0x50)	priv_read_userrange1	2	Lit le registre USERRANGE1 courant (mode privilège)	NF	NF*	LV*	1
81 (0x51)	priv_read_gc_config	2	Lit le registre GC_CONFIG courant (mode privilège)	NF	NF*	LV*	1
82 (0x52)	priv_read_brk1a	2	Lit le registre BRK1A courant (mode privilège)	NF	NF*	LV*	1
83 (0x53)	priv_read_brk2a	2	Lit le registre BRK2A courant (mode privilège)	NF	NF*	LV*	1
84 (0x54)	priv_read_brk12c	2	Lit le registre BRK12C courant (mode privilège)	NF	NF*	LV*	1
85 (0x55)	priv_read_userrange2	2	Lit le registre USERRANGE2 courant (mode privilège)	NF	NF*	LV*	1
86 (0x56)	Non défini						

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
87 (0x57)	priv_read_versionid	2	Lit le registre VERSIONID courant (mode privilège)	NF	NF*	LV*	1
88 (0x58)	priv_read_hcr	2	Lit le registre HCR courant (mode privilège)	NF	NF*	LV*	1
89 (0x59)	priv_read_sc_bottom	2	Lit le registre SC_BOTTOM courant (mode privilège)	NF	NF	LV	1
90 (0x5a)	read_global0	2	Lit le registre GLOBAL0 courant (mode privilège)	LV	LV*	LV*	1
91 (0x5b)	read_global1	2	Lit le registre GLOBAL1 courant (mode privilège)	LV	LV*	LV*	1
92 (0x5c)	read_global2	2	Lit le registre GLOBAL2 courant (mode privilège)	LV	LV*	LV*	1
93 (0x5d)	read_global3	2	Lit le registre GLOBAL3 courant (mode privilège)	LV	LV*	LV*	1
94 (0x5e)	Non défini						
95 (0x5f)	Non défini ret_from_sub	2	retour de sous-programme	X	X	ENV	7
96 (0x60)	write_PC, ret_from_sub	2	Ecrit le registre PC	NF	NF	BGENV	4
97 (0x61)	write_vars	2	Ecrit le registre VARS	NF	NF	BGENV	2
98 (0x62)	write_frame	2	Ecrit le registre FRAME	NF	NF	BGENV	2
99 (0x63)	write_optop	2	Ecrit le registre OPTOP	NF	NF	BGENV	2
100 (0x64)	priv_write_oplim	2	Ecrit le registre OPLIM (mode privilège)	NF	NF	BGENV	2
101 (0x65)	write_const_pool	2	Ecrit le registre CONST_POOL	NF	NF	BGENV	2
102 (0x66)	priv_write_psr	2	Ecrit le registre PSR (mode privilège)	NF	NF	BGENV	2
103 (0x67)	priv_write_trapbase	2	Ecrit le registre TRAPBASE (mode privilège)	NF	NF	BGENV	2
104 (0x68)	priv_write_lockcount0	2	Ecrit le registre LOCKCOUNT0 (mode privilège)	NF	NF*	BGENV*	2
105 (0x69)	priv_write_lockcount1	2	Ecrit le registre LOCKCOUNT1 (mode privilège)	NF	NF*	BGENV*	2
106 (0x6a)	priv_write_cycle	2	Ecrit le registre CYCLE (mode privilège)	X	NF	BGENV	1
107 (0x6b)	Non défini						
108 (0x6c)	priv_write_lockaddr0	2	Ecrit le registre LOCKADDR0 (mode privilège)	NF	NF*	BGENV*	2
109 (0x6d)	priv_write_lockaddr1	2	Ecrit le registre LOCKADDR1 (mode privilège)	NF	NF*	BGENV*	2

Opcode	Mnémonique	Taille	Descriptions	Groupe Picojava	Groupe JMS	Groupe JMQ	cycle
110 (0x6e)	Non défini						
111 (0x6f)	Non défini						
112 (0x70)	priv_write_userrange1	2	Ecrit le registre USERRANGE1 (mode privilège)	NF	NF*	BGENV *	2
113 (0x71)	priv_write_gc_config	2	Ecrit le registre GC_CONFIG (mode privilège)	NF	NF*	BGENV *	2
114 (0x72)	priv_write_brk1a	2	Ecrit le registre BRK1A (mode privilège)	NF	NF*	BGENV *	2
115 (0x73)	priv_write_brk2a	2	Ecrit le registre BRK2A (mode privilège)	NF	NF*	BGENV *	2
116 (0x74)	priv_write_brk12c	2	Ecrit le registre BRK12C (mode privilège)	NF	NF*	BGENV *	2
117 (0x75)	priv_write_userrange2	2	Ecrit le registre USERRANGE2 (mode privilège)	NF	NF*	BGENV *	2
118 (0x76)	Non défini						
119 (0x77)	Non défini						
120 (0x78)	Non défini						
121 (0x79)	priv_write_sc_bottom	2	Ecrit le registre SC_BOTTOM (mode privilège)	NF	NF	BGENV	2
122 (0x7a)	write_global0	2	Ecrit le registre GLOBAL0	MEM	MEM*	BGENV *	1
123 (0x7b)	write_global1	2	Ecrit le registre GLOBAL1	MEM	MEM*	BGENV *	1
124 (0x7c)	write_global2	2	Ecrit le registre GLOBAL2	MEM	MEM*	BGENV *	1
125 (0x7d)	write_global3	2	Ecrit le registre GLOBAL3	MEM	MEM*	BGENV *	1
126 (0x7e)	Non défini						
127 (0x7f)	Non défini						
128 (0x80)	rstore	2	Déplacement file vers pile (recopie des arguments) (instruction interne)	X	X	NF	1
129 (0x81)	Non défini						
130 (0x82)	store_f2s	2	Déplacement file vers pile	X	X	NF	1
131 (0x83)	search_ref	2	Recherche une donnée dans la file	X	X	NF	n
132 (0x84) à 254 (0xfe)	Non défini						
255 (0xff)	end_of_prog	2	Arrêt du simulateur	X	NF	NF	1

Tableau A-9 - Liste et description des mnémoniques d'extention du Bytecode Java.

A.e Automate d'état du module Fetch

L'automate du module Fetch comporte 14 états, cet automate gère les requêtes d'accès au cache instructions et les réponses associées. Il est capable d'attendre deux requête en attente. Ces requête sont différentiée par un identifiant (0 = pair, 1 = impair). Il est représenté par la Figure A-26.

Figure A-26 - Automate d'état du module Fetch.

Ces états sont les suivants :

- REQ_PAIR : Effectue une requête "pair".
- RIMP_WPA : Effectue une requête "impair" et attend une réponse "pair".
- WPA_WIMP : Attend une réponse "pair" et une réponse "impair".
- WPA_IMP_INV : Attend une réponse "pair" doublée par une réponse "impair".
- RPA_UDP_IMP : Effectue une requête "pair" et met à jour le buffer d'instruction avec une réponse "impair" sauvegardée.

- WT_JUMP_PA : Attente de mise à jour du buffer d'instructions après un saut "pair".
- WT_JP_PA_PWIMP : Attente de mise à jour du buffer d'instructions après un saut "pair" et réponse "impair" en attente.
- REQ_IMPAIR : Effectue une requête "impair".
- RPA_WIMP : Effectue une requête "pair" et attend une réponse "impair".
- WIMP_WPA : Attend une réponse "impair" et une réponse "pair".
- WIMP_PA_INV : Attend une réponse "impair" doublée par une réponse "pair".
- RIMP_UDP_PA : Effectue une requête "impair" et met à jour le buffer d'instruction avec une réponse "pair" sauvegardée.
- WT_JUMP_IMP : Attente de mise à jour du buffer d'instructions après un saut "impair".
- WT_JP_IMP_PWPA : Attente de mise à jour du buffer d'instructions après un saut "impair" et réponse "pair" en attente.

Désignation	**Signaux**
A	fetch_valid&&REQ_ACK&&!flush_valid
B	RSP_VAL
C	((nbvalid<=(NB_OCT-8))&&!jmp_0)‖!valid_0‖(nbvalid==0)
C'	((nbvalid<=(NB_OCT-8))&&!jmp_1)‖!valid_1‖(nbvalid==0)
D	jmp_0
D'	jmp_1
E	RSP_PKTID==0
E'	RSP_PKTID==1
F	!valid_1
F'	!valid_0
G	!valid_0‖(shift_valid&&nbvalid==INS_shift)‖(nbvalid==0)
G'	!valid_1‖(shift_valid&&nbvalid==INS_shift)‖(nbvalid==0)

Tableau A-10 - Concordance des signaux de l'automate du module Fetch.

A.f Résultats complémentaires

Cette partie présente les résultats de simulation complets des programmes de test sur les processeurs en fonction de la taille du cache instructions et des paramètres concernant le nombre d'octets maximum décodé à chaque cycle et la taille du buffer d'instructions. Les suffixes appliqués aux différents processeurs sont les suivants :

- I7 : Décode au maximum 7 octets à chaque cycle.
- I8 : Décode au maximum 8 octets à chaque cycle.
- B16 : La taille du buffer d'instructions est de 16 octets.
- B24 : La taille du buffer d'instructions est de 24 octets.

BubbleSort							
taille (octet)	256	512	1024	2048	4096	8192	16384
JMS_I7_B16	20 345 299	20 345 374	20 344 307	20 344 136	20 344 065	20 344 065	20 344 065
JMS_I8_B16	20 345 299	20 345 374	20 344 307	20 344 136	20 344 065	20 344 065	20 344 065
JMS_I7_B24	20 345 313	20 345 279	20 344 326	20 344 161	20 344 055	20 344 055	20 344 055
JMS_I8_B24	20 345 313	20 345 279	20 344 325	20 344 160	20 344 054	20 344 054	20 344 054
JMS_PRED_I7_B16	18 151 493	18 151 484	18 150 685	18 150 548	18 150 485	18 150 485	18 150 485
JMS_PRED_I8_B16	18 151 493	18 151 484	18 150 685	18 150 548	18 150 485	18 150 485	18 150 485
JMS_PRED_I7_B24	18 151 322	18 151 312	18 150 651	18 150 559	18 150 473	18 150 473	18 150 473
JMS_PRED_I8_B24	18 151 322	18 151 312	18 150 650	18 150 558	18 150 472	18 150 472	18 150 472
JMQ_I7_B16	19 406 914	19 406 845	19 406 674	19 406 516	19 406 099	19 405 962	19 405 962
JMQ_I8_B16	19 406 913	19 406 844	19 406 673	19 406 516	19 406 099	19 405 962	19 405 962
JMQ_I7_B24	19 438 524	19 438 447	19 438 282	19 438 140	19 437 726	19 437 621	19 437 621
JMQ_I8_B24	19 438 523	19 438 446	19 438 281	19 438 140	19 437 726	19 437 621	19 437 621

Tableau A-11 - Influence des paramètres de simulation généraux (BubbleSort).

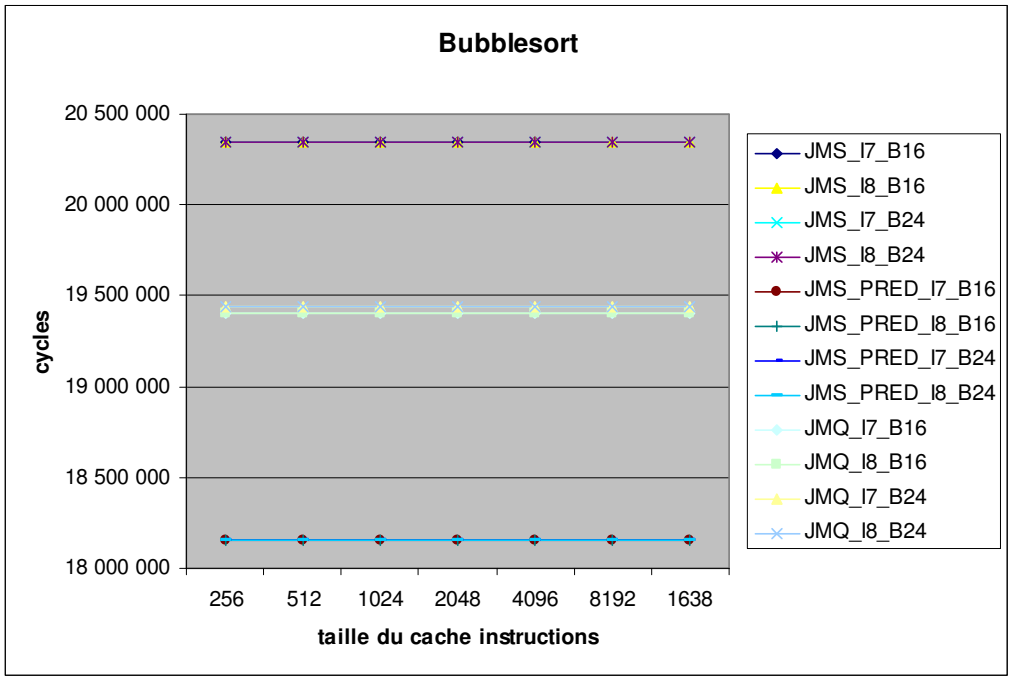


Figure A-27 - Influence des paramètres de simulation généraux (BubbleSort).

QuickSort							
taille (octet)	256	512	1024	2048	4096	8192	16384
JMS_I7_B16	326 718	326 639	326 156	325 927	325 619	325 434	325 426
JMS_I8_B16	326 718	326 639	326 156	325 927	325 619	325 434	325 426
JMS_I7_B24	325 101	324 992	324 593	324 350	323 959	323 767	323 767
JMS_I8_B24	325 100	324 990	324 591	324 348	323 957	323 764	323 764
JMS_PRED_I7_B16	303 238	303 124	302 645	302 413	302 120	301 942	301 941
JMS_PRED_I8_B16	303 238	303 124	302 645	302 413	302 120	301 942	301 941
JMS_PRED_I7_B24	303 918	303 753	303 379	303 172	302 903	302 738	302 738
JMS_PRED_I8_B24	303 917	303 751	303 377	303 170	302 901	302 735	302 735
JMQ_I7_B16	304 425	304 258	303 935	303 782	303 536	302 960	302 960
JMQ_I8_B16	304 421	304 254	303 932	303 780	303 534	302 958	302 958
JMQ_I7_B24	305 645	305 420	305 110	304 985	304 727	304 162	304 162
JMQ_I8_B24	305 643	305 419	305 109	304 985	304 727	304 160	304 160

Tableau A-12 - Influence des paramètres de simulation généraux (QuickSort).

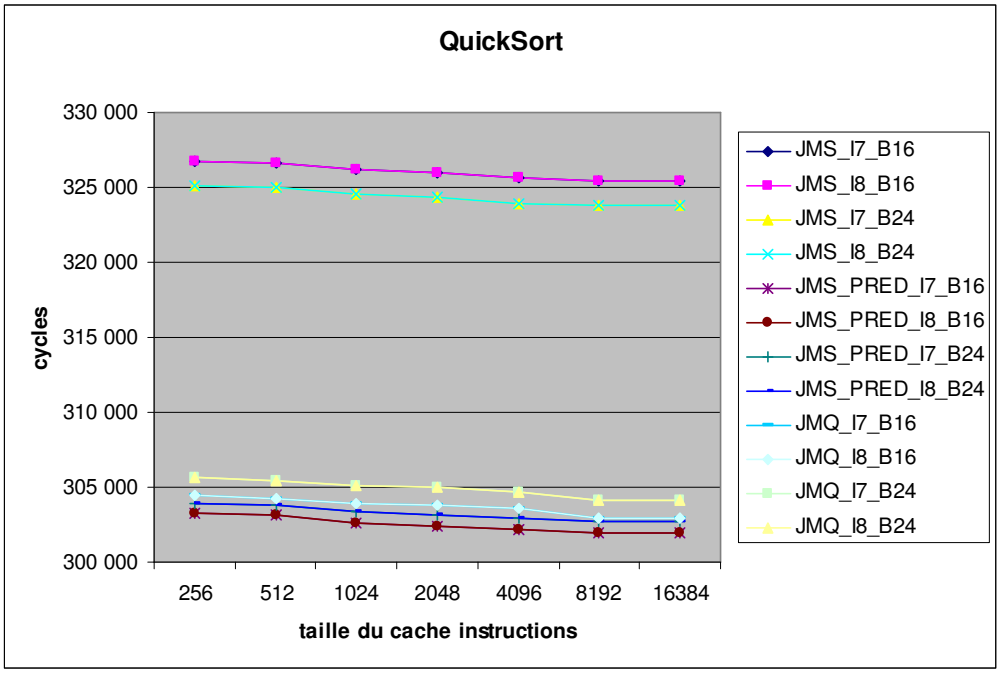


Figure A-28 - Influence des paramètres de simulation généraux (QuickSort).

Raytracer (init)							
taille (octet)	256	512	1024	2048	4096	8192	16384
JMS_I7_B16	635 927	589 930	521 919	483 903	450 930	440 203	435 297
JMS_I8_B16	635 925	589 928	521 917	483 841	450 867	440 140	435 234
JMS_I7_B24	649 478	598 808	532 960	489 797	454 096	442 000	436 163
JMS_I8_B24	649 476	598 804	532 951	489 727	454 025	441 929	436 086
JMS_PRED_I7_B16	484 739	454 157	422 332	395 818	375 510	368 365	364 747
JMS_PRED_I8_B16	484 738	454 157	422 331	395 756	375 447	368 302	364 684
JMS_PRED_I7_B24	468 373	440 701	414 430	391 015	373 509	367 402	364 323
JMS_PRED_I8_B24	468 373	440 697	414 424	390 945	373 438	367 327	364 246
JMQ_I7_B16	441 147	400 680	360 496	342 706	329 081	320 324	315 381
JMQ_I8_B16	441 100	400 633	360 459	342 677	329 057	320 308	315 366
JMQ_I7_B24	427 995	390 942	356 035	340 040	327 264	319 637	314 781
JMQ_I8_B24	427 964	390 658	355 555	339 502	326 728	319 092	314 234

Tableau A-13 - Influence des paramètres de simulation généraux (Raytracer : init).

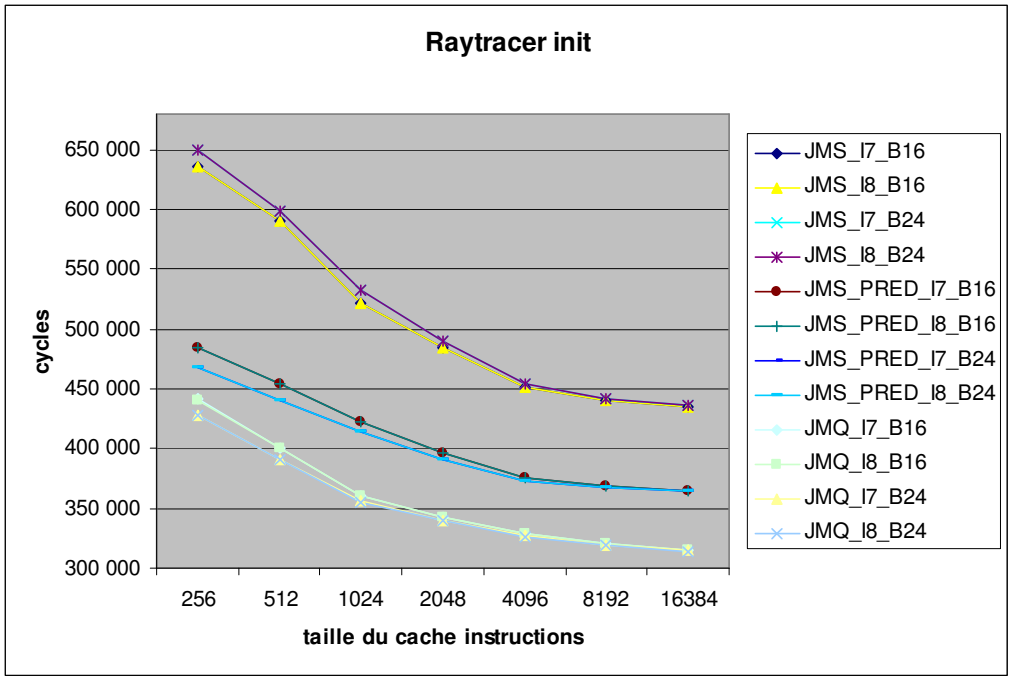


Figure A-29 - Influence des paramètres de simulation généraux (Raytracer : init).

Raytracer (exec)							
taille (octet)	256	512	1024	2048	4096	8192	16384
JMS_I7_B16	163 669 161	150 225 410	139 073 894	136 467 159	134 454 003	132 794 025	132 370 714
JMS_I8_B16	163 669 161	150 225 410	139 073 894	136 467 159	134 454 003	132 794 025	132 370 714
JMS_I7_B24	165 735 002	149 485 675	138 489 222	135 862 958	133 822 917	132 268 464	131 863 198
JMS_I8_B24	165 735 001	149 485 673	138 489 218	135 862 948	133 822 907	132 268 434	131 863 168
JMS_PRED_I7_B16	128 131 962	126 808 064	125 866 701	124 070 115	123 103 219	122 516 675	122 297 065
JMS_PRED_I8_B16	128 131 962	126 808 064	125 866 701	124 070 115	123 103 219	122 516 675	122 297 065
JMS_PRED_I7_B24	131 523 874	130 633 126	125 980 624	124 343 575	123 482 588	123 019 957	122 854 229
JMS_PRED_I8_B24	131 523 873	130 633 124	125 980 620	124 343 565	123 482 578	123 019 927	122 854 199
JMQ_I7_B16	118 503 351	117 253 499	115 544 491	114 466 179	113 544 130	112 783 429	112 664 887
JMQ_I8_B16	118 497 812	117 253 218	115 544 009	114 475 194	113 537 478	112 773 613	112 663 326
JMQ_I7_B24	116 723 426	115 864 189	113 936 711	112 890 695	112 002 151	111 326 916	111 209 020
JMQ_I8_B24	116 717 483	115 857 000	113 925 847	112 888 325	111 991 434	111 315 360	111 194 100

Tableau A-14 - Influence des paramètres de simulation généraux (Raytracer : exec).

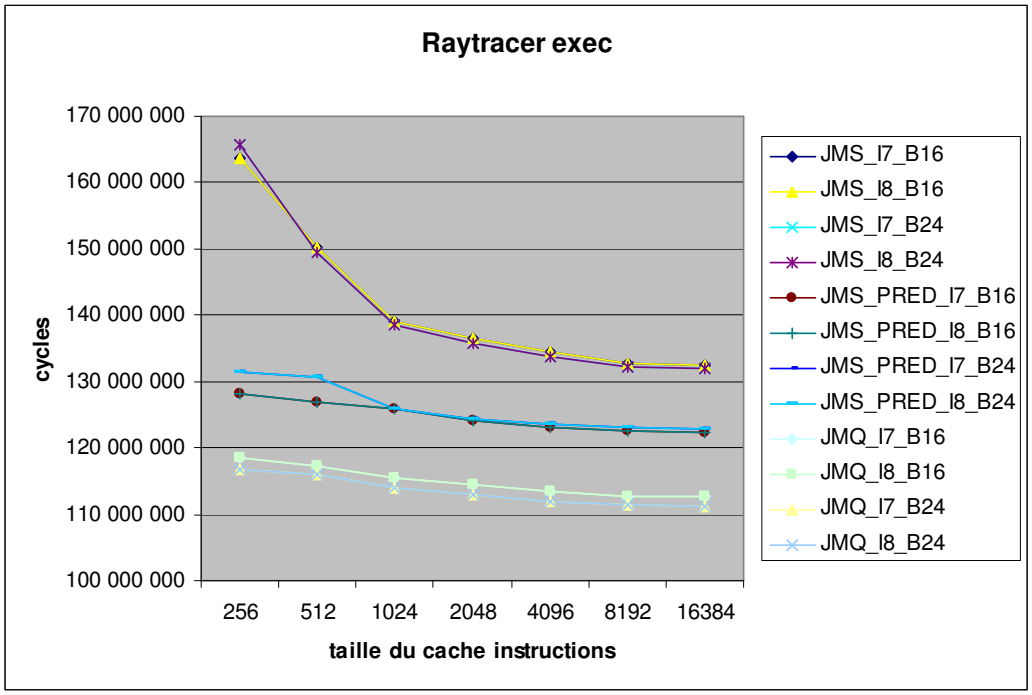


Figure A-30 - Influence des paramètres de simulation généraux (Raytracer : exec).

Raytracer (total)							
taille (octet)	256	512	1024	2048	4096	8192	16384
JMS_I7_B16	164 314 279	150 824 153	139 604 152	136 958 722	134 912 168	133 241 068	132 812 678
JMS_I8_B16	164 314 277	150 824 151	139 604 150	136 958 660	134 912 105	133 241 005	132 812 615
JMS_I7_B24	166 393 578	150 093 222	139 030 537	136 360 348	134 284 180	132 717 286	132 306 048
JMS_I8_B24	166 393 575	150 093 214	139 030 522	136 360 263	134 284 094	132 717 180	132 305 936
JMS_PRED_I7_B16	128 625 036	127 270 293	126 296 640	124 472 866	123 485 292	122 891 182	122 667 915
JMS_PRED_I8_B16	128 625 035	127 270 293	126 296 639	124 472 804	123 485 229	122 891 119	122 667 852
JMS_PRED_I7_B24	132 000 392	131 081 660	126 402 494	124 741 503	123 862 553	123 393 500	123 224 627
JMS_PRED_I8_B24	132 000 391	131 081 652	126 402 482	124 741 418	123 862 467	123 393 390	123 224 515
JMQ_I7_B16	118 952 085	117 661 509	115 911 838	114 815 125	113 879 160	113 109 155	112 985 579
JMQ_I8_B16	118 946 454	117 661 186	115 911 351	114 824 131	113 872 496	113 099 340	112 984 010
JMQ_I7_B24	117 158 795	116 262 351	114 299 533	113 236 894	112 335 295	111 651 879	111 529 035
JMQ_I8_B24	117 152 815	116 254 896	114 288 176	113 233 992	112 324 042	111 639 808	111 513 636

Tableau A-15 - Influence des paramètres de simulation généraux (Raytracer : total).

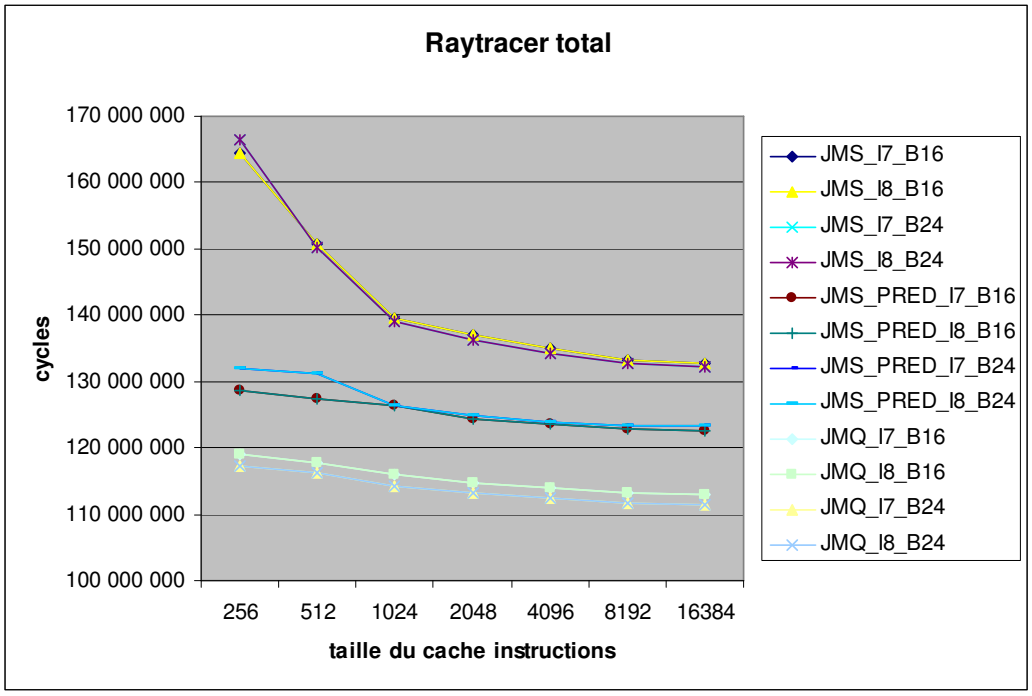


Figure A-31 - Influence des paramètres de simulation généraux (Raytracer : total).

FFT (transformée)							
taille (octet)	256	512	1024	2048	4096	8192	16384
JMS_I7_B16	78 304 333	78 300 044	78 292 430	78 283 335	78 278 281	78 277 006	78 276 273
JMS_I8_B16	78 304 333	78 300 044	78 292 430	78 283 335	78 278 281	78 277 006	78 276 273
JMS_I7_B24	78 302 742	78 299 364	78 292 713	78 283 806	78 278 009	78 276 851	78 276 224
JMS_I8_B24	78 302 742	78 299 364	78 292 713	78 283 806	78 278 009	78 276 851	78 276 224
JMS_PRED_I7_B16	76 794 501	76 790 068	76 784 222	76 778 081	76 775 545	76 774 811	76 774 159
JMS_PRED_I8_B16	76 794 501	76 790 068	76 784 222	76 778 081	76 775 545	76 774 811	76 774 159
JMS_PRED_I7_B24	76 819 922	76 816 128	76 810 380	76 804 427	76 802 310	76 801 645	76 801 121
JMS_PRED_I8_B24	76 819 922	76 816 128	76 810 380	76 804 427	76 802 310	76 801 645	76 801 121
JMQ_I7_B16	55 005 397	55 002 046	54 995 227	54 992 436	54 993 443	54 991 826	54 991 392
JMQ_I8_B16	55 005 362	55 002 011	54 995 191	54 992 402	54 993 406	54 991 790	54 991 356
JMQ_I7_B24	55 012 185	55 009 121	55 000 125	54 997 447	54 995 931	54 997 423	54 996 997
JMQ_I8_B24	55 012 185	55 009 111	55 000 084	54 997 385	54 995 864	54 997 356	54 996 929

Tableau A-16 - Influence des paramètres de simulation généraux (FFT : transformée).

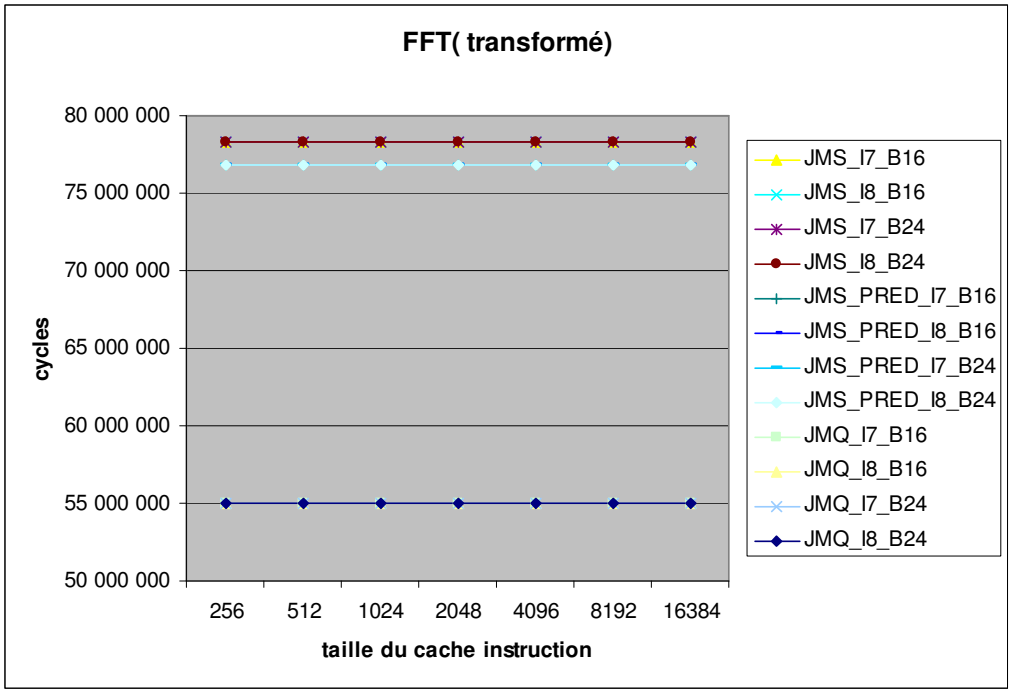


Figure A-32 - Influence des paramètres de simulation généraux (FFT : transformée).

FFT (inverse)							
taille (octet)	256	512	1024	2048	4096	8192	16384
JMS_I7_B16	80 578 018	80 576 187	80 569 614	80 560 076	80 556 164	80 555 325	80 555 043
JMS_I8_B16	80 578 018	80 576 187	80 569 614	80 560 076	80 556 164	80 555 325	80 555 043
JMS_I7_B24	80 577 408	80 575 516	80 570 154	80 560 780	80 555 963	80 555 198	80 554 936
JMS_I8_B24	80 577 408	80 575 516	80 570 154	80 560 780	80 555 963	80 555 198	80 554 936
JMS_PRED_I7_B16	78 814 512	78 810 944	78 806 534	78 800 312	78 798 912	78 798 611	78 798 377
JMS_PRED_I8_B16	78 814 512	78 810 944	78 806 534	78 800 312	78 798 912	78 798 611	78 798 377
JMS_PRED_I7_B24	78 840 454	78 837 224	78 832 938	78 827 022	78 825 813	78 825 497	78 825 289
JMS_PRED_I8_B24	78 840 454	78 837 224	78 832 938	78 827 023	78 825 813	78 825 497	78 825 289
JMQ_I7_B16	56 835 803	56 833 155	56 826 060	56 823 449	56 822 529	56 823 165	56 818 440
JMQ_I8_B16	56 835 768	56 833 120	56 826 025	56 823 415	56 822 495	56 823 131	56 818 407
JMQ_I7_B24	56 839 481	56 836 940	56 830 979	56 828 565	56 825 428	56 824 707	56 824 541
JMQ_I8_B24	56 839 481	56 836 931	56 830 942	56 828 501	56 825 364	56 827 641	56 824 476

Tableau A-17 - Influence des paramètres de simulation généraux (FFT : inverse).

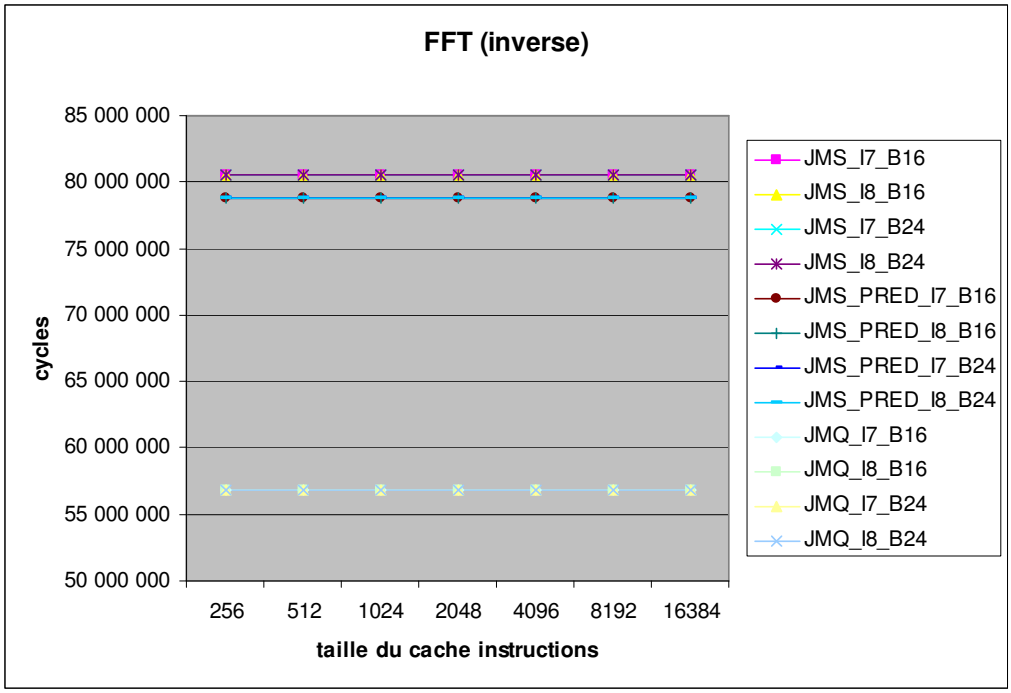


Figure A-33 - Influence des paramètres de simulation généraux (FFT : inverse).

Crypt							
taille (octet)	256	512	1024	2048	4096	8192	16384
JMS_I7_B16	94 328 154	85 832 396	63 272 023	61 982 365	61 981 972	56 847 633	56 847 445
JMS_I8_B16	94 328 154	85 832 396	63 272 023	61 982 365	61 981 972	56 847 633	56 847 445
JMS_I7_B24	88 359 611	82 724 693	64 176 630	62 934 850	62 934 540	56 847 668	56 847 539
JMS_I8_B24	88 359 611	82 724 693	64 176 630	62 934 850	62 934 540	56 847 668	56 847 539
JMS_PRED_I7_B16	69 225 988	62 970 091	52 574 670	52 078 647	52 078 367	49 841 719	49 841 599
JMS_PRED_I8_B16	69 225 988	62 970 091	52 574 670	52 078 647	52 078 367	49 841 719	49 841 599
JMS_PRED_I7_B24	61 846 201	58 380 099	51 243 164	51 221 925	51 221 648	49 861 660	49 861 583
JMS_PRED_I8_B24	61 846 201	58 380 099	51 243 164	51 221 925	51 221 648	49 861 660	49 861 583
JMQ_I7_B16	53 611 152	42 036 759	35 732 169	35 699 912	35 698 920	33 596 703	33 596 241
JMQ_I8_B16	53 611 151	42 036 758	35 732 168	35 699 911	35 698 919	33 596 703	33 596 241
JMQ_I7_B24	49 975 915	40 408 808	35 386 051	35 364 827	35 363 872	33 415 644	33 415 244
JMQ_I8_B24	49 975 916	40 408 807	35 386 050	35 364 826	35 363 871	33 415 644	33 415 244

Tableau A-18 - Influence des paramètres de simulation généraux (Crypt).

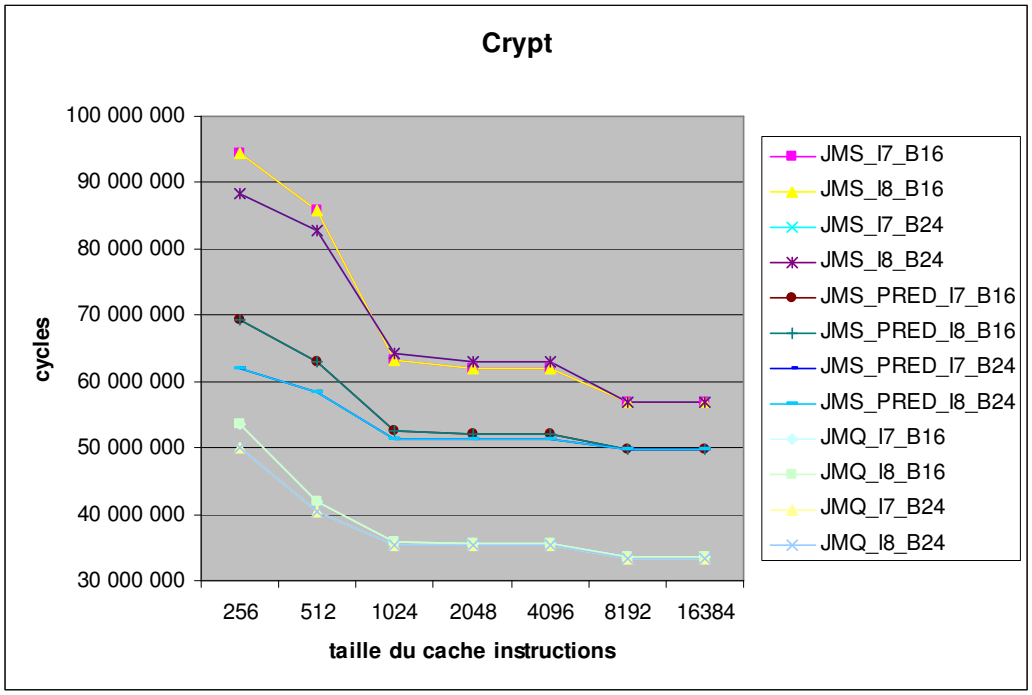


Figure A-34 - Influence des paramètres de simulation généraux (Crypt).

Le Tableau A-19 présente une comparaison du nombre de cycles d'exécution des programmes de test sur le modèle JMQ, un Pentium4 à 1,7GHz et un Solaris Ultra Spark 5. Le Tableau A-20 montre le ratio de gain de performances du modèle JMQ par rapport aux deux autres processeurs.

Ces résultats montrent une amélioration des performances de l'exécution de Java avec le modèle JMQ par rapport aux deus autres processeurs lorsque ceux-ci n'utilise pas de JIT.

		JMQ	Pentium 4 (1,7 GHz)		Solaris Ultra spark 5 (333 MHz)	
			JIT	no JIT	JIT	no JIT
Tri	QuickSort	302 960	3 465 915	3 249 189	7 436 223	52 864 416
	BubbleSort	19 405 962	24 161 507	212 668 673	1 539 543 582	23 096 088 792
Raytracer	init	315 381	17 390 529	20 758 240	8 161 497	7 200 126
	exec	112 664 887	130 834 478	950 395 516	68 529 069	764 335 899
	total	112 985 579	174 767 108	1 001 210 874	90 737 838	786 935 610
FFT	transformée	54 991 392	150 226 331	674 453 155	70 796 799	467 766 432
	inverse	56 818 440	131 506 665	1 001 100 819	52 293 987	481 059 792
Crypt		33 596 241	65 195 434	75 613 497	67 288 977	103 827 069

Tableau A-19 - Comparaison en nombre de cycles JMQ/Pentium4/Ultra Sparc.

		Pentium 4 (1,7 GHz)		Solaris Ultra spark 5 (333 MHz)	
		JIT	no JIT	JIT	no JIT
Tri	QuickSort	11,4	10,7	24,5	174,5
	BubbleSort	1,2	11,0	79,3	1 190,2
Raytracer	init	55,1	65,8	25,9	22,8
	exec	1,2	8,4	0,6	6,8
	total	1,5	8,9	0,8	7,0
FFT	transformée	2,7	12,3	1,3	8,5
	inverse	2,3	17,6	0,9	8,5
Crypt		1,9	2,3	2,0	3,1

Tableau A-20 – Ratio du nombre de cycles Pentium/JMQ et Ultra Spark/JMQ.

A.g Publication

M. Palus et F. Anceau, "JMQ, un proceseur Java de haute performance", sympA2006, Perpignan, 4-6 octobre 2006.

Chapitre B Bibliographie

1 H. Trézéguet, "les microcontrôleurs 16 et 32 bits", Electronique n°154, janvier 2005.
2 E. Quinn et C. Christiansen, "Java Pays -- Possitively", IDC Bulletin #16212, May 1998.
3 D. K. Taft, "Java surpasses C++ on SourceForge", eweek.com, 5, decembre 2005, http://www.eweek.com/article2/0,1895,1896196,00.asp.
4 E. Orgell, "Wireless Developers Lament Current State Of Important Development Tools New Evans Data Survey", Evans Data Corporation, Santa Cruz, CA, October 3, 2005.
5 Cour de JAVA de l'EMI. http://www.emi.ac.ma/etudiants/cours/java/java_histo.html.
6 Article Wikipedia "Java programming language". http://en.wikipedia.org/wiki/Java_programming_language.
7 S. Chaumette, A. Miniussi, Cour de JAVA de l'UREC, CNRS. http://www.urec.cnrs.fr/geret/96.06.www/java/00060/.
8 T. Sylvain. http://membres.lycos.fr/stcad/java/histo.html.
9 D. Reilly, Inside Java : The Java Virtual Machine, http://www.javacoffeebreak.com/articles/inside_java/insidejava-jan99.html , avril 2000
10 C-H A. Hsieh, J. C. Gyllenhaal, W. W. Hwu, "Java Byte-code to Native Code (Translation: The Caffeine Prototype and Preliminary Results", in proceeding of the 29th Annual International Symposium on Microarchitecture (MICRO-29), pp 90-97, Los Alamos, CA, USA, 2-4 Decembre 1996, IEEE Computer Society Press.
11 M. Cierniak et W. Lei, "Just-In-Time optimisations for high-performance Java programs", Concurrency: Practice and Experience, 9(4): 1063-1073, novembre 1997.
12 T. A. Proebsting, G. T. Townsend, P. Bridges, J. H. Hartman, T. Newsham, S. A. Watterson, "Toba: Java For Applications – A Way Ahead of Time (WAT) Compiler", in Proceeding Third Conference on Object-Oriented Technologies and Systems (COOTS'97), 1997.
13 G. Muller, B. Moura, F. Bellard et Charles Consel, "JIT vs Offline Compilers: Limits and Benefits of Bytecode Compilation, rapport technique 1063, IRISA, décembre 1996.
14 T. Lindholm, F. Yellin, "The Java Virtual Machine Specification Second Edition", Copyright © 1997-1999 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303 U.S.A.
15 R. Radhakrishnan, "Microarchitectural Techniques to Enable Efficient Java Execution", thèse de doctorat, Université du Texas, Austin, 2000.
16 C. J. Glossner, "DELFT-JAVA engine", thèse de doctorat, Université de technologie Delft, Hollande, 2001.
17 G. Acher, "JIFFY – Ein FPGA-basierter Java Just-In-Time Compiler fûr eingebettete Anwendungen" , Thése de doctorat, Université technique de Munique, 2003.
18 ARM, "Jazelle – ARM Architecture Extensions for Java Aplication", white paper.
19 ARM, "High performance Java on embedded devices - Jazelle®technology: ARM® acceleration technology for the Java™ Platform", White Paper, 2004. http://www.arm.com/products/esd/jazelle_home.html.
20 H. Shiffman, "JSTAR: Practical Java Acceleration For Information Appliances", février 2000, http://www.disordered.org/Java-Embed.html.
21 Nazomi, "JA108 – Multimedia Application Processor - product brief", http://www.nazomi.com/.

22 Sun Microsystems, "PicoJava™-I Microprocessor Core Architecture", transp. From Sun Microsystems, October 1996.
23 M. O'Connor and M. Tremblay, "PicoJava-I: The Java Virtual Machine in Hardware", IEEE Micro, pp. 45-47, March-April 1997.
24 Sun Microsystems, "PicoJava™-II Microarchitecture Guide", transp. From Sun Microsystems, Part No.: 960-1160-11, Mach 1999.
25 aJile System Inc, "aJ-100 Real-time Low Power Java Processor", preliminary datasheet, 2000.
26 D. S. Hardin, "aJile Systems: Low-Power Direct-Execution JavaTM Microprocessors for Real-Time and Networked Embedded Applications", aJile System Inc, White paper.
27 T. R. Halfhill. Imsys Hedges Bets on Java. *Microprocessor Report*, August 2000.
28 Imsys AB. the Cjip Technical Reference Manual / V0.25, 2004.
29 Imsys AB. ISAJ Reference 2.0, January 2001.
30 PTSC. IGNITE Processor Brochure, Rev 1.0. http://www.ptsc.com.
31 J. Bayko, "Great Microprocessors of the Past and Present", V 13.4.0, Section 7 Part IV, http://www.sasktelwebsite.net/jbayko/cpu7.html#PSC1000.
32 Vulcan ASIC Ltd. Moon v1.0. data sheet, January 2000.
33 Vulcan ASIC Ltd. Moon2 - 32 Bit Native Java Technology-Based Processor, product folder, 2003.
34 Digital Communication Technologies Ltd. Lightfoot 32-bit Java Processor Core. data sheet, September 2001.
35 Derivation Systems Inc, LavaCORE Comfigurable Java Processor Core, data sheet, April 2001.
36 R. Zulauf, "Entwurf eines Java-Mikrocontrollers und prototypische Implementierung auf einem FPGA", Master's thesis, University of Karlsruhe, Germany, 2000.
37 J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and Th. Ungerer, "Realtime Event-handling and Scheduling on a Multithreaded Java microcontroller", *Microprocessors and Microsystems*, 27(1):19–31, 2003.
38 M. Pfeffer, Th. Ungerer, "*Dynamic Real-Time Reconfiguration on a Multithreaded Java-Microcontroller"* ,International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004),Wien, Österreich, Seiten 86-92, Mai 2004.
39 S.A. Ito, L. Carro, and R.P. Jacobi, "Making Java Work for Microcontroller Applications", *IEEE Design & Test of Computers*, 18(5):100–110, 2001.
40 M. Schöberl, "JOP: A Java Optimized Processor for Embedded Real-Time Systems", Technischen Universität Wien, Fakultät für Informatik, janvier 2005.
41 R. Radhakrishnan, N. Vijaykrishnan, L. Kurian John, A. Sivasubramaniam, J. Rubio, and J.Sabarinathan. "Java Runtime Systems: Characterization and Architectural Implications". *IEEE Trans. Comput.*, 50(2):131– 146, 2001.
42 F. Anceau, G. Baille et J.P. Shoellkopf, "machine informatique électronique destinée à l'exécution parallèle d'un language post-fixé", brevet ANVAR n°75 29 473, France, septembre 1975.
43 G. Baille et J.P. Schoellkopf, "Evaluation of polish-form expression on a FIFO queue: a new approach towards the realization of a high-level pipeline computer" Sagamore Computer Conference on Parallel Processing, août 1975.
44 JP. Schoellkopf, "Machine PASC-HLL : Définition d'une architecture pipe-line pour une unite centrale adaptée au langage pascal", Thèse de 3ème cycle, INP6, Grenoble, 28 juin 1977.
45 Sun Microsystems, "PicoJava™-I Microprocessor Core Architecture", transp. From Sun Microsystems, October 1996.

46 H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine For Java Bytecode", IEEE Computer, pp. 22-30, October 1998.
47 Sun Microsystems, "PicoJava™-II Microarchitecture Guide", transp. From Sun Microsystems, Part No.: 960-1160-11, Mach 1999.
48 F. Arzel, "Cache non-bloquant pour processeur superscalaires", rapport de stage, LIP6, septembre 2005.
49 J. Dumesnil,"Modélisation SystemC d'un prédicteur de branchement pour processeur Java", rapport de stage, LIP6, septembre 2005.
50 J. E. Smith, "A Study of Branch Prediction Strategies", Proceedings of the 8th International Symposium on Computer Architecture, Mineapolis, pp.135-148, May 1981.
51 Wikipédia, "Tri à bulles", http://fr.wikipedia.org/wiki/Tri_%C3%A0_bulles.
52 Wikipédia, "Tri rapide", http://fr.wikipedia.org/wiki/Tri_rapide.
53 Wikipédia,"Lancer de rayon", http://fr.wikipedia.org/wiki/Lancer_de_rayon.
54 Wikipédia, "Transformée de Fourier rapide", http://fr.wikipedia.org/wiki/Transform%C3%A9e_de_Fourier_rapide.
55 Wikipédia, "International Data Encryption Algorithm", http://fr.wikipedia.org/wiki/International_Data_Encryption_Algorithm.
56 Sun Microsystems, "PicoJava™-II Programmer's Reference Manual", transp. From Sun Microsystems, Part No.: 805-2800-06, Mach 1999.
57 J-M. Douin, "Machine Virtuelle Java", Techniques de l'ingénieur, traité Informatique, H 1 588, octobre 2000.