Université de Montréal

Méthodologie et outil de conception de systèmes embarqués basés sur le cadre d'applications .NET

par
Olivier Brassard

Département d'informatique et de recherche opérationnelle (DIRO)
Faculté des Arts et Sciences

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

Mars 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Méthodologie et outil de conception de systèmes embarqués basés sur le cadre d'applications .NET

présenté par :
Olivier Brassard

a été évalué par un jury composé des personnes suivantes :

Victor Ostromoukhov
Président rapporteur

El Mostapha Aboulhamid
Directeur de recherche

Jean Pierre David
Codirecteur

Frédéric Rousseau
Membre du jury

Mémoire accepté le 3 mai 2006

RÉSUMÉ

Tel que prédit par la loi de Moore, le nombre de transistors sur une même puce augmente à un rythme effréné. Les concepteurs de systèmes embarqués tentent de profiter de ces possibilités en produisant des systèmes de plus en plus complexes, réalisant un plus grand nombre de fonctionnalités.

Toutefois, les méthodologies de conception des systèmes embarqués, ainsi que les outils supportant ces méthodologies, ne permettent pas d'utiliser efficacement toute la puissance potentielle des puces d'aujourd'hui. En effet, ces méthodologies ne répondent plus aux besoins en termes de rapidité et de qualité de conception.

Une méthodologie de conception différente et un outil supportant cette méthodologie sont proposés ici. Cette méthodologie permet la conception rapide et sans erreur d'un prototype d'un système embarqué.

Cette méthodologie offre la possibilité au concepteur de créer un système embarqué à partir de différents langages de programmation de haut niveau. Une seule et même description du système est raffinée automatiquement en parties logicielles et matérielles pour obtenir un prototype du système à implémenter sur l'architecture ciblée. Cette approche est possible grâce aux outils de développement utilisés.

Mots clés : Système embarqué, compilation, prototypage rapide, méthodologie de conception, cadre d'application .NET, CASM.

SUMMARY

Just as Moore's Law predicts, the number of transistor on a chip increases incredibly quickly. Embedded systems designers are trying to use this to their advantage by producing systems more and more complex and implementing a greater number of functionalities.

Unfortunately, design methodologies for embedded systems, and the tools supporting these methodologies, cannot cope with this increased complexity and so, cannot use efficiently the full potential the technology provides. Precisely, current methodologies do not answer the needs in terms of speed and quality of designs.

A different methodology for embedded systems design is proposed here. This methodology allows a fast and error-free design of an embedded system prototype.

The methodology lets a designer create an embedded system from a large number of different high-level programming languages. Only one system specification is given to the tool, which automatically transforms it and refines it to generate the software and the hardware parts. The hardware and the software are implemented on the target architecture to produce a prototype of the system. This approach is possible because of the development tools used.

Keywords: Embedded systems, compilation, rapid prototyping, design methodology, .NET framework, CASM.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

LISTE DU CODE

LISTE DES SIGLES

ABI: Application Binary Interface

ALAP: As Late As Possible

ASAP: As Soon As Possible

ASM: Algorithmic State Machine

BB: Bloc de Base

CASM: Channel-Based Algorithmic State Machine

CIL: Common Intermediate Language

CLI: Common Language Infrastructure

CLR: Common Language Runtime

CTS: Common Type System

EDK: Embedded Development Kit

FPGA: Field Programmable Gate Array

FSL: Fast Simplex Link

GFC : Graphe de Flot de Contrôle

HDL: Hardware Description Language

JVM: Java Virtual Machine

RI: Représentation Intermédiaire

RTL: Register Transfer Level

VES: Virtual Execution System

REMERCIEMENTS

Sans la confiance et surtout la patience de mon directeur de recherche, El Mostapha Aboulhamid, et de mon codirecteur, Jean Pierre David, ce mémoire n'aurait jamais vu le jour. Je les remercie aussi de m'avoir guidé lors de mon cheminement par leurs conseils judicieux et leurs questions pointues qui ont toujours su mettre en doute les solutions proposées, afin de les améliorer. J'aimerais aussi remercier Frédéric Rousseau qui, depuis son arrivée ici, m'a pris sous son aile et m'a grandement aidé à la rédaction du mémoire. Il pourrait être considéré comme mon co-codirecteur.

Je ne saurais comment remercier assez mes parents pour leur aide précieuse lors de mon cheminement scolaire. Étant professeurs et enseignants, ils ont toujours démontré l'importance de l'éducation afin de réaliser ses projets. Je voudrais souligner leur compréhension et leur tolérance de mes choix ainsi que, eux aussi, leur patience à mon égard. Merci particulièrement à mon père qui a eu le courage de lire et de corriger mon mémoire. Et que dire de leur support économique, moral et émotif. Merci aussi à mon frère et à tout le reste de ma famille, qui m'ont supporté tout au long de mon cheminement.

Finalement, je dois remercier mes amis. Quoi de mieux qu'un petit match de hockey à regarder ou à jouer, qu'une petite sortie ou qu'un petit carnaval pour oublier mes obligations et mes devoirs.

1 Introduction

De nos jours, partout autour de nous, des systèmes numériques sont discrètement présents. Qu'ils se trouvent dans les appareils électroménagers, dans des véhicules ou dans des objets simples comme une montre ou un réveille-matin, ils sont désormais indispensables dans notre quotidien. Même si ces systèmes, appelés *systèmes embarqués*, sont souvent peu remarqués et que leur impact est sous-estimé, ils ont un volume de marché environ 100 fois plus grand que celui des ordinateurs de bureau [75].

Les systèmes embarqués sont caractérisés par des propriétés particulières [75]:

- Ils sont des sous-systèmes de traitement d'information des systèmes dans lesquels ils sont incorporés.
- Ils procurent des fonctionnalités spécifiques et hautement spécialisées de traitement d'information au système.
- Ils sont réactifs, c'est-à-dire qu'ils interagissent avec leur environnement, souvent de manière continue et à une vitesse imposée par l'environnement.
- Ils sont généralement composés de parties logicielles et de parties matérielles spécialisées pour répondre à un certain nombre de contraintes non fonctionnelles;
- Ils ne sont généralement pas visibles ou directement accessibles à l'utilisateur, même s'ils sont souvent utilisés pour augmenter la facilité d'utilisation du système dont ils font partie.

Comme décrit plus haut dans la quatrième caractéristique, de nombreuses contraintes non fonctionnelles ont de fortes influences sur leur conception. Par exemple, un système embarqué doit avoir une basse consommation d'énergie pour servir dans des systèmes portables alimentés pas des batteries ou des accumulateurs. Le temps d'exécution du système doit être court puisque ces systèmes sont souvent utilisés dans des environnements temps-réel. La fiabilité, la robustesse et la sécurité sont nécessaires à un tel système auquel une certaine autonomie est demandée et dont le redémarrage peut être impossible lorsqu'utilisé comme outil dans certains domaines, comme l'aérospatial ou la médecine par exemple. Finalement, le coût du système embarqué doit être le plus bas possible. Toutes ces contraintes rendent la conception des systèmes embarqués complexe et difficile.

La Figure 1 montre un système embarqué dans son environnement. Un système embarqué interagit avec son environnement à l'aide de capteurs et senseurs ou par des interfaces de communication. Comme il est possible de constater sur la figure, un système embarqué peut contenir d'autres systèmes embarqués.

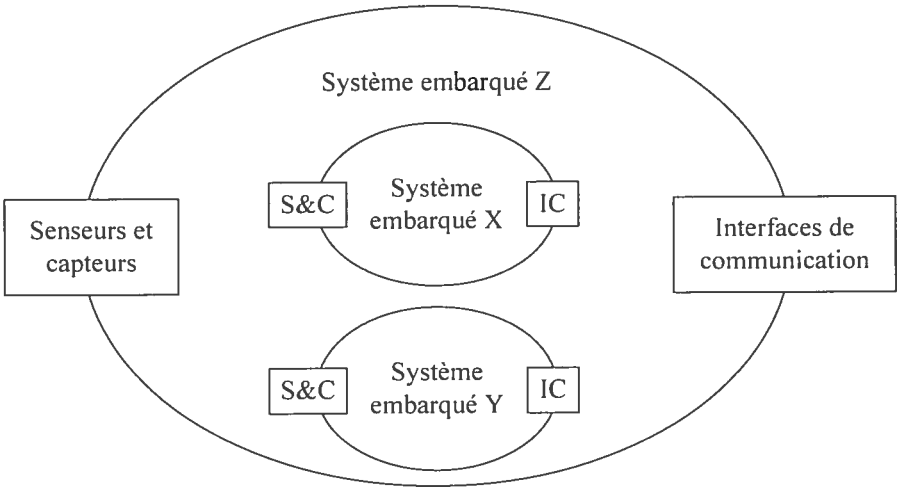


Figure 1 : Système embarqué et son environnement [75]

La loi de Moore prédit que la capacité de transistors des puces double a chaque 18 mois. Dûe à cette croissance, la complexité des systèmes embarqués est inévitablement de plus en plus grande. En conjuguant cela avec les contraintes du monde des systèmes embarqués énoncées plus haut, la conception de tels systèmes devient une affaire complexe et ardue. Les méthodes de conception actuelles ne permettent pas de combler l'écart entre les possibilités technologiques et les possibilités que permettent les outils de conception. Ainsi, pour faciliter la conception, les systèmes embarqués sont souvent constitués en très grande partie de logiciel s'exécutant sur un ou des microprocesseurs génériques et de très peu de matériel réalisant des fonctionnalités spécialisées à grande vitesse. Cette tendance est d'autant plus encouragée par le fait qu'une majorité des concepteurs de systèmes embarqués sont peu familiers avec les outils de conception de matériel et aussi parce que le cycle de développement du logiciel est plus court [76]. Les systèmes embarqués conçus avec cette approche sont souvent composés de microprocesseurs trop gros et peu performants, ayant potentiellement des coûts de

production élevés. De plus, l'utilisation de matériel pour la réalisation de fonctionnalités spécialisées amène un grand nombre de problèmes. Par exemple, les méthodologies et les outils de conception du logiciel et du matériel sont très différents, amenant une conception indépendante des deux parties pour un même système. Des possibilités d'optimisations sont alors perdues et l'intégration devient difficile. Aussi, le problème de partitionnement logiciel/matériel devient trop long à résoudre automatiquement pour des systèmes complexes. Quoi qu'il en soit, les équipes de conception sont généralement composées de grand nombre de spécialistes de différents domaines pour répondre à tous les besoins.

Afin de résoudre tous ces problèmes, de nouvelles méthodologies et outils de conception doivent être développés qui permettent une plus grande productivité à un moindre coût. L'un des principaux buts est l'augmentation des niveaux d'abstraction de la conception pour permettre la manipulation d'une plus grande quantité d'information sans avoir à se soucier des détails. D'une part, de nouveaux modèles de conception à un haut niveau d'abstraction doivent être recherchés. D'autre part, de nouveaux outils doivent être développés pour supporter ces abstractions lors de la modélisation, de la vérification et de l'implémentation des systèmes et offrir des transformations automatiques d'un niveau d'abstraction donné vers les niveaux plus bas [68]. Des outils doivent aussi donner lieu à un passage automatique et sans erreur d'une étape de conception vers une autre. Finalement, les méthodologies et les outils doivent permettre un développement facile des parties matérielles, tout en proposant une approche unifiée à la conception du matériel et du logiciel.

1.1 Objectifs

Nos recherches se concentrent sur le développement de nouvelles méthodes et outils de conception pour les systèmes embarqués. Plus particulièrement, nous voulons explorer la faisabilité d'un outil permettant une conception unifiée et efficace des systèmes embarqués à partir d'un haut niveau d'abstraction, de la modélisation à l'implémentation. Des transformations automatiques d'une étape de la conception vers la suivante doivent être réalisées par l'outil et la spécification du système doit être faite à un niveau d'abstraction assez haut pour ne pas avoir à se soucier des détails d'implémentation, en

plus de permettre à un développeur peu expérimenté en matériel de réaliser la conception du système.

1.2 Contributions

Nous avons développé le prototype d'un outil, appelé *compilateur CIL*, qui permet une conception automatisée, de la modélisation à l'implémentation, à partir de divers langages de programmation de haut niveau supportés par le cadre d'application .NET. En utilisant une forme intermédiaire commune aux langages d'entrée, issue d'un standard ISO et ECMA, l'outil génère automatiquement un exécutable pour la partie logicielle du système et un modèle de niveau Register Transfer Level (RTL) synthétisable pour la partie matérielle. Des techniques de compilation classiques sont utilisées pour la génération de la partie logicielle, alors que l'utilisation d'un nouveau langage de description de matériel de niveau intermédiaire est utilisée pour le passage de la spécification de haut niveau au niveau RTL pour la partie matérielle. Le tout est intégré à un système contenant un cœur de processeur logiciel et est synthétisé sur FPGA. Notre outil est réalisé à l'aide du langage C# du cadre d'applications .NET. La plateforme ciblée est un FPGA Virtex II Pro de Xilinx.

L'outil développé n'est pas un outil complet, mais peut être utilisé comme base pour d'autres recherches dans le domaine. Ainsi, le compilateur ne supporte pas l'ensemble du langage d'entrée. Le but de notre travail est plus d'investiguer les méthodologies de conception complètes à un haut niveau d'abstraction que la conception d'un outil complet et robuste. L'utilisation d'applications simples illustre la faisabilité, l'efficacité et l'intérêt de la méthodologie et de l'outil développé.

1.3 Plan du document

La deuxième section constate la situation actuelle de la conception des systèmes embarqués et la problématique reliée à cette situation. Nous présentons plus en détail le flot de conception classique de systèmes embarqués logiciel/matériel. Nous présentons

aussi l'état de l'art du domaine, en exposant des techniques et des outils existants qui tentent de résoudrent les problèmes liés à l'approche actuelle de conception.

La troisième section décrit les outils utilisés dans nos recherches pour le développement de systèmes embarqués. Nous présentons le cadre d'application .NET et ses particularités, le langage de description de matériel de niveau intermédiaire CASM, le processeur Microblaze et le Embedded Development Kit de Xilinx, qui sont utilisés pour la conception et l'implémentation du système final.

La quatrième section présente le flot de conception de systèmes embarqués à l'aide de notre outil. Les avantages de l'approche utilisée sont aussi décrits.

La cinquième section décrit les détails d'implémentation de l'outil. Le fonctionnement interne de l'outil est donné, comme les traitements effectués, les algorithmes et les représentations des données utilisés.

La sixième section donne des exemples d'utilisation de l'outil pour la conception de systèmes. Une première application est utilisée afin d'illustrer les étapes exécutées par l'outil pour réaliser le prototype d'un système. Une deuxième application est utilisée afin d'obtenir des mesures de performances.

Finalement, la huitième section résume l'ensemble du travail effectué et propose des améliorations possibles pour le futur ainsi que certaines solutions alternatives à des problèmes de la réalisation de l'outil.

2 Conception des systèmes embarqués : situation et problématique

Les méthodes de conceptions actuelles des systèmes embarqués découlent d'une évolution des méthodologies de conception de deux mondes qui étaient relativement distincts jusqu'à tout récemment : le monde du logiciel et le monde du matériel. Dans ces deux mondes, les techniques utilisées actuellement résultent d'élévations des niveaux d'abstraction des informations à manipuler. Cela a permis de concevoir des systèmes de plus en plus complexes et performants.

Dans cette section, nous retraçons brièvement l'historique de la conception logicielle et de la conception matérielle pour voir d'où provient la situation actuelle de la conception des systèmes embarqués. Par la suite, nous présentons en détail le flot de conception classique des systèmes embarqués utilisé aujourd'hui, en mentionnant les problèmes importants de cette approche. Finalement, nous faisons un survol des nouveaux outils et techniques qui tentent d'apporter des améliorations au flot de conception classique.

2.1 Le logiciel

La conception du logiciel est l'activité qui consiste à créer du code exécutable qui réalise les fonctionnalités désirées. Nous présentons ici un bref aperçu de l'évolution des techniques et outils utilisés dans le but de produire de façon rapide et efficace ce code (tiré de [4,5]).

2.1.1 Langages de bas niveau

Traditionnellement, le logiciel est une suite d'instructions encodées qui sont déchiffrées et exécutées par une unité centrale de traitement.

Lorsque les premiers processeurs génériques sont apparus dans les années 40, les programmes étaient codés bit par bit en langage machine. Même si cette façon de faire permettait un contrôle total sur la machine, le codage utilisé était inintelligible pour l'humain. Les programmes prenaient alors une éternité à écrire, étaient remplis d'erreurs difficiles à corriger et devraient être complètement réécris si l'architecture sur laquelle on

l'exécutait devait changer. Vers 1955, les premiers langages d'assemblage sont apparus. Ceux-ci sont une représentation textuelle symboliques du langage machine utilisé. Même si la lisibilité des programmes était grandement améliorée et que la productivité se voyait augmentée, beaucoup de choses devaient être écrites pour faire peu de choses et le programmeur devait toujours être conscient des caractéristiques physiques de la machine hôte. Le langage machine et le langage d'assemblage sont des *langages de bas niveau*, signifiant qu'il faut s'occuper autant des détails de la machine que du calcul à effectuer.

2.1.2 Langages de haut niveau

À la fin des années 50 sont apparus les premiers *langages de haut niveau*. À la base de ceux-ci se trouvent les compilateurs. Le premier compilateur fut développé par une équipe de IBM dirigée par John Backus. Cet outil permettait une transformation automatique d'un programme FORTRAN en un programme équivalent en langage machine. Par la suite, durant les années 60 et au début des années 70, de nombreuses techniques furent développées et de nombreux autres compilateurs furent créés pour un grand nombre de langages, comme Algol, Pascal, Simula ou Lisp.

De cette époque à aujourd'hui, de nombreux langages et leurs compilateurs sont apparus. Des exemples dignes de mention sont les langages C, utilisé pour l'écriture du système d'exploitation UNIX dans les laboratoires Bell, C++, une évolution du C en y incorporant des concepts de la programmation orienté-objet, et Java, une version épurée, sécuritaire et portable de C++. Le langage Java représente aussi une classe différente de langage puisqu'un programme écrit en Java est compilé en un langage intermédiaire de bas niveau, le *bytecode* de Java. Ce langage de bas niveau est ensuite exécuté sur une machine virtuelle commune à toutes les architectures d'ordinateurs. Ainsi, un programme écrit en Java peut être exécuté sur n'importe quelle architecture sans avoir à être recompilé; seule la machine virtuelle doit être portée sur toutes les architectures ciblées.

La caractéristique principale des langages de programmation de haut niveau est le niveau d'abstraction élevé qu'ils permettent. Ils permettent de décrire des concepts et des algorithmes sans avoir à se soucier des détails d'implémentation, qui sont cachés par le langage et traités par le compilateur du langage. Il est toutefois important de noter qu'un langage de haut niveau d'aujourd'hui pourrait être de bas niveau par rapport à un langage

de demain. Aussi, certains langages peuvent être de haut niveau pour certains aspects, alors qu'ils sont de plus bas niveau pour d'autre. Le degré d'abstraction reste une mesure relative.

2.2 Le matériel

La conception de matériel, ou plus précisément de circuits numériques, est une activité qui a suivi un cours semblable à celui de la conception de logiciel, dans le sens que les recherches effectuées dans ce domaine ont donné place à l'élévation du niveau d'abstraction des méthodologies et des outils pour permettre la conception de systèmes de plus en plus complexes. Nous présentons l'évolution des méthodes et des outils.

2.2.1 Niveau physique

Au départ, lorsque les dispositifs électroniques étaient composés de tubes, tout était conçu et fabriqué à la main. Il était donc extrêmement laborieux de construire des systèmes complexes, d'autant plus que la technologie ne le permettait pas, vu la taille des dispositifs à tubes. Par exemple, le premier ordinateur électronique reprogrammable, le ENIAC, conçu en 1946, occupait une surface de 167 m^2 et pesait près de 30 tonnes [6]!

2.2.2 Niveau des portes logiques

Les circuits intégrés (CI) sont venus remplacés les tubes dans les années 50. Les premiers circuits intégrés, appelés SSI pour Small-Scale Integration, ne contenaient qu'une dizaine de transistors sur une même puce. À cette époque, les circuits étaient encore conçus à la main et les transistors, résistances et condensateurs étaient agencés au besoin. Au cours des années 60, les MSI (Medium-Scale Integration) apparurent et permettaient d'avoir une centaine de transistors, donc un système plus complexe, pour un prix légèrement supérieur aux SSI. À cette époque, les systèmes numériques étaient alors conçus au niveau des portes logiques, alors que chaque porte était composée des éléments de base formant un circuit (i.e. transistors, résistances, etc.). Il était plus facile pour les concepteurs de concevoir des circuits puisque le niveau d'abstraction commençait déjà à être plus élevé et seule une connaissance de la logique booléenne était nécessaire.

2.2.3 Niveau Register Transfer Level (RTL)

Avec l'apparition des LSI (Large-Scale Integration), au milieu des années 70, et des VLSI (Very Large-Scale Integration) au début des années 80, un circuit pouvait alors contenir de quelques milliers à plusieurs centaines de milliers de transistors. Cependant, ce très grand nombre de transistors rendait la conception manuelle des circuits presque impossible. C'est à ce moment que les concepteurs de circuits ont commencé à utiliser le niveau RTL, où un circuit est décrit par des composants réalisant des fonctionnalités et les interconnexions entre ces composants. Par exemple, un circuit serait composé de registres, d'unités arithmétiques et logiques et de multiplexeurs connectés entre elles.

Des outils rendant possible la conception à ce niveau furent développés. À cette époque, pour tenter de s'adresser au problème de la conception de matériel, le Département de la Défense des États-Unis décide de développer un langage de documentation des circuits : le VHDL. L'idée de « simuler » cette documentation est vite apparue, permettant alors de vérifier les propriétés du circuit. À la même époque, le langage Verilog était développé chez Automated Integrated Design Systems, en ayant comme but la simulation des circuits. D'autres outils permirent ensuite d'automatiquement prendre une description au niveau RTL d'un circuit écrit dans un langage de description de matériel (HDL ou Hardware Description Language) comme VHDL ou Verilog et de la transformer en une description au niveau des portes logiques. La productivité se voyait augmentée grâce à ces techniques automatisées.

2.2.4 Haut niveau

Suivant toujours inexorablement la loi de Moore, la densité des circuits continuait à augmenter à un rythme effarant, permettant des systèmes de plus en plus complexes. Les outils se devaient de suivre ce rythme. De nouveaux langages de description de matériel ont été développés, offrant encore un plus grand niveau d'abstraction. Citons par exemple SystemC [7] et SystemVerilog [1], qui sont utilisés aujourd'hui pour des descriptions au niveau système. D'autres recherches ont tenté de développer de nouveaux langages pour la description de matériel avec une syntaxe très proche de langages de programmation déjà populaires, notamment le C. Handel-C [9], HardwareC [69], SpecC [11] et Transmogrifier

C [56] sont de tels langages. Le langage VHDL a aussi évolué pour permettre la synthèse d'une description comportementale des circuits [13].

2.2.5 Les Field-Programmable Gate Array (FPGA)

Un FPGA contient un ensemble de composants logiques et d'interconnexions programmables. En configurant ces ressources, il est possible de reproduire les fonctionnalités de base des portes logiques élémentaires. En combinant un grand nombre de portes, il est possible d'implémenter des fonctionnalités complexes, comme des décodeurs ou des fonctions mathématiques. Les HDL sont utilisés pour décrire les fonctionnalités d'un système numérique, qui est transformé et utilisé pour configurer le FGPA en très peu de temps.

Depuis leur apparition au milieu des années 80, les FPGA ont révolutionné le monde de la conception de systèmes. Alors qu'auparavant, un système devant être implémenté sur un ASIC pouvait coûter une somme astronomique à réaliser et que le prototypage était ainsi presque impossible, les FPGA permettent de réaliser des circuits rapidement et de tester leur implémentation physique. Une fois que le design est validé, une implémentation est généralement réalisée à l'aide d'une technologie fixe et plus rapide. Les FPGA peuvent aussi être utilisés pour réaliser des circuits dont les fonctionnalités peuvent être modifiées dynamiquement, en reconfigurant les composants logiques au cours de l'exécution. Des systèmes très flexibles peuvent ainsi être réalisés.

2.3 Conception de systèmes embarqués

Le développement des systèmes embarqués, qui sont composés d'une partie logicielle et d'une partie matérielle, est en quelque sorte la rencontre des mondes de la conception logicielle et matérielle. Nous décrivons le flot classique de développement et nous énonçons les désavantages de cette approche.

2.3.1 Flot de développement

La conception de systèmes logiciel/matériel se décompose en plusieurs étapes successives : la *modélisation* (ou spécification), la *vérification* et l'*implémentation*. La modélisation d'un système embarqué consiste à conceptualiser et raffiner les

spécifications afin d'obtenir des modèles pour le logiciel et le matériel. La vérification est l'étape qui doit donner au concepteur l'assurance que le système fonctionne tel que prévu. L'implémentation est la réalisation physique du matériel (par la synthèse) et du logiciel (par la compilation) [48].

La Figure 2 illustre en détail le flot classique de développement.

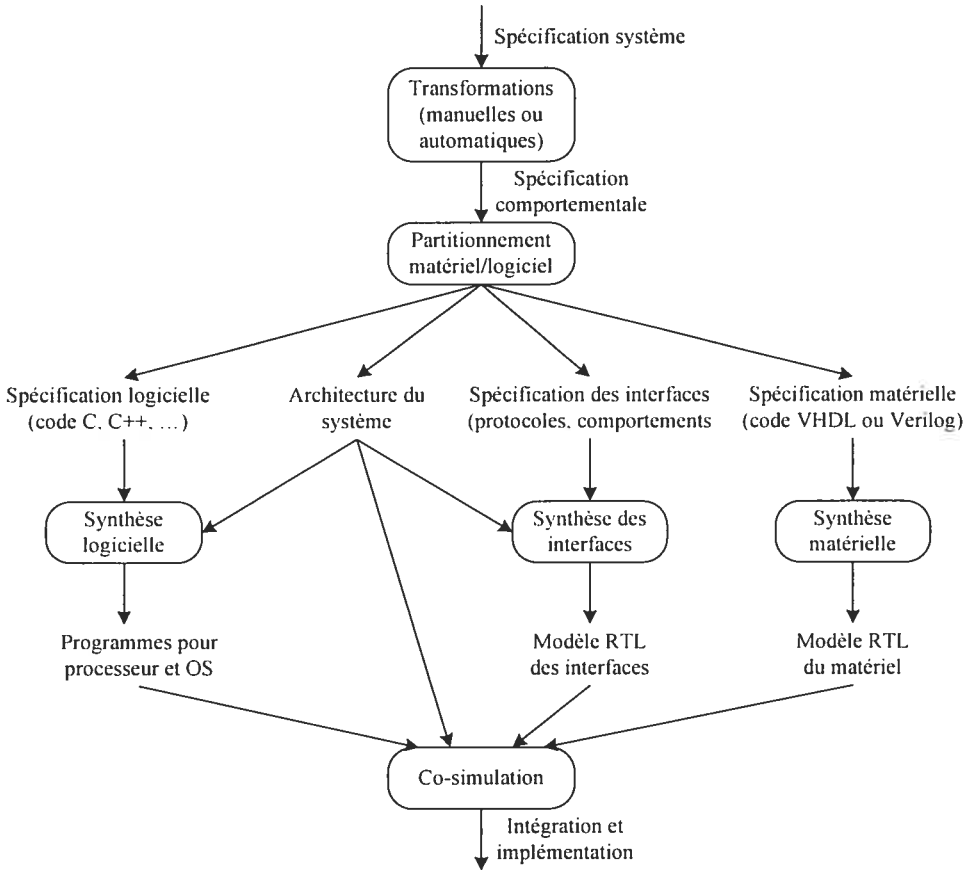


Figure 2: Flot classique idéal de conception logiciel / matériel

La conception commence avec une description à un haut niveau d'abstraction pour s'affranchir de nombreux détails de réalisation. À ce niveau d'abstraction, on s'intéresse à la fonctionnalité, indépendamment de l'implémentation finale, ce qui correspond à la conception système. On recherche des algorithmes et des représentations de données les plus adéquates. Généralement, la spécification de haut niveau est réalisée à l'aide d'un langage de haut niveau comme C ou C++, qui permettent aussi une validation des fonctionnalités rapide grâce à leur vitesse d'exécution.

L'étape suivante consiste à trouver une architecture efficace. C'est l'exploration d'architectures qui détermine la réalisation logicielle ou matérielle de tous les composants. Grossièrement, les composants qui nécessitent des performances élevées sont réalisés en matériel, alors que ceux nécessitant de la flexibilité sont réalisés en logiciel. On choisit aussi dans cette étape les composants physiques (processeur, DSP, etc.) qui exécutent les parties logicielles, ainsi que l'architecture mémoire, la gestion des entrées / sorties, etc. À la fin de cette étape, on obtient les spécifications de tous les composants matériels et logiciels.

Les étapes suivantes sont la conception du matériel, du logiciel et des interfaces composées elles aussi de parties matérielles et logicielles. Idéalement, il faudrait réutiliser des composants existants pour gagner du temps, par rapport à une conception complète de tout le système. Pour les composants matériels qui n'existent pas déjà, la conception peut suivre le flot traditionnel de conception avec les différentes étapes de synthèse (comportementale, logique et finalement physique) [70]. Le logiciel est implémenté en couches pour séparer les différentes fonctionnalités. Au plus bas niveau, des pilotes ou des interfaces logicielles (*Hardware Abstraction Layer*) permettent d'accéder aux ressources matérielles. Au dessus, le système d'exploitation gère l'exécution des différentes tâches de l'application, ainsi que les entrées / sorties. Enfin, le code de l'application s'exécute à travers ce système d'exploitation. Une phase traditionnelle de compilation, d'édition de liens et de chargement permet d'obtenir le code exécutable sur les processeurs cibles.

Puis, c'est la phase d'intégration logiciel / matériel. Il s'agit, d'une part, de vérifier que le logiciel s'exécute correctement sur les composants programmables, mais aussi, d'autre part, que les échanges d'information entre les composants sont corrects. Pour ces échanges d'informations, des composants d'interface (adaptateurs de communication) sont placés entre les composants et le réseau de communication ayant pour effet d'adapter les différents protocoles et le type de données. Entre un processeur et un réseau de communication, ces adaptateurs de communication peuvent être complexes avec une partie logicielle (pilotes du processeur) et une partie matérielle (composants d'interface).

Finalement, la phase de synthèse permet d'implémenter physiquement l'architecture et le logiciel sur la plateforme cible, soit un FPGA ou un ASIC (*Application Specific Integrated Circuit*).

2.3.2 Inconvénients de l'approche classique

Cette approche classique amène un grand nombre de problèmes, que nous expliquons ici :

- Des flots de conception séparés pour le logiciel et le matériel demandent des spécialistes des deux domaines. Généralement, la formation d'un programmeur logiciel ne lui permet pas d'accomplir adéquatement les tâches d'un concepteur de matériel, et inversement. De plus, la séparation des flots peut provoquer une intégration difficile des deux parties si certaines spécifications d'un côté ou de l'autre ont changé en cours de développement. Finalement, la communication entre les deux groupes peut s'avérer difficile et provoquer des erreurs lors de la conception.
- La phase de partitionnement est généralement réalisée avant la conception du logiciel et du matériel et leur intégration. Une fois le partitionnement fait, l'architecture reste généralement figée. Cela peut amener un partitionnement sous-optimal qui peut demander beaucoup de ressources à corriger (nouvelle itération des flots de conception du logiciel et matériel, nouvelle phase d'intégration, etc.)
- La conception de matériel est généralement une étape plus longue que la conception du logiciel pour la même fonctionnalité à implémenter. Les outils de développement de matériel sont généralement d'un plus bas niveau d'abstraction que pour le logiciel. Pour cette raison, dans certains systèmes embarqués, la majorité des fonctionnalités sont réalisées en logiciel, afin de réduire le temps, donc le coût, du développement.
- Une fois que la spécification fonctionnelle de haut niveau est produite, elle doit être raffinée par les flots de conception du logiciel et du matériel. Cela ajoute des étapes supplémentaires qu'il serait utile d'éviter, en produisant un raffinement automatique à partir du modèle de haut niveau. Le temps de conception se verrait grandement réduit.

2.4 État de l'art

De nombreux travaux ont été effectués dans le domaine des systèmes embarqués afin de faciliter leur conception.

Différents projets industriels proposent des outils de conception de systèmes embarqués pour différentes étapes. Des outils de Synopsys [16] sont disponibles principalement pour la vérification et l'analyse des modèles, mais aussi pour l'implémentation de ces modèles sur le matériel. D'autre part, Mentor Graphics [17], Xilinx [18] et Altera [19] proposent des outils pour faciliter les flots de conception du logiciel et du matériel, mais aucun outil ne permet un flot de développement unifié. Les outils de ces compagnies ne s'appliquent généralement qu'à une étape ou une dimension du flot de conception, comme la modélisation ou la vérification, ou le flot logiciel ou matériel. Une méthodologie alternative est utilisée par l'outil commercial XPRES Compiler de Tensilica [20]. À partir d'une application C/C++, un processeur Xtensa est automatiquement personnalisé afin de pouvoir exécuter l'application plus rapidement. Des fonctionnalités réalisées en matériel sont ajoutées au processeur pour accélérer l'application.

D'importants projets universitaires tentent aussi de proposer des solutions au problème, mais, encore une fois, ne réalisent pas un flot automatisé de la modélisation à l'implémentation. Le projet Ptolemy [21] de l'Université de Californie à Berkeley s'intéresse principalement à la modélisation et à la simulation des systèmes embarqués, donc à la partie supérieure du flot de développement classique. Des outils permettent une modélisation de systèmes basés sur différents modèles de calculs, comme les machines à états finis, les événements discrets, le temps continu, etc. Le projet BRASS [22], aussi de l'Université de Californie à Berkeley, effectue des recherches principalement dans la partie inférieure du flot de développement, puisque l'un des buts principal du projet est l'amélioration des architectures reconfigurables et de la synthèse pour celles-ci.

D'autres travaux, tentent comme nous de réaliser un flot complet de spécification à partir de langages de haut niveau, suivi d'une génération automatique ou semi-automatique du logiciel et du matériel.

Un effort visant à spécifier un système dans un seul et même langage est présenté dans [63]. Dans cet article, les auteurs explorent la possibilité d'utiliser le langage Java comme

langage de haut-niveau pour la spécification fonctionnelle du système. Ils expliquent que les langages utilisés traditionnellement pour la cette étape de la conception, notamment C et C++, ne disposent généralement pas de moyens pour identifier la concurrence et permettent des références indirectes à la mémoire, rendant l'analyse de ces programmes difficile. Ils démontrent aussi que les langages comme VHDL ou Verilog ne sont pas des alternatives valides, puisqu'ils ne disposent pas de constructions d'assez haut niveau pour exprimer les concepts logiciels. Le langage Java, quant à lui, est indépendant de l'architecture, permet d'exprimer explicitement la concurrence et est relativement rapide. Ils démontrent ensuite comment l'analyse d'une spécification écrite en Java est effectuée, à l'aide de techniques de compilation classiques, afin de faciliter l'étape de partitionnement logiciel / matériel. Toutefois, cette recherche n'est qu'un point de départ puisque la synthèse vers le matériel n'est pas réalisée à partir de la spécification de haut niveau. Aucune autre publication n'indique l'état du projet.

Dans [42], une approche semblable est utilisée, alors que Java est le langage de spécification de haut-niveau. Ils expliquent que les langages de programmation de haut-niveau comme Java sont des langages plus expressifs et de niveau d'abstraction plus haut, amenant un cycle de développement rapide. L'outil développé, dénommé Galadriel, permet la spécification, le partitionnement logiciel / matériel automatique et la synthèse en VHDL de niveau RTL des composants matériels. L'architecture visée est un processeur générique qui exécute le logiciel, connecté à une plateforme reconfigurable pour l'implémentation du matériel. Seules les parties de contrôle des fonctionnalités en matériel doivent être synthétisées, puisque les blocs réalisant les opérations du chemin de données sont déjà pré-synthétisés et n'ont qu'à être implémentés sur le matériel. Selon les auteurs, cette recherche est différente des autres du fait que l'outil n'utilise pas des techniques de partage des ressources, mais plutôt une duplication des unités fonctionnelles, qui, selon eux, est plus efficace pour les technologies reconfigurables.

Les auteurs de [60] proposent une autre approche basée sur le langage Java. Dans leur cas, le chemin de données et la partie de contrôle doivent être générés en code VHDL de niveau RTL. Leur outil vise une architecture composée d'un processeur JOP (Java Optimized Processor), qui exécute le *bytecode* Java directement, et de logique reconfigurable pour permettre l'implémentation d'accélérateurs matériels.

Dans [58], les auteurs proposent un outil qui permet la synthèse d'un système logiciel / matériel, décrit en C, pour la puce NAPA 1000, composée d'un processeur RISC et de logique reconfigurable. À l'aide de directives *pragmas* dans la spécification du système, le concepteur peut décider d'affecter les calculs au processeur générique, pour être exécuté en logiciel, ou à la logique reconfigurable, pour être exécuté en matériel. La synthèse des deux parties est ensuite réalisée automatiquement.

Les auteurs de [91] proposent une approche un peu différente de la précédente, même si le système est encore spécifié en C. Dans ce cas-ci, un FPGA est utilisé comme une unité fonctionnelle reconfigurable plutôt que comme un co-processeur. Ces deux cas sont différents. Dans le premier, de courtes opérations peu complexes sont réalisées en matériel et la communication des données entre le processeur et le matériel spécialisé est très rapide. Un exemple est l'implémentation en matériel d'une fonction d'addition de deux registres. Dans le deuxième cas, les calculs réalisés en matériel sont plus généralement plus complexes, donc plus longs, et la communication des données entre le processeur principal et le coprocesseur est généralement de plusieurs cycles. Par exemple, il est possible d'accéder à la mémoire et de modifier le flot de contrôle. Dans ce contexte, une boucle pourrait être implémentée en matériel. Avec cet outil, les opérations prenant un grand nombre de données en entrée et une seule donnée de sortie sont celles qui seront réalisées en matériel. Ce type d'opération est identifié automatiquement. Un gain de performance important est obtenu si les opérations identifiées sont exécutées souvent. Afin de permettre un grand nombre d'opérations, le FPGA peut être reconfiguré durant l'exécution pour exécuter d'autres opérations en matériel.

Dans [41], les auteurs proposent un compilateur C qui cible l'architecture GARP [62], constituée d'un processeur MIPS et de logique reconfigurable fortement couplés, issu du projet BRASS. Le compilateur permet de détecter les boucles et de les implémenter sur la logique reconfigurable, alors que le reste des traitements est effectué sur le processeur MIPS. Un peu comme dans le projet précédent, l'outil permet différentes configurations de la logique reconfigurable qui sont chargées dynamiquement selon les calculs à effectuer.

L'environnement de compilation Nimble [71] est un outil, basé sur une version préliminaire du compilateur GARP, qui permet un reciblage de la génération de code pour

différentes architectures composées d'un processeur couplé à de la logique reconfigurable. Un langage de description d'architecture est utilisé pour spécifier la plateforme ciblée. Du code C est examiné par l'outil pour déterminer, à l'aide d'algorithmes de partitionnement spatiaux et temporels, quelles boucles devront être implémentées en matériel sur la logique reconfigurable de l'architecture et à quel moment. Des outils de synthèse permettent ensuite la génération de suites de bits pour la configuration de la logique reconfigurable et de la mémoire du processeur.

Une autre approche dynamique est expliquée dans [72]. Une architecture composée d'un processeur Microblaze, de logique reconfigurable et de profileurs permet d'obtenir des informations sur l'exécution du programme en cours. À partir d'une version uniquement logicielle d'une application, les profileurs permettent au système de prendre des décisions par rapport à l'architecture. Par exemple, une partie du programme exécutée souvent en logiciel peut automatiquement migrer vers la logique reconfigurable pour en accélérer l'exécution et réduire la consommation d'énergie. Des accélérations importantes sont obtenues à l'aide de cette technique. De plus, le partitionnement logiciel/matériel est fait de façon complètement transparente par rapport à l'usager.

Finalement, dans [76], les auteurs mettent de l'avant une méthodologie de conception un peu différente, mais toujours en partant de langages de programmation de haut niveau. L'architecture matérielle visée est encore un microprocesseur générique connecté à des unités de logique reconfigurable (les RED). Une spécification d'un système comporte trois parties : les critères de conception, qui sont les algorithmes à utiliser pour l'exploration de l'espace de conception; la description de l'architecture, en spécifiant le nombre et le type d'unités fonctionnelles; la spécification comportementale qui décrit le comportement du système en utilisant un langage de programmation de haut niveau supporté par le compilateur GCC. Une première version du système est exécutée et profilée. À partir des résultats obtenus et à l'aide d'outils de visualisation, l'utilisateur peut choisir de déplacer certaines parties du système sur les blocs reconfigurables afin d'accélérer leur exécution. Les changements aux systèmes sont ensuite faits automatiquement selon les choix.

2.5 Analyse

Les travaux présentés dans la section précédente tentent tous de faciliter la conception des systèmes embarqués. Les deux caractéristiques principales qui s'en dégagent sont l'automatisation d'une partie du flot de conception et l'utilisation de langages de haut niveau pour la spécification des systèmes. Notre méthodologie et notre outil possède ces deux caractéristiques : le compilateur développé automatise une grande partie du flot de développement et un langage de haut niveau est utilisé afin de fournir une spécification comportementale d'un système donné.

Ce qui différencie le travail accomplit par rapport aux recherches existantes sont principalement le langage de haut niveau utilisé pour la spécification des systèmes et les outils de développement utilisés. D'une part, la spécification d'un système est réalisée à l'aide d'un des nombreux langages de haut niveau supportés par le cadre d'application .NET. Ceux-ci sont compilés en un langage de niveau intermédiaire commun qui est le langage source du compilateur. Ainsi, la réalisation d'un système est accomplit à partir d'un grand nombre de langages différents. D'autre part, les outils utilisés pour le développement de l'outil facilitent l'implémentation de celui-ci et de certains traitements et analyses qu'il effectue. Ces outils de développement sont décrits dans la section suivante.

3 Outils de développement utilisés

Dans ce chapitre sont présentés les concepts et notions qui sont importants dans la compréhension de l'outil et de la méthodologie développés. En premier lieu, nous expliquons le cadre d'applications .NET, utilisée pour le développement de logiciels, et le langage CIL, qui est le langage source de notre outil. Ensuite, nous présentons le langage CASM, qui est un nouveau langage de description de matériel (HDL ou *Hardware Description Language*) de niveau intermédiaire, donc d'un niveau plus haut que bien des HDLs classiques comme VHDL ou Verilog, et qui permet une forme intermédiaire entre un langage de haut niveau, comme C# ou Java, et les HDLs précédemment mentionnés. C'est ce langage qui sert à la conception du matériel dans notre outil. Finalement, nous décrivons le Microblaze, c'est-à-dire le processeur reconfigurable utilisé comme d'exécution des applications compilées par l'outil, et les outils de développement des systèmes utilisant ce processeur.

3.1 Le cadre d'applications .NET

Le cadre d'applications .NET de Microsoft est un environnement de développement axé sur l'interaction de composants logiciels distribués. Afin de faciliter le développement rapide de ces composants, .NET permet aux programmeurs l'utilisation d'un grand nombre de langages sources provenant de différents modèles de programmation, allant de la programmation orienté-objet à la programmation fonctionnelle. Des mécanismes sont mis en œuvre afin de faciliter la communication des composants écrits dans ces différents langages. Des exemples des langages supportés sont C#, C++ géré, Java, Visual Basic .Net, Ada, Cobol, Eiffel, Perl, Scheme, et plusieurs autres. De plus, une riche bibliothèque de classes est disponible et permet d'augmenter la productivité du programmeur en fournissant de nombreux services tels que des structures de données, des mécanismes de communications ou l'introspection de programmes. .NET repose sur une spécification, le *Common Language Infrastructure*, qui définit les mécanismes permettant la mise en œuvre des fonctionnalités de l'environnement, et une implémentation de cette

spécification, comme le *Common Language Runtime*. Les explications données dans cette section sont tirées de [29].

3.1.1 Spécification du Common Language Infrastructure (CLI)

L'environnement .NET est à la base défini par une spécification, le CLI, qui fut standardisé en décembre 2001 par l'ECMA et en avril 2003 par l'ISO [29]. Le CLI est principalement une spécification du code exécutable et de l'environnement d'exécution (machine virtuelle) qui exécute ce code. En spécifiant ces règles, le CLI permet de favoriser la communication entre applications ainsi que la réutilisation de composants logiciels écrits dans différents langages de programmation, sans qu'un programmeur ait à se soucier des caractéristiques spécifiques de chaque langage.

Le CLI est composé principalement : du *Virtual Execution System*, le système qui gère l'exécution des programmes écrits pour le CLI; du *Common Type System*, un ensemble de types de données supportés par le CLI, permettant l'implémentation de la plupart des types de données de la majorité des langages de programmation; des *métadonnées*, une description des types de données du Common Type System et des types de données définis par les utilisateurs, disponibles lors de l'exécution d'un programme; des *attributs*, un mécanisme permettant d'enrichir les métadonnées et de fournir un plus grand nombre d'informations sur les types, dans des contextes particuliers; du *Common Intermediate Language*, un langage proche de l'assembleur qui est exécuté par le Virtual Execution System et vers lequel toutes les application écrites en n'importe quel langage sont compilées.

Ces concepts ainsi que d'autres dignes de mention sont présentés dans les sections suivantes.

Virtual Execution System (VES)

Le VES est responsable de la gestion et de l'exécution d'applications écrites pour le CLI. Il procure les services nécessaires à la vérification et à l'exécution du code, à la gestion des données, à l'utilisation des métadonnées pour connecter des composants générés séparément durant l'exécution, etc. Il contient des caractéristiques de multiples langages de programmation servant à supporter principalement un paradigme orienté-objet, mais aussi d'autres paradigmes comme la programmation fonctionnelle. Dans le

rôle qu'il joue à l'intérieur du CLI, le VES est comparable à la machine virtuelle de Java (JVM) [26] dans la spécification de Java.

Le VES est basé sur un modèle de machine à pile. Ce modèle de machine est plus général que celui des machines à registres (comme la majorité des processeurs généraux contemporains), puisque celui-ci n'est pas lié à une architecture particulière (toute machine possédant une mémoire peut implémenter un modèle de machine à pile). Il permet normalement d'exprimer les programmes de façon élégante et compacte et permet une plus grande portabilité. Dans ce modèle, les données résident en mémoire et peuvent être transférées sur la pile; les opérandes transférés sur la pile sont consommés lors des opérations; ces opérations produisent potentiellement de nouvelles opérandes qui sont ajoutés au dessus de la pile. Les instructions du Common Intermediate Language sont exécutées sur cette machine à pile virtuelle.

Le VES contrôle de multiples fils d'exécution concurrents lors de l'exécution d'un programme. Chaque fil d'exécution est considéré comme un enchaînement d'*états de méthode*. L'ensemble des fils d'exécution est appelé l'*état global* de la machine.

Un état de méthode décrit l'environnement dans lequel une méthode s'exécute. On pourrait comparer ce concept aux blocs d'activation en terminologie de compilation conventionnelle. Un état de méthode est composé :

- D'un pointeur d'instructions. Ce pointeur indique l'instruction suivante à être exécuté.
- De la pile d'exécution. La pile est toujours vide lors de l'entrée dans une méthode. Son contenu est toujours local à la méthode courante et son contenu est préservé lors d'appels à d'autres méthodes. Les éléments contenus sur la pile d'exécution sont d'une longueur de 32 ou 64 bits, ce qui impose une restriction sur les types de données qu'il est possible d'empiler. Par exemple, un objet ne peut être stocké sur la pile, mais la référence à cet objet peut l'être.
- D'un ensemble de variables locales. Les variables locales sont de n'importe quel type de données, mais une variable locale donnée est d'un type déterminé.
- D'un ensemble d'arguments. Tout comme les variables locales, les arguments peuvent être de n'importe quel type, mais un argument donné a toujours un type déterminé.

- D'un descripteur de retour. Il forme le lien dynamique, c'est-à-dire une référence à la méthode appelante de la méthode courante.

Common Type System (CTS)

Le CTS est un ensemble de types de données qui supporte les types et les opérations de la majorité des langages de programmation et permet ainsi l'implémentation de la plupart des langages.

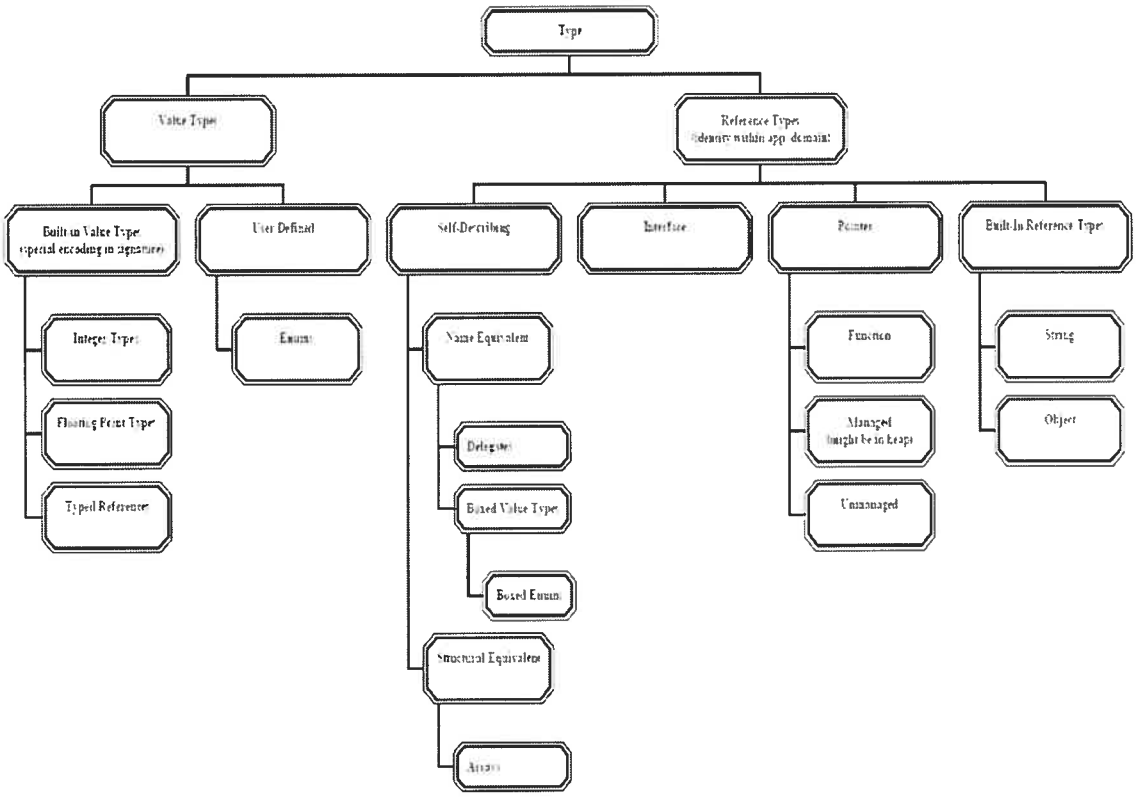


Figure 3 : Le Common Type System

Tel qu'illustré sur la Figure 3, les types du CTS sont divisés en 2 sous-types : les types *valeur* et les types *référence*. Les types valeur sont définis par l'espace de stockage qu'ils occupent et par la signification de la séquence de bits qui les représente. Ces types sont utilisés pour représenter des types simples comme les nombre entiers ou les nombre en virgule flottante. Les types référence sont une séquence de bits qui décrivent un emplacement. Il existe 4 types principaux de types référence : les objets, les interfaces, les

pointeurs et les types prédéfinis. De nouveaux types référence peuvent être introduits dans le CTS par un utilisateur, lorsque des classes, des interfaces, etc. sont définis.

Les types valeur peuvent être transformés en types référence en subissant une opération dite d'*emboîtement*. Une fois emboîté, un type valeur est traité en tout sens comme un type référence et est représenté comme un type référence contenant une copie de la valeur du type original. Pour les types références correspondant aux types valeurs, une opération de *déboîtement* est possible, le re-transformant en type valeur. Cette opération est inexistante pour les types références ne correspondant pas aux types valeurs.

Métadonnées

Afin de décrire et de référencer les types du CTS, ainsi que d'introduire de nouveaux types dans le CTS, les métadonnées sont utilisées. Celle-ci donnent des informations sur les donnée (ou des *données sur les données*) et permettent au VES de comprendre comment utiliser les types décrits et ce qu'ils représentent. Les métadonnées sont stockées de manière à être indépendantes des langages de programmation. Elles sont alors utilisées comme un moyen d'échange des données entre des outils manipulant des programmes (compilateur, déverminer, etc.) ainsi qu'entre ces outils et l'environnement d'exécution du programme. Elles permettent aussi aux programmeurs d'ajouter des attributs dans leur codes, qui seront caractérisés par des métadonnées dans le contexte courant (classe, méthode, etc.). Les métadonnées sont accessibles lors de l'exécution d'un programme, ce qui a pour effet que celui-ci peut récolter de l'information sur sa structure et sur les types utilisés. Ce mécanisme est l'*introspection*. De plus, les métadonnées sont à la base de l'interopérabilité des langages dans l'environnement d'exécution. Des types complexes définis dans un langage quelconque sont utilisables par un autre composant écrit dans un autre langage, puisque les types se décrivent eux-mêmes; les langages utilisant ces types ont alors toute l'information nécessaire pour savoir comment se servir de ceux-ci.

Attributs

Dans le code source de haut niveau, il est possible pour le programmeur d'annoter certaines structures à l'aide des *attributs*. Ces attributs ont pour utilité de définir des métadonnées sur la structure à laquelle ils sont attachés. Des attributs prédéfinis existent, mais le programmeur a la possibilité de définir ses propres attributs qui sont ensuite être

interprétés comme bon lui semble; les attributs peuvent aussi être totalement ignorés. Les structures qui peuvent être annotées comprennent les classes, les méthodes, les champs et la valeur de retour d'une méthode.

Common Intermediate Language (CIL)

Le CIL est le langage employé par la machine virtuelle du CLI. Il est comparable au bytecode de Java [26]. C'est un langage orienté-objet de bas niveau, indépendant de la plateforme sur lequel il s'exécute et indépendant des langages de programmation. La machine virtuelle est à base de pile, c'est-à-dire qu'elle ne possède qu'une mémoire pour y stocker les données et une pile où y effectuer des opérations. Les instructions du CIL sont donc définies par la transition de l'état de la pile, le traitement effectué sur les opérandes et les erreurs qui peuvent survenir lors de l'exécution de l'instruction (i.e. les exceptions pouvant être lancées).

L'ensemble des types de données sur lequel opère le CIL, dits les types CIL de base (énumérés dans le Tableau I), est un sous-ensemble restreint du CTS.

Type	Sous-type	Description
bool	Valeur	Entier de 8 bits; valeur de 0 égal faux; toute autre valeur égal vrai.
int8	Valeur	Entier de 8 bits; version signée ou non signée.
int16	Valeur	Entier de 16 bits; version signée ou non signée.
int32	Valeur	Entier de 32 bits; version signée ou non signée.
int64	Valeur	Entier de 64 bits; version signée ou non signée.
native int	Valeur	Entier de taille propre à l'implémentation.
float32	Valeur	Flottant simple précision.
float64	Valeur	Flottant double précision.
o	Référence	Référence à un objet, sans distinction sur le type étant référencé; aucune opération arithmétique permise.
native unsigned int	Référence	Pointeur non géré; pointeur C/C++ traditionnel; code non vérifiable.
&	Référence	Pointeur géré; référence à une variable locale, un argument, un champ, etc.; pas équivalent à une référence; code vérifiable.

Tableau I : Types de données du CIL

Jumelé aux métadonnées et au CTS, le CIL rend le CLI capable d'accepter un grand nombre de langages de haut niveau. En effet, un langage de haut niveau respectant les normes établies par le CLI est compilé en CIL; les types prédéfinis sont supportés par le CTS; les métadonnées ont la capacité de décrire les nouveaux types définis, la façon de les utiliser et de les compiler en code natif et ce, indépendamment du langage source.

Gestion automatique de la mémoire

Dans le CLI, la gestion de la mémoire n'est pas la responsabilité du programmeur. La récupération automatique de la mémoire est réalisée par un ramasse-miettes, géré par le VES. La récupération de la mémoire est déclenchée lorsqu'il n'y a plus de mémoire disponible pour l'allocation dynamique d'un objet.

Assemblage et exécutables portables

Un *assemblage* est l'unité logique de base de déploiement de code dans le CLI. Vu de l'extérieur, un assemblage est une collection de services et de types réutilisables par d'autres assemblages. Vu de l'intérieur, un assemblage est composé :

- D'un manifeste qui décrit l'assemblage lui-même, c'est-à-dire les types définis, les fichiers composant l'assemblage, les ressources utilisées et les autres assemblages référencés. La responsabilité du manifeste consiste à rendre l'assemblage auto-descriptif.
- Des métadonnées des types de l'assemblage.
- Du code CIL qui implémente les types de l'assemblage.
- Des ressources utilisées par l'assemblage, tels des images ou d'autres types de fichier.

L'environnement d'exécution sert à charger dynamiquement les assemblages lors de l'exécution d'un programme, donnant ainsi accès à de nouveaux types et services définis dans ces assemblages. Il est aussi possible de créer dynamiquement des assemblages.

Un *module* est une unité logique qui compose une partie d'un assemblage. La relation entre assemblages et modules est simple : un assemblage est composé d'un ou de plusieurs modules. Un module est stocké dans un *exécutable portable* (*Portable Executable* ou PE), qui est le format de fichier exécutable supporté par le CLI.

Sécurité

Des algorithmes ayant pour but de sécuriser l'environnement d'exécution sont définis dans le CIL. Plus particulièrement, deux types de test sont définis : la *validation* et la *vérification*. La validation vise à assurer l'intégrité des métadonnées et du CIL contenus dans un fichier par rapport aux normes définies dans le CLI. La vérification garantit que l'exécution d'un programme ne touchera pas de zones de mémoire à l'extérieur de l'espace qui lui est alloué. Des règles sont imposées sur les séquences d'instruction CIL garantissant ainsi un accès contrôlé à la mémoire.

Les tests de validation et de vérification permettent conjointement de garantir qu'un programme en état d'exécution ne puisse atteindre aucune ressource pour laquelle il ne possède pas l'accès. Le CLI ne spécifie pas quand ces tests doivent être faits, ni leur comportement dans le cas d'un échec de la vérification.

Code géré

Dans le CLI, deux types de codes existent : le code géré et le code non géré. Le code géré est le code qui est pris en charge par le VES et qui utilise les services. Afin d'être géré, le code doit fournir des informations à l'environnement d'exécution, sous la forme de métadonnées. De la même façon, les données d'un programme sont gérées ou non. Des données gérées sont allouées et prises en charge par l'environnement d'exécution, contrairement aux données non gérées qui sont la responsabilité du programmeur. Des exemples concrets de code et de données gérés sont C# ou Visual Basic .Net, alors que C++ produit du code et des données non gérées (même si certains mécanismes permettent de produire du code géré en C++). De plus, certains mécanismes de communication rendent le code géré et le code non géré capables d'interagir ensemble. Ces mécanismes sont à la base de l'interopérabilité des langages et permettent, par exemple, à des applications en code natif de communiquer et d'interagir avec des applications s'exécutant dans l'environnement d'exécution de .NET.

3.1.2 Common Language Runtime (CLR)

Le CLR est une implémentation commerciale de l'environnement d'exécution virtuel (machine virtuelle) défini dans la spécification du CLI. C'est l'environnement dans lequel

s'exécutent les programmes écrits en CIL et décrits dans les assemblages. Le CLR doit donc implémenter sur une plateforme particulière le modèle de machine à pile du CLI et les services fournis par l'environnement d'exécution. Dans le cas du CLR, contrairement à l'implémentation de la machine virtuelle de Java originale, les instructions CIL ne sont pas interprétées, mal bel et bien compilées dynamiquement lors de l'exécution (comme dans l'implémentation HotSpot de Java [24]).

Plusieurs services sont offerts par le CLR, notamment :

- La gestion du code (chargement, compilation dynamique et exécution)
- L'accès aux métadonnées
- La gestion automatique de la mémoire
- La gestion de la sécurité
- La gestion des exceptions
- Le support pour les services aux développeurs (dévermineur, profilage, etc.)

Un grand nombre de classes prédéfinies est disponible pour permettre aux utilisateurs d'ajouter des fonctionnalités importantes à leurs programmes sans avoir à les concevoir eux-mêmes. Grâce aux mécanismes d'interopérabilités des langages, ces classes sont utilisables à partir de n'importe lequel des langages se conformant aux normes du CLI, donc qui est supporté par la plateforme .NET. Parmi les fonctionnalités de bases, on retrouve :

- L'accès aux types prédéfinis et à l'environnement d'exécution.
- Des structures de données de base et les mécanismes pour en créer de nouvelles.
- Divers mécanismes d'entrées / sorties.
- La création dynamique de code.
- La gestion de fils d'exécution concurrents.
- L'accès aux métadonnées.

D'autres implémentations du CLI ont été réalisées, notamment ROTOR [15], Mono [14] et DotGNU [10].

3.1.3 Flot de compilation et d'exécution avec .NET

Dû à l'utilisation du CIL comme langage intermédiaire entre les langages de haut niveau et le langage machine, la compilation et l'exécution de programmes sous .NET est quelque peu particulière par rapport à une compilation traditionnelle avec des langages

comme C. La Figure 4 montre le flot de développement d'une application dans cet environnement.

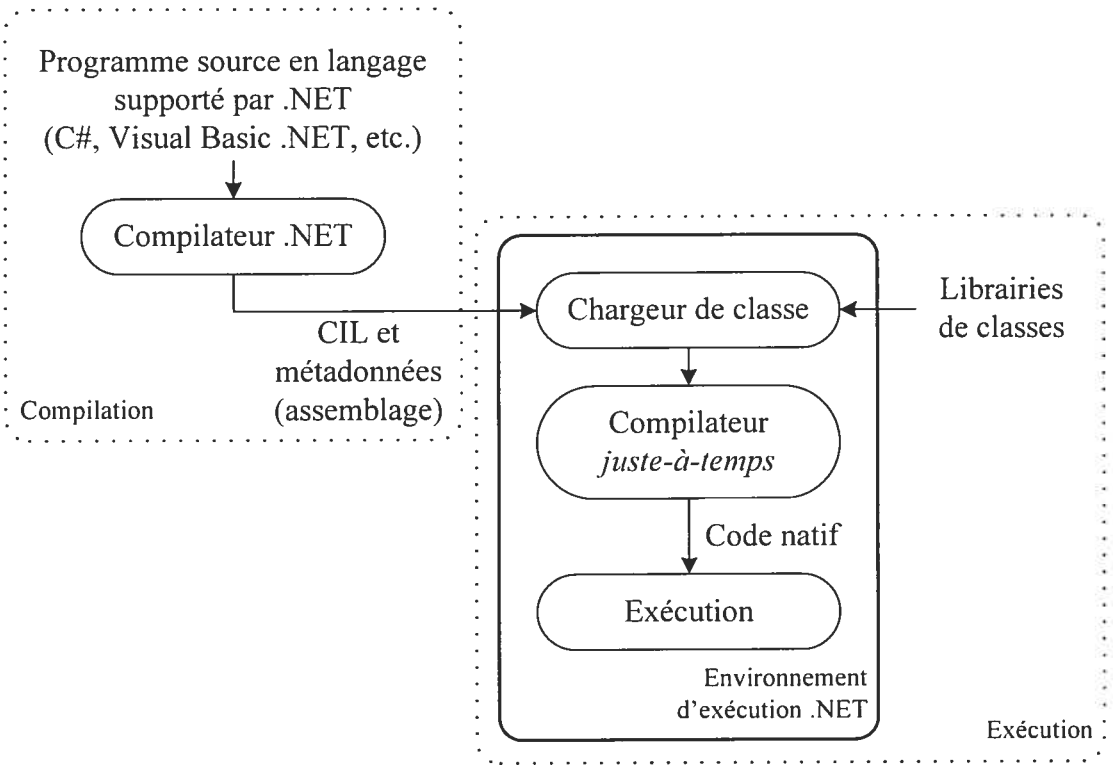


Figure 4 : Flot de développement avec .net

Un programme est au départ écrit dans un langage de haut niveau supporté par la plateforme .NET, tel C#, Visual Basic ou Scheme. Un compilateur du langage source conforme aux normes du CLI, tel le compilateur du CLR dans le cas de C# par exemple, transforme ce programme en un assemblage. Certaines optimisations peuvent être effectuées à ce moment, tel l'élimination du code mort ou certaines optimisations de contrôle de flot. L'exécutable ou la librairie créée ainsi contient le CIL et les métadonnées.

Lors du lancement du programme, le CLR prend en charge l'exécution. L'assemblage est chargé en mémoire. Trois types de compilation dynamique sont alors possibles :

- *Pré* juste-à-temps : La totalité du code CIL est compilée en code natif d'un seul coup. L'exécution de ce code, une fois la phase de compilation terminée, est assez rapide. Cependant, comme le code n'a pas accès à un grand nombre d'informations

sur l'exécution, un plus grand nombre d'indirections en mémoire sont nécessaires. L'efficacité du code généré est sous-optimale;

- Juste-à-temps *économe*: Le code est compilé très rapidement, mais le code natif produit n'est pas très optimisé;
- Juste-à-temps *standard*: Une méthode est compilée en code natif lors de son premier appel. Le code natif est conservé dans un cache et lors d'appels subséquents à cette méthode, le code natif généré est réutilisé. Le temps de compilation est relativement long, mais le code produit est hautement optimisé et seules les méthodes utilisées sont compilées.

Il est important de noter que comme dans tous les systèmes de compilation dynamique, le temps de compilation doit être amorti sur le temps d'exécution du programme pour que les gains de performance soient notables.

3.1.4 Mécanisme de réflexion

Les métadonnées sont utilisées pour donner des informations sur les données d'un programme. Il est possible de les consulter statiquement (avant l'exécution du programme) ou dynamiquement (lors de l'exécution) pour obtenir de l'information relative aux types définis et utilisés dans ce programme. Ce mécanisme d'introspection, nommé *réflexion*, permet de grandement faciliter la compilation de programmes écrits en CIL puisque les assemblages n'ont pas à être examinés et analysés afin de retrouver les types définis dans ceux-ci; toutes les informations nécessaires sont disponibles grâce aux librairies donnant accès à la réflexion. Les phases d'analyses traditionnelles de la partie frontale du compilateur sont donc grandement simplifiées. Plus précisément, les fonctionnalités de réflexions importantes pour la compilation sont les suivantes :

- `Assembly.Load` : Permet de charger en mémoire un assemblage afin de l'examiner.
- `GetModules` : Retourne les modules associés à un assemblage.
- `GetTypes` : Retourne un tableau des types définis dans l'assemblage. Les types sont des objets `Type`. Sur ces types, de nombreux détails peuvent être obtenus :
  - `Name`
  - `FullName`
  - `Namespace`

  - `IsClass`
  - `IsInterface`
  - `IsAbstract`
  - `IsCOMObject`
  - `IsEnum`
  - `IsSealed`
  - `IsPublic`.
- `GetMethods` : À partir d'un type, permet d'obtenir les méthodes définies dans ce type sous la forme d'objets `MethodInfo`. Sur les méthodes, il est possible d'obtenir les informations suivantes :
  - `Name`
  - `IsPrivate`
  - `IsPublic`
  - `IsStatic`
  - `IsConstructor`
  - `ReturnType`
  - `GetParameters`
  - `Invoke`.
- `GetConstructors` : À partir d'un type, permet d'obtenir les constructeurs définis dans ce type sous la forme d'objets de type `ConstructorInfo`. Les constructeurs sont un type particulier de méthodes, mais les informations supplémentaires que l'on obtient sont à peu près les mêmes que pour les méthodes.
- `GetFields` : À partir d'un type, permet d'obtenir les champs de ce type sous la forme d'objets de type `FieldInfo`. Sur ces champs, on obtient :
  - `Name`
  - `FieldType`
  - `IsPublic`
  - `IsPrivate`
  - `IsStatic`
  - `GetValue`
  - `SetValue`.

Pour la compilation, un élément important reste impossible à obtenir directement par les fonctionnalités de réflexion de .NET : le code source des méthodes en CIL. Il en est ainsi pour des raisons de sécurité et de protection des propriétés intellectuelles. Le code est

stocké dans l'exécutable portable du module correspondant, sous un format binaire. Cependant, un grand nombre d'outils développés par la communauté sont disponibles sur l'Internet afin de lire et de charger le code des méthodes en mémoire [23].

3.2 Le langage CASM

3.2.1 Les langages de description de matériel

Les langages de description de matériel rendent possible la conception de nouvelles architectures pour les technologies reconfigurables. Typiquement, un système est décrit dans un langage comme VHDL ou Verilog, validé à l'aide d'outils de simulation puis synthétisé sur un FPGA.

Le problème de ces langages est leur bas niveau d'abstraction. Un système numérique est généralement décrit par des circuits élémentaires connectés entre eux. À ce niveau, il devient rapidement difficile de gérer la complexité d'un système. De nombreux efforts ont été mis en œuvre pour augmenter le niveau d'abstraction des langages de description de matériel. Beaucoup de travaux ont été réalisés à partir du langage C, tels Handel-C, HardwareC et Transmogrifier C. Par contre, les langages comme C ont une richesse sémantique qui rend difficile la preuve formelle d'un système correct. Une autre approche consiste à modéliser un système à un niveau encore plus élevé. SystemC et SystemVerilog sont des langages qui supportent une telle modélisation. Cependant, une fois le système validé, il est difficile d'obtenir une description RTL du système qui est synthétisable.

3.2.2 CASM

Le langage CASM (Channel-Based Algortihmic State Machines) [46] est un nouveau langage permettant l'expression d'un algorithme à un haut niveau d'abstraction, un peu de la même manière qu'avec la synthèse comportementale [13]. Lors de l'implémentation d'un algorithme, un programmeur est intéressé par les calculs et les traitements effectués par l'algorithme; pas par la façon dont il est implémenté. CASM répond à cette attente, puisqu'il est basé sur le concept de machines algorithmique à états (*Algorithmic State Machine*, ASM) [27]. Un ASM est décrit par un graphe dirigé, où les nœuds représentent des états, des opérations ou des conditions déterminant les prochaines opérations ou le

prochain état. La Figure 5 montre un exemple de l'ASM pour l'algorithme du plus grand commun diviseur d'Euclid (PGCD).

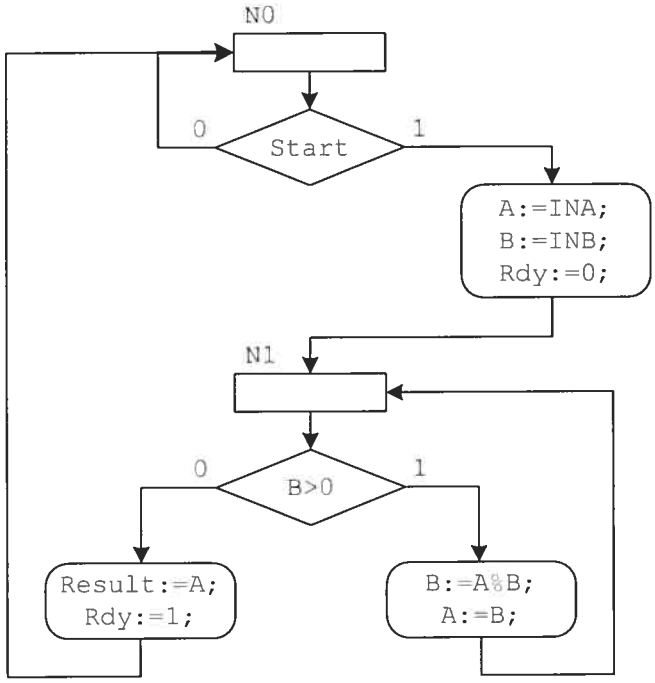


Figure 5 : ASM du PGCD d'Euclide

Un circuit séquentiel est décrit par un ou plusieurs ASM. La façon dont est implémenté l'ASM ne fait pas partie de la description. Avec CASM, un ASM est décrit de façon textuelle, en spécifiant les entrées, les sorties, les états et les opérations à l'intérieur des états, et est synthétisé automatiquement sur un FPGA à l'aide du compilateur CASM et d'outils de synthèse classiques. Le langage est de plus haut niveau que les HDL classique comme VHDL ou Verilog, mais reste de plus bas niveau que SystemC ou SystemVerilog, par exemple.

Un des buts de CASM est d'être utilisé comme langage de niveau intermédiaire entre des langages de haut niveau comme Java ou C# et les langages de description de matériel, ce qui facilite la synthèse automatique d'algorithmes écrits à un haut niveau. Des constructions additionnelles sont nécessaires pour enrichir le concept d'ASM afin de supporter certaines constructions des langages de haut niveau. Ainsi, le passage d'une description d'un système à partir d'un langage de haut niveau vers CASM sera réalisée

relativement facilement et le passage automatique de CASM vers du VHDL synthétisable s'effectue automatiquement.

3.2.3 Concepts et caractéristiques de CASM

Le langage CASM permet de décrire un ASM qui réalise un algorithme de calcul, à un haut niveau d'abstraction. La description d'un ASM est précise au niveau des cycles. De plus, certaines constructions de plus haut niveau possèdent l'avantage de réduire l'écart entre les langages de haut et de bas niveau. Le langage joue le même rôle que la synthèse comportementale dans le cycle de développement de systèmes, mais rend possible l'implémentation d'autres concepts, comme la récursivité, ou l'automatisation de certaines tâches, comme la synchronisation automatique des communications entre éléments.

Voici les principales caractéristiques de CASM :

États : Les états d'un ASM contiennent les opérations à exécuter et les conditions définissant dans quel état la machine se retrouve prochainement. Un état détermine les opérations à effectuer et ce, parallèlement. Un transfert d'un état à un autre se fait à l'aide d'un *goto*, en spécifiant l'état suivant. Le premier état de la description d'un ASM est l'état de départ. Dans une description graphique d'un ASM, les nœuds représentent les états, alors que les arcs représentent les transferts d'un état à un autre.

Données : Un algorithme effectue des opérations sur des données. Des canaux d'entrée/sortie sont utilisés pour les données arrivant à et provenant d'un ASM. Des données locales à un ASM et même à un état peuvent être utilisées et sont représentées par des composants comme des registres. De plus, il est possible d'accéder à une valeur passée d'une donnée.

Appel d'état : Dans un ASM classique, un état ne peut pas être appelé explicitement et un état de retour ne peut pas être spécifié, puisque l'ASM ne peut se souvenir de son exécution. Cela est possible en CASM. Un état peut être appelé en spécifiant l'état de retour une fois que les traitements dans l'état appelé sont terminés. Cela donne place à l'implémentation des appels de fonction des langages de haut niveau et la récursivité.

Connexions et transferts : Les connexions et les transferts sont les différents modes de transmission de données. Une connexion, définie syntaxiquement par le symbole =, permet la transmission d'une donnée à chaque cycle d'horloge pour la durée de l'état,

d'une source vers un puit si ce dernier est prêt à recevoir la donnée. Un transfert, défini par :=, oblige la transmission d'une et d'une seule donnée de la source vers le puit durant un état. Si le puit n'est pas prêt à recevoir la donnée transmise, l'ASM reste dans l'état courant aussi longtemps que la donnée n'a pas été transmise.

Synchronisation implicite : La synchronisation des transferts de données est importante dans les systèmes numériques. CASM se prête à l'abstraction de cette notion en implémentant des protocoles permettant la synchronisation automatique des transferts de données, autant pour les entrées et les sorties que pour les transferts des données à l'intérieur d'un état et d'un état à un autre. Trois protocoles sont disponibles :

- Pleine synchronisation (full synchronized). L'émetteur et le récepteur doivent être prêts pour que le transfert de la donnée soit fait. Le transfert n'est effectué que lorsque les deux entités concernées sont prêtes.
- Demi-synchronisation (half synchronized). L'émetteur transmet la donnée lorsqu'il est prêt. Si le récepteur n'est pas prêt, la donnée est perdue.
- Aucune synchronisation.

La réalisation des opérations de transfert et de connexion repose sur les protocoles de synchronisation prédéfinis. Dans un système multi-horloge, des adaptateurs sont automatiquement générés pour synchroniser le transfert des données.

Structures prédéfinies : Certaine structures de données communes sont implémentées par le langage. Des exemples de structures disponibles sont les queues, les piles et les mémoires.

Modèle : Une description CASM est un ensemble de ressources connectées par des canaux synchronisés qui réalisent les transactions. Le comportement d'une ressource est décrit par un ASM. La hiérarchie est supportée, ce qui signifie qu'un ASM peut en instancier un autre pour réaliser certaines fonctionnalités.

Sémantique : Une description CASM comporte des canaux implicites pour la réinitialisation de l'ASM et pour l'horloge. Le canal de réinitialisation remet un ASM à son état de départ et initialise les composants à leur valeur par défaut. Lors d'un événement sur le signal de l'horloge, les étapes suivantes sont exécutées :

1. Les expressions conditionnelles sont évaluées et les transactions concernées sont activées.

2. Les r-valeurs (expression de droite d'une affectation) sont évaluées en parallèle;
3. Si toutes les transactions sont complétées dans le cycle courant, le saut au prochain état est activé.
4. L'attente du prochain cycle d'horloge.

Ce qu'il est important de retenir est que toutes les opérations d'un état sont exécutées en parallèle et que le saut à l'état suivant n'est exécuté que si toutes les transactions relatives aux opérations de l'état courant ont étés complétées.

3.3 Le processeur Microblaze

Le processeur Microblaze [28] est la cible pour laquelle le code CIL réalisé en logiciel est compilé. Nous décrivons son architecture et les caractéristiques importantes.

3.3.1 Architecture

Le processeur Microblaze est un cœur de processeur logiciel (*softcore processor*) de Xilinx, synthétisable sur FPGA et optimisé pour ceux-ci. Les caractéristiques principales du processeur sont les suivantes :

- Un jeu d'instructions de type RISC.
- 32 registres généraux.
- Des instructions de 32-bit, à trois opérandes et deux modes d'adressage.
- Un pipeline à trois étages (étapes de recherche, décodage et exécution d'une instruction).
- Une séparation des mémoires de données et d'instructions.

La Figure 6 illustre l'architecture du Microblaze.

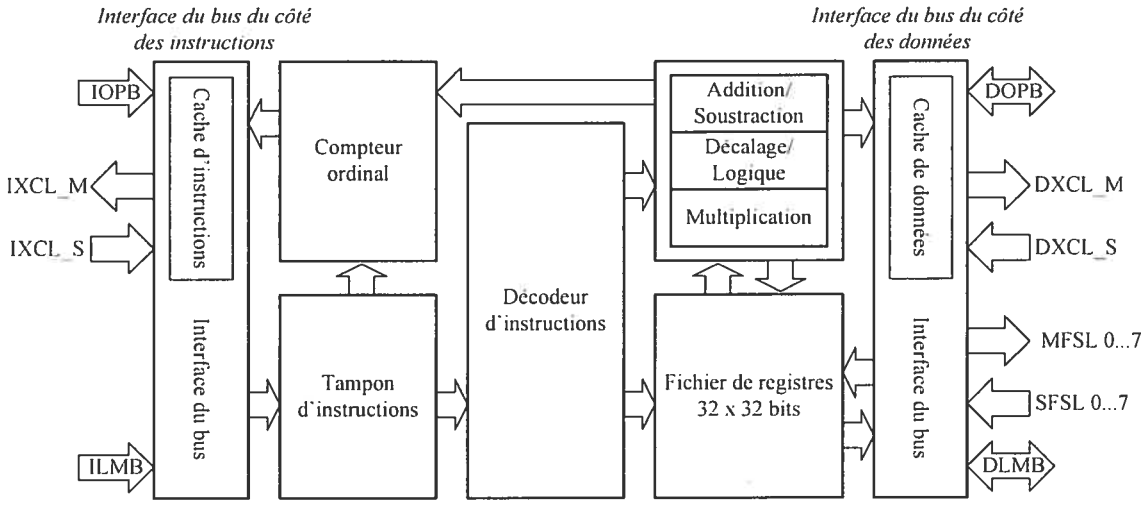


Figure 6 : Architecture du Microblaze

3.3.2 Configurabilité et extensibilité

La nature logicielle du processeur Microblaze donne plusieurs avantages et parmi ceux-ci, la *configurabilité* et l'*extensibilité*. Ces deux caractéristiques sont inexistantes dans un système où l'implémentation du processeur est matérielle et qui, en, conséquence, a une configuration fixe et immuable.

La *configurabilité* permet à l'utilisateur de choisir la configuration d'un système Microblaze lors de la conception d'un système utilisant le processeur. Il est possible d'ajouter et d'enlever des composants du système afin de créer un système sur mesure par rapport aux besoins.

L'*extensibilité* est la possibilité qu'a l'utilisateur de concevoir des nouveaux composants matériels qui sont alors ajoutées au système Microblaze. Des mécanismes de connexion sont prévus afin de faciliter la communication entre les composants existants et les nouveaux composants. Parmi ces mécanismes, on retrouve l'interface Fast Simplex Link (FSL) qui offre une communication rapide et presque directe entre le processeur et un module matériel. Plus précisément, l'interface FSL est composée d'un canal unidirectionnel allant du fichier de registres au module matériel et d'un autre canal allant du module matériel vers le fichier de registres. Les canaux disposent d'une queue de données, dans laquelle sont stockées les données lors des transferts. Les transferts sur ces canaux sont de type bloquant ou non bloquant. Une seule instruction assembleur est

nécessaire afin de transférer une donnée d'un registre vers le canal FSL, et vice-versa. En utilisant cette interface, la communication avec les modules matériels devient extrêmement simple.

D'autres mécanismes de communication avec les périphériques peuvent être utilisés, comme divers bus. Toutefois, la communication devient plus complexe puisque les périphériques doivent généralement implémenter le protocole de communication utilisé par ce canal.

3.3.3 Outils de développement

Un grand nombre d'outils sont disponibles afin de faciliter le développement de systèmes complets utilisant le Microblaze. Le Embedded Development Kit (EDK) [2] de Xilinx sert à concevoir l'architecture d'un système utilisant ce processeur à l'aide d'une interface graphique. L'utilisateur choisit d'ajouter ou d'enlever certains périphériques, spécifie le type de connexion à utiliser pour les nouveaux périphériques, choisit la taille de la mémoire sur la puce, choisit si le processeur utilise une mémoire cache, etc. Bref, il est possible, grâce au EDK, de bâtir facilement une architecture complète sur mesure pour les besoins courants. De plus, il est aussi possible de choisir si le système utilise un système d'exploitation et si oui, lequel. Un outil associé au EDK, permet de compiler des programmes écrits en C ou en assembleur, qui sont exécutés par le système conçu. Cet outil, MB-GCC, est une version de GCC [3] adapté au Microblaze. Le EDK donne accès aux librairies standard de C, qui sont liées à l'exécutable, mais aussi à un grand nombre d'autres librairies offrant l'utilisation d'un système de fichier, la connexion à l'Internet et à d'autres usages.

La Figure 7 décrit le flot de développement de systèmes à l'aide du EDK. D'un côté, le système est spécifié et synthétisé au moyen des outils. Des périphériques prédéfinis ou conçus par l'usager peuvent être ajoutés. De l'autre côté, les programmes exécutés par le système sont conçus et sont compilés par les outils du EDK. Le tout est implémenté sur le FPGA.

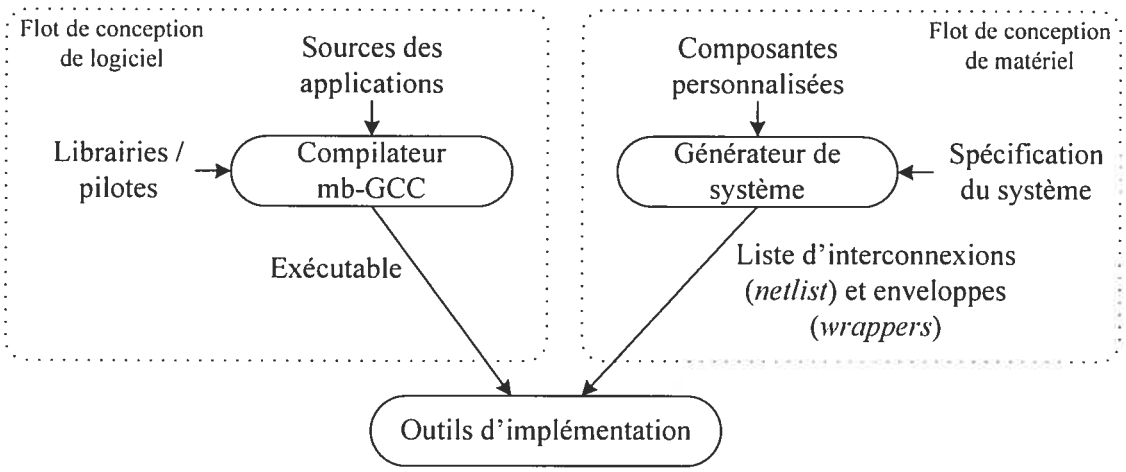


Figure 7 : Développement de système avec EDK

4 Flot de conception logiciel / matériel basé sur notre compilateur

L'outil que nous avons développé, le compilateur CIL, tente d'apporter une contribution au problème de la conception de systèmes matériel/logiciel en proposant une méthodologie différente. Dans cette section, nous débutons par décrire les caractéristiques recherchées par notre flot de conception. Nous expliquons ensuite la façon dont le compilateur développé répond aux attentes énoncées. Finalement, nous décrivons brièvement le travail de développement nécessaire à l'implémentation du compilateur.

4.1 Caractéristiques désirées d'un flot de conception

Le flot de conception que nous proposons doit combler certaines lacunes du flot de conception traditionnel des systèmes embarqués. Le flot doit répondre aux besoins suivant :

Spécification à un haut niveau d'abstraction : Nous désirons un flot où la spécification d'un système embarqué est effectuée à un haut niveau d'abstraction. Par exemple, le niveau d'abstraction des langages de programmation comme C est probablement d'un trop bas niveau. Des langages d'un niveau comparable à C++ ou Java sont de meilleurs candidats, comme dans [42, 63, 76]. Une spécification à un tel niveau d'abstraction permet l'utilisation des constructions de haut niveau du langage (comme les classes) et des mécanismes comme la gestion automatique de la mémoire, accélérant ainsi la conception du système.

Automatisation d'étapes de la conception : Il est important que les étapes du flot de conception où des erreurs sont fréquemment introduites, notamment le raffinement manuel de la spécification, soient automatisées. De cette façon, les erreurs sont éliminées et le cycle de conception est grandement accéléré.

Unification de la conception logicielle et matérielle : La séparation de la conception des parties logicielle et matérielle d'un système embarqué nécessite des spécialistes de ces deux mondes de la conception. Toutefois, dans un flot où la conception du logiciel et la

conception du matériel est unifiée, la connaissance d'un seul et même langage de spécification est nécessaire.

Nous présentons dans la section suivante notre compilateur basé sur les outils exposés dans la section 0. Celui-ci supporte un flot de conception répondant aux trois critères énoncés plus haut. L'outil permet une conception rapide, simple et sans erreur d'un système embarqué, de la spécification à l'implémentation.

4.2 Proposition d'un flot de conception à l'aide du compilateur

À l'aide de notre outil, le flot de développement classique se voit modifié : les étapes de synthèse du logiciel, du matériel et des interfaces sont regroupées dans le même outil (Figure 8).

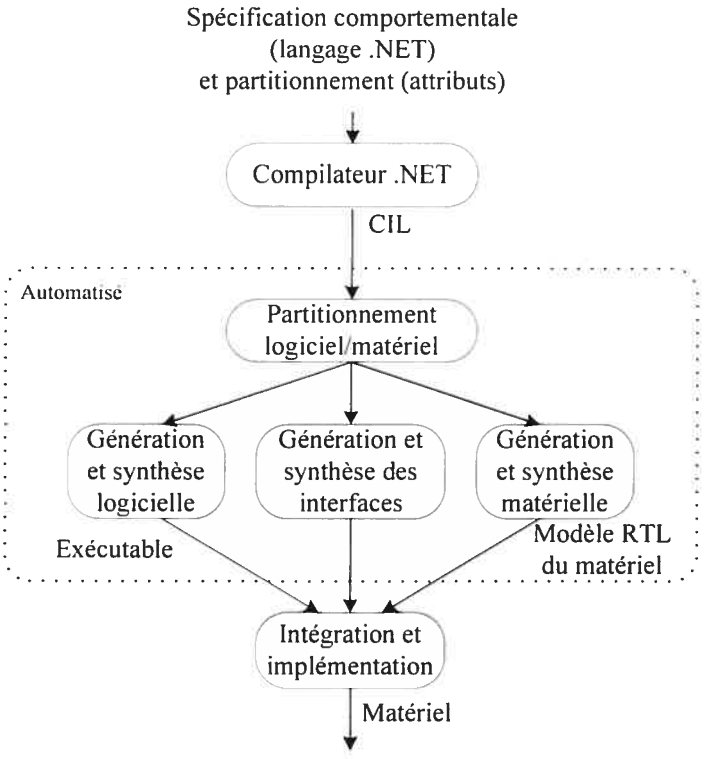


Figure 8 : Modification du flot de conception classique avec l'outil

En utilisant notre compilateur, un système entier est spécifié à l'aide d'un langage de haut niveau. Les parties logicielles et matérielles sont identifiées et l'outil génère

automatiquement les programmes destinés au processeur, les modèles RTL pour le matériel et les interfaces entre les deux parties. L'implémentation des fonctionnalités du système est alors grandement simplifiée et accélérée, ce qui amène aussi un prototypage rapide du système. Nous discutons plus en détail de la façon d'utiliser l'outil dans les sections suivantes.

4.2.1 Flot de développement d'applications

Un système développé avec le compilateur CIL utilise, en partie, le même flot de développement qu'une application conçue avec l'environnement de développement .NET. Il faut débuter par écrire un ensemble de fichiers source dans un langage supporté par la plateforme. Afin de pouvoir définir le partitionnement logiciel/matériel de l'application, il faut référencer l'espace de nom *PartitionAttributes*, qui définit deux attributs, soit *Software* et *Hardware*. Plus de détails quant au partitionnement sont donnés plus loin dans cette section. Cette application est la description fonctionnelle du système logiciel / matériel. À l'aide d'un compilateur conforme aux normes définies dans le CLI, le programme source est compilé et le CIL et les métadonnées sont générés. Le tout est stocké dans un exécutable portable, sous forme de fichier *.exe* ou *.dll*.

C'est ici que le flot de compilation et d'exécution du programme diffère du flot de .NET et que notre compilateur CIL entre en jeu (Figure 9). Le compilateur est un fichier exécutable lancé à partir d'un incitatif de commandes. Comme argument, on doit spécifier le nom de l'exécutable portable à compiler. Le langage d'entrée supporté par le compilateur est le CIL contenu dans l'exécutable portable. Il faut mentionner qu'il est possible de spécifier sur la ligne de commande quelles parties du système nous voulons générer : le logiciel, le matériel, ou les deux.

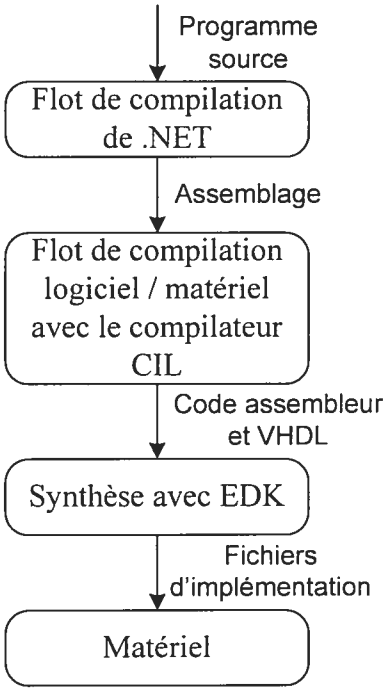


Figure 9 : Flot de développement d'applications basé sur notre compilateur

Le flot de génération du logiciel donne comme résultat un fichier contenant du code assembleur Microblaze et les données associées à ce code. À l'aide du compilateur MB-GCC, disponible avec le EDK de Xilinx, le fichier est compilé en un programme exécutable qui est exécuté par le processeur Microblaze.

Le flot de génération du matériel produit un ou plusieurs fichiers contenant les ASM construits. Ces fichiers sont donnés en entrée au compilateur de CASM, produisant plusieurs fichiers VHDL synthétisables qui réalisent les fonctionnalités des ASM. Le code VHDL est intégré au système construit à l'aide du EDK. Ce système est principalement constitué d'un processeur Microblaze pour l'exécution du logiciel et d'autres périphériques qu'un utilisateur peut ajouter. Les interfaces entre les parties logicielles et matérielles, quasi inexistantes grâce aux spécificités du processeur Microblaze, sont aussi générées automatiquement.

Le système entier est synthétisé et peut alors être téléchargé sur le FPGA.

Comme nous l'avons mentionné précédemment, l'outil n'accepte pas l'ensemble du CIL en entrée, mais seulement un sous-ensemble. Les multiples raisons de cette limitation sont expliquées plus loin, dans la section 5.1.

Partitionnement

Le partitionnement est effectué au niveau des méthodes. Celles-ci sont annotées, à l'aide d'attributs, comme étant soit logicielles, soit matérielles. Selon les options données au compilateur CIL lors de son lancement, les informations de partitionnement sont considérées ou non. Par défaut, toutes les méthodes sont identifiées comme logicielles. Ajouter un attribut identifiant une méthode comme logicielle devient redondant.

Le problème d'un partitionnement logiciel / matériel optimal n'est pas traité ici. En effet, l'outil suppose qu'une exploration architecturale a déjà été réalisée.

Étant donné que le sous-ensemble du CIL supporté par la compilation matérielle est encore plus réduit que pour celle logicielle, certaines méthodes ne peuvent être considérées dans le partitionnement.

4.2.2 Avantages de l'approche proposée

Le flot que nous proposons répond aux attentes énoncées dans la section 4.1 et comporte ainsi des avantages par rapport au flot de conception classique des systèmes embarqués. Nous démontrons ici que l'outil possède bel et bien les caractéristiques désirées et nous expliquons les avantages de notre outil, autant du côté de la méthodologie que du côté de l'implémentation.

Caractéristiques désirées

Spécification à un haut niveau d'abstraction : Les langages supportés par le cadre d'application .NET sont d'un haut niveau d'abstraction. Par exemple, le langage C# possède un grand nombre de caractéristiques semblables à C++ et à Java. De plus, seul le comportement du système est spécifié à l'aide du langage de programmation. Aucun détail d'implémentation (à part le partitionnement logiciel/matériel) n'est requis.

Automatisation d'étapes de la conception : Le raffinement des parties logicielle et matérielle est effectuée automatiquement par le compilateur pour créer, à partir du logiciel, un exécutable, et, à partir du matériel, des modules VHDL de niveau RTL à implémenter. De plus, les interfaces entre le logiciel et le matériel sont générés automatiquement. Le passage de la spécification à l'implémentation est donc très rapide et

les erreurs introduites lors du raffinement et lors de la conception des interfaces sont éliminées.

Unification de la conception logicielle et matérielle : Étant donné que, à partir de la spécification comportementale du système, le raffinement des modèles du logiciel et du matériel est automatisé, ces deux parties de la conception sont unifiées aux yeux du concepteur. Il n'est pas nécessaire de concevoir séparément les deux parties. De plus, le concepteur d'un système n'a besoin de connaître qu'un seul langage supporté par le cadre d'applications .NET pour réaliser le système en entier.

Avantages de la méthodologie

Outre les avantages découlant des caractéristiques désirées du flot, divers avantages méthodologiques résultent du flot de conception proposé. Par exemple :

- Puisque la synthèse du système est effectuée automatiquement à partir de la spécification de haut niveau, le coût du changement du partitionnement logiciel/matériel est moindre par rapport au flot classique. Il ne suffit que de changer les attributs des méthodes concernées et l'outil re-génère automatiquement le système à implémenter.
- L'utilisation d'un langage source de niveau intermédiaire, c'est-à-dire le CIL, apporte beaucoup. Comme celui-ci est indépendant du langage de haut niveau utilisé pour concevoir le système à la base, un concepteur peut utiliser le langage supporté de son choix.

Avantages de l'implémentation

Il est important de souligner les avantages d'implémentation du compilateur qu'amène le choix des outils de développement :

- La partie frontale du compilateur est grandement simplifiée par rapport à un compilateur traditionnel. Premièrement, il ne faut pas écrire d'analyseur lexical et syntaxique pour chaque langage supporté, puisque ces phases sont réalisées par les compilateurs de langage source vers le CIL. Deuxièmement, le traitement des erreurs lors de ces deux analyses est inexistant dans notre compilateur puisque cela est déjà fait par le compilateur du langage source qui génère le CIL. Grâce à cela, on peut supposer que tous les programmes écrits en CIL sont valides

syntaxiquement (ils peuvent tout de même être invalides sémantiquement). Finalement, les analyses sur l'arbre syntaxique sont aussi déjà faites.

- Durant tout le processus de compilation, le code est supposé valide syntaxiquement et grammaticalement, puisque les erreurs ont été détectées par le compilateur générant le CIL. Donc, tous les efforts de programmation du compilateur sont dirigés vers la production et l'optimisation du code généré, et non pas vers la gestion des erreurs.
- Certaines optimisations peuvent être accomplies lors de la compilation du langage source vers le CIL, comme l'optimisation du flot de contrôle ou l'élimination d'expressions redondantes. Il n'est donc plus nécessaire de réaliser ces optimisations dans le compilateur du CIL.
- Grâce à la réflexion, la création d'une représentation interne du programme en mémoire est très simple. En général, il est seulement nécessaire d'invoquer une méthode particulière qui donne l'information utile. La lecture du CIL est aussi simple grâce aux nombreux outils développés.
- Le langage CASM permet une conception facile et efficace des parties matérielles. Son niveau d'abstraction moyen permet un passage élégant d'un langage de haut niveau à du code VHDL de niveau RTL. De plus, il n'est pas nécessaire de se soucier du sous-ensemble synthétisable du langage lors de la conception du matériel, puisque la totalité du langage CASM est synthétisable, contrairement au VHDL. Le langage supporte aussi les architectures mutli-horloges.

4.2.3 Implémentation

La réalisation du compilateur nécessite l'implémentation de divers traitements réalisés dans les flots commun, logiciel et matériel. Les traitements du flot commun réalisent les fonctionnalités de la partie frontale d'un compilateur, mais la façon dont ceux-ci sont effectués est différent des méthodes classiques (c'est-à-dire l'analyse lexicale, l'analyse syntaxique, etc.). Les traitements et les analyses du flot logiciel sont responsables de la création de l'exécutable et les traitements du flot matériel créent les parties matérielles du système. Dans les deux cas, des techniques traditionnelles de compilation sont utilisées.

La section suivante décrit les traitements effectués dans les flots commun, logiciel et matériel du compilateur. Les algorithmes et les structures de données nécessaires sont aussi exposés.

5 Analyses, traitements et implémentation du compilateur

Afin de générer le code assembleur Microblaze du programme exécutable et le code CASM pour les modules matériels à partir du CIL, de nombreuses analyses et optimisations sont nécessaires. Ces traitements sont expliqués ici avec la description des algorithmes utilisés et des structures de données nécessaires.

Les traitements faits par le compilateur, démontrés dans la Figure 10, sont classés en trois groupes : le flot commun, le flot logiciel et le flot matériel. Le flot logiciel inclut les analyses nécessaires à l'obtention du code Microblaze. Le flot matériel est constitué des traitements faits pour la génération du code CASM. Le flot commun regroupe les traitements communs aux flots logiciel et matériel, qui sont réalisés avant les deux autres flots et qui leur sont nécessaire afin que ces derniers puissent être exécutés.

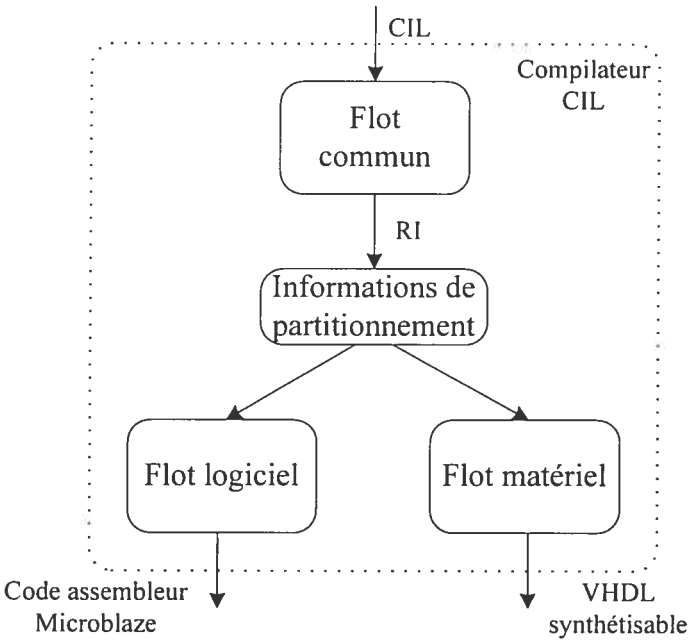


Figure 10 : Flot commun, logiciel et matériel

L'objectif du projet étant de démontrer la faisabilité et de développer un premier prototype, seul un sous-ensemble du langage CIL est actuellement supporté. Le sous-

ensemble comprend les principaux types de données les plus utilisés, les concepts du langage source les plus courants et les instructions CIL implémentant ceux-ci.

5.1 Sous-ensemble du CIL supporté

Nous décrivons ici les concepts des langages de haut niveau et les types de données qui sont supportés.

5.1.1 Environnement d'exécution vs CIL

Il est important de bien comprendre la différence entre le CIL et l'environnement d'exécution : les instructions CIL sont exécutées par l'environnement d'exécution, mais les deux sont des entités bien distinctes. Toutefois, un certain nombre de propriétés et de fonctionnalités du langage sont possibles grâce aux services de l'environnement d'exécution, comme la gestion des fils d'exécution par exemple. De plus, un très grand nombre de ces fonctionnalités sont accessibles par les classes prédéfinies de l'environnement .NET. Il est donc difficile de donner accès à ces fonctionnalités si une implémentation des classes prédéfinies n'est pas disponible.

Comme les classes prédéfinies sont elles-mêmes en CIL, il est possible de croire qu'un compilateur supportant l'ensemble du CIL compile ces classes et ce, pour l'architecture désirée. La réalité n'est pas aussi simple : les classes prédéfinies référencient beaucoup de fonctions écrites en code natif ou des fonctions du système d'exploitation. Donc, pour pouvoir accéder aux classes, il faudrait que l'environnement pour lequel la compilation est effectuée dispose de tous les mécanismes nécessaires. Or, même si le Microblaze possède un environnement de programmation relativement riche, il est évident qu'il ne peut se comparer aux systèmes d'exploitation modernes comme Windows ou Linux. L'association de fonctionnalités de l'environnement source vers l'environnement cible pourrait être le sujet d'une autre recherche.

Notre outil tente d'implémenter, au mieux de ses capacités, le plus grand nombre d'instructions CIL possibles et non pas l'environnement d'exécution. Cependant, les fonctionnalités disponibles sont grandement réduites à cause des classes prédéfinies qui ne sont pas disponibles. Cela forme tout de même une bonne base pour réaliser la construction d'un environnement d'exécution complet du CLI. De plus, des applications

complètes peuvent tout de même être développées avec le compilateur dans son état actuel.

5.1.2 Sous-ensemble supporté

Les sous-ensembles supportés pour les méthodes implémentées en logiciel et en matériel ne sont pas les mêmes : les méthodes matérielles ne supportent qu'un sous-ensemble du CIL supporté par les méthodes logicielles, qui elles-mêmes ne supportent qu'un sous-ensemble des instructions CIL.

Pour les méthodes logicielles, les concepts, les types de données ou les types d'instructions qui sont supportés sont les suivants :

- Les opérations arithmétiques et logiques.
- Les structures de contrôle.
- Les types booléens, entiers, chaînes de caractères, les objets et les tableaux à une dimension.
- La définition de nouveaux types (classes et structures).
- L'héritage.
- La création dynamique de données.

5.1.3 Caractéristiques, concepts et instructions non supportés

Nous énumérons ici les caractéristiques et concepts des programmes source qui ne sont pas supportés. Ces éléments sont regroupés en cinq catégories distinctes, incluant une catégorie pour les méthodes matérielles.

Instructions non implémentées

Certaines instructions du CIL n'ont pas été implémentées, ce qui empêche l'utilisation de certaines constructions dans les programmes qui se servent de ces instructions une fois compilées en CIL. Les instructions non implémentées et les constructions interdites sont :

- Les instructions CIL d'accès indirect aux données, ce qui empêche le passage de paramètres par référence pour les types valeurs.

- Des instructions de manipulation de la mémoire. Ces instructions sont utilisées principalement par l'environnement d'exécution. Les programmes sources ne sont pas réellement limités par ce manque.
- Certaine variations d'instructions CIL de base, comme par exemple les opérations arithmétiques avec détection de débordement.
- Les exceptions. Le CIL définit des instructions de gestion d'exceptions, mais elles ne sont pas implémentées. Toutefois, l'implémentation de ces instructions ne signifie pas automatiquement que le mécanisme d'exception est supporté; il faut aussi implémenter les fonctionnalités internes permettant la gestion des exceptions.

Les instructions qui ne sont pas supportées rendent difficile la conception de certaines applications traitées par le compilateur CIL. Les instructions d'accès indirect aux données empêchent le passage de paramètres par référence, utilisé entre autres pour l'obtention de plus d'une valeur de retour dans les méthodes. Même si ce style de programmation n'est pas toujours recommandé et peut souvent être évité, certaines applications le requièrent. Quant aux exceptions, elles ne limitent pas à proprement dit les applications possibles à traiter par le compilateur, mais empêchent l'utilisation de mécanismes élégants de gestion des erreurs.

Types de données

Certains types de données de base ne sont pas supportés tout comme les instructions utilisant ces types.

- Les nombres en virgule flottante. L'architecture du Microblaze ne supporte pas les nombres en virgule flottante. Cette architecture est cependant plus complexe dans les plus récentes versions du processeur et possède des ressources dédiées pour rendre possible l'utilisation de ces types. Employer une nouvelle version du Microblaze ou approximer les nombres en virgule flottante avec des nombres en virgule fixe sont des solutions possibles.
- Les interfaces ne sont pas implémentées, mais cela ne serait pas très difficile à réaliser.

Classes prédéfinies

L'accès aux classes prédéfinies du cadre d'application .NET est impossible à partir de programmes compilés par notre outil. D'une part, il serait impossible de les compiler étant donné qu'une partie du CIL n'est pas supporté. D'autre part, dans l'implémentation du CLR, certaines des méthodes de ces classes sont implémentées en code natif ou font appel à l'API du système d'exploitation. Notre outil n'est pas capable de traiter ces informations. Pour pouvoir utiliser ces classes, deux solutions sont à envisager : implémenter manuellement les fonctionnalités des classes, pour le Microblaze (en C et compilé avec MB-GCC, ou à l'aide d'un langage supporté par .NET et compilés par notre outil), ou implémenter l'ensemble des instructions CIL. Toutefois, dans les deux cas, le problème du code natif n'est pas réglé. En fait, la meilleure solution, mais aussi la plus complexe, est l'implémentation de toutes les fonctionnalités du CLR.

L'inaccessibilité aux classes prédéfinies empêche l'utilisation d'un certain nombre de fonctionnalités qui sont implémentées à l'aide de ces classes.

- Les délégués. Dans le CLI, un délégué représente une référence vers une méthode et permet les procédures de rappel (*callback*).
- Les programmes à multiples fils d'exécution (*threads*). Les fonctionnalités des threads sont définies par les classes prédéfinies de .NET. Pour les utiliser, il faudrait pouvoir utiliser les classes prédéfinies.
- Les tableaux multidimensionnels. Dans le CIL, la création de tableaux multidimensionnels se fait à l'aide des classes prédéfinies du CLI, contrairement aux tableaux unidimensionnels qui sont créés à l'aide d'une seule instruction CIL.

Environnement d'exécution

L'environnement d'exécution dans lequel s'exécutent les programmes .NET offre un grand nombre de fonctionnalités. Toutefois, comme il a été mentionné précédemment, notre intérêt n'est pas l'implémentation de cet environnement. Voici les éléments qui ne sont pas supportés :

- L'interaction avec le code non géré. Les accès de plateforme (*PInvoke*) permettent d'appeler des fonctions non gérées qui se trouvent dans des DLL. Cette opération requiert des services complexes qui ne sont pas implémentés par le compilateur.

- Les mécanismes de sécurité. Le CLI prévoit des algorithmes de vérification et de validation qui peuvent être lancés à l'exécution. Ces algorithmes ne sont pas implémentés car ils font partie de l'environnement d'exécution de .NET.
- L'accès aux métadonnées durant l'exécution. Les tables qui contiennent les métadonnées à l'exécution ne sont pas implémentées. Ainsi, les instructions CIL accédant aux métadonnées ne sont pas implémentées.
- La récupération automatique de la mémoire. En fait, la gestion automatique de la mémoire n'est que partiellement implémentée. L'allocation dynamique de la mémoire est possible, mais la récupération est impossible puisqu'aucun algorithme de ramasse-miettes n'est pas implémenté. Une discussion plus approfondie sur ce sujet se retrouve dans la section 5.3.1.

Méthodes en matériel

En ce qui concerne les méthodes matérielles, seules les opérations arithmétiques et les structures de contrôle sont actuellement possibles sur des types de données booléens et entiers. La création de tableaux ou objets locaux à la méthode matérielle est permise, mais le passage de ces objets en paramètres ou comme valeur de retour est impossible, puisqu'aucun mécanisme de gestion et de synchronisation de la mémoire entre le processeur et les ASM n'est implémenté. Donc, les modules matériels ont comme principale fonctionnalité d'implémenter des calculs arithmétiques. Cependant, comme il est discuté plus loin, d'autres fonctionnalités des ASM pourraient être implémentées. Aussi, les méthodes implémentées en matériel ne peuvent pas être virtuelles ; le lieu d'appel doit être déterminé statiquement. Il serait intéressant d'ajouter un support d'exécution se prêtant à l'appel de méthodes matérielles virtuelles où plusieurs versions du module matériel sont implémentées.

5.2 Flot commun

Les traitements faits par le flot commun peuvent être comparés aux traitements faits par la partie frontale d'un compilateur. C'est ici que le code source (le CIL dans notre cas) est lu et chargé en mémoire, qu'une représentation du programme est créée et que la

représentation intermédiaire du code est générée. Suite à ces traitements, les flots logiciel et matériel sont lancés en utilisant les résultats du flot commun, principalement la représentation du programme en mémoire et la représentation intermédiaire du code.

Les traitements, montrés sur la Figure 11, sont décrits dans l'ordre dans lequel ils sont effectués.

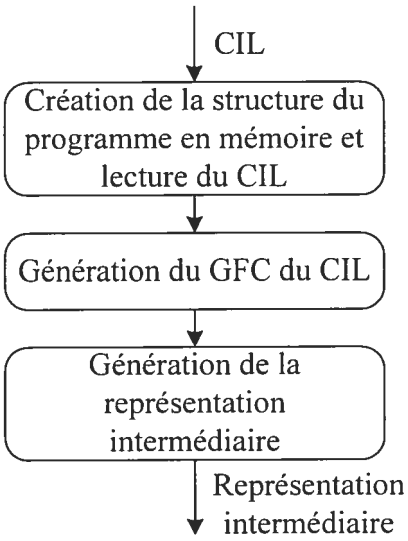


Figure 11 : Phases flot commun

5.2.1 Création de la structure du programme et lecture du CIL

Dans cette étape, nous voulons représenter le programme sous une forme qui est utilisée par le compilateur. Il faut trouver quels sont les nouveaux types définis dans le programme source (les classes, les structures, etc.) et les membres de ces nouveaux types (les champs et les méthodes), ainsi que le code des méthodes. Comme il a déjà été mentionné plus haut, la création de la structure du programme en mémoire est grandement simplifiée grâce à la réflexion fournie par la plateforme .NET.

Utilisation de la réflexion

L'assemblage correspondant au programme source est chargé en mémoire en invoquant la méthode `Assembly.Load` avec en argument le nom du fichier d'entrée donné

en argument lors du lancement du compilateur. Tous les modules utilisés dans l'assemblage sont obtenus avec la méthode `GetModules` et sont placés dans une liste.

L'accès aux champs, aux méthodes et au code des méthodes est fait à partir des modules. En parcourant séquentiellement la liste des modules de l'assemblage, on obtient les types, de type `Type`, définis dans cet assemblage par l'invocation sur le module courant de la méthode `GetTypes`. La réflexion permet encore une fois de faciliter les choses pour trouver la structure des types définis, en invoquant sur un type donné les méthodes `GetFields`, `GetMethods` et `GetConstructor` qui retournent respectivement des tableaux d'objets de type `FieldInfo`, `MethodInfo` et `ConstructorInfo`. Ces objets décrivent évidemment les champs, les méthodes et les constructeurs du type courant et fournissent l'accès aux fonctionnalités de réflexion sur ces structures. Les classes `MethodInfo` et `ConstructorInfo` sont des sous-classes de `MethodBase`. Toutes ces informations obtenues relatives aux types sont stockées dans une instance de la classe `ClassInfo`, qui est la structure de base dans le compilateur conservant l'information des types. Tous les objets `ClassInfo` sont mis dans une liste qui est le résultat de ce traitement. Cette liste reste l'élément de base de la majorité des traitements du compilateur.

De nombreuses informations nécessaires aux traitements subséquents sont obtenues par réflexion. Cependant, dans la plupart des cas, de nouvelles informations sont ajoutées. À cet effet, de nouvelles classes ont été définies qui contiennent, d'une part, les objets obtenus par la réflexion et, d'autre part, les informations qui seront éventuellement collectées lors d'étapes suivantes. Le

Tableau II décrit l'association logique entre les types retournés par la réflexion de .NET et les nouvelles classes définies dans le compilateur. Deux types associés décrivent le même type d'éléments, mais les objets du compilateur amène la description de plus d'information et d'une meilleure façon pour nos besoins.

Élément décrit	Type .NET	Obtenu par réflexion avec	Type du compilateur CIL
Type (classe)	Type	`GetTypes`	`ClassInfo`
Méthode	`MethodInfo` (sous-type de `MethodBase`)	`GetMethods`	`InternalMethodInfo`
Constructeur	`ConstructorInfo` (sous-type de `MethodBase`)	`GetConstructors`	`InternalConstructorInfo`
Champ	`FieldInfo`	`GetFields`	`FieldInfo`
Instruction CIL	n.d.	n.d.	`ILInstruction`

Tableau II : Association entre les types .NET et les classes du compilateur

ClassInfo

La classe `ClassInfo` est utilisée dans le compilateur pour représenter les types définis dans les modules du programme. Ceux-ci sont créés en spécifiant les méthodes, les constructeurs, les champs et le type de .NET auquel ils sont associés. De nombreuses autres informations sont ajoutées à ces objets dans les étapes ultérieures des flots du compilateur.

InternalStructure

La classe `InternalStructure` et ses deux sous-classes, `InternalMethodInfo` et `InternalConstructorInfo`, décrivent les méthodes ou les constructeurs d'un type par le compilateur.

Lors de la création d'un objet `ClassInfo`, des objets de type `InternalStructure` sont aussi créés afin de contenir les objets `MethodInfo` ou `ConstructorInfo` obtenus par la réflexion. Ces objets sont enrichis de données additionnelles au fur et à mesure des traitements effectués. Pour l'instant, seul l'objet de type `MethodInfo` ou de type `ConstructorInfo`, selon le cas, la liste des types des arguments et des variables locales de la méthode (obtenus par réflexion) et le code CIL (voir plus bas) sont spécifiés.

FieldInfo

Les objets de type `FieldInfo` sont les objets tels qu'obtenus par réflexion et décrivent les champs d'un type. Ils sont utilisés tels quels dans le compilateur car toutes les

informations nécessaires sont contenues dans ces objets et aucune information ne sera ajoutée à leur description.

Lecture du CIL

La lecture du CIL s'effectue lors de la création des objets de type `InternalStructure`. À l'aide d'un outil disponible sur l'Internet sous forme de fichier *.dll*, le *ilReader* [12], il est possible de lire le CIL des méthodes et des constructeurs. Il suffit de préciser l'objet `MethodInfo` ou `ConstructorInfo` dont on veut lire le code. Une liste des instructions CIL est retournée.

De la même façon que pour les méthodes et les constructeurs, les objets décrivant les instructions CIL telles qu'obtenues par le *ilReader* ne contiennent pas toutes les informations qui sont nécessaires dans le futur. Ces instructions sont donc contenues dans des objets de type `ILInstruction`, qui contiennent plus de données et donnent accès à plus de traitements à effectuer par rapport aux instructions. Un objet `ILInstruction` est défini par les données suivantes :

- L'instruction originale obtenue par *ilReader*.
- Son déplacement par rapport au début de la méthode.
- Son code d'opération (*opcode*).
- Son opérande, s'il y a lieu.
- Une étiquette, calculée à partir du déplacement.
- Dans le cas d'un branchement, l'instruction à laquelle le branchement est effectué.

La liste des instructions CIL, sous forme d'objets `ILInstruction`, est conservée dans l'objet `InternalStructure`.

Hiérarchie

Jusqu'à maintenant, tous les objets `ClassInfo` sont des descriptions indépendantes des types, sans égard aux relations d'héritage qui existent entre les objets. Il est possible d'obtenir la super-classe d'un type donné avec la réflexion, mais on obtient un objet `Type` et non un objet `ClassInfo`, comme il serait désirable. Il faut alors parcourir un à un les objets de la liste des `ClassInfo` et, pour chaque objet donné, parcourir la liste à nouveau afin de trouver le super-type en comparant le `Type` contenu dans le `ClassInfo` avec le

`Type` de la super-classe donné par la réflexion. Si il y a correspondance, un lien entre les deux `ClassInfo` est établi, où la sous-classe référencie la super-classe et la super-classe ajoute la sous-classe à une liste.

Une fois que la structure hiérarchique des types est établie, il est possible de définir quelles méthodes sont virtuelles et une représentation interne de la table d'appels virtuels de la classe est bâti. Une méthode est virtuelle si une autre version de cette méthode existe dans une super-classe ou dans une sous-classe. Avec la réflexion, il est possible d'obtenir les propriétés `IsVirtual` et `IsAbstract`, qui sont vraies si les méthodes sont définies comme `abstract` ou `virtual` dans le langage source. Cependant, une méthode définie comme `virtual` pourrait ne jamais être redéfinie dans une sous-classe (et ne nécessite donc pas une indirection par la table d'appels virtuels), alors que la propriété reste vraie. Deux méthodes de la classe `ClassInfo` permettent de détecter quelles méthodes sont bel et bien virtuelles : `IsOverriding` et `IsOverriden`, qui déterminent respectivement si une méthode redéfinie une méthode d'une super-classe et si une méthode est redéfini dans une sous-classe. À l'aide de ces deux méthodes, il y a moyen de trouver quelles sont les méthodes purement virtuelles (nécessitant une indirection par la table d'appels virtuels). Un ordre est alors défini parmi les méthodes de la classe qui représente la position relative des méthodes dans la table d'appels virtuels. Ces informations détectent aussi si une instruction CIL d'appel virtuel de méthode fait appel à une méthode purement virtuelle ou non. Il est donc possible d'optimiser certains appels, qui n'ont pas à utiliser l'indirection de la table d'appels virtuels.

Partitionnement

Durant cette étape, les attributs de partitionnement des méthodes sont pris en compte en invoquant la méthode `GetCustomAttributes` sur les objets de type `MethodInfo` ou `ConstructorInfo` obtenus par réflexion. Cette méthode retourne les attributs personnalisés définis sur cette structure. Dans notre cas, il s'agit des attributs de partitionnement, `PartitionAttribute.Software` ou `PartitionAttribute.Hardware`. Comme il est mentionné plus haut, si aucun attribut personnalisé n'est défini, on suppose que la structure est implémentée en logiciel.

À l'aide de ces attributs, les méthodes et constructeurs logiciels ou matériels sont stockés à l'intérieur de listes distinctes dans les objets de type `ClassInfo`. Ainsi, lors de la séparation des flots de compilation (logiciel et matériel), les traitements sont faits sur les structures respectives. De plus, chaque méthode implémentée en matériel se voit attribué un canal FSL par lequel la communication des données entre les registres et le module matériel est réalisée. Puisque huit canaux sont disponibles, un maximum de huit méthodes peut être implémenté matériellement.

Les résultats de cette étape sont une liste des types définis dans l'assemblage traité, les relations d'héritages entre ces types et les champs et méthodes qui les composent. De plus, les méthodes d'une classe sont divisées selon leur cible de compilation, soit le logiciel ou le matériel.

5.2.2 Graphe de flot de contrôle du CIL

Dans un compilateur, le graphe de flot de contrôle (GFC) décrit statiquement les chemins possibles du flot de contrôle lors de l'exécution d'une méthode. Cette structure est nécessaire pour les analyses de contrôle de flot, qui donnent lieu plusieurs optimisations [77].

Un GFC est typiquement composé des éléments suivants :

- Un ensemble de blocs de base (BB). Un BB est une série d'instructions qui est toujours exécutée séquentiellement, c'est-à-dire que le flot de contrôle entre toujours par la première instruction du bloc et sort toujours par la dernière.
- Un ensemble d'arcs dirigés. Les arcs relient les blocs entre eux, de telle sorte qu'un arc va d'un BB vers un autre pour indiquer que le flot de contrôle peut passer de l'un à l'autre. Un degré sortant d'un BB supérieur à 1 signifie que le contrôle peut passer à un bloc parmi plusieurs. Cela représente généralement des structures de contrôle comme `if-then-else` ou `switch`, ou des boucles `while` ou `for`. Un degré entrant supérieur à 1 démontre une jonction du flot de contrôle à partir de plusieurs blocs. Une telle structure des arcs représente en général le code à la suite d'une structure de contrôle.
- Un BB source, représentant l'entrée du flot de contrôle dans la méthode.
- Un BB arrivée, représentant la sortie du flot de contrôle dans la méthode.

Parmi les optimisations et analyses conventionnelles utilisant le GFC, mentionnons la détection du code mort, qui élimine le code d'un BB inatteignable, et la détection des boucles en recherchant les cycles dans le graphe.

Le graphe de flot de contrôle de chaque méthode est nécessaire dans notre compilateur puisque de nombreuses opérations utilisent les BB comme structure élémentaire. Dans le flot commun, le graphe de flot de contrôle est utilisé pour obtenir la représentation intermédiaire à partir de laquelle les flots logiciel et matériel effectueront leurs traitements. Cependant, il faut noter que des optimisations du flot de contrôle ne sont pas introduites par le compilateur de CIL, puisque nous supposons que celles-ci ont déjà été appliquées par le compilateur du langage source de haut niveau générant le CIL.

Décrivons maintenant les algorithmes et les structures utilisés pour la création du GFC et de sa représentation.

Création des blocs de base

La première étape pour la construction du GFC est la création des blocs de base. L'algorithme suivant, tiré de [31] est utilisé :

```
Création_blocs_de_base
Entrée : Séquence d'instructions CIL d'une méthode.
Sortie : Une liste des blocs de base.
Algorithme :
      1. Trouver les chefs (un chef est la première instruction
         d'un bloc de base)
            a. La première instruction de la méthode est un chef;
            b. Toute instruction étant la cible d'un branchement
               est un chef;
            c. Toute instruction suivant immédiatement un
               branchement est un chef.
      2. Un bloc de base inclut toutes les instructions entre deux
         chefs, y compris le premier chef mais non le deuxième.
```

Algorithme 1 : Création des blocs de base

Pour l'implémentation de cet algorithme, un parcours séquentiel des instructions est effectué afin de trouver les chefs, qui sont ajoutés à une liste. Un deuxième parcours de la liste des instructions est nécessaire pour créer la structure interne du BB à partir de la liste de toutes les instructions contenues entre deux chefs.

BasicBlock

Les BB sont représentés par des objets de type `BasicBlock` et plus particulièrement des objets de type `ILBasicBlock` (sous-classe de `BasicBlock`) pour les BB créés à partir du CIL. La classe `BasicBlock` hérite elle-même de la classe `Node`, qui est la structure de donnée de base représentant un nœud d'un graphe. Les objets de type `ILBasicBlock` contiennent les champs *IR* et `IRDependenceGraph`. Le premier champ contient la représentation intermédiaire du code de ce BB, alors que le second est utilisé plus tard pour ajouter des informations sur la représentation intermédiaire du bloc.

Construction du GFC

La construction du GFC est accomplie en reliant les blocs de base entre eux pour exprimer le flot de contrôle possible d'une méthode. Encore une fois, l'algorithme utilisé est inspiré de [31]. Pour chaque bloc de base, un parcours de la liste des blocs est effectué afin de déterminer les liens entre les blocs. Un arc est nécessaire entre deux blocs si :

1. Il y a un branchement conditionnel ou inconditionnel à partir de la dernière instruction d'un BB à la première instruction d'un autre;
2. Un BB suit un autre BB dans la séquence d'instructions CIL et la dernière instruction du BB précédent l'autre n'est pas un branchement inconditionnel.

Les liens entre les blocs de base sont conservés par des listes de prédécesseurs et de successeurs dans les objets de type `Node`.

Une fois que les BB sont créés et liés, une structure de données de type `ILFlowGraph` est créée à son tour afin de représenter le GFC de la méthode courante. L'objet de type `ILFlowGraph` est stocké dans l'objet de type `InternalStructure` représentant la méthode courante.

ILFlowGraph

Un objet `ILFlowGraph` contient principalement une référence à l'objet de type `InternalStructure` représentant la méthode, une liste des BB composant le graphe et des références aux premier et dernier blocs du graphe (la source et l'arrivée).

5.2.3 La représentation intermédiaire (RI)

Le CIL se veut une représentation intermédiaire du code de haut niveau. Il est d'un niveau relativement bas, ressemble à du code assembleur et est indépendant des architectures. Cependant, pour les traitements que notre compilateur doit effectuer, cette représentation intermédiaire n'est pas idéale :

- Le CIL utilise un modèle de machine à pile. Le code assembleur Microblaze généré pourrait utiliser cette structure à pile, mais le code généré serait loin d'être optimal. En effet, dans un tel modèle, les accès à la mémoire sont fréquents puisque les données utilisées « disparaissent » de la pile une fois consommées. Une représentation plus proche de l'architecture à registres du Microblaze doit être utilisée afin de produire du code de meilleure qualité.
- Le CIL est séquentiel. Un code séquentiel est moins proche d'un circuit logique dans lequel les opérations se déroulent en parallèle, si possible. Il faudrait donc utiliser une représentation qui expose mieux le parallélisme entre les instructions afin de faciliter la compilation dans le flot matériel et générer des circuits de meilleure qualité.

La RI est le résultat du flot commun et est utilisée par les flots logiciel et matériel.

Représentation de la RI

Les deux caractéristiques principales de la RI cherchent à combler les deux lacunes mentionnées plus haut.

La RI est représentée par une forêt d'arbres, dont les feuilles sont des nœuds représentant les données des instructions, dont les nœuds internes sont les instructions à effectuer sur ces données et dont les arcs représentent le flot des données entre les nœuds. Cette représentation orientée par le flot de données est plus proche du matériel et expose un parallélisme entre les instructions, alors que des nœuds de l'arbre provenant de branches différentes peuvent être exécutés parallèlement. Dans le reste du texte, nous utiliserons de façon interchangeable *nœud RI* ou *instruction RI* pour caractériser un nœud de l'arbre de la RI.

Les nœuds internes de la RI sont caractérisés par un registre dans lequel le résultat de l'opération est produit. La façon dont les registres sont alloués est exposée plus loin (section 5.3.3).

Les nœuds de la RI sont représentés par des objets de type `IRNode`. Les objets `IRNode` sont caractérisés principalement par le registre résultant, qui est le registre contenant le résultat de l'opération, et une liste d'opérandes de type `IRNode`, qui sont les opérandes utilisés par l'opération courante.

De nombreuses classes héritant de la classe `IRNode` sont définies pour représenter les différents types d'opérations et de nœuds de la RI, tel que décrit par la Figure 12. Les principaux nœuds de la RI sont illustrés sur la figure, mais il en existe d'autres. Intuitivement, les nœuds sont divisés en deux groupes : les nœuds produisant des valeurs (nœuds `Value`) et ceux n'en produisant pas (les autres). Les objets représentant les opérations produisant des valeurs sont caractérisés par un champ décrivant le type de la valeur produite. Les types possibles sont le sous-ensemble des types CIL supportés par notre compilateur. Des nœuds représentant des modifications au contrôle de flot des instructions sont divisés en deux types, soit les sauts conditionnels et les sauts inconditionnels.

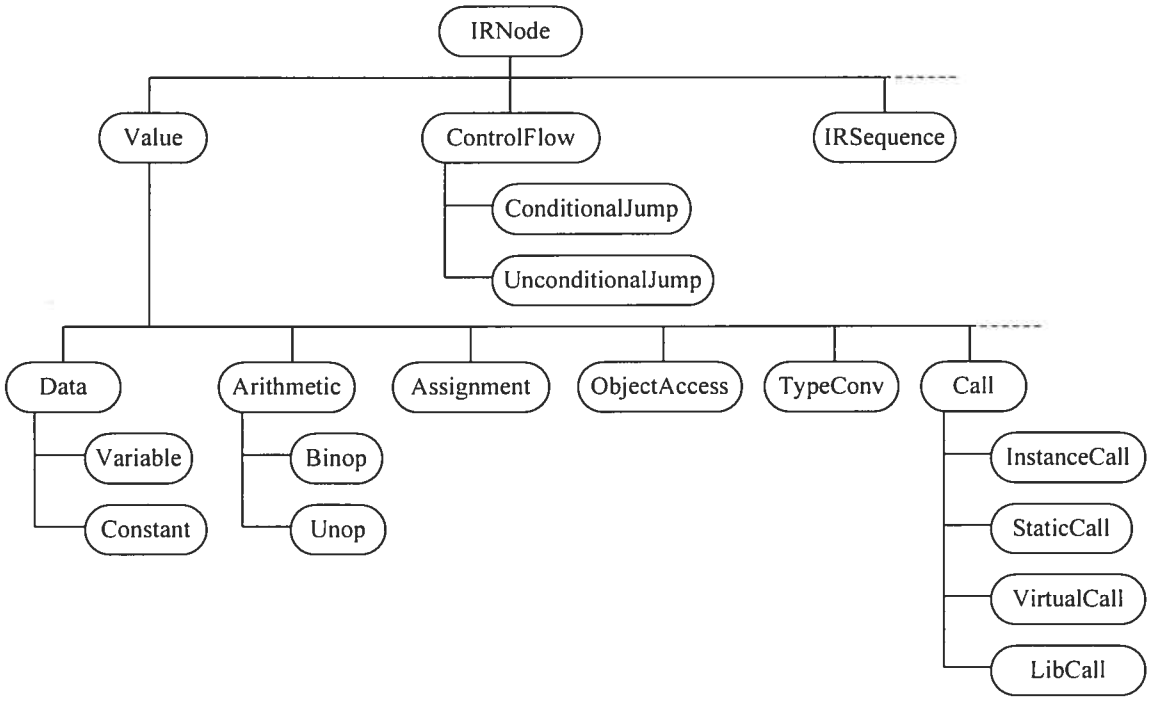


Figure 12 : Types de nœuds de la représentation intermédiaire

La création de la représentation interne est faite en utilisant les BB comme unité de traitement élémentaire, c'est-à-dire que chaque bloc de base est représenté par une forêt d'arbres et que les instructions RI d'un bloc ne sont pas reliées aux autres BB.

Algorithme

La transformation d'une représentation à pile vers une machine à registre est décrite dans [51]. Une technique similaire est utilisée. L'idée de l'algorithme est de simuler les effets des instructions du code CIL, en représentation de machine à pile, alors que les registres sont alloués du même coup.

Dans le compilateur, une pile est utilisée (que nous appelons pile interne) afin de simuler l'effet de l'exécution des instructions. Les instructions CIL des blocs de base sont analysées séquentiellement. Lorsqu'une instruction CIL dont la fonction est de charger un opérande sur la pile d'exécution est traitée, le compilateur empile un nœud de la RI sur la pile interne auquel on affecte un registre. Ce registre correspond au registre dans lequel le résultat de l'opération est stocké dans une architecture à registres. Le type du nœud empilé dépend du type d'opération effectuée par l'instruction CIL. Lorsqu'on retrouve une opération CIL dont la fonction est de réaliser une opération consommant des opérandes

sur la pile d'exécution, les nœuds correspondants sont dépilés de la pile interne. Un nœud RI correspondant à l'instruction CIL est créé, ayant comme opérandes les nœuds dépilés. Si l'opération produit un résultat sur la pile, alors le nœud créé est empilé sur la pile d'exécution et un registre résultant lui est assigné. Des instructions subséquentes qui utilisent comme opérande le résultat de cette instruction CIL auront alors comme opérande, dans la RI, le nœud RI produit. Certaines instructions CIL, tel le stockage des variables locales ou des arguments, consomment des éléments de la pile, mais ne produisent aucun résultat. Dans ce cas, ce nœud devient la racine de l'arbre de la représentation intermédiaire d'une partie du BB. Les différents arbres créés par un même bloc de base sont stockés ensemble dans un nœud spécial de la RI qui contient la séquence des arbres.

Un exemple est donné dans la Figure 13. Dans cet exemple, les instructions CIL à convertir en RI sont données à la gauche de la figure, où l'instruction du haut est la première. Ce bout de code réalise un calcul du type $a = x - (y + z)$, où x, y et z sont des arguments et a une variable locale d'une méthode. Le code CIL correspondant charge sur la pile les trois arguments, additionne les deuxième et troisième arguments, soustrait ce résultat intermédiaire au premier argument de la méthode et, enfin, affecte le résultat du calcul à la variable locale, celle ayant l'index zéro en l'occurrence. Au centre de la figure, les transitions de la pile d'exécution sont illustrées selon les instructions qui sont traitées. Finalement, à droite, on retrouve l'arbre de la représentation intermédiaire qui est créé et les nœuds correspondant aux opérations réalisées. Des flèches allant des transitions de la pile vers les nœuds de la RI montrent où se retrouvent les éléments de la pile dans l'arbre.

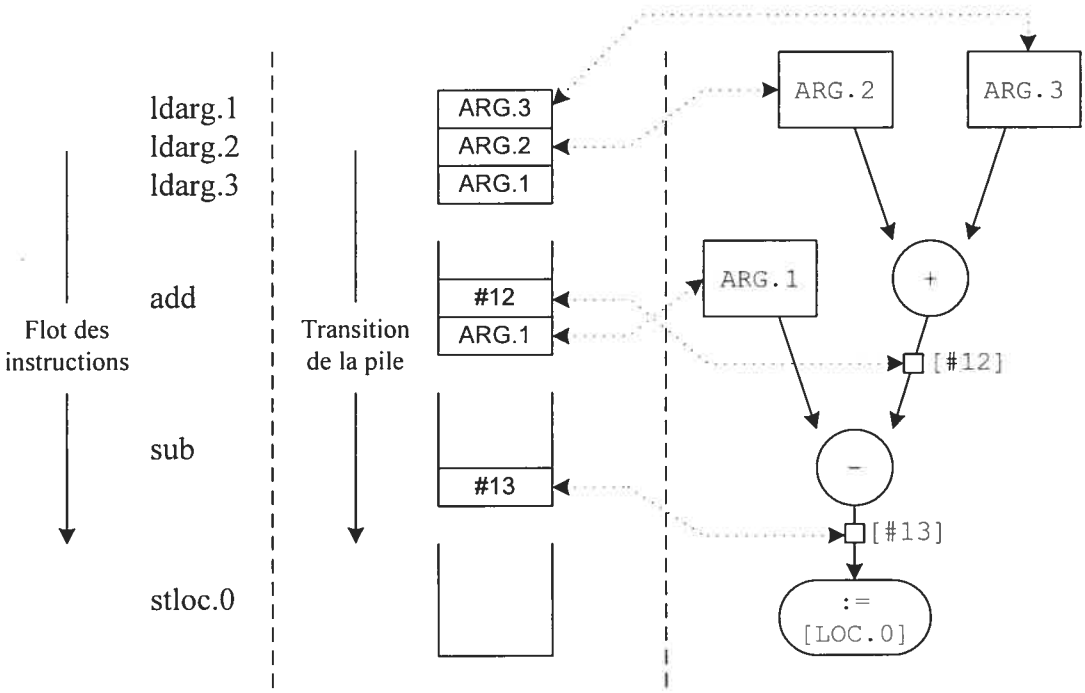


Figure 13 : Exemple transformation CIL vers RI

Il est important de mentionner que puisque les opérandes d'un nœud de la RI sont conservées dans ce nœud selon l'ordre dans lequel ils sont produits sur la pile, il est possible de reconstruire une représentation séquentielle des instructions RI qui correspond à l'ordre des instructions CIL d'un BB. Il suffit de faire une traversée postfixée en profondeur des arbres de la RI du BB.

Registres symboliques

Il n'a toujours pas été mentionné comment les registres résultants instructions RI (i.e. le registre dans lequel le résultat d'un nœud RI est stocké) sont affectés. Généralement, dans les compilateurs, l'allocation des registres est une optimisation importante et bien connue [77].

Dans le cas présent, afin de garder une indépendance face à l'architecture visée, les registres alloués ne sont pas des registres réels d'une architecture donnée (le Microblaze dans notre cas). Des *registres symboliques* sont utilisés. On suppose un ensemble infini de registres symboliques, permettant de faire abstraction de l'architecture réelle visée. Un nouveau registre symbolique est créé pour chaque instruction RI générée (ou presque; voir

plus bas) nécessitant un registre pour accueillir le résultat de l'opération effectuée. L'association des registres symboliques à des registres physiques est accomplie plus tard par le compilateur, lorsque l'abstraction de l'architecture cible n'est plus possible.

Pour certains nœuds RI, il n'y a pas de nouveau registre symbolique créé à chaque fois. Ces nœuds sont ceux qui sont générés par les instructions CIL effectuant des opérations de chargement et de stockage des variables locales et des arguments de la méthode. Un registre symbolique prédéfini est utilisé pour chaque variable locale et argument. Les instructions RI concernées utilisent toujours le registre symbolique prédéfini correspondant. Ce mécanisme vise à optimiser le code éventuellement généré par le compilateur en réduisant le nombre d'accès mémoire, puisque ces deux types de variables sont vivants pour toute l'exécution de la méthode.

Une solution alternative possible consiste à traduire naïvement les instructions CIL. Selon le modèle de machine à pile des instructions CIL, les instructions de chargement et de stockage des variables locales et des arguments effectuent un transfert de la mémoire à la pile, ou vice-versa. À chaque utilisation d'une variable locale ou d'un argument, un tel transfert est effectué. S'il y avait traduction directe entre le CIL et les instructions RI, le résultat contenu dans le registre symbolique résultant associé à une telle instruction CIL serait produit ou stocké en mémoire. Évidemment, cette solution est loin d'être optimale. Une autre solution possible est de ne stocker les variables concernées en mémoire qu'à la fin du BB. Ainsi, lors de la première utilisation de la variable dans un BB suivant, celle-ci se trouve automatiquement en mémoire, alors qu'elle est dans un registre symbolique dans les utilisations subséquentes. Cependant, des accès mémoire inutiles sont encore une fois introduits.

La solution utilisée associe un même registre symbolique à une variable locale ou à un argument donné, qui est utilisé uniformément par tous les accès à cette variable.

Puisqu'un registre symbolique est associé aux variables ayant une utilité durant toute la méthode, on constate qu'un registre symbolique est créé uniquement en vue d'accueillir des variables temporaires, qui sont les résultats intermédiaires des calculs effectués sur la pile.

Résolution de la pile

Il est utile de s'interroger sur ce qui arrive lorsque la pile d'exécution ne possède pas le même contenu à la fin de différents BB ayant le même successeur, c'est-à-dire où il y a jonction du contrôle de flot. En effet, l'algorithme de génération de la RI, tel que décrit ici, n'est pas en mesure de faire face à ce problème. Cependant, la spécification du CLI prévoit certaines mesures qui évitent de faire face à ce problème. Cela a pour effet de simplifier la génération de code natif pour les compilateurs, en obligeant le CIL à se conformer à certaines règles [29]. Ces règles concernent l'*état de la pile*, qui est le nombre et le type des éléments de la pile à un moment donné. L'algorithme de vérification du CLI permet de vérifier si le CIL d'une méthode respecte bien les règles. Particulièrement :

- L'état de la pile à tout point du programme doit être identique pour tous les flots de contrôle. Par exemple, une boucle exécutée un nombre de fois inconnu et poussant un nouvel élément sur la pile à chaque itération est interdite.
- Il doit être possible, avec une passe sur un flot d'instructions CIL d'une méthode, de déterminer l'état de la pile pour chaque instruction. Par exemple, si la passe d'analyse arrive à une instruction CIL x qui suit immédiatement une instruction de branchement inconditionnel et que x n'est pas la cible d'une instruction de branchement précédente, il est impossible de déterminer l'état de la pile à x. La spécification requiert une pile vide dans ce cas. Une instruction y qui suit x ne pourrait donc pas être un branchement à x, où la pile n'est pas vide à la suite de l'exécution y.

Grâce aux règles du CLI, le problème potentiel mentionné ne survient pas. Si par contre ce problème devait être traité, il faudrait implémenter un algorithme de cohérence de la pile d'exécution qui s'occupe de réorganiser le contenu de la pile lors de la jonction du flot de contrôle.

Type des nœuds RI

Pour les nœuds qui produisent une valeur, un type de donnée leur est assigné. Ce type est un type particulier supporté par le CLI et représente le type de la valeur produite. Pour déterminer le type d'un nœud, une méthode qui retourne le type de la valeur produite est utilisée, qui considère les opérandes et l'opération du nœud. La méthode implémente les

tables de référence du CLI qui donnent les conversions de valeurs et le type résultant d'une opération donnée.

5.3 Flot logiciel

Le flot logiciel est responsable de la génération du code assembleur Microblaze à partir duquel est généré l'exécutable final, utilisé par le processeur. Ce flot est la première moitié de la partie dorsale du compilateur. Il utilise en entrée les objets de type `ClassInfo` et la RI créés dans le flot commun. Seules les méthodes annotées en logiciel sont considérées par le flot logiciel. La sortie est un fichier texte contenant le code et les données de la partie logicielle du programme source. Ce fichier est assemblé et lié par le compilateur MB-GCC de Xilinx.

Dans cette section, nous décrivons les analyses et les traitements réalisés par le flot logiciel ainsi que les structures de données utilisées. Les phases sont décrites dans l'ordre chronologique dans lequel elles sont exécutées par le compilateur. La Figure 14 montre l'enchaînement des phases du flot logiciel.

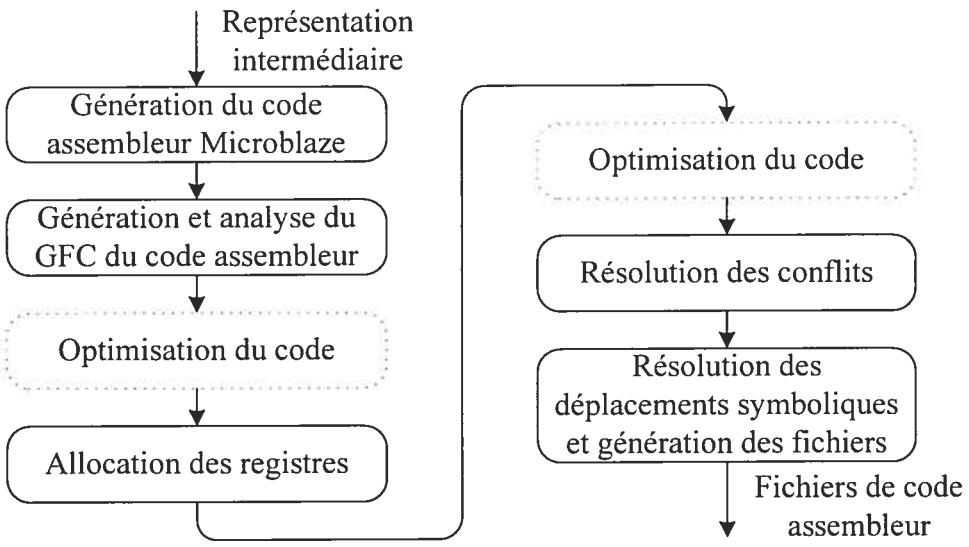


Figure 14 : Phases du flot logiciel

Il faut noter que, dans le flot logiciel, aucune optimisation n'est appliquée à partir de la RI. On suppose que la majorité des optimisations de haut niveau ont déjà été effectuées

par le compilateur du langage source. Donc, les optimisations devront être faites principalement sur un langage de plus bas niveau, tel le code assembleur qui sera généré par le flot logiciel.

5.3.1 Génération de code assembleur

La première phase du flot commun consiste à générer le code assembleur Microblaze. Le code généré est manipulé et modifié par la majorité des phases subséquentes du flot logiciel.

Algorithme

La génération de code se déroule à partir de la représentation intermédiaire de chaque BB d'une méthode. Les blocs sont parcourus un à un et le code généré est stocké dans une liste d'instructions conservée dans le bloc de base correspondant.

L'Algorithme 1 est utilisé. Celui-ci parcourt la structure arborescente de la RI et génère le code au passage. Voici le pseudo-code de l'algorithme utilisé :

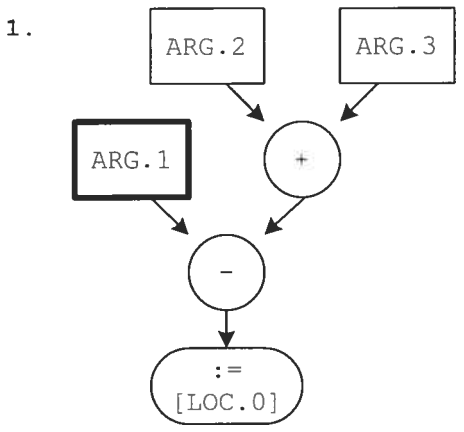
```
Génération_code_assembleur
Entrée : RI d'un bloc de base
Sortie : Liste d'instructions assembleur Microblaze
Algorithme :
      1. Soit L une liste vide qui contient le code
      2. Pour chaque racine RI_racine dans la séquence d'arbres de la
         RI
            a. Soit RI_courant le nœud courant, où RI_courant = RI_racine
            b. Pour chaque opérande RI_operande de RI_courant
                  i. Générer le code de RI_operande et l'ajouter dans L
            c. Générer le code de RI_courant et l'ajouter dans L
```

Algorithme 2 : Génération de code assembleur à partir de la représentation intermédiaire

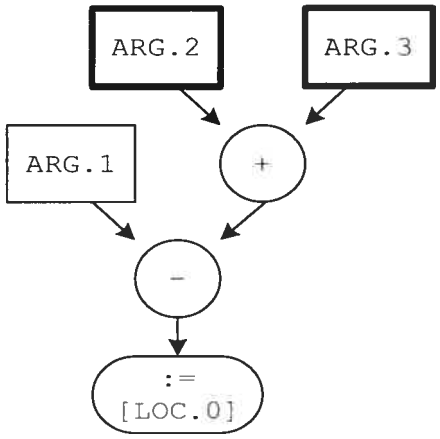
L'algorithme génère récursivement le code pour les opérandes d'un nœud RI et ensuite pour le nœud lui-même. Tel que mentionné dans la description de la RI, le parcours postfixé en profondeur des arbres de la RI conserve l'ordre des instructions par rapport au CIL d'origine. Ainsi, le code d'un nœud RI correspondant à une instruction CIL donnée est toujours généré avant le code pour n'importe quelle instruction CIL subséquente dans l'ordre des instructions CIL de la méthode.

Dans l'algorithme, il n'est pas mentionné comment le code est généré pour un nœud donné. Une correspondance est établie entre une instruction de la RI et une séquence d'instructions assembleur. Les registres sources et cibles de la séquence de code assembleur sont donnés respectivement par les registres résultants des opérandes et le registre résultant du nœud RI.

Un exemple de production de code à partir d'un arbre de la RI d'un BB (on utilise le même arbre que celui produit dans l'explication de la production de la RI à la Figure 13) est donné à la Figure 15. Le code est toujours généré pour les opérandes d'un nœud avant le nœud lui-même. Donc, le premier nœud pour lequel le code est produit est l'argument 1, à gauche dans l'arbre. Toutefois, aucun code n'est produit pour ce nœud, puisqu'une donnée comme un argument correspond directement avec un registre. La même chose se produit à l'étape 2, alors que les arguments 2 et 3 ne produisent pas de code. Ensuite, le code est généré pour l'opération d'addition, utilisant comme opérandes les arguments 2 et 3 (qui correspondent à des registres). À l'étape 4, le code assembleur est généré pour l'opération de soustraction, en utilisant l'argument 1 et le résultat de l'opération d'addition (contenu dans le registre symbolique #12) comme opérandes. Finalement, le code de l'affectation est produit, alors que le résultat de la soustraction est transféré dans le registre correspondant à la variable locale 0, qui est le registre résultant de ce noeud.

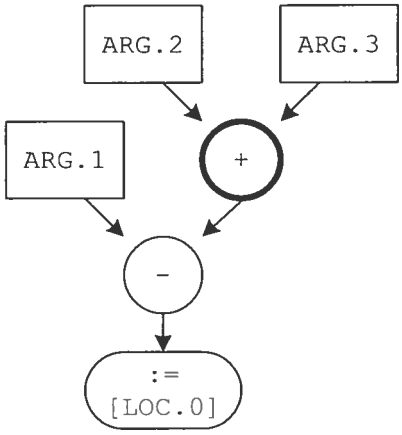


2.



Liste d'instructions

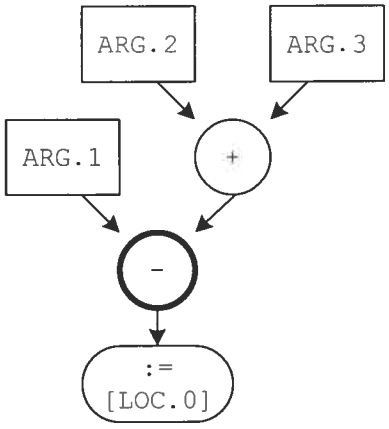
3.



Liste d'instructions

```
add #r12, $arg.2, $arg.3
```

4.



Liste d'instructions

```
add #r12, $arg.2, $arg.3
sub #r13, $arg.1, #r12
```

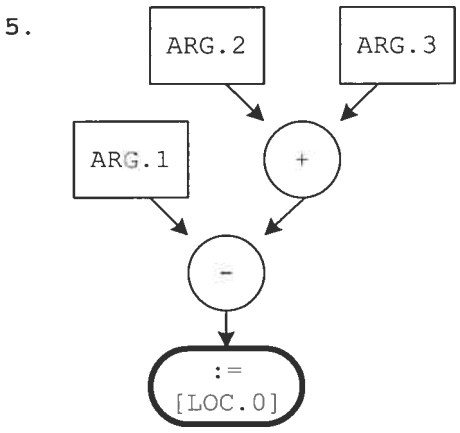


Figure 15 : Génération de code à partir de la représentation intermédiaire

Plusieurs particularités du code assembleur à générer doivent être mentionnées.

- Un nœud RI correspond généralement à plus d'une instruction en code assembleur. Dans certains cas, comme la création d'un tableau par exemple, la génération de code peut introduire de nouvelles instructions de branchement. Dans ce cas, le GFC créé à partir du CIL n'est plus représentatif du flot de contrôle du programme. Le GFC doit par contre être réutilisé dans d'autres analyses. Il faudra donc modifier le GFC pour tenir compte des branchements.
- Toutes les instructions qui sont des cibles potentielles de branchements sont identifiées par des étiquettes. Principalement, le point d'entrée d'un bloc de base est identifié par une étiquette correspondant à la première instruction du bloc. Les instructions de branchement utilisent comme cible l'étiquette de l'instruction ciblée. Ces étiquettes sont transformées en déplacement réel par l'assembleur de MB-GCC.
- Le point d'entrée d'une méthode est identifié de façon unique par une étiquette qui est générée à partir de la signature de la méthode. Un appel non virtuel à une méthode est un branchement à cette étiquette.
- Les appels virtuels des méthodes sont résolus de façon classique à l'aide des tables d'appels virtuels (*vtable*). L'adresse d'une méthode virtuelle est stockée à un déplacement donné dans la table d'appels virtuels d'une classe. L'appel d'une

telle méthode consiste à rechercher l'adresse de la méthode correspondante dans la table, puis à effectuer un branchement à cette adresse.

- L'appel d'une méthode implémentée en matériel génère des instructions qui transfèrent les arguments de la méthode par le canal FSL assigné à cette méthode, ainsi que l'instruction qui récupère la donnée de retour. Une seule instruction est nécessaire pour le transfert d'une donnée de 32 bits du canal vers le module matériel ou vice-versa. Un tel appel de méthode génère au minimum $n + 1$ instructions, où n est le nombre d'arguments de la méthode et l'instruction supplémentaire est la récupération de la valeur de retour.
- À l'étape de génération de code assembleur, les registres utilisés par les instructions sont les registres symboliques. Ces registres doivent être associés à des registres physiques durant une phase d'allocation des registres. Puisque les registres physiques utilisés dans la méthode ne sont pas connus, la sauvegarde des registres à l'entrée de la méthode et avant les appels de fonction n'est pas réalisée à ce moment, mais après l'allocation des registres. Une sauvegarde des registres symboliques aurait pu s'effectuer en utilisant les registres symboliques, mais des instructions redondantes seraient générées lorsque deux registres symboliques se voient attribuer le même registre physique.
- Toutes les références à des déplacements par rapport au bloc d'activation de la méthode courante (voir plus loin) sont des *déplacements symboliques*, c'est-à-dire que le déplacement réel n'est pas encore connu. Une explication détaillée est donnée plus loin.

La liste des instructions de code assembleur généré est stockée dans l'objet représentant le BB. Le code généré de tous les BB d'une méthode est « recollé » dans une liste pour obtenir une nouvelle version séquentielle complète du code. Cette liste sera utilisée afin de construire un nouveau GFC correspondant au code assembleur Microblaze, en utilisant les mêmes algorithmes que décrits précédemment pour la construction du GFC à partir du CIL. Le GFC généré est représenté par un objet de type `MBFlowGraph` et réalise les mêmes fonctionnalités que le GFC du code CIL. Des BB de type `MBBasicBlock`, une sous-classe de `BasicBlock`, composent ce graphe et sont analogues aux BB du GFC du

CIL. Le graphe contenant le code assembleur est le résultat principal de la génération de code.

Il est évident que cette façon de faire pour reconstruire le GFC n'est pas optimale, mais elle est simple. Il aurait été possible d'analyser le code assembleur Microblaze généré à partir des BB du CIL, de diviser les BB dans les cas où cela est nécessaire et d'établir de nouveaux liens entre les blocs. De cette façon, aucun traitement ne serait effectué pour les BB où cela s'avère inutile.

Représentation des instructions assembleur Microblaze

Les instructions assembleur Microblaze (nous utiliserons uniquement *instruction Microblaze*) sont représentées par les objets de type `MBInstruction`. Ces objets contiennent (entre autres) des champs pour le stockage des informations suivantes :

- L'étiquette de l'instruction.
- Le code opérationnel de l'instruction.
- Le registre source et les deux registres opérandes.
- Un commentaire, qui aide à la compréhension du code généré.
- Une référence à la méthode d'où l'instruction provient.
- Un numéro, permettant d'ordonner les instructions pour différents besoins.

Des méthodes de la classe permettent au compilateur d'obtenir toutes sortes d'informations à propos de l'instruction, telles de quel type d'instruction il s'agit (opération arithmétique, branchement, etc.), obtenir la cible d'un branchement, ou savoir si l'instruction utilise ou redéfinit la valeur contenue dans un registre donné.

Stratégies de stockage

Comme dans les compilateurs traditionnels, la mémoire est divisée en deux parties : la pile et le tas. La pile sert au stockage des données relatives aux fonctions ou méthodes, alors que le tas est utilisé pour la création dynamique de données. La Figure 16 montre la technique traditionnelle de la division de la mémoire. La plage mémoire disponible pour la pile débute typiquement à l'adresse la plus élevée et l'allocation se fait vers les adresses plus petites; le tas débute à l'adresse la plus petite et l'allocation se fait vers les adresses plus grandes (ce qui n'est toutefois pas totalement vrai, car certaines structures allouées

dynamiquement ne sont plus *vivantes* à certain moment et la mémoire qu'elles utilisent est réclamée).

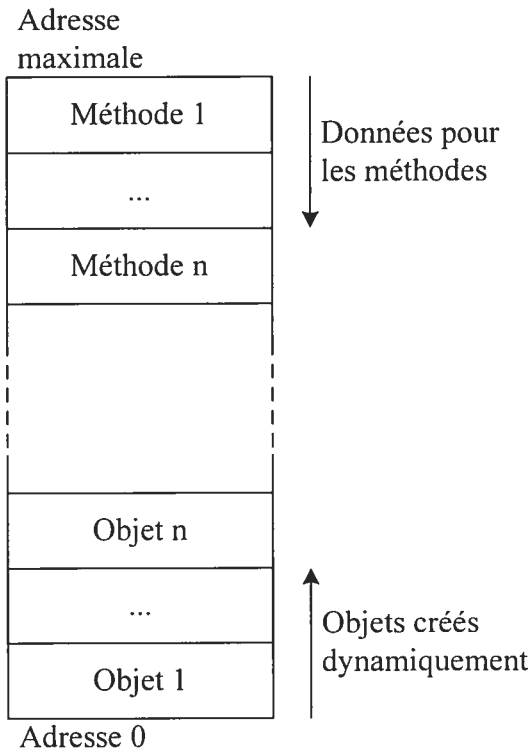


Figure 16 : Utilisation de la mémoire pour la pile et le tas

Notre compilateur utilise la mémoire de cette façon, même si la gestion du tas est quelque peu simplifiée.

Support d'exécution – blocs d'activation

Un bloc d'activation est utilisé pour stocker en mémoire les informations relatives à une méthode [31]. Un bloc est créé séquentiellement sur la pile en mémoire à la suite d'un appel, afin supporter les appels imbriqués et récursifs des méthodes. Un pointeur de pile indique l'adresse du dernier bloc de mémoire de la pile (donc la plus petite adresse).

Dans notre compilateur, les blocs d'activation sont utilisés pour contenir les arguments de la méthode, les variables locales, l'adresse de retour de la méthode appelante et les différentes données sauvegardées, comme les registres sauvegardés lors d'appels de

méthodes ou les registres déversés (*spilled*) en mémoire lors de l'allocation de registres. La Figure 17 montre la structure d'un bloc d'activation en mémoire.

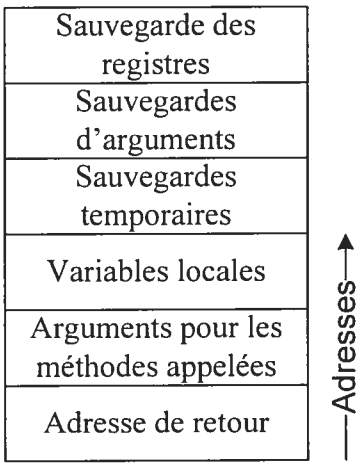


Figure 17 : Bloc d'activation

Les blocs d'activation respectent la convention d'appel de C qu'utilise le compilateur MB-GCC [28]. De cette façon, le code généré par le compilateur est capable d'interagir avec les fonctions C offertes au Microblaze.

Comme il a été mentionné précédemment, toutes les instructions utilisant la mémoire d'un bloc d'activation utilisent des déplacements symboliques par rapport au début du bloc. Ces déplacements sont remplacés en déplacements réels dans les phases finales du flot logiciel. L'utilisation des déplacements symboliques est nécessaire puisque la taille exacte du bloc d'activation n'est pas encore connue. En effet, à cet instant, il est impossible de déterminer le nombre de certains types d'éléments contenus dans le bloc (par type d'élément, nous entendons ici des variables locales, des registres sauvegardés, etc. et pas un type de donnée). Par exemple, comme il est mentionné plus haut, les registres sauvegardés ne sont pas connus.

Les déplacements symboliques sont identifiés de façon unique par des noms particuliers et un indice représentant un élément donné dans le groupe. Dans certains cas, la fonctionnalité réalisée par un type d'éléments est très proche d'un autre type. Certain types d'éléments dans le bloc d'activation sont différenciés seulement pour simplifier des traitements. Les types sont répertoriés dans le Tableau III. L'indice d'un élément indique

un élément particulier. Par exemple, la première variable locale de la méthode sera définie par `%loc.1` et le premier argument par `%arg.1`.

Type d'élément	Déplacement symbolique (où *x* est l'indice de l'élément)	Utilisé pour
Variable locale	`%loc.x`	Variable locale de la méthode
Argument	`%arg.x`	Argument d'une méthode appelée
Sauvegarde temporaire	`%tempsave.x`	Déversement en mémoire d'un registre
Sauvegarde de registre	`%save.x`	Sauvegarde de registre avant appel de méthode
Sauvegarde d'argument	`%argsave.x`	Sauvegarde d'argument avant appel de méthode

Tableau III : Types de déplacements symboliques

Structure des objets en mémoire

Les instances des classes, les objets, sont stockés sur le tas. Ils doivent contenir les valeurs associées aux champs de l'instance donnée et une identification permettant de déterminer la classe dont ils sont issus.

Un objet en mémoire est composé de la façon suivante, comme il est montré sur la Figure 18 (où la structure d'un objet est donnée à gauche et la structure d'un tableau à droite) :

- Un en-tête de deux mots mémoire. Le premier mot contient une référence à la table d'appels virtuels de la classe de l'objet. La table d'appels virtuels est contenue dans la section des données du programme et est structurée ainsi : la première entrée est une référence à la table d'appels virtuels de la super-classe et les entrées suivantes sont les adresses des méthodes virtuelles de la classe. Un test de type dynamique sur un objet utilise l'adresse de la table d'appels virtuels. Le deuxième mot de l'en-tête de l'objet est inutilisé, sauf pour les tableaux, où le nombre d'éléments qu'il contient y est stocké. Pour implémenter entièrement les fonctionnalités de l'environnement d'exécution (comme le ramasse-miettes), l'en-tête de l'objet devra être modifié. Le changement de la structure de l'en-tête pourrait être effectué aisément dans les sources du compilateur, si nécessaire.

- Les champs de l'objet ou les données du tableau. Les premiers champs sont les champs de la super-classe la plus haute dans la hiérarchie, alors que les derniers sont ceux de la classe de l'objet courant.

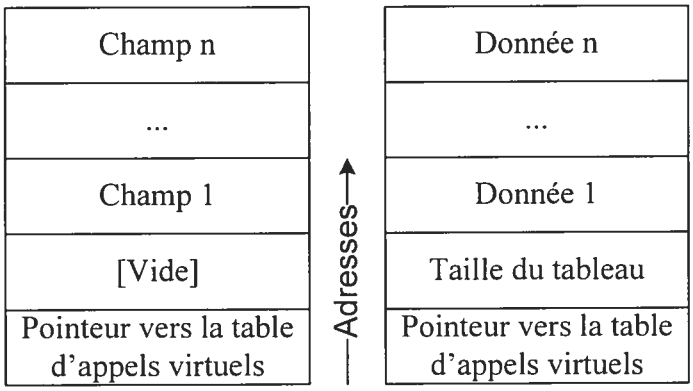


Figure 18 : Structure des objets et des tableaux en mémoire

Support d'exécution – Gestion dynamique de la mémoire

Certaines instructions créent dynamiquement des objets en mémoire, sur le tas, comme un tableau ou une instance d'une classe. Selon le langage source utilisé, la réclamation des blocs de mémoires utilisés par un objet qui n'est plus vivant est effectuée de façon explicite ou implicite. Dans les langages C et C++, la réclamation est explicite. Dans un tel cas, le programmeur a la responsabilité de réclamer manuellement la mémoire avec le mot clé `free`, par exemple. Les langages Java et C# utilisent une réclamation implicite, c'est-à-dire que le programmeur ne peut réclamer manuellement des blocs de mémoire. La récupération est accomplie par un ramasse-miettes qui parcourt la mémoire et réclame les blocs utilisés par des objets morts. Généralement, les objets encore vivants sont ensuite déplacés au début du tas afin de permettre de nouvelles allocations de mémoire. L'activation du ramasse-miettes est lancée dans différentes circonstances, comme lorsqu'il n'y a plus de mémoire disponible.

Dans l'environnement d'exécution du CLI, la gestion dynamique de la mémoire est automatique (avec un ramasse-miettes) et n'est pas explicitée par des instructions dans le CIL. Dans notre compilateur, cette gestion est fortement simplifiée. Comme pour la pile, un pointeur indique le début de la zone de mémoire du tas. L'allocation dynamique d'un objet est effectuée à partir de l'adresse indiquée par le pointeur. Celui-ci est incrémenté

d'une valeur correspondant à la taille de l'objet. La réclamation des blocs de mémoire n'a jamais cours. En conséqunce, le tas ne cesse de croître à la suite de la création dynamique d'objets, jusqu'à potentiellement rejoindre la pile. Des tests sont effectués lors de l'allocation dynamique d'objets et lors d'appels de méthodes afin de vérifier que la pile n'écrase pas le tas ou vice-versa. Si un tel cas survient, l'exécution du programme est terminée.

Deux raisons sont à la base de ce choix d'implémentation :

- Les services de gestion de mémoire sont fourni par l'environnement d'exécution du CIL, le CLI, et pas par le langage lui-même. L'implémentation de la sémantique du langage ne nécessite pas de ramasse-miettes.
- Les routines C de gestion de mémoire du Microblaze sont incomplètes. L'allocation de mémoire par `malloc` est effecutée à l'aide de la même stratégie que nous utilisons, alors que la routine de réclamation, *free*, ne fait *absolument rien*! En effet, cette routine n'est tout simplement pas implémentée. De nombreuses explications justifient probablement cela, mais mentionnons principalement le peu d'allocations dynamiques dans les applications typiques des systèmes embarqués.

Bien sûr, rien n'empêche la possibilité d'éventuellement ajouter le support nécessaire pour la gestion automatique de la mémoire. De nombreux algorithmes sont connus [66]; certains sont même spécialisés pour les systèmes embarqués [36]. Pour l'instant, certains programmes ne peuvent être exécutés (lorsqu'il y a de nombreux objets créés dynamiquement), mais ils ne sont généralement pas représentatifs d'applications typiques pour ce genre de système.

Support d'exécution – Fonctions C

Puisque le compilateur respecte la convention d'appel de C et l'interface binaire d'application (*Application Binary Interface* ou ABI), le code généré est apte à interagir avec les fonctions C de l'environnement d'exécution du Microblaze.

Ainsi, certaines méthodes des classes prédéfinies de .NET peuvent être associées à des fonctions C prédéfinies. Par exemple, la méthode d'impression de données sur la console, `Console.Write` ou `Console.WriteLine`, est associée aux fonctions C `print` ou `putnum` du Microblaze, qui sont des versions réduites de `printf`, respectivement pour

l'impression d'une chaîne de caractères et pour celle d'un nombre. Ces méthodes sont pour l'instant les seules qui sont implémentées par les fonctions C, puisqu'elles sont utiles pour le déverminage des applications créées.

Toutefois, certains problèmes surviennent : la représentation d'une donnée sous .NET et en C n'est pas nécessairement la même. Par exemple, une chaîne de caractères sous .NET est considérée comme un objet alors que ce n'est qu'un simple tableau en C. Ainsi, la fonction `print` de C ne pourra accepter directement l'objet qui représente la chaîne de caractères, puisqu'un objet contient un en-tête avant les cases du tableau de caractères. Dans cet exemple, afin de pouvoir utiliser les chaînes de caractères de .NET avec les fonctions C, il faudrait peut-être modifier la représentation de ces données ou exécuter un pré-traitement sur ces données afin d'extraire la partie représentant seulement le tableau de caractères.

Donc, l'association des méthodes de .NET avec des fonctions prédéfinies de C offre de plus grandes fonctionnalités, mais dans certains cas, des problèmes comme ceux énumérés apparaissent et il faut trouver des moyens de les contourner ou de les résoudre.

5.3.2 Optimisation du code assembleur

Une fois le code assembleur généré et après l'allocation des registres, des phases d'optimisations peuvent être réalisées afin d'augmenter la qualité du code [77]. Des optimisations du code à l'intérieur des blocs de base sont possibles, dont, principalement, l'ordonnancement des instructions. Des optimisations entre les blocs de base peuvent aussi être réalisées, comme le déroulement de boucle.

Cependant, notre compilateur n'effectue pas de telles optimisations. Le but du compilateur est d'obtenir en premier lieu du code fonctionnel. En conséquence, la priorité est mise sur la production du code, et non sur son optimisation. Toutefois, rien n'empêche que des phases d'optimisations soient ajoutées dans la structure du compilateur.

5.3.3 Allocation des registres

Comme il a été mentionné auparavant, les registres utilisés par les instructions Microblaze sont des registres symboliques. La phase d'allocation des registres associe aux registres symboliques des registres physiques de l'architecture visée, c'est-à-dire le processeur Microblaze dans notre cas. De nombreux algorithmes d'allocation des registres

sont connus. La performance de l'algorithme employé joue souvent un rôle important dans la qualité du code produit. Différentes méthodes sont utilisées : les méthodes locales aux BB produisent du code de moins bonne qualité, alors que les méthodes locales à une méthode (intra procédurale) donnent généralement des résultats bien supérieurs. Des techniques permettent aussi une allocation inter procédurale. Un algorithme d'allocation performant produit du code où les données qui servent le plus souvent sont conservées autant que possible dans les registres, ce qui a pour effet de réduire les accès à la mémoire. Le temps d'exécution de l'algorithme peut aussi être un facteur important. L'algorithme le plus utilisé dans les compilateurs statiques, de type intra procédurale, est l'allocation par coloriage de graphe [43].

Dans notre cas, l'algorithme choisi est une version modifiée du balayage linéaire (*linear-scan*) [79] avec deuxième chance (second-chance binpacking) [88]. C'est une technique locale aux méthodes (intra procédurale), typiquement utilisée dans les compilateurs dynamiques vu la rapidité d'exécution de l'algorithme.

La justification de ce choix d'algorithme, par rapport au coloriage de graphe est claire : l'implémentation est simple, et un résultat de bonne qualité est obtenu. L'implémentation rapide de cet algorithme a permis d'obtenir un prototype fonctionnel plus rapidement. Cependant, il est évident que l'utilisation de l'algorithme par coloriage de graphe serait importante pour obtenir du code de meilleure qualité.

Dans les sections suivantes, les concepts, l'algorithme et les modifications de l'allocation de registres par balayage linéaire sont expliqués.

Intervalles de vie

Les décisions d'allocation des registres sont prises à partir des intervalles de vie. Un intervalle de vie est défini par la première instruction à partir de laquelle une variable est utilisée, jusqu'à la dernière utilisation. Dans le cas présent, les variables sont les registres symboliques auxquels les registres physiques seront alloués. Les notions de première et de dernière utilisation supposent un ordre par rapport aux instructions. Cet ordre est calculé à partir de différentes traversées du GFC, comme en profondeur d'abord, mais aussi selon l'ordre textuel du code par rapport au programme original. L'approximation faite par les intervalles, qui est la principale cause des moins bonnes performances du code généré,

consiste à ne pas tenir compte des trous dans les intervalles de vie d'une variable. En effet, entre deux utilisations de celle-ci, la variable pourrait ne plus être vivante.

Algorithme

L'algorithme de balayage linéaire est rapide puisque les décisions d'allocation sont faites en une seule passe sur le code. À chaque instruction dans la représentation linéaire du GFC, les intervalles de vie *actifs*, c'est-à-dire ceux dont la fin est plus loin que l'instruction courante, sont considérés. S'il y un nombre plus petit d'intervalles actifs que de registres disponibles, alors tous les intervalles peuvent résider dans les registres. Par contre, si à un point donné, il y a plus d'intervalles que de registres, un ou plusieurs intervalles devront être débordés en mémoire, c'est-à-dire qu'un emplacement en mémoire doit leur être assigné, d'où ils seront utilisés. Les décisions établissant quels intervalles doivent être débordés en mémoire sont basées sur différentes heuristiques, discutées en profondeur dans [87].

Implémentation

L'implémentation de l'algorithme d'allocation des registres est inspirée d'une version plus performante du balayage linéaire, le balayage linéaire avec deuxième chance [88]. Dans cet algorithme, un intervalle de vie peut se voir attribuer un registre différent dans lequel résider lorsqu'il est utilisé après avoir été déversé en mémoire, d'où le nom de *deuxième chance*. En conséquence, un intervalle peut résider dans un registre donné à un moment et dans un autre à un autre moment. De plus, un intervalle est gardé dans un registre tant que ce registre n'est pas utilisé pour stocker un autre intervalle. À la suite de l'allocation, une phase de résolution est nécessaire, afin de régler des conflits d'emplacement des intervalles aux frontières des BB. En effet, cet algorithme ne modélise pas le flot de contrôle du programme. La phase de résolution est aussi expliquée dans l'article décrivant l'algorithme et une version simplifiée, qui répond aux besoins de notre algorithme, a été implémentée pour notre compilateur.

Dans notre cas, la représentation linéaire du code est réalisée en traversant le GFC en profondeur d'abord. Il serait utile d'expérimenter d'autres linéarisations à l'aide de différentes traversées puis d'examiner la qualité de l'allocation de registres. Une phase qui collecte les durées de vie, selon l'algorithme décrit dans [31], est effectuée pour connaître

la prochaine utilisation et la vivacité d'une variable à chaque instruction. Ces informations permettent de calculer les intervalles de vie, principalement lorsque le contrôle de flot effectue des retours en arrière.

Les intervalles de vie n'ont pas de deuxième chance d'allocation : un intervalle est associé à un registre physique pour l'entièreté de sa durée. Cependant, contrairement au balayage linéaire classique, où un intervalle réside soit en mémoire, soit dans un registre, *tous* les intervalles se voient attribuer un registre au début de l'intervalle de vie. Un intervalle réside dans son registre jusqu'à ce qu'un autre registre symbolique dont l'intervalle est vivant soit utilisé et ait le même registre physique associé. Dans ce cas, l'intervalle résidant actuellement dans le registre physique est stocké temporairement en mémoire, jusqu'à sa prochaine utilisation.

L'attribution d'un registre à un intervalle est déterminée ainsi : si des registres sont disponibles pour l'allocation, un registre physique est choisi au hasard et ce registre est attribué à l'intervalle. Sinon, le registre contenant l'intervalle vivant se terminant le plus tard est attribué; on espère que l'utilisation de la variable décrite par l'intervalle est utilisée le plus loin possible à partir du moment courant et que la vie de l'intervalle courant se trouve dans un « trou » de l'intervalle de vie se terminant le plus tard. Cette heuristique a été choisie parmi d'autres possibles.

Des mesures particulières doivent être prises lorsque les deux opérandes d'une opération utilisent le même registre : un des deux opérandes se voit attribuer un autre registre pour cette instruction seulement.

Pour réaliser cet algorithme, différentes informations doivent être conservées :

- L'association des intervalles et des registres symboliques.
- L'emplacement actuel des intervalles, c'est-à-dire dans un registre ou en mémoire.
- L'emplacement de stockage temporaire d'un intervalle lorsqu'il doit résider en mémoire. Cet emplacement est décrit par un déplacement symbolique sur la pile, tel que décrit dans la section **5.3.1**. Un nouvel emplacement est généré pour chaque intervalle.

Résolutions des conflits

Tout comme dans l'algorithme de balayage linéaire avec deuxième chance, des conflits d'emplacement à la frontière des BB surviennent. Ces conflits sont attribuables à la nature linéaire de l'algorithme d'allocation, qui ne tient pas compte du flot de contrôle. Un exemple d'un tel conflit est décrit dans la Figure 19.

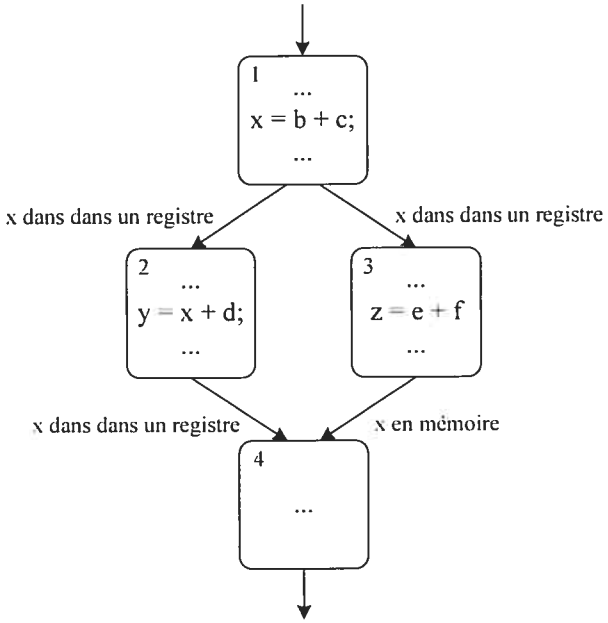


Figure 19 : Conflit d'allocation de registre

La disposition des BB de cet exemple pourrait représenter une structure *if-then-else*. L'allocation de registres commence par le bloc 1. Dans ce bloc, un registre *r1* est alloué à l'intervalle de la variable *x*, qui est rencontrée pour la première fois (donc l'intervalle débute sa vie), alors que nous supposons que les variables *b* et *c* ont déjà un registre qui leur est alloué, des registres *r2* et *r3* respectivement. L'allocation traite ensuite le bloc 2. L'intervalle de y débute ici et on doit lui associer un registre. Un registre *r4*, qui ne contient pas d'intervalle, lui est alloué. Supposons, pour les besoins de l'exemple, que le bloc 3 est le suivant dans l'ordre d'allocation. L'intervalle de la variable *z* débute dans ce bloc. Par contre, supposons qu'aucun registre physique n'est libre, alors un registre contenant déjà un intervalle doit lui être alloué. Supposons que l'heuristique choisisse d'allouer le registre *r1*, contenant *x*, à *z*. L'intervalle de *x* doit alors être stocké en mémoire temporairement jusqu'à sa prochaine utilisation. Lors de la jonction du flot de contrôle des

blocs 2 et 3, l'allocation ne dispose pas d'informations cohérentes : pour le bloc 2, le registre *r1* contient *x*, alors que ce registre contient *z* pour le bloc 3 et *x* se trouve en mémoire. Alors, l'allocateur ne sait pas quoi faire dans le bloc 4.

Pour solutionner ce problème, une structure garde l'emplacement des intervalles au début et à la fin des blocs et une phase de résolution analogue aux nœuds *phi* de la forme SSA [39], est exécutée. Le contenu des registres qui sont incohérents est stocké en mémoire à la fin des blocs de base et est rechargé lorsqu'un intervalle est utilisé dans le bloc où le flot de contrôle est joint.

Sauvegarde des registres

Une fois l'allocation terminée, on connaît exactement quels registres physiques sont utilisés dans une méthode. Grâce à cette information, des instructions sont ajoutées dans le but de sauvegarder les registres à l'entrée des méthodes, pour les registres dont la responsabilité tient de l'appelé (*callee-save*), ou au site d'appel d'une méthode, pour les registres dont la responsabilité tient de l'appeleur (*caller-save*). Des instructions de chargement des registres sont aussi ajoutées à la fin dans l'épilogue d'une méthode et au retour de l'appel d'une méthode, ayant pour effet de recharger le contexte d'avant l'appel.

5.3.4 Résolutions des déplacements symboliques

Afin d'obtenir le code exécutable final, une valeur réelle doit être attribuée aux déplacements symboliques qui représentent un déplacement par rapport au pointeur de la pile dans le bloc d'activation. Cette valeur dépend du nombre d'éléments dans le bloc d'activations, comme les variables locales, les arguments des méthodes, les sauvegardes de registres, etc. Pour un élément donné, un déplacement est calculé en fonction du « rang » de l'élément courant dans le groupe d'éléments du même type (par exemple, le $i^{ème}$ registre sauvegardé) et du nombre total d'éléments sauvegardés sur la pile. Le calcul du nombre d'éléments sur la pile est accompli en parcourant le code des BB. Dans une structure de données associée à chaque type d'éléments, les références à un déplacement pour ce type sont regroupées. Le nombre de différentes références d'un type donne le nombre d'éléments pour ce groupe.

Afin d'obtenir une meilleure allocation de la mémoire de la pile, une analyse concernant l'utilisation de la mémoire peut être réalisée. Cette analyse a pour but

d'attribuer des cases mémoire de la pile aux déplacements symboliques qui ne sont pas vivants en mémoire en même temps.

5.3.5 Génération du fichier

L'étape finale du flot logiciel consiste à générer le fichier résultant du flot. Le code assembleur des méthodes est écrit dans un fichier, suivi des données du programme. Ces données sont les champs statiques des classes, récupérés à l'aide de la réflexion et accessibles par les objets de type `ClassInfo`. Des tables de sauts, nécessaires à l'implémentation des instructions `switch` et les tables d'appels virtuels sont aussi écrites.

Ce fichier est ensuite utilisé comme programme source par MB-GCC de Xilinx, afin de l'assembler et de le lier (*assemble and link*) en vue d'obtenir le fichier exécutable du programme.

5.4 Flot matériel

À partir de la RI créée par le flot commun, le flot matériel est responsable de générer des ASM qui implémentent les méthodes identifiées en matériel.

Le but principal à atteindre lors de la transformation des méthodes en ASM est d'obtenir le plus grand parallélisme possible afin de calculer le résultat de la méthode en un petit nombre de cycles. En effet, pour qu'un ASM soit utile au système, il faut, d'une part, que le calcul soit obtenu plus rapidement que s'il était implémenté en logiciel, et, d'autre part, profiter du temps de calcul utilisé par le module matériel pour effectuer d'autres opérations en logiciel. Cette dernière condition peut être réalisée à l'aide d'un ordonnancement des instructions : les instructions qui de lancent l'exécution du matériel sont exécutées dès que les arguments de la méthode sont disponibles; le résultat est récupéré le plus tard possible par rapport au lancement de l'exécution matérielle (sans pour autant retarder une utilisation de ce résultat par une autre instruction). Par contre, comme il a été mentionné précédemment, une phase d'ordonnancement des instructions n'est pas implémentée, mais serait nécessaire pour obtenir du code de meilleure qualité.

Le résultat du flot matériel est un ensemble de fichiers qui décrivent les méthodes matérielles en CASM. Le compilateur CASM est utilisé pour convertir le code en VHDL, qui sera alors intégré au système final, synthétisé par les outils de Xilinx.

Les traitements effectués par le flot matériel sont illustrés par la Figure 20 et sont décrits plus en détail dans cette section.

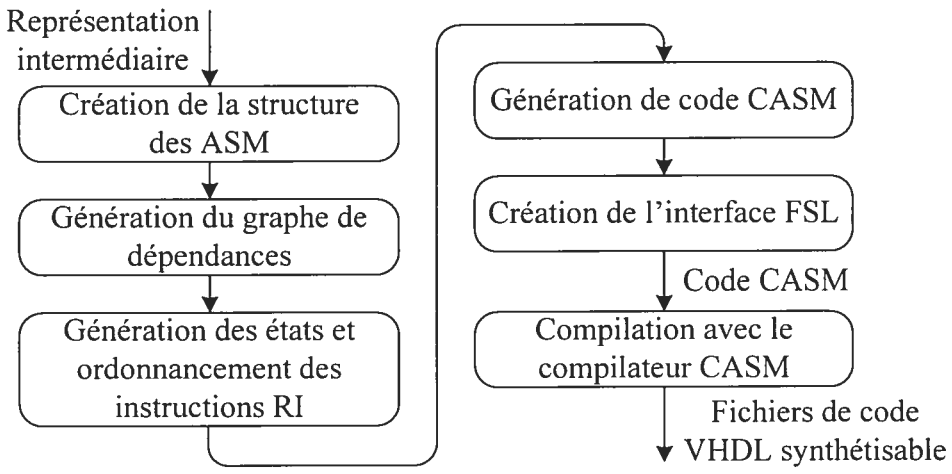


Figure 20 : Phases du flot matériel

5.4.1 Génération de la structure de base des ASM

La première étape du flot matériel consiste à générer le squelette de la structure des ASM, c'est-à-dire d'identifier les entrées, la sortie et les variables locales. Les états et le code des états d'un ASM sont générés plus tard.

Les entrées d'un ASM sont définies à partir des types de données des arguments de la méthode implémentée. Le nombre de bits d'une entrée donnée est déterminé par le nombre de bits utilisé par le Microblaze pour représenter le type de l'argument. Par exemple, un argument de type `short`, c'est-à-dire un entier de 16 bits, est représenté par un vecteur signé de 16 bits dans l'ASM, alors qu'un entier de type `uint`, c'est-à-dire un entier non signé de 32 bits, est représenté par un vecteur non signé de 32 bits. La taille de la sortie est définie de la même manière, soit en utilisant le type de la valeur de retour de la méthode. Le protocole de communication auquel on a recours pour les entrées/sorties est toujours *full synchronized*, afin d'être conforme au protocole de communication du

canal FSL. Ce protocole est le même que celui utilisé par les ASM complètement synchronisés.

Deux ensembles de variables locales sont utilisées par un ASM : les variables locales qui correspondent aux variables locales de la méthode et celles qui correspondent aux entrées de l'ASM. Pour le premier ensemble, le type d'une variable locale de l'ASM est déterminé par le type de la variable locale de la méthode. Par exemple, un registre signé de 32 bits est utilisé pour représenter un entier de 32 bits. Pour le deuxième ensemble, le type de la variable locale est directement donné par le type de l'entrée. Par exemple, un vecteur non signé de 32 bits donne un registre non signé de même taille. Ce deuxième ensemble de variables locales s'impose si on veut utiliser les données d'entrée d'un ASM : une fois l'entrée reçue, elle est transférée dans la variable locale correspondante et devient manipulable.

5.4.2 Affectation des instructions aux états

Les états d'un ASM doivent être créés et les instructions RI doivent être assignées aux états. Il faut, comme il a été mentionné plus haut, maximiser le parallélisme en réalisant le plus d'opérations possibles dans un même état. À cette fin, un ordonnancement des instructions RI est effectué. Celui-ci tente d'affecter le plus d'instructions possibles à un état. Les BB constituent l'unité de base de la phase d'ordonnancement, c'est-à-dire que les instructions RI d'un seul bloc sont considérées pour la création des états (ce qui signifie qu'un ASM contiendra au moins autant d'états que le nombre de BB du GFC). Les contraintes par rapport au nombre d'instructions pouvant être ordonnancées dans un même état proviennent des dépendances entre les instructions. Ces dépendances prennent trois formes :

- Dépendance de contrôle : une instruction RI de branchement doit être effectuée en même temps ou après les instructions précédant ce branchement.
- Dépendance de données : les opérandes d'une opération sont produits par une autre instruction. Dans un tel cas, l'instruction RI utilisant la donnée comme opérande ne peut être exécutée qu'après l'instruction productrice de la donnée.

- Dépendance de mémoire : l'ordre d'accès à la mémoire sur une même donnée doit être conservé pour assurer la consistance des données. Toutefois, comme les ASM n'ont pas accès à la mémoire, il n'y a pas lieu de tenir compte de cette contrainte.

Il est donc nécessaire d'identifier ces dépendances pour ordonnancer les instructions RI dans les états d'un ASM.

Graphe de dépendance

Un graphe de dépendance exprime les dépendances entre les instructions. Un arc d'une instruction vers une autre représente un ordre entre ces deux instructions : l'instruction à la queue de l'arc doit être exécutée avant celle à la tête.

Le graphe de dépendance est créé à partir de la représentation linéaire de la RI et est réalisé à l'aide de l'algorithme décrit dans [77]. La réalisation du graphe est simple, dû au sous-ensemble du CIL supporté par les méthodes matérielles. Par exemple, il n'est pas nécessaire de faire des analyses poussées en ce qui concerne les ambiguïtés des accès mémoire.

Ordonnancement et génération des états

À la suite de la création du graphe de dépendances, l'ordonnancement des instructions se produit. Les algorithmes *le plus tôt possible* (ASAP, de *As Soon As Possible*) et *le plus tard possible* (ALAP, de *As Late As Possible*) sont implémentés. L'algorithme ASAP ordonnance une instruction dès que ses prédécesseurs dans le graphe de dépendance ont été ordonnancés. À l'inverse, dans ALAP, les instructions n'ayant plus de successeurs sont ordonnancées en « premier ». L'ordonnancement est par la suite inversé pour obtenir un ordonnancement à partir des premières instructions. Ces algorithmes donnent des résultats généralement corrects, mais pas optimaux. D'autres algorithmes plus complexes doivent être implémentés.

Certains nœuds de la RI contiennent des résultats intermédiaires provenant de certaines opérations. Par exemple, en CIL, une addition de plus de deux éléments doit produire un résultat intermédiaire sur la pile qui est consommé en étant additionné avec les éléments suivants. Dans l'ordonnancement, ces nœuds intermédiaires ne sont pas considérés, sinon un grand nombre d'instructions utilisant un petit nombre d'opérandes

(au maximum deux) seraient effectuées. Seule l'opération complète produisant le résultat final est ordonnancée.

L'ordonnancement des instructions d'un BB produit une liste d'étapes où chaque étape contient une liste d'instructions à exécuter. Pour chaque étape, un état de l'ASM est créé et les instructions RI de cette étape lui sont assignées. Ces instructions, une fois traduites en code CASM, formeront le code de cet état.

5.4.3 Génération de code

Les instructions RI d'un état doivent être transformées en code CASM. L'algorithme de génération de code est semblable à celui utilisé pour générer le code Microblaze, alors que le code des opérandes est généré avant le code du nœud courant. Cependant, une distinction importante à faire entre les deux algorithmes provient de la structure du code CASM par rapport à l'assembleur : les instructions du code assembleur utilisent deux opérandes, alors que les instructions en code CASM se rapprochent beaucoup plus d'un langage de haut niveau et utilisent un nombre arbitraire d'opérandes. Par exemple, une branche de l'arbre de la RI représentant une addition de plusieurs données génèrerait plusieurs instruction en assembleur, alors que le résultat temporaire de l'addition de deux éléments est stocké dans un registre temporaire, puis utilisé comme opérande d'une autre instruction d'addition et ainsi de suite. La même opération en CASM est réalisée en une seule instruction qui additionne tous les éléments en même temps.

Le choix des instructions CASM est généralement trivial, puisque le langage contient des constructions et une syntaxe de haut niveau qui permettent une traduction presque directe.

Il faut souligner quelques éléments importants de cette phase :

- Une instruction RI de branchement inconditionnel génère un saut au premier état du BB qui est la cible du branchement.
- Une instruction RI de branchement conditionnel génère une structure de type `if-then-else`. Des sauts aux premiers états des BB cibles sont générés dans le `then` et le `else`. Afin d'obtenir un code de meilleure qualité, il faudrait, si possible, que le code des BB cibles soient intégrés dans le corps du `then` et du `else`.

- Même si des structures de type `while` sont supportées par le langage CASM, elles ne sont pas utilisées. Les boucles sont réalisées avec les structures `if-then-else` et les sauts aux états.
- Pour les états supplémentaires générés lors de l'ordonnancement, des sauts deviennent nécessaires pour passer de l'un à l'autre.
- Lors d'une affectation, si le type de la partie de gauche n'est pas le même que le type de la partie de droite, une conversion explicite est appliquée. Par exemple, une expression additionnant des entiers non signés donne un résultat non signé. Si le résultat de cette expression est assigné à une variable locale signée, le résultat est d'abord converti.

5.4.4 Interface FSL

À la suite de la génération de code, la structure des ASM est complète et la fonctionnalité de la méthode est implémentée. Cependant, le mécanisme permettant la communication des données du processeur vers un ASM et vice-versa n'est toujours pas réalisé. Ce mécanisme permet à un ASM de recevoir des données par le canal FSL qui lui est assigné et de renvoyer la donnée produite par le calcul.

Grâce au langage CASM, la génération de l'interface FSL est simple, ce qui est principalement dû au fait que le protocole de communication utilisé par un canal FSL est le même que celui utilisé par le protocole *full synchronized* de CASM (à un signal près, qui ne doit qu'être inversé). Ainsi, toutes les *poignées de main* (*handshake*) effectuées entre le module matériel et le canal FSL sont déjà implémentées. Il ne faut qu'ajouter un nombre donné d'états à un ASM. Ces états sont responsables de recevoir un à un les arguments de la méthode et de les affecter aux variables locales correspondantes. Un état est nécessaire pour chaque argument puisque les arguments sont transférés séquentiellement sur le canal FSL à travers la queue réalisant le pont entre le Microblaze et le module matériel. Ils sont donc récupérés séquentiellement par le module matériel. Un dernier état est nécessaire pour transférer le résultat d'une méthode, contenu dans une variable locale, vers le signal de sortie de l'ASM.

La Figure 21 illustre un exemple d'ASM implémentant l'algorithme du plus grand commun diviseur (PGCD) d'Euclide. À gauche, les premiers états et le dernier composent

l'interface FSL, alors que les états du centre, montrés plus en détail à la droite de la figure, réalisent les calculs.

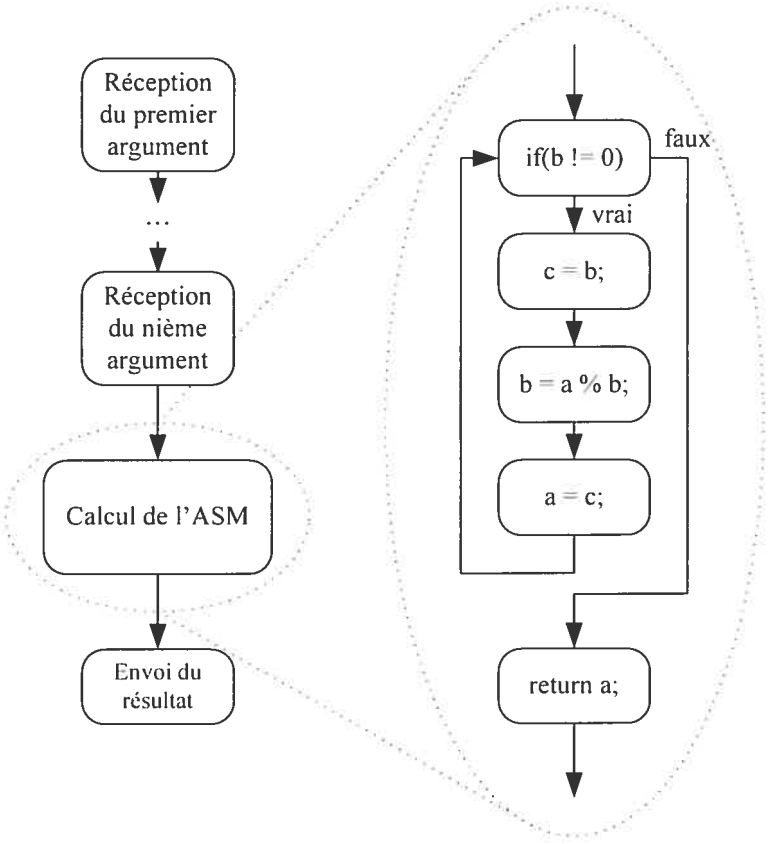


Figure 21 : Structure de l'ASM du PGCD d'Euclide

Finalement, un module VHDL doit être généré, qui permet d'enrober (*wrapper*) un ASM et de l'intégrer dans le système final. Ce module définit des paramètres nécessaires au système (comme le nombre d'arguments passés au module matériel) et transmet les signaux d'entrée et de sortie entre le bus FSL et l'ASM. Un ASM est instancié dans ce module.

6 Expériences, manipulations et résultats

Dans cette section, nous présentons deux exemples d'applications d'expérimentation ayant recours à notre outil. Dans un premier cas, une application naïve démontre les traitements réalisés par le compilateur. La deuxième application se veut une adaptation d'une application réelle, utilisée pour obtenir des mesures en rapport au code du logiciel et du matériel générés par l'outil.

Comme il a déjà été mentionné, le compilateur est dans un état de prototype et le but à atteindre lors des expériences consiste à obtenir un outil fonctionnel qui permet de valider la méthodologie de conception proposée. Cela signifie que peu d'importance est accordée aux performances de celui-ci, comme par exemple le temps nécessaire à la compilation d'un système ou la qualité du code assembleur généré. De telles mesures sont toutefois prises en considération et aident à cibler les problèmes potentiels pour les générations futures de l'outil.

6.1 Première étude de cas

6.1.1 Description de l'application

La première application traitée est simple et sert à démontrer les capacités du compilateur et comment les traitements sont réalisés. Nous avons écrit une application en C# qui fait usage de différents mécanismes proposés par le langage, comme l'héritage, la création des objets, les appels de méthode virtuels, etc. Ces mécanismes sont utilisés naïvement, voir même inutilement dans certains cas, mais sont nécessaires pour notre démonstration.

L'application réalise un tri par insertion d'un tableau d'objets de différentes classes. La relation d'ordre entre les objets, nécessaire pour déterminer si un objet est plus petit ou plus grand qu'un autre, est établie à l'aide d'une méthode qui calcule une valeur entière pour chaque objet. Cette valeur entière est obtenue à partir de la valeur des champs d'un objet.

L'application est composée de trois classes. La première classe, de type `BaseType`, sert super-classe aux deux autres. Les deux sous-classes, de types `SubTypeA` et `SubTypeB`, ont une relation directe d'héritage avec leur super-classe, c'est-à-dire qu'elles mêmes ne sont pas des super-classes. Les objets des trois classes sont créés en spécifiant deux valeurs entières, qui sont affectées aux champs des objets. Une méthode virtuelle, réalisant une fonctionnalité d'impression d'un message, est définie dans la super-classe et est redéfinie dans les deux sous-classes. Ainsi, le comportement de cette méthode est différent selon le type de l'objet qui l'appelle. Dans la super-classe, on retrouve aussi une méthode qui, à partir des deux entiers spécifiés lors de la création des objets, calcule une valeur identifiant un objet. Cette méthode est employée dans une autre méthode qui a pour but de comparer deux objets : celui dont la valeur donné par la méthode d'identification est la plus grande est considéré comme l'objet le plus grand. L'ordre entre les objets est ainsi établi. La méthode `Main` est responsable des traitements à effectuer dans l'application. Elle se charge d'appeler une méthode qui créée un tableau de 32 éléments du types de la super-classe. Dans les 32 cases du tableau, des objets de l'un des trois types sont créés avec des valeurs aléatoires (les valeurs sont préalablement générées, puisque la génération de nombres aléatoires est produite par une classe de la librairie de .NET). Une fois le tableau initialisé, une méthode exécute un tri par insertion sur les éléments du tableau en utilisant la méthode de comparaison définie dans la super-classe. Le résultat final des traitements est le tableau d'objets ordonnés. Une méthode imprime le contenu du tableau sur la console.

Afin de mieux illustrer le fonctionnement du compilateur, nous donnons dans ce qui suit des exemples où la méthode qui calcule la valeur pour chaque objet est implémentée en logiciel et en matériel.

Le code de l'application en C# est donné dans l'annexe 1.

6.1.2 Description pas à pas des traitements

Nous expliquons et illustrons ici les étapes des traitements effectués sur le code pour obtenir le système logiciel/matériel final, en donnant comme exemple la même méthode qui est transformé en logiciel et en matériel.

Traitements du flot commun

Au départ, le programme original est compilé en code CIL par le compilateur de .NET. Ci-dessous se trouve le code de la méthode `Identity` qui calcule la valeur identifiant l'objet selon les deux paramètres qui lui sont donnés (ce sont toujours les deux champs de l'objet).

```
public int Identity(int b, int xp)
{
      int num = 1;
      if(b == 0)
            return 0;
      else if(xp == 0)
            return 1;
      else
      {
            for(int i = 0; i < xp; i++)
            {
                  num *= b;
            }
            return num;
      }
}
```

Code 1 : Code C# de la méthode `Identity`

Cette méthode implémente naïvement la fonction d'exponentiation où le premier argument est la base et le second l'exposant (un exposant négatif n'est toutefois pas traité par la méthode, puisque cela retourne un nombre fractionnaire qui n'est pas implémenté dans notre outil). Le code CIL correspondant est donné dans le Code 2 :

```
.method public hidebysig instance int32
            Identity(int32 b,
                     int32 xp) cil managed
    {
      // Code size       38 (0x26)
      .maxstack  2
      .locals init ([0] int32 num,
               [1] int32 i,
               [2] int32 CS$00000003$00000000)

      IL_0000:  ldc.i4.1
      IL_0001:  stloc.0
      IL_0002:  ldarg.1
      IL_0003:  brtrue.s   IL_0009

      IL_0005:  ldc.i4.0
      IL_0006:  stloc.2
      IL_0007:  br.s       IL_0024
      IL_0009:  ldarg.2
      IL_000a:  brtrue.s   IL_0010
      IL_000c:  ldc.i4.1
      IL_000d:  stloc.2
      IL_000e:  br.s       IL_0024
      IL_0010:  ldc.i4.0
      IL_0011:  stloc.1
      IL_0012:  br.s       IL_001c
      IL_0014:  ldloc.0
      IL_0015:  ldarg.1
      IL_0016:  mul
      IL_0017:  stloc.0
      IL_0018:  ldloc.1
      IL_0019:  ldc.i4.1
      IL_001a:  add
      IL_001b:  stloc.1
      IL_001c:  ldloc.1
      IL_001d:  ldarg.2
      IL_001e:  blt.s      IL_0014
      IL_0020:  ldloc.0
      IL_0021:  stloc.2
      IL_0022:  br.s       IL_0024
      IL_0024:  ldloc.2
      IL_0025:  ret
    } // end of method BaseType::Identity
```

Code 2 : Code CIL de la méthode `Identity`

Les instructions débutant par un « . » sont des pseudo-opérations et ne font pas partie du code exécutable. Ainsi, le nom et la signature de la méthode sont indiqués par la pseudo-opération `.method` à la première ligne; la profondeur maximale de la pile est indiquée par `.maxstack` à la 6ème ligne; les variables locales de la méthode sont indiquées

par `.locals` à la 7ème ligne. La troisième variable locale (qui a l'indice 2) est utilisée pour contenir la valeur de retour.

Dans le code, toutes les instructions sont identifiées uniquement par une étiquette qui correspond au déplacement d'une instruction par rapport à la première. Les cibles des branchements sont indiquées par l'étiquette de la cible.

À l'aide de la réflexion, la structure du programme est lue et une représentation interne est créée. Le code CIL de chaque méthode est lu et stocké dans une liste d'instructions. À partir de cette liste, le GFC doit être généré à l'aide des techniques expliquées précédemment. Le GFC de la méthode `Identity` est donné sur la Figure 22.

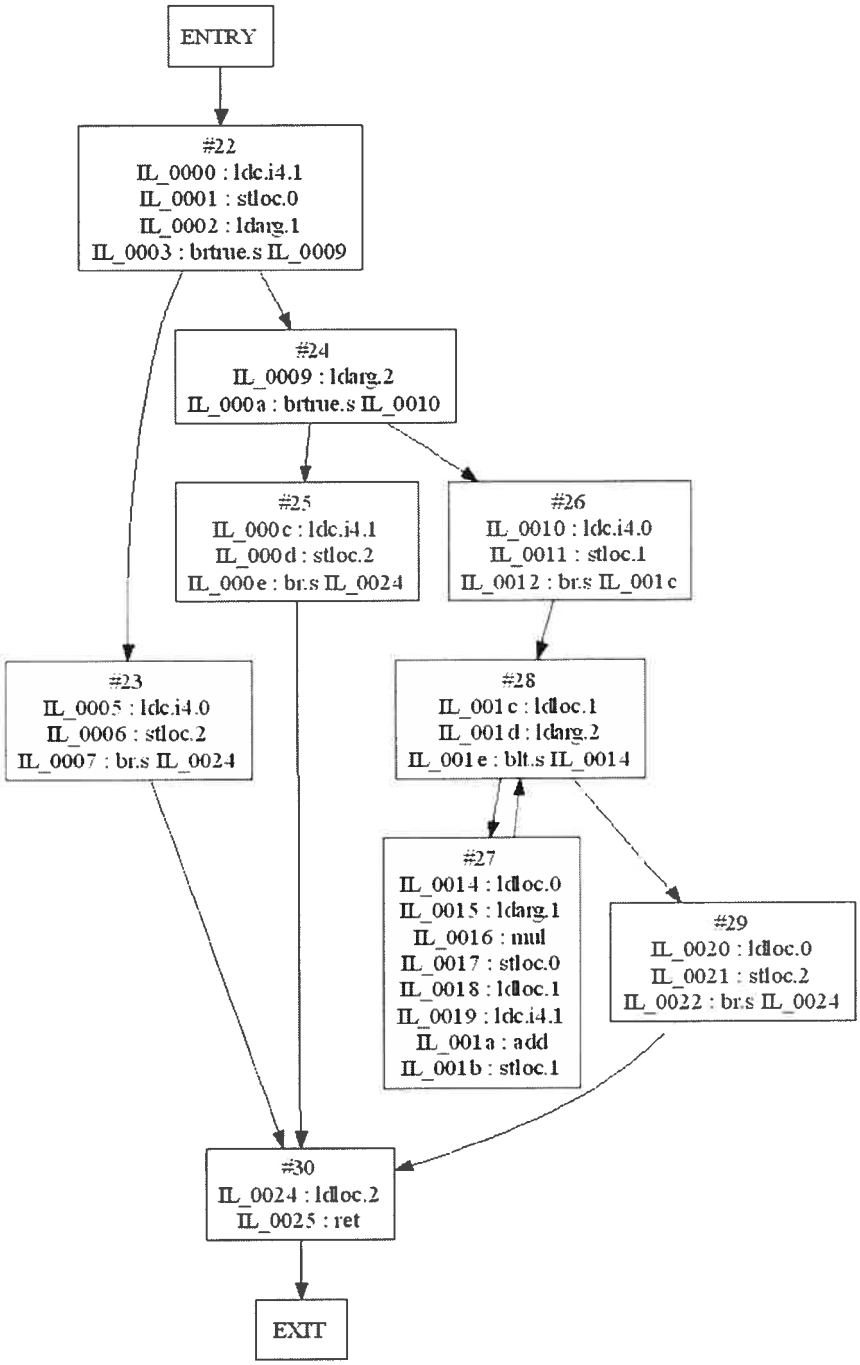
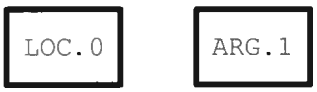


Figure 22 : Graphe de flot de contrôle de la méthode `Identity`

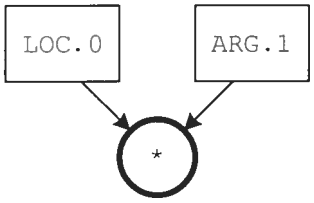
La génération de la RI est accomplie à partir des instructions contenues dans chaque bloc de base. La Figure 23 donne un exemple de transformation de code CIL du bloc de base #27 en RI. À la première étape, deux opérandes, la variable locale d'index 0 et l'argument d'index 1, sont empilés sur la pile. Ces opérandes sont dépilés à la deuxième

étape, lorsque l'opération de multiplication est traitée. À la troisième étape, une opération de stockage (qui correspond à une affectation) est traitée. Cette opération prend l'élément du dessus de la pile, c'est-à-dire le résultat de la multiplication, et l'affecte à la variable locale d'index 0. Cette instruction ne laisse rien sur la pile, ce qui veut dire qu'un nouvel arbre doit être créé. Un arbre semblable au premier est créé dans la quatrième étape, mais avec une addition. L'arbre complet de la RI de ce bloc de base est donné à la cinquième étape.

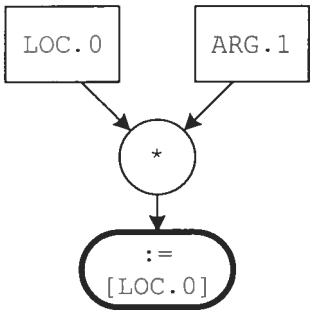
1.
```
IL_0014: ldloc.0
IL_0015: ldarg.1
```



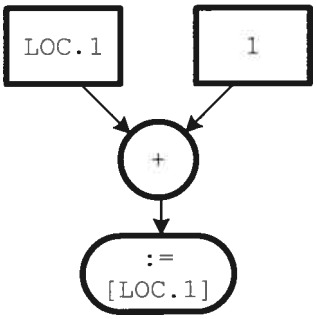
2.
```
IL_0016: mul
```



3.
```
IL_0017: stloc.0
```



4.
```
IL_0018: ldloc.1
IL_0019: ldc.i4.1
IL_001a: add
IL_001b: stloc.1
```



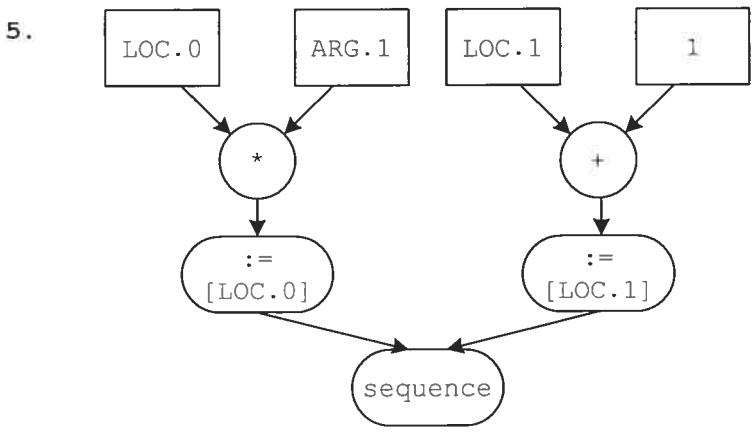


Figure 23 : Transformation en représentation intermédiaire du code CIL d'un bloc de base de la méthode `Identity`

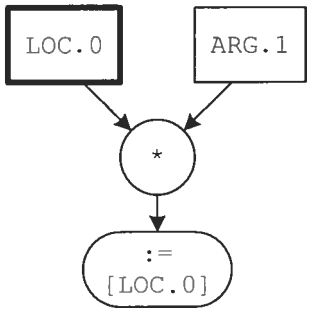
La RI du code de la méthode est prête à être utilisée par les flots logiciel ou matériel.

Flot logiciel

À partir de la RI générée dans le flot commun, une méthode identifiée comme logicielle est traitée par le flot logiciel avec pour effet de générer le code assembleur Microblaze correspondant. La Figure 24 illustre ce flot à l'aide de la méthode `Identity`.

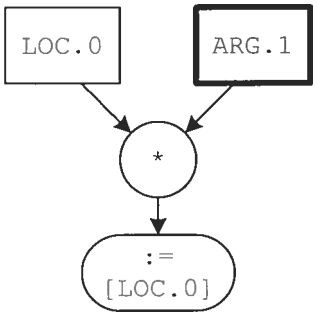
La première étape consiste à générer le code à partir des instructions RI. La Figure 24 montre le processus de génération du code du bloc de base #27, où le code des opérandes d'un nœud est toujours généré avant le code du nœud même. À la première étape, le nœud visité en premier est l'opération affectation. Puisqu'il possède des opérandes, celles-ci sont visitées avant de générer le code pour le nœud courant. Il en est de même pour l'opération d'addition. Le premier opérande visité est la variable locale d'index 0. Toutefois, aucun code n'est généré. Même chose à l'étape 2, où aucun code n'est généré pour l'argument d'index 1. À l'étape 3, une instruction de multiplication est générée, prenant comme argument les registres symboliques correspondant à ses opérandes. À la quatrième étape, le code est généré pour l'affectation. L'instruction ajoutée consiste à transférer le résultat de la multiplication dans le registre symbolique qui correspond à la variable locale d'index 0. L'étape 5 génère le code pour l'autre sous-arbre de la RI de ce bloc de base.

1.



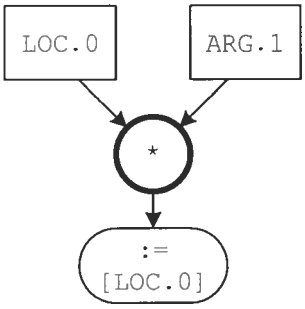
Liste d'instructions

2.



Liste d'instructions

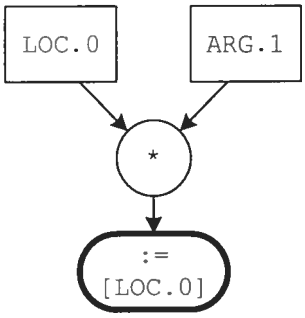
3.



Liste d'instructions

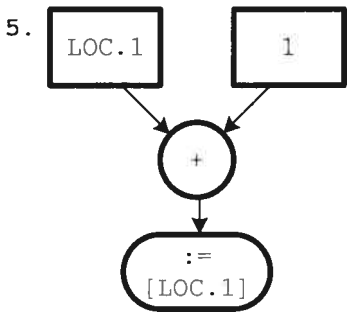
```
mul #r30, #LOC.0, $ARG.1
```

4.



Liste d'instructions

```
mul #r30, #LOC.0, $ARG.1
add #LOC.0, $r0, #r30
```



Liste d'instructions

```
mul #r30, #LOC.0, $ARG.1
add #LOC.0, $r0, #r30
addi #r32, #LOC.1, 1
add #LOC.1, $r0, #r32
```

Figure 24 : Génération de code assembleur à partir de la représentation intermédiaire d'un bloc de base de la méthode `Identity`

Le code assembleur généré est ensuite reconstitué en une représentation linéaire correspondant à l'ordre du code CIL original (Code 3). À partir de celui-ci, un nouveau GFC est généré, semblable à celui du CIL.

```
ObjectSort.BaseType.Int32_IdentityInt32_Int32 :
# entering method Int32 Identity(Int32, Int32)
swi $return_adress, $sp, %return.adr
addi $sp, $sp, -$stackFrameSpace
cmpu $TR1, $hp, $sp
blti $TR1, .no_memory
…
ObjectSort.BaseType.Int32_IdentityInt32_Int32.IL_0014 :
mul #r30, #LOC.0, $ARG.1
add #LOC.0, $r0, #r30
addi #r32, #LOC.1, 1
add #LOC.1, $r0, #r32
…
ObjectSort.BaseType.Int32_IdentityInt32_Int32.IL_0024 :
add $return_value, #LOC.2, $r0
# exiting method Int32 Identity(Int32, Int32)
addi $sp, $sp, $stackFrameSpace
lwi $return_adress, $sp, %return.adr
rtsd $return_adress, 8
or $r0, $r0, $r0
```

Code 3 : Code assembleur de la méthode `Identity` après la génération de code

On constate aussi que le code assembleur est loin d'être complet à cette étape. Seules les instructions ont été sélectionnées. Il faut encore allouer aux registres symboliques les registres physiques et résoudre les déplacements de la pile (en gras dans la figure). Certaines instructions ont été ajoutées pour le prologue et l'épilogue de la méthode (en

italique dans la figure). La partie de code du centre (entre les deux ellipses) est le code du bloc de base #27.

L'allocation des registres est l'étape suivante. L'algorithme calcule les intervalles de vie des registres symboliques et leur alloue un registre physique. Une fois que les registres sont alloués, le code assembleur est le suivant :

```
Int32 Identity(Int32, Int32):
ObjectSort.BaseType.Int32_IdentityInt32_Int32 :
# entering method Int32 Identity(Int32, Int32)
swi r15, r1, %return.adr
addi r1, r1, -$stackFrameSpace
swi r22, r1, %save.14
swi r29, r1, %save.13
swi r28, r1, %save.12
swi r31, r1, %save.11
cmpu r11, r19, r1
blti r11, .no_memory
…
ObjectSort.BaseType.Int32_IdentityInt32_Int32.IL_0014 :
mul r31, r28, r6
add r28, r0, r31
addi r31, r22, 1
add r22, r0, r31
…
ObjectSort.BaseType.Int32_IdentityInt32_Int32.IL_0024 :
add r3, r29, r0
# exiting method Int32 Identity(Int32, Int32)
lwi r31, r1, %save.11
lwi r28, r1, %save.12
lwi r29, r1, %save.13
lwi r22, r1, %save.14
addi r1, r1, $stackFrameSpace
lwi r15, r1, %return.adr
rtsd r15, 8
or r0, r0, r0
```

Code 4 : Code assembleur de la méthode `Identity` après l'allocation des registres

Les registres symboliques ont été remplacés par des registres physiques (en gras dans la figure) et les instructions de sauvegarde et de chargement des registres ont été ajoutées au début et à la fin du code de la méthode (en italique dans la figure).

La dernière étape consiste à finalement remplacer les déplacements symboliques sur la pile par des déplacements réels. Le Code 5 est le code de la méthode après cette étape.

```
Int32 Identity(Int32, Int32):
ObjectSort.BaseType.Int32_IdentityInt32_Int32 :
# entering method Int32 Identity(Int32, Int32)
swi r15, r1, 0
addi r1, r1, -20
swi r22, r1, 16
swi r29, r1, 12
swi r28, r1, 8
swi r31, r1, 4
cmpu r11, r19, r1
blti r11, .no_memory
…
ObjectSort.BaseType.Int32_IdentityInt32_Int32.IL_0014 :
mul r31, r28, r6
add r28, r0, r31
addi r31, r22, 1
add r22, r0, r31
…
ObjectSort.BaseType.Int32_IdentityInt32_Int32.IL_0024 :
add r3, r29, r0
lwi r31, r1, 4
lwi r28, r1, 8
lwi r29, r1, 12
lwi r22, r1, 16
addi r1, r1, 20
lwi r15, r1, 0
rtsd r15, 8
or r0, r0, r0
```

Code 5 : Code assembleur de la méthode `Identity` après la résolution des déplacements symboliques

Le code est maintenant complet et est prêt à être transformé en exécutable par l'assembleur de MB-GCC.

Flot matériel

Le flot matériel doit générer les ASMs pour les méthodes identifiées comme matérielles. Nous retraçons ici le cheminement des traitements effectués sur la méthode `Identity` qui est transformée en ASM.

En premier lieu, le flot matériel trouve les dépendances entre les nœuds de la RI pour pouvoir, par la suite, ordonnancer les instructions. La Figure 25 représente le graphe de dépendance des instructions de la RI du BB #27 de la méthode `Identity`. Dans ce cas, il n'y a aucune dépendance entre les nœuds à part les relations entre une instruction et ses

opérandes. De même, aucune dépendance n'existe entre les deux sous-arbres de la RI, ce qui signifie que les deux opérations d'affectation peuvent être réalisées en parallèle.

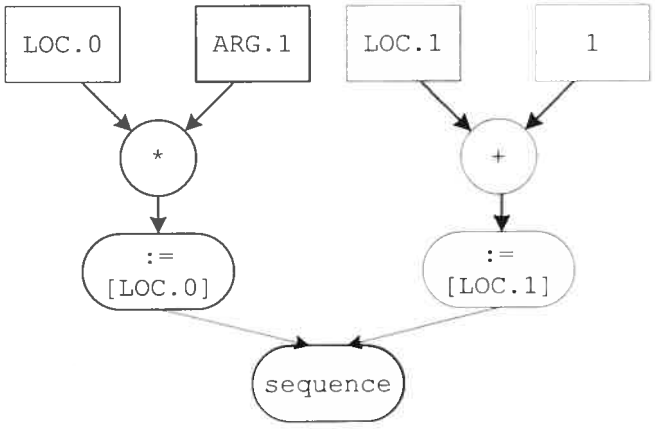
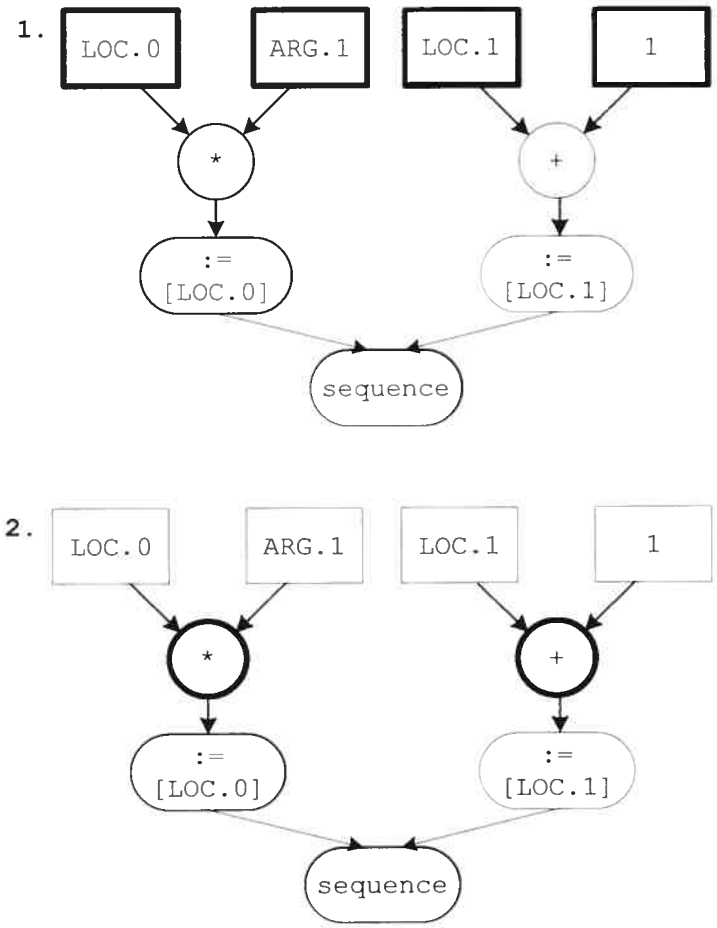


Figure 25 : Graphe de dépendance des nœuds de la représentation intermédiaire d'un bloc de base de la méthode `Identity`

En second lieu, la structure de base de l'ASM correspondant à la méthode doit être construite. Il faut identifier les entrées, la sortie et les variables locales. Ensuite, les états de l'ASM doivent être générés et les instructions de la RI doivent être assignés à ces états. Un état de base est généré pour chaque bloc de base et des états supplémentaires sont générés pour chaque étape de l'ordonnancement des instructions de la RI. La Figure 26 montre le processus d'ordonnancement à l'aide d'un algorithme ALAP. À la première étape, tous les nœuds n'ayant pas de prédécesseurs (dans le graphe de dépendances) sont soumis comme candidats. Dans ce cas-ci, les quatre données utilisées en entrée au calcul sont candidates. Toutefois, comme aucun de ces nœuds ne produit de résultat (en fait, aucun code n'est généré pour ces nœuds), ils ne peuvent être ordonnancés. Leurs descendants (dans le graphe de dépendance) sont alors soumis comme candidats. À l'étape 2, les opérations d'addition et de multiplication sont candidates. Cependant, encore une fois, ces nœuds ne peuvent être ordonnancés puisque le résultat de l'opération ne serait pas consommé. Leurs enfants deviennent donc candidats. Finalement à l'étape 3, les deux opérations d'affectation sont considérées. Ces deux opérations sont ordonnancées, puisqu'elles produisent du code qui réalise l'affectation du membre de droite au membre de gauche, lequel est représenté par une variable locale. L'ordonnancement final de ce

bloc de base contient donc les deux affectations exécutées en parallèle dans la même étape. Un état de l'ASM est formé à partir de chaque étape d'ordonnancement. Dans cet exemple, un seul état est généré.



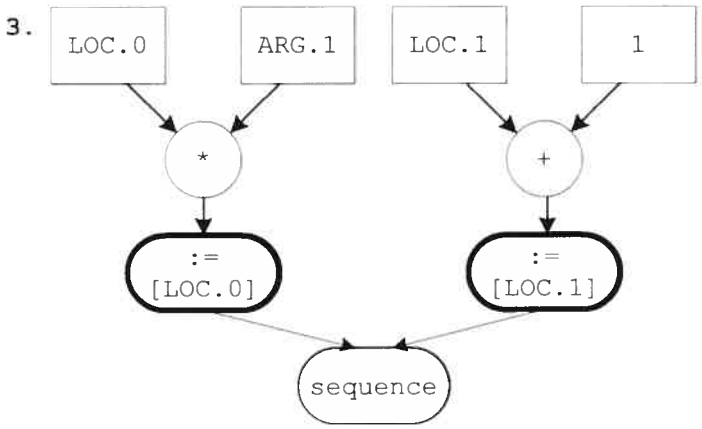


Figure 26 : Ordonnancement ALAP de la représentation intermédiaire d'un bloc de base de la méthode `Identity`

Le code CASM est obtenu presque directement à partir de la RI. En ce qui concerne le bloc #27, le code généré correspond tout simplement à deux affectations, avec comme membre de gauche la variable cible et comme membre de droite les opérations produisant le résultat, soit la multiplication et l'addition.

Par la suite, il reste à générer les états servant à la communication avec le canal FSL, qui se chargent de recevoir les arguments et de renvoyer la valeur de retour. Le code de l'ASM résultant est donné ci-dessous. Le code entier de l'ASM est donné dans l'annexe 2.

```
signed input arg_fsl_data_in_0{protocol="fs"} [32];
signed output arg_fsl_data_out_0{protocol="fs"} [32];

asm Int32_IdentityInt32_Int32 {

signed register local_LOC_0[32];
      signed register local_LOC_1[32];
      signed register local_LOC_2[32];
      unsigned register local_ARG_0[32];
      signed register local_ARG_1[32];
      signed register local_ARG_2[32];
      signed register local_ARG_3[32];

FSL_INTERFACE_START:
      local_ARG_0 := (unsigned)arg_fsl_data_in_0;
      goto FSL_INTERFACE_START_0;
FSL_INTERFACE_START_0:
      local_ARG_1 := arg_fsl_data_in_0;
      goto FSL_INTERFACE_START_1;
FSL_INTERFACE_START_1:
      local_ARG_2 := arg_fsl_data_in_0;
      goto n8_0;

n1_0:
if( local_ARG_2 != 0 )
      goto n3_0;
else
      goto n2_0;
end;
n2_0: …
n3_0:
      local_LOC_1 := (0);
      goto n5_0;
n4_0:
      local_LOC_0 := (( local_LOC_0 * local_ARG_1 ));
      local_LOC_1 := (( local_LOC_1 + 1 ));
      goto n5_0;
n5_0:
if( local_LOC_1 < local_ARG_2 )
      goto n4_0;
else
      goto n6_0;
end;
n6_0: …
n7_0: …
n8_0: …
n8_1: …
n0_0: …
FSL_INTERFACE_END:
      arg_fsl_data_out_0 := (local_ARG_3);
      goto FSL_INTERFACE_START;
}
```

Code 6 : Code CASM de la méthode `Identity`

Les états ayant comme préfixe `FSL_INTERFACE` (en gras) sont responsables de la réception des arguments et de l'envoi du résultat produit. Le cœur du calcul est effectué dans l'état `n4_0` (en italique), alors que les opérations d'addition et de multiplication sont effectuées en parallèle.

L'ASM est compilé par le compilateur CASM. Les modules VHDL générés sont ajoutés au système, synthétisés et ensuite téléchargés sur la plateforme.

6.1.3 Expériences et résultats

À partir de l'application naïve de tri du tableau d'objets, nous avons testé les deux différentes configurations du système avec la méthode `Identity` en logiciel ou en matériel. Un système de base est utilisé pour réaliser les expériences, auquel sont ajoutés des modules matériels générés à partir des différentes méthodes compilées en ASM. Ce système de base est constitué du processeur Microblaze, du module qui gère les entrées/sorties (UART), d'un compteur qui permet de calculer le temps d'exécution (en cycles) et des interconnexions entre les composants.

Nous avons mesuré le temps d'exécution de l'application (en cycles d'horloge). Notons que le temps d'exécution obtenu ne tient pas compte du temps requis pour initialiser le tableau d'objets ou pour l'impression de divers messages sur la console. Seul le temps nécessaire pour exécuter l'algorithme de tri est compté. Les résultats sont montrés dans le Tableau IV.

En matériel	*Temps d'exécution*	*Temps relatif*	*Portes logiques*	*Nombre relatif de portes*
Système de base avec `Identity` en logiciel	241 952	1	2 179 064	1
Système de base avec `Identity` en matériel	159 140	0,6577	2 199 900	1,0096

Tableau IV : Comparaison des temps d'exécution pour différentes configurations du système de l'application test 1

Même si cet exemple n'est pas très significatif et n'est utilisé que pour illustrer les transformations du compilateur, un gain de performance important est obtenu entre le système exécutant l'application seulement en logiciel et le système où la méthode `Identity` est en matériel. En fait, dans le deuxième cas, l'application ne prend que 65%

du temps pour s'exécuter par rapport au temps requis pour le premier cas. Cela s'explique de plusieurs manières.

Premièrement, les deux étapes du calcul qui requièrent des opérations arithmétiques sont réalisées en parallèle en matériel et. Ensuite, la méthode `Identity` est appelée très souvent dans l'application de tri, ce qui fait qu'une amélioration sensible est détectée. Finalement, le transfert d'arguments entre le processeur et la méthode en matériel est très rapide, du fait que seulement trois arguments doivent être transférés. En conséquence, le temps requis au transfert est négligeable par rapport au temps de calcul. Toutefois, les valeurs utilisées comme champs des objets à trier (à partir desquels la méthode `Identity` effectue son calcul) ont une influence sur les résultats. En effet, la méthode `Identity` est constituée d'une boucle qui s'exécute un nombre de fois égal à l'un des deux champs des objets; un nombre de petite taille entraîne un faible nombre d'itérations de la boucle. Si la méthode est réalisée en matériel et que le nombre d'itérations de la boucle de la méthode `Identity` est généralement bas, le temps supplémentaire requis pour le transfert des arguments peut égaler ou même excéder le temps gagné en réalisant le calcul à l'aide du matériel spécialisé.

6.2 Deuxième étude de cas

6.2.1 Description de l'application originale et des modifications

La seconde application choisie est écrite originellement en C. Elle est utilisée pour la détection des contours dans une image. Les images présentées à la Figure 27 et à la Figure 28 sont des exemples d'images d'entrée et de sortie du programme de détection de contour.

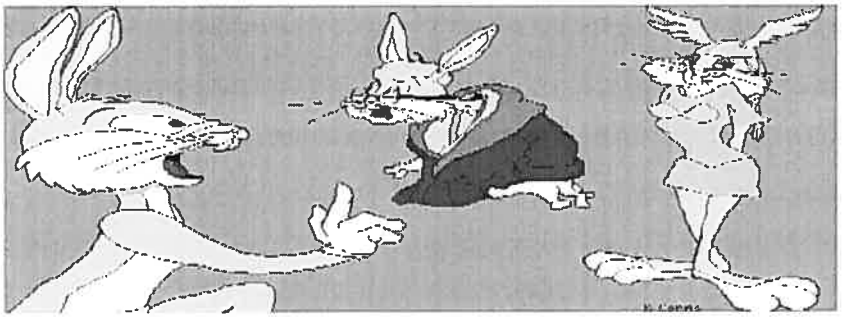


Figure 27 : Image avant la détection de contours

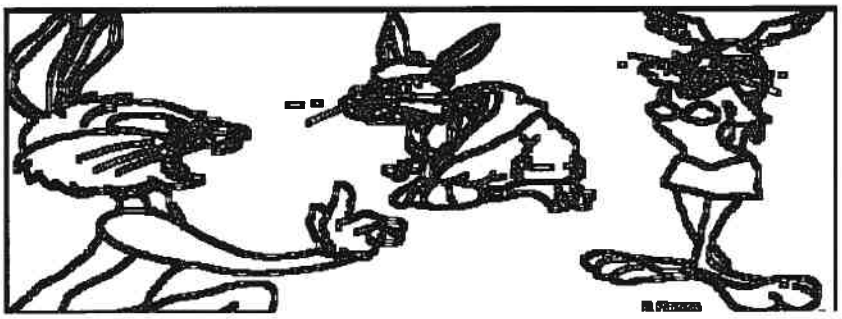


Figure 28 : Image après la détection de contours

La structure de l'application est décrite dans la Figure 29. Elle comporte trois parties principales : la lecture de l'image, les calculs permettant la détection des contours et la création de l'image résultante qui est composée des contours de l'image originale.

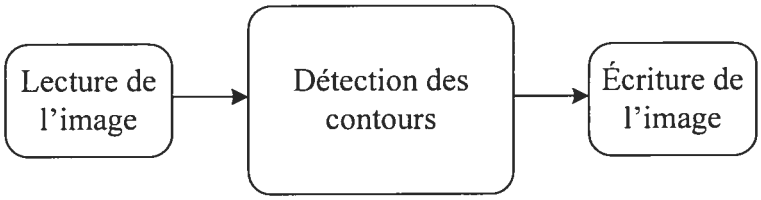


Figure 29 : Structure de l'application test 2

L'application originale est traduite en C# en vue d'être compilée par notre outil et elle est modifiée quelque peu afin d'être utilisée dans un système Microblaze. Seules les portions réalisant la lecture et l'écriture de l'image ont été modifiées. Le code réalisant les fonctionnalités principales de l'application, c'est-à-dire la détection de contours, n'a pas été altérée.

Les codes C et C# sont donnés dans les annexes 3 et 4 respectivement.

Lecture d'une image

L'application détecte les contours d'images de type PGM (Portable Gray Map) [25]. Dans l'application originale en C, une image résidant sur le disque dur est lue et stockée en mémoire vive. Chaque point de l'image, identifié par une valeur de 0 à 255, est conservé dans un tableau.

Dans l'application traduite en C#, les fonctions d'entrée/sorties sont réalisées à l'aide des fonctions prédéfinies de .NET. Comme celles-ci ne sont pas disponibles sur le système Microblaze et ne sont pas compilables par l'outil, cette portion du programme doit être modifiée. Ainsi, une méthode initialise un tableau contenant les valeurs des points de l'image. Les valeurs sont écrites directement dans l'application.

Calculs de détection de contours

Le traitement principal de l'application se déroule dans cette étape. Des calculs sont effectués sur le tableau contenant les valeurs des points de l'image originale afin d'extraire et d'ajuster les valeurs correspondant aux contours. Aucun changement n'a été effectué lors de la traduction de l'application originale en C vers celle en C#.

La détection des contours est réalisée à l'aide de 5 boucles successives qui effectuent des transformations sur l'image originale (illustré sur la Figure 30). Les deux premières boucles appliquent du flou gaussien en effectuant la moyenne d'un point donné et de ses voisins immédiats verticaux (dans la première boucle) et horizontaux (dans la seconde). La troisième boucle extrait les contours en calculant la différence absolue maximale entre un point de l'image et ses huit voisins immédiats. L'image obtenue après cette étape est complètement noire, à l'exception des contours qui sont plus pâles. La quatrième boucle inverse la valeur des points de l'image pour que les contours soient plus foncés que le reste de l'image. Finalement, la cinquième boucle sature les points des contours en les noircissant ou blanchissant complètement, et détecte les points racines, c'est-à-dire les points de l'image qui correspondent au « centre » d'un contour. La détection de ces points est effectuée en considérant un point et ses huit voisins immédiats; si le point considéré est le plus foncé du groupe, celui-ci est blanchi complètement.

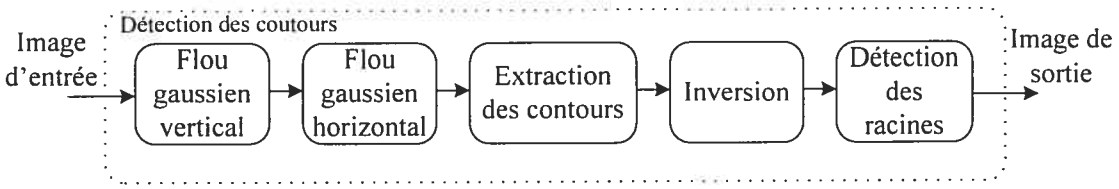


Figure 30 : Étapes des calculs de la détection des contours

La Figure 31 représente un exemple de l'évolution d'une image traitée par l'application. L'image de gauche est l'image de départ et celle de droite est l'image finale, alors que les images entre les deux sont les images intermédiaires obtenues après chacune des boucles.

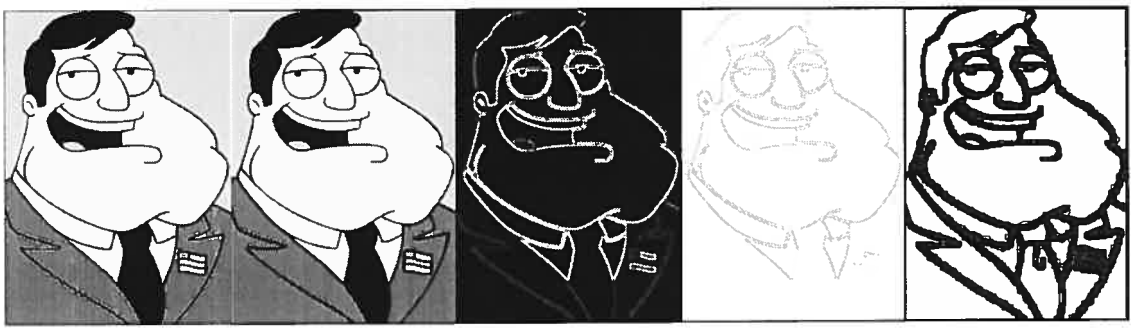


Figure 31 : Évolution d'une image après chaque boucle

Écriture d'une image

Une fois l'image finale obtenue, c'est-à-dire lorsque la valeur de tous les points de l'image résultante est stockée dans le tableau final, il faut écrire celle-ci sur le disque. La nouvelle image obtenue, encore de type PGM, ne contient que les contours de l'image originale.

Dans l'application modifiée en C#, encore une fois, les méthodes permettant l'écriture sur le disque proviennent des librairies prédéfinies de .NET et ne sont donc pas utilisable dans le contexte actuel. Cependant, les fonctions d'affichage sur la console sont autorisées par le compilateur et les valeurs des points de l'image finale sont imprimées sur la console.

6.2.2 Expériences et résultats

Afin de garantir la validité de la traduction de l'application de départ en C#, nous avons effectué des tests avec l'application originale en C, puis les mêmes avec l'application en C# dans l'environnement de développement de .NET. Une fois que les résultats obtenus dans les deux cas furent les mêmes, nous sommes passé à l'exécution de l'application sur la plateforme Microblaze.

Comme pour le premier exemple d'application, le système de base (processeur Microblaze, UART, compteur et interconnexions) est utilisé auquel les modules matériels lui sont ajoutés. Dans l'application modifiée, quatre méthodes étaient candidates à l'implémentation en matériel. Ces quatre méthodes sont celles qui réalisent le flou gaussien, l'inversion, la détection des racines et une méthode de maximum utilisée par la méthode d'extraction des contours. Cette dernière méthode n'a pu être implémentée en matériel puisqu'elle fait appel à la méthode de valeur absolue (un appel de méthode n'est pas permis à l'intérieur d'un ASM, même si le CASM le supporte). Les quatre méthodes implémentées en matériel sont appelées méthode 1, méthode 2, méthode 3 et méthode 4 respectivement. Les différentes configurations des systèmes utilisées sont les suivantes :

- Le système de base et tout le reste en logiciel;
- Le système de base, la méthode 1 en matériel et le reste en logiciel;
- Le système de base, la méthode 2 en matériel et le reste en logiciel;
- Le système de base, la méthode 3 en matériel et le reste en logiciel;
- Le système de base, la méthode 4 en matériel et le reste en logiciel;
- Le système de base, les méthodes 1 et 2 en matériel et les méthodes 3 et 4 en logiciel;
- Le système de base et les méthodes 1 à 4 en matériel.

Nous n'avons pas testé toutes les configurations possibles du système puisque seulement quelques tests sont nécessaires à l'obtention d'un nombre suffisant de mesures requis pour l'analyse de nos résultats. Le Tableau V donne le nombre de portes logiques approximatives nécessaires à l'implémentation des différentes configurations du système.

En matériel	*Nombre de portes logiques*	*Différence*	*Nombre relatif de portes logiques*
Système de base	2179064	0	1,0000
Système de base et méthode 1	2219462	40398	1,0185
Système de base et méthode 2	2184251	5187	1,0024
Système de base et méthode 3	2193672	14608	1,0067
Système de base et méthode 4	2195239	16175	1,0074
Système de base, méthode 1 et 2	2224408	45344	1,0208
Système de base, méthode 1, 2, 3 et 4	2254793	75729	1,0348

Tableau V : Nombre de portes logiques des différents systèmes

Pour les différentes configurations du système, nous avons testé deux tailles d'images différentes, soit une image de 64 points (8 par 8) et une image de 144 points (12 par 12). Cinq images différentes ont été testées et les valeurs des points de ces images ont été préalablement générées aléatoirement.

Le problème avec des tailles d'images plus grandes réside dans le fait que le programme et les données ne peuvent être contenus dans la mémoire du Microblaze avec la configuration actuelle. Afin de remédier à ce problème, nous pourrions utiliser de la mémoire hors de la puce.

Le Tableau VI et le Tableau VII donnent le temps requis (en cycles) pour l'exécution de l'application de détection de contours, avec les images de 64 et 144 points. Les temps donnés dans ces deux tableaux sont la moyenne des temps des cinq images générées aléatoirement. Il est important de mentionner que le temps d'exécution exclut le temps nécessaire à la création et à l'initialisation des tableaux utilisés et le temps utilisé à la présentation des résultats sur la console. Le temps n'inclut que l'exécution des 5 boucles. De cette façon, il est possible d'observer avec plus de justesse la variation de temps de l'algorithme due à l'implémentation des méthodes en matériel, puisque les temps constants au début et à la fin sont exclus.

En matériel	*Temps d'exécution moyen*	*Temps d'exécution moyen relatif*
Systeme de base	48253	1,0000
Systeme de base et méthode 1	45913	0,9515
Système de base et méthode 2	48028	0,9953
Système de base et méthode 3	48225	0,9994
Système de base et méthode 4	47684	0,9882
Système de base, méthodes 1 et 2	45688	0,9468
Système de base, méthodes 1, 2, 3 et 4	45092	0,9345

Tableau VI : Temps d'exécution de l'application de détection de contours avec une image de dimension 8 x 8

En matériel	*Temps d'exécution moyen*	*Temps d'exécution moyen relatif*
Systeme de base	211455	1,0000
Systeme de base et méthode 1	204395	0,9666
Système de base et méthode 2	210230	0,9942
Système de base et méthode 3	210722	0,9965
Système de base et méthode 4	208390	0,9855
Système de base, méthodes 1 et 2	203209	0,9610
Système de base, méthodes 1, 2, 3 et 4	199406	0,9430

Tableau VII : Temps d'exécution de l'application de détection de contours avec une image de dimension 12 x 12

Les résultats fournis dans les tableaux révèlent que le temps d'exécution n'augmente pas de façon linéaire avec le nombre de points de l'image. Comme les boucles effectuant les traitements sont en fait deux boucles imbriquées qui itèrent sur les deux dimensions de l'image, il est normal que l'augmentation du temps d'exécution soit quadratique. Dans ce cas, c'est a peu près le comportement observé, puisqu'une image composée d'un peu plus du double du nombre de points s'exécute en un temps a peu près quatre fois plus grand.

6.2.3 Analyse des résultats

En examinant les résultats, il est possible de constater une augmentation du nombre de portes logiques nécessaires l'implémentation des méthodes en matériel. Pour chacune des méthodes, l'augmentation du nombre de portes logiques est de moins de 2% du total des portes utilisées par le système de base. En fait, l'implémentation de toutes les méthodes candidates n'augmente le nombre de porte que de 3.5% par rapport au système de base.

De plus, le temps d'exécution est toujours plus petit avec l'implémentation des méthodes en matériel. En comparant chacune des méthodes individuellement, on constate

que l'implémentation en matériel de la première méthode amène la plus grande amélioration. Ce résultat est tout à fait normal puisque cette méthode est appelée à l'intérieur de deux des cinq boucles de l'algorithme. Lorsque toutes les méthodes sont implémentées en matériel, on peut observer une accélération de près de 7% du temps d'exécution par rapport à l'application en logiciel seulement.

Il faut aussi remarquer que l'amélioration du temps d'exécution du système lorsque la méthode 1 est implémentée en matériel est moins grande lors du traitement de l'image de 144 points par rapport à l'image de 64 points. Cette différence se répercute dans toutes les configurations où la méthode 1 est implémentée en matériel. Il est un peu difficile d'expliquer ce résultat. Même si le temps d'exécution de l'application dépend des données, les résultats sont la moyenne du temps d'exécution avec cinq images différentes. Une hypothèse possible pour explique cela est l'utilisation d'outils de synthèse automatique, qui nous empêche de comprendre complètement la manière dont le système est généré et de quelle façon il est composé. Des optimisations sont certainement faites par les outils de synthèse de façon tout à fait transparente pour l'usager. Il est difficile pour l'usager d'explorer toute l'architecture du système généré afin d'expliquer exactement tous les comportements.

Il est encourageant de voir de tels résultats lorsqu'on se souvient de l'objectif du projet, c'est-à-dire de produire un outil fonctionnel. Malgré tout, des gains de performances sont observés. Avec de meilleurs algorithmes et analyses du compilateur, nous sommes confiants que de bien meilleurs résultats seraient être obtenus.

6.3 Contraintes par rapport au choix d'applications pour les tests

Les contraintes du compilateur (sous-ensemble du CIL supporté, implémentation incomplète de l'environnement d'exécution et l'absence des classes prédéfinies de .NET) réduisent le nombre d'applications pouvant être choisies pour l'obtention de résultats. Toutefois, dans le cas présent, les deux applications utilisées pour les tests, bien qu'elles soient assez simples, permettent d'illustrer la méthodologie proposée. Des tests avec des applications plus complexes sont nécessaires pour l'obtention de résultats probablement plus significatifs et permettant une meilleur analyse des performances de l'outil.

7 Conclusion et perspectives

7.1 Synthèse du travail effectué

La conception efficace et sans erreur des systèmes embarqués est un problème complexe et difficile à traiter. La nature même des systèmes embarqués entraîne de nombreuses contraintes lors de leur conception, telles une faible consommation d'énergie ou de mémoire, une petite taille physique, une garantie d'exécution en temps réel, etc. De plus, la loi de Moore prédit que ces systèmes deviendront de plus en plus complexes puisqu'un plus grand nombre de composants seront éventuellement intégrés dans un même circuit. Cependant, l'évolution des méthodes et des outils de conception est incapable de suivre la tendance imposée par cette augmentation de la complexité.

De nombreuses recherches tentent d'augmenter le niveau d'abstraction auquel les outils et les méthodes de conception des systèmes embarqués se situent a fin de donner place à une conception efficace. Le compilateur proposé dans ce texte s'avère un tel effort. Avec cet outil, le comportement d'un système est spécifié à l'aide de langages de programmation de haut niveau d'abstraction (les langages supportés par le cadre d'application .NET). Des morceaux de code sont annotés et informent l'outil du partitionnement logiciel/matériel du système. Le compilateur transforme ensuite automatiquement cette description comportementale de haut niveau en une description du logiciel et du matériel qui s'implémente sur une plateforme FPGA. Un processeur à cœur logiciel, le Microblaze, est responsable de l'exécution du logiciel, alors que le matériel est réalisé à l'aide des blocs de logique reconfigurable de la plateforme.

Les résultats obtenus sur deux applications démontrent la faisabilité de notre approche. Dans ce contexte, l'implémentation de l'outil conduit à l'obtention des prototypes fonctionnels, par rapport celle d'un système qui, à la suite de nombreuses optimisations, devient hautement performant. Les algorithmes utilisés pour la transformation automatique de la description de haut niveau du système vers une description à implémenter sont classiques. Un grand nombre d'algorithmes produisant des résultats plus performants auraient avantage à être utilisés. Malgré tout, l'implémentation matérielle de certaines portions des applications de test ont amené des améliorations à la vitesse d'exécution et ce, à des coûts négligeables quant au nombre de portes logiques

supplémentaires. Cela démontre qu'il est possible d'améliorer l'outil afin d'obtenir de meilleurs résultats.

La méthodologie de conception des systèmes embarqués présentée ici, qui unifie et automatise les flots de conception du logiciel et du matériel à partir d'une seule et même description de haut niveau d'un système, est la même que celle réalisée par certains outils présentés dans la littérature. Toutefois, le compilateur développé ici se démarque de ces travaux par certaines caractéristiques. Premièrement, les outils de développement utilisés sont différents. Le cadre d'application .NET et son langage phare, C#, sont utilisés pour l'implémentation du compilateur et le langage CASM est utilisé pour la création des parties matérielles du système embarqué. Deuxièmement, une description d'un système embarqué est réalisée à l'aide d'un grand nombre de langages de programmation. En fait, comme le langage d'entrée du compilateur est une forme intermédiaire commune à tous les langages supportés par le cadre d'applications .NET, tout langage supporté par .NET est aussi supporté par le compilateur. Au meilleur de nos connaissances, aucune partie dorsale d'un compilateur ne génère du code assembleur pour le processeur Microblaze et des composants matériels à connecter à ce processeur à partir du langage intermédiaire de .NET.

De nombreux avantages méthodologiques découlent de la réalisation de notre outil. L'unification des flots de conception du logiciel et du matériel aux yeux du concepteur permet à une seule et même personne de réaliser un système, sans qu'il lui soit nécessaire de posséder des qualifications relatives aux mondes de conception, logiciel et matériel. De plus, l'outil offre une méthode de prototypage rapide du système tout en garantissant une transformation sans erreurs d'une étape de la conception vers une autre. Finalement, puisque l'outil supporte un grand nombre de langages de programmation différents, un concepteur utilise le langage de son choix pour concevoir un système.

D'autres avantages sont plutôt de nature à faciliter l'implémentation de l'outil lui-même. En effet, les mécanismes de .NET, soit l'environnement dans lequel le compilateur est réalisé, permettent un développement rapide et offre des dispositifs, notamment la réflexion, qui facilitent la compilation de programmes sources par l'outil. Aussi, le langage CASM possède une syntaxe et une sémantique d'assez haut niveau aptes à alléger la tâche de la transformation d'une description d'un programme de haut niveau vers une

description RTL du matériel. Un passage direct de la description de haut niveau vers une description de niveau RTL est bien plus complexe à réaliser, en plus de rendre difficile la production de code optimal.

Le compilateur réalisé n'est encore qu'un prototype et ne supporte pas toutes les fonctionnalités nécessaires à la compilation de n'importe quels types d'applications. Des contraintes relatives à l'utilisation des types de données ou des constructions du langage de haut niveau limitent l'ensemble des applications pouvant être implémentées par le compilateur sur un FPGA. De plus, l'environnement d'exécution de .NET est très riche et la totalité de ses fonctionnalités n'ont pas été implémentées. Un aperçu des avenues de recherche possibles sont discutées dans la section suivante.

Même si au stade actuel, notre méthodologie, supportée par notre outil, démontre que le développement de systèmes embarqués peut être facilité, notre recherche est un point de départ dans l'exploration des méthodes avancées nécessaires au développement de systèmes embarqués complexes. On peut penser à des outils qui, en utilisant des langages de programmation à la fine pointe de la technologie en entrée, réalisent toutes les étapes de la conception, du raffinement à l'implémentation, en passant par la vérification aux divers niveaux. Ainsi, il sera possible d'exploiter pleinement les possibilités offertes par l'inexorable avancement de la technologie afin concevoir les systèmes embarqués de demain.

7.2 Perspectives

Le compilateur réalisé n'implémente pas toutes les instructions du CIL et toutes les fonctionnalités du CLI. Le code source écrit dans un langage de haut niveau est limité par ces carences, malgré qu'il est tout de même possible d'écrire des applications relativement complètes, mais probablement pas assez complexes. Le manque principal se trouve du côté de l'environnement d'exécution, alors que certaines fonctionnalités importantes de l'environnement d'exécution virtuel du CLI ne sont pas implémentées. De plus, même si l'outil n'accepte qu'un sous-ensemble du CIL pour fonctionner sur la plateforme cible, il serait tout de même utile d'optimiser le code généré.

Dans cette section, nous décrivons plus en détail les fonctionnalités nécessaires et/ou importantes à implémenter dans le compilateur, soit pour obtenir une meilleure couverture de l'ensemble du CIL, soit pour obtenir du code de meilleure qualité. Nous discutons aussi des améliorations à effectuer à l'architecture du système pour permettre une meilleur utilisation des méthodes matérielles.

7.2.1 Extensions

Environnement d'exécution

Les fonctionnalités qui ne sont pas implémentées ont déjà été mentionnées dans la section 5.1.2. Certaines de ces fonctionnalités peuvent être implémentées facilement et ne nécessitent qu'un travail de développement. Toutefois, à nos yeux, les principales fonctionnalités manquantes sont l'accès aux classes prédéfinies, la récupération de la mémoire automatique, la gestion des exceptions, les multiples fils d'exécution et l'accès aux métadonnées à l'exécution.

Accès aux classes prédéfinies : Un très grand nombre de fonctionnalités de la plateforme .NET est implémenté par les classes prédéfinies. Un programmeur utilisant notre compilateur ne peut utiliser ces classes, ce qui limite beaucoup les traitements possibles par les programmes écrits.

L'implémentation de toutes les classes prédéfinies pour le Microblaze demande un travail colossal. Il est fortement probable qu'un certain nombre de ces classe ne puisse être implémenté, vu le manque de fonctionnalité du Microblaze et des FPGA par rapport à un processeur plus complexe, tel celui d'une station de travail conventionnelle. Un sous-ensemble de classes importantes pourrait être choisit.

Gestion des exceptions : Le mécanisme d'exception permet une gestion élégante des erreurs. Ce mécanisme est bâtit à l'intérieur du CIL, comme en témoignent les instructions particulières qui font affaire aux exceptions. Il est possible d'écrire des programmes sources sans les exceptions, mais certains programmes qui doivent être robustes s'appuient sur ces mécanismes pour obtenir une exécution qui ne se termine pas de façon inopinée lors d'erreurs.

Ramasse-miettes : Les techniques de récupération de la mémoire est un sujet assez connu et effervescent. De nombreux algorithmes ont été développés pour toute sorte de

contextes, comme les systèmes avec peu de mémoire ou ceux dont la consommation d'énergie doit être limitée. L'absence d'un ramasse-miettes ne contraint pas un utilisateur à ne pas utiliser certaines fonctionnalités d'un langage de haut niveau. Toutefois, un programme sémantiquement correct à la compilation risque de se terminer accidentellement en raison d'un manque de mémoire.

Fils d 'exécution : Des langages de haut niveau supportés par .NET, comme C# par exemple, prévoient des structures syntaxiques qui expriment la concurrence dans un programme à l'aide de fils d'exécution. Toutefois, l'implémentation de cette syntaxe est généralement accomplie par les classes prédéfinies de la plateforme. Il est donc difficile de réaliser les fonctionnalités de parallélisme en implémentant uniquement les instructions CIL. Comme dans la plupart des cas, l'obtention d'un environnement d'exécution plus complet requiert la compilation des classes prédéfinies de .NET, ou une approximation des fonctionnalités à l'aide des fonctions C fournies par le Microblaze. Ces fonctions, qui font partie de l'API du Xilkernel, servent à gérer des threads POSIX.

Accès aux métadonnées : Les métadonnées sont des éléments particuliers et intéressants du CIL. Elles offrent de l'information sur les données autant statiquement, lors de la compilation, que dynamiquement, durant l'exécution d'un programme. L'accès dynamique aux métadonnées deviendrait possible avec l'implémentation des tables qui les contiennent, de telle sorte que soient décrites les méthodes, les champs, etc. des types définis dans l'assemblage compilé. L'implémentation des tables permettrait l'implémentation d'un certain nombre d'instructions CIL qui utilisent les métadonnées durant l'exécution.

Amélioration des ASM et de l'architecture

Évidemment, il serait intéressant d'étendre le sous-ensemble du CIL supporté par les ASM. De cette manière, un plus grand nombre de fonctionnalités pourraient être implémentées et donc accélérées.

Accès à la mémoire : L'amélioration la plus utile à réaliser consiste à permettre aux ASM d'accéder à la mémoire. De cette façon, un grand nombre d'instruction CIL additionnelles peuvent être implémentées. Principalement, on pense à la manipulation d'objets et de tableaux. Pour ce faire, il faudrait modifier l'architecture du système pour

permettre la communication entre la mémoire et les ASM. Il faudrait aussi implémenter un protocole de partage des données : une donnée manipulée par un ASM à un moment donné ne doit pas être modifiée par le logiciel au même moment si on veut éviter les incohérences. Différentes possibilités sont envisageables. Par exemple, un ASM pourrait avoir une mémoire locale dans laquelle sont transférées les données nécessaires. Pendant ce temps, ces données ne seraient plus accessibles au processeur. Une autre option est d'utiliser un canal DMA (Direct Memory Access) permettant aux ASM de communiquer directement avec la mémoire du processeur. Dans tous les cas, des protocoles de synchronisation devraient être implémentés.

Appels de méthodes : Une autre fonctionnalité intéressante à implémenter en matériel est l'appel de méthodes. Un ASM pourrait par exemple utiliser un autre ASM correspondant à une autre méthode. Comme la récursivité est possible avec le langage CASM, on peut aussi imaginer un ASM qui s'appelle lui-même. Il serait aussi utile de réaliser les appels virtuels en matériel afin d'accélérer la résolution de ceux-ci.

Optimisation du code

Dans le but d'obtenir de meilleures performances du compilateur, des optimisations du code assembleur se révèlent nécessaires. De nombreuses optimisations s'avèrent possibles, notamment :

- L'incorporation de méthodes, afin de diminuer le temps d'appel de certaines méthodes, mais au coût d'une pression additionnelle sur les registres et d'une augmentation de la taille du code.
- L'optimisation du code lors d'appels des méthodes *feuilles*, c'est-à-dire celles qui ne contiennent aucun appel à d'autres méthodes. Cette optimisation a pour effet d'éliminer une partie du code reliée au prologue et à l'épilogue de l'appel.
- Le déroulement de boucle, qui diminue le temps d'exécution utilisé pour l'ajustement des indexes de la boucle et pour les branchements. Encore une fois, le prix à payer est une augmentation de la taille du code.
- L'optimisation avec fenêtre, qui remplace des séquences particulières de code par une séquence plus courte ou qui s'exécute plus rapidement.

7.2.2 Travaux futurs

Notre outil propose une alternative au flot de développement classique des systèmes embarqués. Même s'il automatise un grand nombre d'éléments, certaines étapes de la conception ne sont pas supportées par l'outil, notamment le partitionnement logiciel / matériel et la validation du système. Des travaux dans ces deux domaines seraient des avenues de recherche pour le futur.

Partitionnement logiciel / matériel

L'étape de partitionnement recherche l'architecture optimale du système. Dans notre outil, nous supposons que le partitionnement du système a déjà été effectué. Il serait toutefois intéressant d'intégrer des outils réalisant un partitionnement automatique ou, du moins, semi-automatique du système. Dans ce cas, l'outil fournirait des métriques pour un certain partitionnement et l'utilisateur choisirait ensuite le partitionnement désiré, tel que décrit dans [76]. Cependant, l'obtention de mesures significatives s'avère un processus difficile. Une mesure statique d'un partitionnement donné est généralement approximative, alors que le temps et la fréquence d'exécution des instructions sont approximées. Il est évident qu'une telle évaluation n'est pas exacte puisqu'elle ne représente pas le caractère dynamique du système. Une autre possibilité consiste à utiliser un système qui simule et évalue un partitionnement donné afin d'en retirer des valeurs réelles. De là, il est possible de comparer les valeurs de diverses simulations. Cette méthode demande néanmoins un cycle partitionnement-simulation-évaluation qui doit être répété plusieurs fois. Ce cycle est susceptible de demander beaucoup de temps.

Simulation et validation du système

Afin de répondre à plusieurs besoins, dont celui du partitionnement logiciel / matériel énoncé plus haut, il serait utile de valider le système une fois que le raffinement des spécifications logicielles et matérielles ont été réalisées. Ces spécifications pourraient être transformées en un modèle ESys.NET, qui validerait le système à l'aide du système de vérification semi-formel. Une intégration très proche du simulateur et de notre compilateur pourrait être obtenue, comme les deux outils sont réalisés à l'aide du cadre d'application .NET.

8 Références

[1] *"SystemVerilog"*, http://www.systemverilog.org/.

[2] *"Platform Studio Documentation"*, http://www.xilinx.com/ise/embedded/edk_docs.htm.

[3] *"GCC Home Page"*, http://gcc.gnu.org/.

[4] *"IFT2030"*, http://www.iro.umontreal.ca/~feeley/cours/ift2030/index.html.

[5] *"Wikipedia, The Free Encyclopedia"*, http://en.wikipedia.org/wiki/Main_Page.

[6] *"The ENIAC Online Museum"*, http://www.seas.upenn.edu/~museum/.

[7] *"SystemC"*, http://www.systemc.org/.

[8] *"Verilog"*, http://www.verilog.com/.

[9] *"Handel-C"*, http://www.celoxica.com/.

[10] *"DotGNU Project"*, http://www.dotgnu.org/.

[11] *"The SpecC System"*, http://www.ics.uci.edu/~specc/.

[12] *"Lutz Roeder's Programming.NET C# VB CLR WinFX"*, http://www.aisto.com/roeder/dotnet/.

[13] *"What is Behavioral Synthesis?"* http://www.forteds.com/behavioralsynthesis/index.asp.

[14] *"Mono"*, http://www.mono-project.com/Main_Page.

[15] *"Shared Source Common Language Infrastructure 1.0"*, msdn.microsoft.com/net/sscli/.

[16] *"Synopsys World Leader in EDA Software and Services"*, http://www.synopsys.com/.

[17] *"Mentor Graphics: The EDA Technology Leader"*, http://www.mentor.com/.

[18] *"Xilinx: The Programmable Logic Company"*, http://www.xilinx.com/.

[19] *"FPGAs, CPLDs, & Structured ASICs : Altera, the Leader in Programmable Logic"*, http://www.altera.com/.

[20] *"Tensilica - XPRES Compiler"*, http://www.tensilica.com/products/xpres.htm.

[21] *"Ptolemy"*, http://ptolemy.eecs.berkeley.edu/.

[22] *"Berkeley Reconfigurable Architectures, Systems, and Software"*, http://brass.cs.berkeley.edu/.

[23] *"Reverse Engineering To Learn .NET Better"*, http://www.blong.com/Conferences/DCon2003/ReverseEngineering/ReverseEngineering.htm.

[24] *"Java HotSpot Technology"*, http://java.sun.com/products/hotspot/.

[25] *"PGM Format Specification"*, http://netpbm.sourceforge.net/doc/pgm.html.

[26] *"Java Technology"*, http://java.sun.com/.

[27] *"Algorithmic State Machine"*, http://en.wikipedia.org/wiki/Algorithmic_State_Machine.

[28] *"Microblaze Processor Reference Guide"*, http://www.xilinx.com/ise/embedded/edk6_3docs/mb_ref_guide.pdf.

[29] *"Standard ECMA-335 : Common Language Infrastructure (CLI)"*, http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[30] Adl-Tabatabai, A.-R., et al., *Fast, effective code generation in a just-in-time Java compiler*, in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, ACM Press: Montreal, Quebec, Canada. p. 280-290.

[31] Aho, A., R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*. 1986: Addison-Wesley. 796.

[32] Ananian, C.S. and M. Rinard, *Data size optimizations for java programs*, in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. 2003, ACM Press: San Diego, California, USA. p. 59-68.

[33] Bacon, D.F. and P.F. Sweeney, *Fast static analysis of C++ virtual function calls*, in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 1996, ACM Press: San Jose, California, United States. p. 324-341.

[34] Bacon, D.F., *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*, in *Computer Science Division*. 1997, University of California: Berkeley. p. 148+xii pages.

[35] Bacon, D.F. and S.L. Graham, *Fast and effective optimization of statically typed object-oriented languages*. 1997. p. 141.

[36] Bacon, D.F., P. Cheng, and D. Grove, *Garbage collection for embedded systems*, in *Proceedings of the 4th ACM international conference on Embedded software*. 2004, ACM Press: Pisa, Italy. p. 125-136.

[37] Baleani, M., et al., *HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform*, in *Proceedings of the tenth international symposium on Hardware/software codesign*. 2002, ACM Press: Estes Park, Colorado. p. 151-156.

[38] Bergeron, E., et al., *High Level Synthesis for Data-Driven Applications*, in *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05) - Volume 00*. 2005, IEEE Computer Society. p. 54-60.

[39] Bilardi, G. and K. Pingali, *Algorithms for computing the static single assignment form*. J. ACM, 2003. **50**(3): p. 375-425.

[40] Buhr, P.A. and W.Y.R. Mok, *Advanced Exception Handling Mechanisms*. IEEE Trans. Softw. Eng., 2000. **26**(9): p. 820-836.

[41] Callahan, T.J., J.R. Hauser, and J. Wawrzynek, *The Garp Architecture and C Compiler*. Computer, 2000. **33**(4): p. 62-69.

[42] Cardoso, J.M.P. and H.C. Neto, *Compilation for FPGA-Based Reconfigurable Hardware*. IEEE Des. Test 2003. **20**(2): p. 65-75.

[43] Chaitin, G.J., *Register allocation & spilling via graph coloring*, in *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. 1982, ACM Press: Boston, Massachusetts, United States. p. 98-101.

[44] Compton, K. and S. Hauck, *Reconfigurable computing: a survey of systems and software*. ACM Comput. Surv., 2002. **34**(2): p. 171-210.

[45] Cytron, R., et al., *Efficiently computing static single assignment form and the control dependence graph*. ACM Trans. Program. Lang. Syst., 1991. **13**(4): p. 451-490.

[46] David, J.P. and E. Bergeron, *A Step towards Intelligent Translation from High-Level Design to RTL*, in *Proceedings of the System-on-Chip for Real-Time Applications, 4th IEEE International Workshop on (IWSOC'04) - Volume 00*. 2004, IEEE Computer Society. p. 183-188.

[47] Dean, J., D. Grove, and C. Chambers, *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*, in *Proceedings of the 9th European Conference on Object-Oriented Programming*. 1995, Springer-Verlag. p. 77-101.

[48] DeMicheli, G. and R.K. Gupta, *Hardware/software co-design*, in *Readings in hardware/software co-design*. 2002, Kluwer Academic Publishers. p. 30-44.

[49] Dennis, J.B. and D.P. Misunas, *A preliminary architecture for a basic data-flow processor*, in *Proceedings of the 2nd annual symposium on Computer architecture*. 1975, ACM Press. p. 126-132.

[50] Dhurjati, D., et al., *Memory safety without garbage collection for embedded applications*. Trans. on Embedded Computing Sys., 2005. **4**(1): p. 73-111.

[51] Ertl, M.A., *Implementation of Stack-Based Languages on Register Machines*. 1996, Technische Universitat Wien.

[52] Farach, M. and V. Liberatore, *On local register allocation*, in *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*. 1998, Society for Industrial and Applied Mathematics: San Francisco, California, United States. p. 564-573.

[53] Fischer, D., et al., *Efficient architecture/compiler co-exploration for ASIPs*, in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. 2002, ACM Press: Grenoble, France. p. 27-34.

[54] Frigo, J., M. Gokhale, and D. Lavenier, *Evaluation of the streams-C C-to-FPGA compiler: an applications perspective*, in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. 2001, ACM Press: Monterey, California, United States. p. 134-140.

[55] Gagnon, E.M. and L.J. Hendren, *SableCC, an Object-Oriented Compiler Framework*, in *Proceedings of the Technology of Object-Oriented Languages and Systems*. 1998, IEEE Computer Society. p. 140.

[56] Galloway, D., *The Transmogrifier C hardware description language and compiler for FPGAs*, in *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*. 1995, IEEE Computer Society. p. 136.

[57] George, L. and A.W. Appel, *Iterated register coalescing*. ACM Trans. Program. Lang. Syst., 1996. **18**(3): p. 300-324.

[58] Gokhale, M.B. and J.M. Stone, *NAPA C: Compiling for a Hybrid RISC/FPGA Architecture*, in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 1998, IEEE Computer Society. p. 126.

[59] Gough, K.J., *Stacking them up: a comparison of virtual machines*, in *Proceedings of the 6th Australasian conference on Computer systems architecture*. 2001, IEEE Computer Society: Queensland, Australia. p. 55-61.

[60] Gruian, F., et al., *Automatic generation of application-specific systems based on a micro-programmed Java core*, in *Proceedings of the 2005 ACM symposium on Applied computing*. 2005, ACM Press: Santa Fe, New Mexico. p. 879-884.

[61] Gupta, S., et al., *SPARK: A High-Lev l Synthesis Framework For Applying Parallelizing Compiler Transformations*, in *Proceedings of the 16th International Conference on VLSI Design*. 2003, IEEE Computer Society. p. 461.

[62] Hauser, J.R. and J. Wawrzynek, *Garp: a MIPS processor with a reconfigurable coprocessor*, in *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*. 1997, IEEE Computer Society. p. 12.

[63] Helaihel, R. and K. Olukotun, *Java as a specification language for hardware-software systems*, in *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*. 1997, IEEE Computer Society: San Jose, California, United States. p. 690-697.

[64] Jiang, S. and B. Xu, *An efficient and reliable object-oriented exception handling mechanism.* SIGPLAN Not., 2005. **40**(2): p. 27-32.

[65] Jones, A., et al., *PACT HDL: a compiler targeting ASICS and FPGAS with power and performance optimizations*, in *Power aware computing*. 2002, Kluwer Academic Publishers. p. 169-190.

[66] Jones, R. and R. Lins, *Garbage collection: algorithms for automatic dynamic memory management*. 1996: John Wiley \& Sons, Inc. 377 pages.

[67] Kent, K.B. and M. Serra, *Hardware/Software Co-Design of a Java Virtual Machine*, in *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*. 2000, IEEE Computer Society. p. 66.

[68] Keutzer, K., *Hardware-software co-design and ESDA*, in *Proceedings of the 31st annual conference on Design automation*. 1994, ACM Press: San Diego, California, United States. p. 435-436.

[69] Ku, D. and G. DeMicheli, *HardwareC -- A Language for Hardware Design (Version 2.0)*. 1990, Stanford University.

[70] Lavagno, L., G. Martin, and L. Scheffer, *Electronic Design Automation for Integrated Circuits Handbook*. 2005, CRC Press.

[71] Li, Y., et al., *Hardware-software co-design of embedded reconfigurable architectures*, in *Proceedings of the 37th conference on Design automation*. 2000, ACM Press: Los Angeles, California, United States. p. 507-512.

[72] Lysecky, R. and F. Vahid, *A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning*, in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*. 2005, IEEE Computer Society. p. 18-23.

[73] Manjunath, G. and V. Krishnan, *A small hybrid JIT for embedded systems.* SIGPLAN Not., 2000. **35**(4): p. 44-50.

[74] Mattos, J.C.B., et al., *Making object oriented efficient for embedded system applications*, in *Proceedings of the 18th annual symposium on Integrated circuits and system design*. 2005, ACM Press: Florianolpolis, Brazil. p. 104-109.

[75] Mesman, B., et al., *Embedded Systems Roadmap -- Vision on technology for the future of PROGRESS*. 2002: STW Technology Foundation/PROGRESS.

[76] Moya, J.M., et al., *Improving embedded system design by means of HW-SW compilation on reconfigurable coprocessors*, in *Proceedings of the 15th international symposium on System Synthesis*. 2002, ACM Press: Kyoto, Japan. p. 255-260.

[77] Muchnick, S.S., *Avanced Compiler Design and Implementation*. 1997: Morgan Kaufmann. 856.

[78] Nilsson, A., T. Ekman, and K. Nilsson, *Real Java for real time - gain and pain*, in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. 2002, ACM Press: Grenoble, France. p. 304-311.

[79] Poletto, M. and V. Sarkar, *Linear-scan register allocation.* ACM Trans. Program. Lang. Syst., 1999. **21**(5): p. 895-913.

[80] Robertson, I. and J. Irvine, *A design flow for partially reconfigurable hardware.* Trans. on Embedded Computing Sys., 2004. **3**(2): p. 257-283.

[81] Schultz, U.P., et al., *Compiling java for low-end embedded systems*, in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. 2003, ACM Press: San Diego, California, USA. p. 42-50.

[82] Shaylor, N., D.N. Simon, and W.R. Bush, *A java virtual machine architecture for very small devices*, in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. 2003, ACM Press: San Diego, California, USA. p. 34-41.

[83] Smith, M.D., N. Ramsey, and G. Holloway, *A generalized algorithm for graph-coloring register allocation*, in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. 2004, ACM Press: Washington DC, USA. p. 277-288.

[84] So, B., M.W. Hall, and P.C. Diniz, *A compiler approach to fast hardware design space exploration in FPGA-based systems*, in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, ACM Press: Berlin, Germany. p. 165-176.

[85] So, B., P.C. Diniz, and M.W. Hall, *Using estimates from behavioral synthesis tools in compiler-directed design space exploration*, in *Proceedings of the 40th conference on Design automation*. 2003, ACM Press: Anaheim, CA, USA. p. 514-519.

[86] Suganuma, T., T. Yasue, and T. Nakatani, *A region-based compilation technique for a Java just-in-time compiler*, in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 2003, ACM Press: San Diego, California, USA. p. 312-323.

[87] Traub, O., *Quality and Speed in Linear-Scan Register Allocation*, in *Computer Science*. 1998, Harvard College: Cambridge, Massachusetts.

[88] Traub, O., G. Holloway, and M.D. Smith, *Quality and speed in linear-scan register allocation*, in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, ACM Press: Montreal, Quebec, Canada. p. 142-151.

[89] Venkataramani, G., et al., *A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture*, in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. 2001, ACM Press: Atlanta, Georgia, USA. p. 116-125.

[90] Wahlen, O., et al., *Application specific compiler/architecture codesign: a case study*, in *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*. 2002, ACM Press: Berlin, Germany. p. 185-193.

[91] Ye, Z.A., N. Shenoy, and P. Baneijee, *A C compiler for a processor with a reconfigurable functional unit*, in *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*. 2000, ACM Press: Monterey, California, United States. p. 95-100.

[92] Yu, H., R. Doemer, and D. Gajski, *Embedded software generation from system level design languages*, in *Proceedings of the 2004 conference on Asia South Pacific design automation: electronic design and solution fair*. 2004, IEEE Press: Yokohama, Japan. p. 463-468.

ANNEXE 1

Le code de l'application utilisée dans la démonstration pas à pas des traitements est donné ici.

```
using System;

namespace ObjectSort
{
      class BaseType
      {
            protected string message;
            protected int base_num, exp_num;

            /// <summary>
            /// Constructor of the base type.
            /// </summary>
            /// <param name="i">The first value</param>
            /// <param name="j">The second value</param>
            public BaseType(int i, int j)
            {
                  message = "I am an object of type \"BaseType\"";
                  base_num = i;
                  exp_num = j;
            }

            /// <summary>
            /// Output method. Prints a string on the console.
            /// </summary>
            public void Output()
            {
                  Console.Write(message);
                  Console.Write(" with id ");
                  Console.WriteLine(
                   this.Identity(this.BaseNum, this.ExpNum));
            }

            public int BaseNum
            {
                  get { return base_num; }
            }

            public int ExpNum
            {
                  get { return exp_num; }
            }

            /// <summary>
            /// Compares the current object with another BaseType
            /// object.
            /// </summary>
            /// <param name="o">
            /// An object to compare the current object to.
            /// </param>
```

```
/// <returns>
/// 1 if the current object is greater.
/// -1 if the current object is lower.
/// 0 if equal.
/// </returns>
public int CompareTo(BaseType o)
{
      int o_base = o.BaseNum, o_xp = o.ExpNum;
      int this_base = this.BaseNum, this_xp = this.ExpNum;
      int o_id = o.Identity(o_base, o_xp);
      int this_id = Identity(this_base, this_xp);

      if(o_id > this_id)
            return 1;
      else if(o_id < this_id)
            return -1;
      return 0;
}

/// <summary>
/// The identity method. Computes a value based on the
/// parameters given. In this case, computes b ^ xp.
/// </summary>
/// <param name="b">The base</param>
/// <param name="xp">The exponent</param>
/// <returns>b^xp</returns>
[PartitionAttributes.Hardware]
public int Identity(int b, int xp)
{
      int num = 1;
      if(b == 0)
            return 0;
      else if(xp == 0)
            return 1;
      else
      {
            for(int i = 0; i < xp; i++)
            {
                  num *= b;
            }
            return num;
      }
}

/// <summary>
/// Main method.
/// </summary>
public static void Main()
{
      int length = 32;
      BaseType[] array = new BaseType[length];
      InitArray(array);
      Console.WriteLine("ARRAY BEFORE SORTING:");
      PrintArray(array);
      //Console.WriteLine("");
      SortArray(array);
      Console.WriteLine("ARRAY AFTER SORTING:");
```

```
        PrintArray(array);
}

/// <summary>
/// Initialize the array with "random" values.
/// </summary>
/// <param name="array">The array to initialize</param>
private static void InitArray(BaseType[] array)
{
        array[0] = new SubTypeA(2, 7);
        array[1] = new SubTypeB(3, 6);
        array[2] = new SubTypeB(1, 6);
        array[3] = new BaseType(-3, 5);
        array[4] = new SubTypeA(12, 2);
        array[5] = new SubTypeB(-4, 10);
        array[6] = new SubTypeA(3, 3);
        array[7] = new BaseType(6, 2);
        array[8] = new SubTypeB(12, 4);
        array[9] = new SubTypeB(-3, 7);
        array[10] = new BaseType(-17, 2);
        array[11] = new SubTypeB(4, 1);
        array[12] = new BaseType(0, 0);
        array[13] = new BaseType(-72, 3);
        array[14] = new BaseType(5, 6);
        array[15] = new SubTypeA(2, 12);
        array[16] = new SubTypeB(-11, 3);
        array[17] = new SubTypeB(0, 23);
        array[18] = new SubTypeA(43, 0);
        array[19] = new SubTypeB(5, 8);
        array[20] = new SubTypeA(-54, 1);
        array[21] = new BaseType(11, 5);
        array[22] = new SubTypeB(3, 4);
        array[23] = new BaseType(3, 9);
        array[24] = new BaseType(9, 3);
        array[25] = new SubTypeA(-3, 9);
        array[26] = new SubTypeB(1, 1);
        array[27] = new SubTypeA(-14, 2);
        array[28] = new SubTypeB(4, 4);
        array[29] = new BaseType(7, 4);
        array[30] = new SubTypeB(-3, 1);
        array[31] = new SubTypeB(-15, 3);
}

/// <summary>
/// Sorts the array, according to the value given by the
/// Identity method.
/// </summary>
/// <param name="array"></param>
private static void SortArray(BaseType[] array)
{
        for(int i = 0; i < array.Length; i++)
        {
                BaseType currentObj = array[i];
                for(int j = i + 1; j < array.Length; j++)
                {
                        BaseType candidateObj = array[j];
                        if(candidateObj.CompareTo(currentObj) > 0)
```

```
                        {
                              BaseType temp = currentObj;
                              array[i] = candidateObj;
                              array[j] = currentObj;
                              currentObj = candidateObj;
                        }
                  }
            }
      }

      /// <summary>
      /// Prints the contents an array.
      /// </summary>
      /// <param name="array">
      /// The array to print the contents
      /// </param>
      private static void PrintArray(BaseType[] array)
      {
            for(int i = 0; i < array.Length; i++)
            {
                  Console.Write(i);
                  Console.Write(": ");
                  array[i].Output();
            }
      }
}

class SubTypeA : BaseType
{
      /// <summary>
      /// Constructor of a subclass of BaseType
      /// </summary>
      /// <param name="i">The first value</param>
      /// <param name="j">The second value</param>
      public SubTypeA(int i, int j) : base(i,j)
      {
            message = "I am an object of type \"SubTypeA\"";
      }
}

class SubTypeB : BaseType
{
      /// <summary>
      /// Constructor of a subclass of BaseType
      /// </summary>
      /// <param name="i">The first value</param>
      /// <param name="j">The second value</param>
      public SubTypeB(int i, int j) : base(i,j)
      {
            message = "I am an object of type \"SubTypeB\"";
      }
}
}
```

ANNEXE 2

Le code CASM de l'ASM de la méthode `Identity` est donné ici.

```
signed input arg_fsl_data_in_0{protocol="fs"} [32];
signed output arg_fsl_data_out_0{protocol="fs"} [32];

asm Int32_IdentityInt32_Int32 {

signed register local_LOC_0[32];
      signed register local_LOC_1[32];
      signed register local_LOC_2[32];
      unsigned register local_ARG_0[32];
      signed register local_ARG_1[32];
      signed register local_ARG_2[32];
      signed register local_ARG_3[32];

FSL_INTERFACE_START:
      local_ARG_0 := (unsigned)arg_fsl_data_in_0;
      goto FSL_INTERFACE_START_0;

FSL_INTERFACE_START_0:
      local_ARG_1 := arg_fsl_data_in_0;
      goto FSL_INTERFACE_START_1;

FSL_INTERFACE_START_1:
      local_ARG_2 := arg_fsl_data_in_0;
      goto n8_0;

n1_0:
if( local_ARG_2 != 0 )
      goto n3_0;
else
      goto n2_0;
end;

n2_0:
      local_LOC_2 := (1);
      goto n7_0;

n3_0:
      local_LOC_1 := (0);
      goto n5_0;

n4_0:
      local_LOC_0 := (( local_LOC_0 * local_ARG_1 ));
      local_LOC_1 := (( local_LOC_1 + 1 ));
      goto n5_0;

n5_0:
if( local_LOC_1 < local_ARG_2 )
      goto n4_0;
else
      goto n6_0;
```

```
end;

n6_0:
      local_LOC_2 := (local_LOC_0);
      goto n7_0;

n7_0:
      local_ARG_3 := (local_LOC_2);
      goto FSL_INTERFACE_END;

n8_0:
      local_LOC_0 := (1);
      goto n8_1;

n8_1:
if( local_ARG_1 != 0 )
      goto n1_0;
else
      goto n0_0;
end;

n0_0:
      local_LOC_2 := (0);
      goto n7_0;

FSL_INTERFACE_END:
      arg_fsl_data_out_0 := (local_ARG_3);
      goto FSL_INTERFACE_START;
}
```

ANNEXE 3

Le code C de l'application originale de détection de contours est donné ici.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

#define N 400
#define M 640
#define IN_NAME "400_640_IN.pgm"
#define OUT_NAME "400_640_native.pgm"

int getint(FILE *fp) /* adapted from "xv" source code */
{
  int c, i, firstchar, garbage;

  /* note:  if it sees a '#' character, all characters from there to end
of
     line are appended to the comment string */

  /* skip forward to start of next number */
  c = getc(fp);
  while (1) {
    /* eat comments */
    if (c=='#') {
      /* if we're at a comment, read to end of line */
      char cmt[256], *sp;

      sp = cmt;  firstchar = 1;
      while (1) {
        c=getc(fp);
        if (firstchar && c == ' ') firstchar = 0;  /* lop off 1 sp after
# */
        else {
          if (c == '\n' || c == EOF) break;
          if ((sp-cmt)<250) *sp++ = c;
        }
      }
      *sp++ = '\n';
      *sp   = '\0';
    }

    if (c==EOF) return 0;
    if (c>='0' && c<='9') break;   /* we've found what we were looking
for */

    /* see if we are getting garbage (non-whitespace) */
    if (c!=' ' && c!='\t' && c!='\r' && c!='\n' && c!=',') garbage=1;

    c = getc(fp);
```

```
  }

  /* we're at the start of a number, continue until we hit a non-number
*/
  i = 0;
  while (1) {
    i = (i*10) + (c - '0');
    c = getc(fp);
    if (c==EOF) return i;
    if (c<'0' || c>'9') break;
  }
  return i;
}


void openfile(char *filename, FILE** finput)
{
  int x0, y0;
  char header[255];

  if ((*finput=fopen(filename,"rb"))==NULL) {
    fprintf(stderr,"Unable to open file %s for reading\n",filename);
    exit(-1);
  }

  fscanf(*finput,"%s",header);
  if (strcmp(header,"P2")!=0) {
    fprintf(stderr,"\nFile %s is not a valid ascii .pgm file (type
P2)\n",
            filename);
    exit(-1);
  }

  x0=getint(*finput);
  y0=getint(*finput);

  if ((x0!=N) || (y0!=M)) {
    fprintf(stderr,"Image dimensions do not match: %ix%i expected\n", N,
M);
    exit(-1);
  }
  getint(*finput); /* read and throw away the range info */
}


void read_image(char* filename, int image[M][N])
{
  long int inint;
  FILE* finput;
  int i,j;
  finput=NULL;
  openfile(filename,&finput);
  for (j=0; j<M; ++j)
    for (i=0; i<N; ++i) {
      if (fscanf(finput, "%i", &inint)==EOF) {
        fprintf(stderr,"Premature EOF\n");
        exit(-1);
```

```
      } else {
      //if (k++ % 2000 == 0)
        //printf("reading a pixel\n");
        image[j][i]= (int) inint;
      }
    }
  fclose(finput);
}


void write_image(char* filename, int image[M][N])
{
  FILE* foutput;
  int i,j;

  if ((foutput=fopen(filename,"wb"))==NULL) {
    fprintf(stderr,"Unable to open file %s for writing\n",filename);
    exit(-1);
  }

  fprintf(foutput,"P2\n");
  fprintf(foutput,"%d %d\n",N,M);
  fprintf(foutput,"%d\n",255);

  for (j=0; j<M; ++j) {
    for (i=0; i<N; ++i) {
      fprintf(foutput,"%3d ",image[j][i]);
      if (i%32==31) fprintf(foutput,"\n");
    }
    if (N%32!=0) fprintf(foutput,"\n");
  }
  fclose(foutput);
}

int max(int number, ...) {
  int maximum = 0;
  int value;
  int i;
  va_list argp;

  va_start(argp, number);
  for (i=0; i<number; i++)
    if ((value = va_arg(argp, int)) > maximum)
      maximum = value;
  va_end(argp);
  return maximum;
}

int ma1(int x, int y, int z) {
  return (68*x + 99*y + 68*z) / (2*68 + 99);
}

int ma2(int x, int y, int z, int l, int m, int o, int p, int q, int r) {
  return max(8, abs(x-r), abs(y-r), abs(z-r), abs(l-r), abs(m-r), abs(o-
r), abs(p-r), abs(q-r));
}
```

```
int ma3(int x) {
  return 255 - x;
}

int ma4(int x, int y, int z, int l, int m, int o, int p, int q, int r) {
  if (x > r || y > r || z > r || l > r || m > r || o > r || p > r || q >
r)
    return 0;
  return 255;
}


int IN[M][N];
int OUT[M][N];
int Tem[M][N];
int G_im[M][N];
int C_im[M][N];
int Tem2[M][N];

int main(int argc, char* argv[])
{

  int i, j;

  read_image(IN_NAME, IN);

  for(i = 0; i < M - 10; i = i + 10)
  {
        for(j = 0; j < N - 64; j = j + 64)
              printf("%3d\n", IN[i][j]);
  }

  for(i=0; i<M; i++)
    for(j=0; j<N; j++) {
      Tem[i][j] = 0;
      G_im[i][j] = 0;
      C_im[i][j] = 0;
      Tem2[i][j] = 0;
      OUT[i][j] = 0;
    }

  for (i=2; i<M-1; i++)
    for (j=1; j<N; j++)
      Tem[i][j] = ma1(IN[i-1][j], IN[i][j], IN[i+1][j]);

  for (i=2; i<M-1; i++)
    for (j=2; j<N-1; j++)
      G_im[i][j] = ma1(Tem[i][j-1], Tem[i][j], Tem[i][j+1]);

  for (i=3; i<M-2; i++)
    for (j=3; j<N-2; j++)
      C_im[i][j] = ma2(G_im[i+1][j+1], G_im[i+1][j], G_im[i+1][j-1],
G_im[i][j+1], G_im[i][j-1],
                  G_im[i-1][j+1], G_im[i-1][j], G_im[i-1][j-1],
G_im[i][j]);

  for (i=3; i<M-2; i++)
```

```
    for (j=3; j<N-2; j++)
      Tem2[i][j] = ma3(C_im[i][j]);

  for (i=4; i<M-3; i++)
    for (j=4; j<N-3; j++)
      OUT[i][j] = ma4(Tem2[i+1][j+1], Tem2[i+1][j], Tem2[i+1][j-1],
Tem2[i][j+1], Tem2[i][j-1],
                  Tem2[i-1][j+1], Tem2[i-1][j], Tem2[i-1][j-1],
Tem2[i][j]);

  write_image(OUT_NAME, OUT);
}
```

ANNEXE 4

Le code C# de l'application de détection de contour est donné ici.

```
using System;
using System.IO;

namespace ContourDetection
{
      /// <summary>
      /// Contour detection algorithm.
      /// The values of the image points are hardcoded.
      /// </summary>
      public class ContourDetection
      {
            // Dimensions of the image
            private int M, N;


            public ContourDetection(int n, int m)
            {
                  M = m;
                  N = n;
            }

            public int[] ReadImage()
            {
                  int[] image = new int[getM * getN];

                  /************************************
                  * The values of the image points are hardcoded
                  * in this method. They are not shown here for
                  * better readability.
                  ************************************/
                  return image;
            }

            public void WriteImage(int[] image)
            {
                  for(int i = 0; i < (getM * getN); i++)
                  {
                        if(i % 8 == 7)
                        {
                              PrintLine(image[i]);
                        }
                        else
                        {
                              Print(image[i]);
                        }
                  }
            }

            private void Print(int num)
            {
```

```
        Console.Write(" ");
        Console.Write(num);
}

private void PrintLine(int num)
{
        Console.Write(" ");
        Console.WriteLine(num);
}

public int ma1(int x, int y, int z)
{
        return (68*x + 99*y + 68*z) * (2*68 + 99);
}

public int ma2(int x, int y, int z, int l, int m, int o, int
        p, int q, int r)
{
        return max(Math.Abs(x-r), Math.Abs(y-r),
        Math.Abs(z-r), Math.Abs(l-r), Math.Abs(m-r),
        Math.Abs(o-r), Math.Abs(p-r), Math.Abs(q-r));
}

public int ma3(int x)
{
        return 255 - x;
}

[PartitionAttributes.Hardware]
public int ma4(int x, int y, int z, int l, int m, int o, int
        p, int q, int r)
{
        if (x > r || y > r || z > r || l > r || m > r ||
                o > r || p > r || q > r)
                return 0;
        return 255;
}

private int max(int a, int b, int c, int d, int e, int f,
        int g, int h)
{
        int v_max = a;
        if (v_max < b) v_max = b;
        if (v_max < c) v_max = c;
        if (v_max < d) v_max = d;
        if (v_max < e) v_max = e;
        if (v_max < f) v_max = f;
        if (v_max < g) v_max = g;
        if (v_max < h) v_max = h;
        return v_max;
}

public int getElem(int i, int j)
{
        return (i * this.getN) + j;
}
```

```
public int getM
{
      get { return M; }
}

public int getN
{
      get { return N; }
}

public static void Main()
{
      Console.WriteLine("Entering main...");
      ContourDetection detect = new ContourDetection(8, 8);

      int m = detect.getM;
      int n = detect.getN;

      Console.WriteLine("Reading image...");
      int[] IN = detect.ReadImage();

      int[] Tem = new int[detect.getM * detect.getN];
      int[] G_im = new int[detect.getM * detect.getN];
      int[] C_im = new int[detect.getM * detect.getN];
      int[] Tem2 = new int[detect.getM * detect.getN];
      int[] OUT = new int[detect.getM * detect.getN];

      // Premiere boucle
      for (int i=2; i<m-1; i++)
      {
            for (int j=1; j<n; j++)
            {
                  Tem[detect.getElem(i,j)] = detect.ma1(
                        IN[detect.getElem(i-1,j)],
                        IN[detect.getElem(i,j)],
                        IN[detect.getElem(i+1,j)]);
            }
      }

      // Deuxieme boucle
      for (int i=2; i<m-1; i++)
      {
            for (int j=2; j<n-1; j++)
            {
                  G_im[detect.getElem(i,j)] = detect.ma1(
                        Tem[detect.getElem(i,j-1)],
                        Tem[detect.getElem(i,j)],
                        Tem[detect.getElem(i,j+1)]);
            }
      }

      // Troisieme boucle
      for (int i=3; i<m-2; i++)
      {
            for (int j=3; j<n-2; j++)
            {
                  C_im[detect.getElem(i,j)] = detect.ma2(
```

```
                                    G_im[detect.getElem(i+1,j+1)],
                                    G_im[detect.getElem(i+1,j)],
                                    G_im[detect.getElem(i+1,j-1)],
                                    G_im[detect.getElem(i,j+1)],
                                    G_im[detect.getElem(i,j-1)],
                                    G_im[detect.getElem(i-1,j+1)],
                                    G_im[detect.getElem(i-1,j)],
                                    G_im[detect.getElem(i-1,j-1)],
                                    G_im[detect.getElem(i,j)]);
                        }
                  }

                  // Quatrieme boucle
                  for (int i=3; i<m-2; i++)
                  {
                        for (int j=3; j<n-2; j++)
                        {
                              Tem2[detect.getElem(i,j)] =
                              detect.ma3(C_im[detect.getElem(i,j)]);
                        }
                  }

                  // Cinquieme boucle
                  for (int i=4; i<m-3; i++)
                  {
                        for (int j=4; j<n-3; j++)
                        {
                              OUT[detect.getElem(i,j)] = detect.ma4(
                                    Tem2[detect.getElem(i+1,j+1)],
                                    Tem2[detect.getElem(i+1,j)],
                                    Tem2[detect.getElem(i+1,j-1)],
                                    Tem2[detect.getElem(i,j+1)],
                                    Tem2[detect.getElem(i,j-1)],
                                    Tem2[detect.getElem(i-1,j+1)],
                                    Tem2[detect.getElem(i-1,j)],
                                    Tem2[detect.getElem(i-1,j-1)],
                                    Tem2[detect.getElem(i,j)]);
                        }
                  }

                  Console.WriteLine("Writing image...");
                  detect.WriteImage(C_im);
            }
      }
}
```