République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieure et de la Recherche Scientifique

Université Mentouri – Constantine

Faculté des Sciences de l'Ingénieur

Département d'Informatique

N° d'ordre : 80 / TE / 2008
Série: 06 / Inf / 2008

Thèse

En vue de l'obtention du titre de
Docteur d'état en Informatique

Par
Salah MERNIZ

Thème

Méthodologie de Vérification Formelle Pour les Microarchitectures RISC
Approche Fonctionnelle

Devant le jury composé de :

Président	Pr. Mahmoud Boufaida,	Université Mentouri – Constantine
Rapporteur	Pr. Mohamed Benmohammed,	Université Mentouri – Constantine
Examinateurs	Pr. Mustapha Lalam,	Université Mouloud Mammeri – Tizi Ouzou
	Pr. Mouloud Koudil,	INI – Alger
	Dr. Allaoua Chaoui,	Université Mentouri – Constantine

A mes trois enfants:

Mohamed El Amine

Lina

Amani Imane

Remerciements

Je tiens à remercier vivement Monsieur Mohamed BENMOHAMMED, Professeur à l'Université de Constantine qui a assuré la direction de ma thèse, et en particulier pour ses conseils constructifs prodigués durant toute la période d'élaboration de ce travail.

Que Messieurs Louis MACKENZIE et Mohamed OULD KHAOUA, Professeurs à l'Université de Glasgow, trouvent ici l'expression de ma profonde gratitude pour les aides précieuses qu'ils ont apportées à ce travail.

Mes sincères remerciements vont également à Monsieur Mahmoud BOUFAIDA Professeur à l'Université de Constantine pour l'honneur qu'il m'a fait en acceptant de présider le jury.

Mes vifs remerciements également à Messieurs : Mouloud KOUDIL, Professeur à l'INI, et Mustapha LALAM, Professeur à l'Université de Tizi Ouzou, pour l'honneur qu'ils m'ont fait en acceptant de participer au jury.

Je tiens à remercier vivement également, Monsieur Allaoua CHAOUI, Maître de conférence à l'Université de Constantine, d'abord pour le suivi administratif de cette thèse en tant que s/directeur chargé de la post graduation, et ensuite pour l'honneur qu'il m'a fait en acceptant de participer au jury.

Je n'oublie pas également de remercier Monsieur Salim CHIKHI Maître de conférence à l'Université de Constantine pour avoir accepté d'assurer mes tâches pédagogiques durant mes stages, et pour l'aide qu'il a apporté au tirage de cette thèse.

Je dois enfin beaucoup et plus à mes parents, à mes frères et à ma sœur, pour leur soutien et leur patience, qu'ils trouvent ici le témoignage de ma sincère gratitude.

[illegible]

نقترح طريقة منهجية لمواصفة وتحقيق قاطع لهندسة المعالج في وسط وظيفي. الطريقة المنهجية المتبعة تقولب هندسة المعالج كآلة حالة وتثبيت صحتها بعد تقسيم الحالة الإجمالية بصفة نظامية . من أجل قابلية للتسلق بصفة أحسن [illegible] ذات تعقيدات متزايدة: تسلسلية أنابيبية ومدرجة.

بالنسبة للهندسة التسلسلية مقياس التصحيح يرتكز أساسا على الرسم الابتدائي الاعتيادي الذي يبين بوضوح التكافؤ الوظيفي للمواصفة والتصميم المتطابق باستعمال دالة وسطية للمعطيات ملائمة. بإتباع طريقة عشيرية المنهجية [illegible] متابعة الاستكشاف من المستوى الذي فشل فيه البرهان.

بالنسبة للهندسة الأنابيبية والمدرجة طريقة الرسم الإبتدائي الإعتيادي لا يمكن تطبيقها مباشرة بسبب استتار حوادث الانابيب من أجل ذلك الطريقة المقترحة تصحح التصميمة الأنابيبية إزاء التصميمية التسلسلية المتزامنة لأن [illegible] تمثل نقاط ملائمة للمقارنة بين النموذجين بطريقة سهلة كذلك لأن النموذجان ينتميان إلى مستوى المكروهندسي المنهجية لا تحتاج إلى أي دالة وسطية للمعطيات تحتاج فقط إلى دالة زمنية للتوقيت بين النموذجين هذا الإختيار الرائد يسمح بتحقيق البرهان باستقراء على مستوى لغة المواصفة بدل من الإستظهار [illegible].

الميزات القوية للمنهجية المقترحة تأكدت من خلال التطبيقات على المعالج MIPS في وسط وظيفي .

Résumé

Nous proposons une méthodologie de preuve pour la spécification et la vérification formelles des microarchitectures de processeurs RISC dans un environnement fonctionnel. La méthodologie modélise des microarchitectures comme machines d'état et prouve leur correction après décomposition de l'état global, d'une manière systématique. Pour une meilleure scalabilité, La méthodologie proposée utilise une approche incrémentale pour modéliser et vérifier des microarchitectures de complexité croissante : séquentielle, pipeline et superscalaire.

Pour les microarchitectures séquentielles, le critère de correction est fondé essentiellement sur le diagramme commutatif standard. Ce dernier permet de démontrer l'équivalence fonctionnelle de la spécification du jeu d'instructions (Instruction-Set-Architecture : ISA) et de l'implémentation micro architecturale (MA) correspondante, en utilisant une fonction d'abstraction appropriée. En adoptant une approche hiérarchique, notre méthodologie raffine la preuve jusqu'au niveau composant observable. Un tel raffinement permet au processus de débogage de continuer l'exploration à partir du niveau où le processus de preuve échoue.

Pour les microarchitectures pipeline et superscalaire, l'approche du diagramme commutatif ne peut pas être appliquée directement à cause de la latence des événements du pipeline. Pour cette raison, notre approche vérifie une implémentation pipeline vis-à-vis d'une implémentation multi cycle séquentielle. Puisque les deux modèles (séquentiel et pipeline) sont formalisés en termes de cycles d'horloge, tous les états intermédiaires synchrones représentent des points utiles où la comparaison entre les deux modèles pourrait être réalisée facilement. Aussi, puisque les deux modèles appartiennent au niveau microarchitecture, la méthodologie n'exige aucune fonction d'abstraction de données (qui reste extrêmement difficile à définir pour la majorité des approches), uniquement une fonction d'abstraction de temps est demandée pour synchroniser les deux modèles. Un avantage majeur de ce choix élégant est l'habilité d'effectuer la preuve par induction au sein du même langage de spécification, plutôt que par la simulation symbolique à travers un outil de preuve qui reste très difficile à manipuler .

Les caractéristiques potentielles de la méthodologie proposée ont été démontrées sur les processeurs MIPS dans un environnement fonctionnel.

Abstract

We propose a proof methodology for the formal specification and verification of RISC processor micro-architectures within a functional framework. The methodology models micro-architectural designs as state machines and proves their correctness after decomposing the global state, in a systematic way. For a better scalability, the proposed methodology uses an incremental approach for modelling and verifying micro-architectures of increasing complexity: sequential, pipeline and superscalar.

For sequential designs, usually the correctness criterion relies on the standard commutative diagram that demonstrates the equivalence of an Instruction-Set-Architecture (ISA) specification and its corresponding Micro-Architectural (MA) implementation using an appropriate abstraction function to map the implementation state to the specification state. By adopting a hierarchical approach, our methodology refines the proof till the component observable level. Such refinement allows the debugging process to proceed from where the proof process fails.

For pipelined and superscalar micro-architectural designs, the commutative diagram approach cannot be applied directly because of the latency of pipeline events. For that reason, our approach verifies a pipelined implementation against a sequential multi-cycle implementation rather than against an ISA specification. Because both pipelined and multi-cycle models are formalized in terms of clock cycles, all synchronous intermediate states represent useful points where the comparison between the two models could be achieved easily. Furthermore, because both models relate to the MA level, there is no need for a data abstraction function which remains very difficult to define in practice, only a time abstraction function is needed to map between the times used by the two models. A major advantage of this elegant choice is the ability to carry out the proof by induction within the same specification language, rather than by symbolic simulation through of a proof tool which remains very tedious.

The potential features of the proposed Methodology have been demonstrated over MIPS RISC processors within a functional framework.

Table des matières

Liste des figures

Liste des tables

Chapitre 1

Introduction

1.1 Motivations

Les microarchitectures des processeurs d'aujourd'hui sont très complexes. Toutes les caractéristiques d'optimisation des performances telles que les caches, les techniques de pipeline, d'exécution spéculative et désordonnée, sont exposées au niveau micro-architecture. En conséquence, les méthodes de simulation conventionnelles (numérique, vecteur de test, etc.), deviennent insuffisantes pour faire face à cette complexité qui, selon la loi de Moore, double presque chaque deux années. Figure 1.1 illustre ce rythme d'évolution pour la famille des microprocesseurs Intel.

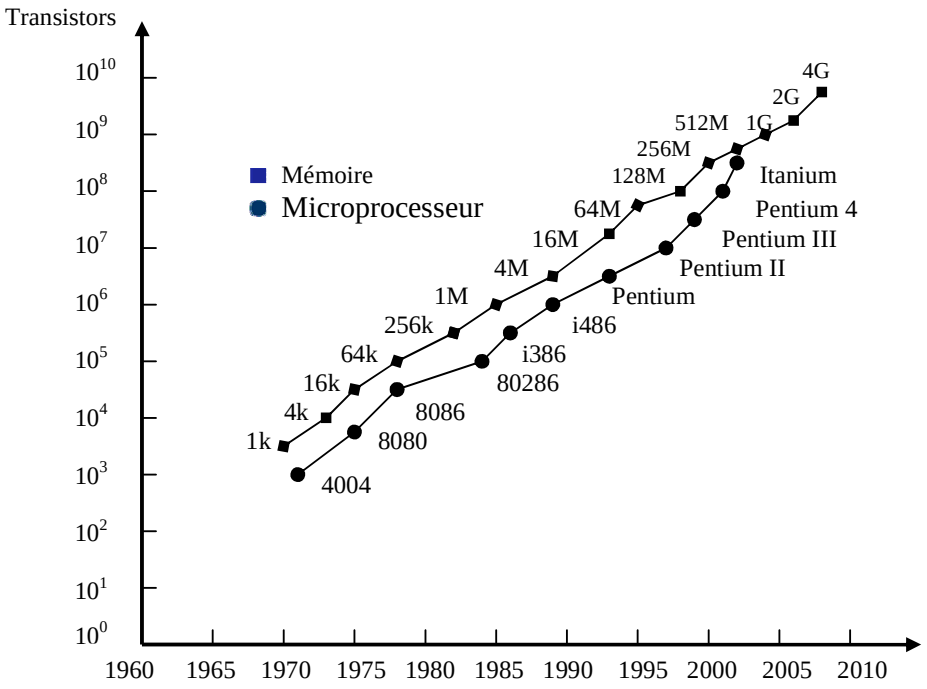


Figure 1.1: Evolution de la complexité des microprocesseurs selon la loi de Moore (Source Intel)

Devant une telle situation, la simulation qui exige un temps de test considérable (presque 70% du temps de conception sont consacrés aux tests de validation [1, 2]), ne peut en contre partie, que réduire le nombre de bugs de la conception (65 répertoriés dans le Pentium III [3], 6 dans l'AMD Athlon [4], plusieurs erreurs dans les révisions 2.2 à 2.6 du PowerPC 7400 [5], etc), mais jamais certifier sa correction. Un bug non révélé revient très cher au concepteur (Intel Pentium: $475 Millions et Toshiba Contrôleur: $2.1 Billions [6]), et peut avoir des conséquences néfastes sur l'utilisateur, particulièrement dans les applications critiques (aérospatiales, centres nucléaires, appareils chirurgicaux, etc.). En plus, le temps de livraison (time-to-market) sera aussi retardé. De telles contraintes ont poussé les concepteurs de circuits intégrés à investir les méthodes de vérification formelles qui deviennent rapidement un sujet d'intérêt majeur.

En effet, les méthodes formelles ont eu un impact signifiant sur la conception des microprocesseurs, en développant des méthodologies et des outils qui ont été utilisés avec succès dans la vérification formelle des architectures des microprocesseurs [7, 8, 9, 10]. Cependant, l'évolution continue de la complexité d'une part et le manque de méthodologies efficaces d'autre part, ont rendu la vérification formelle complète des microprocesseurs hautement optimisés une tache très difficile, voir impossible à accomplir [11]. Encore, il est extrêmement difficile de dériver formellement une implémentation microarchitecturale (MA) employant des optimisations radicales à partir de la spécification du jeu d'instructions (ISA). Plutôt, une implémentation MA est graduellement développée au sein d'un environnement approprié (logique d'ordre supérieur [12], logique de réécriture [13], programmation fonctionnelle [14], etc.), et prouvée contre une spécification ISA, selon un certain critère de correction.

Pour les microarchitectures séquentielles, le critère de correction est fondé principalement sur le diagramme commutatif standard. Ce dernier permet de montrer l'équivalence fonctionnelle de la spécification et de l'implémentation, en définissent une fonction d'abstraction appropriée. Pour les microarchitectures avec pipeline, le diagramme commutatif ne peut pas être appliqué directement à cause de l'exécution partielle des instructions se trouvant dans le pipe (le processeur démarre l'instruction suivante avant la terminaison de l'instruction précédente). Ce phénomène rendra difficile la définition d'une fonction d'abstraction qui transforme les résultats partiels à un état visible. En d'autres mots, il est impossible de trouver un point constructif où la comparaison entre l'état de l'implémentation et l'état de la spécification peut être effectué facilement. Burch et Dill résolvent ce problème en vidant le pipe. Dans leur

travail original, ils ont prouvé le diagramme de correction du pipeline en simulant symboliquement la machine pipeline par la logique de fonctions non interprétées avec des égalités (logic of uninterpreted functions with equalities [7]).

Bien que la technique de vidage (flushing technique) a amélioré les méthodes de vérification par l'utilisation d'une procédure de décision automatique, elle présente d'autre part, plusieurs inconvénients [15, 16]. Particulièrement, elle enlarge la fonction d'abstraction pour des pipelines plus profonds, et rends le raisonnement sur la dépendance inter instruction impossible. La technique a été étendue par la suite par plusieurs chercheurs pour manipuler des architectures plus complexes telles que les superscalaires à exécution ordonnée [17, 18], et à exécution désordonnée [19, 20]. Malheureusement, le même critère de correction (prouver l'implémentation contre une spécification ISA) a été adopté par les extendeurs et par conséquent, les mêmes inconvénients persistent. En outre, des que de nouvelles caractéristiques d'implémentation sont introduites, ces variantes échouent. D'autres notions de correction telles que le théorème à une seule étape (the one step theorem [21, 22]), l'approche de fonction de complétude (the completion function approach [15]), et la bisimulation d'équivalence bien-fondée (Well-founded Equivalence Bisimulation [23, 24]), ont été aussi utilisées pour vérifier des microprocesseurs complexes. Toutes ces approches prouvent l'implémentation contre une spécification ISA, qui reste encore un problème difficilement surmontable au fur et à mesure que la complexité augmente.

A part quelques exemples, la plupart des approches utilisent la simulation symbolique ou des variantes de cette technique (telle que l'évaluation de la trajectoire symbolique) pour effectuer la preuve de correction, à travers un démonstrateur de théorème ou un vérificateur de modèle (ou une combinaison des deux outils). Les deux approches présentent des inconvénients : les démonstrateurs de théorème souffrent encore du manque de raisonnement formel complet, alors que les vérificateurs de modèles souffrent du problème de l'explosion d'état. Un autre inconvénient majeur commun aux deux techniques, est le problème de translation de l'environnement de spécification vers l'environnement de vérification. Si le translateur n'est pas formellement vérifié, il n'y a aucune assurance à ce que le processus de translation préserve la sémantique de la spécification source.

Cette dissertation propose une méthodologie pour la modélisation et la vérification des microarchitectures des processeurs au sein d'un environnement fonctionnel. La méthodologie utilise une approche incrémentale pour modéliser des microarchitectures

de complexité croissante: du noyau séquentiel au superscalaire, et prouve leur correction après décomposition de l'espace d'état d'une manière systématique.

Pour les architectures séquentielles qui sont vérifiées contre une spécification ISA, la méthodologie décompose la preuve jusqu'au niveau composant observable. Cette décomposition hiérarchique facilite le processus de preuve d'une part, et permet au processus de conception de procéder d'une manière incrémentale d'autre part. Pour les architectures pipelines et superscalaires, la méthodologie effectue la preuve plutôt contre une implémentation multi cycle séquentielle. Puisque les deux modèles (séquentiel et pipeline) sont formalisés en termes de cycles d'horloge, tous les états intermédiaires synchrones représentent des points intéressants où la comparaison peut être effectuée facilement. En outre, puisque les deux modèles sont reliés au niveau microarchitecture, la méthodologie n'exige aucune fonction d'abstraction de données (qui reste extrêmement difficile à définir pour la majorité des approches), uniquement une fonction d'abstraction de temps est demandée pour synchroniser les deux modèles. Un avantage majeur de ce choix élégant, est l'habilité d'effectuer la preuve par induction au sein du même langage de spécification plutôt que par simulation symbolique à travers un outil de preuve qui reste encore difficile a manipuler.

Pour montrer pratiquement l'utilité de notre approche, nous l'avons appliqué aux processeurs RISC dans un cadre fonctionnel. Les architectures RISC sont bien structurées et par suite, elles peuvent être hiérarchiquement construites, du noyau implémentant le jeu d'instruction de base, aux architectures hautement optimisées [25]. Par conséquent, elles s'adaptent très bien à l'approche de conception incrémentale. D'autre part, les environnements fonctionnels fournissent à côté de la définition formelle de leur sémantique (pour supporter le raisonnement formel), des caractéristiques puissantes: composition de fonctions, parallélisme, polymorphisme, fonctions d'ordre supérieur, etc, qui ont démontré leur viabilité vis à vis des architectures complexes [26, 27].

1.2 Etat de l'art

1.2.1 Techniques de vérification conventionnelles

Les techniques de vérification formelles permettent d'effectuer des preuves sur des descriptions d'une manière mathématique et exhaustive. Cependant, Il n'existe pas de technique capable de vérifier toutes les formes de propriétés d'une description. Les systèmes réels sont souvent validés par des approches hybrides combinant deux ou

plusieurs techniques qui sont employées pour détecter les erreurs de conception qui échappent à la phase de simulation. Les techniques formelles conventionnelles sont sommairement décrites ci-après

1.2.1.1 Vérification de modèle (Model-checking)

Basée sur les Diagrammes de Décision Binaires (Binary Decision Diagrams [28]), cette technique permet de balayer l'ensemble des états atteignables d'un system dont le comportement est modélisé par les systèmes a transition d'état, pour vérifier une propriété logico temporelle désirée. Des situations telles que la consistance de données ou la vivacité, sont souvent les cibles de ce type de vérification. Cette technique trouve ses racines dans les premiers travaux sur la vérification de modèle temporel proposée par Clarke et Emerson [29]. Bien quelle permet d'effectuer des preuves d'une manière automatique, l'inconvénient majeure de cette approche est le problème de l'explosion combinatoire: l'espace d'état des descriptions des systèmes complexes est typiquement trop grand pour permettre une recherche exhaustive. En 1986, Bryant [30] proposait une nouvelle représentation des fonctions booléennes, appelée OBDDs (Ordered Binary Decision Diagrams) pour représenter symboliquement l'espace des états dans le modèle de la machine â états finis. Il a présenté aussi une procédure efficace de point fixe qui a rendu possible la vérification de circuits de taille importante. En 1987, il développa le vérificateur de modèle symbolique SMV (Symbolic Model Verifier) dans lequel, des systèmes contenant 10^{20} états pouvaient être vérifiés. La technique de vérificateur de modèle symbolique était née. Actuellement, de nombreux travaux de recherche tendent à améliorer cette technique afin d'y incorporer de nouvelles techniques d'abstraction

Les vérificateurs de modèles symboliques sont largement utilisés dans les industries des semi-conducteurs, SMV étant l'un des plus utilisés. Plusieurs compagnies ont intégrés les vérificateurs de modèles symboliques dans leurs propres outils de conception propriétaires. Par exemple, RuleBase développé à IBM Haifa Research Lab, a été utilisé dans des projets consistant à vérifier des protocoles de bus.. Le vérificateur de modèle SVE développé à Siemens a été appliqué dans de nombreux projets de développements internes, et a été aussi commercialisé à des clients externes.

1.2.1.2 Démonstration de théorème (Theorem proving)

Contrairement aux vérificateurs de modèles qui s'appliquent uniquement aux propriétés individuelles des systèmes, les démonstrateurs de théorèmes peuvent être utilisés pour vérifier la correction d'une conception complète. La vérification d'un circuit avec ces démonstrateurs, consiste à formaliser d'abord les descriptions comportementale et structurale du circuit dans la logique du démonstrateur. Cette phase de transformation de l'environnement de spécification à l'environnement de preuve est une phase cruciale. Ensuite, raisonner sur le modèle construit pour vérifier les propriétés du circuit. Le raisonnement qui utilise des logiques telles que l'induction et la réécriture permet de prouver qu'un théorème est dérivé des axiomes. La preuve souvent nécessite le rajout d'une quantité importante de théorèmes et de lemmes stockés en bibliothèque. L'utilisation des démonstrateurs de théorèmes exige en plus d'une expertise dans la logique de ces derniers, des interventions extensives de la part de l'utilisateur dont les difficultés augmentent avec l'augmentation de la complexité du système sous preuve. En conséquence, seuls les gros industriels tels qu'Intel, AMD et IBM, sont capables d'utiliser ces types de vérificateurs. Plusieurs démonstrateurs de théorèmes existent, les plus utilisés dans le monde industriel sont les suivants:

- PVS (Prototype Verification System) [31], est un démonstrateur de théorème inductif, développé à SRI. PVS combine un langage de spécification expressif basé sur une logique d'ordre supérieur et un moteur d'inférence pour ce langage. Bien qu'il dispose de nombreuses règles de déduction, la démonstration n'est pas complètement automatique, et doit être guidée par l'utilisateur. Il a été utilisé pour la vérification de plusieurs microprocesseurs [32, 33].
- ACL2 [34] (A Computational Logic for Applicative Common Lisp) est un démonstrateur de théorèmes inductif développé a l'Université de Texas Austin par J.Moore et M. Kaufmann. Il est l'héritier du célèbre démonstrateur de théorème de Boyer-Moore. Acl2 est à la fois un langage de programmation et un moteur de raisonnement pour ce langage Le modèle ACL2 peut être exécuté à une vitesse proche d'un simulateur C, et il dispose d'une vaste bibliothèque de théorèmes réutilisables (arithmétiques, simplifications,etc.). Il a été utilisé pour la vérification de plusieurs microprocesseurs [35, 36]

1.2.1.3 Simulation symbolique

La simulation symbolique [37] consiste à évaluer symboliquement une description de circuit en lui fournissant en entrée des variables abstraites à la place des valeurs concrètes (cela permet de representer par exemple toures les valeurs d'un vecteur de bits en entrée par un simple symbole). Le résultat d'exécution de la description est une expression symbolique représentant le circuit. L'évaluation symbolique peut être effectuée soit en utilisant des constantes non interprétés, soit des constructeurs de données fonctionnels. Dans ce dernier cas, l'évaluation symbolique est dite fonctionnelle (Symbolic Functional Evaluation: SFE)

La simulation symbolique fut proposée pour la première fois par IBM, vers la fin des années soixante-dix. L'idée est d'utiliser l'exécution symbolique des programmes pour des circuits. Les difficultés de cette technique sont les suivantes:

- Les expressions symboliques croissent exponentiellement avec le nombre de cycles de simulation.
- En présence de branchements conditionnels où la condition est un terme symbolique, tous les chemins possibles doivent être explorés. Ceci produit un arbre de simulation qui croît de manière exponentielle.
- Sans simplification, les résultats de la simulation symbolique sont des termes illisibles.

La raison principale de ces difficultés est l'absence d'un modèle mathématique efficace pour la représentation symbolique du circuit.

La simulation symbolique a été utilisée pour la vérification de plusieurs microprocesseurs [38, 39].

1.2.1.4 Evaluation de la trajectoire symbolique

La technique de l'évaluation de la trajectoire symbolique (Symbolic Trajectory Evaluation: STE) [40] peut être considérée comme une approche hybride combinant les algorithmes de simulation symbolique et de vérification de modèle. En tant que simulateur, elle peut calculer les résultats d'exécution d'un circuit acceptant en entrée un vecteur de valeurs concrètes. En tant que simulateur symbolique, elle peut rendre comme résultat des expressions symboliques (des fonctions avec un nombre d'entrées arbitraire). En tant que vérificateur de modèles, elle peut facilement vérifier la validité d'une propriété temporelle. La technique de l'évaluation de la trajectoire symbolique effectue la vérification de modèle avec des algorithmes basés sur la simulation

symbolique d'une manière plus efficace que les approches de vérificateur de modèles symbolique traditionnelles. Elle est particulièrement adéquate pour la vérification des propriétés des chemins de données. Cependant, comme l'évaluation de la trajectoire symbolique est basée sur les BDDs, certaines structures de circuits ne peuvent pas être manipulées monolithiquement. Par exemple, les multiplicateurs ne peuvent pas être vérifies par la STE toute seule.

La technique de l'évaluation de la trajectoire symbolique a été utilisée dans la vérification de plusieurs parties de microprocesseurs : [41, 42],

1.2.2 Application des Techniques conventionnelles

Initialement, l'application des méthodes de vérification formelle s'est axée essentiellement sur des processeurs séquentiels. Des exemples notables incluent: VIPÈRE [43], vérifié par Cohn, FM8501 [44], vérifié par Hint, (avec le démonstrateur de théorèmes de Boyer-Moore, version initiale du démonstrateur de théorèmes ACL2), TAMARAK [45], vérifié par Joyce, et AVM-1 [46], vérifié par Windely. La machine RISC DLX qui est décrite par Hennessy et Patterson [47] a servi comme base à beaucoup de projets de vérification de processeurs.

Les travaux relatifs à la vérification formelle des processeurs utilisant des caractéristiques avancées sont récapitulés ci-dessous.

Burch et Dill [7] proposent l'approche de vidage du pipe (flushing approach) qui simule l'effet de compléter chaque instruction dans la pipe avant d'appliquer le diagramme commutatif. Après vidage, ils projettent l'état d'implémentation synchronisé à l'état de spécification pour extraire seulement les observables. Dans leur travail original, ils ont prouvé le diagramme de correction du pipeline, en simulant symboliquement l'architecture de la machine pipeline dans leur logique de fonctions non interprétées avec des égalités. Bien que la méthode de vidage ait amélioré des techniques de vérification en employant une procédure de décision automatisée, elle rend en contre partie la taille de la fonction d'abstraction et le nombre de cas examinés très importants pour des pipes plus profonds. Aussi, en vidant le pipe, l'information concernant les instructions en cours d'exécution sera perdue, ce qui rendra difficile à raisonner sur la dépendance inter instruction, tels que les différents types de hasards qui peuvent se produire pendant l'exécution.

J.Swada et Hunt [10], utilisent l'approche de table de trace (trace table approach) Ils construisent une abstraction intermédiaire du processeur appelée table de trace, avec

laquelle ils peuvent formuler beaucoup de propriétés invariantes de l'implémentation pipeline. La méthode qui est basée sur la démonstration de théorème exige énormément d'efforts guidés de la part de l'utilisateur pendant le processus de vérification, d'abord en découvrant des invariants, et en ensuite, en les prouvant à travers le démonstrateur de théorème ACL2.

Hosabettu, Srivas et Gopalakrishnan, [15], proposent une méthodologie de vérification basée sur l'approche « de fonctions de complétude ». Dans une telle approche, la fonction d'abstraction est définie comme un ensemble de fonctions de complétude, où chaque fonction correspond à une instruction non finie dans la pipe. Chaque fonction de complétude spécifie l'effet désiré (sur les observables) de terminer l'instruction dans la pipe. En d'autres termes, comment une instruction non finie dans le pipe sera complétée d'une manière atomique en supposant celles en avant, déjà complétées. Tandis que leur approche a un grand avantage en décomposant la preuve en un ensemble de conditions de vérification qui rendent le raisonnement de chaque cas plus simple, il exige d'autre part un effort manuel considérable, en premier lieu, pour la définition des fonctions de complétude nécessaires pour la construction de la fonction d'abstraction, et en second lieu, pour guider la preuve à travers le démonstrateur de théorème PVS.

A.C.J. Fox, et N.A.Harmman [21,22], emploient les théorèmes à une seule étape (the one step theorems) pour prouver la correction de plusieurs types de microprocesseur s'étendant du séquentiel au superscalaire dans un cadre algébrique. Leur approche limite la condition de correction juste aux temps 0 et 1. Cependant, les fonctions d'état modélisant la spécification et l'implémentation doivent être consistantes au temps (time-consistent) vis-à-vis d'une fonction d'immersion (abstraction temporelle entre la spécification et l'implémentation). Cette propriété demeure une tâche difficile pour des architectures complexes et la mécanisation complète de la preuve demeure un objectif non accompli.

V.Bhagwati, et S. Devadas [48] ont proposé une méthodologie de preuve pour les processeurs pipelines, basée sur la notion de relation-β (relation établie entre l'implémentation séquentielle et l'implémentation pipeline). Pour effectuer la preuve, ils formalisent d'abord les deux implémentations séquentielle et pipeline en terme de machines d'états et les caractérisent ensuite en tant que machines "définies" (machine qui, pour une certaine constante *k*, sa sortie dépend uniquement des *k* dernières entrées). La condition de correction, est que la relation-β doit être vérifiée entre les deux implémentations. Pour montrer la validité de leur approche, ils décrivent les deux implémentations dans le langage de haut niveau BDS, et effectuent la preuve par

simulation symbolique. Cependant, ce travail ne montre pas clairement la mise à jour des différents composants observables manipulés par la machine pipeline (Compteur ordinal, Mémoire, registres, etc), durant les *k* derniers cycles d'horloges avant d'effectuer la comparaison.

Velev et Bryant [49] ont développé des méthodes combinant des symboles de fonctions non interprétées avec la vérification de modèle symbolique basée sur les BDD. Cependant, leur travail qui était concentré sur des techniques complètement automatiques, ne peut traiter des architectures hautement optimisées.

Bien que, les exemples que nous avons rapportés précédemment (parmi beaucoup d'autres) aient énormément contribué à l'effort scientifique de recherche dans la vérification formelle des architectures complexes, la vérification formelle automatique complète des microprocesseurs état-de-l'art n'est pas encore couverte.

1.3 Contributions

La méthodologie de preuve proposée dans cette dissertation développe des modèles fonctionnels précis (représentant des programmes fonctionnels) qui pourraient être utilisés aussi bien pour la vérification formelle que pour la simulation (Les implémentations réelles sont validées par la combinaison de ces deux techniques). Pour les architectures séquentielles comme pour les architectures pipelines, la méthodologie décompose la preuve globale en un ensemble de conditions de vérification plus simple à vérifier.

Pour les architectures séquentielles, la méthodologie raffine la preuve globale jusqu'au niveau composant observable. Un tel raffinement permet au processus de vérification de procéder en parallèle, et d'une manière incrémentale avec le processus de conception.

Pour les architectures pipelines, la méthodologie offre beaucoup d'avantages par rapport aux approches alternatives, en particulier :

- La possibilité de raisonner sur la dépendance inter instruction tels que les différents types de hasards qui peuvent se produire pendant l'exécution, contrairement à la technique de vidage, où un tel raisonnement est impossible. En outre, il est possible de raisonner soit sur des instructions individuelles soit sur des groupes d'instructions telles que des instruction-registres, des instructions-memoires et des instructions de branchement
- La possibilité d'instancier l'ensemble des conditions de vérification pour n'importe quelle architecture particulière, et par conséquent, elle offre une meilleure scalabilité pour la vérification des architectures hautement optimisées

- Puisque les deux modèles à comparer (séquentiel et pipeline) se rapportent au niveau microarchitecture, la méthodologie ne nécessite pas de fonction d'abstraction de données (qui demeure très difficile à définir en pratique), seulement une fonction d'abstraction de temps est nécessaire pour synchroniser les deux modèles. En plus, une telle synchronisation exige peu de cas par rapport à ceux employés par les approches alternatives
- Le point clé de la méthodologie de preuve proposée est l'habilité d'effectuer la preuve par induction dans le même langage de spécification, plutôt que par évaluation symbolique à travers un outil de preuve qui exige encore des efforts considérables.

1.4 Organisation de la thèse

Chapitre 2 discute les limites et les mérites des langages de description de matériel, impératifs et fonctionnels. Il présente en particulier le langage fonctionnel Haskell, utilisé le long de cette thèse en tant que cadre formel pour la spécification et la vérification des circuits digitaux.

Chapitre 3 présente la méthodologie de preuve proposée. Il commence par définir les couches d'intérêt du modèle de conception hiérarchique vis-à-vis de notre méthodologie, et décrit ensuite les différents ingrédients de l'environnement de vérification, ainsi que la stratégie de preuve suivie.

Chapitre 4 sera consacré entièrement à la spécification et la vérification des microarchitectures séquentielles. Après une description détaillée des différentes étapes de la preuve, le chapitre termine par une étude de cas détaillée sur la preuve formelle de la machine séquentielle MIPS.

Chapitre 5 introduit d'abord le principe du pipeline, et décrit ensuite les différentes étapes nécessaires à la vérification d'une microarchitecture pipeline. Une fonction de temps est définie pour synchroniser les deux modèles à comparer ; séquentiel et pipeline. Deux cas d'exécution avec et sans hasard sont discutés. Ce chapitre termine avec une étude de cas sur la preuve formelle de la machine pipeline MIPS.

Chapitre 6 présente en premier lieu le principe des architectures superscalaires, ensuite, il montre comment la même méthodologie peut explorer les résultats du pipeline pour modéliser et vérifier des superscalaires.

Finalement, chapitre 7 conclut le travail réalisé dans cette thèse, et suggère des directions futures.

Chapitre 2

Langages de description de matériel

2.1 Langages impératifs

Les langages de description de matériel (Hardware Description Languages: HDLs) développés principalement pour la description et la simulation du matériel, continuent malheureusement à être utilisés dans tout le processus de conception; de la spécification comportementale à la génération du produit final, impliquant ainsi d'autres activités telles que la synthèse et l'optimisation. Ils fournissent un environnement de développement qui permet à une spécification comportementale d'être graduellement transformée en une implémentation structurale (par synthèse et simulation). Dans une telle méthodologie de conception, la condition de correction fondamentale est que l'implémentation structurale générée au bas niveau, doit se comporter exactement comme la spécification comportementale de haut niveau. Les HDLs impératifs souffrent encore de l'absence de sémantique formelle, et le nombre important de bugs révélés dans les microprocesseurs modernes [50], montre que ces derniers ne peuvent pas être employés en tant qu'outils de synthèse formels. D'autre part, la description comportementale cachant tous les détails de l'implémentation, a besoin de mécanismes d'abstraction et de structuration plus puissants pour des architectures complexes. Donc, l'absence de combinateurs appropriés (combining forms) rend les descriptions impératives, longues, moins concises et extrêmement difficiles à lire.

Les études sur les HDLs impératifs standards, ont été la plupart du temps limitées à VHDL [51, 52, 53, 54], et à Verilog [55, 56, 57, 58]. VHDL a été créé en 1980 par le département de la défense des Etats-Unis, dans le cadre de leur programme VHSIC (Very High Speed Integration Circuit), en tant que langage de simulation. Par conséquent, la majorité des approches traitant la sémantique de VHDL sont opérationnelles. D'autres approches : fonctionnelle [59, 60], algèbre de processus [61], etc, ont aussi été utilisées pour formaliser la sémantique d'un sous ensemble de VHDL. La nécessité de se restreindre a un sous ensemble de ce langage est due à sa complexité qui rend l'expression de toutes ses caractéristiques une tache extrêmement difficile. Ces approches se limitent à la vérification de certaines propriétés (telle que la validation de l'état d'un composant observable après un nombre arbitraire de cycles d'horloges), ou d'équivalence sur le langage (de deux blocs d'instructions), à travers des démonstrateurs de théorèmes [62]. Donc des outils traducteurs de VHDL vers l'environnement du démonstrateur sont nécessaires. Ces outils posent encore le problème de crédibilité parce qu'ils sont difficilement prouvables. Par suite, la majorité des approches échouent à fournir une méthode de vérification pratique rigoureuse.

Verilog a été développé en 1984, par Phil Moorby (at Gateway Design Automation), et normalisé en 1990 par IEEE. Peu d'approches ont été proposées pour formaliser la sémantique de ce langage. Une approche informelle décrite dans [63] investit un sous ensemble de ce langage appelé Verilog V. Par suite, le même sous ensemble a été investit différemment par plusieurs chercheurs. Ceci constitue un avantage par rapport à VHDL où chaque groupe de travail investit un s/ensemble différent. Une autre approche adressant le s/ensemble V, utilisant la logique temporelle est décrite dans [64]. Similaire à VHDL, Verilog fournit toutes les propriétés désirables d'un HDL impératif, en particulier, la possibilité de décrire un circuit en plusieurs niveaux d'abstraction. Contrairement à VHDL qui est basé sur le langage ADA, Verilog est basé sur le langage C. En conséquence, il a une vitesse de simulation plus rapide que celle de VHDL. Une description plus détaillée des deux langages aussi bien qu'une comparaison entre eux, est donnée dans [64]. Un désavantage commun à ces deux langages, réside dans leur définition sémantique informelle (non rigoureusement définie) qui les rend indésirables pour la vérification formelle.

.

2.2 Langages fonctionnels

2.2.1 Caractéristiques

Les langages fonctionnels fournissent à côté de leur sémantique formelle, des caractéristiques puissantes qui ont démontré leur efficacité vis-à-vis de la conception formelle du matériel. Les caractéristiques les plus intéressantes sont discutées ci-après.

- **Transparence référentielle**: L'absence de l'effet de bord, est l'une des caractéristiques les plus importantes des langages purement fonctionnels. Une telle caractéristique permet de raisonner sur une spécification, en vue de dériver son implémentation, la transformer en une version plus efficace, ou démontrer sa correction.
- **Composition de fonction**: Puisque les opérations sont fonctionnelles, la composition de fonction permet d'exprimer directement des relations inter opérations. Dans la conception du matériel, les circuits complexes sont construits à base de composants primitifs, de la même façon que la composition de fonctions, par application, abstraction et nommage local.
- **Parallélisme**: Le style de programmation fonctionnelle n'impose aucun ordre d'exécution. Le programme entier consiste en un ensemble de définitions de fonctions qui peuvent être exécutées dans n'importe quel ordre. Par conséquent, les langages fonctionnels s'adaptent très bien aux traitements parallèles .
- **Classes de types**: Fournissent des mécanismes d'abstraction très commode pour la description de circuits, Elles fournissent aussi des moyens pour dériver génériquement des classes spécifiques de circuits. En typant fortement les fonctions, la notion de classe, permet de capturer autant d'erreurs que possible lors de la compilation.
- **Sémantique non-stricte**: utilisée avec l'évaluation paresseuse (lazy evaluation), cette caractéristique permet de raisonner sur des spécifications impliquant des structures illimitées. Par exemple les signaux peuvent être modélisés d'une manière élégante par des listes infinies (streams). En outre, l'évaluation paresseuse peut mener à la terminaison pendant que l'évaluation désireuse (eager evaluation) échoue.
- **Polymorphisme paramétrique**: Pour rendre la définition de certaines fonctions plus générale (polymorphique), la définition de type peut être paramétrée par des variables (appelées variables de type). Ceci permet à des spécifications génériques d'être utilisées dans différents contextes. Par exemple, une spécification polymorphique d'un composant, peut être réutilisée à différents niveaux d'abstraction.

- **Fonctions d'ordre supérieure**: Le mérite des fonctions d'ordre supérieure (combining forms) est très reconnu par les concepteurs de matériel. Leur capacité de faire des abstractions leur permet d'être utilisées pour réaliser plusieurs objectifs : structuration et réutilisation des spécifications de circuit, description et transformation des architectures régulières, etc. Cependant, l'utilisation des types d'ordre supérieure rend leur preuve de correction très difficile.

- **Description comportementale et structurale**: Il y a deux manières différentes pour la description des circuit: Structurale et comportementale. La description structurale exige du concepteur d'indiquer les différents composants à utiliser et la manière de les connecter (les composants connectés sont aussi décrits d'une manière structurale, ou utilisés en tant que composants primitifs par un HDL fonctionnel). Dans la description comportementale, un composant peut être décrit en spécifiant uniquement son comportement ou sa fonctionnalité. La structure correspondante peut être synthétisée automatiquement si le circuit est relativement simple, ou développée séparément si le circuit est complexe. Dans ce cas, une preuve de correction est nécessaire. Les langages fonctionnels possèdent des caractéristiques qui leur permettent de spécifier les deux aspects (comportemental et structural) d'un circuit. Donc, on peut utiliser un langage fonctionnel soit en tant que Meta-langage pour décrire le comportement, soit utiliser uniquement un sous ensemble du langage (embedded HDL) pour décrire la structure.

- **Simulation fonctionnelle**: Les spécifications fonctionnelles sont des spécifications exécutables qui peuvent être directement exécutées pour observer le comportement d'un circuit au cours du processus de développement (plutôt qu'attendre la terminaison de sa conception entière). En outre, les spécifications exécutables permettent de comparer les sorties d'un outil de simulation à celles inférées par une spécification formelle de haut niveau. En conséquence, la création des spécifications exécutables constitue l'un des objectifs de la méthodologie développée dans cette thèse.

2.2.2 Langages de Description de matériel incorporés

Il y a deux approches bien connues pour la conception et l'implémentation des langages de description de matériel:

- Construire un langage à partir de zéro. Une telle approche exige concevoir d'abord une syntaxe pour des descriptions de circuit, ensuite, décider d'une sémantique appropriée pour implémenter un interpréteur ou un compilateur. Tandis qu'une telle approche semble être la plus évidente pour définir des sémantiques alternatives qui correspondent à des

activités de conception alternatives (Simulation, synthèse, vérification, etc), pour la même syntaxe, elle demeure cependant très difficile pour le développement des interpréteurs ou des compilateurs entiers. En particulier, pour la conception des circuits digitaux complexes qui demandent des langages très riches.

- Construire un langage sur le substrat d'un langage existant. Une telle approche appelée approche d'incorporation (embedding approach), consiste à construire une bibliothèque (domain-specific) sur un langage général existant. Une telle bibliothèque fournit différentes fonctions sémantiques, chaqu'une correspond à une activité de conception particulière, telles que simulation, synthèse, vérification, etc. L'idée derrière l'approche de langage incorporé est la réutilisation de la syntaxe, des caractéristiques, et des outils d'un langage de programmation existant – appelé langage hôte. Le choix du langage hôte est fondamentalement important, parce que parfois, les caractéristiques du langage hôte n'assortissent pas les caractéristiques désirées du langage incorporé. Les langages fonctionnels ont des caractéristiques clés qui les rendent bien adaptés pour le développement des HDLs incorporés. Une brève description des HDLs fonctionnels incorporés les plus importants est donnée ci-après:

- **Hydra** [65] Consiste en un ensemble d'outils logiciels incorporés dans Haskell [66] pour la conception de circuits digitaux. Il permet la modélisation des circuits à différents niveaux d'abstraction, s'étendant de la représentation abstraite utilisant des listes paresseuses, aux réalisations à base de portes CMOS et NMOS. Hydra utilise des fonctions d'ordre supérieur (combining forms) pour décrire et transformer des circuits en vue d'une implémentation directe (sous forme de netlists). L'approche d'interprétation multiples (pour effectuer plusieurs activités possibles : simulation, synthèse, etc) est aussi supporté par l'environnement Hydra. Un tel HDL a été utilisé dans l'enseignement des architectures des ordinateurs au niveau de graduation de l'université de Glasgow.

- **Lava** [67] est un langage de description structurale incorporé dans la langage fonctionnel Haskell. L'incorporation est implémentée comme bibliothèque de description du matériel, se composant de deux parties: La première partie de la bibliothèque fournit des fonctions primitives pour construire des circuits: portes, registres, mémoires, etc. et des fonctions d'ordre supérieure (Combining formes) pour structurer des conceptions complexes. La deuxième partie de la bibliothèque de Lava fournit un ensemble d'outils qui peuvent effectuer des interprétations intéressantes, telles que la simulation pour tester le comportement, la vérification pour prouver les propriétés du circuit (à travers des

démonstrateurs de théorèmes qui peuvent être reliés directement a Lava), et la génération de code VHDL pour l'implémentation des FPGAs .

▪ **Hawk** [68] est un autre langage de description de matériel récent incorporé dans Haskell. L'objectif de Hawk est de fournir des spécifications claires et concises pour des microarchitectures de microprocesseur état-de-l'art. Pour réaliser ceci, Hawk fournit des fonctions de bibliothèque pour décrire des microarchitectures complexes telles que les transactions (des instructions machine groupées avec leurs états). Comparé à d'autres HDLs, Hawk tackle des spécifications de haut niveau. Similairement à Hydra et Lava, Hawk fournit aussi des fonctions d'ordre supérieure (combining forms) pour effectuer des abstractions à haut niveau.

2.3 Haskell : Présentation générale

Haskell est un langage purement fonctionnel incorporant plusieurs caractéristiques importantes telles que la sémantique non stricte, les fonctions d'ordre supérieure, et le polymorphisme. Il a été utilisé en tant que langage hôte pour incorporer plusieurs HDLs. Ses caractéristiques principales sont discutées ci-après:

2.3.1 Définition des types

2.3.1.1 Types de base

Puisque Haskell est purement fonctionnel, tous les calculs sont effectués à travers l'évaluation des expressions (termes syntaxiques) qui produisent des valeurs statiques. Chaque valeur a un type qui lui est associé. Les types de base prédéfinis incluent : *Int* pour les entiers, *Bool* pour les booléens, et *Char* pour les caractères.

Le système de type statique assure qu'un programme Haskell est typé d'une manière correcte, par conséquent, toutes les erreurs de type sont détectées à la compilation.

2.3.1.2 Types algébriques

- **Types énumérés**

Un type de données algébrique introduit un nouveau type et des constructeurs sur ce type utilisant la déclaration *data.* Par exemple, le type Bool peut être défini en énumérant ses éléments *False* et *True.*

Data Bool = False | Ttrue

Bool est appelé constructeur de type, tandis que False et True sont appelés des constructeurs de données (ou juste des constructeurs).

- **Types récursifs**

Similairement, nous pouvons définir un type récursif comme suit :

data list = Nil | Cons Object List

Cela signifie qu'un objet de type liste est soit vide (Nil), soit il est construit (récursivement) par le constructeur de données Cons, en préfixant un objet à une liste.

- **Types polymorphiques**

Les types algébriques peuvent être aussi polymorphiques, où une (ou plusieurs) variable de type, peut être utilisée en tant que paramètre dans la définition de type. Dans l'exemple précédent, on peut introduire une variable de type 'a' pour rendre le type List polymorphique comme suit:

data List a = Nil | Cons a (List a)

Cette définition quantifie pour une famille de listes : Liste de booléens, liste d'entiers, etc. Donc, cette définition paramétrée peut être réutilisée pour plusieurs types de listes

2.3.1.3 Synonymes de types

Les synonymes de types ne définissent pas de nouveaux types, mais assignent de nouveaux noms à des types existants. Les nouveaux noms sont souvent plus courts. Les synonymes de noms peuvent améliorer la lisibilité des programmes par des mnémoniques. Les synonymes de types sont crées par la déclaration : type. Par exemple:

type Address = Int
type PC = Address

Donc, les deux types : Address et PC sont des synonymes du type Int.

2.3.1.4 Classes de types

Le system de type de Haskell a été étendu par la notion de classe de types (ou classe tout court) pour résoudre le problème des opérations surchargées. Par exemple, l'opérateur numérique +, opère sur plusieurs types de nombres qui rendent le comportement de ce dernier souvent ambigu. Donc, les classes de type sont introduites pour contrôler ce

'surchargement', souvent appelé ad hoc polymorphisme (overloading). Une déclaration de Classe introduit une classe de types et les opérations surchargées qui doivent être supportées par n'importe quel type qui est une instance de cette classe. Une déclaration d'instance indique qu'un type est une instance d'une classe, et inclue la définition des opérations surchargées appelées méthodes. Par exemple, nous pouvons introduire une nouvelle classe appelée Num pour surcharger les opérations (+) et negate sur les types Int et Float comme suit :

```
class Num a where
  (+) :: a – a – a
  negate :: a – a
```

Maintenant nous pouvons déclarer Int et Float comme instances de cette classe.

```
Instance Num Int where
  x + y = addInt x y
  negate x = negateInt x

instance Num Float where
  x + y = addFloat x y
  negatex = negateFloat x
```

Les fonctions addInt, addFloat, negateInt, and negateFloat sont des fonctions primitives, mais qui peuvent être aussi des fonctions définies par l'utilisateur.

2.3.1.5 Les listes

Dans Haskell, les listes sont des types de données algébriques avec deux constructeurs. Le premier constructeur c'est la liste nulle (vide) : [ ], et le second constructeur c'est ' **:**' (lire cons), qui préfixe un élément à la liste. Tous les éléments de la liste ont le même type. La liste des éléments : x1,…xn, est dénotée comme suit : [x1,…xn], c'est une contraction de la liste : x1 **:**,…,**:** xn **:**[]. Ainsi, une liste polymorphique récursive paramétrée par la variable de type 'a', peut être redéfinie comme suit : data [a] = [ ] | a **:** [a]

2.3.1.6 Les tuples

La différence entre une liste et un tuple est que les éléments d'un tuple peuvent avoir différents types tandis que les éléments d'une liste doivent avoir le même type. En outre, une liste est chaînée et peut avoir une longueur infinie, alors qu'un tuple est un

enregistrement avec un nombre fini de champs (au mois deux champs). Chaque champ contient un objet avec un type qui lui est associé. Haskell utilise des crochets pour dénoter des listes et des parenthèses pour dénoter des tuples.
Supposons que le format d'une instruction machine est divisé en 5 champs. Donc, le type de l'instruction peut être défini comme suit :

type Field = [Bit]
type IF = (Field, Field, Field, Field, Field).

Il faut noter que les différents champs peuvent en générale,avoir des types différents

2.3.2 Définition des fonctions

2.3.2.1 Définition par équations

Les fonctions sont définies par des équations de la forme : f x1,…,xn = E, ou f est le nom de la fonction, x_1,…,x_n, sont les arguments (paramètres formels) de la fonction, et E est une expression. L'application de la fonction est dénotée par une simple juxtaposition de la fonction et de ses arguments réels (curried form) comme suit : f a_1,…,a_n.
Soient t_i les types des arguments x_i, x_1:: t_1,…,x_n:: t_n et t, le type du résultat attendu. Alors, le type (appelé aussi signature) de la fonction f est exprimé comme suit :

f :: t1 – t2,…,tn – t

Exemple : add :: Int – Int – Int
add x y = x + y
Dans Haskell, la première lettre de la fonction doit être en minuscule

2.3.2.2 Définition par abstraction lambda

Au lieu d'utiliser les équations pour définir des fonctions, Haskell donne la possibilité de les définir d'une manière anonyme, à travers des abstractions lambda. Par exemple, la fonction add pourra être définie par abstraction lambda comme suit:

Add = \ x y – x+y

En générale, étant donnés, x ::t1, exp :: t2, alors \x – exp, possède le type t1 – t2

2.3.2.3 Fonctions non strictes

Une fonction f est dite stricte si et seulement si : f(⊥) = ⊥, où ⊥ (lire bottom), est une expression indéfinie. En d'autres termes, une fonction est dite stricte si, quand elle est appliquée à une expression indéfinie, elle échoue de terminer.

La majorité des langages de programmation sont stricts (avant d'appliquer la fonction il faut d'abord évaluer tous les arguments). Ce n'est pas le cas pour le langage fonctionnel Haskell, où les fonctions sont non-strictes (leurs arguments sont évalués paresseusement ou par besoin). Considérons la fonction polymorphique suivante :

first ::a – a – a
first(x,y) = x

Puisque le résultat dépend uniquement de la valeur du premier argument, la fonction ne tente jamais d'évaluer son second argument. Donc, si on applique la fonction aux valeurs suivantes : x = 1, y = 1/0, la fonction first, retourne un résultat valide, bien que le second argument est indéfini. Un langage ayant une sémantique stricte tel que SML, retourne une erreur (runtime error). La majorité des fonctions logiques sont non-strictes.

2.3.2.4 Infixité des opérateurs

Dans Haskell, les opérateurs numériques tels que (+) et (-), peuvent être aussi, définis d'une manière infixe. Par exemple, les opérateurs de conjonction (&&), et de disjonction (||), peuvent être définis comme suit :

x && y = if x then y else False
x || y = if x then True else y

Du point de vue lexical, les opérateurs infixes sont formés entièrement par des symboles. Contrairement, aux identificateurs normaux qui sont alphanumériques.

2.3.2.5 Formes de définition

Haskell fournit plusieurs manières de définir une fonction. Nous donnons brièvement ci-après quelques formes de définition.

- **Par Patterns**

Les fonctions sont définies par des 'patterns' et le processus d'évaluation est effectué par comparaison des arguments réels auxquels la fonction est appliquée, aux arguments

formels corrects de la liste des cas spécifiés dans la définition. Ci-après, l'exemple de la fonction factorielle qui est définie pour deux patterns : 0 et n.

```
fac 0 = 1
fac n = n*fac(n-1)
```

Dans ce cas, l'algorithme rentre en boucle pour des tentatives d'évaluer la fonction fac pour des arguments négatifs.

- **Patterns avec des gardes**

Le problème précédent peut être résolu par le rajout des gardes (expressions booléennes sur des patterns) comme suit :

```
fac 0 = 1
fac n | n > 0  =  n*fac(n-1)
```

Le test rajouté (n>0) évite d'évaluer la fonction pour des valeurs n négatives. Une version plus améliorée contenant une seule occurrence du nom de la fonction est donnée ci-après.

```
fac n  | n = = 0  = 1
   |  n > 0  = n*fac(n-1)
```

- **Expressions conditionnelles**

La fonction fac peut être aussi définie par des expressions conditionnelles comme suit :

```
fac n =  if n = = 0  then 1  else n*fac(n-1)
```

Une telle définition ne protège pas la fonction contre des valeurs indésirables

- **Expressions de cas**

Les expressions de cas font apparaître les patterns d'une manière plus explicite.

```
fac n = case n of
      0 –  1
      m | m > 0 –  n * fac(n-1)
```

Dans cet exemple nous voyons une liste de deux patterns: 0 et m, qui sont comparés avec n. Si l'une des comparaisons réussit, l'expression correspondante est choisie. Si aucune comparaison ne réussit, un message d'erreur apparaît.

- **Compréhension de listes**

Haskell fournit une expression élégante de création des listes (list comprehensions), qui a la forme suivante :

[f x | x ← xs, px] : Lire comme suit : 'la liste de tous les f x, tel que x est généré de xs', où xs est une expression de listes, f une fonction sur les listes, et p une fonction booléenne. La signification de l'expression de la compréhension de liste, est de nouveau une liste construite comme suit :

- Les éléments de liste xs sont scannés de gauche a droite
- Sur chaque élément x, le test p est effectué
- Si p x = Vrai, f x est placé dans la liste résultat.
- Autrement, x est ignoré.

Par exemple, la fonction zip qui accepte deux listes et retourne une liste de paires (combinant les éléments de même rang des deux listes d'entrée), peut être définie d'une manière simple et compacte par compréhension de listes.

```
zip :: [a] – [b] – [(a,b)]
zip as bs = [(a,b) | a ← as & b ← bs]
```

- **Définitions locales**

- **Expression Let**

L'expression let permet d'introduire une liste de déclarations imbriquées (nested déclarations) locales à une fonction. Elle a la forme suivante : Let {d1,…dn} in e. La portée (scope) des déclarations di est l'expression e. L'exemple suivant illustre son utilisation.

```
f x y = let  a = square (y +1)
        in  (x +a)
```

L'expression let, permet d'exprimer des structures hiérarchiques d'une manière élégante.

- **Clause where**

La clause where permet de nommer des expressions pour rendre la définition d'une fonction plus simple et plus lisible. L'exemple précédent est réutilisé avec la clause where.

```
f x y = x + a
  where a = square (y+1)
```

2.3.2.6 Fonctions d'ordre supérieur

Les fonctions qui acceptent d'autres fonctions en tant que paramètres ou qui produisent des fonctions comme résultats sont appelées des fonctions d'ordre supérieur. Une telle caractéristique qui permet de traiter des fonctions en tant qu'objets de données est extrêmement importante dans les langages fonctionnels, parce qu'elle permet de faire des abstractions pour structurer et réutiliser des spécifications. Nous donnons ci-après quelques exemples qui nous serons utiles par la suite.

- **Fonction map**

Comme premier exemple, considérons la fonction map définie par patterns et d'une manière récursive sur les listes comme suit :

```
map :: (a – b) – [a] – [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Sémantiquement parlant, la fonction map applique la fonction polymorphique f (d'une manière itérative) à tous les éléments de la liste d'entrée xs. Soit xs = [x1,…,xn], alors le résultat est la liste suivante : [fx1, …,fxn].
La fonction map est très utilisée dans la définition des opérateurs logiques.

- **Fonction zipwith**

La fonction zipwith est définie comme suit :

```
zipWith:: (a – b – c) – [a] – [b] – [c]
zipWith f (a:x) (b:y) = f a b: zipWith f x y
```

Cette fonction accepte en entrée une fonction polymorphique f, et deux listes. Elle donne comme résultat une liste dont les éléments sont obtenus en appliquant la fonction f aux éléments de même rang de chaque liste.
Similairement à la fonction map, la fonction zipwith peut être utilisée pour exprimer d'une manière élégante les opérateurs logiques

- **Fonction fold**

La définition de la fonction fold est plus générale dans la mesure où elle permet de convertir les types de listes en d'autres types. Elle existe sous deux formes : foldr et foldl qui sont définies comme suit :

- *Pliage a droite*

```
foldr :: (a – b – b) – b – [a] – b
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
```

- *Pliage a gauche*

```
foldl::(a – b – a) – a – [b] -> a
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

La fonction foldr prend un opérateur et l'insère entre les éléments d'une liste qui commence à droite par une valeur donnée. La fonction foldl assume la même tache, mais en commençant par la gauche. Les deux formes produisent un singleton. Tous les opérateurs numériques peuvent être exprimés par la fonction fold.

- **Fonction Scan**

Parfois, il serait utile de connaître les valeurs internes (partielles) calculées par la fonction fold. Ceci exige la définition d'une liste de résultats partiels; c'est ce qu'on appelle scannage de la liste. L'opération de scannage existe aussi sous deux formes : scanl et scanr qui sont définies comme suit :

- *Scannage à droite*

```
scanr:: (a – b – b) – b – [a] – [b]
scanr f a [] = []
scanr f a (x:xs) = (f a x): scanr f a xs
```

- *Scannage à gauche*

```
scanl :: (a – b – a) – a – [b] – [a]
scanl f a [] =[]
scanl f a (x;xs) = a: scanl f (f a x) xs
```

La fonction scanl calcule les résultats partiels de la fonction foldl et la fonction scanr calcule les résultats partiels de la fonction foldr.

- **Fonction mapscan**

La fonction mapscan combine les trois formes : map, fold et scan en même temps. Elle existe sous deux formes : mscanl et mscanr.

- *Fonction mscanl*

```
mscanl :: (a – b – (a,c)) – a – [a] – (a,[c])
mscanl f a [] =(a, [])
mscanl f a (x:xs) = (a'' , y:ys)
 where  (a', y)    = f a x
      (a'', ys) = mscan l f a' xs
```

- *Fonction mscanr*

```
mscanr :: (a – b – (a,c)) – a – [a] – (a,[c])
mscanr f a [] =(a, [])
mscanr f a (x:xs) = (a'' , y:ys)
 where (a'', y) = f a' x
   (a', ys) = mscanr f a xs
```

La forme mscanr envoie l'information de gauche à droite, calcule les résultats intermédiaires a', et produit le résultat final du tuple (a'', y :ys). La forme mscanl est similaire à la forme mscanr, mais elle envoie l'information dans le sens opposé.

D'autres combinateurs très utiles pour la structuration des spécifications peuvent être trouvés dans le standard prélude de Haskell.

2.4 Haskell en tant que HDL

Les langages fonctionnels ont été utilises avec succès dans la conception du matériel. Dans cette section nous montrerons comment le langage fonctionnel Haskell pourra être utilisé en tant que HDL pour décrire la structure d'un circuit.

2.4.1 Définition des signaux

2.4.1.1 Classes de signaux

Il est plus convenable d'utiliser différents types pour représenter un signal. Par exemple, nous pouvons utiliser uniquement le type booléen pour supporter l'algèbre de Boole. Pour raisonner sur des portes logiques CMOS, nous aurons besoin du type voltage pour représenter des valeurs spéciales telles que : 'Isolated' et 'Shorted'. Le comportement

dynamique des circuits séquentiels exige une représentation par listes infinies (streams). La conclusion est de rendre le type signale polymorphique. Par exemple : inv :: a – a
Mais cette définition montre aussi que l'inverseur peut opérer sur plusieurs types. Ce problème de surchargement peut être résolu par la déclaration de classe de type suivante:

```
Class signal a where
inv :: a – a
```

Qui a pour effet d'assigner à l'opérateur inverseur, le type suivant :

```
inv :: (Signal a) ⇒a – a
```

Cela signifie que pour n'importe quel type a, qui est une représentation valide d'un signal, l'opérateur inv aura le type : a – a
Haskell supporte aussi la notion d'inclusion de classe qui est très utile dans le contexte de conception du matériel. Par exemple, la déclaration de classe suivante :

```
Class (Eq a, Num a) ⇒Signal a where
 Zero, one :: a
 Inv :: a – a
 Or2, and2 :: a – a – a
```

Signifie qu'un type arbitraire 'a' est dans la classe Signal, s'il est aussi dans la classe Eq et dans la classe Num. La classe Eq permet aux signaux d'être comparés, alors que la classe Num permet d'exprimer des fonctions logiques telles que : *and*, *inv*, etc, en utilisant les opérateurs conventionnels numériques : *, +, et l'opérateur unaire (-).

Les signaux peuvent être statiques pour représenter une seule valeur, ou dynamique pour représenter une séquence de valeurs dans le temps. Les signaux dynamiques sont représentés par des listes, ou le nième élément de la liste représente la valeur du signal durant le cycle d'horloge *n.*

Nous pouvons définir une classe pour chaque genre de signal avec les opérations qui lui sont associées comme suit :

```
Classe (Signal a) ⇒Static a where
 Inv :: a – a
Classe (Signal a) ⇒Dynamic a where
 Latch :: a – a
```

Les deux classes Statique et dynamique sont des sous classes de la classe Signal. Par conséquent, elles héritent toutes les opérations relatives à la classe Signal. L'effet de la déclaration ci-dessous est d'assigner aux opérateurs inv et latch les types suivants :

Inv :: (static) $\Rightarrow$a $-$ a

Latch :: (Dynamic) $\Rightarrow$a $-$ a

D'autres classes telles: les 'Latices', peuvent être définies pour introduire des opérations spécifiques à d'autres parties du matériel (tels que les portes à trois états).

2.4.1.2 Représentation des signaux

La représentation des signaux dépend du niveau d'abstraction manipulé et de l'activité de conception désirée. Par exemple au niveau microarchitecture, l'information est interprétée en termes de bits, alors qu'au niveau transistor, l'information est interprétée en termes de voltage. D'autre part, une activité de simulation n'exige pas plus que la notion de liste pour la modélisation de signaux, alors que l'activité de synthèse (pour la génération de netlistes) exige une représentation de signal qui permet de révéler la structure du circuit. Pour des raisons de simulation, un signal peut être défini d'une manière polymorphique comme suit :

type Signal a = [a]

Si on s'intéresse par exemple au niveau microarchitecture ou l'information est interprétée en termes de bits, on peut instancier la variable de type par le type Bit, et modéliser le type Signal par une liste infinie (streams) de bits.

Pour des raisons de synthèse, un signale exige une représentation qui révèle la structure du circuit. Une représentation valide est donnée ci-après.

data Signal = Var String | Comp String [Signal]

Cette définition signifie qu'un signale est soit le nom d'une variable d'entrée, soit le nom du composant qui est sollicité par les variables d'entrée.

Pour pouvoir détecter le partage et résoudre le problème de boucle, une solution consiste à nommer explicitement les composants par le rajout d'un identificateur tag comme suit :

Data Signal = Var String | Comp Tag String [Signal]

Plus de détailles sur la représentation de signaux par Haskell peut être trouvée dans la littérature suivante [26, 65, 67].

2.4.2 Définition des Circuits

Les circuits complexes sont décrits d'une manière hiérarchique en termes de composants primitifs de la même manière que la composition de fonctions, par application, abstraction et nommage local. Dans cette section, nous allons utiliser le langage Haskell pour illuster ce concept a travers des exemples simples.

2.4.2.1 Circuits combinatoires

Ci-après, quelques exemples de composants primitifs modélisés en tant que fonctions qui opèrent sur un type signal définit (en section 2.4.1.2), par la notion de liste, en tant que sequence infinie de bits.

```
data Bit = Zero | One

inv :: Signal Bit – Signal Bit
inv xs = map not xs

and :: Signal Bit – Signal Bit – Signal Bit
and xs ys = zipwith (&&) xs ys

or :: Signal Bit – Signal Bit – Signal Bit
or xs ys = zipwith ( || ) xs ys

xor :: Signal Bit – Signal Bit – Signal Bit
xor xs ys = zipwith (/= ) xs ys
```

En utilisant les composants primitifs définis ci dessous, un demi additionneur sera construit d'une manière très simple comme suit :

```
halfAdd :: Signal Bit – Signal Bit – (Signal Bit, Signal Bit)
halfAdd x y = (and x y, xor x y)
```

Incrémentalement, on peut construire un additionneur complet comme suit:

```
fullAdd :: Signal Bit – Signal Bit – Signal Bit – (Signal Bit, signal Bit)
fullAdd  a b c = let (s1,c1) = halfAdd a b
                     (s2, c2) = halfAdd s1 c
                 in (s2, xor c1 c2)
```

Maintenant, on peut construire un additionneur qui peut opérer sur des mots de longueur arbitraire comme suit :

```
type Word = [Bit]

wadder :: Signal Word – Signal Word – Signal Bit – (Signal Bit, Signal Word)
waddrer [] [] c = (c,[])
wadder (x:xs) (y:ys) c = let (c'',s) = fullAdd (x, y,c')
                             (c', ss) = wadder x sys c
                         in (c'', s:ss)
```

Une telle spécification construit l'architecture de l'additionneur mot, par une série d'additionneurs complets connectés à travers les fils modélisant les signaux des retenus (en sortie), comme l'indique le schéma de la figure 2.1

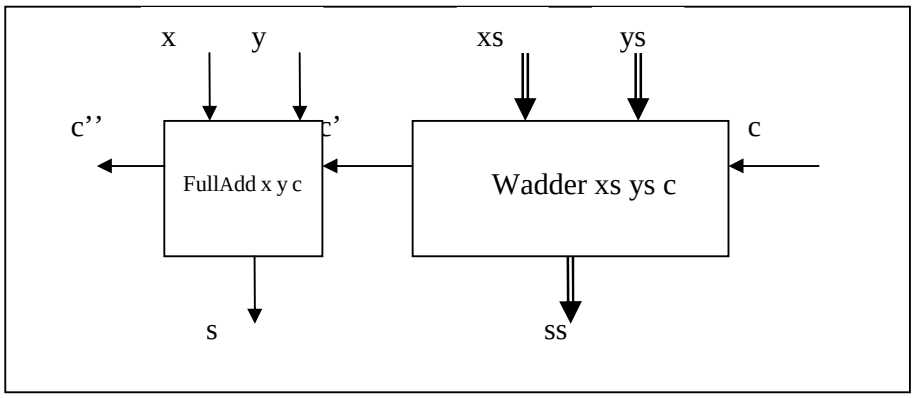


Figure 2.1. Additionneur mot

Une spécification plus élégante de l'architecture de l'additionneur peut être construite en utilisant le combinateur mscanr défini dans la section 2.3.2.6

En premier lieu nous devons zipper les deux mots d'entrée de telle manière à ce que chaque position de bit i reçoit une paire (xi, yi). De cette manière, un additionneur mot d'une longueur arbitraire peut être défini en une seule équation comme suit :

```
wadder xs ys c = mscanr fulladd c (zip xs ys).
```

Cependant, l'utilisation des types d'ordre supérieure rend la preuve de correction du circuit décrit, très difficile.

2.4.2.2 Circuits Séquentiels

Contrairement aux circuits combinatoires qui ont un comportement statique, les circuits séquentiels ont un comportement dynamique variable dans le temps. Donc, la notion de stream (liste infinie), fournie par le langage Haskell est fondamentalement importante pour modéliser les signaux sous forme de séquences de valeurs infinies (variables dans le temps). L'idée est que l élément de la liste de rang n, représente la valeur du signal durant le cycle n. L'exemple le plus simple est celui d'une bascule D, souvent appelée, "latch". La sortie d'un latch à l'instant t, c'est la valeur de son entrée à l'instant t-1. Fonctionnellement, un latch peut être spécifié comme suit :

```
Latch :: a – Signal a – Signal a
Latch  x xs = x : xs
```

Si on fixe la valeur initiale à Zero, et on instancie la variable de type 'a' par le type de données Bit, la spécification du latch simplifie comme suit :

```
Latch :: Signal Bit – Signal Bit
Latch xs = Zero : xs
```

Maintenant, on peut utiliser la définition du latch pour spécifier un 'toggle' avec une boucle (feedback) explicite comme suit :

```
toggle :: Signal a – Signal a
toggle in = let out = latch (xor in out) in out.
```

Figure 2.2, schématise une telle définition.

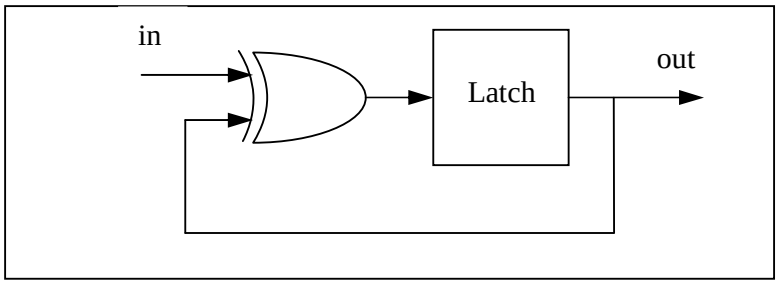


Figure 2.2: Circuit avec boucle

Noter que dans cette définition, la valeur out dépend d'elle même (définition mutuellement dépendante). Dans une sémantique stricte, la signification de cette expression serait indéfinie.

2.5 Conclusion

Pour combler les insuffisances citées auparavant des HDLS impératifs, (en particulier l'aspect formel et l'aspect de structuration), des approches multi paradigmes combinant des techniques fonctionnelles et impératives ont été proposées [69]. Dans ces approches, Les techniques fonctionnelles sont réservées a la description du comportement qui est utilisé en tant que modèle de référence, soit pour dériver une description VHDL structurale, soit pour valider une description structurale décrite séparément par des techniques impératives. Pendant que la première approche demeure extrêmement difficile, la seconde approche demeure aussi insuffisante, en particulier pour des architectures hautement optimisées. Puisque les HDLs impératif restent inconvenables vis-à-vis de la spécification et la vérification formelle, le reste de cette thèse se focalise sur des HDLs purement fonctionnels qui fournissent des caractéristiques plus intéressantes qui ont démontré leur viabilité vis-à-vis de la conception des architectures complexes.

Chapitre 3

Méthodologie de preuve

3.1 Objectif

Le processus de conception des circuits digitaux suit un modèle en couches qui transforme progressivement une description abstraite de haut niveau en un produit final généré au bas niveau. A travers ce processus de transformation qui est connu sous le nom de synthèse, chaque couche représente un raffinement de la couche immédiatement supérieure (formellement parlant, chaque couche représente une implémentation qui doit préserver les propriétés de la couche immédiatement supérieure). Dans ce travail, nous nous intéressons à la couche de haut niveau et celle immédiatement inférieure. Figure 3.1 montre les deux couches d'intérêt. Naturellement, la couche de haut niveau décrit l'architecture du jeu d'instructions du microprocesseur (Spécification ISA). La couche immédiatement inférieure décrit la Micro-Architecture (implémentation[1] MA) implémentant les opérations du microprocesseur (d'autres approches implémentent le niveau de transfert de registre : RTL, dans cette couche). Actuellement, il est extrêmement difficile, voire impossible de dériver formellement une implémentation microarchitecturale hautement optimisée à partir d'une spécification ISA. Un autre obstacle majeure rendant cette implémentation plus difficile est la différence de sémantique : Une description ISA a une sémantique mathématique, alors qu'une implémentation microarchitecturale a une sémantique en termes de circuits. Ces deux problèmes ont menés les concepteurs de circuits intégrés à développer séparément une description microarchitecturale

[1] Dans notre contexte, une implémentation formelle et appelée aussi spécification

(ou RTL) à partir de laquelle le processus de synthèse doit démarrer. Par suite, La correction du circuit final généré au bas niveau dépend de la correction de la description microarchitecturale.

A travers cette thèse, nous proposons une méthodologie de preuve permettant de modéliser et de vérifier des microarchitectures RISC, dans un environnement fonctionnel. En d'autres termes, la méthodologie proposée permet de préparer des descriptions microarchitecturales correctes pour le processus de synthèse.

Les sections qui suivent décrivent les ingrédients nécessaires à l'environnement de preuve, ainsi que la stratégie de preuve suivie.

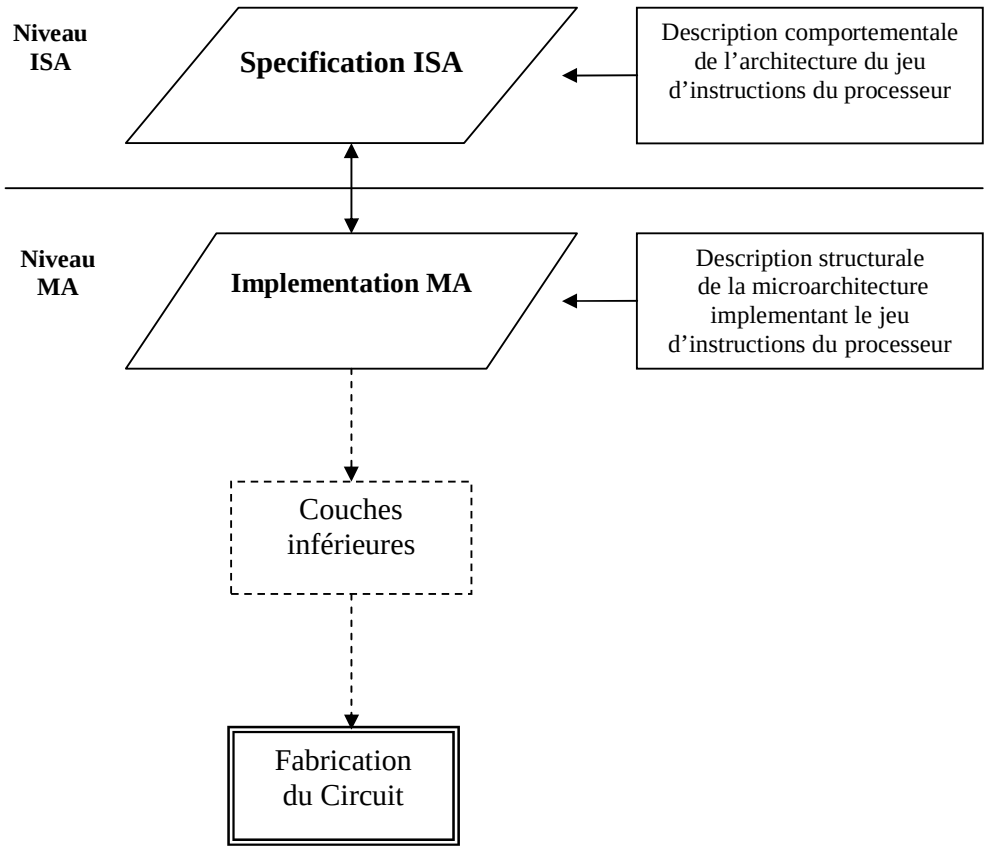


Figure 3.1 : Modèle hiérarchique de conception des circuits digitaux
Montrant les couches d'intérêt.

3.2 Ingredients de l'environnement de preuve

3.2.1 Fonction d'etat

3.2.1.1 Definition

Soient, S: un ensemble non vide appelé espace d'état, c: un état initial, et $f :: S - S$, une fonction de transition. Une *fonction d'état* F, est récursivement définie comme suit:

$F :: (Int, S) - S$

$F(0, c) = c$

$F((n+1), c) = f(F(n,c))$

Puisque l'état suivant est toujours fonction de l'état précédent, in système modélisé par la notion de fonction est déterministique. La transition entre deux états adjacents observables est appelé: *étape*. Par exemple, F(n,c) représente l'état après n étapes, étant donnés un état initial c, et une fonction de transition *f*. Sa valeur est donnée par: $F(n,c) = f^{n}(c)$.

3.2.1.2 Décomposition d'état

La distribution de l'espace d'état d'une machine sur ses différents composants exige la décomposition de la fonction d'état et la fonction de transition en coordonnées.

Soit $S = (S_1,...,S_k)$, l'espace d'état d'une machine, distribue sur k composants (les observables), où S_i est l'état du composant i, pour $1 \leq i \leq k$.. Ainsi, les fonctions d'état et de transition seront décomposées comme suit:

$F(n, c_1,..., c_k) = (F_1(n, c_1,..., c_k),...,F_k(n, c_1 ..., c_k))$

$f(c_1,\ldots, c_k) = (f_1(c_1,..., c_k),\ldots, f_k(c_1,..., c_k))$

Où, $F_i :: (Int, S) - S_i$

$F_i(0, c_1,\ldots, c_k) = c_i$

$F_i(n, c_1,\ldots, c_k) = f_i(F_1((n-1), c_1,..., c_k),...,F_k((n-1), c_1,\ldots, c_k))$

Et $f_i :: S - S_i$, pour $1 \leq i \leq k$.

De cette manière, chaque coordonnée F_i calcule uniquement l'état du i^{ieme} composant de la fonction d'état F, et chaque coordonnée f_i calcule uniquement l'état du i^{ieme} composant de la fonction de transition f.

3.2.1.3 Aspect observationnel de la fonction d'état

Redéfinissons F_i comme suit:

$$F_i(n, c_1, \ldots, c_k) = f_i(F_1((n\text{-}1), c_1, \ldots, c_k), \ldots, F_k((n\text{-}1), c_1, \ldots, c_k))$$

$$= f_i(F((n\text{-}1), c_1, \ldots, c_k))$$

Alors,

$$F(n, c_1, \ldots, c_k) = (F_1(n, c_1, \ldots, c_k), \ldots, F_k(n, c_1, \ldots, c_k))$$

$$= (f_1(F((n\text{-}1), c_1, \ldots, c_k)), \ldots, f_k(F((n\text{-}1), c_1, \ldots, c_k)))$$

En tenant compte de l'état initial, F sera redéfinie plus précisément comme suit:

$F :: (Int, S) - S$

$$F(0, c_1^0, \ldots, c_k^0) = (c_1^0, \ldots, c_k^0)$$

$$F(n, c_1^0, \ldots, c_k^0) = \text{let } c_1^n = f_1(F((n\text{-}1), c_1^0, \ldots, c_k^0))$$

$$\vdots$$

$$c_k^n = f_k(F((n\text{-}1), c_1^0, \ldots, c_k^0))$$

$$\text{in } (c_1^n, \ldots, c_k^n)$$

La réécriture de F sous cette forme, révèle plusieurs avantages, en particulier:

- Elle convient très bien aux calculs parallèles. Toutes les coordonnées f_i opèrent en parallèle
- Elle convient très bien a la notion d'équivalence observationnelle [70] (très utile pour les systèmes complexes ou quelqu'un est intéressé uniquement a quelques observations parmi plusieurs autres).
- Elle convient très bien à l'approche de conception incrémentale. Si nous étendons la conception par des extra composants observables, nous aurons à définir uniquement quelques extra fonctions de transition.

3.2.1.4 Forme à une seule étape

Le comportement de F, après une seule étape, appelé : *la forme a une seule étape* coïncide avec la fonction de transition *f*, qui est redéfinie ci-après.

$f :: S - S$

$$f(c_1^0, \ldots, c_k^0) = \text{let } c_1^1 = f_1(c_1^0, \ldots, c_k^0)$$

$$\vdots$$

$$c_k^1 = f_k(c_1^0, \ldots, c_k^0)$$

$$\text{in } (c_1^1, \ldots, c_k^1)$$

Une telle forme est très importante pour la spécification des systèmes séquentiels qui exigent l'emploi d'une seule étape, telle que l'exécution d'une seule instruction.

3.2.2 Environnement fonctionnel

La notion de fonction est un outil suffisamment puissant pour spécifier les deux aspects comportemental et structural d'un circuit. La spécification comportementale est définie par l'architecture du jeu d'instructions. Une telle spécification constitue un ensemble d'assertions définissent l'effet de chaque opération fonctionnelle sur l'état visible a l'utilisateur. La spécification structurale décrit la microarchitecture implémentant le jeu d'instructions du circuit. Dans notre contexte, les deux spécifications : comportementale et structurale, seront modélisées en termes de *fonctions d'états* (représentant des machines d'état), dans un *environnement fonctionnel.*

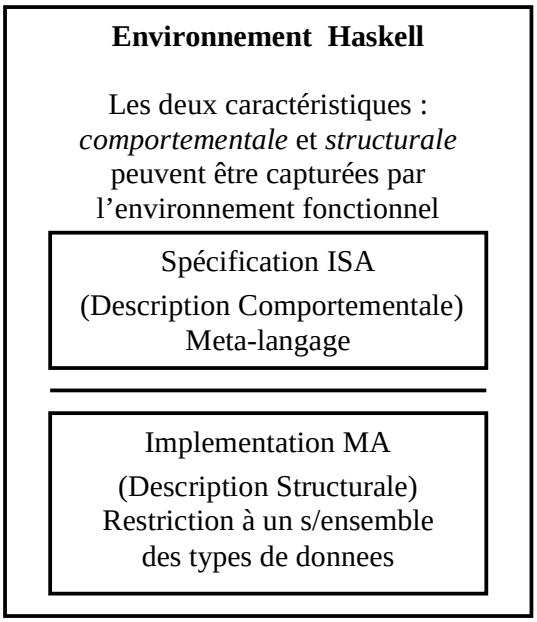


Figure 3.2 : Specifications Comportementale et structurale dans Haskell

La sémantique formelle de Haskell, permet de raisonner sur les deux spécifications en vue de prouver leur équivalence fonctionnelle. En outre, puisque les modèles des deux spécifications représentent des programmes fonctionnels, ils peuvent être exécutés pour comparer les résultats. Le processus de simulation peut être implémenté par une fonction d'ordre supérieure récursive acceptant en tant que paramètres : les fonctions modélisants le comportement et la structure, un état initial, et un nombre arbitraire de cycles d'horloges. Le résultat est une liste de tuples représentant les différents états produits par les deux modèles à comparer.

3.2.3 Architectures RISCs

Les architectures des processeurs peuvent être classées selon la complexité du jeu d'instructions en deux grandes familles:

3.2.3.1 **Les Architectures CISC**

Les architectures CISC (Complex Instruction Set Computers) qui sont issues de la longue histoire des ordinateurs possèdent un jeu d'instructions très complexes. Les instructions d'un processeur CISC ont un format variable (des instructions sur double mot, des instructions sur mot, des instructions sur demi mot, etc), et lisent souvent leurs opérandes dans la mémoire central. La fréquence d'occurrence de certaines instructions est extrêmement faible, ce qui pose le problème de la pertinence de leur existence. Par exemple, la gamme des microprocesseurs Motorola 680x0, et la gamme des microprocesseurs Intel x86, sont des processeurs CISC.

3.2.3.2 **Les Architectures RISC**

Les architectures RISC (Reduced Instruction Set Computers) sont nées des travaux de John Cocke sur le projet IBM 801 [71]. Ce projet était basé sur le fait que certaines instructions des processeurs CISC avaient un taux d'utilisation très faible qui ne justifiait pas le matériel mis en oeuvre pour les exécuter. L'idée de réaliser des processeurs simplifiés vit ainsi le jour. Ces processeurs possèdent un jeu d'instruction très simple avec un format fixe. L'accès aux opérandes est souvent limite aux registres. Une telle approche permet de tirer un meilleur parti du matériel. Tous les processeurs conçus après les années 80 furent de ce type. Par exemple, la gamme des processeurs IBM-Motorola PowerPC, la gamme SPARC de SUN, les processeurs ALPHA de Digital Equipment, et les processeurs MIPS sont des processeurs RISC.

Les architectures risques se distinguent en général par les caractéristiques suivantes: Des instructions simples avec peu de modes d'adressages, un format d'instruction fixe, et une architecture modulaire. Par suite, ils s'adaptent convenablement à la conception incrémentale: En d'autres termes, Les architectures RISC peuvent être hiérarchiquement construites du noyau implémentant le jeu d'instruction de base, aux architectures hautement optimisées [25], comme l'indique la figure 3.3. De cette manière, les architectures RISCs offrent l'avantage d'être plus simple à modéliser et plus simple à vérifier.

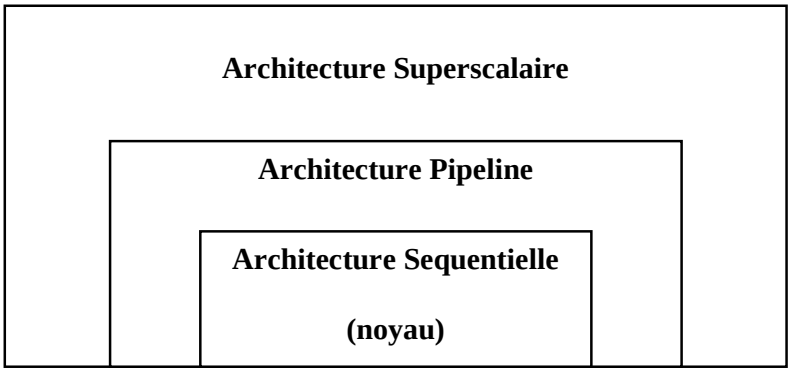


Figure 3.3 : Conception hiérarchique utilisant des architectures RISCs

3.3 Strategie de preuve

3.3.1 Modele hierarchique vertical-horizontal

Notre vue vis-à-vis de la vérification formelle des microprocesseurs suit le modèle vertical-horizontal en couches illustré par la figure 3.4. Le niveau ISA, représente la spécification de l'architecture du jeu d'instructions qui décrit la sémantique des opérations du processeur. Le niveau MA, représente le niveau top implémentant la spécification ISA. Il décrit les caractéristiques structurales des microarchitectures implémentant les opérations du processeur. Toutes les implémentations MA (qui peuvent être construites hiérarchiquement l'une sur l'autre) représentent différentes implémentations pour la même spécification ISA. Dans ce travail, nous nous intéressons à trois types d'implémentation: L'implémentation MA séquentiel (SMA), l'implémentation MA pipeline (PMA), et l'implémentation MA superscalaire (SSMA). L'implémentation SMA dont la preuve peut être effectuée facilement contre une spécification ISA, représente le noyau de référence sur lequel seront développées hiérarchiquement les implémentations PMA et SSMA, et contre lequel seront vérifiées également (contrairement aux approches alternatives où les implémentations PMA et SSMA sont prouvées contre une spécification ISA). Les couches inférieures représentent des raffinements successifs.

Dans notre contexte, toutes les implémentations MA sont modélisées en terme de fonctions d'état (représentant des machines d'état), au sein d'un environnement fonctionnel utilisant le langage de programmation Haskell.

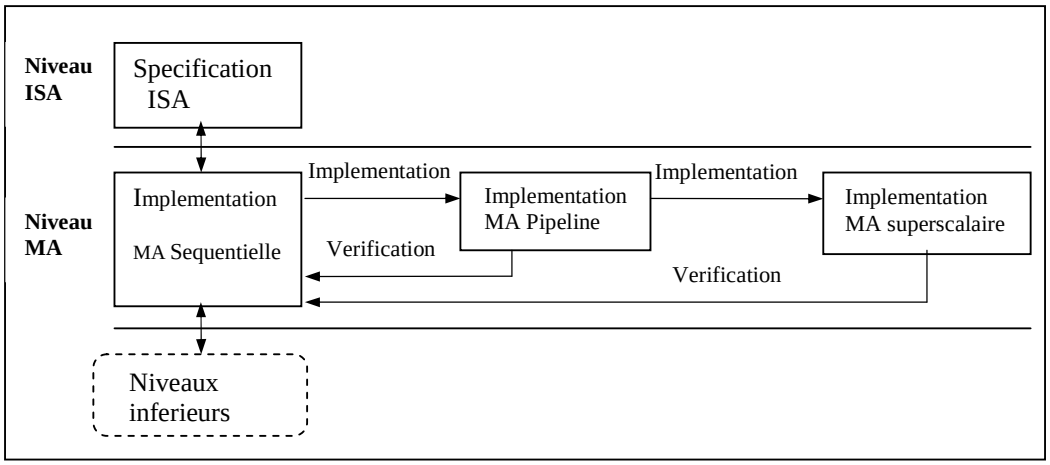


Figure 3.4 :.Modèle Vertical-Horizontal en couches

3.3.2 Etapes de preuve

La méthodologie utilise une approche incrémentale pour modéliser et vérifier des microarchitectures RISCs de complexité croissante. Pour satisfaire le diagramme de la figure 5, la méthodologie exige les étapes suivantes:

3.3.2.1 Vérification de la machine MA séquentielle (SMA)

La vérification formelle de l'implémentation microarchitecturale de la machine séquentielle est effectuée contre la spécification de l'architecture du jeu d'instruction de cette dernière. La différence de niveau exige l'utilisation d'une fonction d'abstraction de données qui transforme l'état d'implémentation à l'état de spécification. Dans notre contexte, la spécification ISA et l'implémentation MA seront formalisées en termes de fonctions d'état (représentant des machines d'état) dans un environnement fonctionnel. Par suite, le processus de vérification consiste en trois étapes:

- Développer la spécification ISA
- Développer l'implémentation MA
- Montrer que l'implémentation MA satisfait correctement la spécification ISA.

Une fois l'implémentation MA est vérifiée, elle sera considérée comme une spécification formelle contre laquelle d'autres implémentations du même niveau ou de niveau inférieure peuvent être vérifiées.

3.3.2.2 Vérification de la machine MA pipeline (PMA)

Contrairement aux approches alternatives qui vérifient une implémentation pipeline contre une spécification ISA, notre approche vérifie une implémentation pipeline contre une implémentation multi cycle séquentielle. Puisque les deux modèles sont formalisés en termes de cycles d'horloge, tous les états intermédiaires synchrones représentent des points utiles oŭ la comparaison peut être effectuée facilement. En effet, a chaque cycle d'horloge, une architecture pipeline avec S étages révèle S résultats partiels chaqu'un produit par une instruction séparée dans le pipe. Donc, on peut construire une variante du modèle séquentiel - appelé modèle séquentiel des composants - qui simule l'effet de calculer les mêmes résultats séquentiellement. Puisque les deux modèles (séquentiel et pipeline) se rapportent au niveau MA, la fonction d'abstraction de données n'est pas exigée, uniquement une fonction d'abstraction de temps est demandée pour synchroniser les états produits par ces deux derniers.

Le processus de vérification d'une implémentation pipeline passe par les étapes suivantes :

- Construire le modèle séquentiel des composants (sur le modèle séquentiel)
- Construire le modèle pipeline
- Vérifier le modèle pipeline contre le modèle séquentiel des composants.

3.3.2.3 Vérification de la machine MA superscalaire (SSMA)

Les architectures superscalaires étendent les architectures pipeline par la réplication des étages pipeline. Dans ce travail, nous limitons l'étude uniquement aux architectures a exécution ordonnée. De cette manière, une implémentation superscalaire sera construite facilement sur une implémentation pipeline et par suite, sera vérifiée aussi facilement contre une implémentation séquentiel. Donc, le processus de vérification d'une implémentation superscalaire passe par les étapes suivantes :

- Construire le modèle séquentiel des composants (sur le modèle séquentiel) relatif à un superscalaire
- Construire le modèle superscalaire (sur le modèle pipeline)
- Vérifier le modèle superscalaire contre le modèle séquentiel des composants.

3.4 Conclusion

L'utilisation de la notion de fonctions d'état permet de modéliser non seulement les systèmes déterministiques mais aussi les systèmes non déterministiques opérant dans des environnements ou le flux de données peut être perturbé par l'arrivée des événements non prédictibles tels que les exceptions et les interruptions.

L'adoption du modèle de conception hiérarchique vertical-horizontal permet de satisfaire le processus de vérification pour la microarchitecture séquentielle qui est prouvée contre la spécification ISA et pour les microarchitectures pipelines et superscalaires qui sont prouvées contre la microarchitecture séquentielle vérifiée.

L'utilisation d'un environnement fonctionnel permet de produire des modèles fonctionnels qui peuvent être utilisés en même temps pour la vérification formelle et la simulation (les systèmes réels sont validés par la combinaison de ces deux techniques).

Cependant, l'utilisation d'un environnement de réécriture (en particulier la réécriture de graphes fonctionnels [72]) est plus bénéfique pour le raisonnement formel mécanique.

Chapitre 4

Machines Séquentielles

Les architectures séquentielles constituent les noyaux fondamentaux sur lesquels des architectures pipelines et superscalaires sont construites. Il devient donc vital de valider formellement ces noyaux sur lesquels des architectures hautement optimisées sont bâties. Pour cette raison, le reste de ce chapitre sera entièrement consacré a la spécification et la vérification des architectures séquentielles multi cycles, qui seront prises dans ce travail, en tant que machines de référence contre lesquelles des architectures plus complexes seront formellement validées.

4.1 Spécification de la machine ISA

. Dans notre contexte, le plus haut niveau (d'abstraction) de la conception d'une machine constitue la spécification du jeu d'instructions (spécification ISA) qui décrit l'effet des instructions individuelles sur les états de la machine visibles au programmeur. Elle est complètement indépendante des détails de l'implémentation. Pour développer une spécification ISA, il est nécessaire de développer d'abord une spécification de l'étape ISA (ISA-step) qui décrit l'effet d'exécution d'une seule instruction (mais n'importe qu'elle instruction) à la fois.

4.1.1 Spécification de l'étape ISA

Au niveau spécification, une étape (step) corresponds toujours à l'exécution d'une seule instruction. Dans ce cas, la forme à une seule étape (the one step form) appelée : *fonction étape-ISA* (ISA-Step function), pourra être directement utilisée pour spécifier une instruction de la machine ISA. La spécification lit en premier lieu l'instruction à exécuter, et ensuite met à jour les états des composants visibles correspondants. Chaque état de composant sera calculé par une fonction coordonnée de l'etape-ISA, qui accepte en plus de l'état, l'instruction à exécuter (en utilisant uniquement l'état en tant qu'argument, exige de chaque fonction coordonnée la recherche de l'instruction à partir de la mémoire).

Soit *isa*::S – S, la fonction de l'étape-ISA, modélisant une instruction de la machine ISA sur l'état global S = (S_p, S_r, S_m), distribué sur trois composants observables : Le compteur ordinal (Program Counter), le fichier des registre (Registre file), et la mémoire de données. Chaque composant a une fonction coordonnée qui calcule son état:
isa = (isaPc, isaRf, isaMr), avec isaPc :: I – S – S_p, isaRf :: I – S – S_r, isaMr :: I – S – S_m
où I, est le type de l'instruction à exécuter.

Soit read :: Sm – Sp – I, la fonction de lecture, alors la définition de la fonction *isa* sur S, étant donné un état initial s_0 = (ps,rs,ms), est donnée ci-après.

```
isa :: S – S
isa(ps, rs, ms) = (ps', rs', ms')
   where  i   = read ms ps
    ps' = isaPc(i, ps, rs)
     rs' = isaRf(i, rs, ms)
     ms'= isaMr(i, rs, ms
```

Figure 4.1 : Spécification de l'étape-ISA en utilisant la clause *where*

On remarque que la spécification montre uniquement les états des composants observables auxquels on s'intéresse, pendant qu'elle cache l'instruction. Ceci est très important pour la preuve de correction qui sera effectuée en considérant uniquement l'aspect observationnel.

Si on remplace maintenant la clause "*where"* par l'expression "*let*", qui est très utile pour exprimer les aspects compositionnels et hiérarchiques, alors, la spécification de l'étape-ISA ci-dessus, réécrit comme suit:

```
isa :: S – S
isa(ps,rs,ms) =  let   i  =  read ms ps
                       ps' = isaPc(i, ps,rs,ms)
                    rs' = isaRf(i, ps,rs,ms)
                       ms' = isaMr(i, ps,rs,ms)
                    in  (ps',rs',ms')
```

Figure 4.2 : Spécification de l'étape-ISA en utilisant l'expression *let*

Pour pouvoir effectuer la preuve à différents niveaux hiérarchiques (en particulier au niveau instruction), il est nécessaire de développer une spécification hiérarchique révélant les niveaux hiérarchiques auxquels on s'intéresse. La figure 4.3 montre une spécification de l'étape-ISA équivalente avec trois niveaux hiérarchiques: Le niveau machine (*isa*), le niveau instruction (execs), et le niveau composant (isaPc, isaRf, isaMr).

```
isa:: S – S
isa(ps,rs,ms) =   let  i  =    read ms ps
                  in  (ps',rs',ms') = execs(i, ps,pr,pm)
 where,
execs(i, ps,rs,ms)  = let  ps'' = isaPc(i, ps,rs,ms)
                        rs'' =  isaRf(i, ps,rs,ms)
                           ms''  = isaMr(i, ps,rs,ms)
                    in  (ps'',rs'',ms'')
```

Figure 4.3 : Spécification hiérarchique de l'étape-ISA

En pratique, chaque fonction coordonnée exige uniquement quelques paramètres pour calculer l'état de sortie correspondant. Par exemple, pour calculer l'état du compteur ordinal, la fonction isaPC n'a pas besoin de l'état mémoire. Similairement, les fonctions isaRf et isaMr, n'ont pas besoin de l'état du compteur ordinal pour calculer l'état correspondant.

4.1.2 Spécification de la machine ISA

Une machine ISA est définie par une fonction d'état récursive qui calcule le résultat d'exécution de n instructions. Par exemple, la spécification d'une machine ISA définie sur l'état S = (S_1,…,S_k), distribué sur k composants, avec un état initial: (s_1,…s_k), et une fonction d'étape-ISA : isa = (isa_1,...,isa_k), où: isa_i ::I – S – S_i , est donnée ci-après:

ISA:: (Int, S) – S

ISA(0, s_1,…s_k) = (s_1,…s_k)

ISA(n, (s_1,…s_k)) = isa (ISA((n-1), s_1,…s_k))

Figure 4.4 : Spécification de la machine ISA

4.2 Implémentation de la machine MA séquentielle

4.2.1 Implémentation de l'étape-MA

4.2.1.1 Aspect compositionnel

Au niveau microarchitecture, la notion d'étape sera implémentée en termes de cycles d'horloges. Dans ce cas, la forme à une seule étape, appelée *fonction d'étape-MA*, se décompose horizontalement en un ensemble de fonctions comme suit:

$$ma = m_n \mathbf{o} \ldots \mathbf{o}\ m_1$$

Où m_i représente la fonction composante qui effectue (durant un cycle) la fonctionnalité de l'étage i pour $1 \leq i \leq n$. La figure 4.5 montre une telle décomposition.

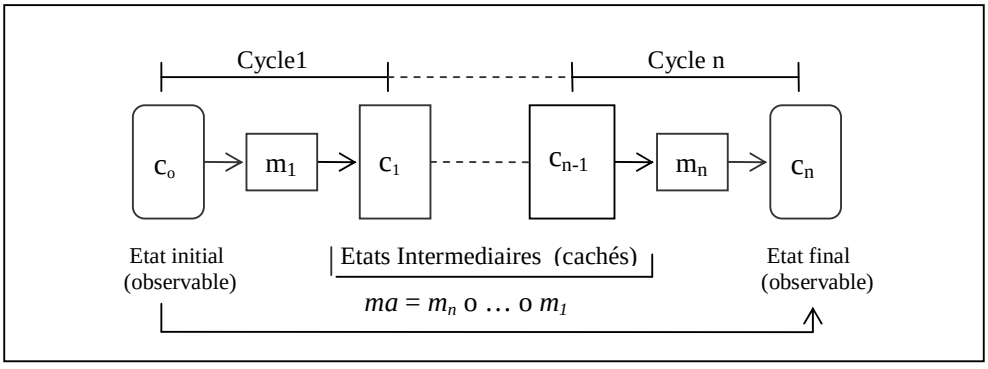


Figure 4.5 : Décomposition horizontale de la fonction d'étape-MA

Pour satisfaire la distribution de l'état, chaque fonction composante m_i, sera à son tour décomposée en coordonnes: $m_i = (m_i^1,\dots ,m_i^k)$, chaqu' une calcule un état de composant. Par suite, la fonction d'étape-MA, se décompose horizontalement et verticalement comme suit:

$$ma = (m_n^1,\dots,m_n^k)\ \mathbf{o}\dots\mathbf{o}\ (m_1^1,\dots,m_1^k)$$

la figure 4.6 illustre une telle décomposition. Toutes les opérations m_i^j listées dans un cycle d'horloge s'exécutent en parallel.

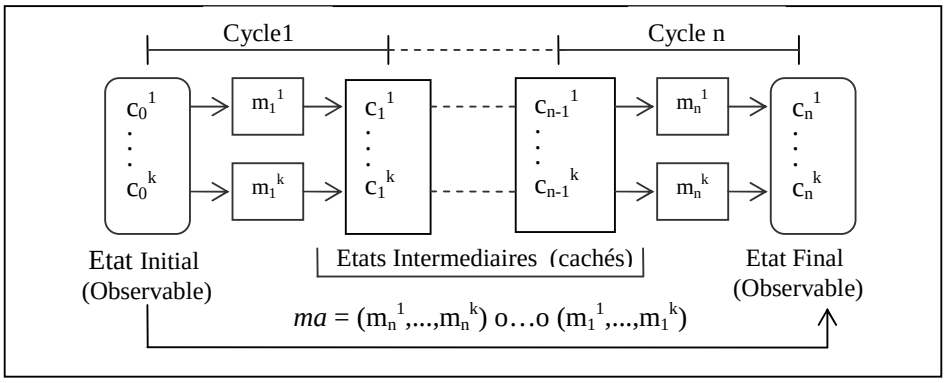


Figure 4.6 : Décomposition horizontale et verticale de fonction d'étape-MA

4.2.1.2 Implémentation Plate

La figure 4.7 montre comment le style fonctionnel peut satisfaire non seulement le parallélisme mais aussi l'aspect compositionnel des coordonnées de la fonction d'etape-MA. La première partie "*let*" permet de calculer en parallèle tous les états intermédiaires durant un cycle donné, alors que la seconde partie "*in*" permet de les utiliser (par les fonctions composantes ultérieures), au cycle d'horloge suivant. En outre, en capturant l'aspect synchrone de l'implémentation MA, cette forme a le mérite de modéliser le chemin de données et le contrôle, et par suite, elle permet de simuler d'une manière précise le rôle d'un processeur. [73]

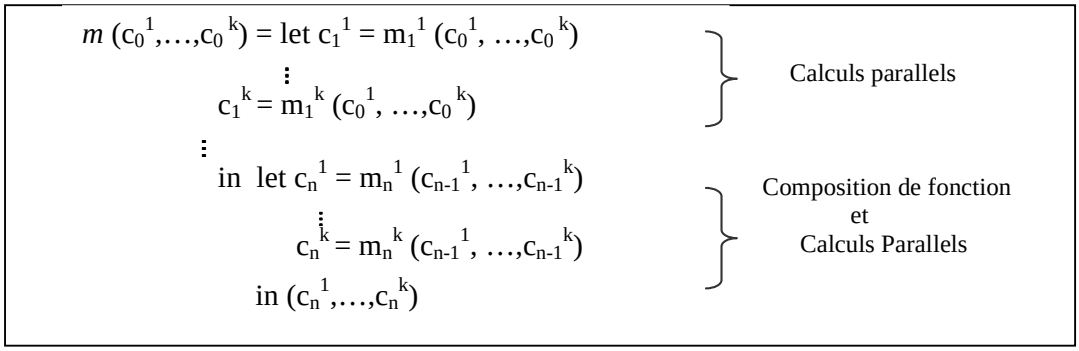


Figure 4.7. Implémentation plate de l'étape-MA pour une machine multi cycle

Cependant, si une spécification plate a le mérite de satisfaire l'aspect synchrone (et par conséquent, satisfaire l'aspect simulation), elle reste d'autre part moins intéressante vis-à-vis de la preuve de correction, dans la mesure où elle révèle uniquement le niveau machine. Par suite, une spécification hiérarchique est plus intéressante.

4.2.1.3 Implémentation hiérarchique

Au niveau MA, une instruction est recherchée à partir de la mémoire en tant que liste de bits, décodée en tant que liste de champs et ensuite exécutée. Les cycles de recherche et de décodage sont communs à toutes les instructions, alors que le reste des cycles sont spécifiques à chaque instruction.

Soit *ma* la fonction d'étape-MA modélisant la machine MA sur l'état global W=(Wp,Wr,Wm), distribué sur trois composants observables: le compteur ordinal, le fichier registre et la mémoire. Supposons aussi que, *ma* soit décomposée horizontalement en trois fonctions: *ma* = *ex_ma* **o** *decode* **o** *fetch*

- *fetch:* décrit la portion du chemin de données implémentant la fonctionnalité de la phase de recherche
- *decode:* décrit la portion du chemin de données implémentant la fonctionnalité de la phase de décodage
- *ex_ma:* décrit la portion du chemin de données implémentant la fonctionnalité de la phase d'exécution

Supposons que ex_ma soit décomposée verticalement comme suit: ex_ma = (maPc, maRf, maMr)

- maPc: décrit la portion du chemin de données qui met a jour le compteur ordinal.
- maRf: décrit la portion du chemin de données qui met a jour le fichier registre
- maMr: décrit la portion du chemin de données qui met a jour la mémoire.

Maintenant, nous pouvons construire la spécification MA hiérarchique impliquant trois niveaux d'hiérarchie: Le niveau machine (ma), le niveau instruction (ex_ma) et le niveau composant observable (maPc, maRf, maMr) comme indiqués dans la figure 4.8.

```
ma:: W – W
ma (pw,rw,mw)  = let  i  =   fetch mw pw
                        in let  di   =  decode(i)
                            in (pw',rw',mw') = ex_ma(di,pw,rw,mw)
where,
 ex_ma(di,pw,rw,mw) =let pw'= maPc(di,pw,rw,mw)
                                rw' = maRf(di,pw,rw,mw)
                               mw'= maMr(di,pw,rw,mw)
                             in (pw',rw',mw')
```

Figure 4.8: Implémentation MA hiérarchique

4.2.2 Implémentation de la machine MA

Une machine MA séquentielle sera définie par une fonction d'état récursive qui retourne l'état MA après l'exécution de n instructions. Par exemple, la spécification d'une machine SMA sur l'état $W=(W_{1,...}W_k)$, distribué sur k composants, étant donnés un état initial $(c_0^1,\ldots,c_0^k)$, et une fonction d'étape MA: ma::W – W, est donnée ci-dessous:

SMA:: (Int, W) – W

SMA(0, $c_0^1,\ldots,c_0^k$) = ($c_0^1,\ldots,c_0^k$)

SMA (n, ($c_0^1,\ldots,c_0^k$)) = ma(SMA((n-1), $c_0^1,\ldots,c_0^k$))

Figure 4.9 : Implémentation de la machine MA

4.3 Critère de correction de l'etape-MA

4.3.1 Critère de correction

Supposons maintenant, que les deux machines: étape-ISA et étape-MA démarrent du même état initial et effectuent une étape pour exécuter la même instruction.

Informellement parlant, nous pouvons dire que la machine étape-MA exécute correctement cette instruction, ou plus précisément implémente correctement cette instruction, si elle donne le même état final que la machine étape-ISA, quant elle démarre du même état que cette dernière.

Prouver formellement la correction de l'étape-MA vis-à-vis de l'étape-ISA revient à prouver la propriété commutative suivante:

$$\forall\ pw :: Wp,\ rw :: Wr,\ mw :: Wm$$

$$abs(\mathbf{ma}(pw,rw,mw)) = \mathbf{isa}(abs(pw,rw,mw)) \quad (1)$$

Cette propriété reflète le diagramme commutatif suivant:

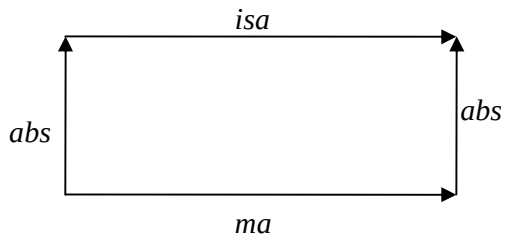


Figure 4.10 : Critère de Correction de l'étape-MA

Où, *abs*::W – S, est une fonction d'abstraction qui convertit l'état MA à l'état ISA.

4.3.2 Décomposition de la preuve

Si la preuve réussit, elle signifie tout simplement que l'étape-MA est correcte vis-à-vis de l'étape-ISA. Cependant, si la preuve échoue, il n'y'a aucun moyen de procéder au débogage de la machine MA. Dans ce cas, un raffinement de la preuve est nécessaire.

Soit *ai* :: Wi – Si, une fonction d'abstraction qui convertit l'instruction décodée du niveau MA au niveau ISA, telque: i = *ai*(di), ∀ di :: Wi. Alors, les deux cotés de l'équation (1) réécrivent selon les spécifications de l'étape-MA et l'étape-ISA, comme suit:

$$abs(ma(pw, rw.\ mw)) = abs(ex\_ma(di,pw,rw,mw))$$

$$isa(abs(pw,rw,mw)) = ex\_isa(ai(di),abs(pw\ rw,mw))$$

Donc, la preuve de l'équation (1), implique la preuve de l'équation suivante qui permet de prouver la machine MA au niveau instruction.

$$\forall\ di :: Wi,\quad pw::Wp,\quad rw:: Wr,\quad mw::Wm$$

$$abs(ex\_ma(di,pw,rw,mw)) = ex\_isa(ai(di), abs(pw,rw,mw)) \quad (2)$$

L'aspect hiérarchique des spécifications permet encore de raffiner cette équation jusqu'au niveau composant observable. Supposons que *abs est* décomposée comme suit:

$$abs(pw,rw,mw) = (ap(pw), ar(rw), am(mw))$$

où, ap:: Wp – Sp, ar:: Wr – Sr, am:: Wm – Sm

Alors, les deux cotes de l'équation (2) réécrivent comme suit:

$$abs(ex\_ma(di,pw,rw,mw)) = [\ ap(maPc(di,pw,rw,mw)),\ ar(maRf(di,pw,rw,mw)),\ am(maMr(di,pw,rw,mw))]$$

$$ex\_isa(a_i(di),abs(pw,rw,mw)) = [\ isaPc(\mathbf{ai}(di), \mathbf{ap}(pw), \mathbf{ar}(rw), am(mw)),\ isaRf(ai(di), ap(pw), ar(rw), am(mw)),\ isaMr(ai(di), ap(pw), ar(rw), am(mw))]$$

Ainsi, la preuve de l'équation (2), se décompose en sous preuves des équations suivantes qui permettent de prouver la machine MA au niveau composant observable.

$$\forall\ di :: Wi,\quad pw::Wp,\quad rw:: Wr,\quad mw::Wm \quad (3)$$

$$ap(maPc(di, pw,rw,mw)) = isaPc(ai(di), ap(pw), ar(rw), am(mw))$$

$$\wedge\ ar(maRf(di, pw,rw,mw)) = isaRf(ai(di), ap(pw), ar(rw), am(mw))$$

$$\wedge\ am(maMr(di, pw,rw,mw)) = isaMr(ai(di), ap(pw), ar(rw), am(mw))$$

4.3.3 Discussion

- Nous avons développé le critère de correction sur trois observables uniquement. Cependant, la méthode est suffisamment générale pour manipuler un nombre arbitraire d'observables.
- L'équation (2) et l'ensemble des équations (3), supposent que l'instruction décodée:

$$di = decode(fetch\ mw\ pw)$$

a été recherche et décodée d'une manière correcte. Cette supposition reste vrai si la preuve réussit. Cependant, si la preuve échoue, la condition de vérification suivante doit être satisfaite aussi.

$$\forall\ mw :: Wm,\ \ pw :: Wp \quad (4)$$
$$ai(decode(fetch\ mw\ pw)) = read(am\ (mw)\ \ (ap(pw))$$

- Pour chaque instruction, le nombre d'équations à prouver, dépend du nombre des observables

- Si uniquement le compteur ordinal est observable, la preuve se réduit à une seule équation:

$$ap\ (maPc\ (di, pw)) = isaPc\ (ai(di), ap(pw))$$

- Si, à coté du compteur ordinal, le fichier des registres est aussi observable, alors, il faut prouver deux équations:

$$ap\ (maPc\ (di, pw,rw,)) = isaPc(ai(di), ap(pw), ar(rw))$$
$$\wedge ar\ (maRf\ (di, pw,rw)) = isaRf(\ ai(di), ap(pw), ar(rw))$$

- Si nous avons trois observables, il faut prouver trois équatios (3), et ainsi de suite.

- L a décomposition de la preuve en un ensemble de conditions de vérifications permet de raisonner sur chaque cas séparément.
- Chaque instruction a ses propres conditions de vérifications, par conséquent, la preuve peut procéder d'une manière incrémentale.

4.4 Critère de correction de la machine MA

4.4.1 Critère de correction

Prouver l'équivalence fonctionnelle de la machine MA vis-à-vis de la machine ISA, exige prouver la condition de vérification suivante:

$$\forall\ pw ::Wp,\ \ rw :: Wr,\ \ mw :: Wm$$
$$ISA(n, abs(pw,rw,mw)) = abs(MA(n, pw,rw.mw)) \quad (5)$$

4.4.2 Preuve de correction

La preuve qui sera effectuée par induction sur le nombre des étapes, suppose que la propriété de vérification exigée par la forme a une seule étape est satisfaite.

- **Cas 0**

ISA(0, abs(pw,rw,mw)) = abs(pw,rw,mw) definition de ISA
abs(MA(0, pw,rw.mw)) = abs(pw,rw,mw) definition de MA
Ce qui verifie le cas 0

- **Cas n+1**

Supposons que pour l'étape n, l'équation (5), est vrai, et essayons de la prouver pour l'étape n+1, également.

ISA((n+1), abs(pw,rw,mw)) = isa(ISA(n, abs(pw,rw,mw)) définition de ISA
= isa (abs(MA(n, pw,rw.mw)) induction case
= abs(ma(MA(n, pw,rw.mw))) propriété (1)
= abs(MA((n+1), pw,rw,mw)) définition de MA

Ce qui vérifie le cas n+1, aussi

Cela signifie que pour démontrer la correction de la machine MA séquentielle pour n etapes, il est suffisant de la démontrer pour une seule étape. Il est inutile d'aller au delà d'une seule étape parce que le même diagramme commutatif sera répété infiniment. En définissant les fonctions de transition d'étape ISA et d'étape MA par analyse de cas, les conditions de vérification de toutes les instructions peuvent être captures en une seule étape.

4.5 Application

Cette section montre comment la méthodologie proposée peut être appliquée pour spécifier, implémenter et vérifier d'une manière systématique la machine séquentielle MIPS [74]. La spécification des deux machines: ISA et MA, ainsi que la preuve de leur équivalence fonctionnelle seront effectuées au sein de l'environnement Haskell.

4.5.1 Spécification ISA de la machine MIPS

La spécification du jeu d'instructions de la machine MIPS sera limitée a trois types d'instructions: L'instruction arithmétique *Add*, les instructions de référence mémoire : *Load* et *Store*, et l'instruction de branchement conditionnelle: *Beq.* Leurs descriptions informelles sont données ci-après.

Syntax	**Semantics**
Add rd rs rt	regfile[rd] := regfile[rs] + regfile[rt]; pc:= pc +1
Load rs rt im	regfile[rt] := dmemory([rs] + im]); pc:= pc +1
Store rs rt im	dmemory([rs] + im]) := regfile[rt]; pc:= pc +1
Beq rs rt im	If (regfile[rs] = regfile[rt]) **then** pc:= pc + im **else** pc := pc +1

Table 4.1 : Description informelle du jeu d'instructions de la machine MIPS sequentielle

Elles sont introduites par le constructeur de données data comme suit:

data Sinstruction = Add Rd Rs Rt | Load Rs Rt Im | Store Rs Rt Im | Beq Rs Rt Im

L'espace d'état sera distribué sur quatre composants observables: Le compteur ordinal, le fichier des registres, la mémoire de données et la mémoire d'instructions. A ce niveau, ces composants seront modélisés en tant que listes. Deux opérations primitives *read* et *write* seront utilisées pour accéder à ces structures de données.

read :: [Data] – Address – Data
write :: [Data] – Address – Data – [Data]

En utilisant la fonction étape-ISA, la spécification ISA du processeur MIPS sera construite comme suit :

```
type PC = Address
type RegFile = [Data]
type Dmemory = [Data]
type Imemory = [Sinstruction]
typeSstate = (PC, RegFile, Dmemory, Imemory)
isa :: Sstate – Sstate
isa(ps, rs, ms, is)   = let i  = read is ps
                  in  (ps', rs', ms', is) = ex_isa(i,ps,rs,ms)
where
ex_isa(i, ps,rs,ms)  = let  ps' = isaPc(i, ps, rs)
                        rs' =  isaRf(i, rs, ms)
                         ms'= isaMr(i, rs, ms)
                            in (ps',rs',ms')
```

Figure 4.11: Spécification de l'étape-ISA du processeur MIPS

4.5.2 Spécification MA de la machine MIPS

4.5.2.1 Microarchitecture du Processeur

La figure 4.12 montre la microarchitecture simplifiée du processeur MIPS, implémentant le jeu d'instructions spécifié dans la section précédente. Son chemin de données est divisé en cinq étages: R*echerche, décodage, exécution, accès mémoire, écriture.* Les étages de *recherche* et de *décodage* sont communs à toutes les instructions, alors que le reste des étages sont spécifiques à chaque instruction. Chaque étage effectue sa fonctionnalité en un cycle d'horloge.

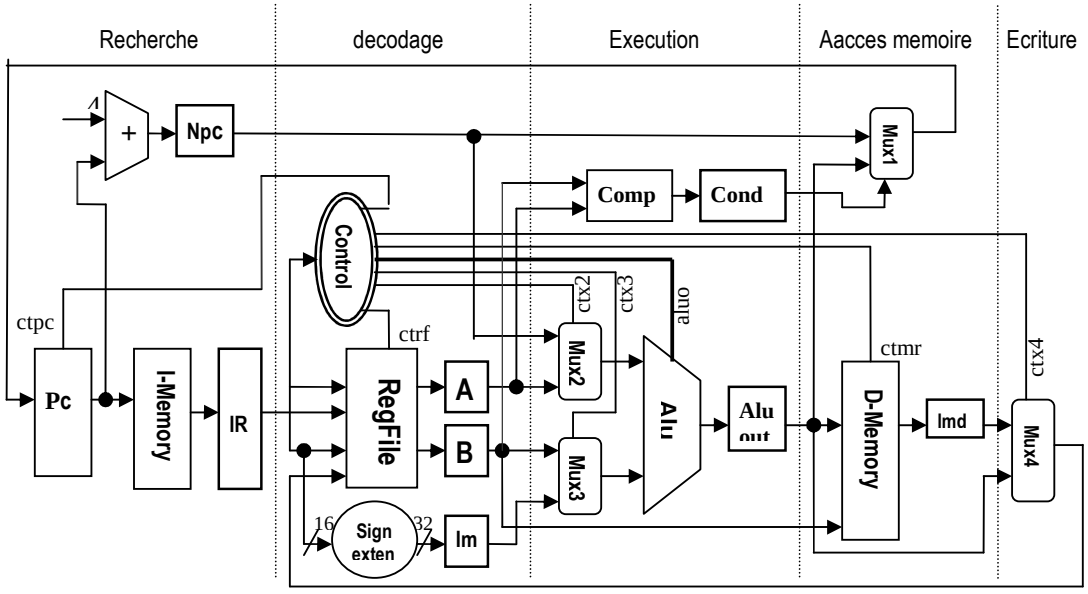


Figure 4.12: Microarchitecture simplifiée du processeur MIPS

La fonctionnalité de chaque étage est informellement décrite dans la table donnée ci-dessous.

Recherche	Decodage	Execution	Acces Memoire	Ecriture
Fetch instruct. IR = Imem[Pc] **Increment PC** Npc = Pc + 4	**Decode instruct** **Fetch registers** A = RegFile[rs] B = RegFile[rt] **Sign_extend im** Im = S_Ext[im]	*Case instruction of* **add** Aluout = A + B **Load** Aluout = A + Im **store** Aluout = A + Im **Beq** Aluout = Npc + Im Cond = (A op B)	**Bypassed** **Load** Lmd = Dmem[Aluout] **Store** Dmem[Aluout] = B **Beq** If Cond then Pc = Aluout else Pc = Npc	**add** Regfile[rd] = Aluout **Load** Regfile[rt] = Lmd

Table 4.2: Description informelle des étages du processeur MIPS.

Cette étude de cas considère l'implémentation de trois types d'instructions MIPS: Une instruction de type Registre (add), deux instructions de type référence mémoire (load et store), et une instruction de type branchement (beq). Leurs formats sont schématisés dans la table 4.3. La sémantique des différents champs est donnée ci-après.

Type-R: *rs* et *rt* sont les registres sources et *rd* est le registre destination. *Shamt* et *funct* sont les champs des opérations de décalage et d'Alu respectivement. Dans notre cas, les deux champs seront ignorés (le code d'opération est suffisant pour exécuter les quatre instructions: add, lw, sw, et beq).

Type-M: *rs* est le registre de base, *rt* est le registre source pour store, et le registre destination pour load, *im* est la valeur offset.

Type-B: *rs* et *rt* sont les registres sources et *im* est la valeur offset.

	31 26	25 21	20 16	15 11	10 6	5 0
R-type	Cop (add)	Rs	Rt	Rd	shamt	Funct
M-type	Cop (lw/sw)	Rs	Rt	Im		
B-type	Cop (beq)	Rs	Rt	Im		

Table 4.3: Format des instructions MIPS

4.5.2.2 Spécification des composants MA du MIPS

La sémantique en termes de circuits du niveau MA, exige de la spécification MA de prendre en compte les caractéristiques physiques des composants de ce niveau. Au niveau MA, l'information est interprétée en termes de bits et les différents composants sont interprétés en termes de circuits digitaux. Chaque composant mémoire possède un "latch" qui mémorise son état. Les différents composants MA du processeur MIPS sont spécifiés fonctionnellement (curried functions) ci-dessous.

- Un compteur ordinal implémenté en tant que registre mot. Son interface est définie comme suit: *regPC :: RW – Address – Address*
- une mémoire d'instructions implémentée en tant que mémoire a lecture seule. Son interface est définie comme suit: *imem :: Address – Instruction*
- Un décodeur d'instruction spécifié comme suit:

  decode::Instruction – (Field,Field,Field,Field,Field)

  decode(i) = (cop, rs ,rt, rd, im)
- Un fichier registre à deux ports pouvant accepter en parallèle deux adresses sources pour donner deux sorties. Son interface accepte deux signaux de contrôle: Un signal RS qui permet de lire soit une paire de données soit l'état entier (lors d'une preuve) du fichier registre, et un signal RW qui contrôle les opérations de lecture/ecriture:

regFile :: RS – RW – Address – Address – Address – Data – [Data]

- Une ALU intégrant la majorité des opérations arithmétiques et logiques (d'autres approches simulent une telle ALU par un ensemble de fonctions non interprétées [7]). Son interface est spécifié comme suit: *alu :: Aluop – Operand – Operand – Result*
- Une mémoire de données qui offre la possibilité de lire soit un élément de donnée soit l'état entier. Son interface est spécifié comme suit: *dmem::RS – RW – Address – Data – [Data]*
- Un additionneur mot séparé pour incrémenter le compteur ordinal. Son interface est spécifié comme suit: *adder::Word – Word – Bit – (Word, Bit)*
- Un comparateur mot pour tester le contenu des registres A et B. Il est défini comme suit: *Comp :: Word – Word – Bit*
- Un ensemble de multiplexeurs mots pour le partage des ressources communes. L'interface multiplexeur est définie comme suit: *mux :: Bit – Word – Word – Word.*

Finalement, l'unité de contrôle qui accepte le code de l'opération et l'état courant et génère en sortie l'état suivant et un ensemble de signaux de contrôle pour exécuter l'instruction.

Control ::Cop – Cstate – (Nstate, Aluop, Ctrl,Ctrl, Ctrl, Ctrl, Ctrl, Ctrl, Ctrl, Ctrl, Ctrl ,Ctrl)

Une implémentation possible des types utilisés est décrite dans la table ci-après.

type RS = Bit type RW = Bit type Ctrl = Bit	type Word = [Bit] type Data = Word type Address = Word type Instruction = Word	type Operand = Word type Result = Word type Field = Word	type Cop = Field type Aluop = Field type Cstate = Field type Nstate = Field

Table 4.4: types synonymes utilisés dans la spécification MA séquentielle du processeur MIPS

4.5.2.3 Spécification MA plate

Maintenant, la spécification MA de la machine MIPS (chemin de données et contrôle) implémentant les quatre instructions décrites auparavant peut être construite comme suit: Soit *ma* la fonction d'étape-MA modélisant la microarchitecture de la machine MA, avec l'espace d'état distribué sur quatre composants observables: Le compteur ordinal, le fichier registre, la mémoire de données et la mémoire des instructions. Pour satisfaire la sémantique microarchitecturale, nous définissons la fonction *ma* en tant que fonction d'ordre supérieure qui accepte un ensemble de fonctions et qui retourne un ensemble de fonctions. En d'autres termes, elle accepte un ensemble de composants physiques (modélisés en tant que fonctions) et retourne un ensemble de composants physiques avec leurs états à jour (la mémoire des instructions reste inchangée). Ce nouveau style de spécification (l'approche classique accepte directement l'état en tant que structure de données) simule précisément le rôle du processeur. La spécification plate complète est donnée ci-après.

```
type Pc = a → [a] → [a]
type Regfile = a → a → [a] → [a] → [a] → [a] → [[a]]
type Dmem = a → a → [a] → [a] → [[a]]
type Imem = [a] → [a]
type State  =  (Pc, Regfile, Dmem, Imem)
```

```
ma :: Mstate → Mstate
ma(pc ,regf, ram, rom)  =
==========================================================*Fetch cycle*
let (decodst, aluop, fs, fr, ms, mr, pc_r, pc_w, s_npc, ctx2, ctx3, ctx4) = control(anyop, fetchst)
                                    ir   = rom (pc  pc_r  anyA)
                                  (carry, npc) =  adder (pc  pc_r  anyA) four Zero
                                        pc' =  pc  pc_w  (mux0 s_npc npc anyA)
==========================================================*Decode cycle*
in let (dispachst, aluop, fs, fr ms, mr, pc_r, pc_w, ctx0, ctx2, ctx3, ctx4)=control(anyop, decodst)
                                  (cop, rs, rt, rd, im) =  decoder(ir)
                                              [ra,rb] =  regf fs fr rs rt anyA anyD
                                                 ri  =  signExtend(im)
 ——————————————————————————————————————————————————————— Execute cycle
in case cop of
add → let (addst, aluop, fs, fr, ms, mr, pc_r, pc_w, ctx0, s_ra, s_rb,ctx4)=control(addop,dispachst)
        aluout = alu aluop (mux2 s_ra npc ra)(mux3 s_rb rb ri)
-------------------------------------------------------------------------------------------------*Write Back cycle
in let (fetchst, aluop, fs, fw, ms, mr, pc_r, pc_w, ctx0, ctx2, ctx3, s_aluout) = control(addop, addst)
      regf'= regf fs fw anyA anyA rd (mux4 s_aluout lmd aluout)
   in  (pc', regf', ram, rom)
——————————————————————————————————————————————————————— Execute cycle
lw → let (lwst1, aluop, fs, fr, ms, mr, pc_r, pc_w, ctx0, s_ra, s_ri, ctx4) = control(lwop, dispachst)
  aluout =alu aluop (mux2 s_ra npc ra)(mux3 s_ri rb ri)
------------------------------------------------------------------------------------------- Memory access cycle
in let (lwst2, aluop, fs, fr, ms, mr, pc_r, pc_w, ctx0, ctx2, ctx3, ctx4) = control(lwop, lwst1)
  lmd = ram ms mr aluout rb
-------------------------------------------------------------------------------------------------*Write Back cycle
in let (fetchst, aluop, fs, fw ms, mr, pc_r, pc_w, ctx0, ctx2, ctx3, s_lmd) = control(lwop, lwst2)
  regf'= regf fs fw anyA anyA rt (wmux4 s_lmd lmd aluout)
    in  (pc', regf', ram, rom)
——————————————————————————————————————————————————————*Execute cycle
sw → let (swst, aluop, fs, fr, ms, mr, pc_r, pc_w, ctx0, s_ra, s_ri, ctx4) =control(swop, dispachst)
  aluout =alu aluop (mux2 s_ra npc ra)(mux3 s_ri rb ri)
------------------------------------------------------------------------------------------- Memory access cycle
in let (fetchst, aluop, fs, fr ms, mw, pc_r, pc_w, ctx0, ctx2, ctx3, ctx4) = control(swop, swst)
  ram' = ram ms  mw  aluout  rb
   in  (pc', regf, ram', rom)
——————————————————————————————————————————————————————— Execute cycle
beq → let (beqst, aluop, fs, fr, ms, mr, pc_r, pc_w, ctx0, s_npc, s_ri, ctx4)=control(beqop, dispachst)
    aluout = alu aluop (mux2 s_npc npc ra)(mux3 s_ri rb ri)
    cond = comp ra rb
-------------------------------------------------------------------------------------------------*Memory access cycle
in let (fetchst, aluop, fs, fr, ms, mr, pc_r, pc_w, s_mux1, ctx2, ctx3, ctx4) = control(beqop, beqst)
   regPc' = regPc pc_w  (mux0 s_mux1 npc (mux1 cond  npc  aluout))
   in  (pc', regf, ram, rom)
```

Figure 4.13 : Spécification MA plate du processeur MIPS

Comme nous pouvons le remarquer, l'aspect synchrone de l'implémentation MA est capturé d'une manière élégante. A chaque cycle d'horloge, la logique de contrôle envoie les signaux de contrôle nécessaires pour réaliser la fonctionnalité de l'étage correspondant.

4.5.2.4 Spécification MA hiérarchique

Maintenant, la spécification hiérarchique peut être facilement construite à partir du modèle développé dans la section 4.2.1.3. La figure 4.14 montre explicitement trois niveaux d'hiérarchies. Le niveau machine (ma), le niveau instruction (ex_ma), et le niveau composant (maPc, maRf, maMr). Puisque l'aspect synchrone n'est pas capturé, les signaux de contrôle sont simulés d'une manière asynchrone.

```
ma :: Mstate – Mstate
ma(pc ,regf, ram, rom) =
----------------------------------Instruction Fetch
      let    ir  = rom (pc  pc_r  anyA)
--------------------------------Instruction Decode
     in  let  (cop, rs, rt, rd, im) =  decoder(ir)
----Execute-----Memory Access-----Write Back
         in case cop of
add – let (pc', regf ', ram') =
  ex_ma((add, rs, rt, rd), pc, regf, ram)
  in (pc', regf ', ram',rom)
lw  – let (pc', regf ', ram') =
  ex_ma((lw, rs, rt, im), pc, regf, ram)
  in (pc', regf ', ram',rom)
sw – let (pc', regf ', ram') =
  ex_ma((sw , rs, rt, im), pc, regf, ram)
  in (pc', regf ', ram',rom)
beq – let (pc', regf', ram') =
   ex_ma((beq , rs, rt, im), pc, regf, ram)
  in (pc', regf ', ram',rom)
```

```
where,
ex_ma(add, rs, rt, rd, pc, regf, ram) =
  let   pc'  =  maPc((add, rs,rt,rd),pc,regf)
       regf' = maRf((add, rs,rt,rd),regf,ram)
        ram' = maMr((add,rs,rt,rd),regf,ram)
   in  (pc', regf', ram')
ex_ma(lw, rs, rt, im, pc, regf, ram) =
    let  pc'  =  maPc((lw, rs,rt,im),pc,regf)
         regf' = maRf((lw, rs,rt,im),regf,ram)
         ram' = maMr((lw,rs,rt,im),regf,ram)
     in  (pc', regf', ram')
ex_ma(sw, rs, rt, rd, pc, regf, ram) =
    let  pc'  =  maPc((sw, rs,rt,im),pc,regf)
         regf' = maRf((sw, rs,rt,im),regf,ram)
         ram' = maMr((sw,rs,rt,im),regf,ram)
     in  (pc', regf', ram')
ex_ma(beq, rs, rt, rd, pc, regf, ram) =
  let   pc'  =  maPc((beq, rs,rt,im),pc,regf)
       regf' = maRf((beq, rs,rt,im),regf,ram)
       ram' = maMr((beq,rs,rt,im),regf,ram)
   in  (pc', regf', ram')
```

Figure 4.14. Spécification hiérarchique de l'implémentation MA du processeur MIPS

Les spécifications des composants observables ISA et de leurs implémentations MA correspondantes sont illustrées dans la figure 4.15 donnée ci-après.

Specification des composants ISA observables	Specification des composants MA observables
isaPc :: Sinstruction ─ Spc ─ SregFile ─ Spc **isaPc**(Add r1 r2 r3,ps,rs) = ps +1 **isaPc**(Load r1 r2 im,ps,rs)=ps +1 **isaPc**(Store r1 r2 im,ps,rs)= ps+1 **isaPc**(Beq r1 r2 im, ps, rs) = let v1 = read rs r1 v2 = read rs r2 npc = ps + 1 in ps' = if not (v2 = = v1) then npc else (npc + im)	**maPc** :: Minstruction ─ Mpc ─ MregFile ─ Mpc **maPc**((add, rs,rt,rd),pc,regf) = let (carry, npc) = adder (pc pc_r anyA) four Zero in pc' = pc pc_w (mux0 s_npc npc anyA) **maPc**((lw, rs,rt,im),pc,regf) = let (carry, npc) = adder (pc pc_r anyA) four Zero in pc' = pc pc_w (mux0 s_npc npc anyA) **maPc**((sw, rs,rt,im),pc,regf) = let (carry, npc) = adder (pc pc_r anyA) four Zero in pc' = pc pc_w (mux0 s_npc npc anyA) **maPc**((beq, rs,rt,im),pc,regf) = let (carry, npc) = adder (pc pc_r anyA) four Zero (ra,rb) = regf fs fr rs rt anyA anyD ri = signExtend(im) aluout=alu aluop (mux2 s_ra npc ra)(mux3 s_ri rb ri) cond = compw ra rb in pc'= pc pc_w (mux0 s_x1 npc (mux1 cond npc aluout))
isaRf::Sinstruction ─ SregFile ─ Sdmem ─ SregFile **isaRf**(Add r1 r2 r3, rs, ms) = let v2 = read rs r2 v3 = read rs r3 v1 = v2 + v3 in rs' = write rs r1 v1 **isaRf**(Load r1 r2 im, rs, ms) = let v1 = read rs r1 a = v1 + im v = read ms a in rs' = write rs r2 v **isaRf**(Store r1 r2 im, rs, ms) = rs **isaRf**(Beq r1 r2 im, rs, ms) = rs	**maRf**::Minstruction ─ MregFile ─ Mdmem ─ MregFile **maRf**((add, rs,rt,rd),regf, ram) = let [ra,rb] = regf fs fr rs rt anyA anyD aluout = alu aluop (mux2 s_ra npc ra)(mux3 s_rb rb ri) in regf '=regf fs fw anyA anyA rd (mux4 s_aluout lmd aluout) **maRf**((lw, rs,rt,im),regf, ram) = let [ra,rb] = regfile fs fr rs rt anyA anyD ri = signExtend(im) aluout =alu aluop (mux2 s_ra npc ra)(mux3 s_ri rb ri) lmd = ram ms mr aluout rb in regf '= regf fs fw anyA anyA rt (mux4 s_lmd lmd aluout) **maRf**((sw, rs,rt,im),regf, ram) = regf **maRf**((beq, rs,rt,im),regf, ram) = regf
isaMr::Sinstruction ─ SregFi ─ Sdmem ─ Sdmem **isaMr**(Add r1 r2 r3, rs,ms)= ms **isaMr**(Load r1 r2 im,rs,ms)=ms **isaMr**(Beq r2 im, rs, ms) = ms **isaMr**(Store r1 r2 im, rs, ms) = let v1 = read rs r1 v2 = read rs r2 a = v1 + im in ms' = write ms a v2	**maMr**:: Minstruction ─ MregFile ─ Mdmem ─ Mdmem **maMr**((add, rs,rt,rd),regf, ram) = ram **maMr**((lw, rs,rt,im),regf, ram) = ram **maMr**((beq, rs,rt,im),regf, ram) = ram **maMr**((sw, rs,rt,im),regf, ram) = let [ra,rb] = regf fs fr rs rt anyA anyD ri = signExtend(im) aluout =alu aluop (mux2 s_ra npc ra)(mux3 s_ri rb ri) in ram' = ram ms mw aluout rb

Figure 4.15: Specifications des composants ISA et de leurs implementations MA correspondantes

4.5.3 Preuve de correction

- Au niveau machine, il faut prouver l'équation suivante pour un état initial arbitraire.

$$abs(\ \mathbf{ma}(\text{pc}, \text{regf}, \text{ram}, \text{rom})) = \mathbf{isa}(abs(\text{pc}, \text{regf}, \text{ram}, \text{rom}))$$

La fonction d'abstraction doit d'abord lire l'etat interne des composants observables, et ensuite convertit cet état du niveau MA vers le niveau ISA.

- Au niveau instruction, la preuve doit être effectuée incrémentalement comme suit:

En premier lieu il faut prouver la correction des portions du chemin de données implémentant les fonctionnalités des cycles de recherche et de décodage (communs à toutes les instructions), en vérifiant l'équation suivante pour une adresse arbitraire.

$$ai(\text{decode}(\text{rom pw})) = \text{read}(am\ (\text{rom pw})\ (ap(\text{pw}))$$

Par suite, il faut s'assurer que la machine MA implémente correctement chaque instruction en prouvant sa condition de vérification séparément.

- Pour l'instruction add

$abs(\mathbf{ex\_ma}((\text{add,rs,rt,rd}),\text{pc, regf, ram, rom})) = \mathbf{ex\_isa}(\text{ai(add,rs,rt,rd)},abs(\text{pc, regf, ram, rom}))$

- Pour l'instruction *load*

$abs(\mathbf{ex\_ma}((\text{lw,rs,rt,im}),\text{pc, regf, ram, rom})) = \mathbf{ex\_isa}(\text{ai(lw,rs,rt,im)},abs(\text{pc, regf, ram, rom}))$

- Pour l'instruction *store*

$abs(\mathbf{ex\_ma}((\text{sw,rs,rt,im}),\text{pc, regf, ram, rom})) = \mathbf{ex\_isa}(\text{ai(sw,rs,rt,im)},abs(\text{pc, regf, ram, rom}))$

- Pour l'instruction *beq*

$abs(\mathbf{ex\_ma}((\text{beq,rs,rt,im}),\text{pc, regf, ram, rom})) = \mathbf{ex\_isa}(ai\ (\text{beq,rs,rt,im}),abs(\text{pc, regf, ram,rom}))$

- Au niveau composant, les conditions de vérification seront interprétées comme suit:

Pour l'instruction add

$$ap\ (\mathbf{maPc}((\text{add rs rt rd}), \text{pc}, \text{regf})) = \mathbf{isaPc}(ai(\text{add rs rt rd}), ap(\text{pc}), ar(\text{regf}))$$

$$\wedge\ ar\ (\mathbf{maRf}((\text{add rs rt rd}), \text{regf}, \text{ram})) = \mathbf{isaRf}(ai(\text{add rs rt rd}), ar(\text{regf}), am(\text{ram}))$$

$$\wedge\ am\ (\mathbf{maMr}((\text{add rs rt rd}),\text{regf},\text{ram})) = \mathbf{isaMr}(ai(\text{add rs rt rd}), ar(\text{regf}), am(\text{ram}))$$

- Pour l'instruction load

$$ap\ (\mathbf{maPc}((\text{lw rs rt im}), \text{pc}, \text{regf})) = \mathbf{isaPc}(ai(\text{lw rs rt im}), ap(\text{pc}), ar(\text{regf}))$$

$$\wedge\ ar\ (\mathbf{maRf}((\text{lw rs rt im}), \text{regf}, \text{ram})) = \mathbf{isaRf}(ai(\text{lw rs rt im}), ar(\text{regf}), am(\text{ram}))$$

$$\wedge\ am\ (\mathbf{maMr}((\text{lw rs rt im}),\text{regf},\text{ram})) = \mathbf{isaMr}(ai(\text{lw rs rt im}), ar(\text{regf}), am(\text{ram}))$$

- Pour l'instruction store

$$ap\ (\mathbf{maPc}((\text{sw rs rt im}), pc, regf)) = \mathbf{isaPc}(ai(\text{sw rs rt im}), ap(pc), ar(regf))$$

$$\wedge\ ar\ (\mathbf{maRf}((\text{sw rs rt im}), regf, ram)) = \mathbf{isaRf}(ai(\text{sw rs rt im}), ar(regf), am(ram))$$

$$\wedge\ am\ (\mathbf{maMr}((\text{sw rs rt im}),regf,ram)) = \mathbf{isaMr}(ai(\text{sw rs rt im}), ar(regf), am(ram))$$

- Pour l'instruction Beq

$$ap\ (\mathbf{maPc}((\text{beq rs rt im}), pc, regf)) = \mathbf{isaPc}(ai(\text{beq rs rt im}), ap(pc), ar(regf))$$

$$\wedge\ ar\ (\mathbf{maRf}((\text{beq rs rt im}), regf, ram)) = \mathbf{isaRf}(ai(\text{beq rs rt im}), ar(regf), am(ram))$$

$$\wedge\ am\ (\mathbf{maMr}((\text{beq rs rt im}),regf,ram)) = \mathbf{isaMr}(ai(\text{beq rs rt im}), ar(regf), am(ram))$$

comme nous pouvons le remarquer, chaque instruction possède ses propres conditions de vérifications, donc, si nous rajoutons des instructions supplémentaires au jeu d'instructions, les conditions de vérification des instructions précédentes restes valides, il faut vérifier uniquement les conditions de vérification des instructions rajoutées. Par conséquent, la méthodologie de preuve proposée convient très bien à l'approche de conception incrémentale.

Puisque toutes les fonctions: les spécifications des composants (isaPc, isaRf, isaMr), leurs implémentations correspondantes (maPc, maRf, maMr), ainsi que les fonctions d'abstraction (abs, ap, ar, am, ai) représentent des programmes fonctionnels, la preuve peut être effectuée par simulation symbolique fonctionnelle.

4.6 Conclusion

La vérification des microarchitectures des processeurs séquentiels exige encore des méthodologies plus efficaces pour structurer la preuve. Pour satisfaire ce critère de structuration, nous avons présenté une approche méthodologique qui permet de développer des implémentations micro architecturales et de prouver leur correction à différents niveaux d'abstraction. A chaque niveau la preuve se décompose en un ensemble de conditions de vérifications plus facile à prouver.

Le niveau machine permet de prouver la correction de la microarchitecture implémentant le jeu d'instruction complet.

Le niveau instruction permet de prouver la correction du chemin de données implémentant une instruction particulière. Cela signifie que nous pouvons implémenter le jeu d'instruction et le vérifier d'une manière incrémentale.

Le niveau composant permet de prouver la correction du chemin de données implémentant un composant observable particulier. Similairement au niveau instruction, si nous étendons la conception par d'autres composants observables, les conditions de vérification des composants précédents restent valides, il faut tester uniquement les conditions de vérification relatives aux composants rajoutes.

Donc, l'habilité d'effectuer des preuves hiérarchiques permet au processus de débogage de continuer à partir du point où le processus de preuve échoue.

Chapitre 5

Machines Pipelines

5.1 Introduction

5.1.1 Objectifs du pipeline

La technique de pipeline constitue la caractéristique d'optimisation des performances la plus importante des microprocesseurs modernes. Elle permet à plusieurs instructions de chevaucher en exécution. Ce chevauchement potentiel appelé parallélisme au niveau instruction (instruction level parallelism : ILP) permet d'augmenter la capacité de traitement du processeur central (throughput) en réduisant le nombre de cycles par instruction (CPI). Le CPI idéal dans une architecture pipeline est égale à un. Cela signifie qu'a chaque cycle d'horloge, une nouvelle instruction peut être recherchée de la mémoire. Pour atteindre cet objectif, les architectures pipelines étendent les architectures multi cycles séquentielles par l'addition des registres pipelines entre les étages successifs, et par une logique de contrôle appropriée pour faire fonctionner ces étages en parallèle. La structure modulaire des architectures RISC, simplifie énormément la conception des machines pipelines. Cependant, leur vérification reste encore une tache très difficile à cause du chevauchement des instructions. La logique de contrôle reste plus difficile, parce que le séquencement du contrôle pipeline est incorporé dans la structure pipeline elle-même (les signaux de contrôle sont sauvegardés dans les registres pipelines et suivent chaque instruction jusqu'à sa terminaison). Une description détaillée des architectures pipelines est illustrée dans [74].

5.1.2 Limitation des performances du pipeline

Il existe des situations, appelées hasards qui empêchent la technique de pipeline de réaliser des performances idéales. Pendant que certains hasards peuvent être traités de telle manière à laisser le pipeline continuer son fonctionnement régulier, d'autres hasards, malheureusement, interrompent le pipeline (une telle situation est appelée: stalling). Par conséquent, la vitesse d'exécution attendue se dégrade. Il existe trois types de hasards qui peuvent se provoquer dans les architectures pipelines. Les Hasards structuraux, les hasards de données et les hasards de contrôle (ou de branchement).

- **Hasards structuraux:** Proviennent du conflit des ressources hardwares, lorsque deux ou plusieurs opérations tentent d'accéder en même temps la même ressource. Par exemple, une opération accède la mémoire pour une recherche de données, alors qu'une autre opération accède la mémoire pour une recherche d'instructions. Une telle situation peut être résolue naturellement par la duplication des ressources hardwares. L'utilisation d'une seule ressource partageable haltera le pipeline pour un cycle quant le conflit se produit.
- **Hasards de données:** Apparaissent quant une instruction dépends des résultats d'une autre instruction précédente se trouvant encore dans le pipe. Il existe trois types de hasards de données dépendant de l'ordre des opérations de lecture/écriture dans le programme. Soient i et j, deux instructions telle que l'instruction i s'exécute avant l'instruction j, alors les différents hasards de données qui peuvent se produire sont les suivants:

- **RAW** (Read After Write): L'instruction j tente de lire une opérande qui n'est pas mise à jour par i. A mois que certaines mesures sont prises, l'instruction j aura une valeur incorrecte. Par exemple, une opération de chargement (load) suivie par une opération arithmétique qui utilise directement le résultat du load, provoque systématiquement un hasard RAW.
- **WAW** (Write After Write): L'instruction j tente d'écrire une opérande avant qu'elle ne soit écrite par l'instruction i. Les hasards WAW se provoquent uniquement dans les pipelines qui écrivent dans plus d'un étage, ou permettent à une instruction de continuer même si une instruction précédente est arrêtée momentanément (stalled). Ce type de hasard ne peut pas se produire dans une machine pipeline MIPS classique, parce qu'elle écrit dans un registre uniquement dans l'étage d'écriture (write back stage).

- **WAR** (Write After Read): L'instruction j tente d'écrire dans une destination avant quelle ne soit lue par l'instruction i. A moins que certaines mesures soient prises, l'instruction i lira la nouvelle valeur. Les hasards WAR se provoquent soit quant des instructions écrivent des résultats tôt dans le pipeline alors que d'autres instructions lisent une source tard du pipeline, ou quant des instructions sont réordonnées en utilisant une organisation (sheduling) dynamique. Ce type d'hasard ne peut pas se provoquer dans les pipelines à issue statique, parce que toutes les lectures sont effectuées tôt dans l'étage de décodage de l'instruction, et toutes les écritures sont effectuées tard dans l'étage d'écriture.

Les hasards de données peuvent être minimisés matériellement par la technique de court-circuit (short-circuiting), appelée aussi "forwarding ou by-passing". Cette technique évite le "stalling" du pipeline par le passage d'une donnée disponible d'une instruction non terminée, aux instructions suivantes (qui la demandent immédiatement). Malheureusement, les hasards de données ne peuvent pas être résolus tous par le mécanisme de by-pass. Par exemple, l'instruction *load* de la machine pipeline MIPS a une latence qui ne peut pas être éliminée seulement par le mécanisme de by-pass. Dans ce cas on aura besoin de rajouter une logique matérielle supplémentaire appelée "interlock" du pipeline qui détecte le hasard et bloque le pipeline jusqu'a l'élimination complète du hasard. Les hasards de données peuvent être aussi éliminés par le mécanisme de renommage des registres [74]

- **Hasards de control:** Ces types de hasards se provoquent quant le compteur ordinal change d'adresse. L'instruction suivante ne peut pas être recherchée jusqu'à ce que la valeur du compteur ordinal soit calculée. Ils existent plusieurs approches pour la résolution du problème des hasards de branchement. Citons quelques exemples:

- Ne prendre aucun branchement conditionnel
- Prendre tous les branchements conditionnels
- Retarder les branchements

5.2 Model de la machine MA pipeline

5.2.1 Construction du model

Tandis que le pipeline permet d'exécuter plusieurs instructions en parallèle pour augmenter la capacité de traitement, il cache d'un autre coté les frontières d'exécution entre les instructions (une instruction démarre avant que la précédente termine). Par conséquent, il devient difficile d'observer l'effet d'une seule instruction dans le pipe.

Puisque le niveau instruction n'est pas observable, le modèle PMA sera formalisé au niveau programme, mais toujours en termes de cycles d'horloge. Il démarre d'un état vide (flushed state), remplit progressivement le pipe et continue d'une manière interminable (contrairement à la technique de flushing de Burch et Dill, qui termine sur un état après vidage) tel qu'il est montré dans la figure 5.1

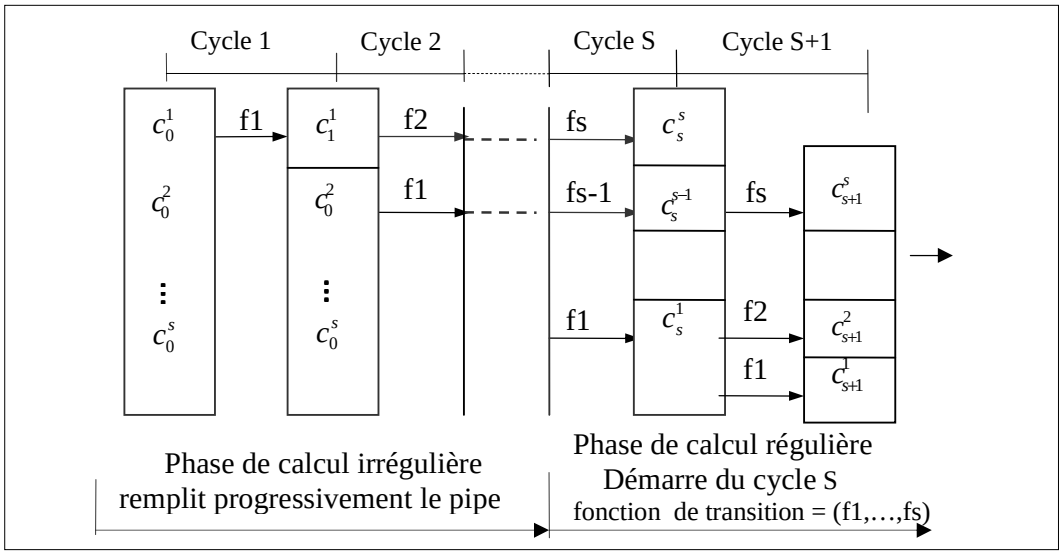


Figure 5.1 :.Diagramme du Modèle Pipeline

La solution clé pour construire le modèle PMA consiste à décomposer l'état PMA. Soient S, le nombre d'étages du pipeline, W= (W1,…,Ws), l'état PMA distribué sur S composants observables, et fi, pour $1 \leq i \leq S$, la fonction de l'étage i. Par suite, la construction du modèle PMA, consiste en deux étapes:

- La première étape effectue un calcul irrégulier. Elle permet de remplir progressivement le pipe jusqu'au cycle S-1. A chaque cycle d'horloge, une nouvelle fonction d'étage est activée pour calculer un nouvel état de composant. Ainsi, partant d'un état initial: $c_0^1,...,c_0^s$ l'état au cycle S-1, est calculé par la fonction PMA comme suit:

$$\text{PMA}((s\text{-}1),\ c_0^1,...,c_0^s) = [(f_{s-1},\ldots,f_1)\ o \ldots o\ (f_2, f_1)\ o\ f1](\ c_0^1,...,c_0^s)$$

- La seconde étape qui démarre du cycle S, effectue un calcule régulier. Elle permet de calculer l'état PMA d'une manière récursive par application répétée de la fonction de transition: $f = (f_1,..,f_s)$, qui se construit automatiquement après S cycles d'horloges. Donc, l'état PMA au cycle $k \geq S$, est calculé par la function PMA comme suit:

$$\text{PMA}(k,\ c_0^1,...,c_0^s) = (f_1,\ldots,f_s)\ (\text{PMA}((k\text{-}1),\ c_0^1,...,c_0^s))$$

La spécification fonctionnelle complète est donnée ci-après.

PMA :: (Int, W) – W

PMA(0, $c_0^1,...,c_0^s$) = ($c_0^1,...,c_0^s$)

PMA(1, $c_0^1,...,c_0^s$) = let $c_1^1 = f_1$ (PMA(0, $c_0^1,...,c_0^s$)

in (σ_1^1, …, c_0^s)

PMA(2, $c_0^1,...,c_0^s$) = let $c_2^1 = f_1$ (PMA(1, $c_0^1,...,c_0^s$)

$c_2^2 = f_2$ (PMA(1, $c_0^1,...,c_0^s$)

in (σ_2^1, σ_2^2 …, c_0^s)

⋮

PMA((s-1), $c_0^1,...,c_0^s$) = let $c_{s-1}^1 = f_1$ (PMA((s-2), $c_0^1,...,c_0^s$)

⋮

$c_{s-1}^{s-1} = f_{s-1}$ (PMA((s-2), $c_0^1,...,c_0^s$)

in ($\sigma_{s-1}^1,\ldots,\sigma_{s-1}^{s-1}$, c_0^s)

Execution régulière (pour k≥S)

PMA(k, $c_0^1,...,c_0^s$) = let $c_k^1 = f_1$ (PMA ((k-1), $c_0^1,...,c_0^s$))

⋮

$c_k^s = fs$ (PMA ((k-1), $c_0^1,...,c_0^s$))

in ($\sigma_k^1,...,\sigma_k^s$)

Figure 5.2 : Spécification fonctionnelle du model PMA pour k≥S

On remarque que lorsque la forme régulière est atteinte, à chaque cycle k (pour k≥S), toutes les opérations *fi* sont effectuées en parallèle.

5.2.2 Paramètres actifs et passifs

L'état produit par une fonction d'étage fi au cycle k, dépend uniquement des états produits par les fonctions d'étages parallèles fj, au cycle k-1, tel que : i-1≤ j ≤ S. Ces états sont appelés: paramètres actifs, les autres états sont appelés paramètres passifs. Par exemple, l'état c_{s+1}^s calculé par la fonction fs au cycle S+1, montré en figure 5.1, dépend uniquement des états c_s^s et c_s^{s-1}, produits au cycle S. En pratique, seulement quelques paramètres actifs seront exigés pour le calcule de l'état de sortie du composant observable. Ceci est très important pour le raisonnement sur la preuve de correction des architectures pipelines.

5.2.3 Support pour les mécanismes de by-pass

Les mécanismes de by-pass et d'interlock sont souvent considérés comme les parties les plus critiques des architectures pipelines, à chaque fois que la vérification formelle de ces systèmes est mise en jeu. Donc, il est fondamentalement important d'utiliser des formalismes suffisamment puissants pour décrire ces mécanismes. En décomposant l'état, notre approche permet de supporter naturellement le by-pass. En effet, et tel qu'il est montré dans la figure 5.1, les résultats produits précédemment par toutes les fonctions d'étage fi, pour $1 \leq i \leq S$, au cycle k (considérer uniquement les paramètres d'état actif) sont prêtes à l'utilisation en tant que paramètres d'état par n'importe qu'elle fonction d'étage au cycle k+1. Puisque le by-pass résolue le hasard, la fonction de transition reste la même. Ceci est un grand avantage par rapport aux approches alternatives. Cependant, si on aura besoin au début d'un cycle, d'un résultat qui sera produit a la fin de ce même cycle (ou dans les cycles suivants), alors, il est nécessaire de retarder le calcul (stalling le pipe) jusqu'à ce que les résultats demandés soient disponibles. Ceci montre que notre approche a l'habilité de raisonner naturellement sur le hasard. Cependant, nous limitons l'étude uniquement au raisonnement manuel parceque l'occurrence du hasard peut faire changer dynamiquement les coordonnées de la fonction de transition, et par conséquent, le raisonnement automatique reste relativement difficile.

5.3 Stratégie de la preuve de correction

5.3.1 Modèle de l'etape-MA

La décomposition horizontale verticale de la fonction d'etape-MA définie au chapitre 4 est fondamentalement importante pour l'implémentation et la vérification des architectures MA séquentiels. Cependant, une telle définition reste insuffisante pour la vérification des architectures MA pipeline, parce que les états intermédiaires ne sont pas observables. Cette définition prend un état initial et retourne un état final après l'exécution d'une instruction. Pour cette raison, nous allons décomposer la fonction d'étape-MA de telle manière à observer l'évolution d'état à chaque cycle d'horloge comme suit

$$ma = [f_1,\ f_2 \mathbf{o} (f_1,\ldots,f_1^s), \ldots, f_s \mathbf{o} (f_{s-1}^1,\ldots,f_{s-1}^s) \mathbf{o}\ldots\mathbf{o} (f_1,\ldots,f_1^s)]$$

Pour simplifier la notation, les coordonnées fi^i seront dénotées tout simplement *fi*. Dans cette forme, toutes les coordonnées *fi* sont des transformateurs, alors que les autres coordonnées sont des sélecteurs (pour lire d'un pipe et écrire dans le pipe suivant). De cette manière, tous les états des composants qui sont calculés par les coordonnées *fi* à travers les différents étages sont capturés, voir figure 5.3. Pour être réaliste, nous avons limité l'observation à un seul état observable par étage. Par exemple, la machine MIPS, met à jour l'état du compteur ordinal à l'étage de recherche, l'état de la mémoire à l'étage d'accès mémoire, et l'état du fichier des registres à l'étage d'écriture.

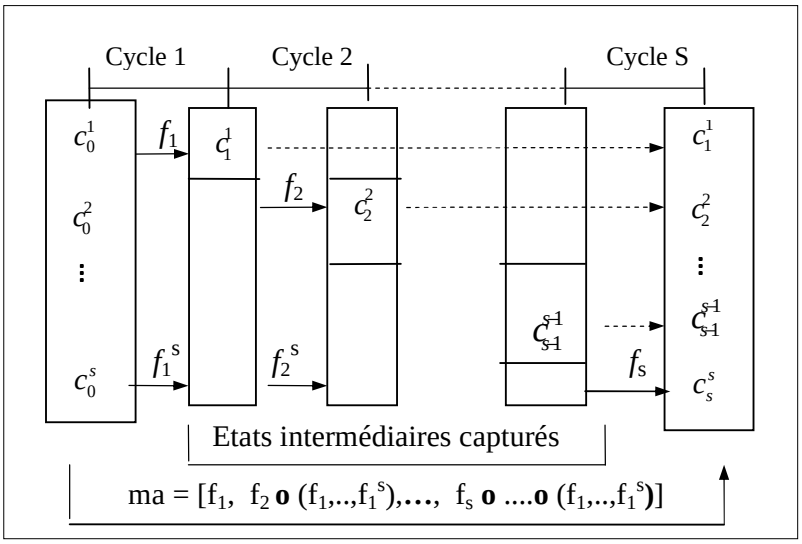


Figure 5.3: Evolution des états à travers les étages

Une implémentation fonctionnelle de la forme ci-dessus est donnée ci-après

$$
\begin{aligned}
&ma:::W - W\\
&ma(c_0^1,...,c_0^s) = \text{let } c_1^1 = f_1(c_0^1,...,c_0^s)\\
&\qquad\vdots\\
&\qquad c_s^s = (f_s \mathbf{o} \ldots \mathbf{o} (f_1,\ldots,f_1^s))\,(c_0^1,...,c_0^s)\\
&\qquad\qquad \text{in } (c_1^1,\ldots c_s^s)
\end{aligned}
$$

Figure 5.4 : Implémentation de l'etape-MA capturant les états intermédiaires

5.3.2 Modèle de la machine MA séquentielle

Une machine MA séquentielle sera définie par une fonction d'état récursive qui produit l'état MA après exécution de n instructions (En appliquant *n* fois l'étape MA).

SMA:: (Int, W) – W

SMA(0, c_0^1,…,c_0^s) = (c_0^1,…,c_0^s)

SMA (n, (c_0^1,…,c_0^s)) = *ma*(SMA((n-1), c_0^1,…,c_0^s))

En développant la fonction *ma*, la définition SMA réécrit comme suit:

SMA(**n**, $c_0^1,...,c_0^s$) =
  let $c_n^1 = f_1$(SMA((**n-1**), $c_0^1,...,c_0^s$)
    ⋮
    c_n^s = (*fs* o ..o (f_1,…,f_1^s)) (SMA((**n-1**), $c_0^1,...,c_0^s$))
in ($c_n^1,...,c_n^s$)

Figure 5.5 : Spécification fonctionnelle du model SMA

Une telle forme est très importante parcequ'elle montre explicitement les différents temps où les différents états sont mis à jour.

5.3.3 Modèle MA séquentiel des composants

5.3.3.1 Diagramme de synchronisation

La majorité des méthodologies de preuve (incluant celle de Burch et Dill) vérifient une implémentation pipeline contre une spécification ISA, et par conséquent, il est impossible de trouver un point significatif où l'état d'implémentation et l'état de spécification peuvent être comparés facilement. Une solution alternative à ce problème est de vérifier une implémentation PMA contre une implémentation multicycle séquentielle. Puisque les deux modèles sont formalisés en termes de cycles d'horloge, tous les états intermédiaires synchrones représentent des points utiles oŭ la comparaison peut être effectuée facilement. En effet, à la fin de chaque cycle d'horloge, une architecture PMA avec S étages révèle S résultats partiels, où chaque résultat est produit par une instruction dans le pipe. Donc, on peut construire une variante du model SMA – appelée model SMA des composants (Component SMA: CSMA) qui simule l'effet de calculer les mêmes résultats séquentiellement comme le montre la figure 5.6

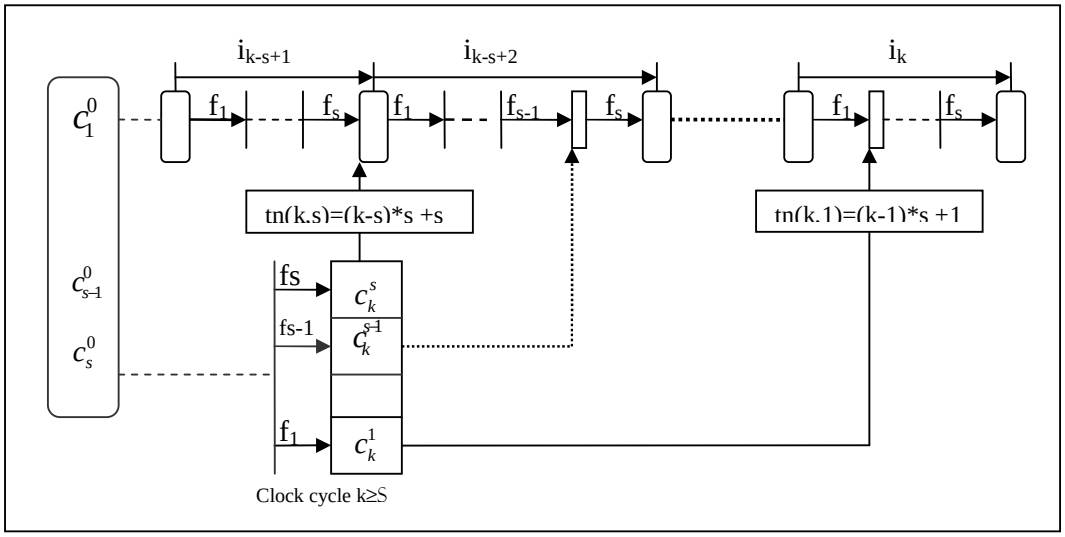


Figure 5.6: Diagramme de synchronisation entre les modèles: séquentiel et pipeline

Puisque les deux modèles (séquentiel et pipeline) se rapportent au niveau MA, la fonction d'abstraction de données n'est pas exigée, uniquement une fonction d'abstraction de temps est demandée pour synchroniser les états produits par ces deux derniers. Soit c_k^j, l'état d'un composant PMA produit par la fonction d'étage *f*j, au cycle d'horloge k, pour k≥S.:

$$c_k^j = f_j(\text{PMA}((k-1),\ c_0^1,...,c_0^s)\ .$$

S'il n' y'a pas de hasards, la synchronisation est effectuée, en utilisant la fonction de temps suivante:

$$t_n(k,j) = (k-j)*s + j \quad (\mathbf{t1})$$

Cela signifie, que nous aurons besoin de (k-j) *S cycles d'horloges pour exécuter séquentiellement (k-j) instructions par le model SMA (avec S étages), et nous aurons besoin encore de j cycles d'horloge supplémentaires, pour atteindre l'état séquentiel correspondant qui est calculé comme suit:

$$s^j_{t(k,j)} = (f_j \mathbf{o} \ldots \mathbf{o} (f_1,\ldots,f_1^s))\ (\text{SMA}((k-j),\ c_0^1,...,c_0^s))$$

S'il y'a des hasards, la fonction de temps réécrit comme suit:

$$t_s(k,j,i) = ((k-j)-i)*s + j \quad (\mathbf{t2})$$

où i, indique le nombre de 'stalls'.

5.3.3.2 Model MA séquentiel des composants

Le model CSMA que nous proposons ici, prend en entrée le même cycle d'horloge k, que le modèle PMA, contrairement au modèle SMA qui prend le nombre d'instructions à exécuter (section 5.3.2). Pour chaque cycle k≥S, il construit S termes (sur le modèle SMA), où chaque terme calcule un résultat partiel pour une instruction dans le pipe tel qu'il est montré dans la figure 5.7 (contrairement au model SMA qui retourne S résultats pour la même instruction).

CSMA:: (Int, W) – W

CSMA(0, $c_0^1,...,c_0^s$) = ($c_0^1,...,c_0^s$)

CSMA(1, $c_0^1,...,c_0^s$) = let $c_1^1 = f1$(SMA(0, $c_0^1,...,c_0^s$))

in ($\mathbf{c}_1^1,..,c_0^s$)

CSMA(2, $c_0^1,...,c_0^s$) = let $c_1^2 = f1$(SMA((1, $c_0^1,...,c_0^s$))

$c_2^2 = (f2 \text{ o } (f1,\ldots f_1^s))$(SMA((0, $c_0^1,...,c_0^s$))

in ($\mathbf{c}_1^1, \mathbf{c}_2^1..,c_0^s$)

⋮

CSMA((s-1), $c_0^1,...,c_0^s$) = let $c_{s-1}^1 = f1$(SMA((s-2), $c_0^1,...,c_0^s$))

⋮

c_{s-1}^{s-1} = (*f*s-1 o, ,o (*f*1,…f_1^s)) (SMA(0, $c_0^1,...,c_0^s$))

in ($\mathbf{c}_{s-1}^1,\ldots,\mathbf{c}_{s-1}^{s-1}, c_0^s$)

For k≥S

CSMA(k, $c_0^1,...,c_0^s$) = let c_k^1 =*f*1(SMA((k-1), $c_0^1,...,c_0^s$)

⋮

c_k^s = (*f*s o, ..,o (*f*1,…f_1^s)) (SMA((k-s), $c_0^1,...,c_0^s$))

in ($\mathbf{c}_k^1,...,\mathbf{c}_k^s$)

Figure 5.7. Spécification Fonctionnelle du modèle CSMA

5.3.4 Critère de correction

La preuve de l'équivalence fonctionnelle du modèle PMA vis avis du modèle CSMA, exige la preuve de l'équation suivante:

$$\forall \quad k :: Int, \quad \forall \; c_0^1 :: W1, \ldots c_0^s :: Ws$$

$$PMA(k, c_0^1,...,c_0^s) = CSMA(k, c_0^1,...,c_0^s)$$

Cela signifie que, partant du même état initial: $c_0^1,...,c_0^s$, les deux modèles doivent révéler à chaque cycle k, le même état. La preuve de cette équation se décompose systématiquement aux preuves des équations suivantes:

$$f_1 \text{ (PMA ((k-1), } c_0^1,...,c_0^s)) = f_1 \text{ (SMA((k-1), } c_0^1,...,c_0^s) \quad \textbf{(e1)}$$

$$\vdots$$

$$\wedge\ fs\text{(PMA ((k-1), } c_0^1,...,c_0^s)) = (\ fs\ \textbf{o}\ ...\textbf{o}\ (\ f_1,\ldots,f_1^s)) \text{ (SMA((k-s), } c_0^1,...,c_0^s)) \quad (\textbf{es})$$

5.3.5 Discussion

- Les équations établies ci-dessus sont prouvables séparément par induction sur les cycles d'horloges [75]. Ceci évite l'utilisation de l'évaluation symbolique qui reste difficile et insuffisante pour les systèmes complexes [76].
- Aussi, ces équations peuvent être instanciées pour n'importe qu'elle architecture particulière, en spécifiant tout simplement les fonctions des étages *fi*. Donc, la méthodologie de preuve proposée escalade avec la complexité des architectures.
- Le nombre des équations à prouver dépend du nombre des observables. Cela signifie que nous pouvons limiter la preuve uniquement aux observations auxquelles on s'intéresse. Ceci est très utile pour les systèmes pipeline impliquant plusieurs observations.

5.4 Application

Pour montrer la validité de notre approche, nous allons l'appliquer a la vérification formelle de la machine pipeline MIPS, dont le bloc diagramme est donné dans la figure 5.8, vis-à-vis de la version multicycle séquentielle décrite au chapitre 4. Les fonctionnalités des différents étages des deux machines sont les mêmes. Pour la machine séquentielle, les étages sont activés séquentiellement (round robin), alors que pour la machine pipeline, les étages son activés en parallèle.

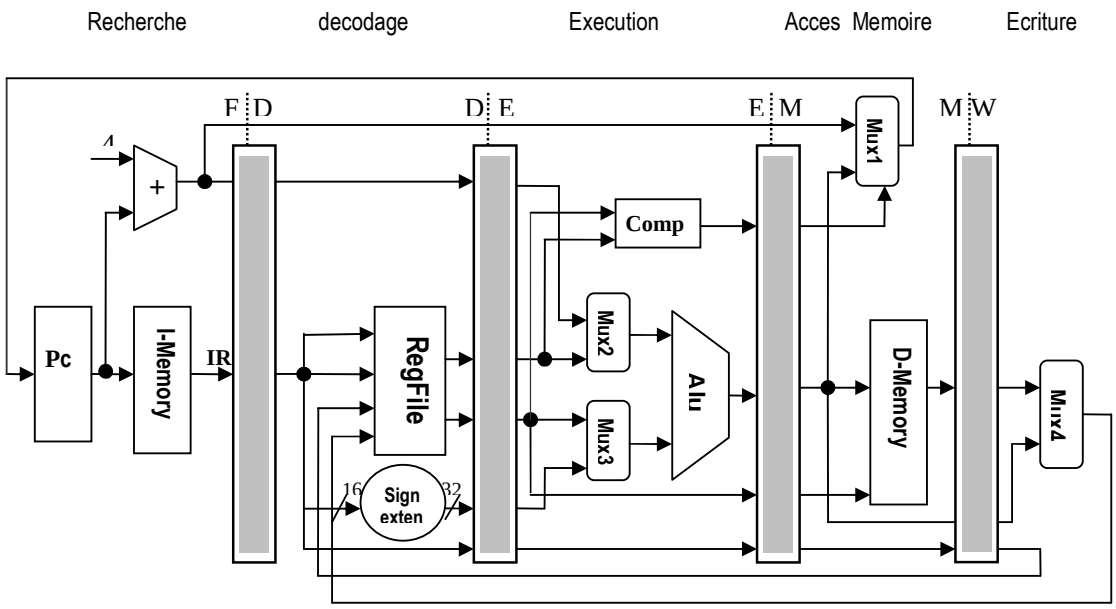


Figure 5.8: Bloc diagramme de la machine pipeline MIPS

5.4.1 Définition de l'état MA

L'observation sera limitée a trois composants: le compteur ordinal (PC), le fichier-registre, et la mémoire de données (la mémoire des instructions reste inchangée). Ces composants sont typés comme suit:

type Word = [Bit]

type PC = Word

type RegFile = [Word]

type Dmem = [Word]

type Imem = [Word]

L'état MA inclus en plus des observables, les registres pipelines qui stockent temporairement l'information entre les différents étages.

type PipeFD = (PC, IR)

type PipeDE = (PC, IR, RA, RB, RI)

type PipeEM = (IR, RB, Aluout, Cond)

type PipeMW = (IR, Aluout, Lmd)

type MA_state = (PipeFD, PipeDE, PipeEM, PipeMW, PC, RegFile, Dmem, Imem)

5.4.2 Spécification des étages de la machine MA

La spécification des signatures (interfaces) des différents étages est donnée ci-après. Une description informelle est donnée dans [74].

fe :: MA_state – PipeFD

de:: MA_state – PipeDE

ex:: MA_state – PipeEM

me:: MA_state – (Dmem, PipeMW)

wb:: MA_state – RegFile

Pour simuler une architecture pipeline, nous aurons besoin aussi de fonctions sélecteurs pour copier les états des composants (qui seront demandés plus tard) d'un pipe au suivant. Ces fonctions sélecteurs sont typées comme suit:

fs:: MA_state – (PipeDE, PipeEM, PipeMW)

ds:: MA_state – (PipeFD, PipeEM, PipeMW)

es:: MA_state – (PipeFD, PipeDE, PipeMW)

ms:: MA_state – (PipeFD, PipeDE, PipeEM)

ws:: MA_state – (PipeFD, PipeDE, PipeEM, PipeMW)

5.4.3 Modèle pipeline

Nous limitons l'étude uniquement à la phase régulière (pour k≥ 5) où tous les étages sont activés en parallèle. Bien sure, le système commence par remplir progressivement les pipes avant de se stabiliser. A la fin de chaque cycle, le modèle PMA révèle cinq résultats partiels, chaqu'un se rapportant à une instruction dans le pipe.

pma :: (Int, MA_state) – MA_state

pma(k, fd, de, em, mw, pc rf,, dm, im) =

Let (**pc'**, ir1) = **fe**(pma((k-1), fd, de, em, mw, pc rf, dm,im))

(pc2, ir2, ra, rb, ri) = **de**(pma((k-1), fd, de, em, mw, pc, rf, dm,im))

(ir3, rb2 ,aluout, cond) = **ex**(pma((k-1), fd, de, em, mw, pc,rf, dm,im))

(**dm',** ir4, aluout2**,** lmd) = **me**(pma((k-1),fd, de, em, mw, pc,rf, dm,im))

rf ' = **wb**(pma ((k-1),fd, de, em, mw, pc,rf**,** dm,im))

in **(** fd' = (pc', ir1),

de' = (npc2, ir2, ra, rb, ri),

em' = (ir3, rb2, aluout, cond'),

mw' = (ir4, aluout2, lmd),

pc', rf ', dm', im)

5.4.4 Modèle séquentiel

Le modèle séquentiel retourne les états après exécution de k instructions.

sma :: (Int, MA_state) – MA_state
sma (**k,** fd, de, em, mw ,pc, rf, dm,im) =
let (**pc'** ir1) = ***fe*** (**sma**((**k-1**), fd, de, em, mw, pc,rf, dm,im))
(npc2, ir2, ra, rb, ri) = (***de*** **.** (fe, fs)) (**sm**a ((**k-1)**, fd, de, em, mw, pc, rf, dm, im))
(ir3, rb2, aluout, cond) = (***ex*** **.**(de, ds) **.** (fe, fs)) (**sma**((**k-1**), fd,de,em,mw, pc,rf,dm,im))
(**dm'**,ir4,aluout2,lmd) =(**m**e **.**(*ex*,*es*) **.** (*de*,ds) **.** (*fe*,fs))(**sma**((**k-1**),fd,de,em,mw, pc,rf,dm,im))
rf '= (**wb .** (me,ms) **.** (ex**,**es) **.** (de,ds) **.** (fe,fs)) (**sma**((**k-1**),fd, de,em,mw, pc,rf,dm,im))
in (fd' = (**pc'**, ir1),
de' = (npc2, ir2, ra, rb, ri),
em' = (ir3, rb2, aluout, cond'),
mw' = (ir4, aluout2, lmd),
pc', rf ', dm', im **)**

5.4.5 Modèle séquentiel des composants

Le modèle CSMA qui est défini sur le modèle SMA retourne les mêmes résultats partiels calculés par le modèle pipeline. Au cycle k, chaque étage i calcule le résultat partiel appartenant a l'instruction (k+1)-i. Le spécification fonctionnelle du modèle CSMA simulant le calcule pipeline du processeur MIPS est donnée ci-après.

csma :: (Int, MA_state) – MA_state
csma (k, fd, de, em, mw, pc, rf, dm, im) =
Let (**pc'** ir1) = ***fe***(sma((**k-1**), fd, de, em, mw,pc, rf, dm im))
(npc2, ir2, ra, rb, ri) = (***de*** **.** (fe, fs)) (sma((**k-2**), fd, de, em, mw, pc,rf,dm, im))
(ir3, rb2, aluout, cond) = (***ex*** **.** (de, ds) **.** (fe,fs)) (sma((**k-3**), fd, de, em, mw, pc,rf,dm, im))
(**dm'**,ir4,aluout2,lmd)=(***me*** **.** (*ex*,*es*) **.** (*de*,ds) **.**(*fe*,fs))(sma((**k-4**),fd,de,em,mw, pc,rf,dm,im))
rf ' = (***wb*** **.** (me,ms) **.** (ex,es) **.** (de,ds) **.** (fe,fs)) (sma((**k-5**),fd,de,em,mw, pc,rf,dm,im))
in (fd' = (**pc'**, ir1),
de' = (npc2, ir2, ra, rb, ri),
em' = (ir3, rb2, aluout, cond),
mw' = (ir4, aluout2, lmd),
pc', rf ', dm', im **)**

5.4.6 Critère de correction

Selon la section **5.3.4**, la preuve de correction de la machine pipeline MIPS, est assurée si les équations suivantes sont satisfaites.

fe(pma((k-1), ma_state)) = **fe**(sma((**k-1**), ma_state) (**a1**)
$\wedge$ **de**(pma((k-1), ma_state)) = **(de .** (fe, fs)**)** (sma((**k-2**), ma_state)) (**a2**)
$\wedge$ **ex**(pma((k-1), ma_state)) = **(ex .** (de, ds) **.** (fe,fs)**)** (sma((**k-3**), ma_state)) (**a3**)
$\wedge$ **me**(pma((k-1), ma_state)) = **(me .** (ex,*es*) **.** (de,ds) **.** (fe,fs)**)** (sma((**k-4**), ma_state)) (**a4**)
$\wedge$ **wb**(pma((k-1), ma_state)) = **(wb .** (me,ms) **.** (ex,es) **.** (de,ds)**.**(fe,fs)**)** (sma((**k-5**), ma_state)) (**a5**)

Bien qu'il est possible de prouver les fonctionnalités de tous les étages, nous limitons la preuve de correction uniquement a trois observations: Le compteur ordinal, le fichier registre, et la mémoire. Par suite, nous aurons besoin de projeter aussi les états du compteur ordinal et de la mémoire, après leur mise à jour par les étages de recherche et d'accès mémoire respectivement. Cette projection est omise pour l'état du fichier-registre parcequ'il est le seul état observé à l'étage d'écriture.

projPc :: MA_state – PC
projMr :: MA_state – Dmem

Dans le cas d'absence de hasards ou dans le cas de présence de hasards qui peuvent être résolus par les mécanismes de by-pass (pas de stalls), la preuve de correction de la machine pipeline MIPS est assurée si les équations suivantes sont satisfaites:

projPc(fe(pma((k-1), ma_state))) = projPc (fe(sma((k-1), ma_state))) (**b1**)
$\wedge$ pro jMr(me(pma((k-1),ma_state)))=projMr**(**(me**.**(ex,es)**.**(de,ds)**.**(fe,fs))(sma((k-4),ma_state))**)** (**b2**)
$\wedge$ wb(pma ((k-1),ma_state)) =(wb **.** (me,ms)**.**(ex,es) **.**(de,ds) **.**(fe,fs)) (sma((k-5),ma_state)) (**a5**)

Dans le cas d'un branchement pris (taken branch), exigeant un ‘stall’, la condition de vérification de l'état du compteur ordinal, réécrit comme suit:

projPc(fe(pma((k-1), ma_state))) = projPc (fe(sma(((k-1)-i), ma_state))) (**b3**)

Où i, représente le nombre de cycles pour lesquels le system doit être en situation de ‘stalls’. Il faut noter que les équations (b1) et (b2) sont conséquentes des équations (a1) et (a4). Si (a1) et (a4) sont satisfaites, alors (b1) et (b2) suivent systématiquement.

Maintenant, il devient facile de raisonner sur chaque condition de vérification séparément. En outre, on peut raisonner soit sur des instructions individuelles ou sur des groupes d'instructions, telles que des instructions de type registre, de type mémoire, ou de type branchement.

5.4.7 Preuve de correction

La preuve sera effectuée par induction sur des cycles d'horloges. Puisque les deux modèles démarrent du même état initial, Le cas de base est implicite. Donc, nous allons considérer uniquement le cas inductif. Pour simplifier la preuve, nous allons considérer pour chaque fonction d'étage, uniquement les paramètres d'entrée qui sont nécessaires pour le calcule de l'état de sortie correspondant (les autres paramètres sont nécessaires uniquement pour le typage correcte des fonctions d'étage).

5.4.7.1 Etat du PC

- *Instructions de typr-R et de type-M*: Pour calculer l'état suivant du PC pour ces types d'instructions, la fonction de l'étage de recherche du modèle pipeline, exige comme paramètres actifs, uniquement le résultat produit précédemment par le même étage (voir diagramme de la figue 5.9a).

Maintenant, nous allons supposer que l'équation (a1) est vraie pour le cycle k, et nous allons la prouver pour le cycle k+1, également.

fe (pma(k, ma_state)) = fe(fe (pma(k-1), ma_state)) *definition de pma (un seul parameter actif)*
= fe(fe (sma(k-1), ma_state)) *cas inductif*
= fe (sma(k, ma_state)) *definition de sma*

Par Conséquent,

projPc(fe (pma(k, ma_state))) = projPc (fe (sma(k, ma_state)))

Ce qui termine rapidement la preuve.

- *Instructions de branchement:* Pour ces instructions, la fonction de l'étage de recherche dépend du résultat produit par l'étage d'exécution (voir [74], p A33).

Supposons, (ir3', rb',aluout',cond') = **ex**(pma((k-1), ma_state))

Deux cas seront discutés

- *Cas 1*: cond = False (Branchement non pris)

Dans ce cas, l'exécution continue en séquence, et le PC est mis à jour (incrémenté), en utilisant uniquement le résultat de l'étage de recherche. Par suite, la preuve est facile parceque ce cas est similaire à celui discuté auparavant.

- *Cas 2:* cond = True (Branchement pris)

Dans ce cas, le PC est mis à jour, en utilisant le résultat de l'étage d'exécution au cycle k, comme paramètre actif.

fe (pma(**k**, ma_state)) = fe(ex(pma(k-1), ma_state)) *definition de pma*

= fe ((ex **.** (de,ds) **.** (fe,fs)) **.** (sma((k-3),ma_state))) *cas inductif*

=fe(sma((**k-2**), ma_state)) *definition de sma avec la fonction de temps **t2**, (2 stalls)*

Par consequent,

projPc(fe (pma(**k**, ma_state))) = projPc(fe(sma((**k-2**), ma_state)))

Cela signifie que nous aurons besoin d'un 'stall' de deux cycles avant de mettre à jour le PC avec la nouvelle adresse de branchement (les deux dernières instructions sont ignorées comme le montre la figure 5.9b). Donc, selon l'équation (b3), l'approche donne des résultats corrects.

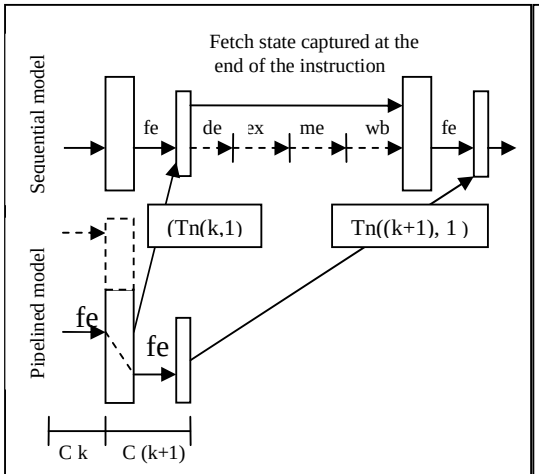


Figure 5.9a: Diagramme de correction du PC pour les instructions de type-R et type-M

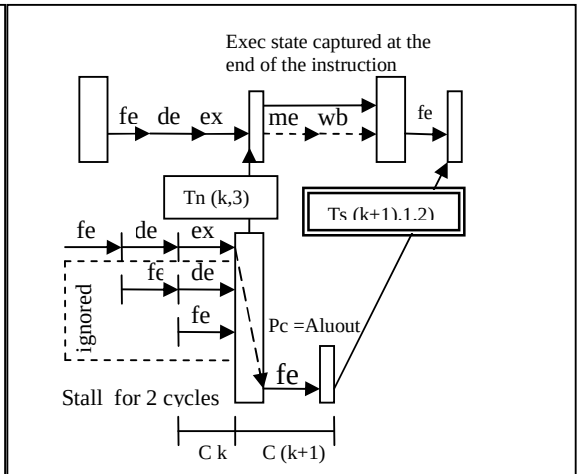


Figure 5.9b : Diagramme de correction du PC pour un branchement pris: PC mis à jour par le résultat de l'etage d'execution au cycle k (stall pour 2 cycles)

5.4.7.2 Etat de la mémoire

- *Instructions de type-R*: Pour les instructions de type-R, la fonction de l'étage d'accès mémoire prend comme paramètres actifs uniquement le résultat produit par l'étage d'exécution et le fait passer du registre pipeline EX/MEM (à travers les sélecteurs), au registre pipeline MEM/WB. Donc, la preuve est très facile.
- *Instructions de type-M:* Pour ces instructions, la fonction de l'étage d'acces-memoire prend comme paramètres d'entrée, les résultats produits par les étages d'exécution et d'acces-memoire.

me (pma(k, ma_state)) = me **[me**(pma((k-1),ma_state)), **ex**(pma((k-1), ma_state))**]**

= me**[(**me**.**(ex,es)**.**(de,ds)**.**(fe,fs)) (sma((**k-4**), ma_state))**,** *1st parametre: Cas Inductif*

(ex**.**(de,ds)**.**(fe,fs)) (sma((k-3),ma_state))**]** *2d parameter : Cas Inductif*

= me**[(**es**.**(de,ds)**.**(fe,fs))(sma((**k-3**),ma_state))**,** *1st parametre: capture a la fin de l'inst. K-4,*

(ex**.**(de,ds)**.**(fe,fs)**)**(sma((k-3),ma_state))**]** *2nd parameter : reste le meme*

= me**[**((ex,es) **.**(de,ds)**.**(fe,fs))(sma((k-3),ma_state))**]** :*factorisation des deux paramètres*

Par conséquent,

projMr(me (pma(k, ma_state))) = projMr**(**(me**.**(ex,es) **.**(de,ds)**.**(fe,fs)) (sma((k-3),ma_state))**)**

Ceci termine la preuve.

5.4.7.3 Etat du fichier-registre

- *Instructions de type-R et instruction load*: La fonction de l'étage d'écriture est la même pour les instructions de type-R et l'instruction load [64, p A32]. Elle accepte deux paramètres: Le résultat produit par l'étage d'acces-memoire de l'instruction en cours et le résultat produit par l'étage d'écriture de l'instruction précédente, et met à jour le fichier-registre.

wb(pma(k,ma_state) = wb **[me** (pma((k-1),ma_state)), **wb**(pma ((k-1),ma_state))**]**

= wb **[(**me**.**(ex,es)**.**(de,ds)**.**(fe,fs)**)** (sma((k-4), ma_state))**,** *1st parametre: cas inductif*

(wb**.**(me,ms)**.**(ex,es)**.**(de,ds)**.**(fe,fs)**)** (sma((k-5),ma_state))**]** *2nd parametre :. Cas inductif*

= wb **[(**me**.**(ex,es)**.**(de,ds)**.**(fe,fs)) (sma((k-4), ma_state)), *1st parametre: reste le meme*

(ms**.**(ex,es)**.**(de,ds)**.**(fe,fs)) (sma((k-4), ma_state))**]** *2nd parametre : capture via selecteurs*

= wb **[(**(me,ms)**.**(ex,es)**.**(de,ds)**.**(fe,fs)**)** (sma((k-4), ma_state))**]** *Factorisation des deux parametres*

Ceci établit la preuve.

5.5 Conclusion

En vérifiant une implémentation microarchitecturale pipeline contre une implémentation multi cycle séquentielle, notre approche offre plus d'avantages par rapport aux approches alternatives qui tentent de vérifier une implémentation micro architectural pipeline contre une spécification ISA:

- Elle décompose la preuve globale d'une manière systématique en un ensemble de conditions de vérification plus simple à vérifier. En particulier, nous pouvons raisonner sur la dépendance inter instruction, tels que les différents types de hasards qui peuvent apparaître durant l'exécution, contrairement à la technique de "flushing" où ce

raisonnement est impossible. En outre, il est possible de raisonner soit sur des instructions individuelles soit sur des groupes d'instructions telles que des instruction-registres, des instruction-mémoires, ou des instructions de branchement.

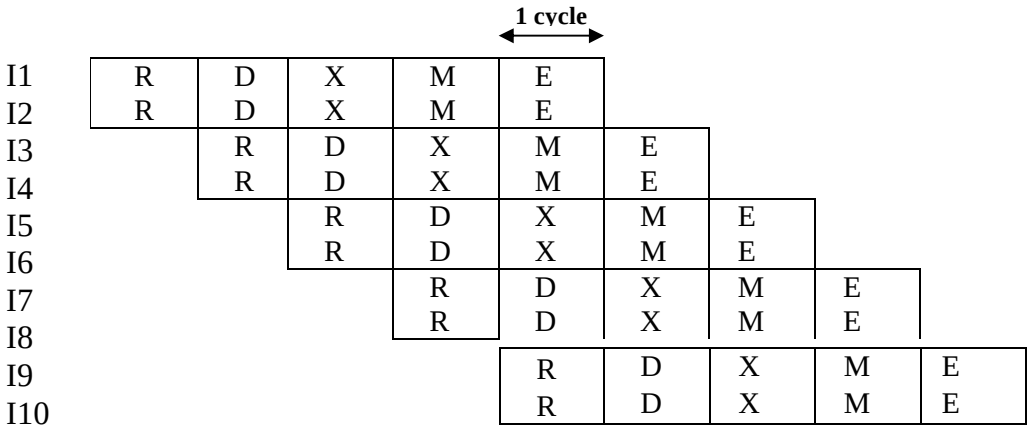
- Puisque les deux modèles (séquentiel et pipeline) se rapportent au niveau MA, notre approche n'exige pas de fonction d'abstraction de données qui reste difficile à définir en pratique, uniquement une fonction d'abstraction de temps est demandée pour synchroniser les deux modèles.
- La possibilité d'instancier l'ensemble des équations pour n'importe qu'elle architecture particulière, offre une meilleure scalabilité pour la vérification des architectures hautement optimisées.
- La puissance clef de l'approche de preuve proposée, est son habilité d'effectuer la preuve par induction sur des cycles d'horloges au sein du même langage de spécification , plutôt que par évaluation symbolique à travers un outil de preuve qui demande encre des efforts considérables.

Chapitre 6

Machines Superscalaires

6.1 Introduction

Les machines superscalaires qui étendent les machines pipelines, sont construites en mettant plusieurs pipelines en parallèle. Cela signifie que ces machines peuvent initier plusieurs instructions durant le même cycle d'horloge. En d'autres termes, la technique des superscalaires permet de réduire encore le CPI, en le rendant inférieur à un (1). Le problème des dépendances peut alors devenir très complexe puisque celles-ci peuvent se produire entre des instructions qui sont chargées dans différents pipelines. Pour réduire ce genre de dépendance inter instruction, les pipelines peuvent être spécifiques à certains types d'instructions (entiers, flottants, etc.). Figure 6.1 montre un modèle simple d'une machine superscalaire avec deux pipelines symétriques, où chaque pipeline possède cinq étages. A chaque cycle d'horloge, deux instructions sont recherchées en parallèle (à partir d'une cache a lecture associative). En absence de hasards, dix instructions peuvent être traitées en parallèle.



R : Recherche, **D** : Décodage, **X** : Exécution, **M**: Acces-Mémoire, **E**: Ecriture

Figure 6.1 : Superscalaire avec deux pipelines symétriques

6.2 Classification des machines superscalaires

Deux modèles de processeurs superscalaires se distinguent selon la stratégie d'ordonnancement : Les superscalaires statiques (à exécution ordonnée) et les superscalaires dynamique (à exécution désordonnée).

6.2.1 Les superscalaires à exécution ordonnée (in-order execution superscalars)

Ces processeurs sont construits par la réplication de plusieurs pipelines qui fonctionnent en parallèle. Ils utilisent un ordonnancement statique (static sheduling) qui initie et exécute des instructions dans l'ordre du programme. En absence de hasards, Le nombre d'instructions qui peuvent être exécutés simultanément dépend du nombre de pipelines et la longueur de ces derniers. Le nombre d'instructions à initier (issue) à chaque cycle pour l'exécution dépend de la présence de hasards. Par exemple dans une machine de degré 2 (avec 2 pipelines), il est possible d'initier de zéro à deux instructions. L'initiation à l'exécution est bloquée (stalled) quand il y'a un conflit d'accès à une unité fonctionnelle ou quant l'instruction en cours dépends d'un résultat qui n'est pas encore disponible. Dans ce cas, les instructions sont initiées en série. Les résultats des instructions sont écrits dans l'ordre de recherche de ces dernières. Des exemples tels que MIPS R5000 [77], et RM7000 [78], font partie de cette classe. Figure 6.2 montre une microarchitecture typique d'un superscalaire à exécution ordonnée avec deux pipelines symétriques qui fonctionnent selon le diagramme de la figure 6.1 Les fonctions des étages pour ce modèle, restent très similaires a celles d'une machine séquentielle : La phase recherche effectue une lecture par bloc de deux instructions à partir d'une mémoire cache associative. La phase décodage lit les opérandes des instructions à partir du fichier des registres, prédicte des branchements, et prépare les instructions selon leur type (entière, flottante) pour la phase d'exécution où elles seront exécutées en parallèle. La phase d'accès mémoire traite uniquement des instructions de références mémoire telles que load et store, enfin la phase d'écriture permet de mettre à jour les résultats dans le fichier des registres.

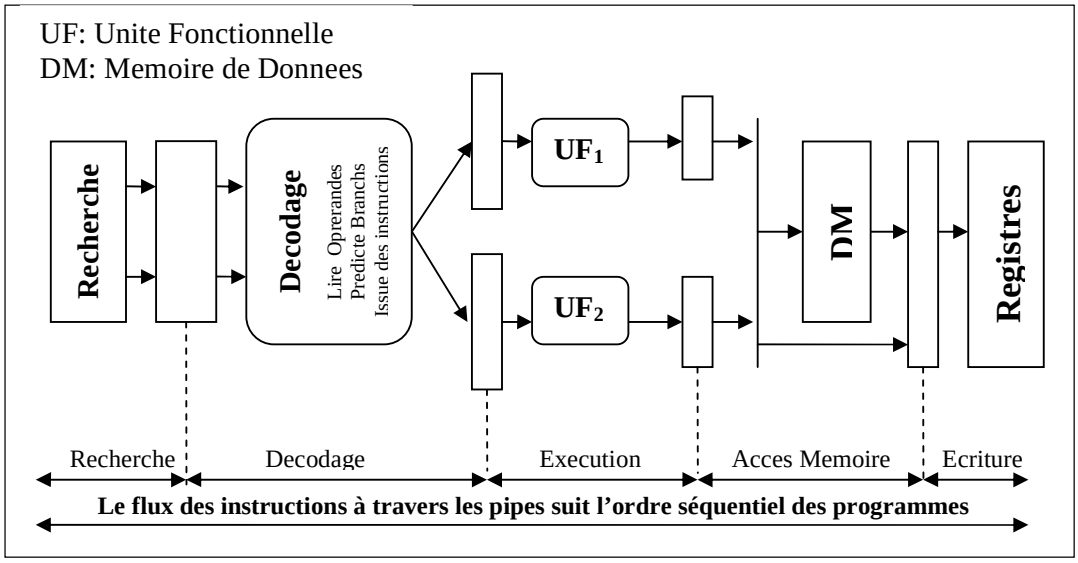


Figure 6.2. Microarchitecture d'un superscalaire a exécution ordonnée

6.2.2 Les superscalaires à exécution désordonnée (out-of-order execution superscalars)

Ces processeurs permettent une exécution désordonnée des instructions selon l'algorithme de Tomassulo [79]. Par conséquent, un ordonnancement dynamique (dynamic sheduling) des instructions est nécessaire. Une microarchitecture typique de ce genre de superscalaire est donnée par la figure 6.3. Les instructions sont lues par blocs (à partir d'une mémoire cache) dans le buffer des instructions où elles sont analysées vis-à-vis de la dépendance de contrôle qui peut être provoquée par les branchements conditionnels. L'historique des branchements est sauvegardé dans une table dite : table de prédiction des branchements Pour maintenir un degré de parallélisme élevé, des méthodes de prédiction de branchements conditionnels avec une exécution spéculative sont utilisées. Le compteur ordinal est mis à jour dans cette phase de recherche. Le problème de dépendance de données (du particulièrement aux Hasards RAW) est résolu dans la phase de décodage. Durant cette phase, les instructions sont assignées des 'tags' permettant de renseigner sur la validité de leurs opérandes. Les instructions décodées sont placées dans la fenêtre d'issue où elles seront examinées par la logique d'issue (unité d'ordonnancement) selon le critère de validité des opérandes et disponibilité des unités fonctionnelles. Les instructions qui sont prêtes a l'exécution seront immédiatement affectées aux buffers associés aux unités fonctionnelles correspondantes où elles seront exécutées en parallèle. Deux méthodes principales sont utilisées pour l'implémentation de la fenêtre d'issue.

- Stations de réservation : La fenêtre d'issue est partagée en plusieurs partitions appelées stations, dont chaqu'une contient uniquement les instructions qui sont associées a une unité fonctionnelle spécifique, ce qui simplifie énormément l'ordonnancement de cette dernière.

- Fenêtre d'issue centrale : Une fenêtre d'issue centrale garde trace de tous les instructions en attente d'être initiées pour l'exécution, indépendamment de leur types. Elle doit déterminer l'instruction qui est prête à l'exécution et l'unité fonctionnelle libre sur laquelle elle doit s'exécuter. La logique d'ordonnancement pour une telle stratégie est plus compliquée.

Les résultats en sortie des unités fonctionnelles repasseront à la logique d'issue (pour une mise à jour dynamique des informations relatives aux opérandes des instructions en cours d'exécution), en même temps qu'ils seront stockés dans un buffer spécial : reorder-buffer (implémentant une file FIFO), où ils seront réorganisées selon le modèle séquentiel des instructions, avant la phase finale de mise à jour. L'utilisation de la fenêtre d'issue permet plus de concurrence entre les instructions et par conséquent, engendre plus de performance que l'exécution ordonnée. Des études sur l'évaluation des performances des superscalaires montrent qu'il y'a une amélioration approximative de 40% lors du passage d'un superscalaire à issue unique à un superscalaire à quatre issues avec exécution ordonnée, et une amélioration encore de 40% lors du passage d'un superscalaire à quatre issues avec exécution ordonnée à un superscalaire à quatre issues avec exécution désordonnée [80]. Des exemples tels que AMD K5 [81], MIPSR10K [82] et MIPS 74K [83], sont des superscalaires à exécution désordonnée.

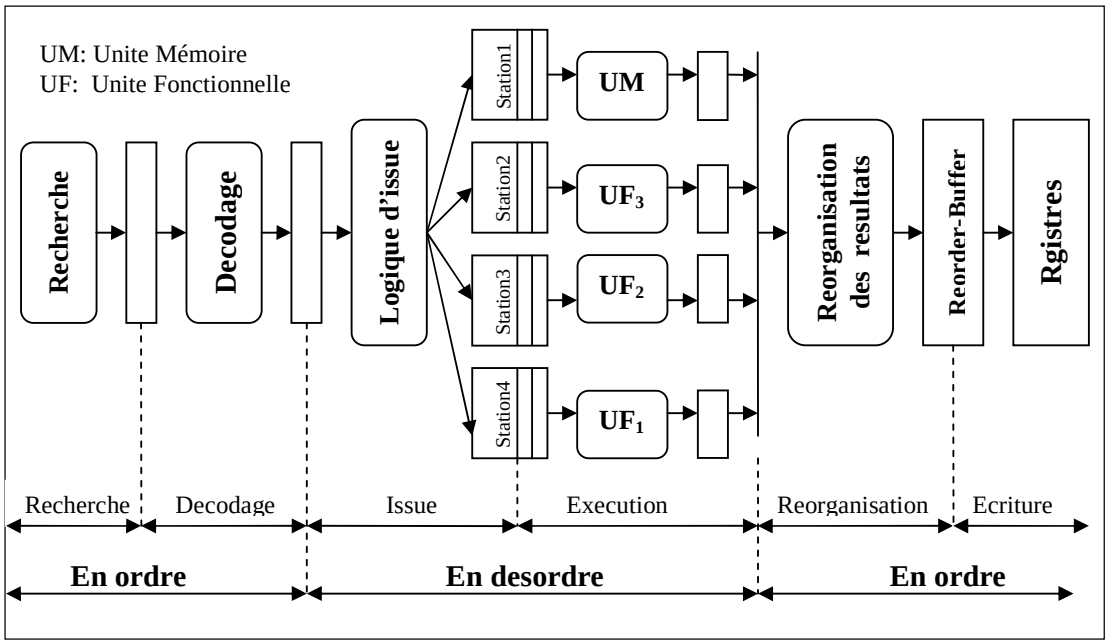


Figure 6.3 Microarchitecture d'un superscalaire à exécution désordonnée

6.3 Eléments de performance des machines superscalaires

L'exécution parallèle des instructions nécessite: la détermination des relations de dépendance entre les instructions, les ressources matériels adéquats pour exécuter multiple opérations en parallèle, des stratégies pour déterminer quant une opération est prête pour l'exécution, et des techniques pour passer des valeurs d'une opération a une autre. Des mécanismes pour réordonner les résultats des instructions dans l'ordre du programme doivent être aussi prévus, une fois l'exécution des instructions est terminée. Plus précisément, en termes de matériel, un superscalaire doit implémenter les éléments de performance suivants:

- Stratégies de recherches d'instructions: Pour la recherche simultanée de plusieurs instructions (des mémoires caches avec un algorithme LRU sont utilisées par la plupart des processeurs superscalaires). Ces stratégies prédictent aussi des instructions de branchements conditionnels.
- Méthodes pour la détermination des dépendances impliquant des valeurs de registres et des mécanismes pour communiquer ces valeurs là où elles sont demandées durant l'exécution.
- Méthodes pour l'initiation des instructions multiples en parallèle.

- Des ressources pour l'exécution parallèle de plusieurs instructions, incluant des unités fonctionnelles pipelines, et des hiérarchies de mémoires capable de servir multiples références mémoires.
- Méthodes pour remettre l'état final dans l'ordre du programme. Ces mécanismes maintiennent l'apparence d'une exécution séquentielle.

Des détails sur ces éléments de performance sont traités dans [84]

6.4 Modélisation des machines MA superscalaires

Les architectures superscalaires étendent les architectures pipeline par la réplication des étages pipeline de telle manière à lancer plusieurs instructions par cycle. Dans ce travail, nous limitons l'étude uniquement aux architectures à exécution ordonnée où les instructions sont lancées dans l'ordre du programme et les résultats sont écrits aussi dans le même ordre (l'application de notre méthodologie aux architectures à exécution désordonnée est encore sous test). De cette manière, un modèle SSMA sera construit sur un modèle PMA comme suit:

Soient n et S, le nombre de pipelines, et le nombre d'étages de chaque pipeline .respectivement. Soient $W=((W_1^1,\ldots,W_1^s),\ldots,(W_n^1,\ldots,W_n^s))$, l'état SSMA distribué sur n pipelines, et PMAi pour $1 \leq i \leq n$, la fonction qui réalise la tache du pipeline i. Donc, étant donné un état initial:$((c_0^{11},\ldots,c_0^{s1}),\ldots,(c_0^{1n},\ldots,c_0^{sn}))$, l'état d'une machine SSMA au cycle $k \geq S$, résulte de la combinaison des états des différents pipelines calculés comme suit:

$$
\begin{aligned}
&\text{SSMA}(n, k, (c_0^{11},\ldots,c_0^{s1}),\ldots,(c_0^{1n},\ldots,c_0^{sn})) = \\
&\quad \text{let} \quad (c_k^{11},\ldots,c_k^{s1}) = \text{PMA}_1(k, c_0^{11},\ldots,c_0^{s1}) \\
&\qquad\qquad \vdots \\
&\qquad (c_k^{1n},\ldots,c_k^{sn}) = \text{PMA}_n(k, c_0^{1n},\ldots,c_0^{sn}) \\
&\quad \text{in} \quad ((c_k^{11},\ldots,c_k^{s1}), \ldots,(c_k^{1n},\ldots,c_k^{sn}))
\end{aligned}
$$

Figure 6.4. Spécification fonctionnelle du modèle SSMA pour k≥S

En pratique, les phases de recherche et de décodage sont communes à toutes les instructions, alors que le reste des phases sont spécifiques à chaque instruction. Une telle configuration peut être implémentée comme suit :

Soient Icache, Pc, i et k', La mémoire cache des instructions, l'adresse début du bloc à lire, le nombre d'instructions à lire et le nombre des étages restant après la phase de

décodage respectivement. Alors la définition d'un tel modèle est illustrée par la figure 6.5 donnée ci-après.

SSMA(n, k, (c_0^{11},...,c_0^{s1}),…,(c_0^{1n},...,c_0^{sn})) =

Let (i_1,…i_n) = fetch Icache Pc, i

in let (di_1,…di_n) = decode(i_1,…i_n)

in let (c_k^{11},...,c_k^{s1}) = PMA'$_1$(k', di_1 c_0^{11},...,c_0^{s1})

⋮

(c_k^{1n},...,c_k^{sn}) = PMA'$_n$(k', di_n c_0^{1n},...,c_0^{sn})

in ((c_k^{11},...,c_k^{s1}), …,(c_k^{1n},...,c_k^{sn}))

Figure 6.5 Aspect pratique du Modèle SSMA

On remarque bien que les deux aspects ; compositionnel et parallèle sont parfaitement capturés par le style fonctionnel dans le modèle superscalaire.

6.5 Vérification des machines MA superscalaires

6.5.1 Diagramme de synchronisation

Le diagramme de synchronisation d'un modèle SSMA généralise le diagramme de synchronisation du modèle PMA. La figure 6.6 montre une telle synchronisation pour une machine superscalaire utilisant un nombre arbitraire de pipelines

Soit c_k^{ji} = f_j(PMA$_i$((k-1), c_0^{1i},...,c_0^{si}), l'état d'un composant SSMA produit par la fonction d'étage f_j, du pipeline i, au cycle k ≥S.:

Dans le cas d'absence de hasards, la fonction de synchronisation es définie comme suit:

$t_n(k,j,i) = (n*(k-j) +(i-1))*s +j$ (**t3**), où n, est le nombre de pipelines

L'état séquentiel correspondant est calcule comme suit:

$s^{j}_{t(k,j,i)}$ = (f_j **o** … **o** (f_1,…,f_1^s)) (SMA((n*(k-j) +(i-1)), c_0^{1i},...,c_0^{si}))

Dans le cas de présence de hasards, la fonction de synchronisation est définie comme suit:

$t_s(k,j,i,e) =(n*(k-j)+(i-1))-e)*s +j$ (**t4**), où e, est the nombre de stalls

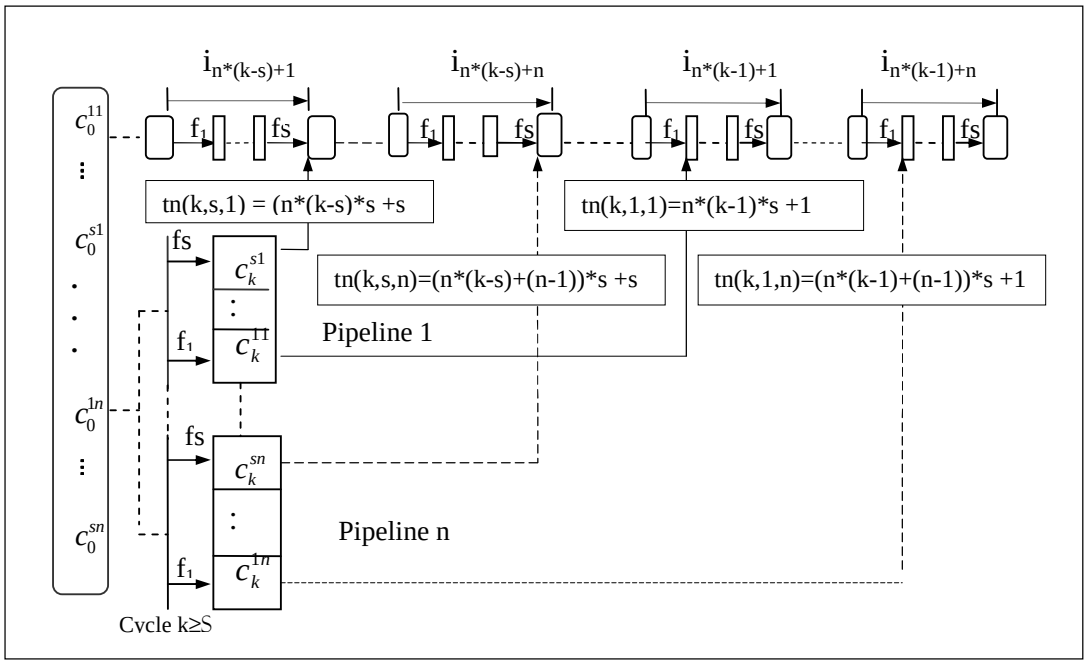


Figure 6.6. Synchronisation entre les modèles superscalaire et séquentiel.

6.5.2 Modèle CSMA pour des architectures superscalaires

Le modèle séquentiel des composants pour une architecture superscalaire appelé CSSMA sera construit sur un modèle CSMA utilisé pour une architecture pipeline. Il accepte les mêmes paramètres que le modèle SSMA, et retourne comme résultat, l'état attendu contre lequel le modèle SSMA sera comparé. La figure 6.7 montre un modèle effectuant un calcule régulier (pour k≥ S).

CSSMA (n, k, (c_0^{11},..., c_0^{s1}),…,(c_0^{1n},..., c_0^{sn})) =
let (c_k^{11},..., c_k^{s1}) = $CSMA_1$(k, c_0^{11},..., c_0^{s1})
⋮
(c_k^{1n},..., c_k^{sn}) = $CSMA_n$(k, c_0^{1n},..., c_0^{sn})
in ((c_k^{11},..., c_k^{s1}), …,(c_k^{1n},..., c_k^{sn}))

Figure 6.7.Specification fonctionnelle d'un modèle CSMA
Pour une architecture superscalaire (k ≥ S)

6.5.3 Critère de correction

La preuve de l'équivalence fonctionnelle du modèle SSMA vis-à-vis du modèle CSSMA, exige la preuve de l'équation suivante:

$$\forall \quad n, k :: \text{Int}, \quad \forall \; c_0^{11} :: W_1{}^1, \ldots, c_0^{s1} :: W_1{}^s, \qquad c_0^{1n} :: W_n{}^1, \ldots, \; c_0^{sn} :: W_n{}^s$$

$$\text{SSMA}(n, k, (c_0^{11}, \ldots, c_0^{s1}), \ldots, (c_0^{1n}, \ldots, c_0^{sn})) = \text{CSSMA}(n, k, (c_0^{11}, \ldots, c_0^{s1}), \ldots, (c_0^{1n}, \ldots, c_0^{sn}))$$

La preuve d'une telle équation se décompose systématiquement en sous preuves suivantes:

$$\text{PMA}_1(k, \; c_0^{11}, \ldots, c_0^{s1}) = \text{CSMA}_1(k, \; c_0^{11}, \ldots, c_0^{s1})$$

$$\vdots$$

$$\wedge \; \text{PMA}_n(k, \; c_0^{1n}, \ldots, c_0^{sn}) = \text{CSMA}_n(k, \; c_0^{1n}, \ldots, c_0^{sn})$$

Maintenant, nous pouvons utiliser les définitions des modèles PMA et CSMA définies dans le chapitre 5, pour résoudre ces équations.

6.5.4 Discussion

- Le système d'équations établies pour les processeurs superscalaires, généralise le système d'équations établies pour les processeurs pipelines. Par conséquent, elles peuvent être instanciées facilement pour des types particulières d'architectures superscalaires. Cela signifie que la méthodologie proposée escalade correctement avec l'évolution de la technologie [85].
- Puisque ces équations généralisent les équations établies pour les processeurs pipelines, elles peuvent être facilement décomposées et prouvées séparément par induction sur les cycles d'horloge au sein du même langage de spécification.
- Bien que le nombre d'équations à prouver dépend du nombre d'étages (après décomposition d'un pipeline en étages), nous pouvons aussi limiter la preuve uniquement aux composants observables auxquels on s'intéresse. Ceci est très utile pour les systèmes complexes impliquant plusieurs observations.

6.6 Conclusion

En décomposant l'état global d'une part, et en développant des modèles superscalaires hiérarchiques (modèles SSMA et CSMA) d'autre part, la preuve globale d'un superscalaire de n pipelines, se décompose systématiquement en un ensemble de n

conditions de vérifications, chaqu' une se rapportant a un pipeline. Par conséquent, ces conditions de vérifications peuvent être facilement prouvées séparément et par induction, en utilisant les modèles PMA et CSMA relatifs aux architectures pipelines..

Tout au long de ce chapitre nous avions considéré uniquement des modèles de microarchitectures avec d'éventuels hasards mais sans exceptions. Car ces derniers évènements nécessitent une étude particulière pour raisonner sur le non déterminisme.

Conclusions et Perspectives

A travers cette thèse, nous avons développé une méthodologie pour la modélisation et la vérification des microarchitectures RISC dans un environnement fonctionnel. En décomposant l'espace d'état (qu'est une opération naturelle dans un système distribué), la méthodologie décompose systématiquement la preuve globale en un ensemble de conditions de vérification plus facile à prouver et réutilisables pour différentes implémentations. Pour une meilleure scalabilité, la méthodologie utilise une approche incrémentale pour modéliser et vérifier des microarchitectures de complexité croissante : séquentielle, pipeline et superscalaire.

Pour les microarchitectures séquentielles (qui constituent le noyau des systèmes complexes), la méthodologie raffine la preuve jusqu'au niveau composant observable. Un tel raffinement permet au processus de débogage de continuer l'exploration à partir du niveau où le processus de preuve échoue.

Pour des architectures pipelines et superscalaires, la méthodologie offre énormément d'avantages par rapport aux approches alternatives, en particulier :

- La possibilité de raisonner sur la dépendance inter instructions tels que les différents types de hasards qui peuvent se produire pendant l'exécution, contrairement à la technique de vidage, où un tel raisonnement est impossible. En outre, il est possible de raisonner soit sur des instructions individuelles soit sur des groupes d'instructions telles que des instruction-registres, des instructions-memoires et des instructions de branchement
- La possibilité d'effectuer la preuve par induction dans le même langage de spécification, plutôt que par évaluation symbolique à travers un outil de preuve qui exige encore des efforts considérables.

Enfin, l'adoption d'un environnement fonctionnel permet de développer des spécifications exécutables qui peuvent être utilisées à coté de la vérification formelle, directement pour la simulation fonctionnelle.

Les points cités ci-dessus ne constituent qu'une modeste contribution dans un domaine très vaste. Par suite, les aspects qui nous paraissent utiles d'explorer à travers ce travail dans le future sont résumés ci-après.

1 Modélisation et vérification des processeurs pipeline en présence des exceptions

La présence des exceptions signifie des propriétés de non déterminisme, alors que les fonctions de transition nécessaires à la mise à jour des différents états du système doivent être déterministiques. Cette idée incite à réfléchir sur la façon de modéliser les fonctions de transitions de telle manière à obtenir un état cohérent des composants observables. Aussi, un cas supplémentaire doit être prévu par la fonction de synchronisation pour manipuler les exceptions.

2 Modélisation et vérification des superscalaires à exécution ordonnée avec des exceptions et Hasards

Le raisonnement sur des exceptions et des hasards qui se provoquent dans différents pipelines rend le processus de preuve de ces systèmes plus compliqué. Cette idée incite à réfléchir sur la manière d'étendre le langage Haskell avec des mécanismes suffisamment puissants pour capturer ces événements.

3 Modélisation et vérification de superscalaires avec exécution désordonnée.

Incrementallement, on pourra passer aux superscalaires à exécution désordonnée qui demandent une étude exhaustive sur la partie synchronisation, en particulier en présence d'exceptions et de hasards qui peuvent se provoquer dans différents pipelines. L'extension par une exécution spéculative rend encore le processus de preuve plus compliqué, parce qu'il faut sauvegarder toutes les informations nécessaires à la spéculation (pour les deux cas : branchement pris et branchement non pris), sur lesquelles il doit raisonner.

4 Mécanisation de la preuve

L'environnement Haskell est un environnement purement fonctionnel et par conséquent, il reste difficilement utilisable pour raisonner mécaniquement sur des preuves formelles : Une fonction quant elle est définie peut être utilisée dans les deux sens, en conséquence, le système peut boucler et la terminaison n'est pas

garantie. Pour pouvoir garantir la terminaison il faut utiliser un environnement de réécriture où la notion de fonction est remplacée par la notion de règle de réécriture qui est définie d'une manière unidirectionnelle. De cette manière, un système peut terminer si les propriétés préétablies sont satisfaites. Cette idée suggère utiliser un environnement fonctionnel basé sur les règles de réécriture, en particulier la réécriture de graphes, (parceque physiquement, un circuit a une structure de graphe et non de terme) telque le langage Clean, pour développer un outil logiciel supportant le raisonnement formel mécanique.

5 Développement d'un HDL pour la réécriture de graphes.

L idée d'utiliser un environnement de réécriture adéquat pour réaliser le processus de preuve suggère le développement d'un HDL basé sur la représentation des graphes fonctionnels (FGRS-Based HDL) qui sera utilisé non seulement pour les phase de description et de vérification mais aussi pour la phase de synthèse qui exige une représentation en termes de graphes qui permettent en outre de résoudre facilement les problèmes de cyclicité et de partage. Par suite, le développement d'un HDL combinant les caractéristiques potentielles des graphes et des règles de réécriture est d'un intérêt capital pour satisfaire les différentes activités de conception.

Bibliographie

[1] M. Zhu, J. Bian, W. Wu, "A novel collaborative scheme of simulation and model checking for system properties verification". Computers in Industry 57, 752–757, Elsevier, July 2006

[2] M.N. Velev, "Integrating Formal Verification into an Advanced Computer Architecture Course", *ASEE Annual Conference& Exposition*, June 2003

[3] Intel Corp. Pentium iii processor specication update. ftp://download.intel.com/

[4] Motorola Inc. Mpc7400 part number specication. http://e-www.motorola.com/

[5] Amd athlon processor model 1 and model 2 revision guide. http://www.amd.com/

[6] M. N. Velev, R. E. Bryant, "Automatic formal verification of pipelined microprocessors": A Tutorial: http://www.ece.cmu.edu//~mvelev.

[7] J. Burch and D. Dill, "Automatic verification of pipelined microprocessors control" CAV'94, volume 818 of *LNCS* 818, 68–80, Springer, 1994.

[8] W.Damm and A. Pnueli, Verifying out-of-order executions, *CHARME'97*, 23–47, 1997.

[9] K. McMillan, "Verification of an implementation of Tomasulo's algorithm by compositional model checking". *CAV'98*,, *LNCS* 1427, Springer, 1998.

[10] J. Sawada and W. A. Hunt. "Verification of the FM9801 microprocessor: An out- of-order microprocessor model with speculative execution, exceptions, and self-modifying code". Formal Methods in Systems Design, 20(2):187–222, March 2002.

[11] N.Ayewah, N.Kikkeri, P.M.Seidel, S.Beyer, "Challenges in the Formal Verification of Complete State-of-the-Art Processors" http://www-wjp.cs.unisb.de/projects/verification

[12] A. C .J.Fox. "An algebraic framework for modeling and verifying microprocessors using HOL". Tech. Report 512, Univ. of Combridge, computer laboratory, Apr 2001

[13] X.Shen and Arvind. "Using Term Rewriting Systems to design and verify processors", IEEE Micro Special issue on Modeling and Validation of Microprocessors, May/June 99

[14] D.Schostak. " Methodology for the formal specification of RTL RISC processor Design*s"*, PhD thesis, the University of Leeds, 2003.

[15] R.M. Hosabettu, M. Srivas, and G. Gopalakrishnan, 'Decomposing the Proof of Correctness of Pipelined Microprocessors', (CAV'98), A.J. Hu and M.Y. Vardi, *eds*., LNCS 1427, Springer-Verlag, June1998.

[16] M.Velev and R.Bryant. 'Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction', DAC '00, June 2000

[17] M.N. Velev, "Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors,"Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01), LNCS 2031, 252–267, Springer-Verlag, April 2001

[18] M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," Journal of Symbolic Computation (JSC), Vol. 35, No. 2, 2003.

[19] M.N.Velev, 'Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer'. DATE'02, March 2002

[20] S. K. Lahiri, S. A. Seshia, and R. E. Bryant, 'Modeling and verification of out-of-order microprocessors in UCLID, FMCAD'02, LNCS 2517, Nov 2002.

[21] A.C.J. Fox. 'An algebraic framework for verifying the correctness of hardware with input and output: A formalization in HOL', CALCO 2005, LNCS, 3629, 2005.

[22] A.C.J Fox. 'Verifying the ARM Block Data Transfer Instructions', DCC 2004, Barcelona, 2004.

[23] P. Manolios, 'Well Founded Equivalence Bissimulation', Technical report, Department of Computer Science, University of Texas at Austin, 2000.

[24] P. Manolios, 'Correctness of pipelined machines'. In W.A. Hunt, and S. D. Johnson editors, FMCAD 2000, LNCS. Springer-Verlag, 2000.

[25] Tahar and R. Kumar, 'A Practical Methodology for the Formal Verification of RISC Processors', Formal Methods in Systems Design, 13(2), September 1998.

[26] K.C. Claessen, An Embedded Language Approach to Hardware Description and Verification', PhD thesis, Chalmers University of technology. 2001.

[27] J. R. Matthews: 'Algebraic specification and verification of processor Micro-architectures' PhD thesis, Oregon Graduate Institute of science and Technology, 2000.

[28] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," IEEE Transactions on Computers, Vol. 35, No. 8, pp. 677-691, August 1986.

[29] E.M. Clarke, E.A.Emerson, et A. P.Sistla, "Automatic Verification of Finite State Concurrent System Using Temporal Logic", ACM Transactions on Programming Languages and Systems, Vol. 8920, 1986.

[30] R.E.Bryant, "Symbolic Boolean manipulation with ordered binary decision digrams" ACM Computing Surveys. 24(3), 393-318, Sept 92.

[31] S. Owre, J. M. Rushby, and N. Shankar. "PVS: A prototype verification system". 11th International Conference on Automated Deduction (CADE), volume 607, Lecture Notes in Artificial Intelligence, June 1992. Springer-Verlag.

[32] S. Beyer. "Putting it all together – Formal Verification of the VAMP". PhD thesis, Saarland University, Germany, 2005.

[33] M. K. Srivas and S. P. Miller, "Applying Formal Verification to the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods," Formal Methods in System Design, 8(2), pp. 153-188, March 1996.

[34] M. Kaufmann, P.Manolios, and J S.Moore,. "Computer Aided Reasoning: ACL2 Case Studies". Kluwer Academic Press, 2000

[35] J. Sawada, "Verification of a Simple Pipelined Machine Model," in *Computer-Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, and J.S. Kluwer Academic Publishers, London, 2000, pp. 137–150

[36] D. M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. In *FMCAD, LNCS*. 1954, Springer, 2000.

[37] Nancy A. Day, Je_rey R. Lewis, and Byron Cook. "Symbolic simulation of microprocessor models using type classes in Haskell". Technical Report CSE-99-005, Department of Computer Science, Oregon Graduate Institute, June 1999.

[38] M. Velev, R. E. Bryant, and A. Jain, "Efficient Modeling of Memory Arrays in Symbolic Simulation", in Computer-Aided Verification, CAV-97, June 1997.

[39] P. Mishra, N.Dutt, N.Krishnamurthy, and M.Abadir: "A Methodology for validation of Microprocessors using symbolic simulation", International Journal of Embedded Systems. Vol 1, Number 1 / 2, 2005.

[40] A. Jain, "Formal Hardware Verification By Symbolic Trajectory Evaluation", Phd Thesis, Electrical and computer engineering, Pittsburgh, Pennsylvania, July, 1997

[41] S. Hazelhurst and C. J. H. Seger, "Composing Symbolic Trajectory Evaluation Results," Lecture Notes in Computer Science, Computer Aided Verification, 6th International Conference, CAV 94, pp. 273-285, 1994.

[42] M. Pandey, R. Raimi, D. L. Beatty, and R. E. Bryant, "Formal Verification of PowerPC Arrays Using Symbolic Trajectory Evaluation," 33rd Design Automation Conference, pp.649- 654, June 1996.

[43] Avra J. Cohn. "A proof of correctness of the VIPER microprocessor: The first level". Graham Birtwistle and P.A. Subrahmanyam, editors, VLSI Specification, Verification and Synthesis, 27–71. Kluwer Academic Publishers, 1987.

[44] W. Hunt, "FM8501: A Verified Microprocessor". University ofTexas, Austin, Tech. Report 47, 1985.

[45] J. J. Joyce. "Formal specification and verification of microprocessor systems". Microprocessing & Microprogramming, 24(1-5):371–8, 1988.

[46] P. J. Windley. "Formal Modeling and verification of microprocessors", IEEE Transactions on Computers, 44(1):54–72, 1995.

[47] J. L. Hennessy and D.A. Patterson, "Computer Architecture:A Quantitative Approach". 3rd Edition, Morgan Kaufmann Publishers Inc, San Francisco CA, 2003

[48] V.Bhagwati. "Automatic Verification of Pipelined Microprocessors". S.M.Thesis, MIT,93

[49] M.N. Velev, and R.E. Bryant, “TLSim and EVC: A Term-Level Symbolic Simulator and an Efficient Decision Procedure for the Logic of Equality with Uninterpreted Functions and Memories,” International Journal of Embedded Systems (IJES), 2004.

[50] Miroslav N. Velev, “Collection Of High-Level Microprocessor Bugs From Formal Verification Of Pipelined And Superscalar Designs”, Itc International Test Conference, 2003

[51] E. Borger, U. Glasser, and W. Muller. “Formal definition of an abstract VHDL '93 simulator by EA-machines”. In C. Delgado Kloos and P.T. Breuer, editors, Formal Semantics for VHDL, 107-139, Kluwer Academic Press Boston/London/Dordrecht, 1995

[52] K.G.W. Goossens, “Reasoning about VHDL using operational and observational semantics”. Technical Report SI/RR-95/06, Dipartimento di Scienze dell' Informazione, Universita degli Studi di Roma,`La Sapienza', via Salaria, 113-I-00198, Roma, April 1995

[53] Carlo Delgado Kloos and Peter T.Breuer. “Formal Semantics for VHDL”. Number 307 in The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.

[54] IEEE.10766-2004 “IEEE standard for VHDL Registre transfer Level (RTL) synthesis”, 2004 Donald E. Thomas. The Verilog HDL. Kluwer Academic, Boston, Massachusetts, second edition, 1995.

[55] Donald E. Thomas. The Verilog HDL. Kluwer Academic, Boston, Massachusetts, second edition, 1995.

[56] U. Golze. VLSI Chip Design with the Hardware Description Language VERILOG: an introduction based on a large RISC processor. Springer, 1996

[57] G. J. Pace and Jifeng He. Formal reasoning with Verilog HDL. In Proceedings of the Workshop on Formal Techniques in Hardware and Hardware-like Systems, Marstrand, Sweden, June 1998

[58] IEEE standard Verilog, Hardware Description Language, NY. IEEE Std 1364-2001

[59] Peter T. Breuer, Luis Sanchez Fernandez, and Carlos Delgado Kloos. A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for Unit- Delay VHDL. In Formal Methods in System Design, volume 7, Number 1/2, 27-51, August 1995

[60] S. Read and M. Edwards. A Formal Semantics of VHDL in Boyer-Moore Logic. Conference on Concurrent Engineering and EDA (CEEDA), Poole, Great Britain,1994

[61] C.Bayol, B.Soulas, F.Corno, P.Prinetto, and D.Borrione. A process algebra interpretation of a verification oriented over language of VHDL. In EURO-DAC 1994/EURO-VHDL '94, IEEE CS Press, 1994

[62] P. Georgelin, Vérification formelle de systèmes digitaux synchrones basés sur la simulation symbolique. Thèse de doctorat, Université Joseph Fourier- Grenoble, 2001.

[63] M. Gordon. The semantic challenge of Verilog HDL. In the proceedings of the tenth annual IEEE symposium on Logic in Computer Science (LICS '95), San Diego, California,136-145, June 1995

[64] G. J. Pace, "Hardware Design Based on Verilog HDL", Phd thesis, Oxford University Computing Laboratory, 1998

[65] John O_Donnell, "From transistors to computer architecture:Teaching functional circuit specification in Hydra". In Symposium on Functional Programming Languages in Education, July 1996

[66] S L. Peyton Jones et al, "Haskell 98: A non strict,, purely functional language" Revised; February 1999. Available at:: http://www.haskell.org/onlinereport

[67] P. Bjesse., K.Claessen., M.Sheeran, And S.Singh, "Lava: Hardware design in Haskell". In International Conference on finctional Programming, Baltimore, July 1998

[68] Byron Cook, John Launchbury, and John Matthews, "Specifying superscalar microprocessors in Hawk", In Formal Techniques for Hardware and Hardware_like Systems, Marstrand, Sweden, 98.

[69] Ghiat El Sammane, Simulation symbolique des circuits décrits au niveau algorithmique, Thèse de doctorat, Université Joseph Fourier, Grenoble, 2005

[70] F. Orejas, J. Navarro, A Sanchez, 'Implementation and Behavioural Equivalence: A survey', Proc. 8th Workshop on Algebraic Specification. Dourdan, 1991.

[71] G. Randin, "The 801 Minicomputer", Proc. Symp. Architectural Support for programming Languages and operating systems, March, 1982

[72] R.Plasmeijer and M.V.Eekelen, "Functional programming and parallel graph rewriting". Addison Wesley 1993.

[73] S. Merniz, M. Benmohammed, "Formal Specification and verification of Processor Micro-Architectures: Functional approach", International Review On Computers and Software, Vol. 3, N° 3, May 2008. Praise-Worthy-Prize Publisher, Naples, Italy.

[74] J.L.Hennessy and D.A.Patterson, "Computer Architecture: A Quantitative Approach".3rd Edition, Morgan Kaufmann Publishers Inc, San Francisco CA, 2003

[75] S. Merniz, M. Benmohammed, "A Scalable proof Methodology for RISC Processor Designs" : Functional Approach, ITNG 2008, Las Vegas, USA, 2008.

[76] R.E. Bryant, S.K. Lahiri, and S.A. Seshia. 'Convergence testing in term level bounded model checking'. CHARME 2003.

[77] MIPS R5000 Microprocesseur user guide: web site: http://www. techpubs.sgi.com

[78] RM7000™ Microprocessor with On-Chip Secondary Cache DatasheetReleased, PMC- Sierra, Inc. Web Site: http://www.pmc-sierra.com

[79] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, pp. 25-33, January 1967.

[80] J.H.C. Tseng, "Banked Microarchitectures for Complexity-Effective Superscalar Microprocessors", PhD thesis, Massachusetts Institute of Technology 2006.

[81] M. Slater, ‘‘AMD’s K5 Designed to Outrun Pentium,’’ Microprocessor Report, pp. 1,6-11, Oct. 24, 1994

[82] P. Mishra N. Dutt A. Nicolau, “A Study of Out-of-Order Completion for the MIPS R10K Superscalar Processor”, Technical Report #01-06, Dept. of Information and Computer Science, University of California, Irvine, CA 92697, USA, 2006

[83] K.R. Kishore, V.Rajagopalan, G.Beloev, and R.Thekkath, “Architectural Strengths of the MIPS32 74K Core Family”, 2007: http://en.wikipedia.org/wiki/MIPS_architecture

[84] D. Kroning, “Formal Verification Of Pipelined Microprocessors”, PhD thesis, Université de Saarlandes, Saarbrucken, 2001

[85] S.Merniz, M.Benmohammed, “Formal Verification of superscalar Microarchitectures: Functional Approach”, International Journal of Circuits, Systems and signal Processing, NAUN Journals, Sept 2008.